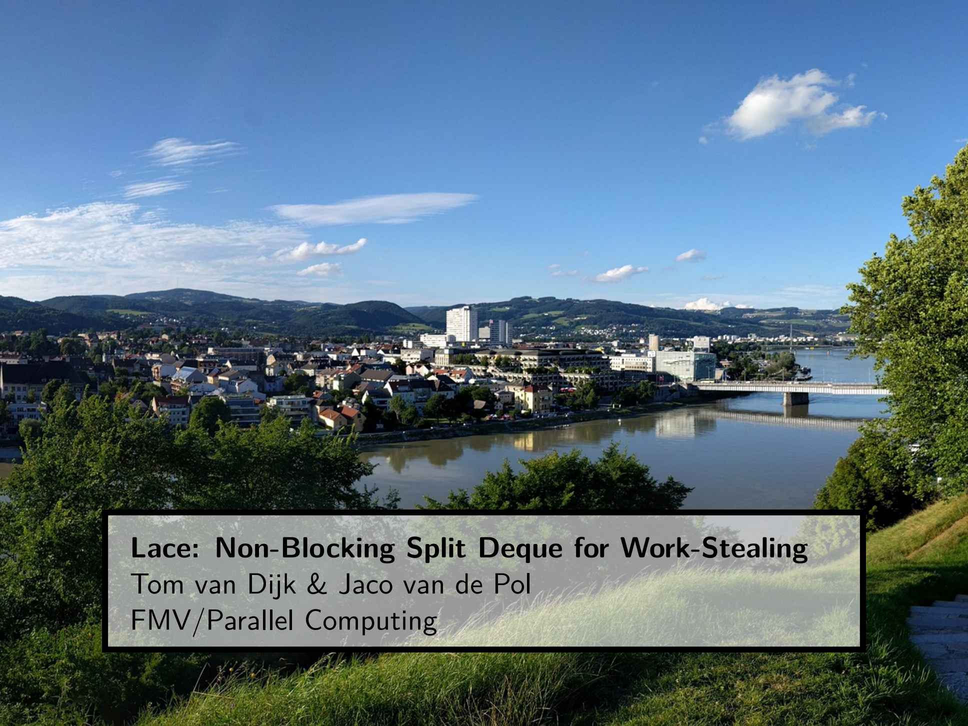


Lace: Non-Blocking Split Deque for Work-Stealing

Tom van Dijk & Jaco van de Pol

FMV/Parallel Computing

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 SAT
  - Parity games

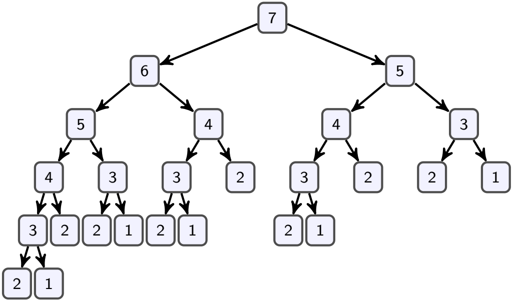
Guide

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.

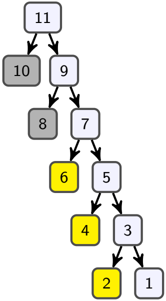
Task parallelism

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



Example: calculate `fib(11)`

Task graph:



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)

- 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)

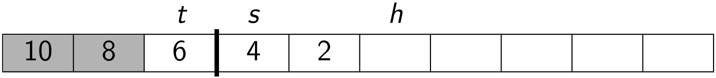
Deques for work-stealing

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

Deque in Lace

Implemented non-blocking split deque in Lace.
Deque is described by variables tail (t), split (s), head (h).



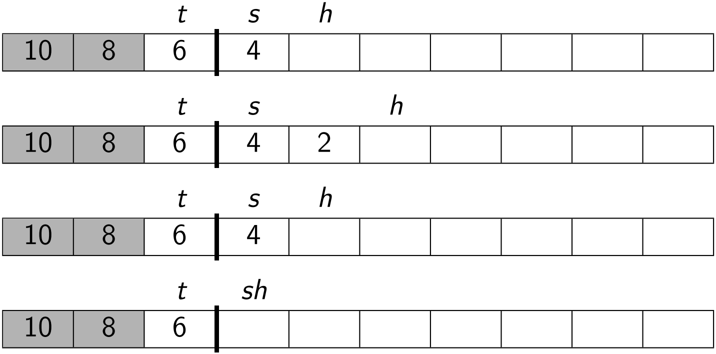
		t	s		h				
10	8	6	4	2					

- 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.



		t	s	h					
10	8	6	4						

		t	s		h				
10	8	6	4	2					

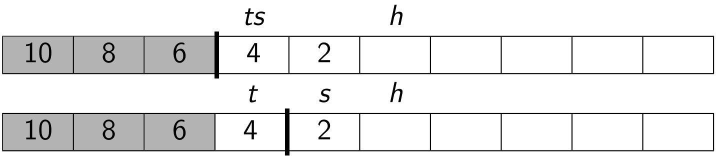
		t	s	h					
10	8	6	4						

		t	sh						
10	8	6							

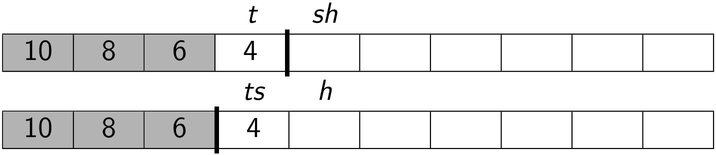
Deque in Lace

Grow and shrink

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



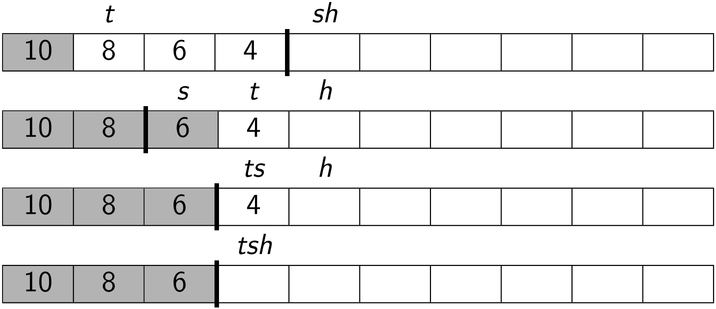
- `shrink`: set *s* between *t* and $h - 1$ (round up).



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



Private deque

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 (`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

- `fib`(50) – 20,365,011,073 tasks
- `N-queens`(15) – 171,129,071 tasks
- `uts`(T3L) – Unbalanced Tree Search, 111,345,630 tasks
- `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.

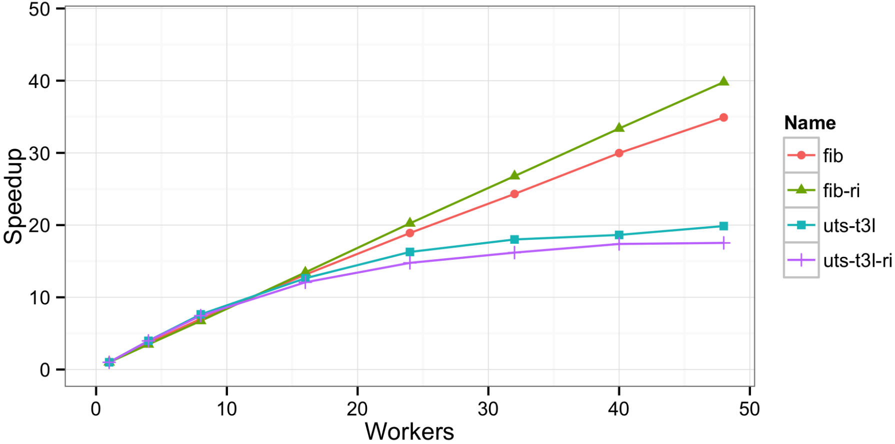
Experimental results

Benchmark	T_S	Lace		Speedup	
		T_1	T_{48}	T_S/T_{48}	T_1/T_{48}
fib 50	149.2	144	4.13	34.5	34.9
uts T3L	43.11	44.2	2.23	18.7	19.9
uts T3L *	43.11	44.3	1.154	37.4 *	38.2
queens 15	533	602	12.63	42.2	47.7
matmul 4096	773	781	16.46	47.0	47.5
		Private deque			
fib 50	149.2	208	5.22	23.2	39.8
uts T3L	43.11	44.8	2.55	17.3	17.5
uts T3L *	43.11	44.8	1.240	34.8 *	36.2
queens 15	533	541	11.34	43.3	47.7
matmul 4096	773	774	16.34	47.3	47.4

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

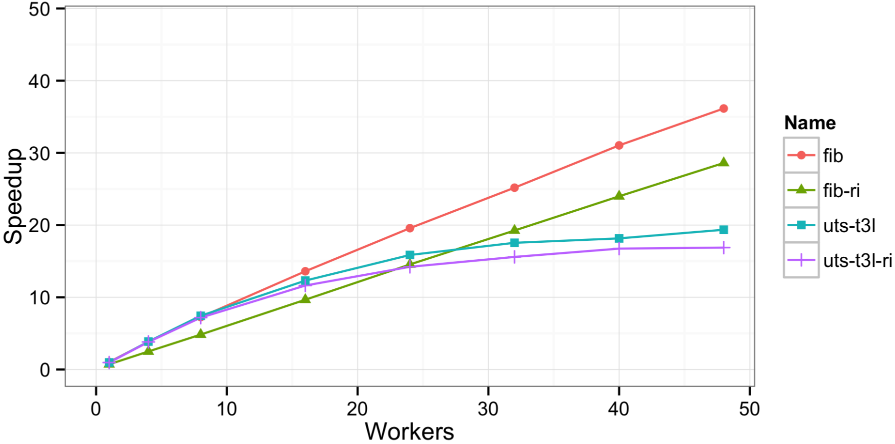
Scalability

Scalability T_1/T_n (relative to itself; does not show overhead)



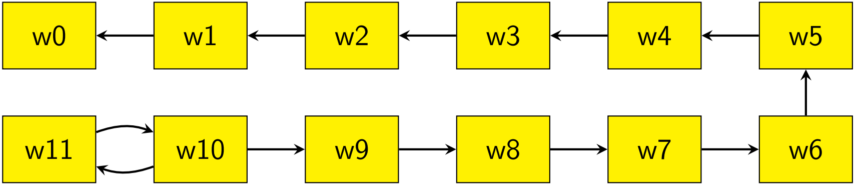
Scalability

Scalability T_S / T_n (relative to sequential; shows overhead)



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

- 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

Algorithm	T_1	T_{48}	T_S / T_{48}	T_1 / T_{48}
Old Lace	44.2	2.23	18.7	19.9
Old Private Deque	44.8	2.55	17.3	17.5
Old Wool	44.3	2.12	19.4	20.9
Lace	44.26	1.154	37.4	38.3
Private Deque	44.83	1.240	34.8	36.2
Wool	44.27	1.172	36.8	37.8

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

Conclusions

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

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

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
  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()
```

Algorithm outline

```
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
```

Algorithm outline

```
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
```