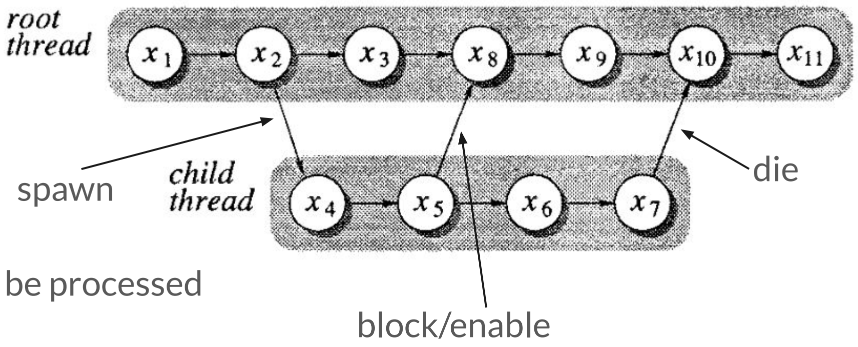
Thread Scheduling for Multiprogrammed Multiprocessors

N.S. Arora, R.D. Blumofe, C.G. Plaxton (2001)

Presented by Jason Liu

DAG-based computation model

- Each vertex takes one processor one unit of time
- Each vertex represents an instruction
- Up to two children per vertex
- Up to two parents per vertex
- All ancestors must be processed before a vertex may be processed
- Work T_1 is total number of nodes
- Span T_∞ is length of longest path



Thread Scheduling on Dedicated Processors

- P processors available
- Lower bounds for runtime are T_1/P and T_∞
- Greedy scheduling achieves runtime of $T_1/P + T_\infty$

Modifications for multi-programmed machines

- Programmer assigns vertices/tasks to processes
- Kernel chooses processes to run on processors
- Kernel is adversarial, but may have different levels of adversariality
- Expected runtime will be $O(T_1/P_A + T_\infty P/P_A)$

Difficulties for multi-programmed machines

- May not always have P processors available for use
- Processor availability varies over algorithm runtime
  - If some p_i processors are available at each step i, then average availability is $P_A = \frac{1}{T}\sum_{i=1}^{T} p_i$
- Adversarial kernel may be able to choose which p_i of the P total processes to run

	q_1	q_2	q_3
1	✓	✓	
2	✓	✓	✓
3			
4	✓		✓
5		✓	✓
6	✓	✓	✓
7			✓
8	✓	✓	
9	✓	✓	✓
10		✓	✓

(a)

	q_1	q_2	q_3
1	x_1	I	
2	x_2	I	I
3			
4	x_3		x_4
5		I	x_5
6	x_8	x_6	I
7			x_9
8	x_7	I	
9	I	I	x_{10}
10		x_{11}	I

(b)

Basic bounds for Multi-programmed parallel processing

Theorem 1. *Consider any multithreaded computation with work T_1 and critical-path length T_∞, and any number P of processes. Then for any kernel schedule, every execution schedule has length at least T_1/P_A, where P_A is the processor average over the length of the schedule. In addition, for any number P'_A of the form $T_\infty P/(k + T_\infty)$ where k is a nonnegative integer, there exists a kernel schedule such that every execution schedule has length at least $T_\infty P/P_A$, where P_A is the processor average over the length of the schedule and is in the range $\lfloor P'_A \rfloor \leq P_A \leq P'_A$.*

Theorem 2 (Greedy Schedules). *Consider any multithreaded computation with work T_1 and critical-path length T_∞, any number P of processes, and any kernel schedule. Any greedy execution schedule has length at most $T_1/P_A + T_\infty(P-1)/P_A$, where P_A is the processor average over the length of the schedule.*

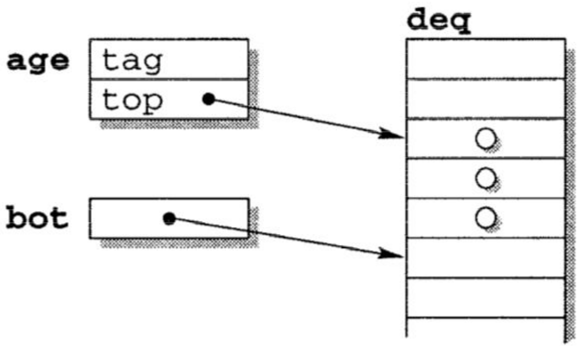
Work-Stealing

- Each process maintains a double-ended queue of ready threads/nodes and an assigned node
  - Processes with empty deques and no assigned node are thieves
- Non-thief processes push/pop from bottom of own deque
- Thieves pick random deques to pop top until they find something and are no longer thieves
- To deal with adversarial kernels, thieves will yield between steal attempts to restrict kernel behavior

```
// Assign root node to process zero.
assignedNode ← NIL
if self = processZero
    assignedNode ← rootNode

// Run scheduling loop.
while computationDone = FALSE

    // Execute assigned node.
    if assignedNode ≠ NIL
        (numChildren, child1, child2) ← execute (assignedNode)

        if numChildren = 0                        // Terminate or block.
            assignedNode ← self.popBottom()
        else if numChildren = 1                   // No synchronization.
            assignedNode ← child1
        else                                      // Enable or spawn.
            self.pushBottom (child1)
            assignedNode ← child2

    // Make steal attempt.
    else
        yield()                                   // Yield processor.
        victim ← randomProcess()                  // Pick victim.
        assignedNode ← victim.popTop()            // Attempt steal.
```

Deque semantics

- Ideal semantics: there should exist linearization times
  - Pick distinct times for each operation between their start and end times
  - Return values should be consistent with serial execution in this order
- Relaxed semantics: there should exist linearization times, except that popTop may return NIL even if queue is nonempty if a different process has just popped the top.




```
void pushBottom (Node node)
  load localBot ← bot
  store node → deq[localBot]
  localBot ← localBot + 1
  store localBot → bot
```

```
Node popTop()
  load oldAge ← age
  load localBot ← bot
  if localBot ≤ oldAge.top
      return NIL
  load node ← deq[oldAge.top]
  newAge ← oldAge
  newAge.top ← newAge.top + 1
  cas (age, oldAge, newAge)
  if oldAge = newAge
      return node
  return NIL
```

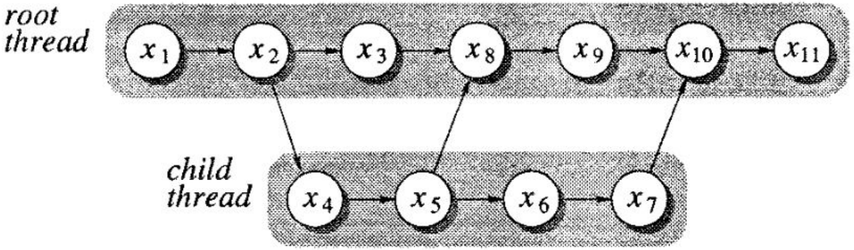
```
Node popBottom()
  load localBot ← bot
  if localBot = 0
      return NIL
  localBot ← localBot − 1
  store localBot → bot
  load node ← deq[localBot]
  load oldAge ← age
  if localBot > oldAge.top
      return node
  store 0 → bot
  newAge.top ← 0
  newAge.tag ← oldAge.tag + 1
  if localBot = oldAge.top
      cas (age, oldAge, newAge)
      if oldAge = newAge
          return node
  store newAge → age
  return NIL
```

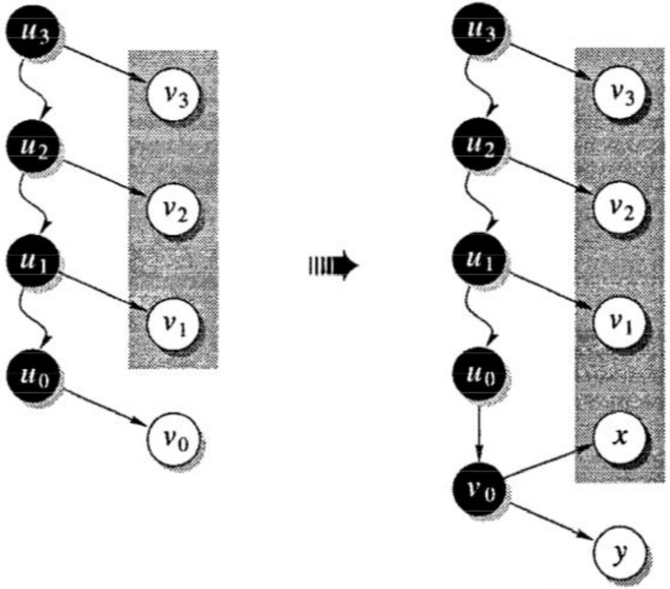
Structural Lemma

For any node in the computation, we can define the *designated parent* to be the parent which enables the child.

Lemma: For any deque, the *designated parents* of the nodes in the deque (and the assigned node) lie, in top-to-bottom order, on a root-to-leaf path in the computation. Furthermore, all the parents are distinct except potentially those of the bottommost node and the assigned node.

Proof: Induction!





(a) (b)

$$\varphi_i(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned;} \\ 3^{2w(u)} & \text{otherwise.} \end{cases}$$

Potential

$$w(u) = T_\infty - d(u)$$

$$\Phi_0 = 3^{2T_\infty - 1}$$

- Let d(u) be the depth of a node in computation
- Potential is sum of node potentials for all nodes in all deques
- Potential always decreases, is always integral, and ends at zero
- By structural lemma, most of the potential of each deque is in the top node of the deque

Runtime analysis

- Split computation into rounds of at least two iterations of the scheduling loop
- Mark scheduled processes as either *throws* or *successes* depending on whether or not the second iteration of each loop is a steal
- For S throws the runtime is $O(T_1/P_A + S/P_A)$
- Within sequences of $\Theta(P)$ throws, enough steals will succeed in expectation so that total potential decreases by a constant fraction (for valid yield/kernel pairs)
  - Enough non-empty deques are probably targeted by steals to decrease potential
  - With yields, enough empty-deque processes will have their assigned nodes executed and therefore the potential will be used
- By Chernoff bounds, at most $O((T_\infty + \lg(1/\varepsilon))P)$ throws with high probability

Theorem statements

Lemma 6 (Top-Heavy Deques). *Consider any round i and any process q in D_i. The topmost node u in q's deque contributes at least three-quarters of the potential associated with q. That is, we have $\varphi_i(u) \geq \frac{3}{4}\Phi_i(q)$.*

Lemma 7 (Balls and Weighted Bins). *Suppose that P balls are thrown independently and uniformly at random into P bins, where for $i = 1, \ldots, P$, bin i has a weight W_i. The total weight is $W = \sum_{i=1}^{P} W_i$. For each bin i, define the random variable X_i as*

$$X_i = \begin{cases} W_i & \text{if some ball lands in bin } i\,; \\ 0 & \text{otherwise.} \end{cases}$$

If $X = \sum_{i=1}^{P} X_i$, then for any β in the range $0 < \beta < 1$, we have $\Pr\{X \geq \beta W\} > 1 - 1/((1-\beta)e)$.

Types of adversarial kernels

- Benign
  - Picks random processes to schedule
  - Yield doesn't need to do anything
- Oblivious
  - Chooses processes to schedule, but offline
  - Yield requires kernel to schedule a random other process before rescheduling current process, essentially forcing the kernel to behave benignly for part of the time when processing many throws
- Adaptive
  - Chooses processes to schedule, possibly online
  - Yield forces kernel to schedule every other process before the rescheduling current process

	q_1	q_2	q_3
1	✓	✓	
2	✓	✓	✓
3			
4	✓		✓
5		✓	✓
6	✓	✓	✓
7			✓
8	✓	✓	
9	✓	✓	✓
10		✓	✓

(a)

	q_1	q_2	q_3
1	x_1	I	
2	x_2	I	I
3			
4	x_3		x_4
5		I	x_5
6	x_8	x_6	I
7			x_9
8	x_7	I	
9	I	I	x_{10}
10		x_{11}	I

(b)

Future work

- Ideal semantics for deque
- Implementation
- Less powerful yield for adaptive adversary