

Understanding Task Scheduling Algorithms

Kenjiro Taura

Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

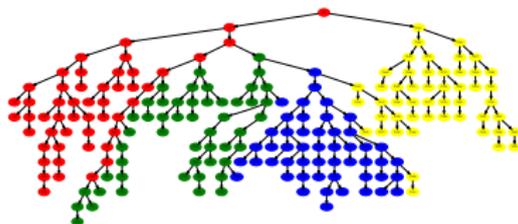
Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

Introduction

- in this part, we study
 - how tasks in task parallel programs are scheduled
 - what can we expect about its performance

```
void ms(elem * a, elem * a_end,  
        elem * t, int dest) {  
    long n = a_end - a;  
    if (n == 1) {  
        ...  
    } else {  
        ...  
        create_task(ms(a, c, t, 1 - dest));  
        ms(c, a_end, t + nh, 1 - dest);  
        wait_tasks;  
    }  
}
```



Goals

- understand a state-of-the-art scheduling algorithm (*work stealing scheduler*)

Goals

- understand a state-of-the-art scheduling algorithm (*work stealing scheduler*)
- execution time (without modeling communication):
 - how much time does a scheduler take to finish a computation?
 - in particular, *how close is it to greedy schedulers?*

Goals

- understand a state-of-the-art scheduling algorithm (*work stealing scheduler*)
- execution time (without modeling communication):
 - how much time does a scheduler take to finish a computation?
 - in particular, *how close is it to greedy schedulers?*
- data access (communication) cost:
 - when a computation is executed in parallel by a scheduler, how much data are transferred (caches \leftrightarrow memory, caches \leftrightarrow cache)?
 - in particular, how much are they worse (or better) than those of the serial execution?

Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

Model of computation

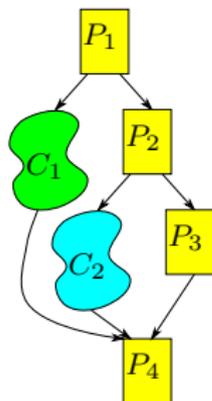
- assume a program performs the following operations
 - `create_task(S)`: create a task that performs S
 - `wait_tasks`: waits for completion of tasks it has created (but has not waited for)
- e.g.,

```
1  int fib(n) {
2      if (n < 2) return 1;
3      else {
4          int x, y;
5          create_task({ x = fib(n - 1); }); // share x
6          y = fib(n - 2);
7          wait_tasks;
8          return x + y;
9      }
10 }
```

Model of computation

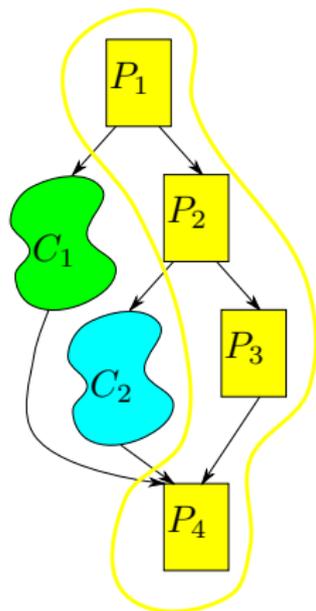
- model an execution as a DAG (directed acyclic graph)
 - node: a sequence of instructions
 - edge: dependency
- assume *no other dependencies besides induced by create_task(S) and wait_tasks*
- e.g., (note C_1 and C_2 may be subgraphs, not single nodes)

```
1 P1
2 create_task(C1);
3 P2
4 create_task(C2);
5 P3
6 wait_tasks;
7 P4
```



Terminologies and remarks

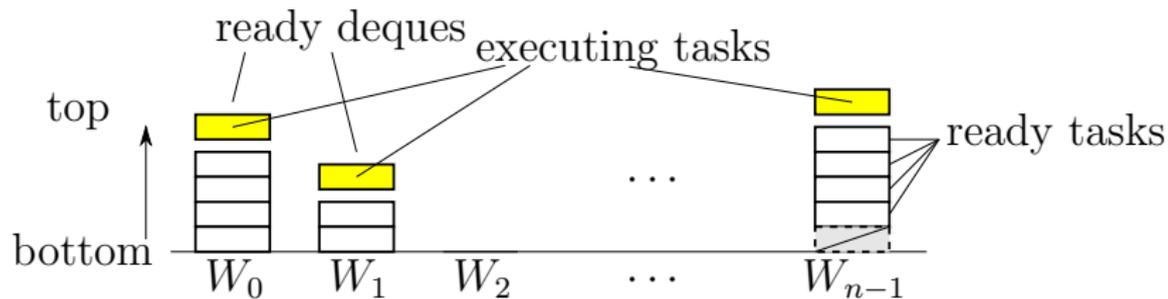
- a single node in the DAG represents a sequence of instructions performing no task-related operations
- note that a *task* \neq a single node, but = a *sequence of* nodes
- we say *a node is ready* when all its predecessors have finished; we say *a task is ready* to mean a node of it becomes ready



Work stealing scheduler

- a state of the art scheduler of task parallel systems
- the main ideas invented in 1990:
 - Mohr, Kranz, and Halstead. *Lazy task creation: a technique for increasing the granularity of parallel programs*. ACM conference on LISP and functional programming.
- originally termed “Lazy Task Creation,” but essentially the same strategy is nowadays called “work stealing”

Work stealing scheduler: data structure



- each worker maintains its *“ready deque”* that contains ready tasks
- the top entry of each ready deque is an executing task

Work stealing scheduler : in a nutshell

- **work-first**; when creating a task, the created task gets executed first (before the parent)

Work stealing scheduler : in a nutshell

- **work-first**; when creating a task, the created task gets executed first (before the parent)
- **LIFO execution order within a worker**; without work stealing, the order of execution is as if it were a serial program
 - **create_task(S)** $\equiv S$
 - **wait_tasks** \equiv noop

Work stealing scheduler : in a nutshell

- **work-first**; when creating a task, the created task gets executed first (before the parent)
- **LIFO execution order within a worker**; without work stealing, the order of execution is as if it were a serial program
 - **create_task(S)** $\equiv S$
 - **wait_tasks** \equiv noop
- **FIFO stealing**; it partitions tasks at points close to the root of the task tree

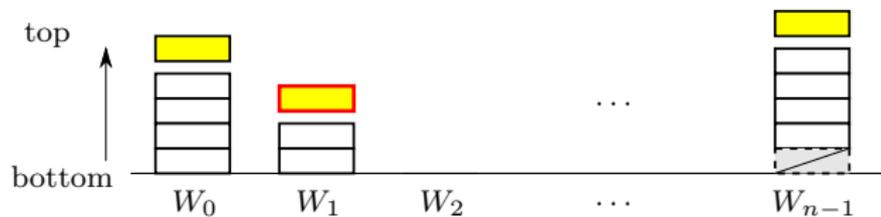
Work stealing scheduler : in a nutshell

- **work-first**; when creating a task, the created task gets executed first (before the parent)
- **LIFO execution order within a worker**; without work stealing, the order of execution is as if it were a serial program
 - **create_task(S)** $\equiv S$
 - **wait_tasks** \equiv noop
- **FIFO stealing**; it partitions tasks at points close to the root of the task tree
- it is a practical approximation of a greedy scheduler, in the sense that any ready task can be (eventually) stolen by any idle worker

Work stealing scheduler in action

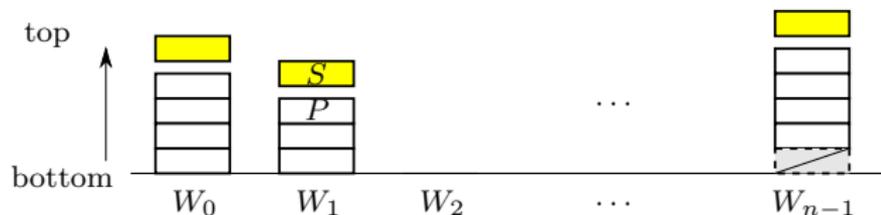
- describing a scheduler boils down to defining actions on each of the following events
 - (1) `create_task`
 - (2) a worker becoming idle
 - (3) `wait_tasks`
 - (4) a task termination
- we will see them in detail

Work stealing scheduler in action



(1) worker W encounters `create_task(S)`:

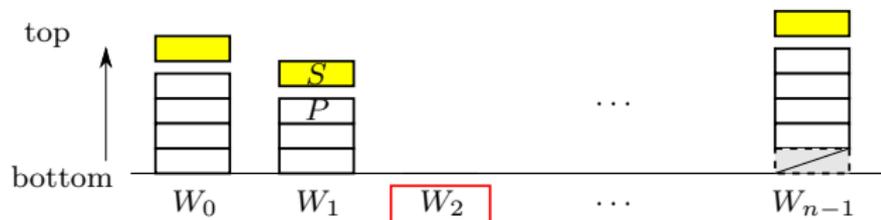
Work stealing scheduler in action



(1) worker W encounters `create_task(S)`:

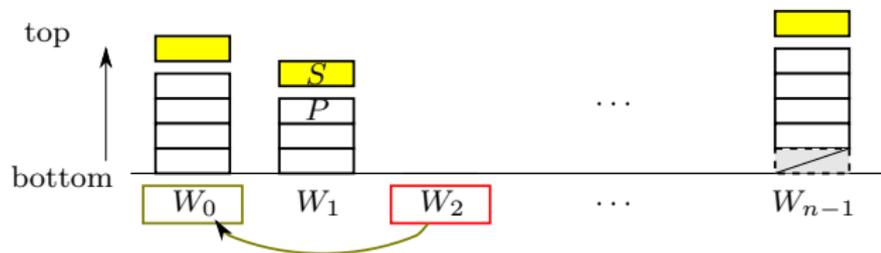
- 1 W pushes S to its deque
- 2 and *immediately starts executing S*

Work stealing scheduler in action



(2) a worker with empty deque repeats *work stealing*:

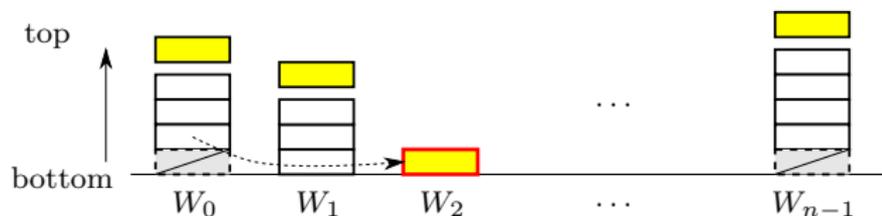
Work stealing scheduler in action



(2) a worker with empty deque repeats *work stealing*:

- 1 picks a random worker V as the victim

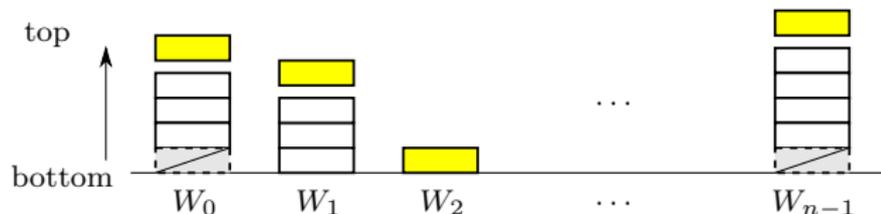
Work stealing scheduler in action



(2) a worker with empty deque repeats *work stealing*:

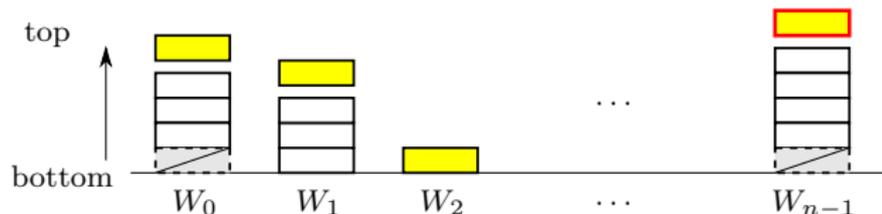
- 1 picks a random worker V as the victim
- 2 steals the task at the *bottom of V 's deque*

Work stealing scheduler in action



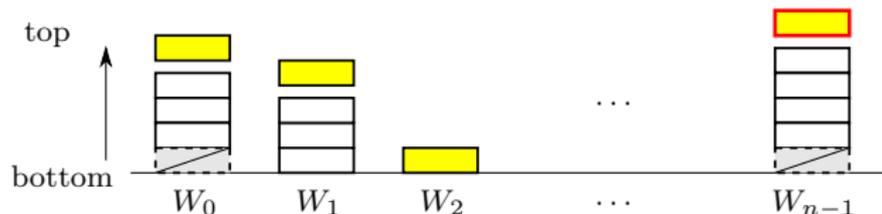
(3) a worker W encounters `wait_tasks`: there are two cases

Work stealing scheduler in action



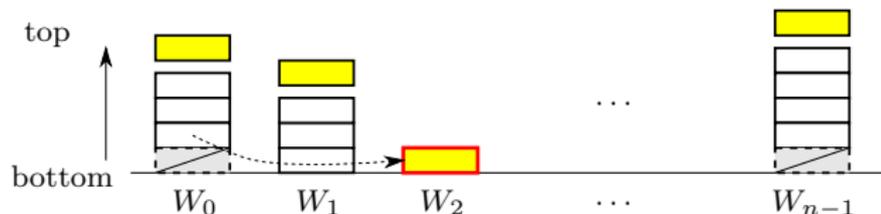
- (3) a worker W encounters `wait_tasks`: there are two cases
- 1 tasks to wait for have finished $\Rightarrow W$ just continues the task

Work stealing scheduler in action



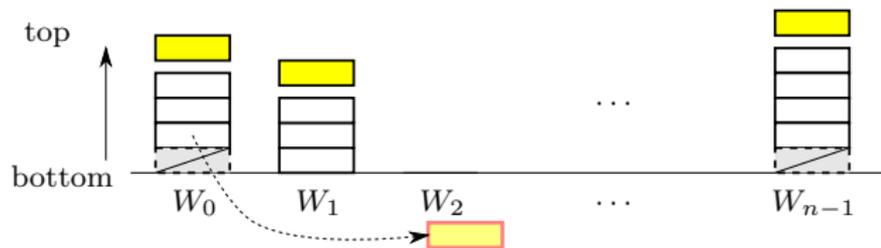
- (3) a worker W encounters `wait_tasks`: there are two cases
- 1 tasks to wait for have finished $\Rightarrow W$ just continues the task

Work stealing scheduler in action



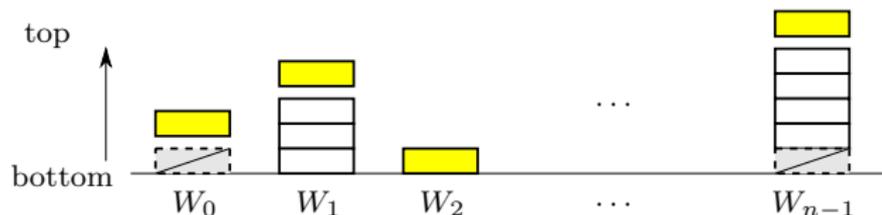
- (3) a worker W encounters `wait_tasks`: there are two cases
- 1 tasks to wait for have finished $\Rightarrow W$ just continues the task
 - 2 otherwise \Rightarrow pops the task from its deque (the task is now *blocked*, and W will start work stealing)

Work stealing scheduler in action



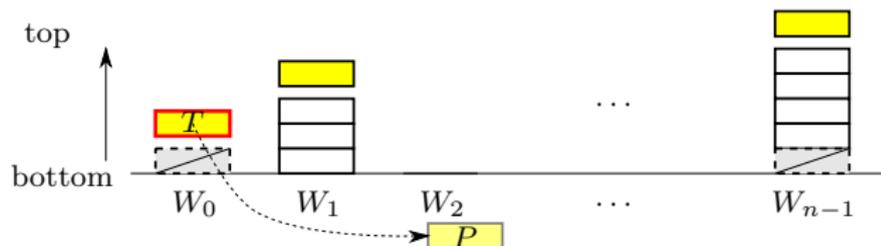
- (3) a worker W encounters `wait_tasks`: there are two cases
- 1 tasks to wait for have finished $\Rightarrow W$ just continues the task
 - 2 otherwise \Rightarrow pops the task from its deque (the task is now *blocked*, and W will start work stealing)

Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

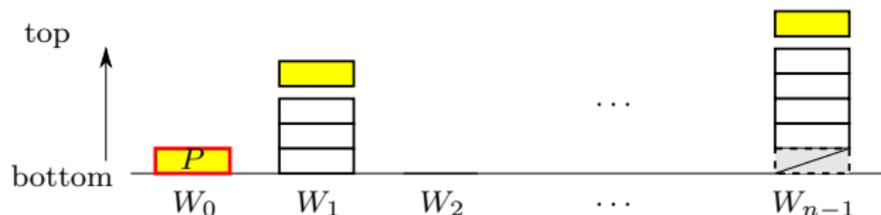
Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

- 1 P has been blocked and now becomes ready again $\Rightarrow W$ enqueues and continues to P

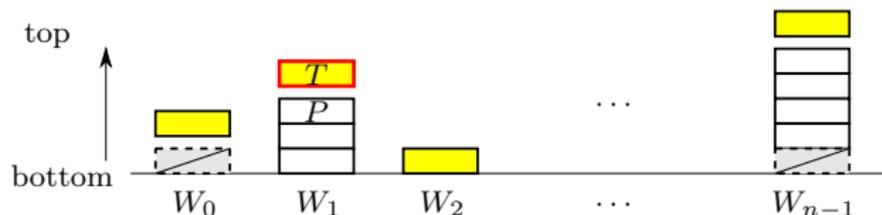
Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

- 1 P has been blocked and now becomes ready again $\Rightarrow W$ enqueues and continues to P

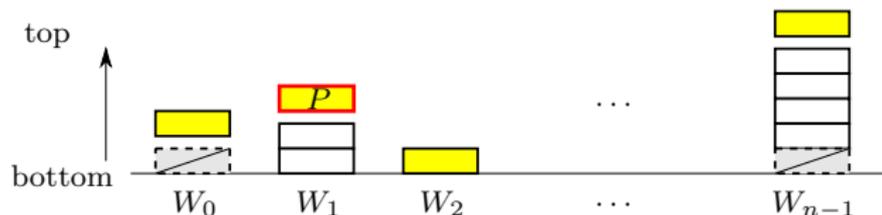
Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

- 1 P has been blocked and now becomes ready again $\Rightarrow W$ enqueues and continues to P
- 2 other cases \Rightarrow no particular action; continues to the next task in its deque or starts work stealing if it becomes empty

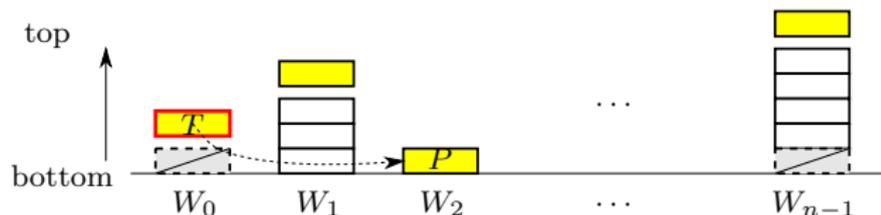
Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

- 1 P has been blocked and now becomes ready again $\Rightarrow W$ enqueues and continues to P
- 2 other cases \Rightarrow no particular action; continues to the next task in its deque or starts work stealing if it becomes empty

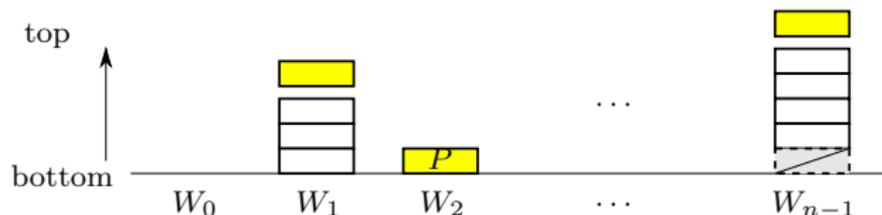
Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

- 1 P has been blocked and now becomes ready again $\Rightarrow W$ enqueues and continues to P
- 2 other cases \Rightarrow no particular action; continues to the next task in its deque or starts work stealing if it becomes empty

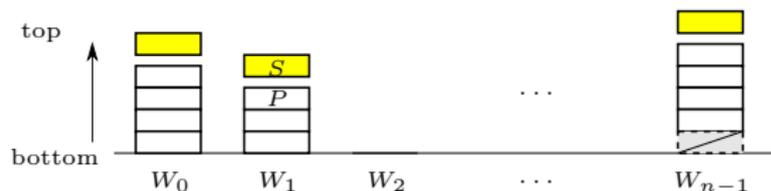
Work stealing scheduler in action



(4) when W encounters the termination of a task T , W pops T from its deque. there are two cases about T 's parent P :

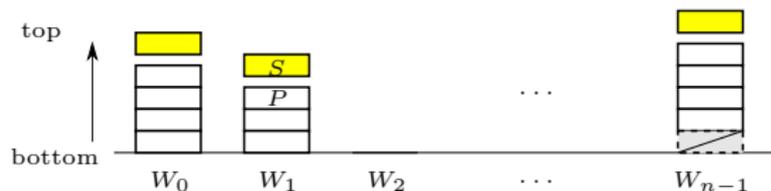
- 1 P has been blocked and now becomes ready again $\Rightarrow W$ enqueues and continues to P
- 2 other cases \Rightarrow no particular action; continues to the next task in its deque or starts work stealing if it becomes empty

A note about the cost of operations



- with work stealing, *the cost of a task creation is cheap, unless its parent is stolen*
 - 1 a task gets created
 - 2 the control jumps to the new task
 - 3 when finished, the control returns back to the parent (as it has not been stolen)

A note about the cost of operations



- much like a procedure call, except:
 - the parent and the child each needs a separate stack, as the parent *might be* executed concurrently with the child,
 - as the parent *might be* executed without returning from the child, the parent generally cannot assume callee save registers are preserved
- *the net overhead is ≈ 100 - 200 instructions*, from task creation to termination

What you must remember when using work stealing systems

- when using (good) work stealing scheduler, don't try to match the number of tasks to the number of processors
 - **bad idea 1:** create $\approx P$ tasks when you have P processors
 - **bad idea 2 (task pooling):** keep exactly P tasks all the time and let each grab work
- they are effective with OS-managed threads or processes but not with work stealing schedulers
- **remember:** keep the granularity of a task above a constant factor of task creation overhead (so the relative overhead is a sufficiently small constant. e.g., 2%)
 - **good idea:** make the granularity ≥ 5000 cycles

Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time**
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time**
 - **Introduction**
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

Analyzing execution time of work stealing

- we analyze execution time of work stealing scheduler in terms of T_1 (total work) and T_∞ (critical path)

Blumofe et al. *Scheduling multithreaded computations by work stealing* Journal of the ACM 46(5). 1999.

- due to the random choices of victims, the upper bound is necessarily probabilistic (e.g., $T_P \leq \dots$ with a probability $\geq \dots$)
- for mathematical simplicity, we are satisfied with a result about *average (expected)* execution time

Analyzing execution time of work stealing

- the main result: with P processors,

$$E(T_P) \leq \frac{T_1}{P} + cT_\infty,$$

with c a small constant reflecting the cost of a work steal

- remember the greedy scheduler theorem?

$$T_P \leq \frac{T_1}{P} + T_\infty$$

Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time**
 - Introduction
 - DAG model and greedy schedulers**
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

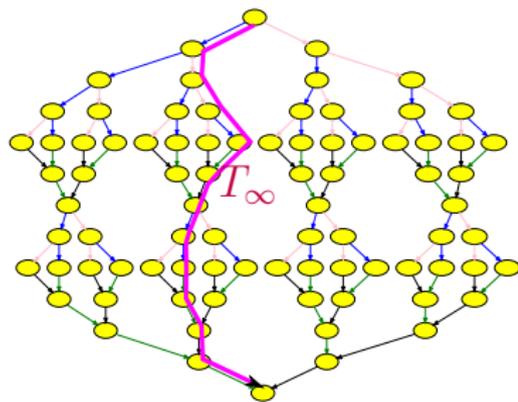
Recap : DAG model

- recall the DAG model of computation
- T_1 : total work (= execution time by a single processor)
- T_∞ : critical path (= execution time by an arbitrarily many processors)
- T_P : execution time by P processors
- two obvious *lower bounds* of execution time of *any* scheduler:

$$T_P \geq \frac{T_1}{P} \text{ and } T_P \geq T_\infty$$

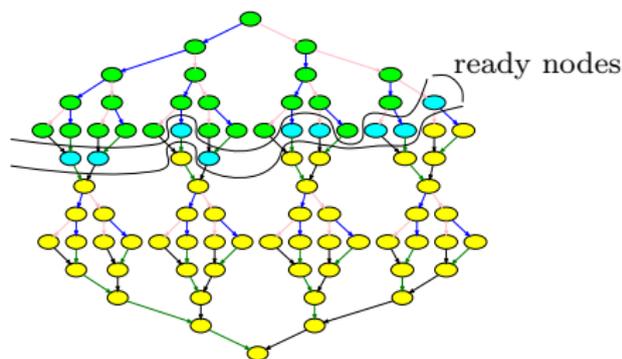
or equivalently,

$$T_P \geq \max \left(\frac{T_1}{P}, T_\infty \right)$$



Recap : greedy scheduler

- greedy scheduler: *“a worker is never idle, as long as any ready task is left”*



- the greedy scheduler theorem: *any greedy scheduler* achieves the following upper bound

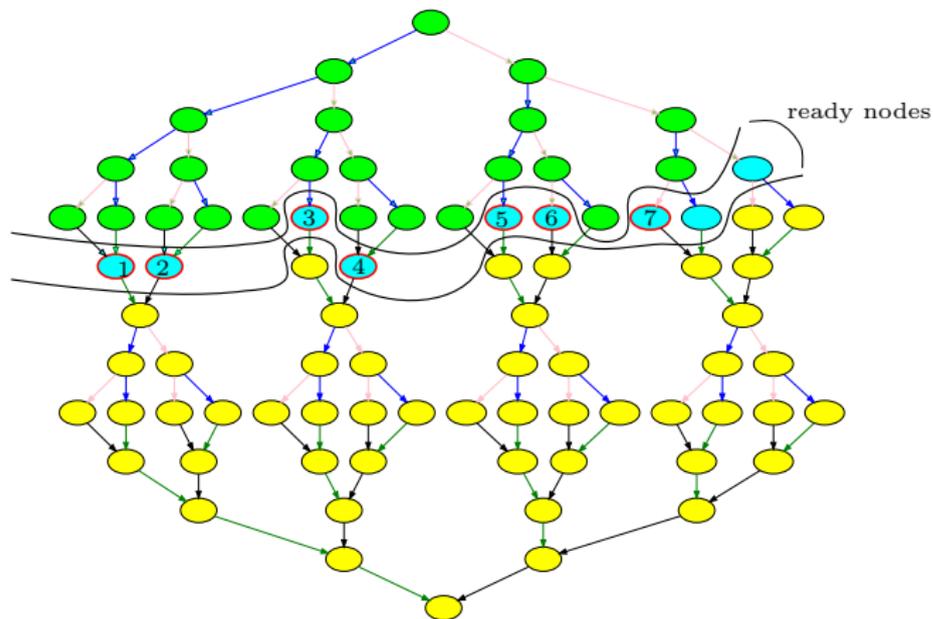
$$T_P \leq \frac{T_1}{P} + T_\infty$$

- considering both $\frac{T_1}{P}$ and T_∞ are lower bounds, this shows any greedy scheduler is within a factor of two of optimal

Proof of the greedy scheduler theorem : settings

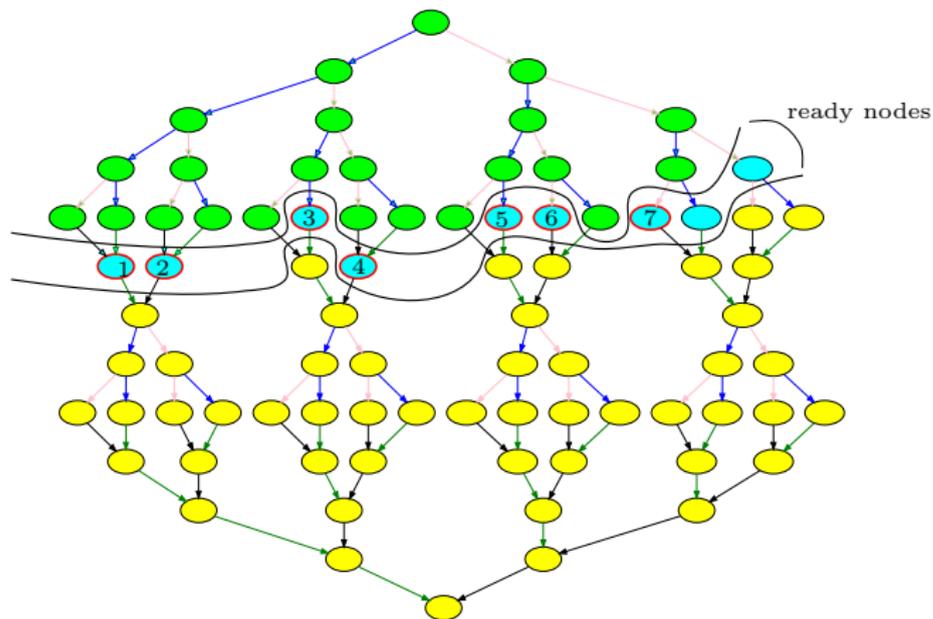
- for the sake of simplifying analysis, assume all nodes take a unit time to execute (longer nodes can be modeled by a chain of unit-time nodes)
- there are P workers
- workers execute in a “lockstep” manner

Proof of the greedy scheduler theorem : settings



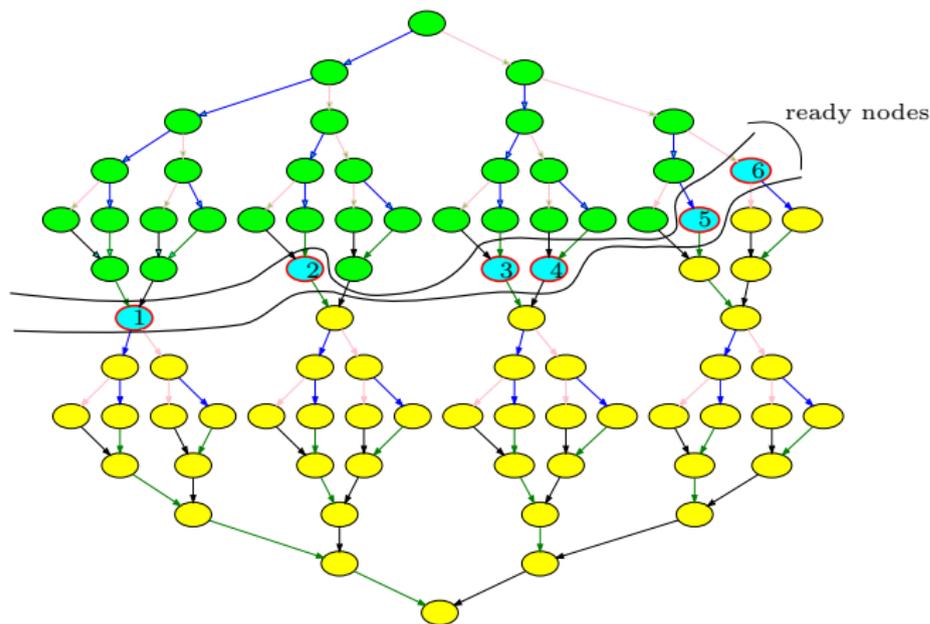
- in each time step, either of the following happens

Proof of the greedy scheduler theorem : settings



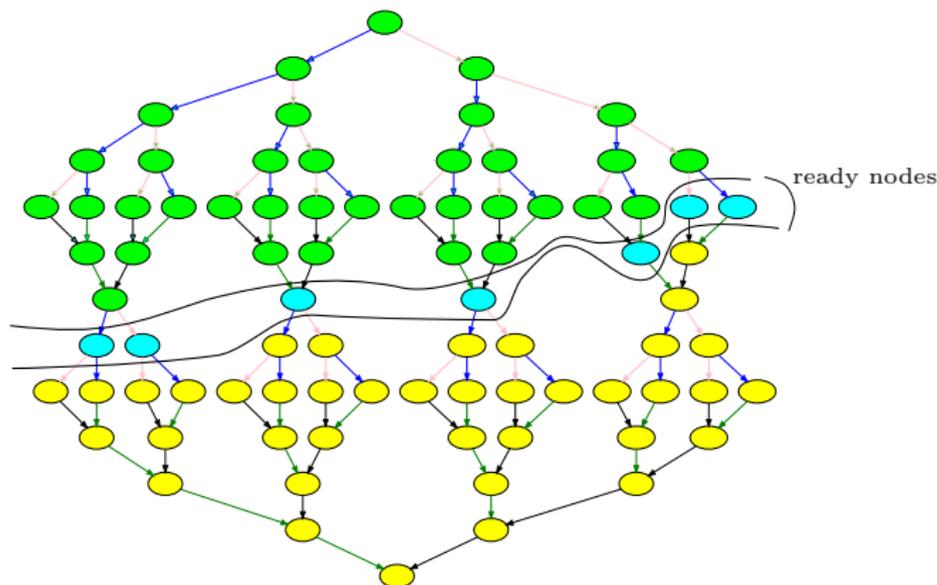
- in each time step, either of the following happens
 - (S) there are $\geq P$ ready tasks
 \Rightarrow each worker executes any ready task (*Saturated*)

Proof of the greedy scheduler theorem : settings



- in each time step, either of the following happens
 - (S) there are $\geq P$ ready tasks
 \Rightarrow each worker executes any ready task (*Saturated*)
 - (U) there are $\leq P$ ready tasks
 \Rightarrow each ready task is executed by any worker (*Unsaturated*)

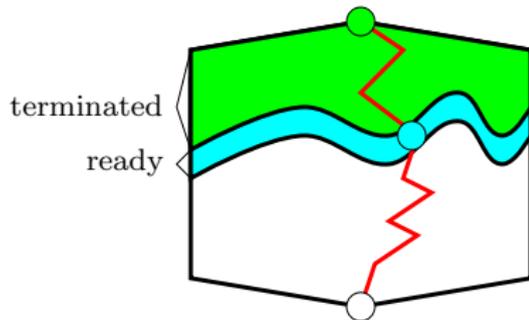
Proof of the greedy scheduler theorem : settings



- in each time step, either of the following happens
 - (S) there are $\geq P$ ready tasks
 \Rightarrow each worker executes any ready task (*Saturated*)
 - (U) there are $\leq P$ ready tasks
 \Rightarrow each ready task is executed by any worker (*Unsaturated*)

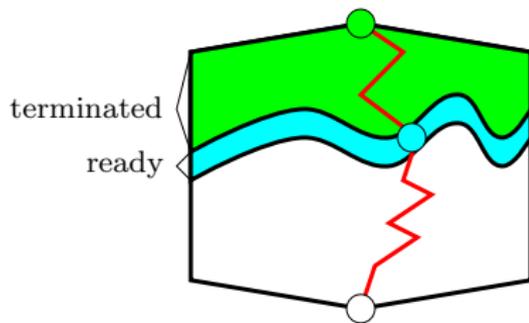
Proof of the greedy scheduler theorem

- there is a path from the start node to the end node, along which a node is always ready (let's call such a path *a ready path*)
 - exercise: prove there is a ready path
- at *each* step, either of the following must happen



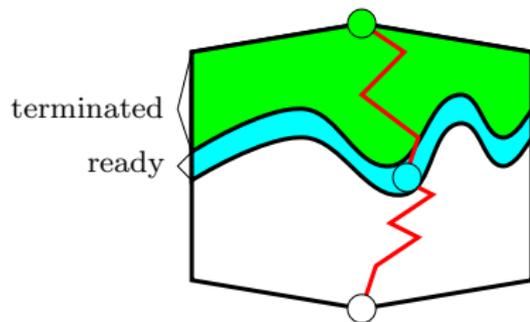
Proof of the greedy scheduler theorem

- there is a path from the start node to the end node, along which a node is always ready (let's call such a path *a ready path*)
 - exercise: prove there is a ready path
- at *each* step, either of the following must happen
 - (S) all P workers execute a node, or



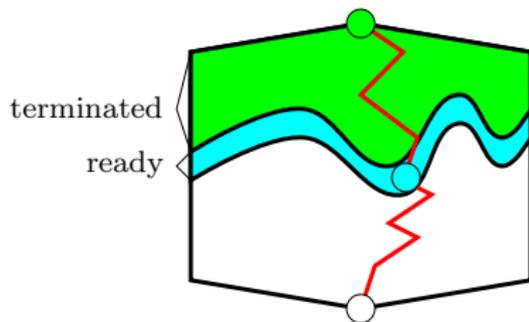
Proof of the greedy scheduler theorem

- there is a path from the start node to the end node, along which a node is always ready (let's call such a path *a ready path*)
 - exercise: prove there is a ready path
- at *each* step, either of the following must happen
 - (S) all P workers execute a node, or
 - (U) the ready node on the path gets executed



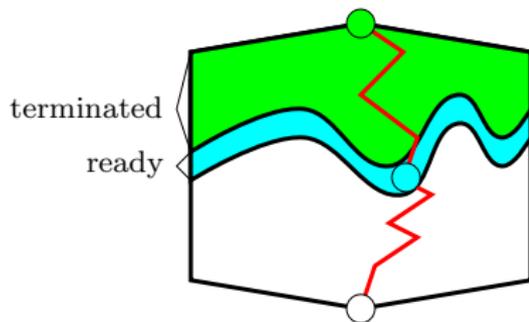
Proof of the greedy scheduler theorem

- there is a path from the start node to the end node, along which a node is always ready (let's call such a path *a ready path*)
 - exercise: prove there is a ready path
- at *each* step, either of the following must happen
 - (S) all P workers execute a node, or
 - (U) the ready node on the path gets executed
- (S) happens $\leq T_1/P$ times and (U) $\leq T_\infty$ times



Proof of the greedy scheduler theorem

- there is a path from the start node to the end node, along which a node is always ready (let's call such a path *a ready path*)
 - exercise: prove there is a ready path
- at *each* step, either of the following must happen
 - (S) all P workers execute a node, or
 - (U) the ready node on the path gets executed
- (S) happens $\leq T_1/P$ times and (U) $\leq T_\infty$ times
- therefore, the end node will be executed within



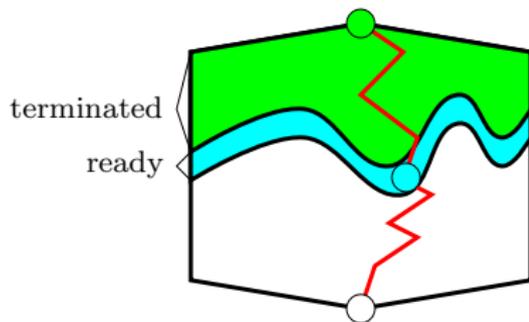
$$T_P \leq (T_1/P + T_\infty) \text{ steps}$$

Proof of the greedy scheduler theorem

- there is a path from the start node to the end node, along which a node is always ready (let's call such a path *a ready path*)
 - exercise: prove there is a ready path
- at *each* step, either of the following must happen
 - (S) all P workers execute a node, or
 - (U) the ready node on the path gets executed
- (S) happens $\leq T_1/P$ times and (U) $\leq T_\infty$ times
- therefore, the end node will be executed within

$$T_P \leq (T_1/P + T_\infty) \text{ steps}$$

- note: you can actually prove $T_P \leq (T_1/P + (1-1/P)T_\infty)$ steps



Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time**
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers**
- 4 Analyzing cache misses of work stealing
- 5 Summary

What about work stealing scheduler?

- what is the difference between the genuine greedy scheduler and work stealing scheduler?
- the greedy scheduler finds ready tasks with *zero delays*

What about work stealing scheduler?

- what is the difference between the genuine greedy scheduler and work stealing scheduler?
- the greedy scheduler finds ready tasks with *zero delays*
- any practically implementable scheduler will inevitably cause some *delays*, from the time a node becomes ready until the time it gets executed

What about work stealing scheduler?

- what is the difference between the genuine greedy scheduler and work stealing scheduler?
- the greedy scheduler finds ready tasks with *zero delays*
- any practically implementable scheduler will inevitably cause some *delays*, from the time a node becomes ready until the time it gets executed
- in the work stealing scheduler, the delay is the time to randomly search other workers' deques for ready tasks, without knowing which deques have tasks to steal

Analyzing work stealing scheduler : settings

- a similar setting with greedy scheduler
 - all nodes take a unit time
 - P workers

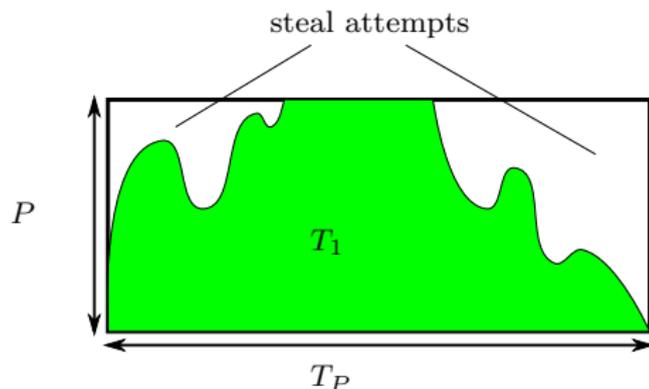
Analyzing work stealing scheduler : settings

- a similar setting with greedy scheduler
 - all nodes take a unit time
 - P workers
- each worker does the following in each time step
 - its deque is not empty \Rightarrow executes the node designated by the algorithm
 - its deque is empty \Rightarrow attempts to steal a node; if it succeeds, the node will be executed in the next step

Analyzing work stealing scheduler : settings

- a similar setting with greedy scheduler
 - all nodes take a unit time
 - P workers
- each worker does the following in each time step
 - its deque is not empty \Rightarrow executes the node designated by the algorithm
 - its deque is empty \Rightarrow attempts to steal a node; if it succeeds, the node will be executed in the next step
- remarks on work stealing attempts
 - if the chosen victim has no ready tasks (besides the one executing), an attempt fails
 - when two or more workers choose the same victim, only one can succeed in a single time

The overall strategy of the proof

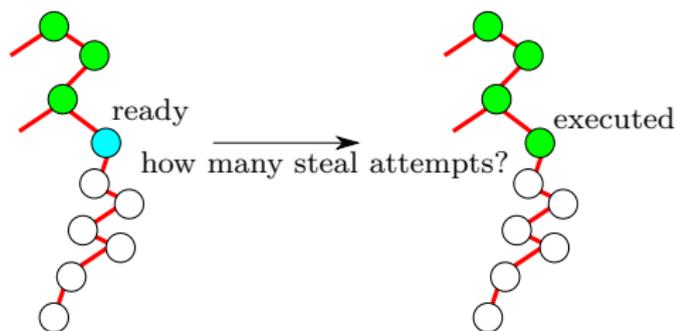


- in each time step, each processor either executes a node or attempts a steal
- \Rightarrow if we can estimate the number of steal attempts (succeeded or not), we can estimate the execution time, as:

$$T_P = \frac{T_1 + \text{steal attempts}}{P}$$

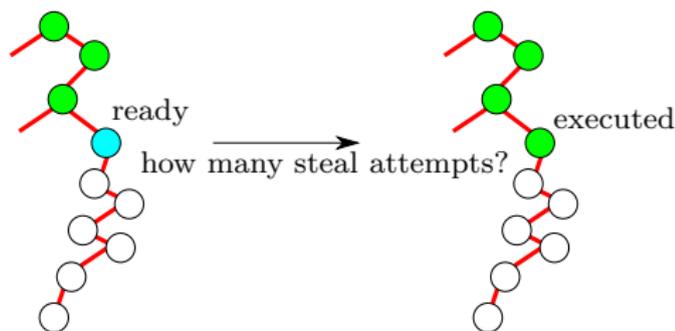
- *our goal is to estimate the number of steal attempts*

Analyzing work stealing scheduler



- similar to the proof of greedy scheduler, consider a ready path (a path from the start node to the end node, along which a node is always ready)

Analyzing work stealing scheduler



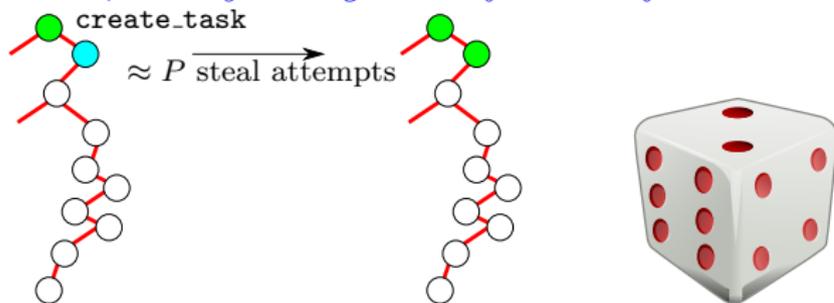
- similar to the proof of greedy scheduler, consider a ready path (a path from the start node to the end node, along which a node is always ready)
- the crux is to estimate how many steal attempts are enough to make a “progress” along the ready path

The number of steal attempts: the key observation

- a task at the bottom of the deque will be stolen by a steal attempt with probability $1/P$

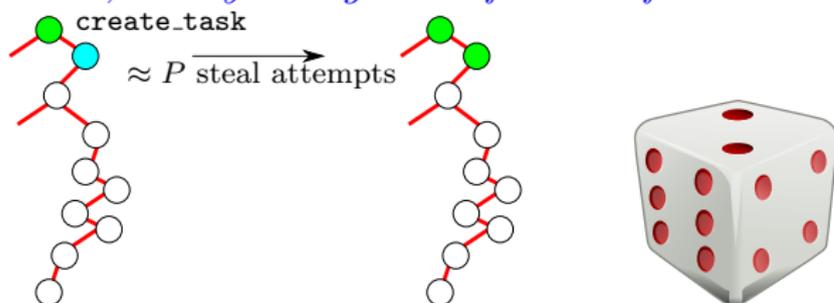
The number of steal attempts: the key observation

- a task at the bottom of the deque will be stolen by a steal attempt with probability $1/P$
- thus, on average, such a task will be stolen, on average, with P steal attempts, and is executed in the next step
you roll a dice, and you'll get the first 6 after six rolls



The number of steal attempts: the key observation

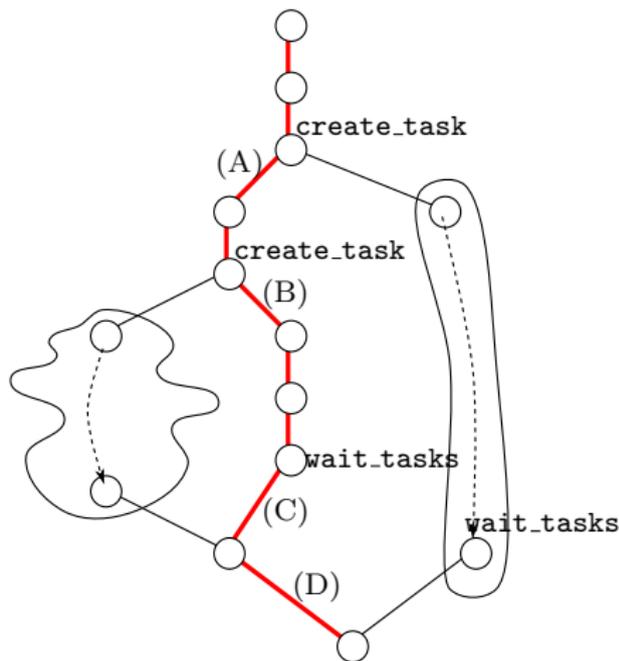
- a task at the bottom of the deque will be stolen by a steal attempt with probability $1/P$
- thus, on average, such a task will be stolen, on average, with P steal attempts, and is executed in the next step
you roll a dice, and you'll get the first  after six rolls



- we are going to extend the argument to *any* ready task along a ready path, and establish an average number of steal attempts for any ready task to get executed

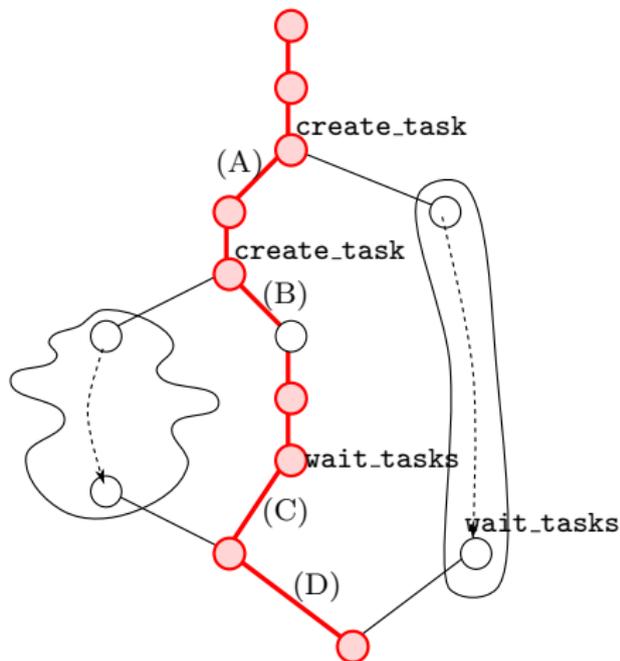
How many steal attempts to occur for *any* ready task to get executed?

- there are five types of edges
 - (A) create_task \rightarrow child
 - (B) create_task \rightarrow the continuation
 - (C) wait_task \rightarrow the continuation
 - (D) last node of a task \rightarrow the parent's continuation after the corresponding wait
 - (E) non-task node \rightarrow the continuation



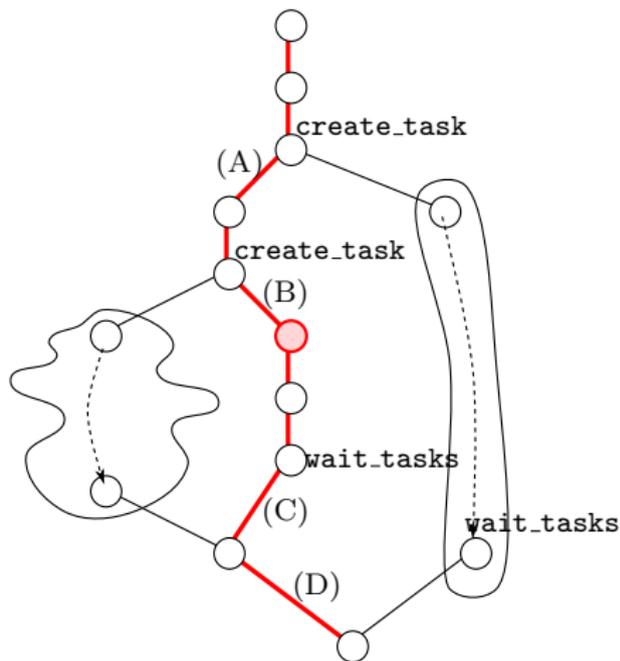
How many steal attempts to occur for *any* ready task to get executed?

- the successor of type (A), (C), (D), and (E) is executed immediately after the predecessor is executed. *there are no delays on edges of these types*



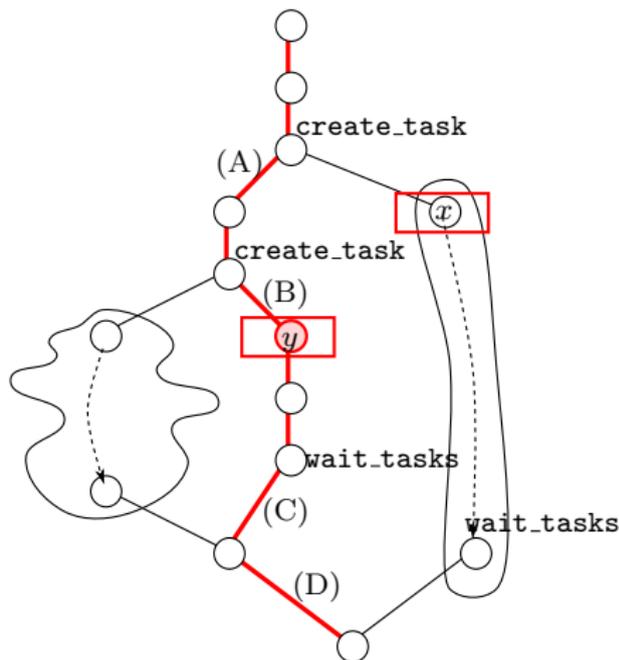
How many steal attempts to occur for *any* ready task to get executed?

- only the successor of type (B) edges may need a steal attempt to get executed
- as has been discussed, once a task is at the bottom of a deque, it needs $\approx P$ steal attempts on average until it gets stolen



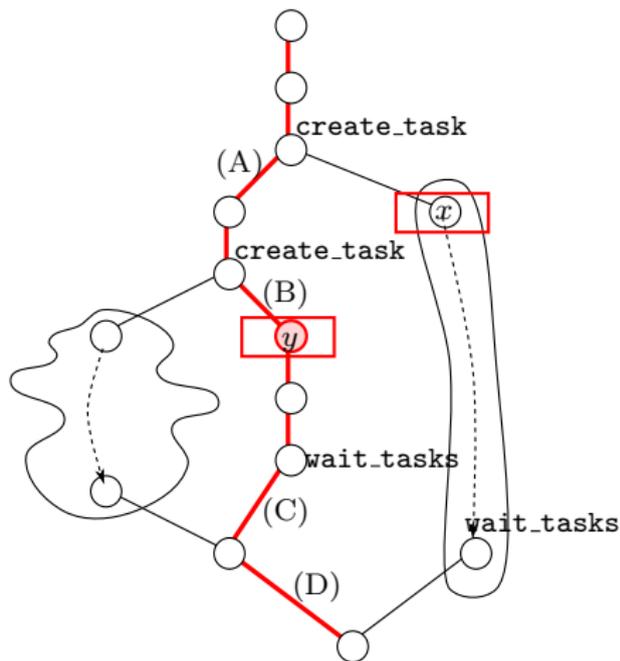
How many steal attempts to occur for *any* ready task to get executed?

- note that a successor of a type (B) edge (continuation of a task creation) is not necessarily at the bottom of a deque
- e.g., y cannot be stolen until x has been stolen



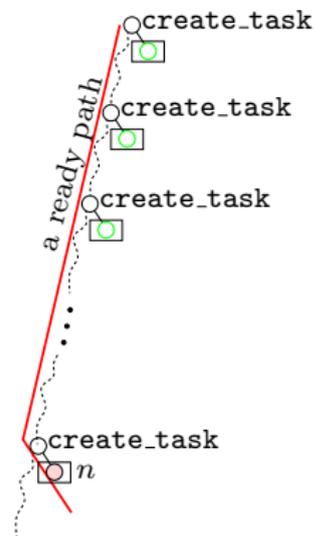
How many steal attempts to occur for *any* ready task to get executed?

- stealing such a task requires an accordingly many steal attempts
- e.g., stealing y requires $2P$ attempts on average (P to steal x and another P to steal y)



When a task becomes stealable?

- in general, in order for the continuation of a `create_task` (n) to be stolen, continuations of all `create_tasks` along the path from the start node to n must be stolen



Summary of the proof

Now we have all ingredients to finish the proof

the average number of steal attempts to finish the ready path

therefore,

$$\text{average of } T_p \leq \frac{T_1 + PT_\infty}{P} = \frac{T_1}{P} + T_\infty$$

Summary of the proof

Now we have all ingredients to finish the proof

the average number of steal attempts to finish the ready path
 $\approx P \times$ the length of the ready path

therefore,

$$\text{average of } T_p \leq \frac{T_1 + PT_\infty}{P} = \frac{T_1}{P} + T_\infty$$

Summary of the proof

Now we have all ingredients to finish the proof

the average number of steal attempts to finish the ready path

$\approx P \times$ the length of the ready path

$\leq PT_\infty$

therefore,

$$\text{average of } T_p \leq \frac{T_1 + PT_\infty}{P} = \frac{T_1}{P} + T_\infty$$

Extensions

- we assumed a steal attempt takes a single time step, but it can be generalized to a setting where a steal attempt takes a time steps,

$$E(T_P) \leq \frac{T_1}{P} + aT_\infty$$

- we can also probabilistically bound the execution time
- the basis is the probability that a critical node takes cP steal attempts to be executed is $\leq e^{-c}$

$$\because \left(1 - \frac{1}{P}\right)^{cP} \leq e^{-c}$$

- based on this we bound the probability that a path of length l takes ClP steal attempts, for a large enough constant C

Contents

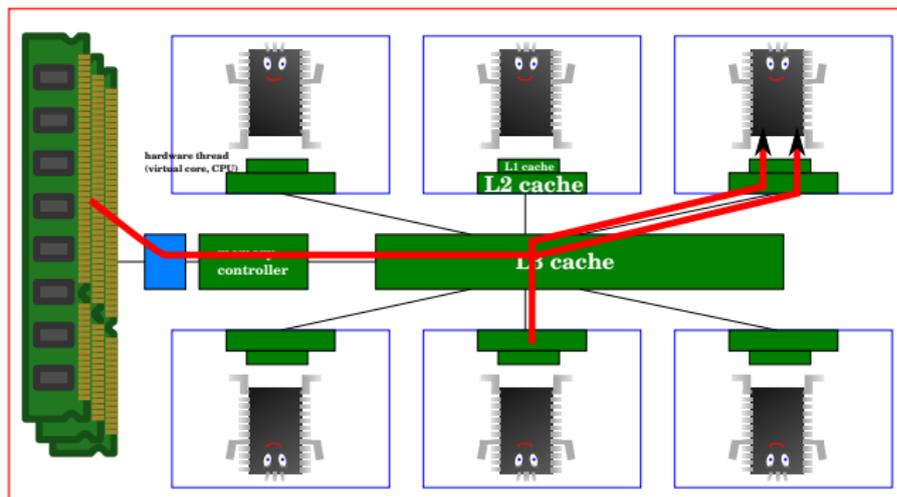
- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

Analyzing cache misses of work stealing

- we like to know the amount of data transfers between a processor's cache and { main memory, other caches }, under a task parallel scheduler

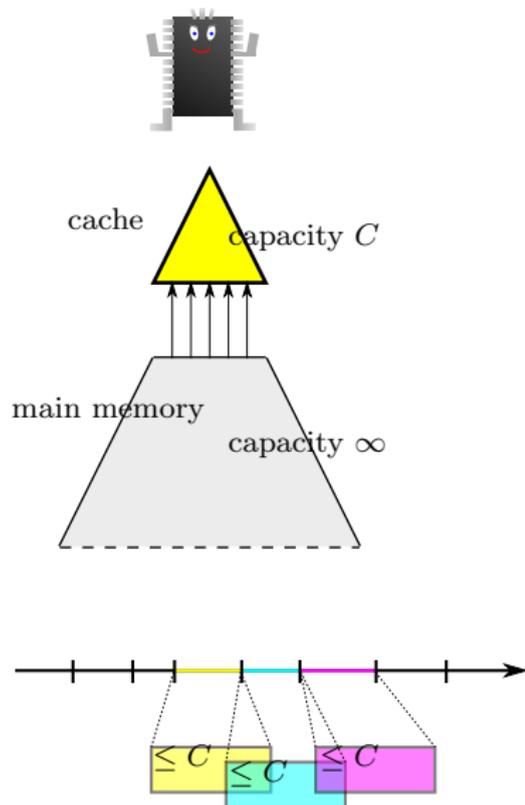
Analyzing cache misses of work stealing

- we like to know the amount of data transfers between a processor's cache and { main memory, other caches }, under a task parallel scheduler
- in particular, we like to understand how much can it be worse (or better) than its serial execution



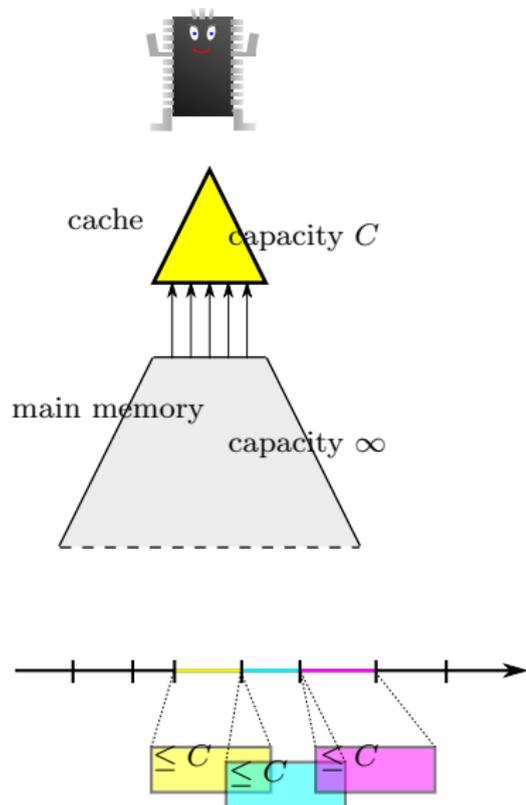
An analysis methodology of serial computation

- we have learned how to analyze data transfer between the cache and main memory, in single processor machines



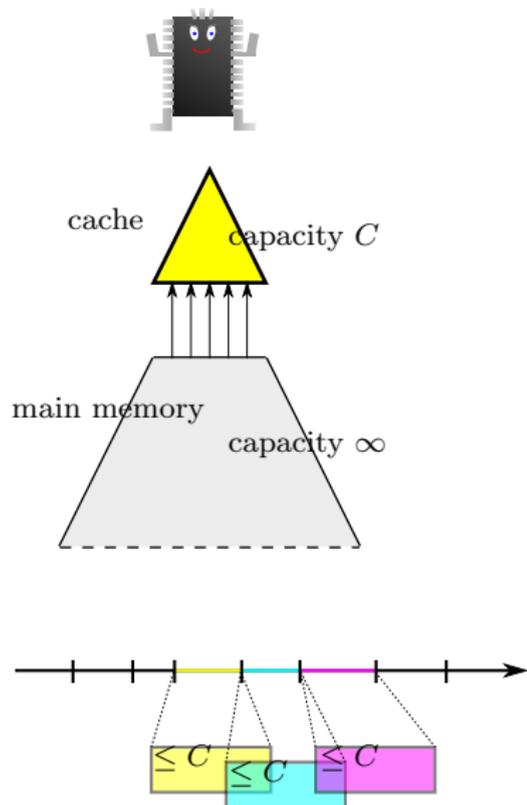
An analysis methodology of serial computation

- we have learned how to analyze data transfer between the cache and main memory, in single processor machines
- the key was to identify “cache-fitting” subcomputations (working set size $\leq C$ words); and



An analysis methodology of serial computation

- we have learned how to analyze data transfer between the cache and main memory, in single processor machines
- the key was to identify “cache-fitting” subcomputations (working set size $\leq C$ words); and
- a cache-fitting subcomputation induces $\leq C$ words data transfers

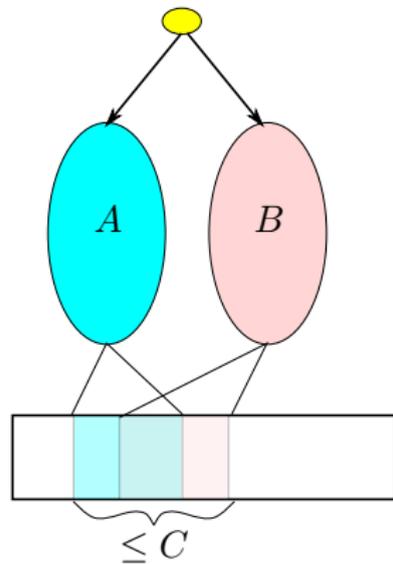


Minor remarks (data transfer vs. cache misses)

- we hereafter use “a single cache miss” to mean “a single data transfer from/to a cache”
- in real machines, some data transfers do not induce to cache misses due to prefetches
- we say “cache misses” for simplicity

What's different in parallel execution?

- the argument that “cache misses by a cache-fitting subcomputation $\leq C$ ” no longer holds in parallel execution

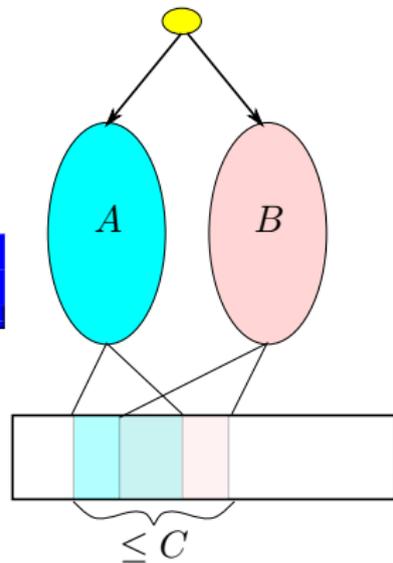


What's different in parallel execution?

- the argument that “cache misses by a cache-fitting subcomputation $\leq C$ ” no longer holds in parallel execution
- consider two subcomputations A and B

```
1 create_task({ A });  
2 B
```

- assume A and B together fit in the cache
- even so, if A and B are executed on different processors, originally a cache hit in B may miss



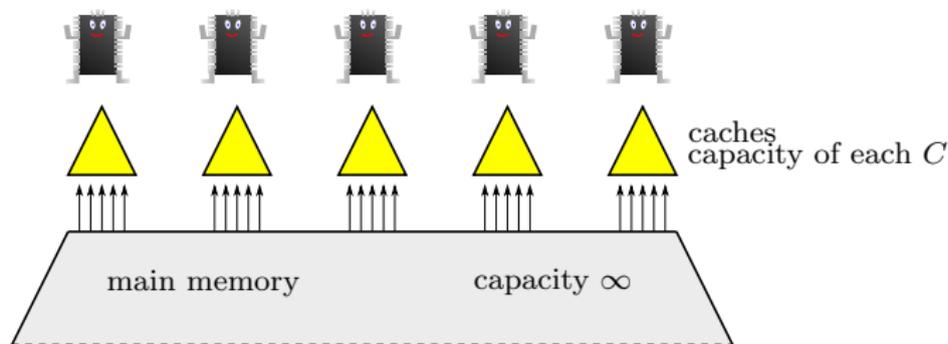
What's different in parallel execution?

- so a parallel execution might increase cache misses
- but how much?

The data locality of work stealing. SPAA '00 Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures.

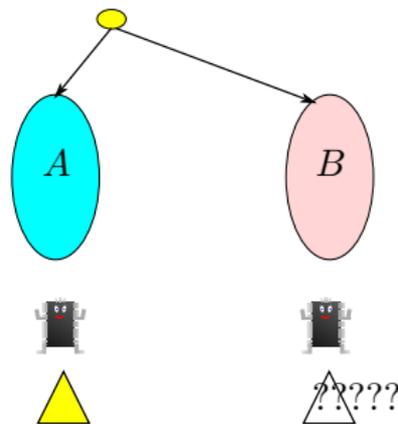
Problem settings

- P processors ($\equiv P$ workers)
- caches are *private* to each processor (no shared caches)
- consider only a single-level cache, with the capacity of C words
- *LRU replacement*: i.e., a cache holds most recently accessed C distinct words



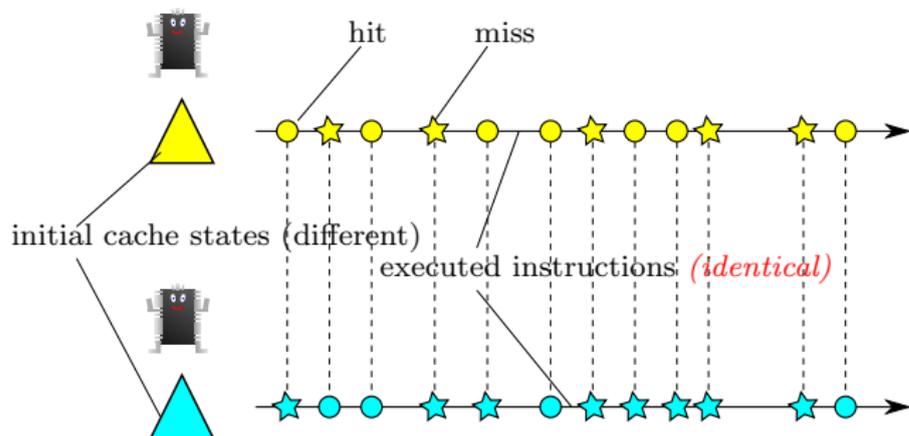
The key observation

- we have learned the work stealing scheduler tends to preserve much of the serial execution order
- \Rightarrow extra cache misses are caused by work stealings
- a work stealing essentially brings a subcomputation to *a processor with unknown cache states*



The key questions

- **key question 1:** how many *extra* misses can occur when a subcomputation moves to an unknown cache states?



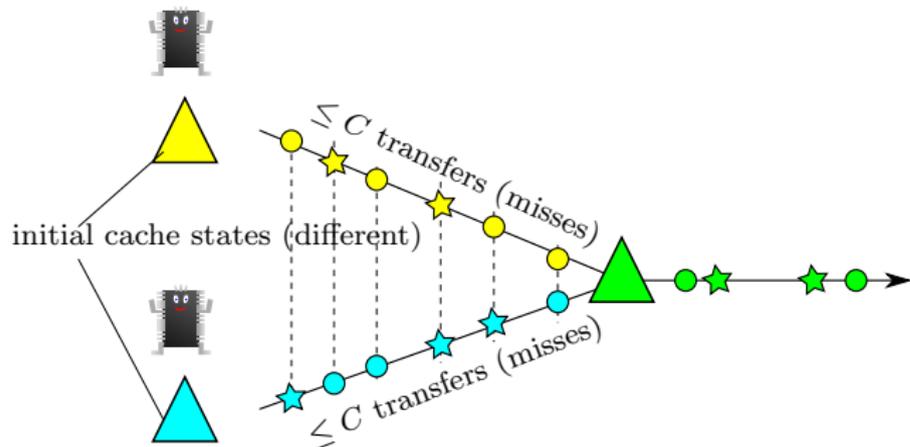
- **key question 2:** how many times work stealings happen? (we know an answer)

Roadmap

- (1) bound the number of extra cache misses that occurs each time a node is *drifted* (i.e., executed in a different order with the serial execution)
- (2) we know an upper bound on the number of steals
- (3) from (2), bound the number of drifted nodes
 - combine (1) and (3) to derive an upper bound on the total number of extra cache misses

Extra misses per drifted node

- when caches are LRU, the two cache states converge to an identical state, after no more than C cache misses occur in either cache

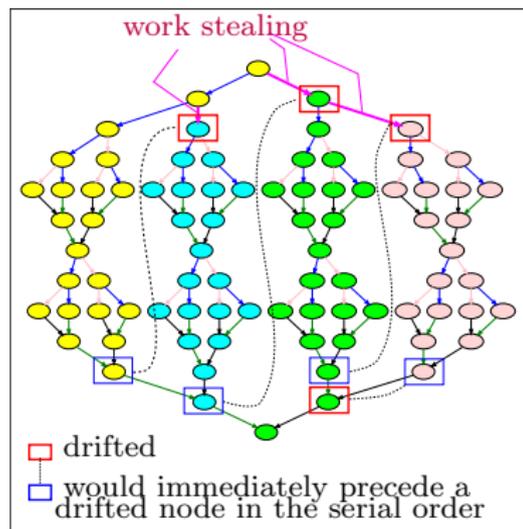


- this is because the cache is LRU (holds most recently accessed distinct C words)

\therefore *extra cache misses for each drifted node* $\leq C$

A bound on the drifted nodes

- let v a node in the DAG and u the node that would immediately precedes v in the serial execution
- we say v is *drifted* when u and v are not executed consecutively on the same processor



- without a detailed proof, we note:

$$\begin{aligned} & \text{the number of drifted nodes in the work stealing scheduler} \\ & \leq 2 \times \text{the number of steals} \end{aligned}$$

The main result

- the average number of work stealings $\approx PT_\infty$
- \Rightarrow the average number of drifted nodes $\approx 2PT_\infty$
- \Rightarrow the average number of extra cache misses $\leq 2CPT_\infty$
- average execution time

$$T_P \leq \frac{T_1}{P} + 2mCT_\infty,$$

where m is the cost of a single cache miss

Contents

- 1 Introduction
- 2 Work stealing scheduler
- 3 Analyzing execution time
 - Introduction
 - DAG model and greedy schedulers
 - Work stealing schedulers
- 4 Analyzing cache misses of work stealing
- 5 Summary

Summary

- the basics of work stealing scheduler
 - work-first (preserve serial execution order)
 - steal tasks from near the root
- average execution time (without cost of communication)

$$T_P \leq \frac{T_1}{P} + T_\infty$$

- with the cost of communication

$$T_P \leq \frac{T_1}{P} + 2mCT_\infty$$

where mC essentially represents the time to fill the cache