Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds

M. A. Bender and M. O. Rabin. "Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds." *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 13-21, 2000.

Presented by Svetlena Taneva

References

- **Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk**, by M. A. Bender and M. O. Rabin. *Theory of Computing Systems Special Issue on SPAA '00, 35: 289-304, 2002.*
- **The Cilk Project Website**, available at: http://supertech.lcs.mit.edu/cilk/papers/index.html
- **Cilk-5.3 Reference Manual**, by Supercomputing Technologies Group. June 2000, available at: http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf
- **Lecture notes from MIT's Theory of Parallel Systems course**, available at: http://theory.lcs.mit.edu/classes/6.895/fall03/

This is NOT about…

- Cilk implemention
- Resource scheduling
- Inter-program concurrency

The topic is…

- How to schedule one (parallel) program on multiple processors

Outline

- Glance at the architecture
- What does Cilk code look like?
- Define the problem
- Introduce concepts
- Present an ideal scheduler
- The standard algorithm
- The enhanced algorithm

What is Cilk?

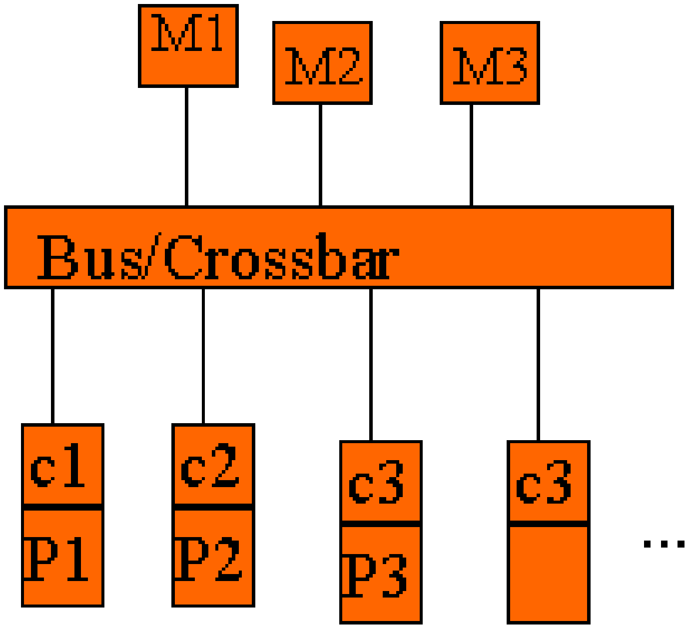
- An *algorithmic* multithreaded language
  - Guaranteed efficient performance
- Expose parallelism & exploit locality
- Runtime system takes care of scheduling
  - Load balancing
  - Paging
  - Communication

Retrospect

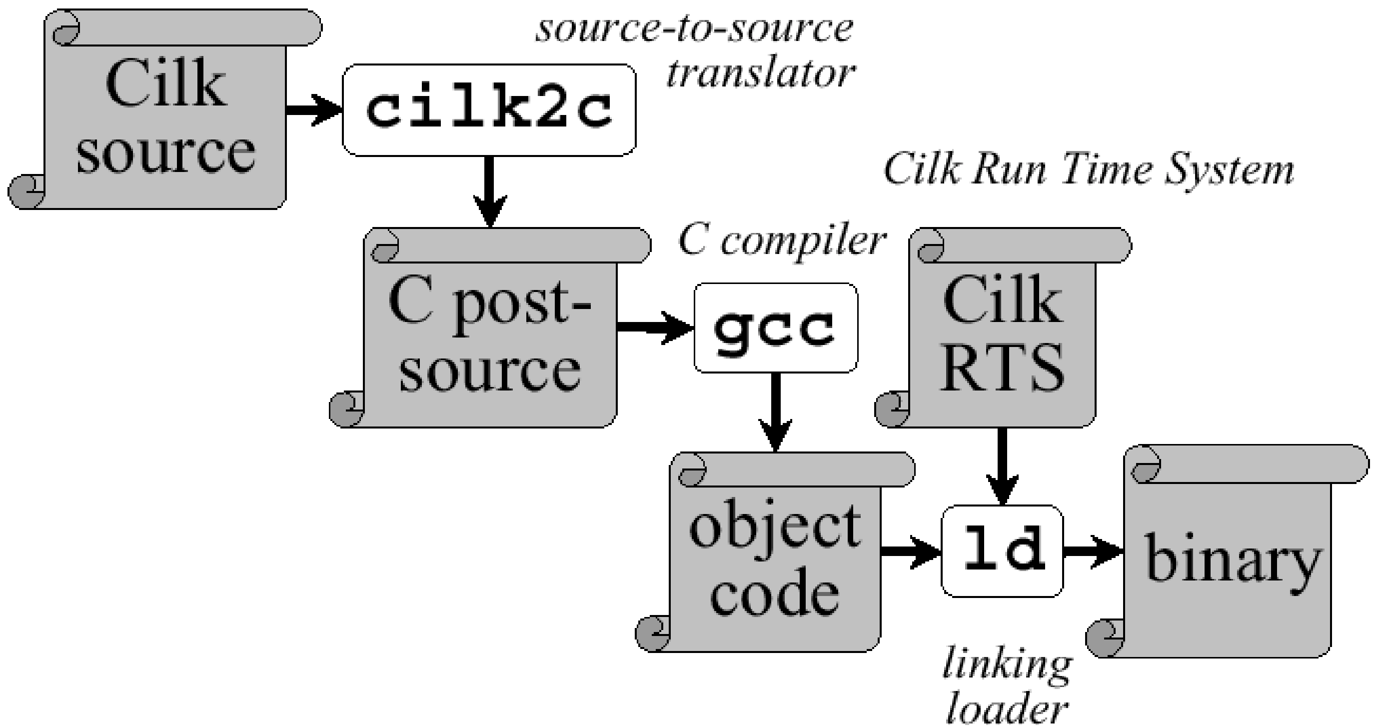
- Theory
  - Scheduling multithreaded computations
  - Work-stealing

Introduction to Cilk

- SMP Computer
  -



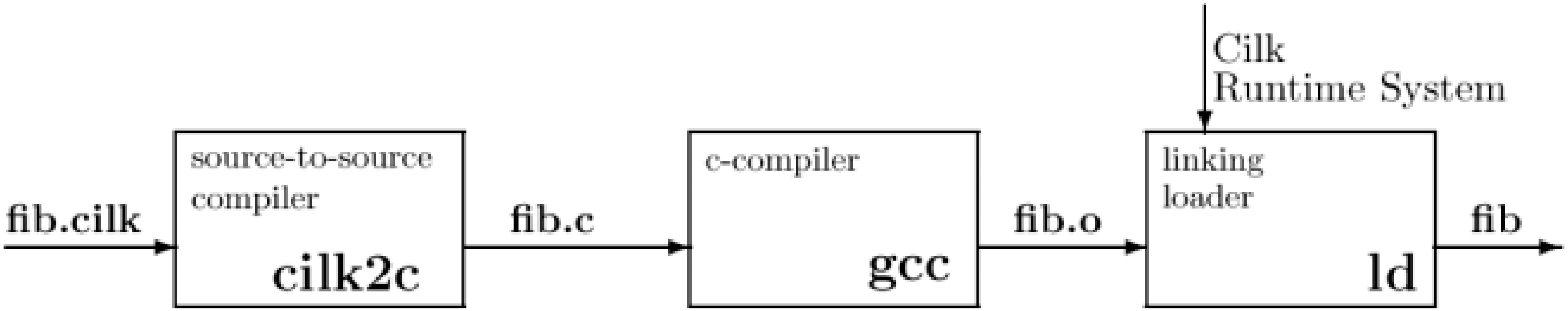
The way it works



Example: Fibonacci

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
        {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
        }
}
```

Compiling the Fibonacci



The Problem

- Execute parallel Cilk programs, on a collection of processors of different and possibly changing speeds
- Factors to be considered:
  - Efficiency
  - Preemptions
  - Migration cost
  - Centralized vs. distributed

The Parallel Setting

- Constraints
  i. Rapid decisions
  ii. Partial knowledge
  iii. Only local state is visible

Solution

- A distributed scheduling algorithm in which each processor maintains an estimate of its own speed, where communication between processors has a cost and all scheduling is done online
    - Processor speed fairly consistent
    - May change occasionally

Outline

- Review concepts & background theory
- Heterogeneous settings
- The Cilk Scheduler
- The Efficiency & Practicality Issues
    - Maximum Utilization Schedule
    - High Utilization Schedule
- Enhanced Cilk Scheduler

Background

- Asynchronous Parallel Computing
  - Correctness and steadiness guaranteed
  - Too pessimistic
- Scheduling Theory
  - Constant processor speeds
  - Global/powerful/offline

- *The above provide an insight*
- *Model of this paper is a bridge between the two*

Concepts

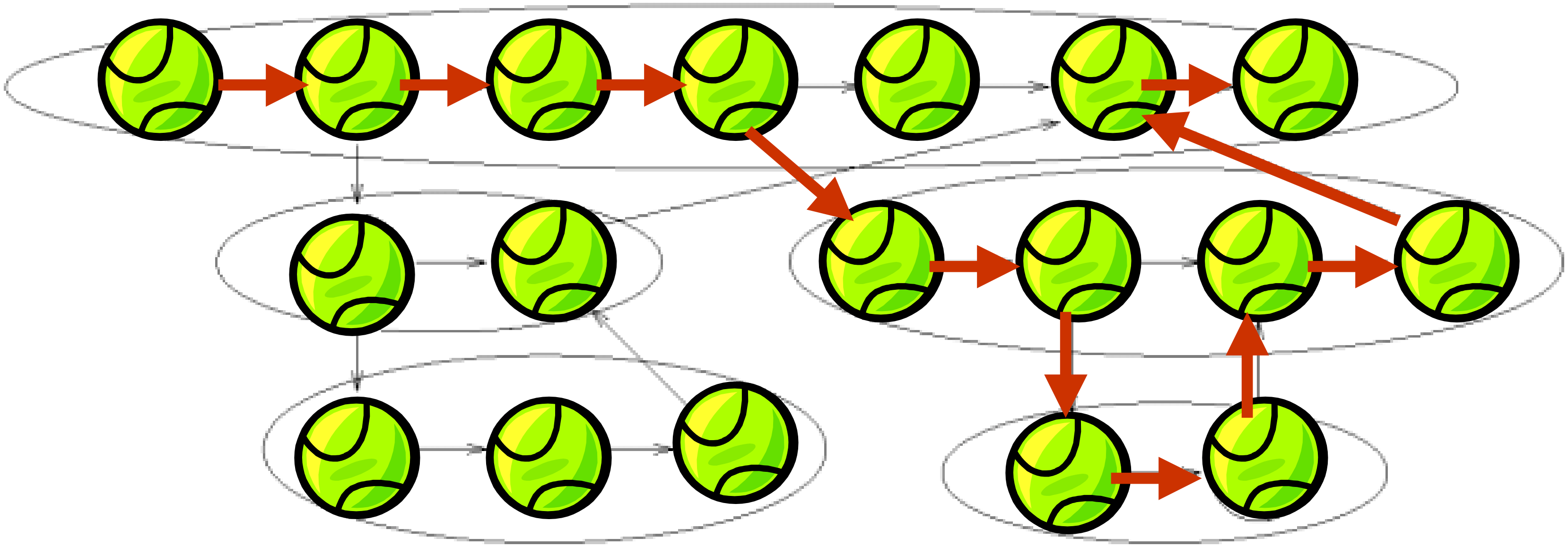
- *Heterogeneous* processors – of different speeds
- *Homogeneous* processors – identical processors
- Makespan

The Heterogeneous Setting

- Greedy schedules – no idling allowed
- Homogeneous processors – comparable makespans
- Heterogeneous processors – there may be an unbounded ratio between makespan of best & worst schedule

⇨Find a scheduler that uses a heterogeneous setting as efficiently as if it was homogeneous

DAG



a node = a thread
an edge = dependencies
subroutine = a group of tasks

What is a thread?

A maximal sequence of instructions that ends with a spawn, sync, or return

(either explicit or implicit) statement.

Threads are…

- *ready* –if all of its predecessors in G have been executed
- *executed*
- *waiting* for predecessors to complete
- *preempted*
- *migration* – the state of the system is moved from one processor to a different one

Back to Fibonacci

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
        {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
        }
}
```

Goal

- Schedule a parallel program represented as a DAG to minimize the makespan
  - NP-hard problem
  - Approximation algorithms
  - Approximation ratio not reliable for heterogeneous settings

Notation

- 1….p — processors
- $\pi_{ave} = \pi_{tot} / p$ — avg speed of the processors
- W_1 — total work (total number of nodes)
- W_∞ — critical path length (# of nodes in longest chain)
- T_p — time to execute the dag on p processors (makespan)

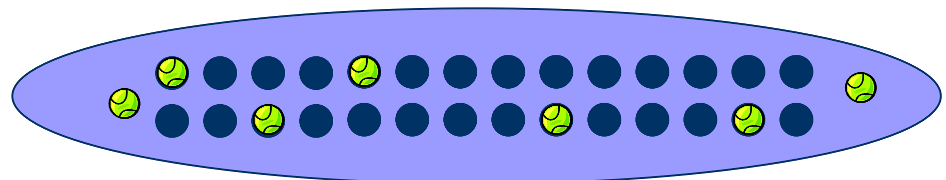
Ideal scheduler (1)

- Greedy scheduler
  - Execute anything that is ready
    in any order
    utilizing as many processors as you have ready tasks

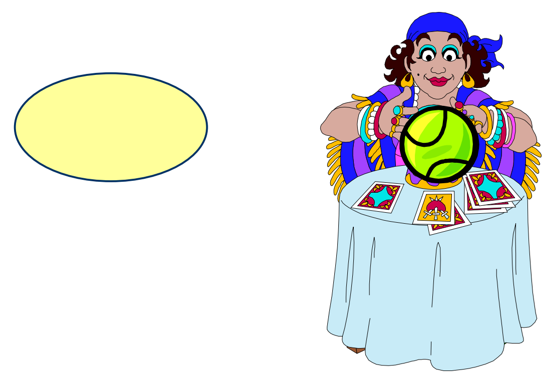
$$T_p \leq \frac{W_1}{P} + W_\infty$$

Ideal scheduler (2)

- Busy Leaves scheduler (to reduce space)

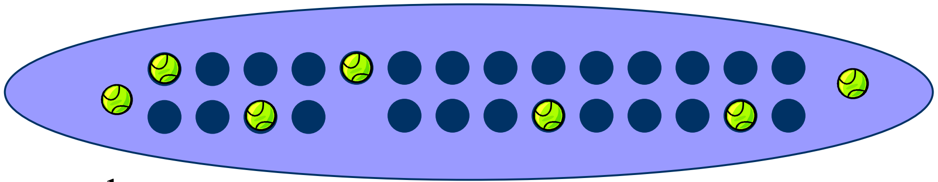


1. If empty, get a new process A from the pool

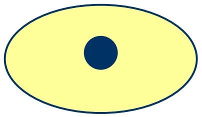


Ideal scheduler (2)

- Busy Leaves scheduler (to reduce space)

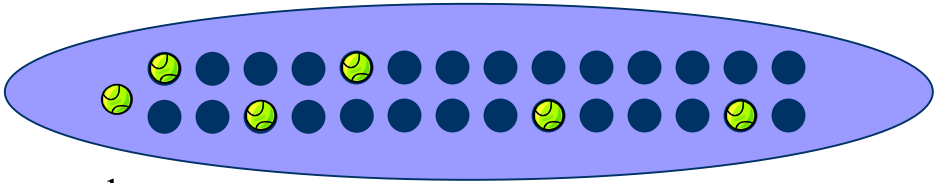


1. If empty, get a new process A from the pool
2. If A spawns a thread B, return A to the pool and commence work on B

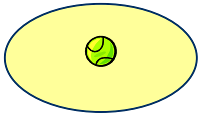


Ideal scheduler (2)

- Busy Leaves scheduler (to reduce space)

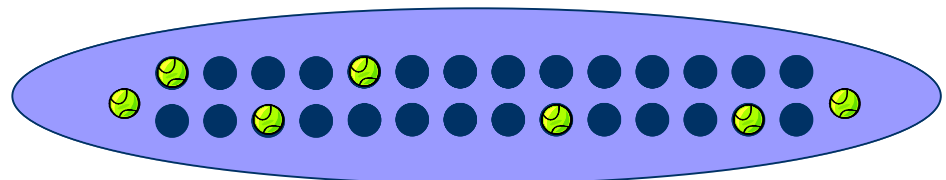


1. If empty, get a new process A from the pool
2. If A spawns a thread B, return A to the pool and commence work on B
3. If A stalls, return A to the pool.

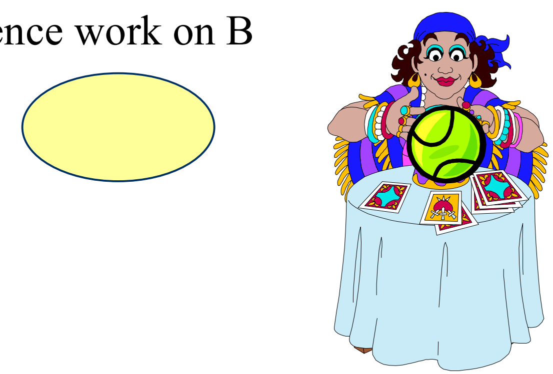


Ideal scheduler (2)

- Busy Leaves scheduler (to reduce space)



1. If empty, get a new process A from the pool
2. If A spawns a thread B, return A to the pool and commence work on B
3. If A stalls, return A to the pool
4. If B returns, check if parent's children have returned. If so, and if A still in the pool, commence work on A.



Ideal scheduler (3)

- Small number of migrations/steals

$$\#\,steals \leq O(PW_{\infty})$$

  - This would give us the following bound:

$$T_P \leq \frac{W_1}{P} + O(W_{\infty})$$

So we want…

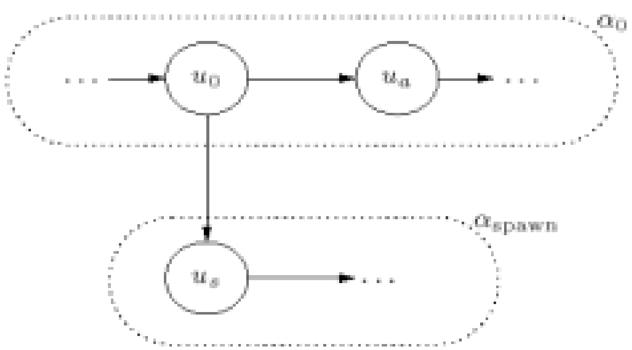
1. To be greedy
2. To be busy
3. And to steal moderately

Goal revisited

- Develop a distributed scheduler that approximates the performance of an "ideal" global scheduler

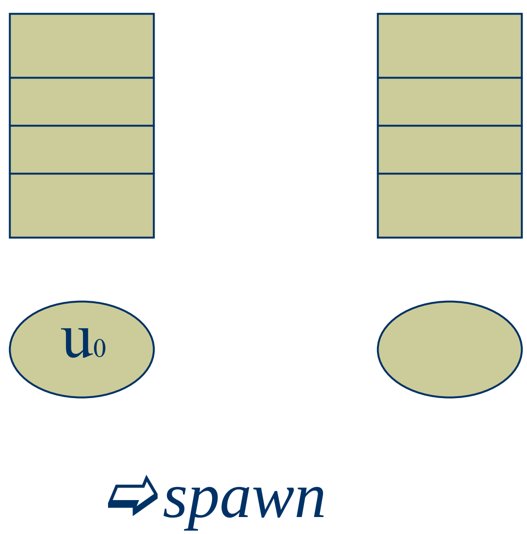
The strategy is…

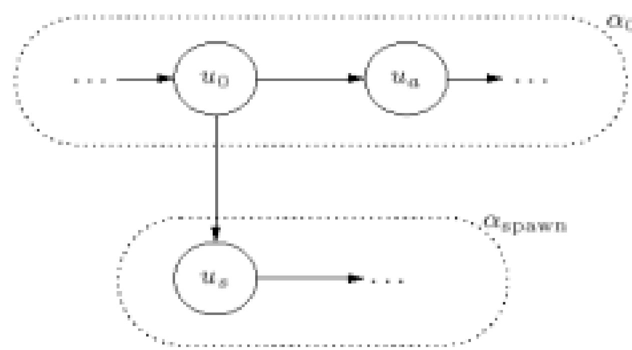
- Online greedy scheduling
- Work-stealing
- NOT work-sharing



The strategy is…

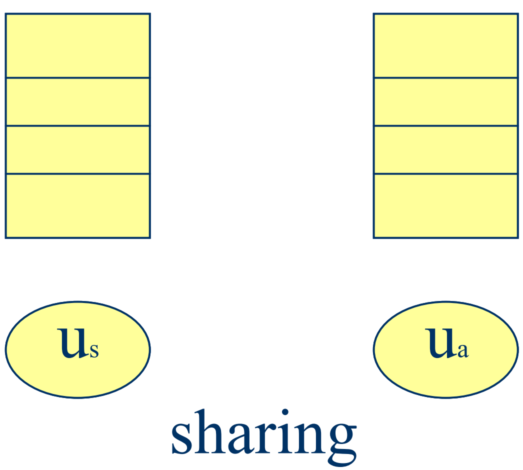
- Online greedy scheduling
- Work-stealing
- NOT work-sharing

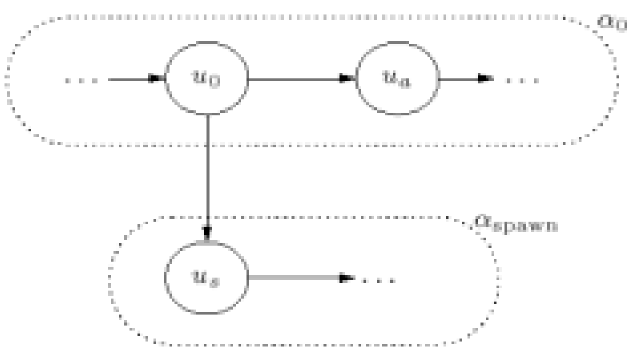




The strategy is…

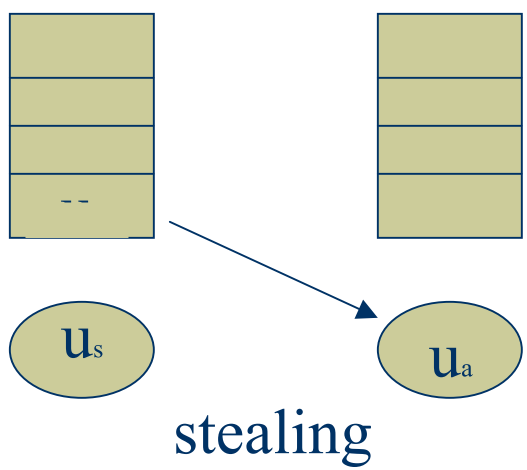
- Online greedy scheduling
- Work-stealing
- NOT work-sharing

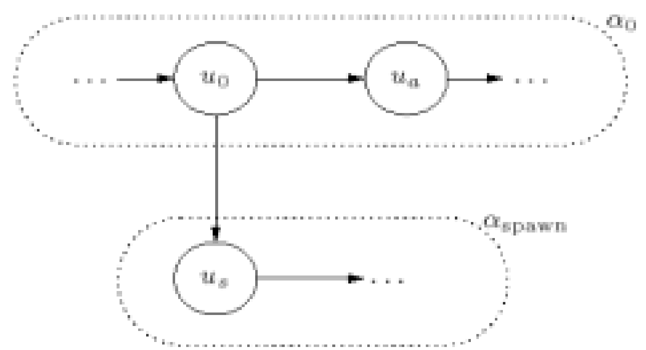




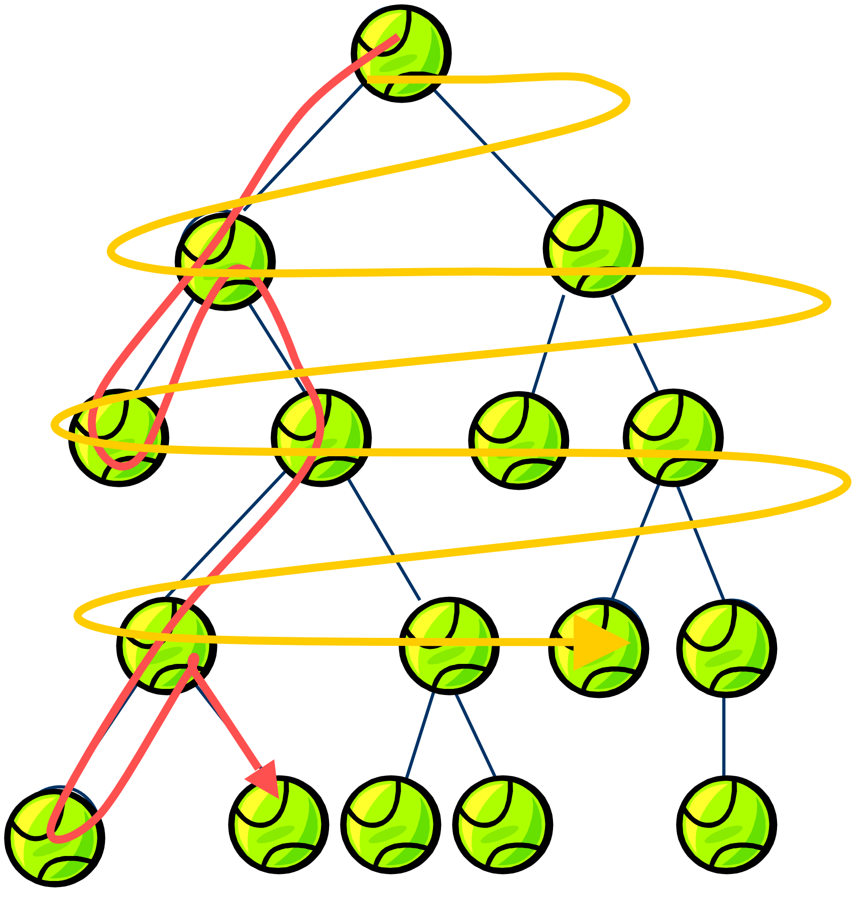
The strategy is…

- Online greedy scheduling
- Work-stealing
- NOT work-sharing

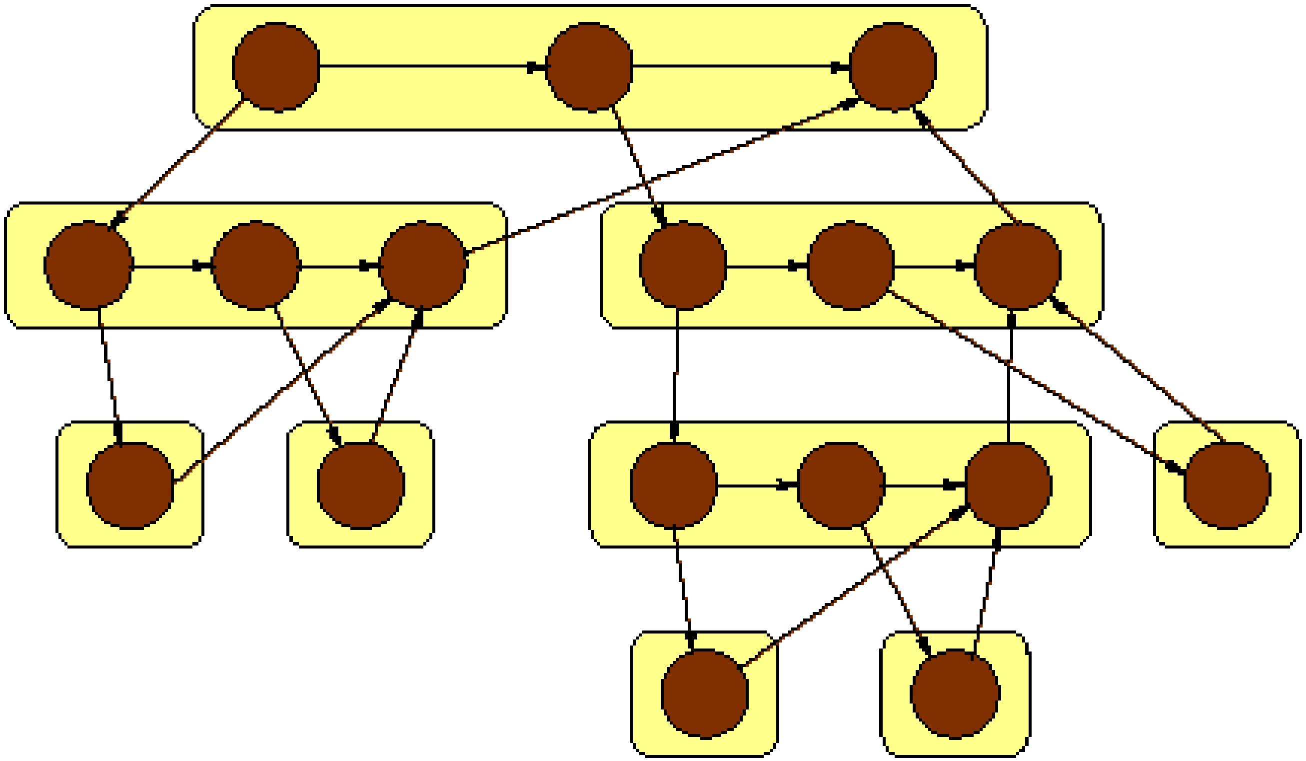




The computational difference

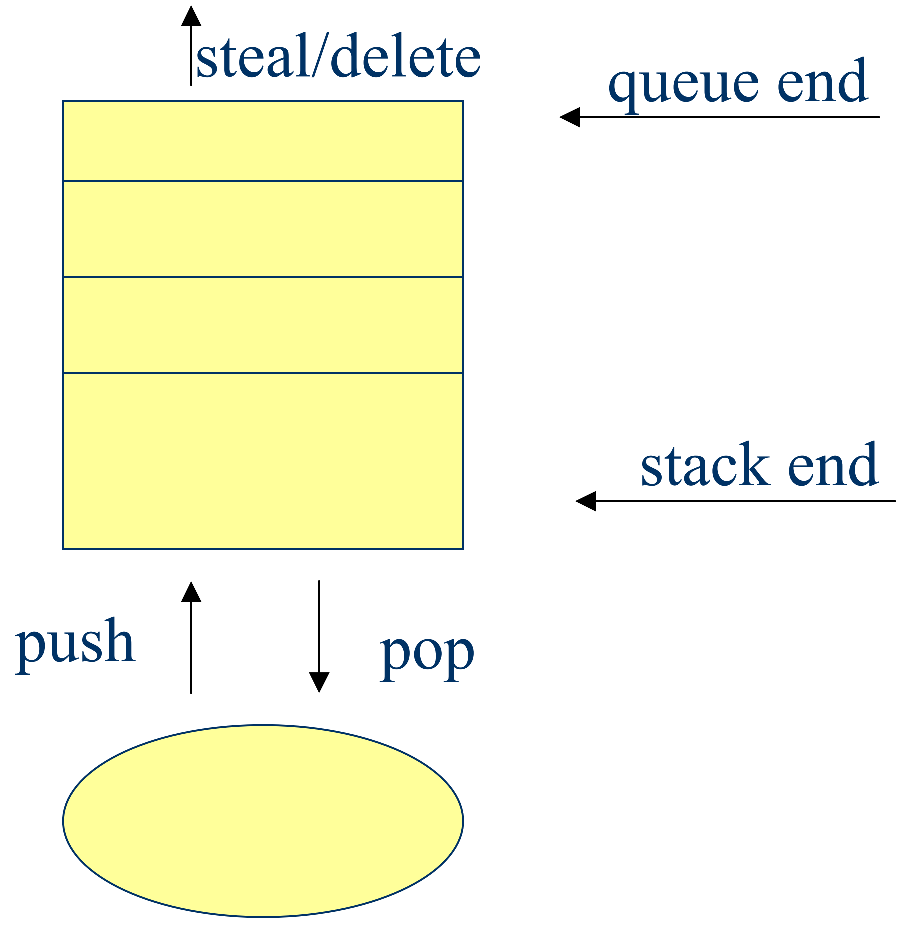


Cilk program



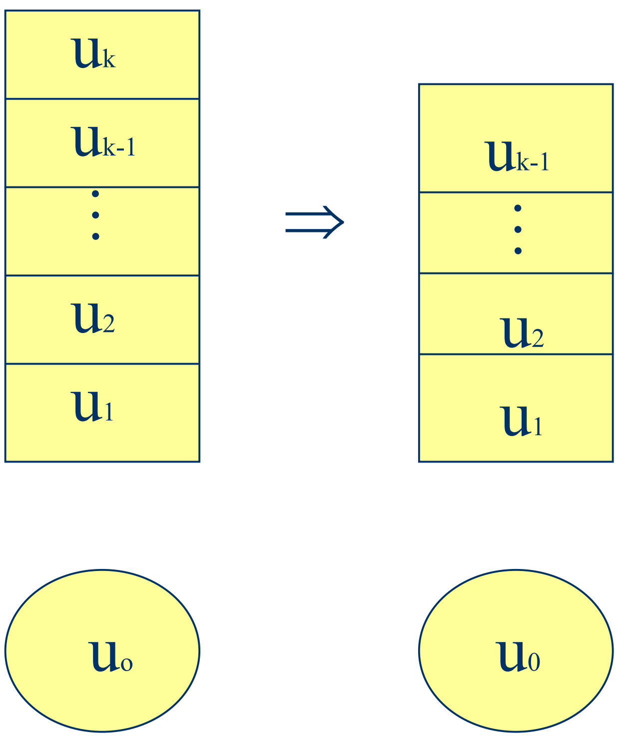
Main Data Structure

- Ready DEQUE

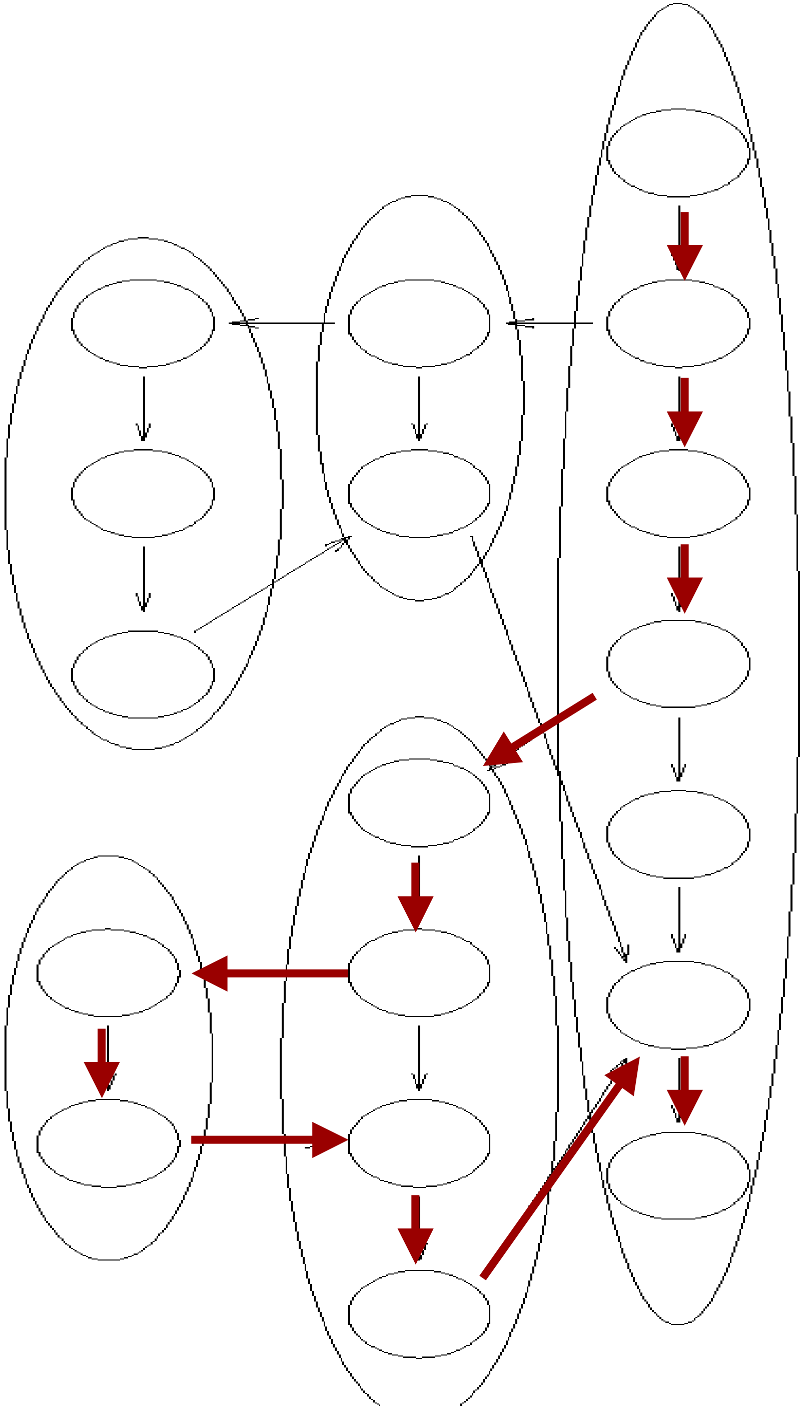


Work Stealing

- Distributed algorithm
- "Work-Stealing"



1. Choose a victim
2. Attempt to steal until successful
3. Steal oldest thread and begin working on it

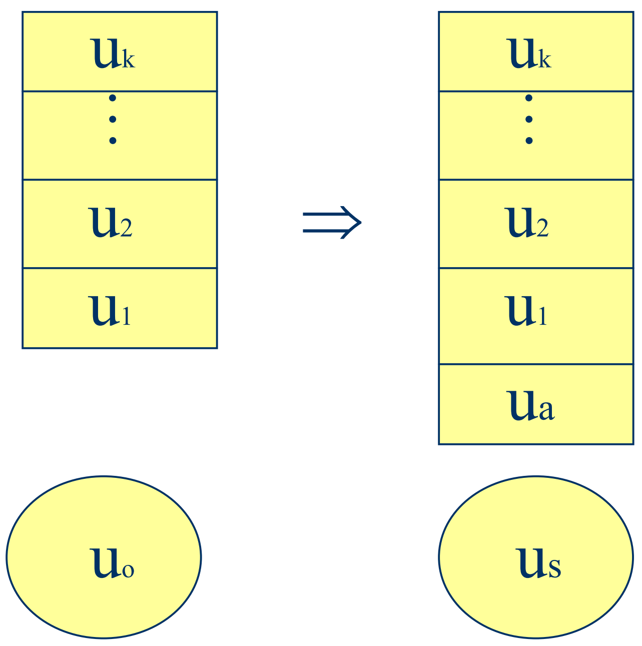


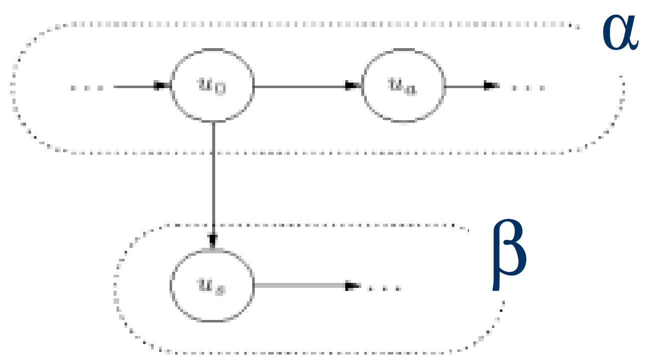
Work Stealing (2)

- But how do we decide which thread to we steal?
  - Closest to beginning of DAG, but…
  - Not necessarily the root

Cilk Scheduling Algorithm

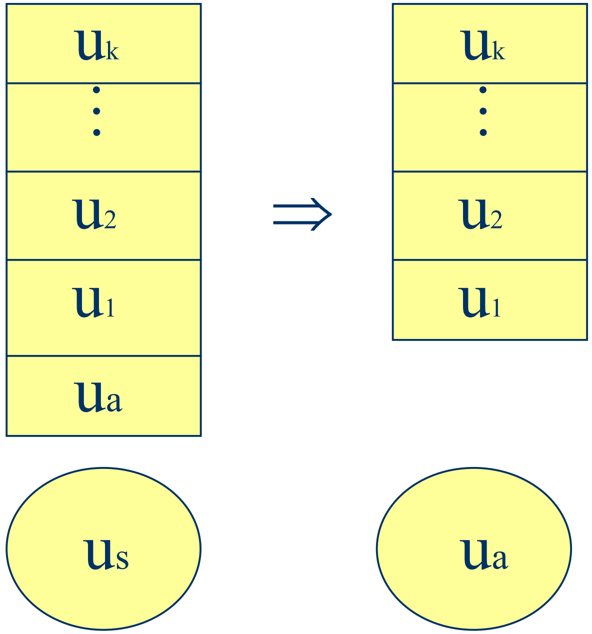
- A processor works on a thread until:
  - The thread spawns another thread





Cilk Scheduling Algorithm (2)

- A processor works on a thread until:
  - The thread returns/terminates



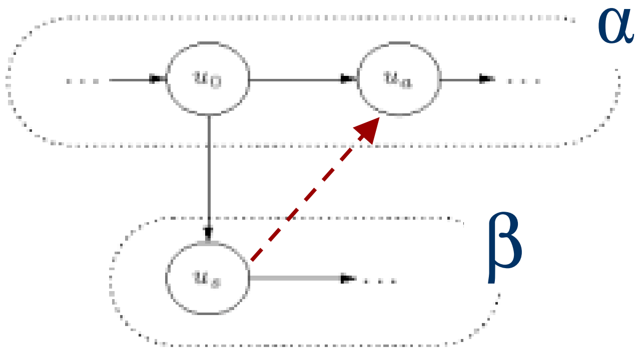
- If deque is nonempty, POP

- If deque is empty, try to execute thread's parent

- If thread's parent is busy, work-steal

Cilk Scheduling Algorithm (3)

- A processor works a thread until:
  - The thread reaches a sync point



If there exists outstanding children and the computation cannot proceed, then the processor worksteals

At a glance

1. Choose a victim
2. If its deque is empty, attempt to steal again
3. Otherwise, steal the top thread and execute it until:
    i. The thread spawns another thread
    ii. The thread returns/terminates
    iii. The thread reaches a sync point

Literature – *Maximum Utilization Schedule*

if i ready threads, $i < p$

then assign threads to i fastest processors

- Preemptive
- $O\left(\sqrt{p}\right)$ approximation algorithm

Generalization – *High Utilization Schedule*

1. **if** i ready threads, $i < p$

    then assign threads to i fastest processors

2. **if** $i \geq$ p

    then all processors work

if i ready threads, $i < p$

then the fastest **idle** processor is at most β times faster than the slowest **busy** processor

⇨ almost optimal; practical

Performance

- *Theorem 2:* any maximum utilization schedule has makespan

$$T_p \le \frac{W_1}{p\pi_{ave}} + \left(\frac{p-1}{p}\right)\frac{W_\infty}{\pi_{ave}}$$

work steals

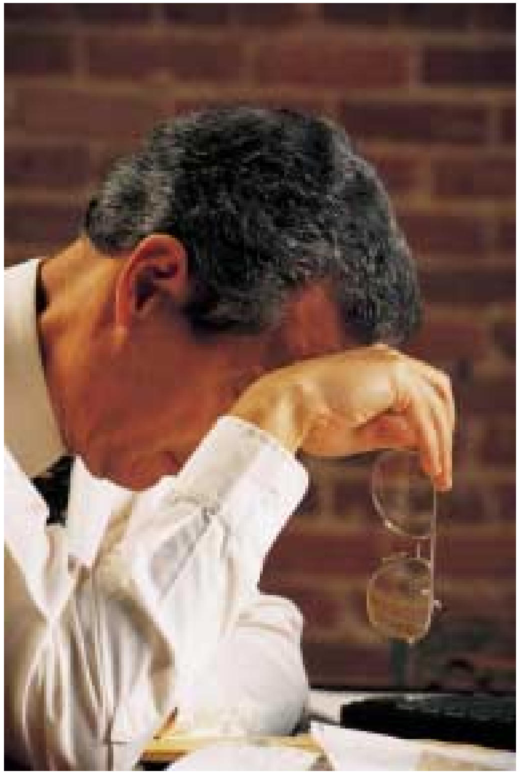
- *Theorem 4*: Any high utilization schedule has makespan

$$T_p \le \frac{W_1}{p\pi_{ave}} + \left(\frac{p-1}{p}\right)\frac{\beta W_\infty}{\pi_{ave}}$$

  - For parallel programs – almost optimal

If nothing makes sense…

- Work W_1
- Critical path length W_∞



Enhancing the Cilk Scheduler

- Migrations
  - Steals
  - Muggings
- Design assumptions:
  - each processor steals rate proportional to its speed
  - steals completed in time proportional to the speed of the processor
- Can manipulate times for steals and muggings for efficiency

This is where we started

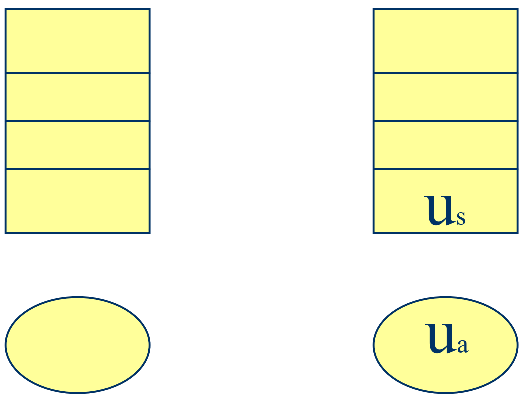
1. Choose a victim

2. If its deque is empty, attempt to steal again

3. Otherwise, steal the top thread and execute it until:

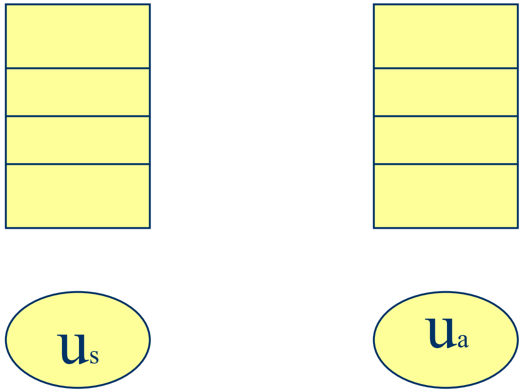
    i. The thread spawns another thread
    ii. The thread returns/terminates
    iii. The thread reaches a sync point

Enhanced Cilk Scheduler



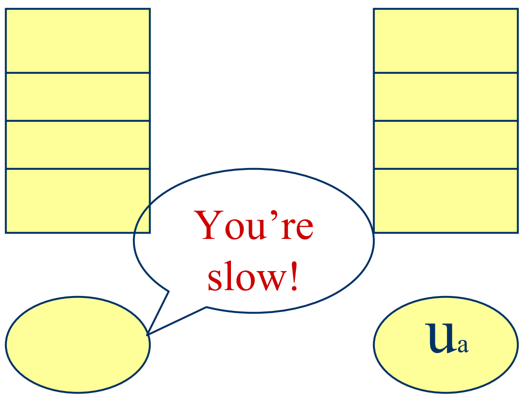
1. Choose a victim
2. If deque != empty then steal

Enhanced Cilk Scheduler



1. Choose a victim
2. If deque != empty then steal

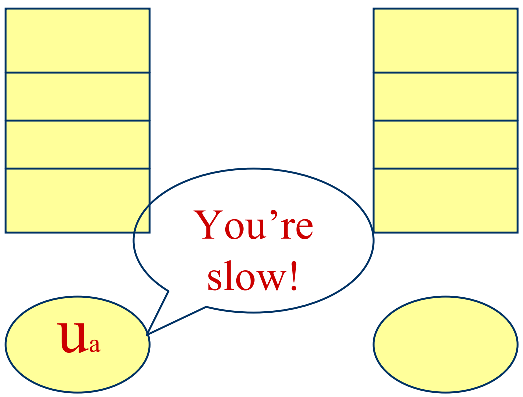
Enhanced Cilk Scheduler



1. Choose a victim
2. If deque != empty
   then steal

3. If deque = empty &
victim is working on a thread &
its speed is β times slower
then mug it and take the thread

Enhanced Cilk Scheduler



1. Choose a victim
2. If deque != empty
   then steal

3. If deque = empty &
   victim is working on a thread &
   its speed is β times slower
   then mug it and take the thread

Enhanced Cilk Scheduler

A processor works on a thread until:

i. The thread spawns another thread
ii. The thread returns/terminates
iii. The thread reaches a sync point
iv. The processor is mugged
    -- its thread is migrated to another processor
    -- this processor attempts to work steal

At a Glance

1. Choose a victim
2. If deque != empty, then steal
3. If deque = empty & victim is SLOW & working on a thread then mug it
4. If you got a thread, work on it until:
   1. The thread spawns another thread
   2. The thread returns/terminates
   3. The thread reaches a sync point
   4. The processor is mugged
5. ⇨ Otherwise, there is a failed steal attempt; try to steal again!

What's better…

- Less migrations $\Rightarrow$ cheaper!
- Ability to adjust efficiency
- Keep spirit of original algorithm

Discussed was…

- DAGs
- Cilk and programming in Cilk
- Ideal scheduler (greedy, busy, stealing)
- Work-stealing
- Standard and enhanced Cilk scheduler

Contributions

- New analysis to *maximum utilization scheduler*
  - prove a bound on the makespan and number of preemptions
  - Generalize algorithm and define the *high utilization scheduler*
- New algorithm for scheduling Cilk multithreaded parallel programs on heterogeneous processors

Questions?