Parallel & Concurrent Programming: Multiprogrammed Multiprocessors

Emery Berger

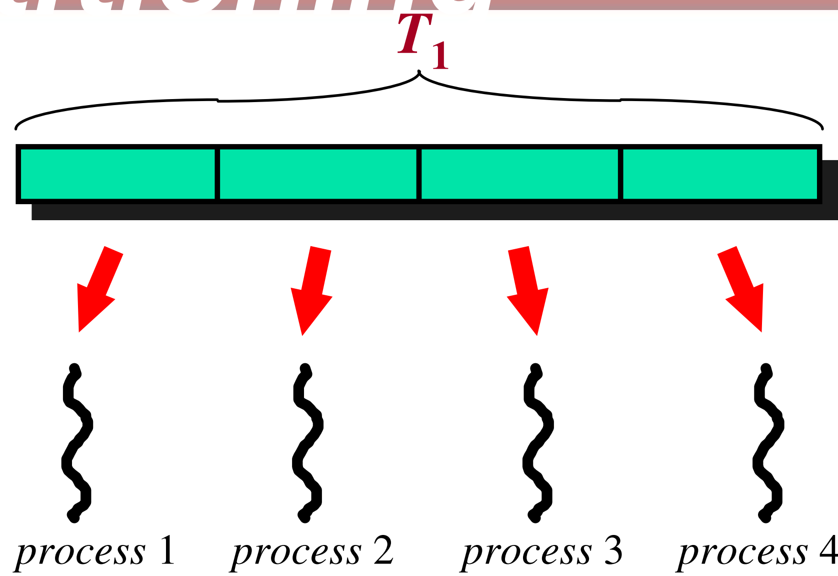
CMPSCI 691W
Spring 2006

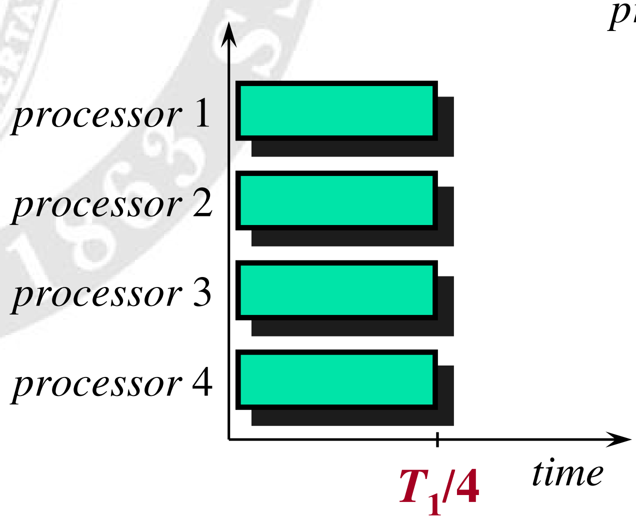
Outline

- Last time:
  - Parallel language taxonomy
  - Cilk parallel programming language
  - “Work-first” principle
- Today:
  - Multiprogrammed multiprocessors
  - “Hood” library

Static Partitioning

- Program partitions work T_1 evenly among P (light-weight) processes
  - a.k.a. kernel threads
- Each process performs T_1/P work

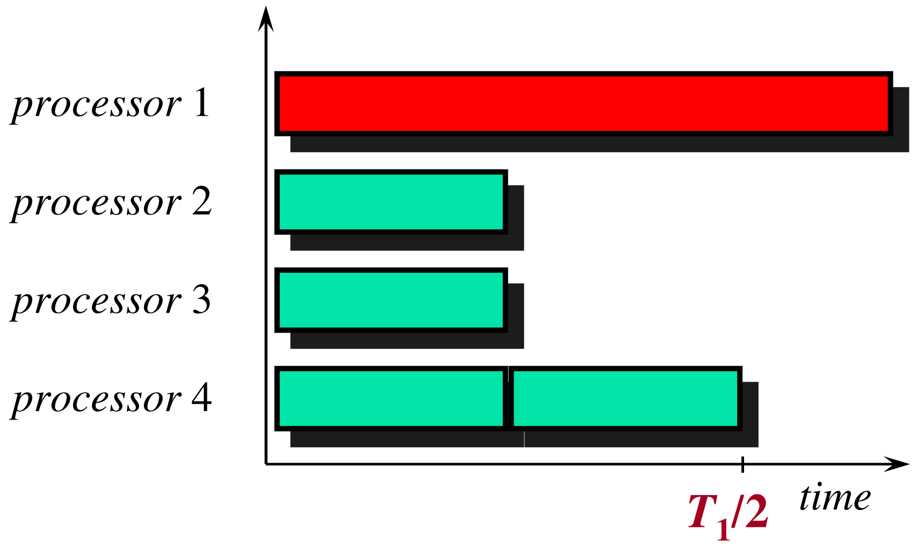




- At runtime, P processors execute P processes in parallel
  - Time = T_1/P
  - linear speedup

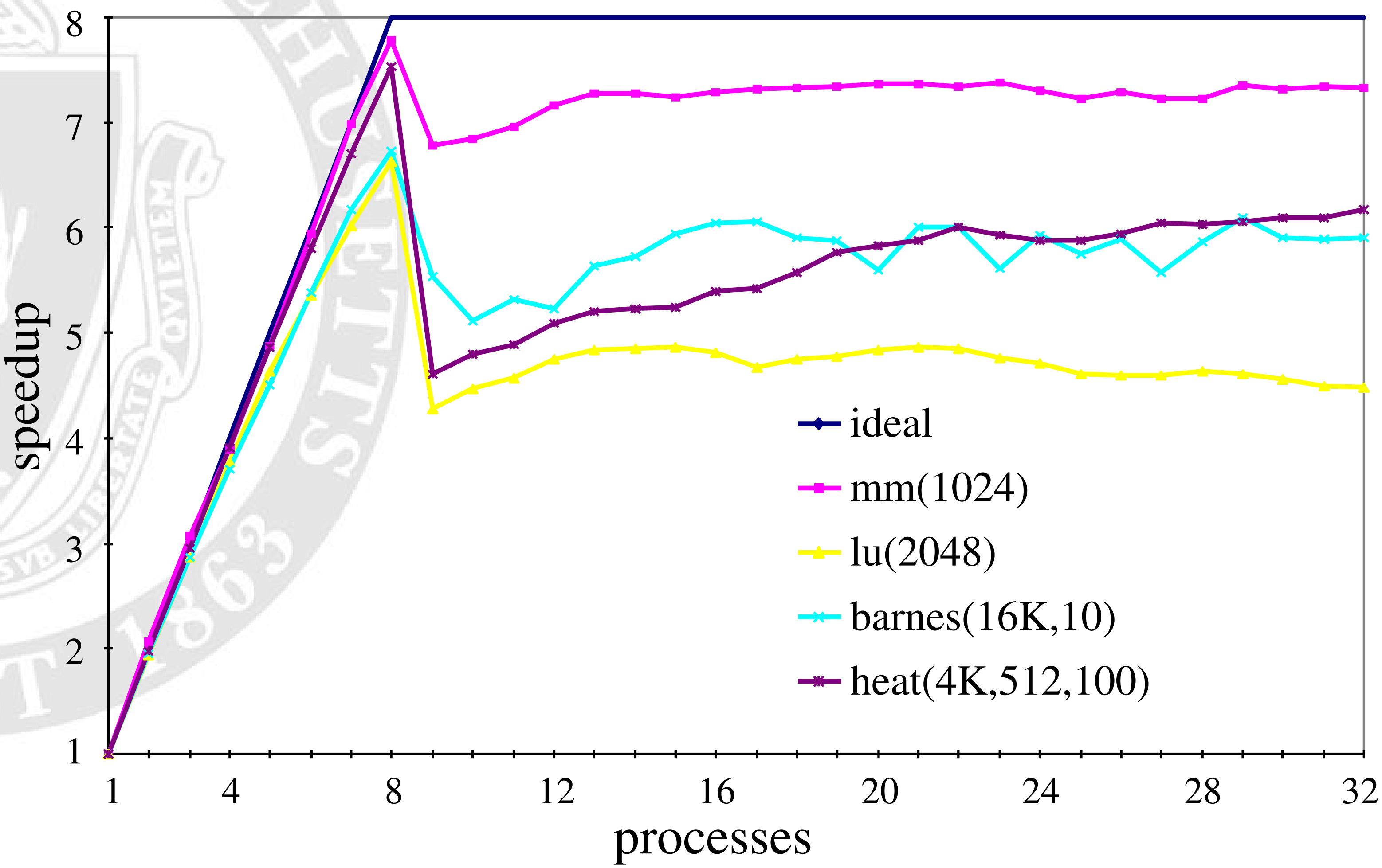
Multiprogramming

- If another program is running concurrently, **P** processes may execute on $\mathbf{P_A < P}$ processors



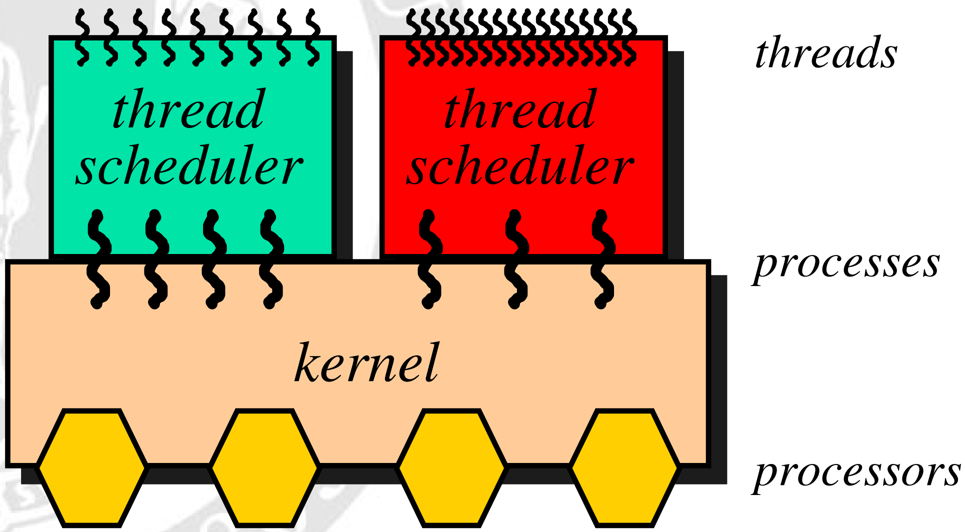
- Desired execution time = $\mathbf{T_1/P_A}$
  - Linear speedup
- Statically partitioned program may fall far short:
  - In this example, execution = $\mathbf{T_1/2}$, but $\mathbf{P_A = 3}$

Static Partitioning



Dynamic Scheduling

Program partitions work into (user-level) **threads** to expose all parallelism. Computation may create millions of threads, all dynamically scheduled through two levels



Each computation has a (user-level) thread scheduler that maps its threads to its processes

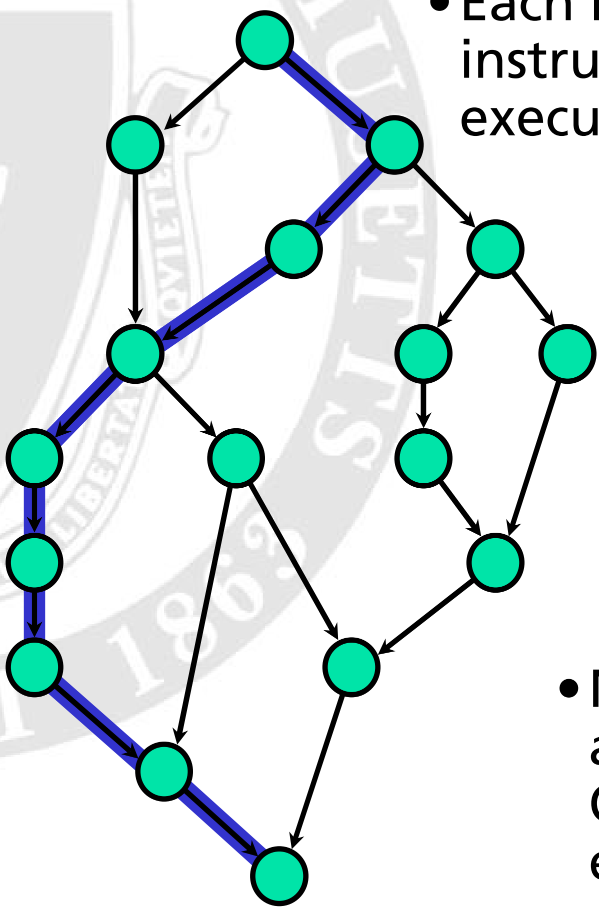
Kernel maps all processes to all processors

Define **processor average P_A** of computation as time-average number of processors on which computation executes, as determined by the kernel.

Goal: execution time $T \approx T_1/P_A$, **irrespective of kernel scheduling**.

Dag Model

Multithreaded computation modeled as **dag** (directed acyclic graph)



- Each node represents one executed instruction and takes one time unit to execute.
- Assume single source node and out-degree at most 2
- Work $\mathbf{T_1}$ = number of nodes. Critical-path length $\mathbf{T_\infty}$ = length of a longest (directed) path
- Node is **ready** if all of its ancestors have been executed. Only ready nodes can be executed.

Theory and Practice

Hood uses a **non-blocking work stealer** whose execution time T satisfies the following bounds:

T_∞ = **critical-path length**, theoretical minimum execution time with infinitely many processors

Theory: $\mathrm{E}[T] = O(T_1/P_A + T_\infty P/P_A)$.

- Kernel assumed to be adversary
- Bound optimal to within constant factor
- For any $\varepsilon > 0$, we have $T = O(T_1/P_A + (T_\infty + \lg(1/\varepsilon))P/P_A)$ with probability at least $1-\varepsilon$

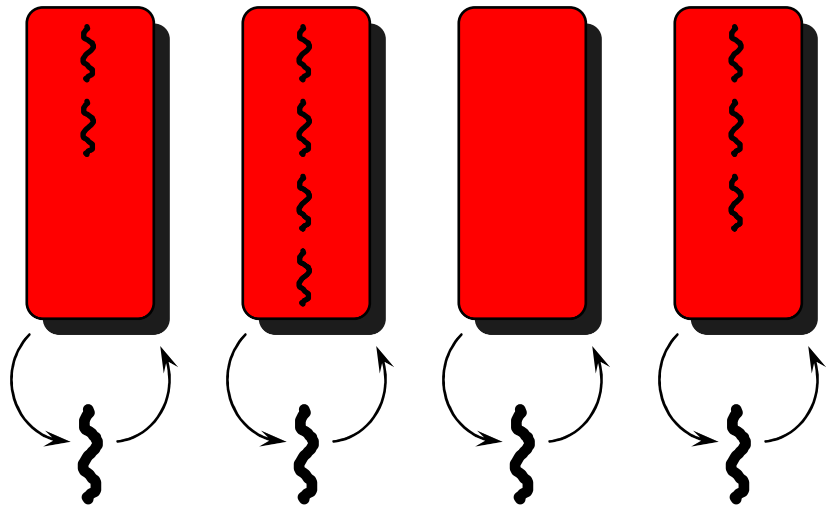
Practice: $T \approx T_1/P_A + T_\infty P/P_A$.

- We have $T \approx T_1/P_A$ whenever P is small relative to **average parallelism,** T_1/T_∞.

Work Stealing

Each process maintains “pool” of ready threads organized as a ***deque*** (double-ended queue) with a top and a bottom

Process obtains work by popping the bottom-most thread from its deque and executing that thread



- **If the thread blocks or terminates, then the process pops another thread.**
- **If the thread creates or enables another thread, then the process pushes one thread on the bottom of its deque and continues executing the other.**

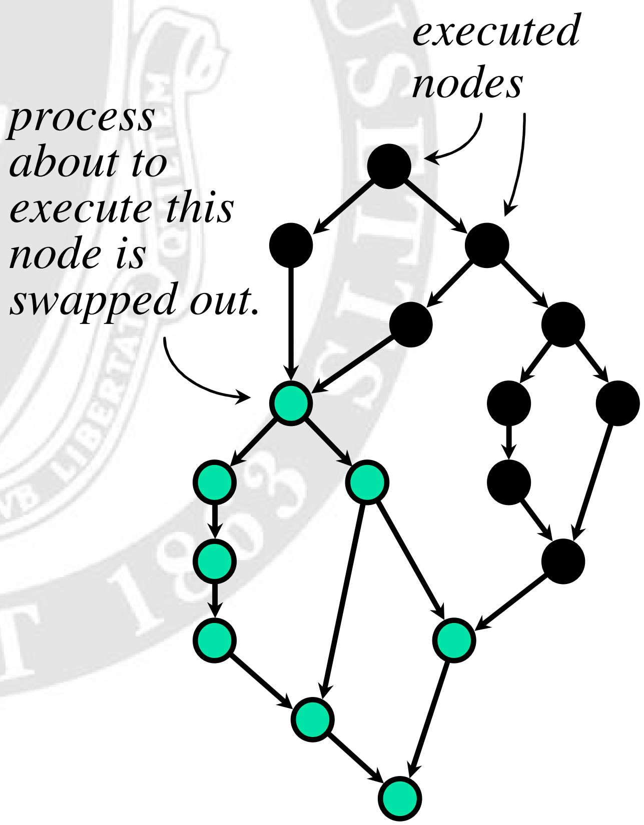
If a process finds that its deque is empty, then it becomes a ***thief*** and steals the top-most thread from the deque of a randomly chosen ***victim*** process.

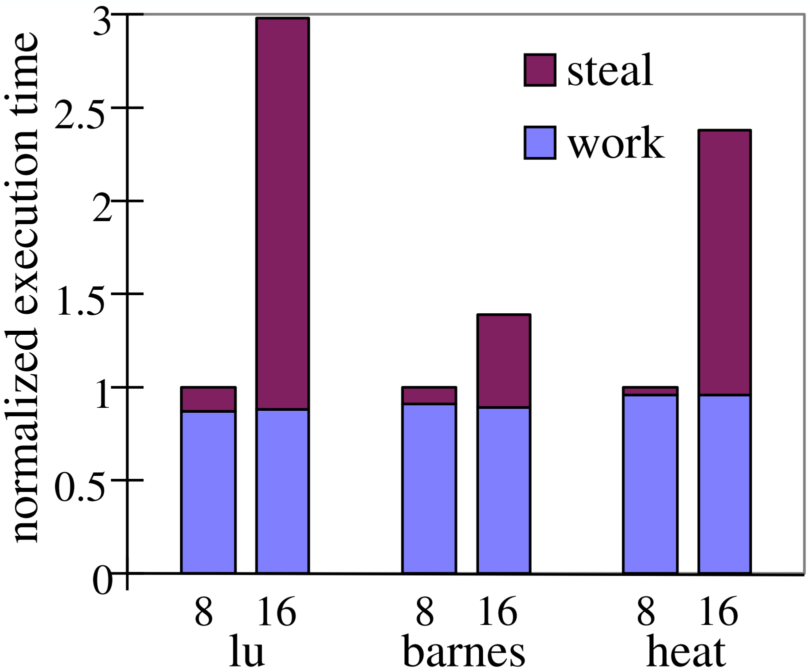
Non-Blocking Stealer

Implementation of work stealing with following features:

1. deques implemented with non-blocking synchronization
   - Instead of locks, atomic load-test-store machine instructions are used. Examples: **`load-linked/store-conditional`** and **`compare-and-swap`**.
   - There exists constant c (≈ 10) such that if process performs a deque operation, then after executing c instructions, *some* process has succeeded in performing deque operation
2. Each process, between consecutive steal attempts, performs a **`yield`** system call

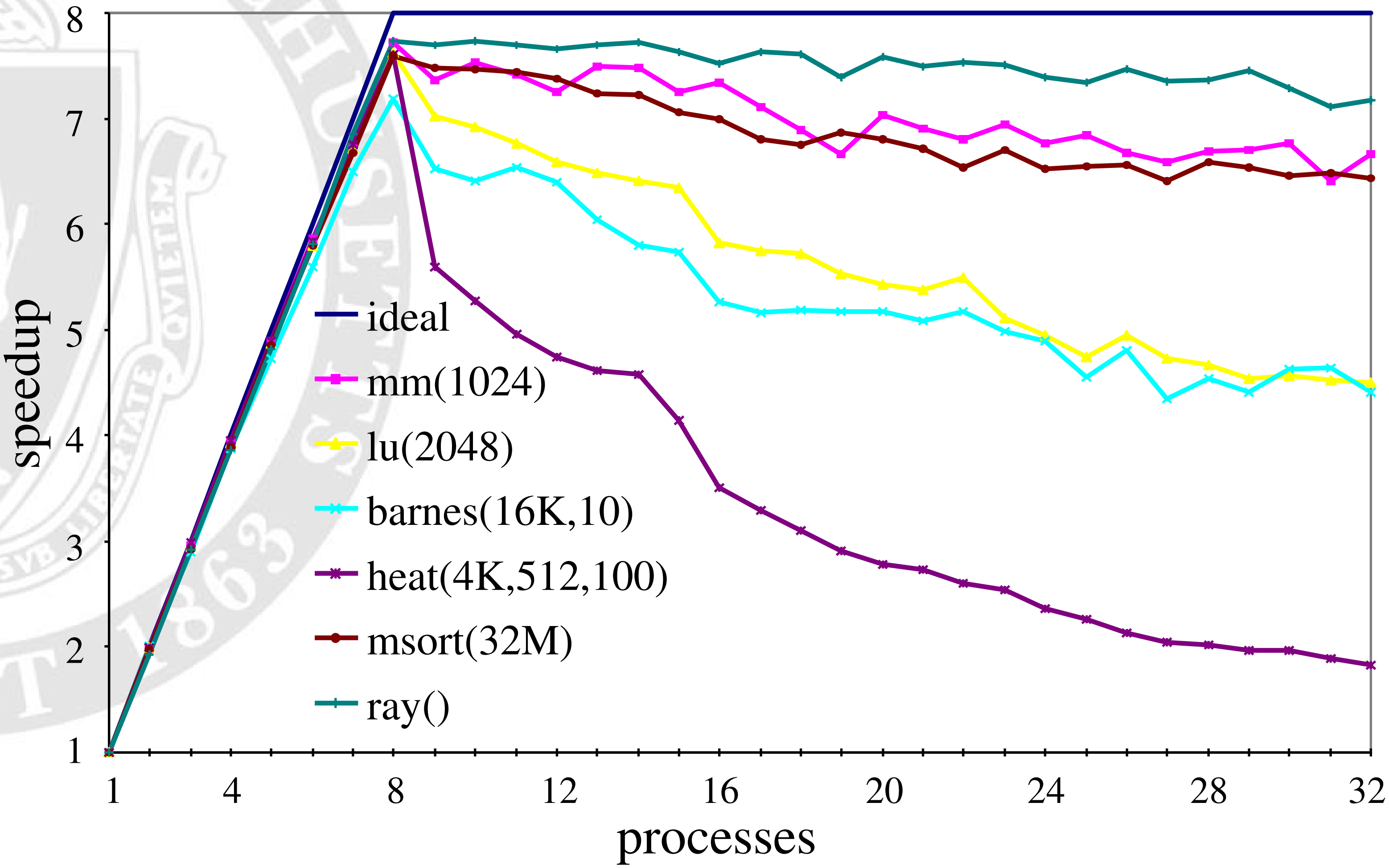
Why Yield?





Processes spin making steal attempts, but all deques empty

Performance w/o Yield



Lower Bounds

At each time step $i = 1, 2, \ldots, T$, the kernel chooses to ***schedule*** any subset of the P processes, and those scheduled processes execute one instruction. Let p_i denote the number of processes scheduled at step i.

Processor average defined by $P_A = \frac{1}{T} \sum_{i=1}^{T} p_i$

Execution time given by $T = \frac{1}{P_A} \sum_{i=1}^{T} p_i$

- $T \geq T_1/P_A$, because $\sum_{i=1}^{T} p_i \geq T_1$.
- $T \geq T_\infty P/P_A$, because kernel can force $\sum_{i=1}^{T} p_i \geq T_\infty P$.

  There must be at least T_∞ steps i with $p_i \neq 0$, and for each such step, the kernel can schedule $p_i = P$ processes.

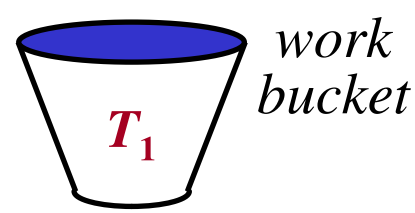
Greedy Schedules

A schedule is ***greedy*** if at each step i, the number of nodes executed is equal to the minimum of p_i and the number of ready nodes.

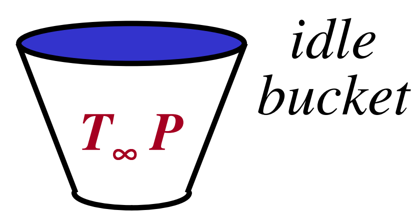
Theorem: Any greedy schedule has length at most $T_1/P_A + T_\infty P/P_A$.

Proof: We prove that $\sum_{i=1}^{T} p_i \le T_1 + T_\infty P$. At each step each scheduled process pays one token.

- If the process executes a node, then it places a token in the ***work bucket***. Execution ends with T_1 tokens in the work bucket.



- Otherwise, the process places a token in the ***idle bucket***. There are at most T_∞ steps at which a process places a token in the idle bucket, and at each such step at most P tokens are placed in the idle bucket. ■



Analysis

Theorem: The non-blocking work stealer runs in expected time $O(T_1/P_A + T_\infty P/P_A)$.

Proof sketch: Let S denote the number of steal attempts. We prove that $\sum_{i=1}^{T} p_i = O(T_1 + S)$ and $\mathbf{E}[S] = O(T_\infty P)$. At each step each scheduled process pays one token.

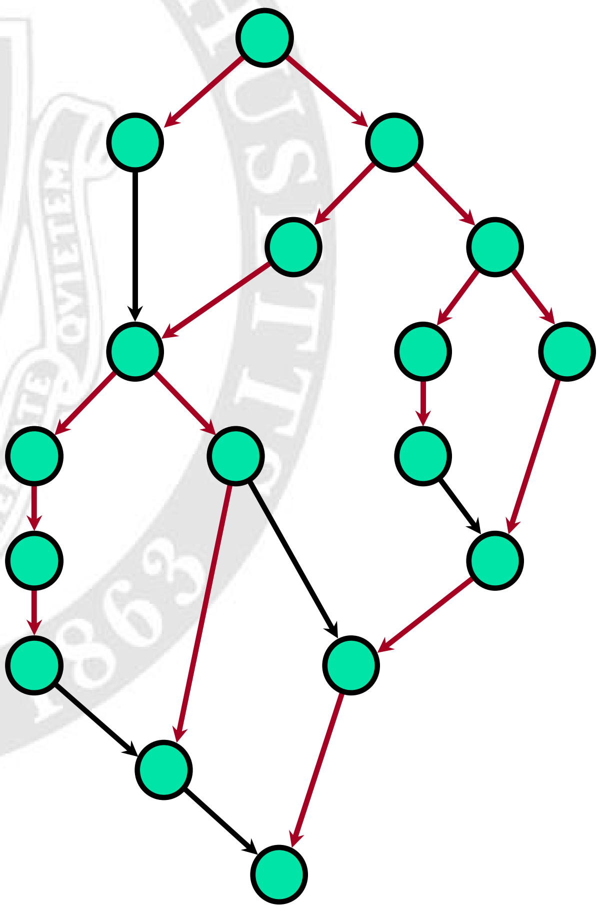
- If the process is "working," then it places a token in the ***work bucket***. Execution ends with $O(T_1)$ tokens in the work bucket.

work bucket

- Otherwise, the process places a token in the ***steal bucket***. Execution ends with $O(S)$ tokens in the steal bucket.

steal bucket

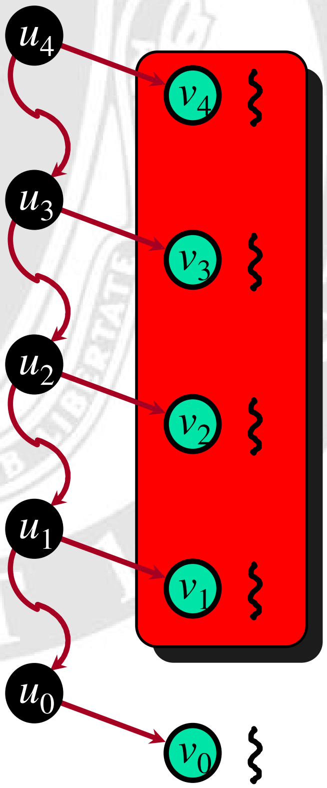
Enabling Tree



- An edge **(u,v)** is an **enabling edge** if the execution of **u** made **v** ready. Node **u** is the **designated parent** of **v**.
- The enabling edges form an **enabling tree**.

Structural Lemma

For any deque, at all times during the execution of the work-stealing algorithm, the designated parents of the nodes in the deque lie on a root-to-leaf path in the enabling tree.



Consider any process at any time during the execution.

- v_0 is the ready node of the thread that is being executed.
- $v_1, v_2, \ldots, v_k$ are the ready nodes of the threads in the process's deque ordered from bottom to top.
- For $i = 0, 1, \ldots, k$, node u_i is the designated parent of v_i.

Then for $i = 1, 2, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree.

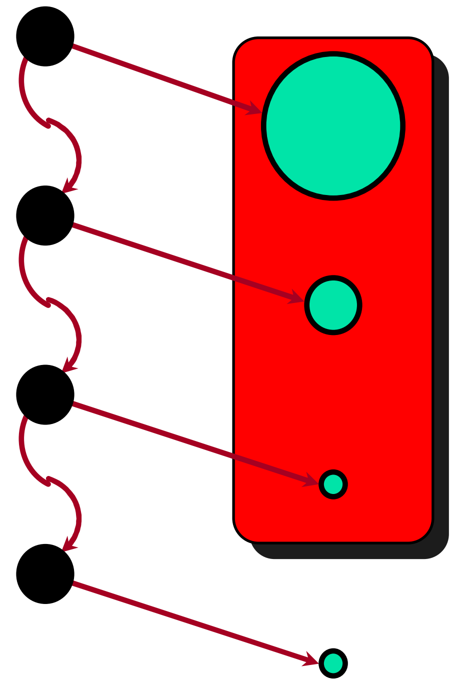
Steal Attempts

We use a potential function to bound the number of steal attempts.

At each step i, each ready node u has potential $\phi_i(u) = 3^{T_\infty - d(u)}$, where $d(u)$ is the depth of u in the enabling tree.

The potential Φ_i at step i is the sum of all ready node potentials.

- *The deques are top-heavy:* the top-most node contributes a constant fraction.
- With constant probability, P steal attempts cause the potential to decrease by a constant fraction.
- The initial potential is $\Phi_0 = 3^{T_\infty}$, and it never increases.
- The expected number of steal attempts until the potential decreases to 0 is $O(T_\infty P)$. ■



Performance Model

Execution time: $T \le c_1 T_1 / P_A + c_2 T_\infty P / P_A$.

Utilization: $$\frac{T_1}{P_A T} \ge \frac{T_1}{c_1 T_1 + c_2 T_\infty P} \ge \frac{1}{c_1 + c_2 P / (T_1 / T_\infty)}$$

The ratio $P/(T_1/T_\infty)$ is the ***normalized number of processes***.

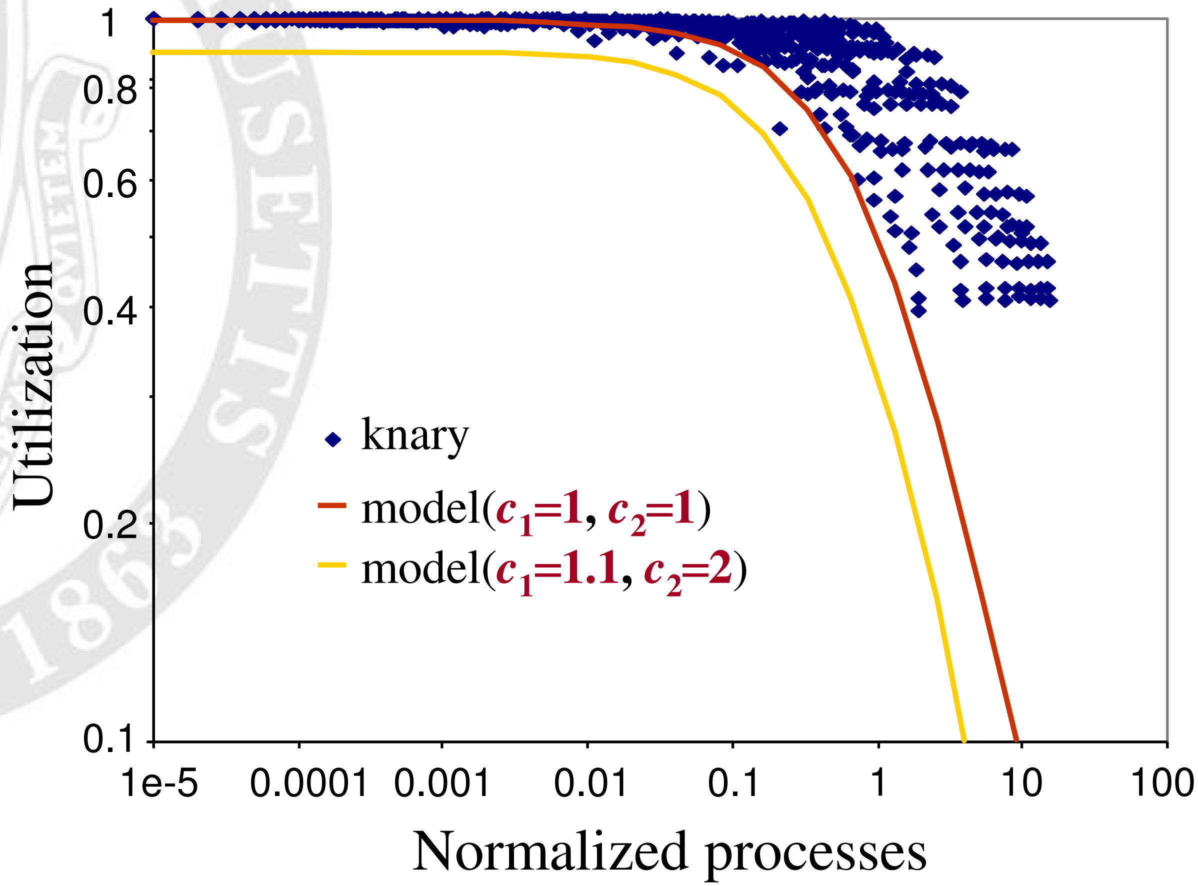
For all multithreaded applications and all input problems, the utilization can be lower bounded as a function of one number, the normalized number of processes.

We test this claim with a synthetic application, **`knary`**, that produces a wide range of work and critical-path lengths for different inputs.

Knary Utilization

Utilization measured on 8-processor Sun Ultra Enterprise 5000.

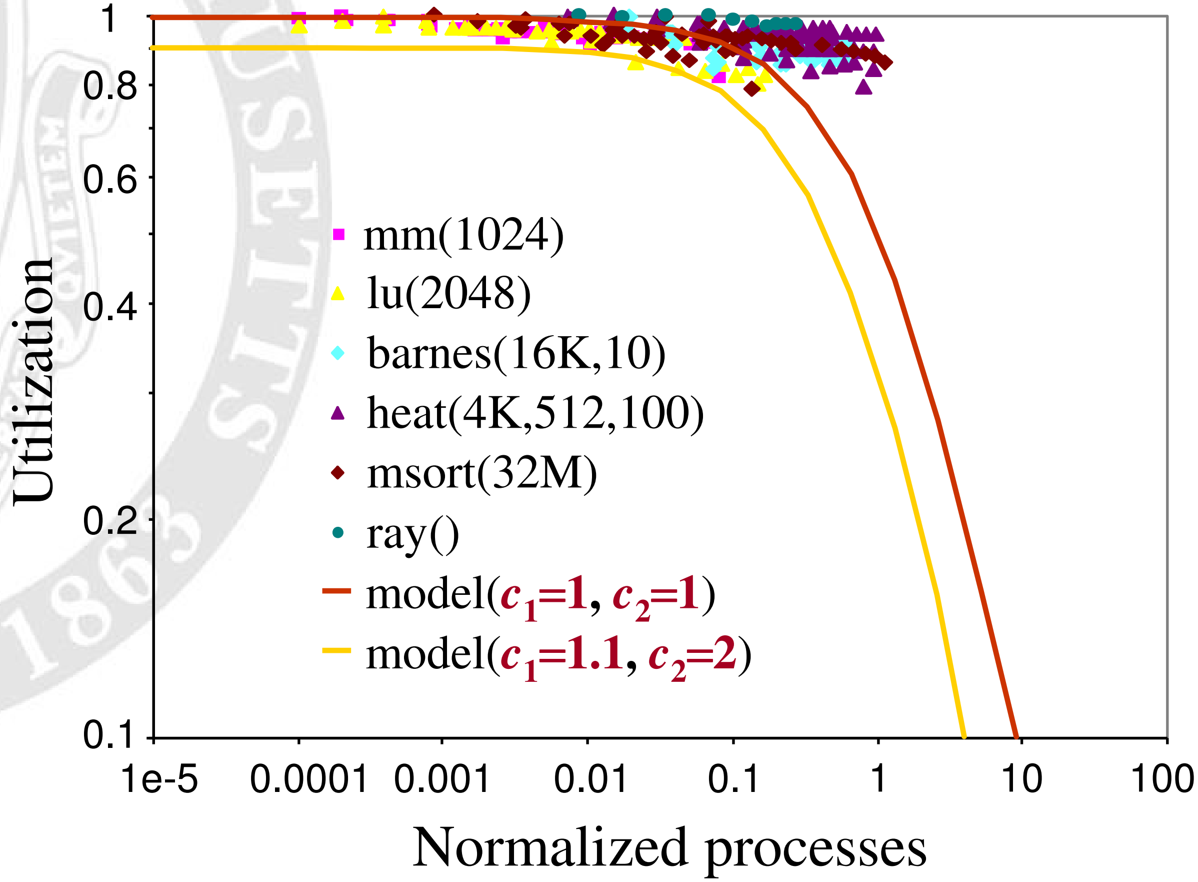
No other program is running, so $P_A = \min\{8, P\}$.



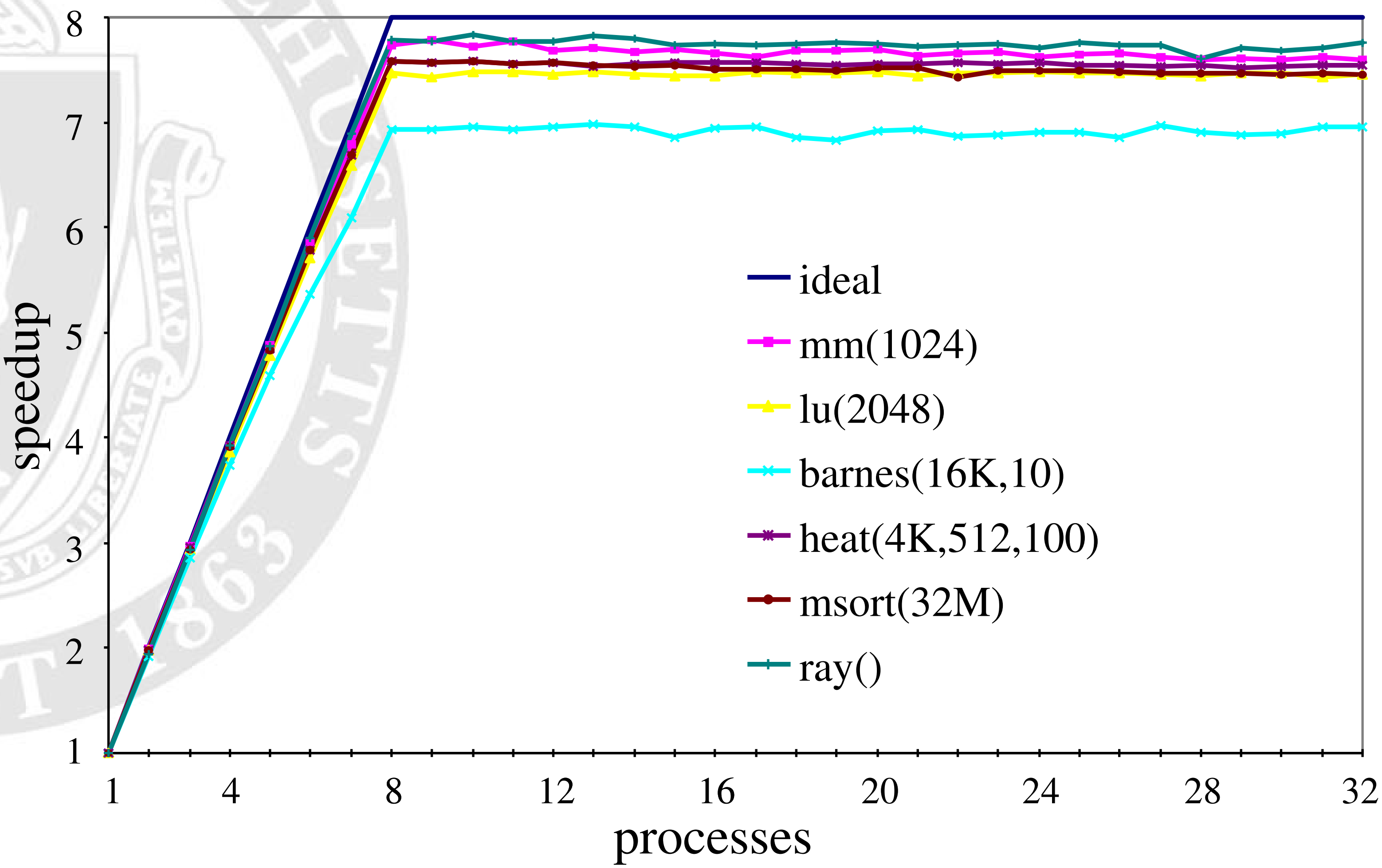
Application Utilization

Utilization measured on 8-processor Sun Ultra Enterprise 5000.

No other program is running, so $P_A = \min\{8, P\}$.



Hood Performance



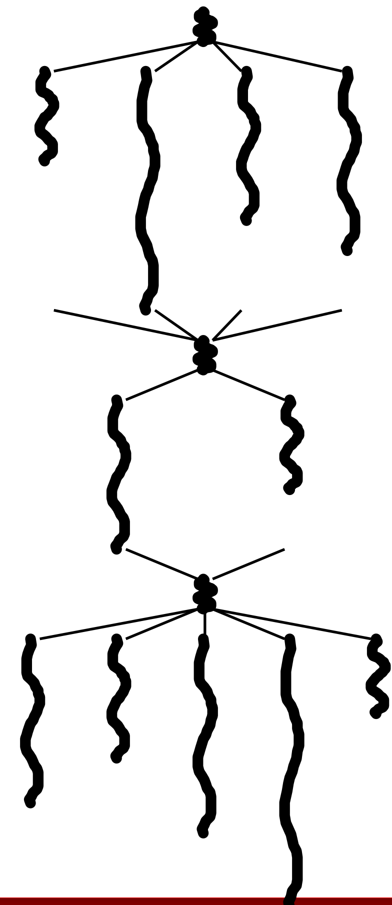
Varying # Processors

To test the model when the number of processors varies over time, we run the test applications concurrently with a synthetic application, **`cycler`**.

Repeatedly, **`cycler`** creates a random number of processes, each of which runs for a random amount of time.

- Each process repeatedly increments a shared counter.
- At regular intervals, the counter value and a timestamp are written to a buffer.

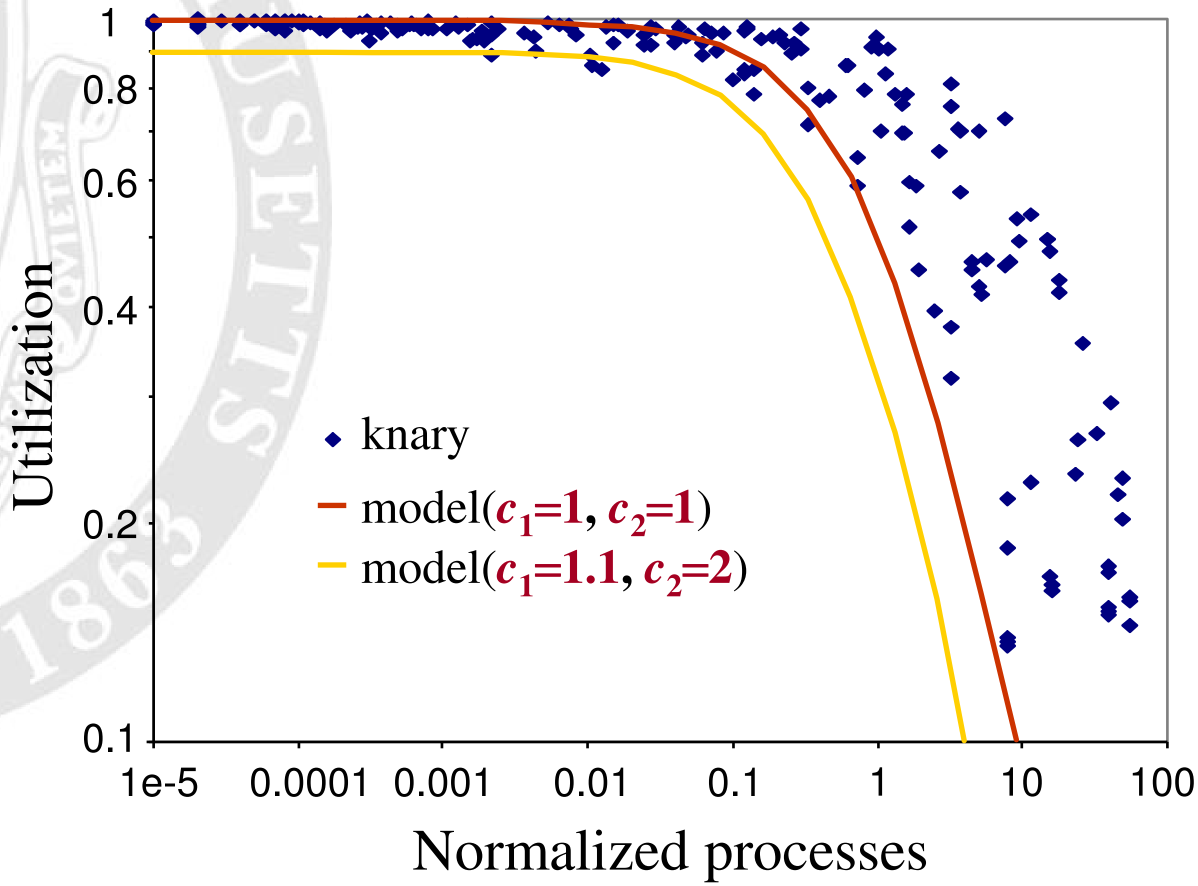
For any time interval, we can look at the counter values at the start and end to determine the processor average P_A(**`cycler`**) for **`cycler`** over that interval.



Knary Utilization

Utilization measured on 8-processor Sun Ultra Enterprise 5000.

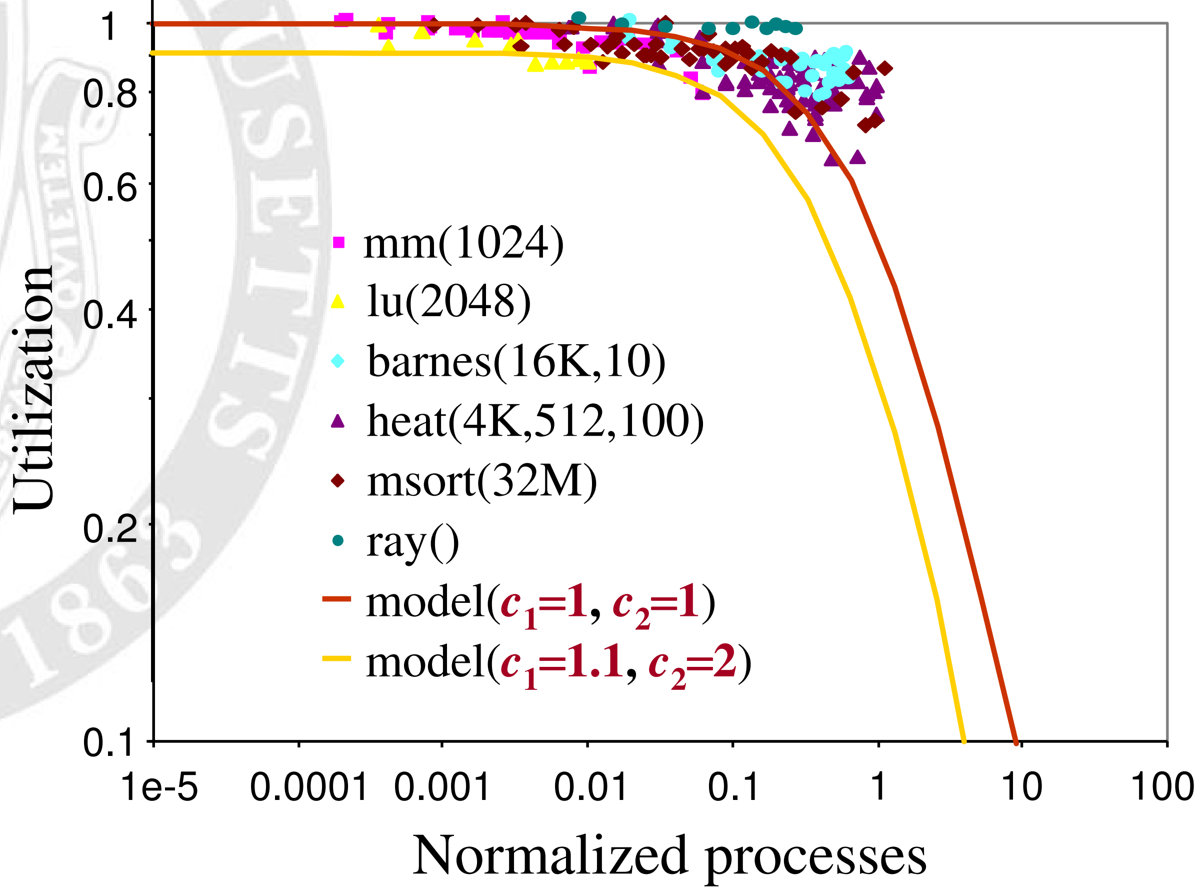
`Cycler` is also running, so $P_A = \min\{8 - P_A(\texttt{cycler}), P\}$.



Application Utilization

Utilization measured on 8-processor Sun Ultra Enterprise 5000.

`Cycler` is also running, so $P_A = \min\{8 - P_A(\texttt{cycler}), P\}$.

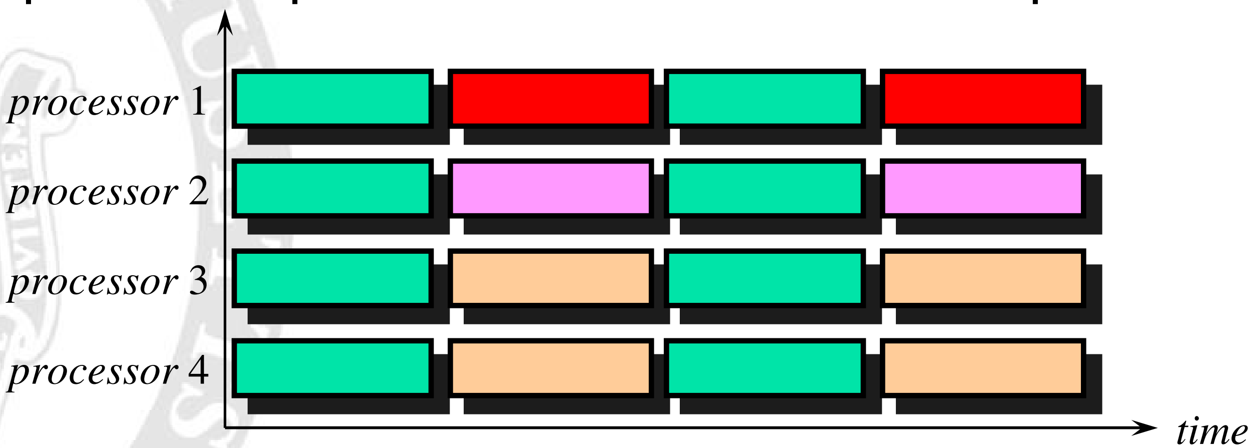


Summary

- Non-blocking work stealer provides predictable, good performance on commodity OS
- Related work (OS side):
  - **coscheduling**
  - **process control**

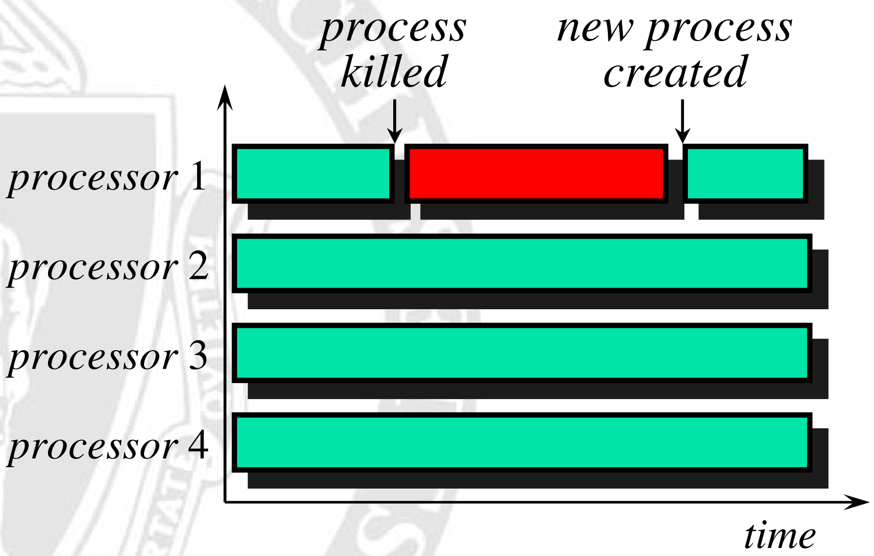
Coscheduling

Coscheduling (gang scheduling) –
all computation's processes scheduled to run in parallel



- ☹ For some computation mixes, coscheduling not effective. Example: Computation with 4 processes and computation with 1 process on a 4-processor machine
- ☺ Resource-intensive may require coscheduling for high performance. Example: Data-parallel programs with large working sets

Process Control



With **process control**, each computation creates and kills processes dynamically: always runs with number of processes equal to number of processors assigned to it.

Process control & non-blocking work stealer complement each other

- With work stealing, new process can be created at any time, and process can be killed when its deque is empty
- With non-blocking work stealer, little penalty for operating with more processes than processors
- Process control can help keep P close to P_A.

The End

- Next week: Spring Break
- Week after that: travel
  - **Plenty** of time to work on homework (due 29th) and…
  - **Project report**: describe your proposed work and implementation plan, including division of responsibilities if appropriate, and timeline with milestones.