Scheduling Multithreaded Applications by Work Stealing

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

The Challenge

- **Efficiently execute a dynamic, multithreaded computation on a MIMD computer**
  - —parallelism not known a priori
    - – dynamically grows and shrinks as computation unfolds
    - – ill-suited to static scheduling
  - —threads depend upon one another
- **Scheduler goals**
  - —ensure that an appropriate # of threads are active at each step
    - – enough to keep all processors busy
  - —bound memory footprint of active threads
  - —minimize interprocessor communication
    - – keep related threads on same processor

Two Scheduling Paradigms

- **Work sharing**
  - —migrate new threads to other processors that might be underutilized
- **Work stealing**
  - —underutilized processors attempt to steal work from others

Intuition: thread migration is less frequent with work stealing

Why? When all processors have work

work sharing migrates threads

work stealing does not

Work Stealing has a Rich History

- **Use of work stealing**

  1981 Burton and Sleep: execution of functional programs on a virtual tree of processors

  1984 Halstead: Multilisp

- **Analysis of work stealing**

  1991 Rudolph, Slivkin-Allalouf, and Upfal: randomized work stealing for load balancing independent jobs

  1993 Karp and Zhang: randomized work stealing for backtracking search

  1994 Zhang and Ortynski: bounds on communication requirements for work stealing of backtracking search

Focus of This Work

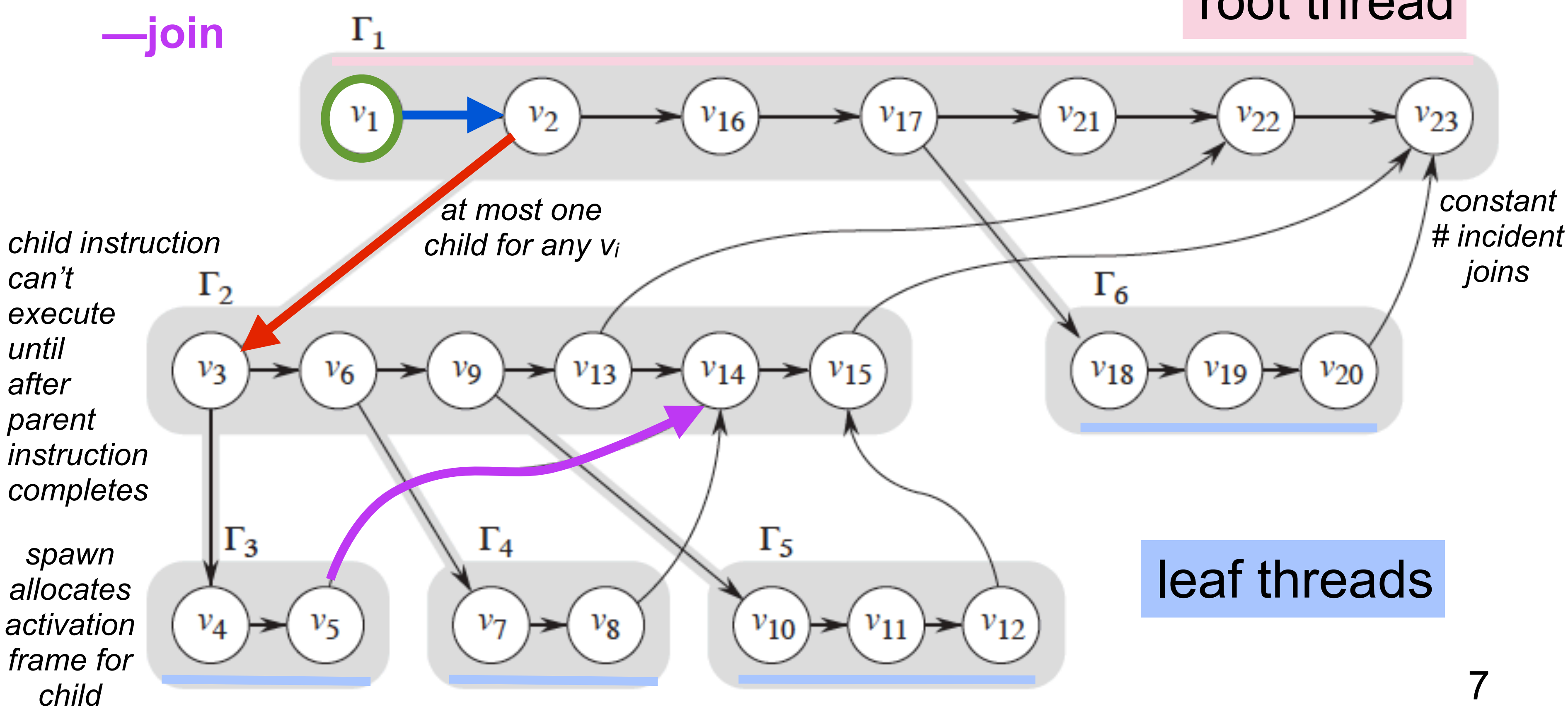
- **Fully-strict, multithreaded computations**
  - **—includes backtrack search, divide & conquer, data flow**
- **Randomized work stealing**
- **Analysis of space, time, and communication of computations scheduled using randomized work stealing**

Topics

- Graph model of multithreaded computations
- Simple scheduling algorithm using a central queue
- Work stealing scheduler based on “busy leaves” algorithm
- Atomic access model to analyze execution time and communication costs

A DAG Model for Multithreaded Computation

- Thread: sequential ordering of unit-time instructions
- Dependency edges: partial ordering on instructions
  - —continue
  - —spawn edges form a spawn tree
  - —join



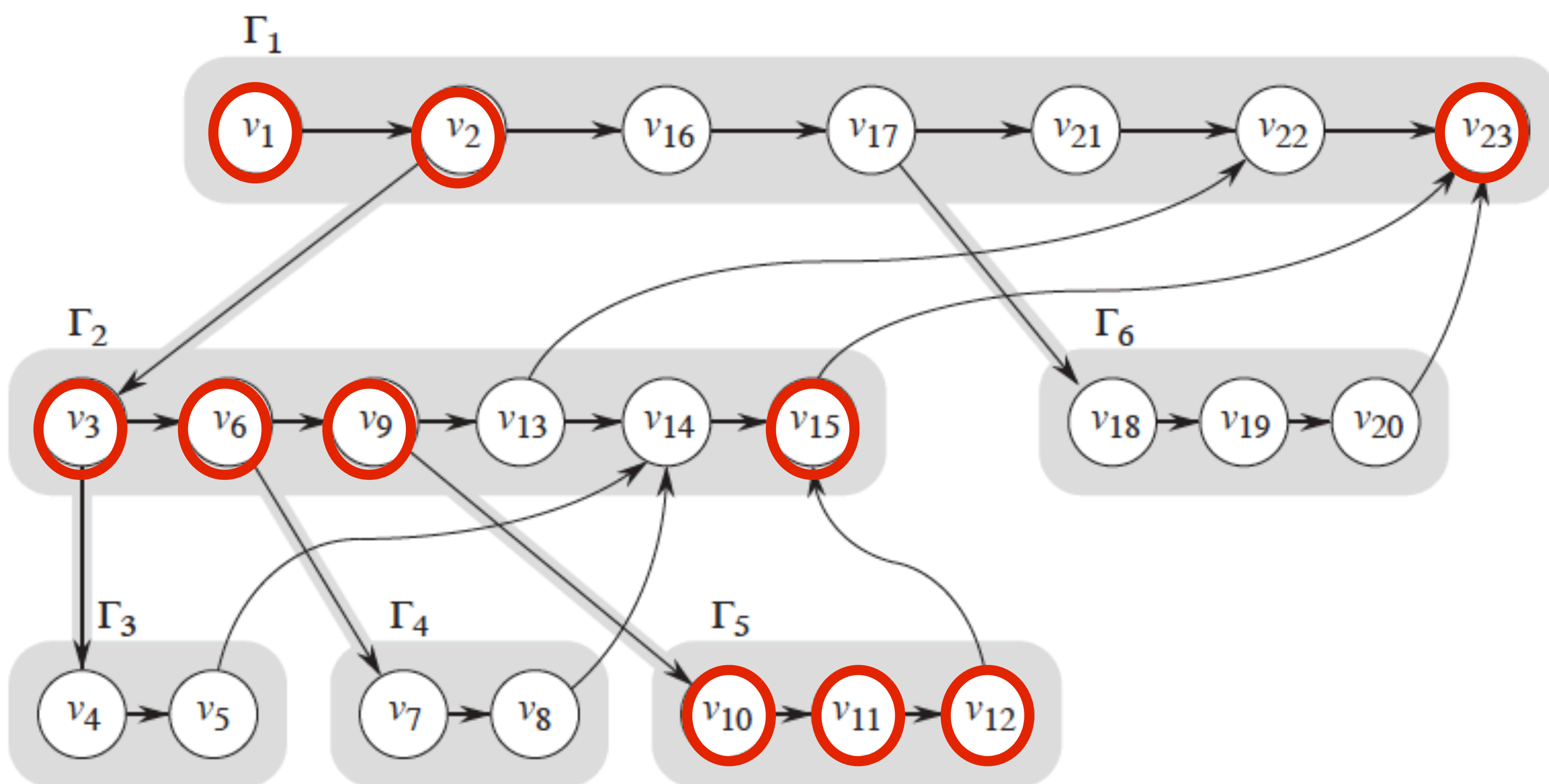
DAG Model Notes

- An instruction is ready if all its predecessors have executed
- A parent thread is alive until all its children die
  - —its activation frame has the same lifetime
- Classes of computations
  - —deterministic: computation for an input is schedule independent
  - —non-deterministic: one input can yield two or more computations
- Strict computation
  - —all join edges from a thread go to an ancestor in the spawn tree
- Fully-strict computation
  - —all join edges from a thread go to its parent in the spawn tree

Claim: any multithreaded computation that can be executed depth first can be made strict or fully strict without changing the semantics

DAG Performance Measures

- T1 (work) = total number of instructions = 23
- T_∞ (critical path length) = longest path in DAG = 10



Greedy Scheduling

- Types of schedule steps
  - —complete step
    - at least P threads ready to run
    - select any P and run them
  - —incomplete step
    - strictly < P threads ready to run
    - greedy scheduler runs them all
- Theorem: On P processors, a greedy scheduler executes any computation G with work T_1 and critical path of length T_∞ in time $T_p \le T_1/P + T_\infty$
- Proof sketch
  - —only two types of scheduler steps: complete, incomplete
  - —cannot be more than T_1/P complete steps, else work > T_1
  - —every incomplete step reduces remaining critical path length by 1
    - no more than T_∞ incomplete steps

Efficient Greedy Schedules

- Execution time for greedy schedule = $T_p \leq T_1/P + T_\infty$
- Interested in schedules that achieve linear speedup, $O(T_1/P)$
- Linear speedup occurs when $T_1/P \gg T_\infty$, i.e., $T_1/T_\infty = \Omega(P)$
  - —namely, T_1/T_∞ is bounded from below by P
  - —“parallel slackness”

Space for a Multithreaded Computation

- Stack depth of a thread
  —sum of the sizes of all its ancestors, including itself
- S_1 = minimum possible space for a 1-processor execution
  —stack depth of the execution
- Let S(X) be space for P-processor execution of schedule X of a multithreaded computation
- Interested in execution schedules that exhibit at most a linear expansion of space, i.e., $S(X) = O(S_1 P)$

Busy Leaves

- **In a strict computation**
  - —once a thread Γ as been spawned
  - —a single processor can complete the computation rooted at Γ
    - – even if no other thread makes any progress
- **Corollary**
  - —there is always one ready thread in computation Γ that is ready
- **No leaf thread in a strict multithreaded computation can stall**
  - —enables a multithreaded computation to keep leaves busy
- **Greedy schedule + busy leaves = schedules that achieve linear speedup and linear expansion of space**

Busy Leaves Algorithm

- On line algorithm: executes each step w/o knowledge of future
- Maintain all live threads in a single pool, available to all P
  - —spawn: add new thread to pool
  - —work step: remove ready thread from pool
- Start with root thread in global pool, all processors idle
- At beginning of each step, each processor idle or has work
- Each idle thread attempts to remove a ready thread from pool
  - —if enough threads in pool, each processor gets one
- Each processor with a thread executes next instruction in thread until spawn, stall, or die

Busy Leaves Algorithm

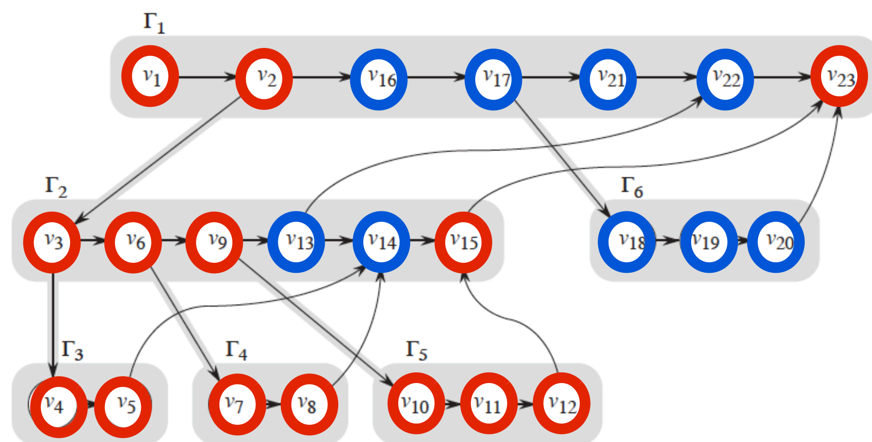
- **Spawn: if a thread spawns a child in a step**
  - —finish step by returning parent thread to pool
  - —begin next step working on child thread
- **Stall: if a thread stalls**
  - —finish step by returning current thread to pool
  - —begin next step idle
- **Die: If a thread dies in a step**
  - —finish step by checking if parent thread has any living children
  - —if parent has no living children, begin next step executing parent
  - —else begin next step idle

Busy Leaves Example

Two processor execution

step	thread pool		processor activity p_1		p_2	
1			Γ_1:	v_1		
2				v_2		
3			Γ_2:	v_3	Γ_1:	v_{16}
4		Γ_2	Γ_3:	v_4		v_{17}
5	Γ_1	Γ_2		v_5	Γ_6:	v_{18}
6	Γ_1		Γ_2:	v_6		v_{19}
7	Γ_1	Γ_2	Γ_4:	v_7		v_{20}
8		Γ_2		v_8	Γ_1:	v_{21}
9	Γ_1		Γ_2:	v_9		
10	Γ_1		Γ_5:	v_{10}	Γ_2:	v_{13}
11	Γ_1			v_{11}		v_{14}
12		Γ_2		v_{12}	Γ_1:	v_{22}
13	Γ_1		Γ_2:	v_{15}		
14			Γ_1:	v_{23}		

thread pool lists living threads in the pool after each idle thread has removed a ready thread



Properties: greedy; maintains "busy leaves" - every leaf has a processor working on it in every step it is live

Busy Leaves Properties

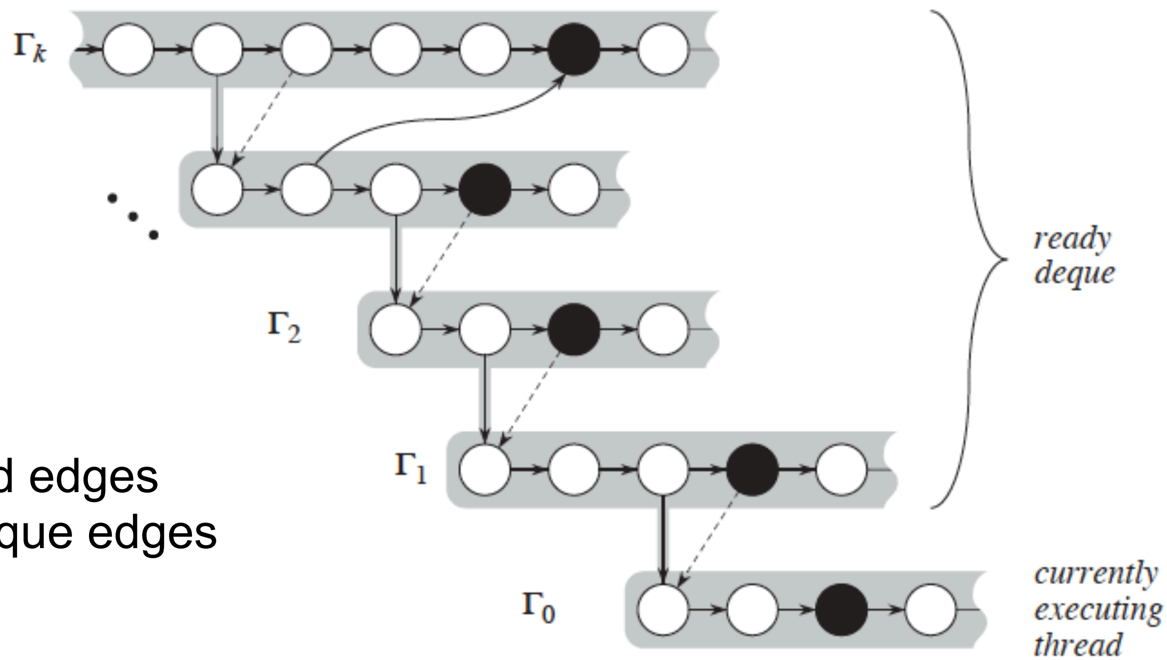
- **Lemma 2 For any multithreaded computation with stack depth S_1, any P processor schedule X that maintains busy leaves has space $S(X) \leq S_1P$**
  - —each spawn subtree has at most P leaves at time t
  - —for each leaf, space used by it and ancestors is at most S_1
  - —therefore, space in use at any time is at most S_1P
- **Theorem 3 For any number P of processors, and any strict multithreaded computation with work T_1, and critical path T_∞ and stack depth S_1, busy leaves algorithm computes an execution schedule X that satisfies $T(X) \leq T_1/P + T_\infty$ and whose space satisfies $S(X) \leq S_1P$**
  - —time bound follows from greedy schedule theorem
  - —space bound follows from Lemma 2
- **Weakness of busy leaves: centralized queue**

Randomized Work Stealing

- Each processor maintains a ready deque, with top & bottom
  - —local thread pushes and pops at bottom
  - —steals occur at top
- Spawn: if a thread spawns a child in a step
  - —return parent thread to bottom of ready deque
  - —begin next step working on child thread
- Stall: if a thread stalls
  - —check ready deque
  - —if non-empty, begin work on bottom thread
    else, begin work on thread stolen from top of randomly chosen deque
- Die: If a thread dies in a step, follow rule for stall
- Enable: if a thread enables another, place the enabled thread on the bottom of the processor's ready deque
- Note: a thread can simultaneously enable a stalled thread, and stall or die

A Ready Deque

- **Lemma 4: For k > 0, threads in a ready deque satisfy:**
  - **—for i=1,2,...,k, thread Γ_i is the parent of Γ_{i-1}**
  - **—if we have k > 1, then for i=1,2,...,k-1, thread Γ_i has not been worked on since it spawned Γ_{i-1}**



dashed edges
are deque edges

Space Bound of Work Stealing

- **Theorem 5: For any fully strict multithreaded computation with stack depth S_1, the work stealing algorithm run on P processors uses at most $S_1 P$ space**
- **Proof**
  - —enough to prove work stealing algorithm maintains busy leaves
  - —at every time step, every leaf must be ready, so it is either
    - – in the ready deque
    - – has a processor working on it
  - —lemma 4 guarantees that no leaf sits the a ready deque while a processor works on another thread

Work Stealing and Contention

Main Result

- If requests are
  - —made randomly by P processors to P deques
  - —each processor has at most one outstanding request
- then, total amount of time processors spend waiting for their requests to be satisfied is likely to be proportional to the total number M of requests
  - —no matter which processors make the requests
  - —no matter how the requests are distributed over time
- Proof by balls and bins game

(P,M) Recycling Game

- **P: # balls in the game, which is equal to the # of bins**
- **M: total # ball tosses executed by the adversary**
- **Game rules**
  - **—adversary removes some balls in the reservoir, tosses each ball to a bin, which is selected uniformly and independently at random**
  - **—for each bin that has at least one ball, adversary removes any one of the balls in the bin and returns it to the reservoir**
- **Model servicing of steal requests**
  - **—each ball and each bin owned by distinct processor**
  - **—if ball is in reservoir: owner is not making steak request**
  - **—ball in bin: owner has made steal request to bin's owner**
  - **—ball removed from bin and returned to owner: request serviced**

Contention Delay Analysis

- n_t denotes # balls left in the bins at step t
- Delay of a ball r is a random variable that denotes the total number of steps that finish with ball r in a bin
- Define the total delay $D = \sum_{t=1,T} n_t$
- Goal of adversary: maximize D
- Lemma 6
  - —for any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the total delay of the (P, M) recycling game is $O(M + P \lg P + P \lg (1/\varepsilon))$, and the expected total delay is at most M

Atomic Accesses

- Assumption: concurrent accesses to a data structure are serially queued by an adversary
- If concurrent steal requests are made to a deque, in one time step
  - —one request is satisfied
  - —others are queued by an adversary
- Adversary cannot choose to serve none if there is at least one request

Execution Time Analysis

- **At each step, we collect P dollars, one from each process**
- **At each step, each processor places its dollar in one of three buckets**
  - **—if it executes an instruction, put it in the WORK bucket**
  - **—if it executes a steal, put it in the STEAL bucket**
  - **—if it waits, put it into the WAIT bucket**
- **Lemma 7**
  - **—The execution of a fully strict computation with work T_1 by the work stealing algorithm on a computer with P processors terminates with exactly T_1 dollars in the WORK bucket.**
- **Lemma 12**
  - **—For any epsilon > 0, with probability at least $1 - \varepsilon$, at most $O(P(T_\infty + \lg(1/\varepsilon))$ work steal attempts occur. The expected number is $O(PT_\infty)$.**

Contribution

Randomized work stealing algorithm for fully strict computations

- **Provably efficient in time, space, and communication**
  - —expected running time = $T_1/P + O(T_\infty)$
    - – T_1 is the serial time
    - – T_∞ is the minimum execution time on an infinite number of processors
  - —space bound = $S_1 P$
    - – S_1 is the minimum serial space
    - – better than previous bound for work stealing

      helped in part by fully-strict model of computation
  - —expected total communication is at most $O(P\,T_\infty(1+n_d)S_{max})$
    - – S_{max} : size of the largest activation record of any thread
    - – n_d : maximum number of times any thread synchronizes with its parent
    - – bound justifies intuition that work stealing has less communication than work sharing
- **Results are practical and the basis for Cilk's scheduler**

Beyond This Paper

- Showed that the time bound $O(T_1/P + T_\infty)$ applies to arbitrary multithreaded computations
  - —need not be strict or fully strict
- Extended these ideas to an efficient scheduler for multiprogrammed environments