CSE 613: Parallel Programming

Lecture 4
(Scheduling and Work Stealing)

(inspiration for some slides comes from lectures given by Charles Leiserson)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2012
A runtime/online scheduler maps tasks to processing elements dynamically at runtime. The map is called a schedule.

An offline scheduler prepares the schedule prior to the actual execution of the program.
A strand / task is called *ready* provided all its parents ( if any ) have already been executed.

- executed task
- ready to be executed
- not yet ready

A *greedy scheduler* tries to perform as much work as possible at every step.
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )

- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
Let $p = \text{number of cores}$

At every step:
- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )
- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
A Centralized Greedy Scheduler

Let $p = \text{number of cores}$

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  (complete step)

- if $< p$ tasks are ready:
  execute all of them
  (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )

- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )

$p = 3$
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )

- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )

- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them ( complete step )
- if $< p$ tasks are ready: execute all of them ( incomplete step )
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  (complete step)

- if $< p$ tasks are ready:
  execute all of them
  (incomplete step)
Let $p$ = number of cores

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
Let \( p \) = number of cores

At every step:

- if \( \geq p \) tasks are ready:
  
  execute any \( p \) of them
  
  (complete step)

- if \( < p \) tasks are ready:
  
  execute all of them
  
  (incomplete step)
A Centralized Greedy Scheduler

Let \( p \) = number of cores

At every step:

- if \( \geq p \) tasks are ready:
  execute any \( p \) of them
  ( complete step )

- if \( < p \) tasks are ready:
  execute all of them
  ( incomplete step )
Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
Greed Scheduling Theorem

Theorem [Graham’68, Brent’74]:
For any greedy scheduler,
\[ T_p \leq \frac{T_1}{p} + T_\infty \]

Proof:
\[ T_p = \# \text{complete steps} + \# \text{incomplete steps} \]

- Each complete step performs \( p \) work:
  \[ \# \text{complete steps} \leq \frac{T_1}{p} \]

- Each incomplete step reduces the span by 1:
  \[ \# \text{incomplete steps} \leq T_\infty \]
Optimality of the Greedy Scheduler

Corollary 1: For any greedy scheduler $T_p \leq 2T_p^*$, where $T_p^*$ is the running time due to optimal scheduling on $p$ processing elements.

Proof:

Work law: $T_p^* \geq \frac{T_1}{p}$

Span law: $T_p^* \geq T_\infty$

∴ From Graham-Brent Theorem:

$$T_p \leq \frac{T_1}{p} + T_\infty \leq T_p^* + T_p^* = 2T_p^*$$
Corollary 2: Any greedy scheduler achieves \( S_p \approx p \) (i.e., nearly linear speedup) provided \( \frac{T_1}{T_\infty} \gg p \).

Proof:

Given, \( \frac{T_1}{T_\infty} \gg p \Rightarrow \frac{T_1}{p} \gg T_\infty \)

\[ \therefore \text{From Graham-Brent Theorem:} \]

\[ T_p \leq \frac{T_1}{p} + T_\infty \approx \frac{T_1}{p} \]

\[ \Rightarrow \frac{T_1}{T_p} \approx p \Rightarrow S_p \approx p \]
Work-Sharing and Work-Stealing Schedulers

Work-Sharing

– Whenever a processor generates new tasks it tries to distribute some of them to underutilized processors
– Easy to implement through centralized (global) task pool
– The centralized task pool creates scalability problems
– Distributed implementation is also possible (but see below)

Work-Stealing

– Whenever a processor runs out of tasks it tries to steal tasks from other processors
– Distributed implementation
– Scalable
– Fewer task migrations compared to work-sharing (why?)
Cilk++’s Work-Stealing Scheduler

- A randomized distributed scheduler
- Time bounds
  - Provably: $T_p = \frac{T_1}{p} + O(T_\infty)$ (expected time)
  - Empirically: $T_p \approx \frac{T_1}{p} + T_\infty$
- Space bound: $\leq p \times$ serial space bound
- Has provably good cache performance
Cilk++’s Work-Stealing Scheduler

- Each core maintains a work dqueue of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it steals one from the top of the dqueue of a random core
Cilk++’s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core
Cilk++’s Work-Stealing Scheduler

- Each core maintains a work dqueue of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it steals one from the top of the dqueue of a random core
Cilk++’s Work-Stealing Scheduler

- Each core maintains a work dqueue of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it steals one from the top of the dqueue of a random core
Cilk++’s Work-Stealing Scheduler

- Each core maintains a work dqueue of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it steals one from the top of the dqueue of a random core
Cilk++’s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core
Cilk++’s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core
Space Usage of Cilk++’s Scheduler
( Problem with Linear Stacks )

– C/C++ uses a *linear* ( contiguous ) *stack* to store function activation records ( i.e., stack frames )

– When a function is called
  o The caller pushes the return address onto the stack
  o The callee allocates its local variables in the stack space

– The callee’s stack frame lies directly above the caller’s one

– But linear stacks do not work well for parallel programs ( why? )
Space Usage of Cilk++’s Scheduler (Cactus Stack)

- Cilk++ uses a *cactus stack*
  - A heap allocated tree of stack frames
  - Not necessarily contiguous
- A cactus stack supports several views of the stack in parallel
Space Usage of Cilk++’s Scheduler

Theorem: Let $S_1$ be the stack space required by a serial execution of a Cilk++ program. Then the stack space used when run on $p$ processing elements is, $S_p \leq pS_1$.

Proof:

- At any given time step, the spawn subtree can have at most $p$ leaves
- For each such leaf, the stack space used by it and all its ancestors is at most $S_1$