

Today:

- Amortized Analysis (examples)
- Multithreaded Algs.

COSC 581, Algorithms

March 11, 2014

Reading Assignments

- Today's class:
 - Chapter 17 (Amortized analysis)
 - Chapter 27 (Multithreaded algs)
- Reading assignment for next class:
 - Chapter 27 (continued)
- **Announcement: Exam #2 on Tuesday, April 1**
 - Will cover greedy algorithms, amortized analysis
 - HW 6-9

Recall from last time: In-Class Exercise

Suppose we perform a sequence of n operations on a data structure in which the i th operation costs i if i is an exact power of 2, and 1 otherwise. Let c_i be the cost of the i th operation. Use **aggregate analysis** to determine the amortized costs per operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
...	...

Cost of n operations:

$$\begin{aligned} \sum_{i=1}^n c_i &\leq n + \sum_{j=0}^{\lg n} 2^j \\ &= n + \frac{2^{\lg n+1} - 1}{2 - 1} \\ &= n + (2n - 1) \\ &< 3n \end{aligned}$$

Thus, average cost of operations = $\frac{\text{Total cost}}{\# \text{ operations}} < 3$.

By aggregate analysis, the amortized cost per operation = $O(1)$

Variant: In-Class Exercise #1

Suppose we perform a sequence of n operations on a data structure in which the i th operation costs i if i is an exact power of 2, and 1 otherwise. Let c_i be the cost of the i th operation. Use **accounting method** to determine the amortized costs per operation.

In-Class Exercise #2

Suppose we wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter.

(Hint: Keep a pointer to the high-order 1.)

Multithreaded Algorithms

Motivation

- We have discussed **serial algorithms** that are suitable for running on a **uniprocessor** computer. We will now extend our model to **parallel algorithms** that can run on a **multiprocessor computer**.

Computational Model

- There exist many competing models of parallel computation that are essentially different. For example, one can have shared or distributed memory.
- Since multicore processors are ubiquitous, we focus on a parallel computing model with shared memory.

Dynamic Multithreading

- Programming a shared-memory parallel computer can be difficult and error-prone. In particular, it is difficult to partition the work among several threads so that each thread approximately has the same load.
- A **concurrency platform** is a software layer that coordinates, schedules, and manages parallel-computing resources. We will use a simple extension of the serial programming model that uses the concurrency instructions **parallel**, **spawn**, and **sync**.

Spawn

- **Spawn**: If spawn proceeds a procedure call, then the procedure instance that executes the spawn (the parent) may continue to execute in parallel with the spawned subroutine (the child), instead of waiting for the child to complete.
- The keyword **spawn** does not say that a procedure must execute concurrently, but simply that it may.
- At runtime, it is up to the scheduler to decide which sub-computations should run concurrently.

Sync

- The keyword `sync` indicates that the procedure must wait for all its spawned children to complete.

Parallel

- Many algorithms contain loops, where all iterations can operate in parallel. If the `parallel` keyword precedes a for loop, then this indicates that the loop body can be executed in parallel.

Fibonacci Numbers – Definition

- The Fibonacci numbers are defined by the recurrence:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2} \text{ for } i > 1.$$

Naive Algorithm

- Computing the Fibonacci numbers can be done with the following algorithm:

FIB(n)

if $n \leq 1$ then return n;

x = FIB (n-1);

y = FIB (n-2) ;

return x + y;

Caveat: Running Time

- Let $T(n)$ denote the running time of $\text{FIB}(n)$. Since this procedure contains two recursive calls and a constant amount of extra work, we get
- $T(n) = T(n-1) + T(n-2) + \theta(1)$

$$\text{which yields } T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

- Since this is an **exponential growth** in n , this is a particularly bad way to calculate Fibonacci numbers.
- But, to illustrate the principles of parallel programming, we will use this naive (bad) algorithm anyway

Parallelization of Fibonacci

- Parallel algorithm to compute Fibonacci numbers:

P-FIB(n)

if $n \leq 1$ **return** n;

else x = **spawn** P-FIB (n-1); // parallel execution

y = **spawn** P-FIB (n-2) ; // parallel execution

sync; // wait for results of x and y

return x + y;

Computation DAG (Prelim)

- Multithreaded computation can be better understood with the help of a **computation directed acyclic graph** $G=(V,E)$.
- The vertices V in the graph are the **instructions**.
- The edges E represent **dependencies** between instructions.
- An edge $(u,v) \in E$ means that the instruction u must execute before instruction v .
- [Problem: Somewhat too detailed. We will group the instructions into strands.]

Strands

- A sequence of instructions containing no parallel control (spawn, sync, return from spawn, parallel) can be grouped into a single **strand**.

Computation DAG

- A computation DAG= (V,E) consists a vertex set V that comprises the strands of the program.
- The edge set E contains an edge (u,v) if and only if the strand u needs to execute before strand v .
- If there is an edge between strand u and v , then they are said to be (logically) in series. If there is no edge, then they are said to be (logically) in parallel.

Edge Classification

- A **continuation edge** (u,v) connects a strand u to its successor v within the same procedure instance.
- When a strand u spawns a new strand v , then (u,v) is called a **spawn edge**.
- When a strand v returns to its calling procedure and x is the strand following the parallel control, then the **return edge** (v,x) is included in the graph.

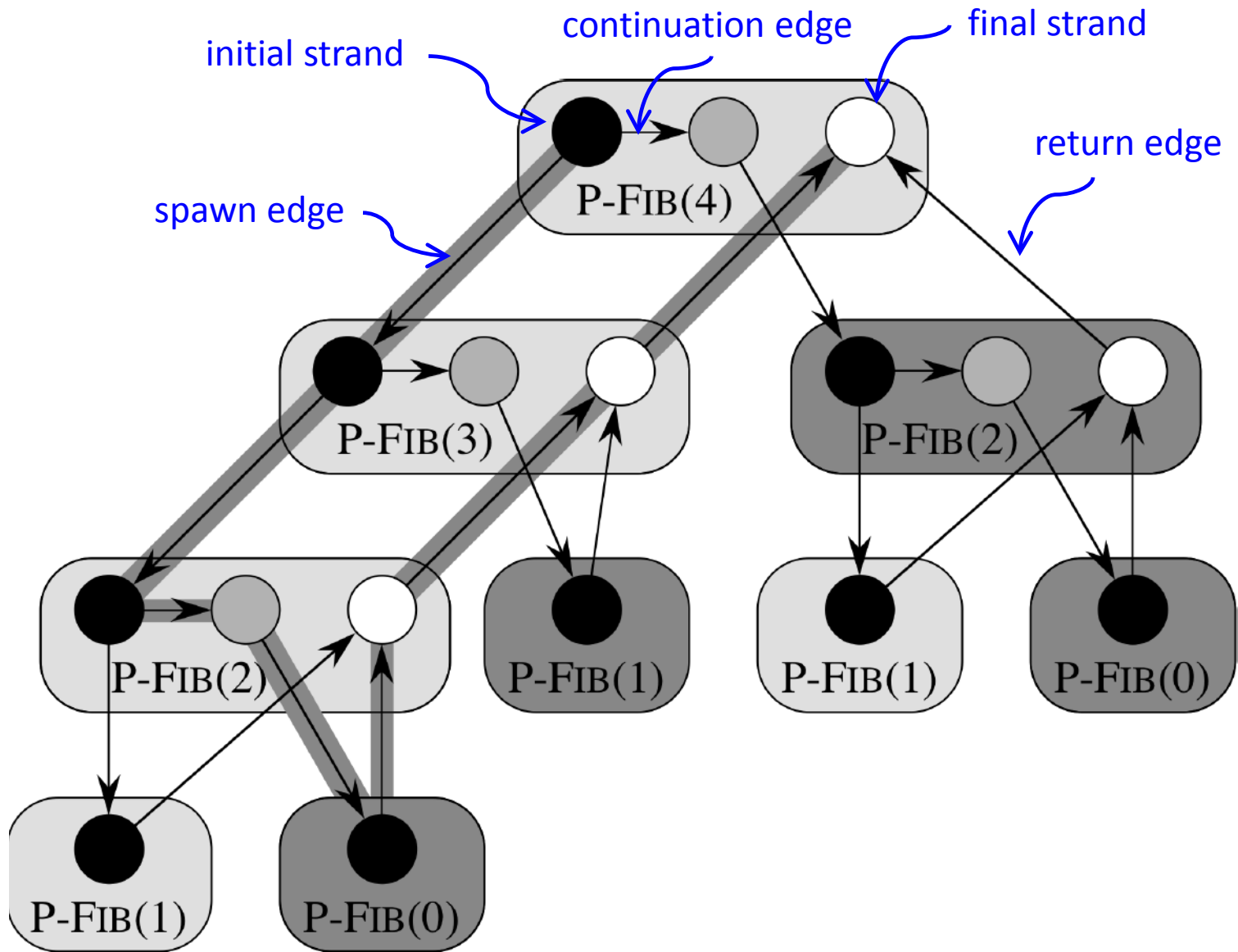
Fibonacci Example

- Parallel algorithm to compute Fibonacci numbers:

P-FIB(n)

```
if  $n \leq 1$  return n;    // initial strand
else x = spawn P-FIB (n-1); // next lower strand
      y = spawn P-FIB (n-2); // second strand // lower strand
      sync; // final strand
      return x + y;
```

Fibonacci Computation DAG



Performance Measures

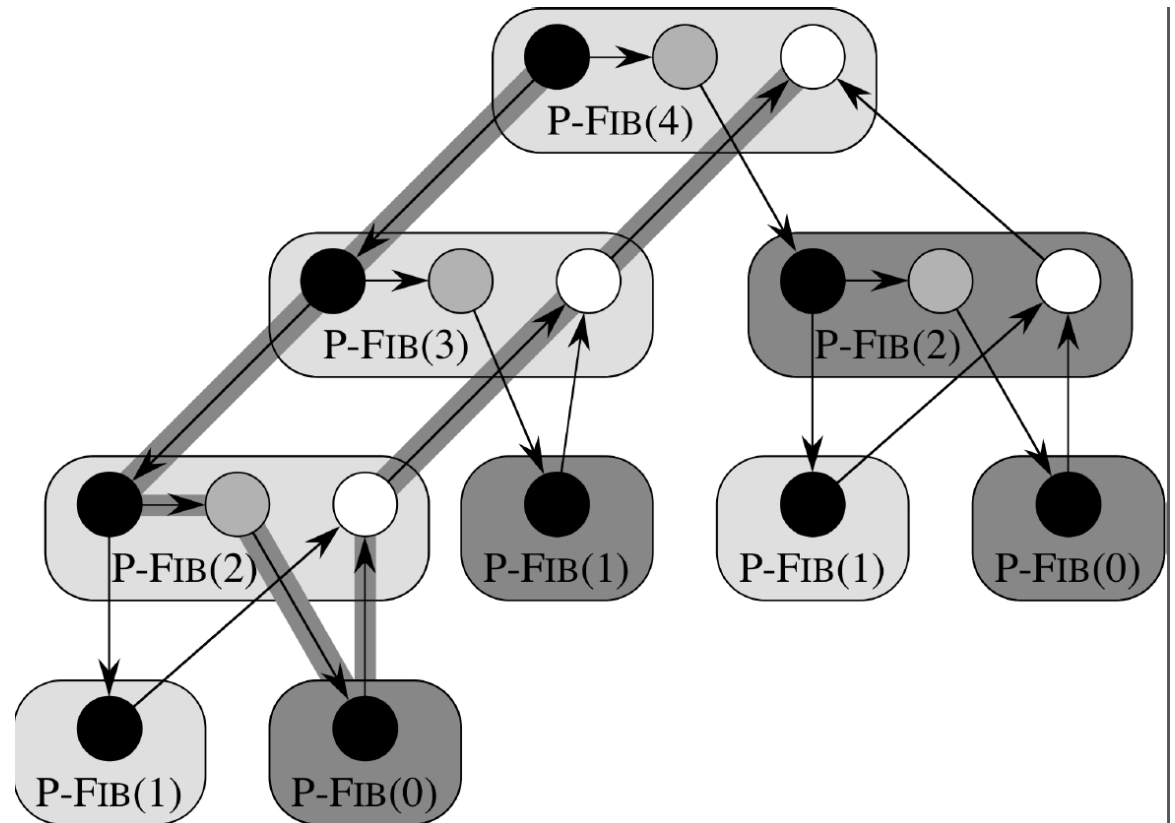
- The **work** of a multithreaded computation is the total time to execute the entire computation on one processor.
- $\text{Work} = \text{sum of the times taken by each strand}$

Performance Measures

- The **span** is the longest time to execute the strands along any path of the computational directed acyclic graph.

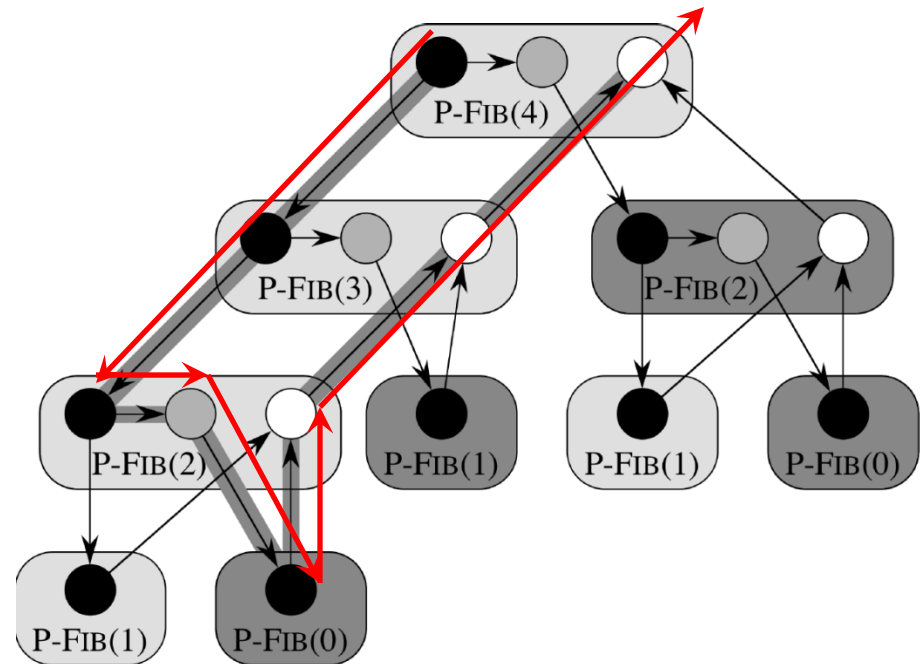
Performance Measure for P-FIB(4)

- What is work?
- What is span?



Performance Measure Example

- In P-Fib(4), we have:
 - 17 vertices = 17 threads.
 - 8 vertices on longest path.
- Assuming unit time for each thread, we get:
 - work = 17 time units
 - span = 8 time units



Taking into account # processors...

- The actual running time of a multithreaded computation depends not just on its work and span, but also on how many processors (cores) are available, and how the scheduler allocates strands to processors.
- Running time on P processors is indicated by subscript:
 - T_1 running time on a single processor
 - T_P running time on P processors
 - T_∞ running time on unlimited processors

Work Law

- An ideal parallel computer with P processors can do at most P units of work. Total work to do is T_1 .
- Thus, $PT_P \geq T_1$
- The work law is:

$$T_P \geq \frac{T_1}{P}$$

Span Law

- A P-processor ideal parallel computer cannot run faster than a machine with unlimited number of processors.
- However, a computer with unlimited number of processors can emulate a P-processor machine by using simply P of its processors. Therefore,

$$T_P \geq T_\infty$$

- This is called the **span law**.

Speedup and Parallelism

- The **speed up** of a computation on P processors is defined as: $\frac{T_1}{T_P}$
- The **parallelism** of a multithreaded computation is given by: $\frac{T_1}{T_\infty}$

Scheduling

- The performance depends not just on the work and span. Additionally, the **strands must be scheduled efficiently**.
- The strands must be mapped to static threads, and the operating system schedules the threads on the processors themselves.
- The scheduler must schedule the computation with no advance knowledge of when the strands will be spawned or when they will complete; it must operate online.

Greedy Scheduler

- We will assume a greedy scheduler in our analysis, since this keeps things simple. A **greedy scheduler** assigns as many strands to processors as possible in each time step.
- On P processors, if at least P strands are ready to execute during a time step, then we say that the step is a **complete step**; otherwise we say that it is an **incomplete step**.

Greedy Scheduler Theorem

- On an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with work T_1 and span T_∞ in time:

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

- Given the fact the best we can hope for on P processors is $T_P = T_1/P$ by the work law, and $T_P = T_\infty$ by the span law, the sum of these two gives the lower bounds

Reading Assignments

- Today's class:
 - Chapter 17
- Reading assignment for next class:
 - Chapter 17 (continued)
 - (Later) Chapter 27 (Multithreaded algs)
- **Announcement: Exam #2 on Tuesday, April 1**
 - Will cover greedy algorithms, amortized analysis, multi-threaded algorithms
 - HW 6-9