CSE 332 Summer 2024
Lecture 20: Analysis

Nathan Brunelle

http://www.cs.uw.edu/332

Parallel Algorithm Analysis

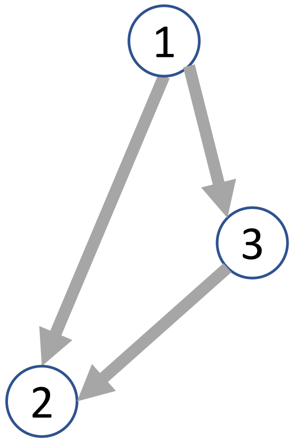
- How to define efficiency
  - Want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors

Work and Span

- Let $T_P(n)$ be the running time if there are P processors available
- Two key measures of run time:
  - Work: How long it would take 1 processor, so $T_1(n)$
    - Just suppose all forks are done sequentially
    - Cumulative work all processors must complete
    - For array sum: $\Theta(n)$
  - Span: How long it would take an infinite number of processors, so $T_\infty(n)$
    - Theoretical ideal for parallelization
    - Longest “dependence chain” in the algorithm
    - Also called “critical path length” or “computation depth”
    - For array sum: $\Theta(\log n)$

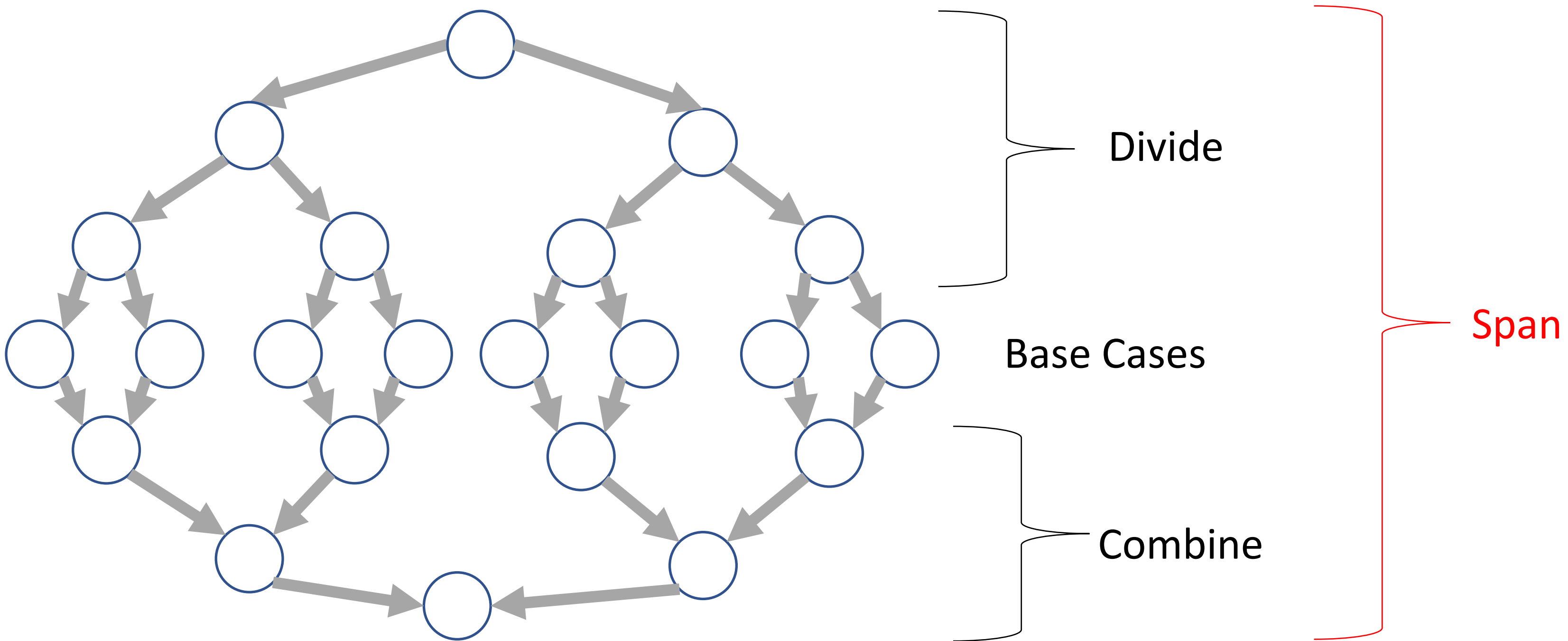
Directed Acyclic Graph (DAG)

- A directed graph that has no cycles
- Often used to depict dependencies
  - E.g. software dependencies, Java inheritance, dependencies among threads!



ForkJoin DAG

- “Sketches” what parts of the algorithm may be done in parallel vs. must be done in-order
  - Each node is a “step” of the algorithm that may depend on other steps (draw an edge) or not
- Fork and Join each create a new node
  - When calling fork/compute
    - Algorithm creates two new threads, there is a dependency from the creating code to the code done by these threads
  - When calling join
    - There is a dependency from the code done by the other thread to the code after join



Work Law

- States that $P \cdot T_P(n) \geq T_1(n)$
  - P processors can do at most P things in parallel
    - Work must match the sum of the operations done by all processors, so if this does not hold then the parallel algorithm somehow skipped steps that sequential version would have done.
  - If the "division of labor" across processors is uneven then it can be that $pT_P(n) > T_1(n)$

More Vocab

- Speedup:
  - How much faster (than one processor) do we get for more processors
    - Identifies how well the algorithm scales as processors increases
    - May be different for different algorithms
  - $T_1(n)/T_P(n)$
- Perfect linear Speedup
  - The "ideal" speedup
  - $\frac{T_1}{T_P} = P$
- Parallelism
  - Maximum possible speedup
  - T_1/T_∞
  - At some point more processors won't be more helpful, when that point is depends on the span
- Writing parallel algorithms is about increasing span without substantially increasing work

Asymptotically Optimal T_P

- T_P cannot be better than $\frac{T_1}{P}$
  - Because of the Work Law
- T_P cannot be better than T_∞
  - A finite number of processors can't outperform an infinite number ("Span Law")
- Considering both of these, we can characterize the best-case scenario for T_P
  - $T_P(n) \in \Omega\left(\frac{T_1(n)}{P} + T_\infty(n)\right)$
  - $T_1(n)/P$ dominates for small P, $T_\infty(n)$ dominates for large P
- ForkJoin Framework gives an expected time guarantee of asymptotically optimal!

Division of Responsibility

- Our job as ForkJoin Users:
  - Pick a good algorithm, write a program
  - When run, program creates a DAG of things to do
  - Make all the nodes a small-ish and approximately equal amount of work
- ForkJoin Framework Developer’s job:
  - Assign work to available processors to avoid idling
    - Abstract away scheduling issues for the user
  - Keep constant factors low
  - Give the expected-time optimal guarantee

And now for some bad news…

- In practice it’s common for your program to have:
  - Parts that parallelize well
    - Maps/reduces/filters over arrays and other data structures
  - Parts that don’t parallelize at all
    - Reading a linked list, getting input, or computations where each step needs the results of previous step
- These unparallelizable parts can turn out to be a big bottleneck

Amdahl's Law (mostly bad news)

- Suppose $T_1 = 1$
  - Work for the entire program is 1
- Let S be the proportion of the program that cannot be parallelized
  - $T_1 = S + (1 - S) = 1$
- Suppose we get perfect linear speedup on the parallel portion
  - $T_P = S + \frac{1-S}{P}$
- For the entire program, the speedup is:
  - $\frac{T_1}{T_P} = \frac{1}{S+\frac{1-S}{P}}$
- The parallelism (infinite processors) is:
  - $\frac{T_1}{T_\infty} = \frac{1}{S}$

Ahmdal's Law Example

- Suppose 2/3 of your program is parallelizable, but 1/3 is not.
  - $S = \frac{2}{3}$
  - $T_1 = \frac{2}{3} + \frac{1}{3} = 1$
- $T_P = S + \frac{1-S}{P}$
- If T_1 is 100 seconds:
  - $T_P = 33 + \frac{67}{P}$
  - $T_3 = 33 + \frac{67}{3} = 33 + 22 = 55$
  - $T_6 =$
  - $T_{12} =$

Conclusion

- Even with many *many* processors the sequential part of your program becomes a bottleneck
- Parallelizable code requires skill and insight from the developer to recognize where parallelism is possible, and how to do it well.

Other Reasons to Use Threads

- Code Responsiveness:
  - While doing an expensive computation, you don’t what your interface to freeze
- Processor Utilization:
  - If one thread is waiting on a deep-hierarchy memory access you can still use that processor time
- Failure Isolation:
  - If one portion of your code fails, it will only crash that one portion.

Memory Sharing With ForkJoin

- Idea of ForkJoin:
  - Reduce span by having many parallel tasks
  - Each task is responsible for **its own portion** of the input/output
  - If one task needs another's result, use join() to ensure it uses the final answer
- This does not help when:
  - Memory accessed by threads is overlapping or unpredictable
  - Threads are doing independent tasks using same resources (rather than implementing the same algorithm)

Example: Shared Queue

```
enqueue(x){
	if ( back == null ){
		back = new Node(x);
		front = back;
	}
	else {
		back.next = new Node(x);
		back = back.next;
	}
}
```

Imagine two threads are both using the same linked list based queue.

What could go wrong?

Concurrent Programming

- Concurrency:
  - Correctly and efficiently managing access to shared resources across multiple possibly-simultaneous tasks
- Requires synchronization to avoid incorrect simultaneous access
  - Use some way of “blocking” other tasks from using a resource when another modifies it or makes decisions based on its state
  - That blocking task will free up the resource when it’s done
- Warning:
  - Because we have no control over when threads are scheduled by the OS, even correct implementations are highly non-deterministic
  - Errors are hard to reproduce, which complicates debugging

Bank Account Example

- The following code implements a bank account object correctly for a synchronized situation
- Assume the initial balance is 150

```
class BankAccount {
        private int balance = 0;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount); }
        // other operations like deposit, etc.
}
```

What Happens here?

```
withdraw(100);
withdraw(75)
```

Bank Account Example - Parallel

- Assume the initial balance is 150

```
class BankAccount {
        private int balance = 0;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount); }
        // other operations like deposit, etc.
}
```

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

Interleaving

- Due to time slicing, a thread can be interrupted at any time
  - Between any two lines of code
  - Within a single line of code
- The sequence that operations occur across two threads is called an interleaving
- Without doing anything else, we have no control over how different threads might be interleaved

A “Good” Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

Thread 1	Thread 2

	int b = getBalance();
	if (amount > b)
	throw new Exception();
	setBalance(b – amount);

int b = getBalance();
if (amount > b)
throw new Exception();
setBalance(b – amount);	

A “Bad” Interleaving

- Assume the initial balance is 150

Thread 1:	Thread 2:
withdraw(100);	withdraw(75);

int b = getBalance();	
	int b = getBalance();
	if (amount > b)
	throw new Exception();
	setBalance(b – amount);
if (amount > b)	
throw new Exception();	
setBalance(b – amount);	

A Bad Fix

- Assume the initial balance is 150

```
class BankAccount {
        private int balance = 0;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                if (amount > getBalance())
                        throw new WithdrawTooLargeException();
                setBalance(getBalance() – amount); }
        // other operations like deposit, etc.
}
```

A still "Bad" Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

Thread 1	Thread 2
	if (amount > getBalance()) throw new Exception();
if (amount > getBalance()) throw new Exception();	
setBalance(getBalance() – amount);	
	setBalance(getBalance() – amount);
setBalance(getBalance() – amount);	

What we want – Mutual Exclusion

- While one thread is withdrawing from the account, we want to exclude all other threads from also withdrawing
- Called mutual exclusion:
  - One thread using a resource (here: a bank account) means another thread must wait
  - We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a **critical section**.
- The programmer must implement critical sections!
  - It requires programming language primitives to do correctly

A Bad attempt at Mutual Exclusion

```
class BankAccount {
        private int balance = 0;
        private Boolean busy = false;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                while (busy) { /* wait until not busy */ }
                busy = true;
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount);
                busy = false;}
        // other operations like deposit, etc.
}
```

A still "Bad" Interleaving

- Assume the initial balance is 150

Thread 1:	Thread 2:
withdraw(100);	withdraw(75);
while (busy) { /* wait until not busy */ }	
busy = true;	
int b = getBalance();	
	while (busy) { /* wait until not busy */ }
	busy = true;
	int b = getBalance();
	if (amount > b)
	throw new Exception();
	setBalance(b – amount);
	busy = false;
if (amount > b)	
throw new Exception();	
setBalance(b – amount);	
busy = false;	

Solution

- We need a construct from Java to do this
- One Solution – A **Mutual Exclusion Lock** (called a Mutex or Lock)
- We define a **Lock** to be a ADT with operations:
  - New:
    - make a new lock, initially “not held”
  - Acquire:
    - If lock is not held, mark it as “held”
      - These two steps always done together in a way that cannot be interrupted!
    - If lock is held, pause until it is marked as “not held”
  - Release:
    - Mark the lock as “not held”

Almost Correct Bank Account Example

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                lk.acquire();
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount);
                lk.release();}
        // other operations like deposit, etc.
}
```

Questions:

1. What is the critical section?
2. What is the Error?

Try…Finally

- Try Block:
  - Body of code that will be run
- Finally Block:
  - Always runs once the program exits try block (whether due to a return, exception, anything!)

Correct (but not Java) Bank Account Example

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                try{
                        lk.acquire();
                        int b = getBalance();
                        if (amount > b)
                                throw new WithdrawTooLargeException();
                        setBalance(b – amount); }
                finally { lk.release(); } }
        // other operations like deposit, etc.
}
```

Questions:

1. Should deposit have its own lock object, or the same one?
2. What about getBalance?
3. What about setBalance?

A still “Bad” Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
if(getBalance()<75)
        setBalance(75);
```

Thread 1:

```
try{
        lk.acquire();
        int b = getBalance();
        if (amount > b)
                throw new Exception();


        setBalance(b – amount); }
finally { lk.release(); }
```

Thread 2:

```
if(getBalance() < 75)
        setBalance(75);
```

What’s wrong here…

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int setBalance(int x) {
                try{
                        lk.acquire();
                        balance = x; }
                finally{ lk.release(); } }
        void withdraw(int amount) {
                try{
                        lk.acquire();
                        int b = getBalance();
                        if (amount > b)
                                throw new WithdrawTooLargeException();
                        setBalance(b – amount); }
                finally { lk.release(); } }}
```

Withdraw calls setBalance!

Withdraw can never finish because in setBalance the lock will always be held!

Re-entrant Lock (Recursive Lock)

- Idea:
  - Once a thread has acquired a lock, future calls to acquire on the same lock will not block progress
- If the lock used in the previous slide is re-entrant, then it will work!

Re-entrant Lock Details

- A re-entrant lock (a.k.a. recursive lock)
- "Remembers"
  - the thread (if any) that currently holds it
  - a count of "layers" that the thread holds it
- When the lock goes from not-held to held, the count is set to 0
- If (code running in) the current holder calls acquire:
  - it does not block
  - it increments the count
- On release:
  - if the count is > 0, the count is decremented
  - if the count is 0, the lock becomes not-held

Java’s Re-entract Lock Class

- java.util.concurrent.locks.ReentrantLock
- Has methods lock() and unlock()
- Important to guarantee that lock is always released!!!
- Recommend something like this:

```
myLock.lock();
try { // method body }
finally { myLock.unlock(); }
```

How this looks in Java

```
java.util.concurrent.locks.ReentrantLock;
class BankAccount {
        private int balance = 0;
        private ReentrantLock lck = new ReentrantLock();
        int setBalance(int x) {
                try{
                        lk.lock();
                        balance = x; }
                finally{ lk.unlock(); } }
        void withdraw(int amount) {
                try{
                        lk.lock();
                        int b = getBalance();
                        if (amount > b)
                                throw new WithdrawTooLargeException();
                        setBalance(b – amount); }
                finally { lk.unlock(); } }}
```

Java Synchronized Keyword

- Syntactic sugar for re-entrant locks
- You can use the synchronized statement as an alternative to declaring a ReentrantLock
- Syntax:

```
synchronized( /* expression returning an Object */ ) {statements}
```

- Any Object can serve as a “lock”
  - Primitive types (e.g. int) cannot serve as a lock
- Acquires a lock and blocks if necessary
  - Once you get past the “{“, you have the lock
- Released the lock when you pass “}”
  - Even in the cases of returning, exceptions, anything!
  - Impossible to forget to release the lock

Back Account Using Synchronize (Attempt 1)

```
class BankAccount {
	private int balance = 0;
	private Object lk = new Object();
	int getBalance() {
		synchronized (lk) { return balance; }
	}
	void setBalance(int x) {
		synchronized (lk) { balance = x; }
	}
	void withdraw(int amount) {
		synchronized (lk) {
			int b = getBalance();
			if (amount > b)
				throw new Exception();
			setBalance(b – amount); } } // deposit would also use synchronized(lk)
}
```

Back Account Using Synchronize (Attempt 2)

```
class BankAccount {
        private int balance = 0;
        int getBalance() {
                synchronized (this) { return balance; }
        }
        void setBalance(int x) {
                synchronized (this) { balance = x; }
        }
        void withdraw(int amount) {
                synchronized (this) {
                        int b = getBalance();
                        if (amount > b)
                                throw new Exception();
                        setBalance(b – amount); } } // deposit would also use synchronized(lk)
}
```

Since we have one lock per account regardless of operation, it's more intuitive to use the account object itself as the lock!

More Syntactic Sugar!

- Using the object itself as a lock is common enough that Java has convenient syntax for that as well!
- Declaring a method as “**synchronized**” puts its body into a synchronized block with “this” as the lock

Back Account Using Synchronize (Final)

```
class BankAccount {
	private int balance = 0;
	synchronized int getBalance() { return balance; }
	synchronized void setBalance(int x) { balance = x; }
	synchronized void withdraw(int amount) {
		int b = getBalance();
		if (amount > b)
			throw new WithdrawTooLargeException();
		setBalance(b – amount); }
	// other operations like deposit (which would use synchronized)
}
```