CSE 332 Winter 2024
Lecture 21: Analysis

Nathan Brunelle

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

Parallel Pack

Input:	10	16	4	18	8	2	14	9
Output:	10	16	18	14				
	0	1	2	3	4	5	6	7

$, f(x) = x > 9$

1. Do a map to identify the true elements

1	1	0	1	0	0	1	0

2. Do prefix sum on the result of the map to identify the count of true elements seen to the left of each position

1	2	2	3	3	3	4	4

3. Do a map using the previous results fill in the output

10	16	18	14

3. Do a map using the result of the prefix sum to fill in the output

Input:	10	16	4	18	8	2	14	9
Map Result:	1	1	0	1	0	0	1	0
Prefix Result:	1	2	2	3	3	3	4	4

- Because the last value in the prefix result is 4, the length of the output is 4
- Each time there is a 1 in the map result, we want to include that element in the output
- If element i should be included, its position matches prefixResult[i]-1

```
Int[] output = new int[prefixResult[input.length-1]];
FORALL(int i = 0; i < input.length; i++){
        if (mapResult[i] == 1)
                output[prefixResult[i]-1] = input[i];
}
```

Parallel Algorithm Analysis

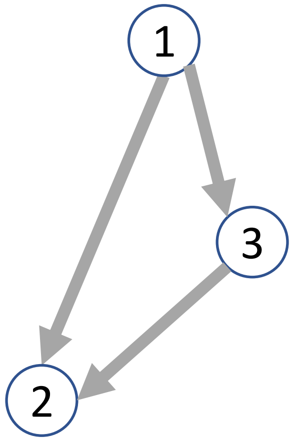
- How to define efficiency
  - Want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors

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: $\Theta(\log n)$

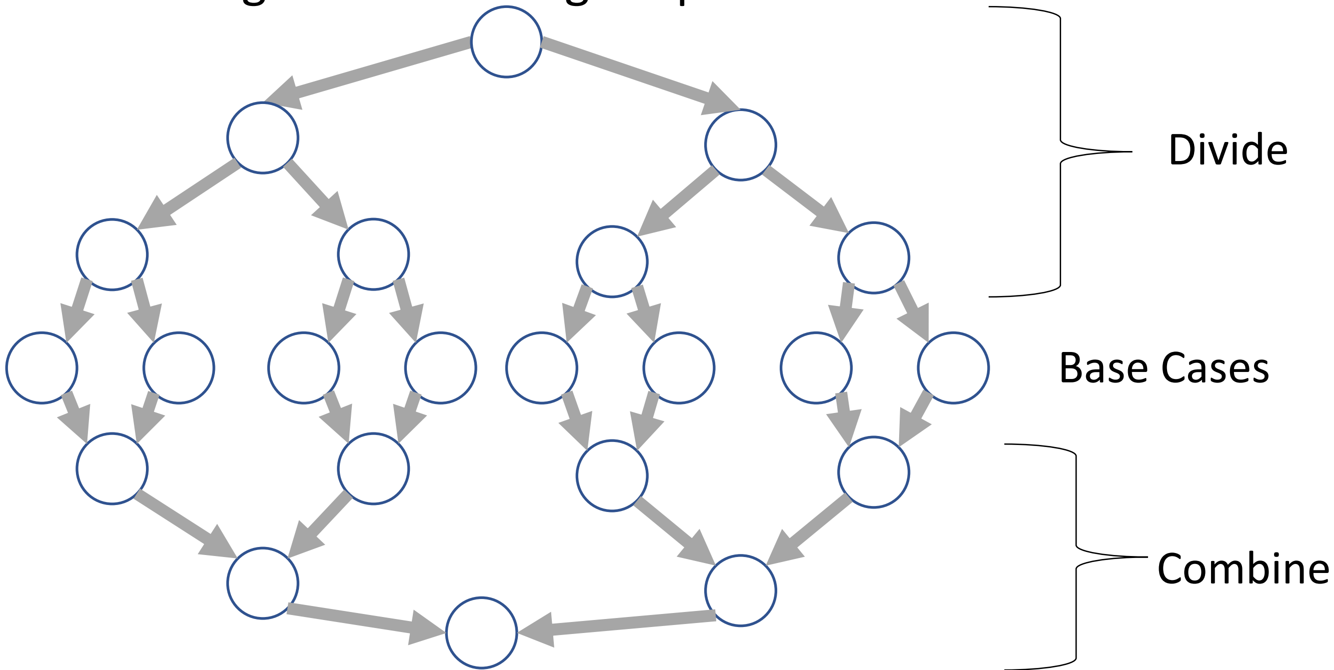
Directed Acyclic Graph (DAG)

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



ForkJoin DAG

- Fork and Join each create a new node
  - Fork branches into two threads
    - Those two threads “depended on” their source thread to be created
  - Join combines to threads
    - The thread doing the combining “depends on” the other threads to finish



More Vocab

- Speed Up:
  - How much faster (than one processor) do we get for more processors
  - $T_1(n)/T_P(n)$
- Perfect linear Speedup
  - $\frac{T_1}{T_P} = P$
  - Hard to get in practice
  - “Holy Grail” or parallelizing
- 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

- We know how to compute T_1 and T_∞, but what about T_P?
  - T_P cannot be better than $\frac{T_1}{P}$
  - $T_P$cannot be better than T_∞
- An asymptotically optimal execution would be
  - $T_P(n) \in O\left(\frac{T_1(n)}{P} + T_\infty(n)\right)$
  - $T_1(n)/P$ 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 over arrays and other data structures
  - And 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 unparallelized 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 speed is:
  - $\frac{T_1}{T_P} = \frac{1}{S+\frac{1-S}{P}}$
- And so 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.
  - $S = \frac{2}{3}$
  - $T_1 = \frac{2}{3} + \frac{1}{3} = 1$
- $T_P = S + \frac{1-S}{P}$
- So if T_1 is 100 seconds:
  - $T_P = 33 + \frac{67}{P}$
  - $T_3 = 33 + \frac{67}{3} = 33 + 22 = 55$

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.

Reasons to use threads (beyond algorithms)

- 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);
```

	`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);	

Another result?

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

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

	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 */ }
                                                while (busy) { /* wait until not busy */ }
busy = true;
                                                busy = true;
int b = getBalance();
                                                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?