Lecture 31: Parallelism II & Concurrency I

CS 62

Spring 2018

Alexandra Papoutsaki & William Devanny

Some slides based on those fom Dan Grossman, U. of Washington

ForkJoin

- Create a ForkJoinPool
- Don't subclass `Thread` → Subclass `RecursiveTask<V>`
- Don't override `run` → Do override `compute`
- Do not use an `ans` field → Do return a `V` from `compute`
- Don't call `start` → Do call `fork`
- Call `join` that returns answer

Getting good results in practice

- Sequential threshold
    - Library documentation recommends doing approximately 100-5000 basic operations in each "piece" of your algorithm
- Library needs to "warm up" – May see slow results before the Java virtual machine reoptimizes the library internals
- Wait until your computer has more processors
    - Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important

Work and Span

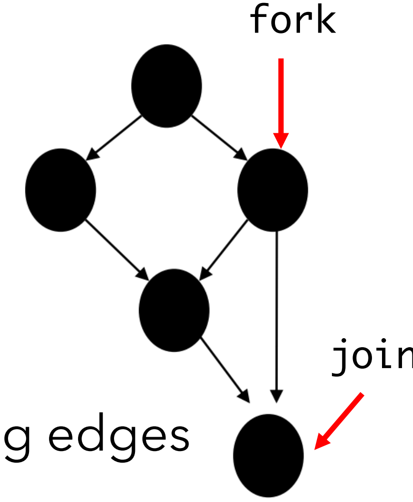
- With a sequential algorithm, we consider $T(n)$ as its runtime
- For a parallel algorithm, we will consider T_P or $T_P(n)$ as the runtime of the algorithm using P processors.
- There are two important runtime quantities for a parallel algorithm:
  - How long it would take if it were to run on one processor (**work**)
  - How long it would take if it were as parallel as possible (**span**)

Definitions

- **Work**: $T_1(n) = T(n)$ or T_1 is how long it takes to run on one processor, that is the total of all the running times of all the pieces of the algorithm
- **Span**: $\mathrm{T}_\infty(n)$ or T_∞ is how long it takes to run on an unlimited number of processors
  - Not necessarily $O(1)$ time
  - Still need to do forking and combine results

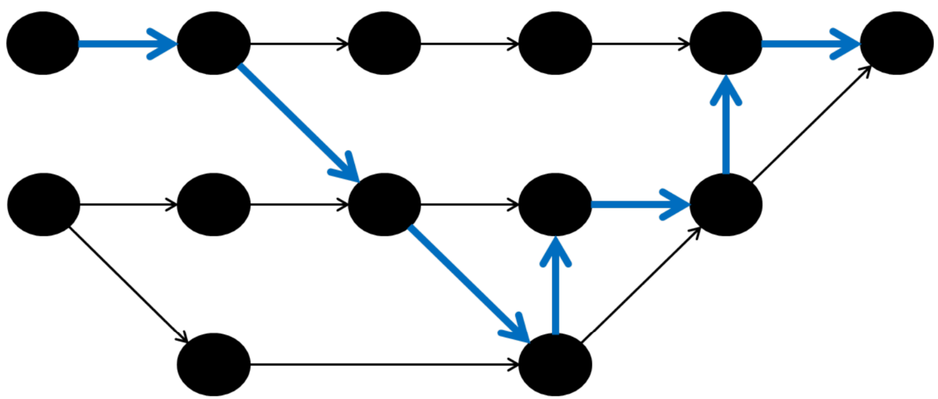
Program Graph

- A program execution using `fork` and `join` can be seen as a DAG
  - A DAG is a graph that is directed (edges have direction (arrows)), and those arrows do not create a cycle (path that starts and ends at the same node).
- Nodes: Pieces of work, each $O(1)$ amount of work
- Edges: Dependencies –
  Source must finish before destination starts
- A `fork` "ends a node" and makes two outgoing edges
  - New thread and continuation of current thread
- A `join` "ends a node" and makes a node with two incoming edges
  - Node just ended and last node of thread joined on



Work and Span on Program Graph

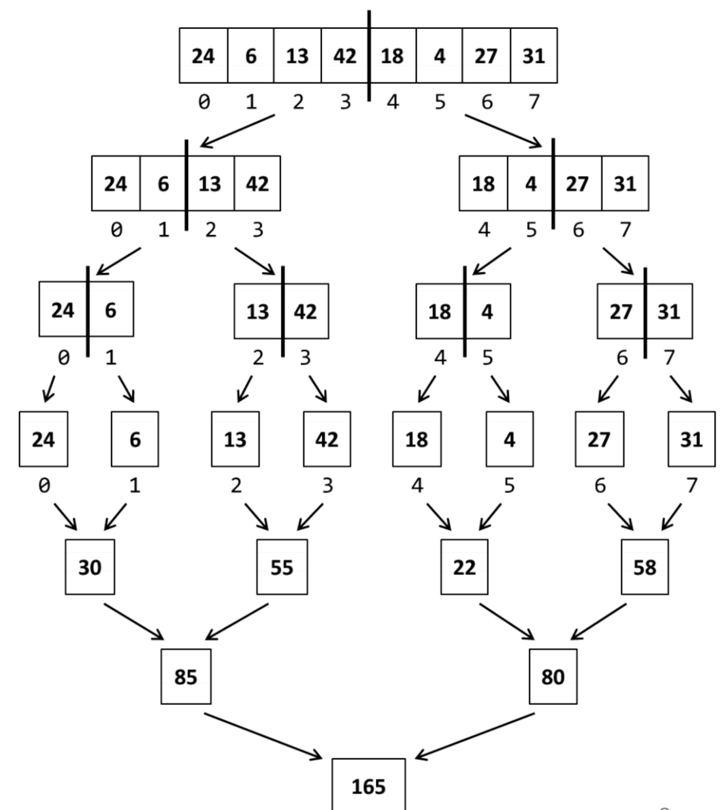
- We can now describe work and span as:
- **Work**: How long it would take on 1 processor = T_1
  Sum of run-time of all nodes in DAG, i.e. number of nodes
- **Span**: How long it would take infinity processors = T_∞
  Sum of all run-time of all nodes on most expensive path in DAG, i.e. length of longest path in DAG



$T_\infty = 7$

Execution DAG on summing an array

- The work in the nodes in the top half is to create two subproblems.
- The work in the nodes in the bottom half is to combine two results.
- T_1 is $O(n)$ since there are approximately $2n$ nodes.
- T_∞ is $O(\log n)$ two trees of height $\log n$ each.



Performance

- **Speedup** on P processors: $\frac{T_1}{T_P}$
  - Ratio of how much faster it would run on P processors
  - E.g., if T_1 is 20 and T_4 is 8, then speedup is 2.5
- **Perfect speedup**: P as we vary P
  - E.g., 4 for the example above
  - Rare due to overhead of thread creation and communication
- **Perfect linear speedup**: doubling P cuts running time in half
  - Not upper limit

Parallelism

- Reporting T_1/T_P can overstate advantages of parallelism
  - T_1 is runtime of *parallel* algorithm on 1 processor
  - Likely much slower than *sequential* algorithm
- More realistic speedup definition S/T_P
  - S time for sequential algorithm
  - Lower than T_1/T_P
- **Parallelism**: T_1/T_∞
  - Maximum possible speedup
  - At least as great as speedup for any P
  - e.g., for our sum array problem, parallelism is $O(n/\log n)$
  - We can hope for an exponential speedup over sequential version

ForkJoin guarantees expected bound

- $T_P = O((T_1 \,/\, P) \;+\; T_\infty)$
  - Given P processors, no framework can beat T_1/P or T_∞ by more than a constant factor
  - When P is small, T_1/P is dominant, giving roughly linear speedup
  - When P grows, limit influenced by span
- Framework on average gives best performance, assuming user did follow the paradigm as best as possible:
  - All threads ~ same work, careful with load balancing

- Bottom line:
  - Focus on your algorithms, data structures, and cut-offs rather than number of processors and scheduling.
  - Just need T_1, T_∞, and P to analyze running time

Examples for $T_P = O((T_1/P) + T_\infty)$

- For summing:
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect $T_P = O\left(\frac{n}{P} + \log n\right)$
- If instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - Then expect $T_P = O(\frac{n^2}{P} + n)$

Amdahl's Law

- Upper bound on speed-up!
- Suppose the work is 1 unit time.
- Let S be portion of execution that cannot be parallelized.
- $T_1 = S + (1 - S) = 1$
- Suppose we get perfect speedup on parallel portion.
  - $T_P = S + \frac{(1-S)}{P}$
- Then overall speedup with P processors (Amdahl's law):
  - $\frac{T_1}{T_P} = \frac{1}{(S+\frac{1-S}{P})}$
  - *Parallelism* (∞ processors) is: $\frac{T_1}{T_\infty} = \frac{1}{S}$

Bad news

- *Parallelism* (∞ processors) is: $\frac{T_1}{T_\infty} = \frac{1}{S}$
- If 33% of program is sequential, then absolute best speedup is $\frac{1}{0.33} = 3$
    - That means infinitely many processors won't help us get more than a 3 times speed-up!
- From 1980 - 2005, every 12 years gave 100x speedup
    - Now suppose clock speed is same but 256 processors instead of 1.
    - To get 100x speedup, need $100 \leq \frac{1}{(S+\frac{1-S}{P})}$
    - Solve to get solution $S \leq 0.61\%$, so need code to be 99.4% perfectly parallel.

So let’s give up?

- Amdahl tells us that if a particular algorithm has too many sequential computations, it’s better to find a more parallelizable algorithm than to just add more processors.
- Not all is lost. We can change what we compute
    - Computer graphics now much better in video games with GPU’s -- not much faster, but much more detail.
- Side note: Moore’s law is just an observation, while Amdahl’s law is an actual mathematical theorem

Sharing resources

- We’re done talking about parallelism.
- Our goal is no longer (necessarily) “to make the program faster”.
- The ForkJoin Framework is great, but it doesn’t actually allow us to share resources.
    - Two threads only interact at birth and death
- Strategy won’t work well when:
    - Memory accessed by threads is overlapping or unpredictable
    - Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)
- For the next few lectures, we’ll investigate what happens when we lift that restriction.
    - Two threads can run different algorithms now

Concurrent Programming

- Allowing simultaneous or interleaved access to shared resources from multiple clients.
- Requires coordination, particularly synchronization to avoid incorrect simultaneous access: make somebody block
  - join is not what we want
  - block until another thread is “done using what we need” not “completely done executing”

Very complicated, very quickly

- Concurrent code gets very complicated very quickly. Why?
- Concurrency introduces non-determinism!
- In sequential programming, when you run the same program multiple times, you get the same result
- This is no longer true for concurrent programs. Threads can run in any order giving unpredictable results.
- How threads are scheduled affects *what* operations from other threads they see and *when* they see them.
- Non-repeatability complicates testing and debugging.

Examples

- Multiple threads:
  - Processing different bank-account operations
  - What if 2 threads change the same account at the same time?
- Using a shared cache of recent files
  - What if 2 threads insert the same file at the same time?
- Creating pipeline with queue for handing work to next thread in sequence?
  - What if enqueuer and dequeuer adjust a circular array queue at the same time?

Threads again?!

- Not about speed, but code structure for responsiveness
- Example: Respond to GUI events in one thread while another thread is performing an expensive computation
- Processor utilization (mask I/O latency)
  - If 1 thread "goes to disk," have something else to do
- Failure isolation
  - Convenient structure if we want to interleave multiple tasks and don't want an exception in one to stop the other

Sharing is caring

- Common to have different threads access the same resources in an unpredictable order or even at about the same time
- But program correctness requires that simultaneous access be prevented using synchronization
- Simultaneous access is rare
  - Makes testing difficult
  - Must be much more disciplined when designing / implementing a concurrent program
  - We will discuss common idioms known to work