# Lecture 19: Analysis of Fork-Join Parallel Programs

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

# Announcements

- EX08 due today
- EX09 due Monday
- EX10 released today
- Exam 2 information posted here:
  - https://courses.cs.washington.edu/courses/cse332/25su/exams/final.html
  - **Note: it will be hard to accommodate makeups; only four days to grade**
  - If you can't make proposed makeup dates (e.g., sickness/emergency), some options:
  - Option 1: Exam 1 is worth 40% instead of 20% of overall grade
  - Option 2: Take the final exam in the next CSE 332 offering

# Today

- Java Thread Library
- Java ForkJoin Library
- Simple Parallel Patterns: Map + Reduce

- Analyzing Parallel Algorithms
  - Work and Span
  - Amdahl's Law

# Today

- Java Thread Library
- Java ForkJoin Library
- Simple Parallel Patterns: Map + Reduce

- Analyzing Parallel Algorithms
  - Work and Span
  - Amdahl's Law

# Today

- Java Thread Library
- Java ForkJoin Library
- Simple Parallel Patterns: Map + Reduce

- Analyzing Parallel Algorithms
  - Work and Span
  - Amdahl's Law

# Today

- Java Thread Library
- Java ForkJoin Library
- Simple Parallel Patterns: Map + Reduce

- Analyzing Parallel Algorithms
  - Work and Span
  - Amdahl's Law

# Analyzing Algorithms: Work and Span

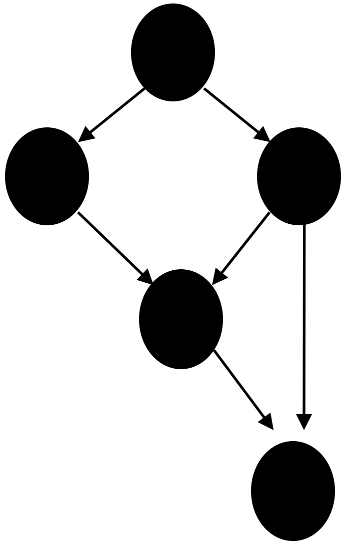
Let $\mathbf{T_P}$ be the running time if there are **P** processors available

Two key measures of run-time:

- Work: How long it would take 1 processor = $\mathbf{T_1}$
  - Just "sequentialize" the recursive forking
  - Cumulative work that all processors must complete
- Span: How long it would take infinity processors = $\mathbf{T_\infty}$
  - The hypothetical ideal for parallelization
  - This is the longest "dependence chain" in the computation
  - Example: $O(\log n)$ for summing an array
    - Notice in this example having > $n/2$ processors is no additional help
  - Also called "critical path length" or "computational depth"

# The DAG (Directed Acyclic Graph)

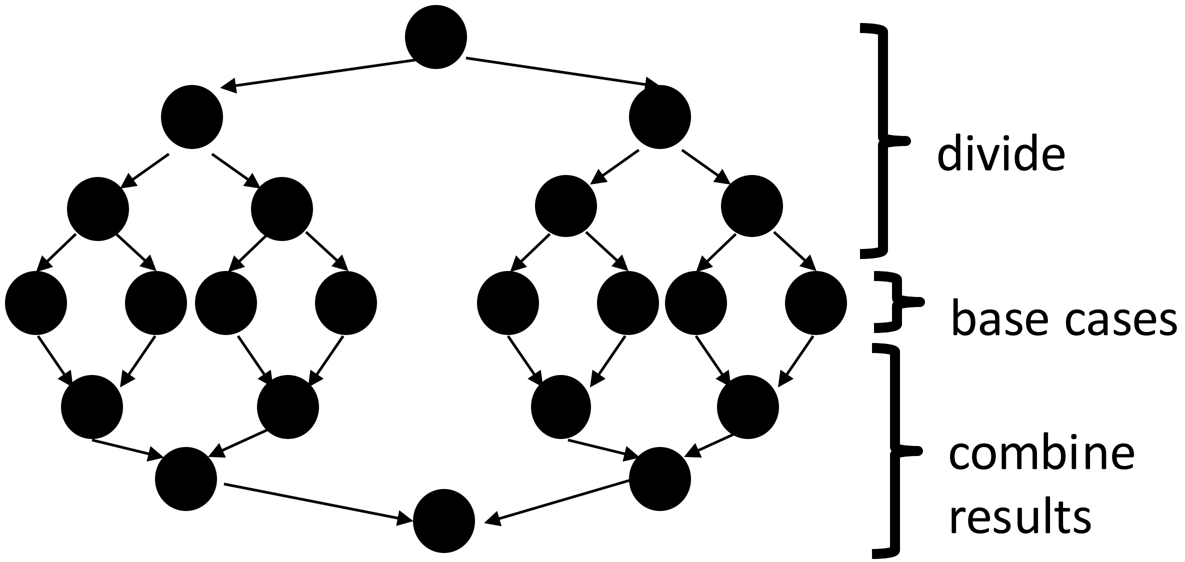
- A program execution using **`fork`** and **`join`** can be seen as a DAG
- [A DAG is a graph that is directed (edges have direction (arrows)), and those arrows do not create a cycle (ability to trace a path that starts and ends at the same node).]
  - **Nodes**: Pieces of work
  - **Edges**: Source must finish before destination starts



- A **`fork`** "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A **`join`** "ends a node" and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

# Our simple examples, in more detail

Our **`fork`** and **`join`** often look like this:




In this context, the span ($T_\infty$) is:

- The longest dependence-chain; longest 'branch' in parallel 'tree'
- Example: *O*(log *n*) for summing an array; we halve the data down to our cut-off, then add back together; *O*(log *n*) steps, O(1) time for each
- Also called "critical path length" or "computational depth"

# Connecting to performance

Recall: $\mathbf{T_P}$ = running time if there are **P** processors available

Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG

- That lonely processor does everything
- Any topological sort is a legal execution
- $O(n)$ for simple maps and reductions

Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG

- Note: costs are on the nodes not the edges
- Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
- $O(\texttt{log}\ n)$ for simple maps and reductions

# Definitions

A couple more terms:

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$

- If speed-up is **P** as we vary **P**, we call it perfect linear speed-up
  - Perfect linear speed-up means doubling **P** halves running time
  - Usually our goal; hard to get in practice

- Parallelism is the maximum possible speed-up: $\mathbf{T_1 / T_\infty}$
  - At some point, adding processors won't help
  - What that point is depends on the span

*Parallel algorithms is about decreasing span without increasing work too much*

# Optimal $T_P$: Thanks ForkJoin library!

- So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)

- Ignoring memory-hierarchy issues (caching), $\mathbf{T_P}$ can't beat
  - $\mathbf{T_1 / P}$ why not?
  - $\mathbf{T_\infty}$ why not?

- So an *asymptotically* optimal execution would be:

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

  - First term dominates for small **P**, second for large **P**

- The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!
  - Guarantee requires a few assumptions about your code…

# Division of responsibility

- Our job as ForkJoin Framework 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*

- The framework-writer's job:
  - Assign work to available processors to avoid idling
    - Let framework-user ignore all scheduling issues
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did his/her job

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

# Examples

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

In the algorithms seen so far (e.g., sum an array):

- $\mathbf{T_1} = O(n)$
- $\mathbf{T_\infty} = O(\texttt{log}\ n)$
- So expect (ignoring overheads): $\mathbf{T_P = \mathit{O}(\mathit{n}/P + \texttt{log}\ \mathit{n})}$

Suppose instead:

- $\mathbf{T_1} = O(n^2)$
- $\mathbf{T_\infty} = O(n)$
- So expect (ignoring overheads): $\mathbf{T_P = \mathit{O}(\mathit{n}^2/P + \mathit{n})}$

# And now for the bad news…

So far: talked about a parallel program in terms of work and span

In practice, it's common that your program has:

a) parts that **parallelize well**:

- Such as maps/reduces over arrays and trees

b) …and parts that **don't parallelize at all**:

- Such as reading a linked list, getting input, or just doing computations where each step needs the results of previous step

These **unparallelized** parts can turn out to be a big bottleneck, which brings us to Amdahl's Law …

# Amdahl's Law (mostly bad news)

Let the ***work*** (time to run on 1 processor) be 1 unit time

Let **S** be the ***portion*** of the execution that can't be parallelized

Then: $T_1 = T_1 S + T_1(1 - S)$

Suppose we get perfect linear speedup *on the parallel portion*

Then: $T_P = T_1 S + \frac{T_1(1-S)}{P}$

**So the theoretical overall speedup with P processors is (Amdahl's Law):**

$$\boldsymbol{\frac{T_1}{T_P} = \frac{1}{S+(1-S)/P}}$$

And the parallelism (infinite processors) is:

$$\frac{T_1}{T_\infty} = \frac{1}{S}$$

# Amdahl's Law

$$T_P = T_1 S + \frac{T_1(1-S)}{P}$$

Suppose our program takes 100 seconds.

And $S$ is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors

6 processors

22 processors

67 processors

1,000,000 processors (approximately).

# Amdahl's Law

$$T_P = T_1 S + \frac{T_1(1-S)}{P}$$

Suppose our program takes 100 seconds.

And $S$ is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors: 33 + 67/3 ≈ 55 seconds

6 processors: 33 + 67/6 ≈ 44 seconds

22 processors: 33 + 67/22 ≈ 36 seconds

67 processors 33 + 67/67 ≈ 34 seconds

1,000,000 processors (approximately). ≈ 33 seconds

# Amdahl's Law

- This is BAD NEWS
- If 1/3 of our program can't be parallelized, we can't get a speedup better than 3.
- No matter how many processors we throw at our problem.
- And while the first few processors make a huge difference, the benefit diminishes quickly.

# Any Questions?