CSE 332 Autumn 2024
Lecture 22: Analysis

Nathan Brunelle + Amanda Yuan

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

Parallel Algorithm Analysis

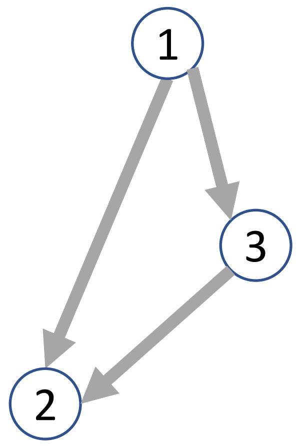
- How to define efficiency
  - Want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors

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: $\Theta(\log n)$

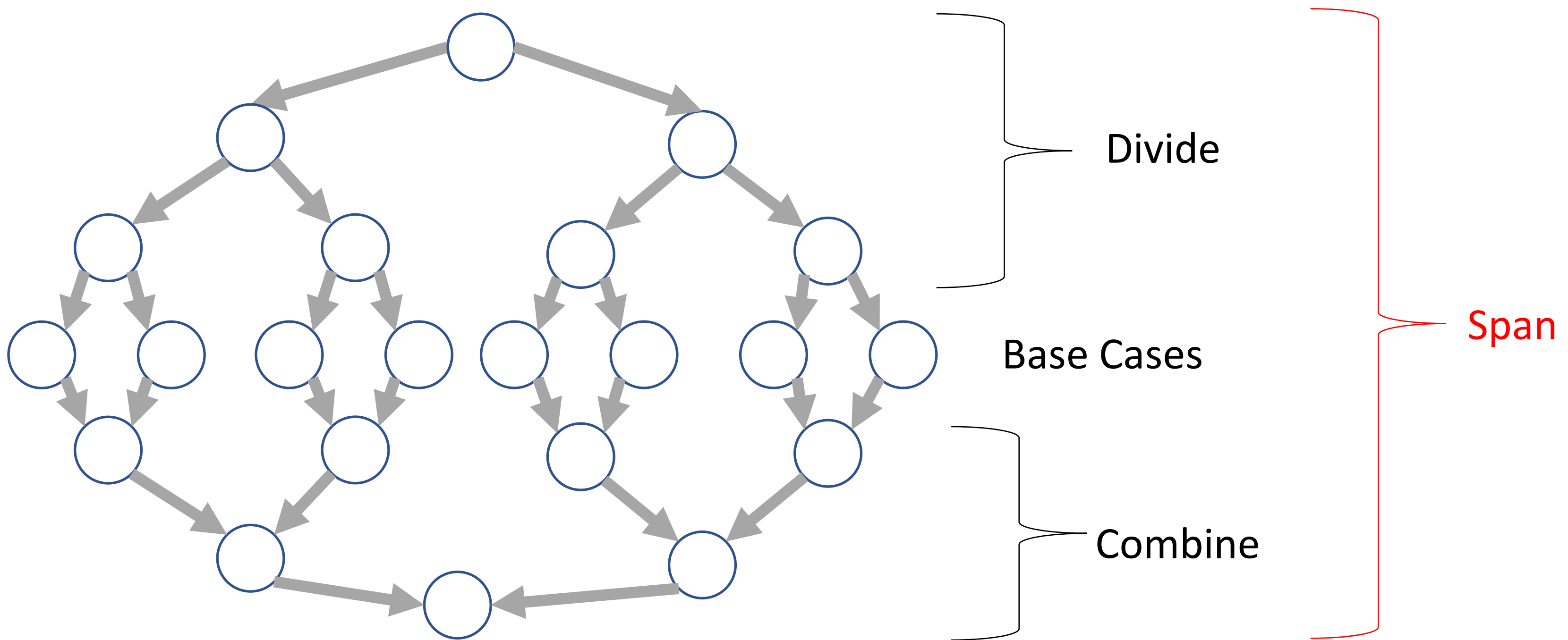
Directed Acyclic Graph (DAG)

- 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/Compute each create a new node
    - When calling fork/compute
        - Algorithm creates 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.
  - $S = \frac{1}{3}$
  - $T_1 = \frac{2}{3} + \frac{1}{3} = 1$
- $T_P = S + \frac{1-S}{P} = \frac{1}{3} + \frac{2/3}{P}$
- If T_1 is 100 seconds:
  - $T_P = 33 + \frac{67}{P}$

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.