Parallelism Analysis

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann Arya GJ Finn Johnson

Joon Chong Kimi Locke Peyton Rapo

Rahul Misal Winston Jodjana

Announcements

- P2 due tomorrow
- P3 out Thursday (spec hopefully out by tomorrow)
- Partner matching – we hope to run the script tonight.
- Midterm due Wednesday
  - Reminder, it should only take about 2.5 hours total for everything. Since you already have the answers, no need to spend long hours to make it perfect.

Lecture Outline

- **Recap of Parallelism Intro**
- Fork/Join Library
- More examples of parallel programs
  - Common patterns: reduce and map
  - Non-array inputs
- Asymptotic Analysis for Fork/Join-style Parallelism

Sequential vs Parallel

- Classic example: Cooking
- Another example: Working with a partner
- Different from what we've done before: Multiple things may be happening at the same time!

Shared Memory with Threads

- Think of Threads as "what one processor will be doing in sequence"
  - One thread will go through its own code from top to bottom

- *Shared Memory with Threads*: A running program has
  - A set of ***threads***, each with its own *program counter* and *call stack*
    - But each thread cannot access to another thread's local variables
  - Threads implicitly share *static fields* and the *heap* (ie, objects)
    - Communication via writing values to some shared location

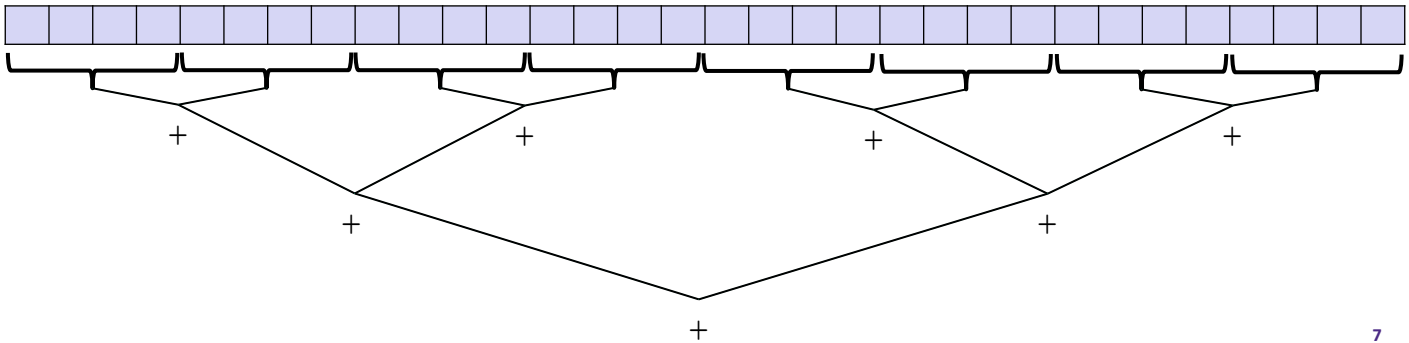
- Can synchronize (ie, make sure the things that need to be done sequentially are done sequentially) with functions like `wait()` or `join()`

Initial approach to Parallel code: problems

- Initially, we split our array up into **k** subarrays and did those operations in parallel.
  - This should give us **k** times speedup, right..?
- What happens when less processors are available than we thought?
- Why break up problem into small chunks?

Fork/Join-style Parallelism

- The key is in parallelizing both the executor-creation and the result-combining phases
  - If enough processors, runtime is **height of the tree:** $O(\log n)$
    - Optimal and exponentially faster than sequential $O(n)$
  - Relies on operations being associative (like +)
- We'll write all our parallel algorithms in this style
  - But using a special library engineered for this style



gradescope

gradescope.com/courses/275833

Problem from Friday:

- Assume that thread creation and joining are expensive. Which of the following optimizations might improve our constants?
  1. Use a cutoff, after which computation proceeds sequentially
  2. Somehow create fewer threads
  3. Somehow reuse threads when they're done

A. Cutoff only
B. Cutoff + Fewer Threads
C. Cutoff + Thread Reuse
D. Cutoff + Fewer Threads + Thread Reuse
E. I'm not sure …

Lecture Outline

- Recap of Parallelism Intro
- **Fork/Join Library**
- More examples of parallel programs
  - Common patterns: reduce and map
  - Non-array inputs
- Asymptotic Analysis for Fork/Join-style Parallelism

Finally! The ForkJoin Library

- Even using fork/join-style code, java.lang.Thread is still too "heavyweight"
  - Constant factors, especially space overhead
  - Creating 20,000 Java threads just a bad idea ☹
- So use the **ForkJoin Library** instead
  - Introduced in Java 8 (2014)
  - Similar libraries available for other languages
    - C/C++: Cilk (inventors), Intel's Thread Building Blocks
    - C#: Task Parallel Library
    - …
  - Its implementation is a fascinating but advanced topic

Thread -> ForkJoin: Terminology

Java Built-in Threads	ForkJoin Library
Subclass `Thread`	Subclass `RecursiveTask<V>`
Override `run()`	Override `compute()`
Call `start()` to begin parallel computation	Call `fork()` to begin parallel computation
Return results via member fields (eg, `ans`)	Return results via return value (ie, an instance of `V`)
Call `join()`, then check its "returned" member field	Call `join()`, then check its return value
Halve created threads by calling `run()` directly	Halve created threads by calling `compute()` directly
Begin recursion with top-level call to `run()` (instead of `start()`)	Begin recursion by creating a `ForkJoinPool` and calling its `invoke()`

Fork/Join-style Parallelism with ForkJoin (1 of 2)

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;     // just the "input" arguments!

  protected Integer compute() {  // override: implement "main"
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      // Just do the calculation in this thread
      int ans = 0;   // local variable instead of a member field
      for (int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;    // direct return of answer
    } else {
      // Create ONE new thread to calculate the left sum
      SumTask left  = new SumTask(arr, lo, (hi+lo)/2);
      SumTask right = new SumTask(arr, (hi+lo)/2, hi);
      left.fork();   // create a thread and call its compute()
      int rightAns = right.compute(); // call compute() directly

      // Combine results
      int leftAns  = left.join();
      return leftAns + rightAns;
    }
  }
}
```

Fork/Join-style Parallelism with ForkJoin (2 of 2)

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;         // input: arguments
  SumTask(int[] a, int l, int h) { … }
  protected Integer compute() { … } // override: implement "main"
}
```

```
static final ForkJoinPool POOL = new ForkJoinPool();

int sum(int[] arr) {
  SumTask task = new SumTask(arr, 0, arr.length);

  // invoke() returns the value which is returned by the
  // top-level compute()
  return POOL.invoke(task);
}
```

Note: Performance Tuning the Library

- Sequential threshold
  - Library documentation recommends doing approximately 100-5000 basic operations in each "piece" of your algorithm
- ForkJoin library needs to "warm up"
  - May see slow results before JVM re-optimizes the library internals
  - Put computations in a loop to see the "long-term benefit"

Summary: Parallelism

- **Parallelism**: increasing efficiency/decreasing total runtime
- **Concurrency**: correctly accessing shared resources
  - They intersect when *parallel computations access shared resources*

- Model: shared memory with explicit threads:
  - **Threads** are the minimum fields necessary to represent "computation": a program counter and a stack
  - Everything else is **shared** (eg, static variables, heap)

- Threading:
  - `run()`/`compute()` are "regular" function calls, but `start()`/`fork()` create a new thread and then call `run()`/`compute()`
  - Parallelizing many small chunks of work is portable, adaptable, and load-balancable

Summary: Fork/Join Parallelism

- **Fork/Join Parallelism** is a *model* that grows the parallelism to fit the problem size using recursion

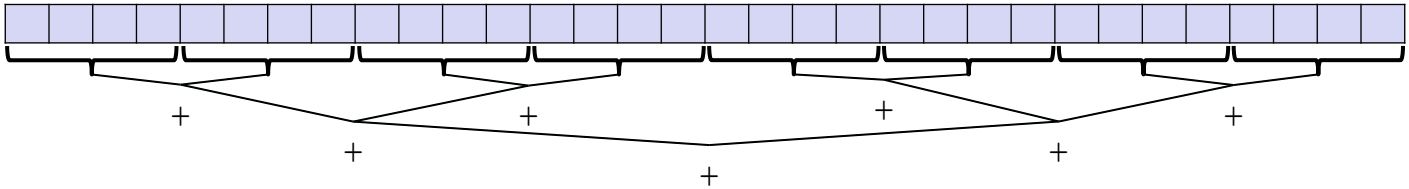
- The **ForkJoin library** that cleanly enables this model
  - You still need to manually specify the sequential cutoff
  - Halving the created threads also requires manual intervention

Lecture Outline

- Recap of Parallelism Intro
- Fork/Join Library
- More examples of parallel programs
  - **Common patterns: reduce and map**
  - Non-array inputs
- Asymptotic Analysis for Fork/Join-style Parallelism

A Common Pattern

- Summing went from $O(n)$ sequential to $O(\texttt{log}\ n)$ parallel
  - Assuming **a lot** of processors and very large n
  - Exponential speed-up in theory: n / $\texttt{log}\ n$ grows exponentially)

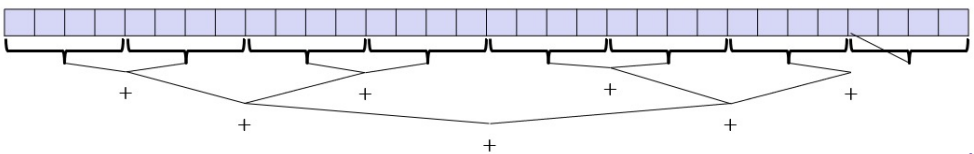


- Any solution which can merge two subsolutions in $O(1)$ time has this property!

- Just need to "plug in" 2 parts:
  - How to compute the result at the cut-off
    (Parallel-Sum: Iterate through sequentially and add up)
  - How to merge results
    (Parallel-Sum: Just add 'left' and 'right' results)

Examples of our Common Pattern

- Assume the input is an array; how would we do the following?
  1. Maximum or minimum element
  2. Is there an element satisfying some property (e.g., is there a 17)?
  3. Left-most element satisfying some property (e.g., first 17)
  4. Smallest rectangle encompassing a number of points
  5. Counts; for example, number of strings that start with a vowel
  6. Are these elements in sorted order?



A Common Pattern: Reductions

- This class of computations are called **reductions**
  - We 'reduce' a large array of data to a single final result
  - Intermediate results must be combined with an **associative operator**
  - *Examples*: max, count, leftmost, rightmost, sum, product, …
- Intermediate and final results can be "aggregates": arrays or multi-field objects
  - *Example*: histogram from a much larger array of test results
- Some things are inherently sequential
  - *Example*: How we process **`arr[i]`** depends entirely on the result of processing **`arr[i-1]`**

Another Common Pattern: Maps

- A **map** transforms each element of a collection independently, creating a new-but-same-sized collection of modified elements
  - No combining results
- *Example*: Vector addition

```
int[] vectorAdd(int[] arr1, int[] arr2) {
  assert(arr1.length == arr2.length);

  result = new int[arr1.length];
  FORALL (i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

- Just need to "plug in" one part:
  - How to map element E to transformed E'
  - *(Vector-add: generate result[i] from arr1[i])*

Maps in the ForkJoin Library (1 of 2)

- Many small tasks still helps with load balancing
  - Maybe not for vector-add, but definitely for compute-intensive maps
  - The forking is O(log n); theoretically other approaches are O(1)

```
class VectorAdd extends RecursiveAction {
  // input: arguments
  int lo; int hi; int[] res; int[] v1; int[] v2;

  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        res[i] = v1[i] + v2[i];
    } else {
      int mid = (hi+lo)/2;
      VectorAdd left = new VectorAdd(lo, mid, res, v1, v2);
      VectorAdd right= new VectorAdd(mid, hi, res, v1, v2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
```

Maps in the ForkJoin Library (2 of 2)

```
class VectorAdd extends RecursiveAction {
  // input: arguments
  int lo; int hi; int[] res; int[] v1; int[] v2;

  protected void compute() { … } // override: implement "main"
}
```

```
static final ForkJoinPool POOL = new ForkJoinPool();

int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);

  // Use ans as an "output argument" instead of looking at the
  // top-level compute()'s return value (which is void).
  int[] ans = new int[arr1.length];

  POOL.invoke(new VectorAdd(0, arr.length, ans, arr1, arr2);
  return ans;
}
```

Map and Reduce in the ForkJoin Library

- Map (vector-add)
  - `VectorAdd` extended `RecursiveAction`
  - Result was an output parameter; nothing returned from `compute()`
- Reduce (parallel-sum):
  - `SumTask` extended `RecursiveTask`
  - Result directly returned from `compute()`
- … but it doesn't *have* to be this way
  - Map could've used `RecursiveTask` to return an array
  - Reduce could've used `RecursiveAction` and returned result as an output parameter

Maps and Reductions, Generally

- Maps and reductions are the "workhorses" of parallel programming
  - By far, the two most important and common patterns
    - Two more-advanced patterns in next lecture

- These are important to help you recognize cases where you can just use a pattern rather than come up with a solution yourself.
  - Also helpful to use maps and reductions to describe (parallel) algorithms

- Goal: programming them becomes "trivial"
  - Exactly like sequential for-loops seem second-nature nowadays

gradescope

gradescope.com/courses/275833

- In Exercise 10, `hasOver`, you'll be returning true or false if a given array has any elements greater than some input. Is this an example of a Reduction or a Map?

Lecture Outline

- Recap of Parallelism Intro
- Fork/Join Library
- More examples of parallel programs
  - Common patterns: reduce and map
  - **Non-array inputs**
- Asymptotic Analysis for Fork/Join-style Parallelism

Parallelized Computation on Trees

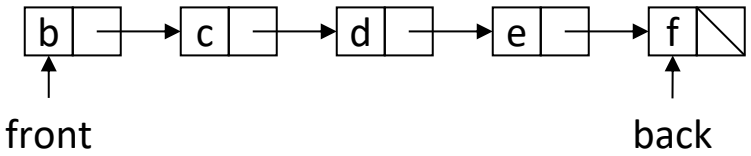
- Maps and reductions work on trees
  - Divide-and-conquer each child rather than array sub-ranges
  - Correct for unbalanced trees, but won't get much speed-up unless tree is balanced

- *Example*: minimum in an *unsorted*-but-balanced binary tree
  - $O(\log n)$ time given enough processors

- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or could approximate it with, e.g., AVL-tree height

Parallelized Computation on Linked Lists

- Can you parallelize maps or reduces over linked lists?
  - *Example*: Increment all elements of a linked list
  - *Example*: Sum all elements of a linked list



- Parallelism still helps with expensive per-element operations

- Once again, data structures matter!
  - Balanced trees allow faster access to all the data: O(log n) vs. O(n)
  - Trees and lists have the same flexibility compared to arrays (eg, inserting an item in the middle of the list)

Lecture Outline

- Recap of Parallelism Intro
- Fork/Join Library
- More examples of parallel programs
  - Common patterns: reduce and map
  - Non-array inputs
- **Asymptotic Analysis for Fork/Join-style Parallelism**

Analyzing Parallel Algorithms

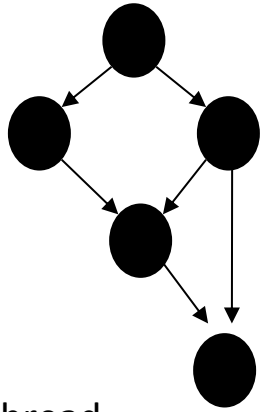
- How to measure efficiency?
  - Want asymptotic bounds
  - Want an analysis that's independent of a specific number of processors
- Fork/Join parallelism gets *asymptotically optimal* runtime for the available number of processors
  - So we can analyze algorithms assuming this guarantee

Modelling Fork/Join Parallelism with DAGs

- A program execution using can be modeled as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination can start

A directed acyclic graph (DAG) is:
- A graph that is directed (edges have direction/arrows)
- And whose edges do not create a cycle (ability to trace a path that starts and ends at the same node)

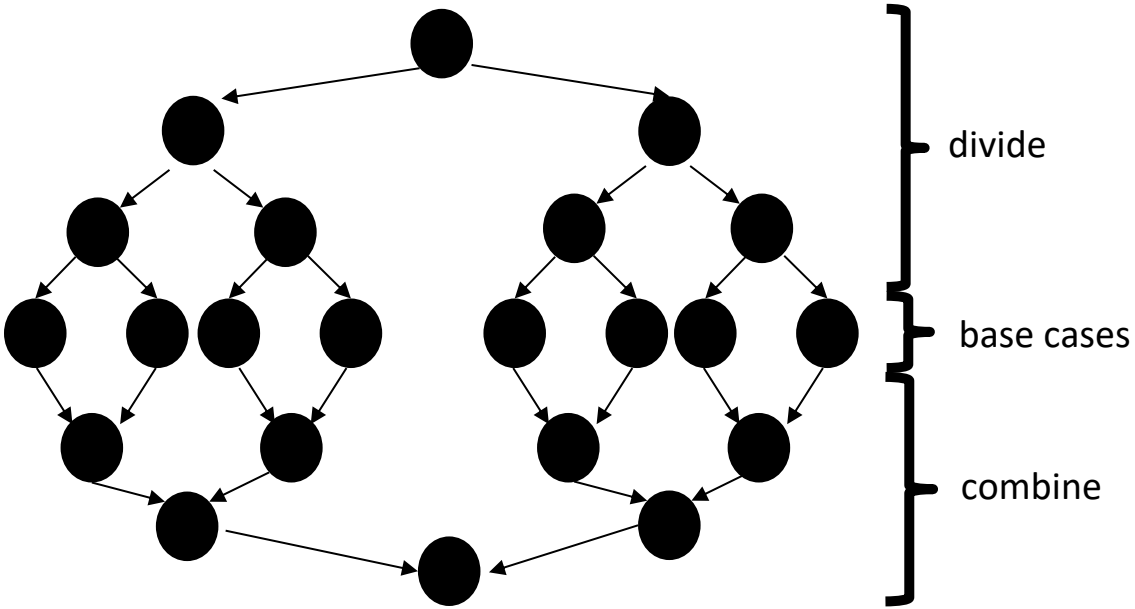
- A `fork` makes two outgoing edges:
  - New thread
  - Continuation of current thread



- A `join` takes two incoming edges
  - The final node of the joined thread
  - The computation that just finished in the current thread

Our Simple Examples

- **`fork`** and **`join`** are very flexible, but maps and reductions use them in a very basic way
  - A (perfect) tree, on top of an upside-down (perfect) tree



Aside: More Interesting DAGs?

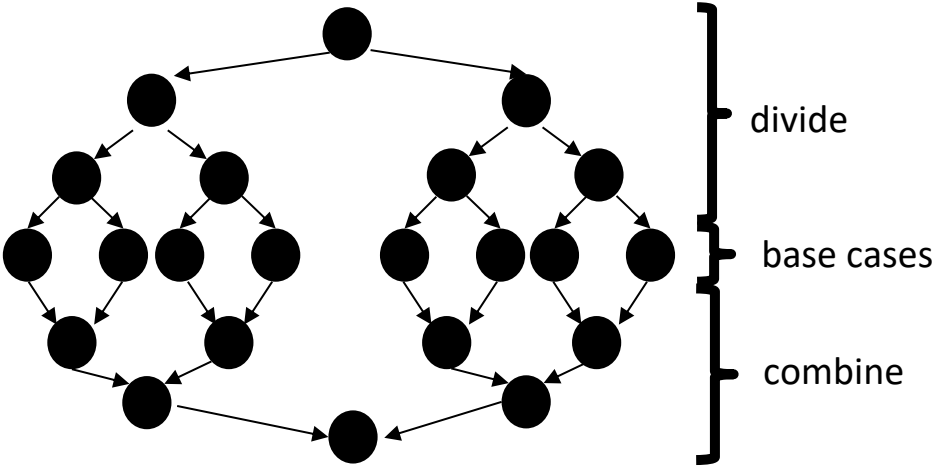
- The execution DAGs are not always this simple
  - *Example*: combining results might so expensive that we parallelize it. Then each node in the *inverted* tree would expand into another set of nodes for that parallel computation

Definitions: Work and Span

- Let T_P be the *running time* if there are **P** *processors* available
- Two important definitions:
  - **Work**: How long it would take with 1 processor (ie, T_1)
    - Just "sequentialize" the recursive forking
    - Cumulative work that all processors must complete
  - **Span**: How long it would take with infinitely many processors (ie, T_∞)
    - The hypothetical ideal; aka "critical path length" or "computational depth"
    - This is the longest "dependence chain" in the computation
    - *Example*: $O(\log n)$ for summing an array
      - Notice how having $>n/2$ processors doesn't reduce the span

Our Simple Examples + Our Definitions

- In this context, the span (T_∞) is:
  - The longest dependence-chain; i.e., longest 'branch' in parallel 'tree'
  - *Example*: O(log n) for summing an array
    - We halve the data down to our sequential cut-off, then add back together
    - 2 * log n steps, O(1) time for each: O(log n)



Work and Span in Fork/Join-style DAGs

- **Span** (T_∞) = sum of runtime of all nodes in the DAG's *most-expensive path*
  - Note: costs are on the nodes not the edges
  - O(log n) for simple maps and reductions

- **Work** (T_1) = sum of runtime of all nodes in the DAG
  - Any topological sort is a legal execution
  - O(n) for simple maps and reductions

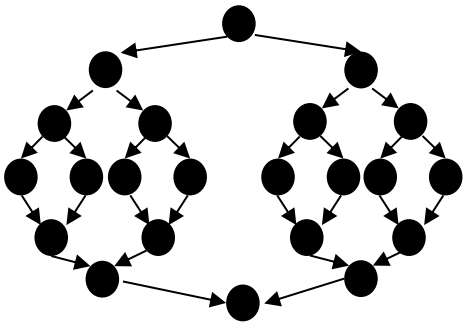
More Definitions: Speed-up and Parallelism

- **Speed-up**, using **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, but hard to get in practice
- **Parallelism**: $\mathbf{T_1 / T_\infty}$
  - Parallelism is the maximum possible speed-up; the point at which adding processors doesn't help
  - That point depends on the span

Parallel algorithms attempt to decrease span without increasing work too much

Obtaining Optimality for T_P

- What is the asymptotically optimal T_P, for any value of **P**?
  - (as usual, we ignore memory-hierarchy issues; i.e. caching)



- We know T_P is greater than or equal to:
  - T_1 / P *(why?)*
  - T_∞ *(why?)*

- So an *asymptotically optimal* execution must be:

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

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

Optimal T_P: Thanks, ForkJoin library!

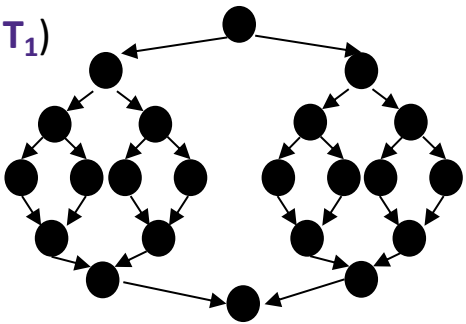
- The ForkJoin library gives an *expected-time guarantee* of asymptotically optimal!
  - "Expected time" because it flips coins when scheduling

- To obtain this guarantee, our job as ForkJoin library users is to make all the nodes in our execution DAG *small-ish* and *approximately equal*

- In exchange, the library-writers:
  - Assign work to avoid **idling**; we can ignore **scheduling** issues
  - Keep constant factors low
  - Give the **expected-time optimal guarantee** (assuming the library user did their job): $T_P = O((T_1 / P) + T_\infty)$

Review: Work and Span

- Let T_P be the *running time* if there are **P** *processors* available
- Two important definitions:
  - **Work**: How long it'd take with 1 processor (ie, T_1)
    - Just "sequentialize" the recursive forking
    - Sum of all nodes in the graph
    - Simple map/reduction:
      - (assuming equal work done in every node and cutoff=1)



  - **Span**: How long it'd take with infinitely many processors (ie, T_∞)
    - Sum of all the nodes *on the longest path* in the graph
    - Simple map/reduction:
      - (assuming equal work done in every node and cutoff=1)

Review: Speed-up, Parallelism, and Optimality

- **Speed-up**, using **P** processors: T_1 / T_P
- **Perfect linear speed-up** occurs when $T_1 / T_P = P$
  - Perfect linear speed-up means doubling **P** halves running time
- **Parallelism**: T_1 / T_∞
  - Maximum possible speed-up; adding processors won't help

- We know T_P MUST BE greater than or equal to:
  - T_1 / P *(why?)*
  - T_∞ *(why?)*

- So an *asymptotically optimal* execution must be:

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

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

And Now for the Good / Bad News …

- In practice, it's common that a program has:
  a) Parts that **parallelize** well:
     - E.g. maps/reduces over arrays and trees
  b) … and parts that **don't parallelize** at all:
     - E.g. reading a linked list
     - E.g. waiting on input
     - E.g. computations where each step needs the results of previous step

- These unparallelizable parts turn out to be a big bottleneck, which brings us to Amdahl's Law …

Amdahl's Law

- Let the work (T_1) be 1 unit of time and **S** be the unparallelizable portion of execution time:

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

- Suppose *perfect linear speed-up* on the parallelizable portion. Then:

$$T_P = S + (1-S)/P$$

- Amdahl's Law states the speed-up with P processors is:

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

- and the parallelism (maximum possible speed-up) is:

$$T_1 / T_\infty = 1 / S$$

Amdahl's Law Example

- Recall: $T_1 = 1 = S + (1-S)$ and $T_P = S + (1-S)/P$

- Suppose: $T_1 = 1/3 + 2/3 = 1$ *(eg, T_1 = 100s = 33s + 67s)*
- Then: $T_P = 33 \text{ sec} + (67 \text{ sec})/P$

  $T_3 = 33 \text{ sec} + (67 \text{ sec})/3 =$

  $T_6 = 33 \text{ sec} + (67 \text{ sec})/6 =$

  $T_{67} = 33 \text{ sec} + (67 \text{ sec})/67 =$

- If 33% of a program is sequential, a billion processors won't give a speedup over 3!!!
- No matter how many processors you use, your speedup is bounded by the sequential portion of the program

Implications of Amdahl's Law

Speedup:	$T_1 / T_P = 1 / (S + (1-S)/P)$
Max Parallelism:	$T_1 / T_\infty = 1 / S$

- In "the good old days" (1980-2005), ~12 years = 100x speedup

- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1. What portion of the program must be parallelizable to get 100x speedup?
  - *For 256 processors to get at least 100x speedup, we need*

    $$100 \leq 1 / (\mathbf{S} + (1-\mathbf{S})/256)$$

  - *Which means* $\mathbf{S} \leq .0061$ *(i.e., 99.4% must be parallelizable)*

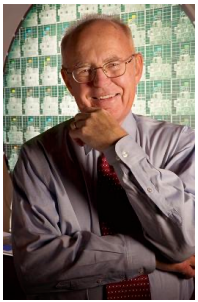
The Challenge Posed by Amdahl's Law

- Amdahl's Law tells us unparallelized parts become a bottleneck very quickly
  - But it *doesn't* tell us additional processors are worthless
- … because we can find new parallel algorithms
  - Some things that seem sequential turn out to be parallelizable
  - Eg: How can we parallelize a 'running sum' array?

input	6	4	16	10	16	15	2	8
output	6	10	26	36	52	67	69	77

- We can also change the problem we're solving
  - Eg: Video games use tons of parallel processors; they are not rendering 10-year-old graphics faster

Moore and Amdahl



- Moore's "Law" is an **observation** about the progress of the semiconductor industry
  - Transistor density doubles roughly every 18 months
- Amdahl's Law is a **mathematical theorem**
  - Diminishing returns of adding more processors
- Both are incredibly important in designing computer systems