CSE 332: Data Structures & Parallelism Lecture 14: Analysis of Fork-Join Parallel Programs

Arthur Liu

Summer 2022

Reduce

It shouldn't be too hard to imagine how to modify our code to:

1. Find the maximum element in an array.
2. Determine if there is an element meeting some property.
3. Find the left-most element satisfying some property.
4. Count the number of elements meeting some property.
5. Check if elements are in sorted order.
6. [And so on…]

In O(log N) !!!

Reduce

You'll do similar problems in section tomorrow.

The key is to describe:

1. How to compute the answer at the cut-off.
2. How to merge the results of two subarrays.

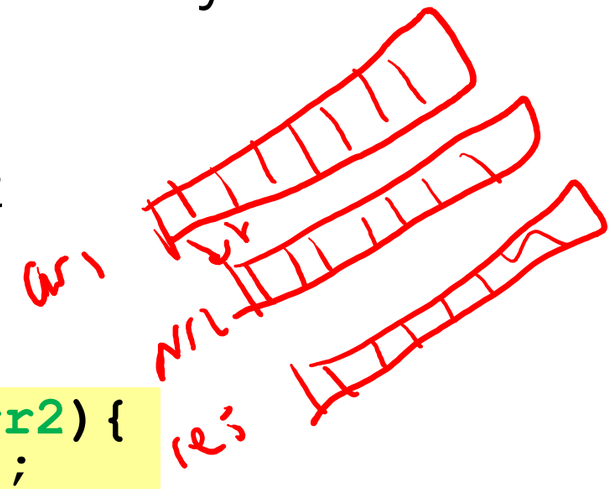
We say parallel code like this "reduces" the array

- We're reducing the arrays to a single item
- Then combining with an **associative** operation.
- e.g. sum, max, leftmost, product, count, or, and, …

Doesn't have to be a single number, could be an object.

Even easier: Maps (Data Parallelism)

- A map operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support



- Canonical example: Vector addition

```
int[] vector_add(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;
}
```

Maps in ForkJoin Framework

RecursiveTask<Integer>

```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool POOL = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  POOL.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

Maps and reductions

Maps and reductions: the “workhorses” of parallel programming

- By far the two most important and common patterns
  - Two more-advanced patterns in next lecture

- Learn to recognize when an algorithm can be written in terms of maps and reductions

- Use maps and reductions to describe (parallel) algorithms

- Programming them becomes “trivial” with a little practice
  - Exactly like sequential for-loops seem second-nature

Map vs reduce in ForkJoin framework

In our examples:

- Reduce:
  - Parallel-sum extended RecursiveTask
  - Result was returned from compute()
- Map:
  - Class extended was RecursiveAction
  - Nothing returned from compute()
  - In the above code, the ‘answer’ array was passed in as a parameter

Analyzing Algorithms: Work and Span

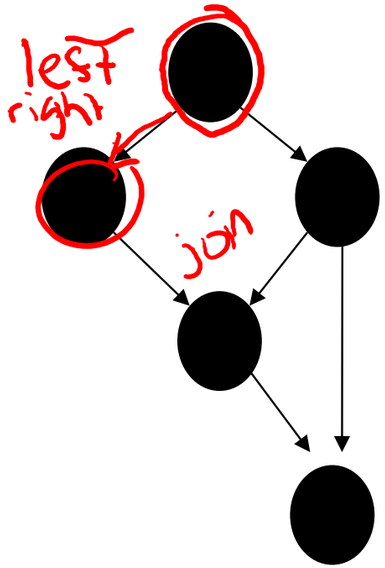
Let T_P be the running time if there are P processors available

Two key measures of run-time:

- Work: How long it would take 1 processor = T_1
  - Just "sequentialize" the recursive forking
  - Cumulative work that all processors must complete
- Span: How long it would take infinity processors = T_∞
  - The hypothetical ideal for parallelization
  - This is the longest "dependence chain" in the computation
  - Example: $O(\texttt{log}\ n)$ 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)

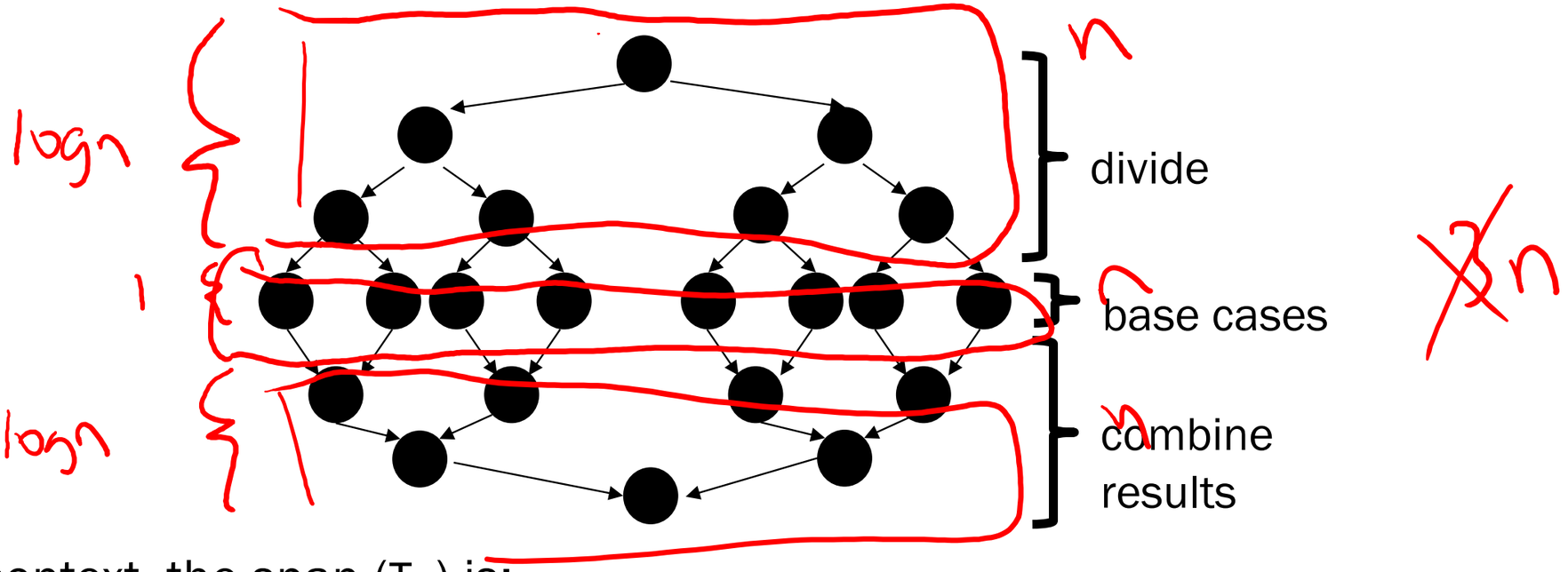
- A program execution using **`fork`** and **`join`** can be seen as a DAG
- [A DAG is a graph that is directed (edges have direction (arrows)), and those arrows do not create a cycle (ability to trace a path that starts and ends at the same node).]
  - **Nodes:** Pieces of work
  - **Edges:** Source 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_∞) 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: T_P = running time if there are **P** processors available

Work = 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 = T_∞ = 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
- $O(\log n)$ for simple maps and reductions

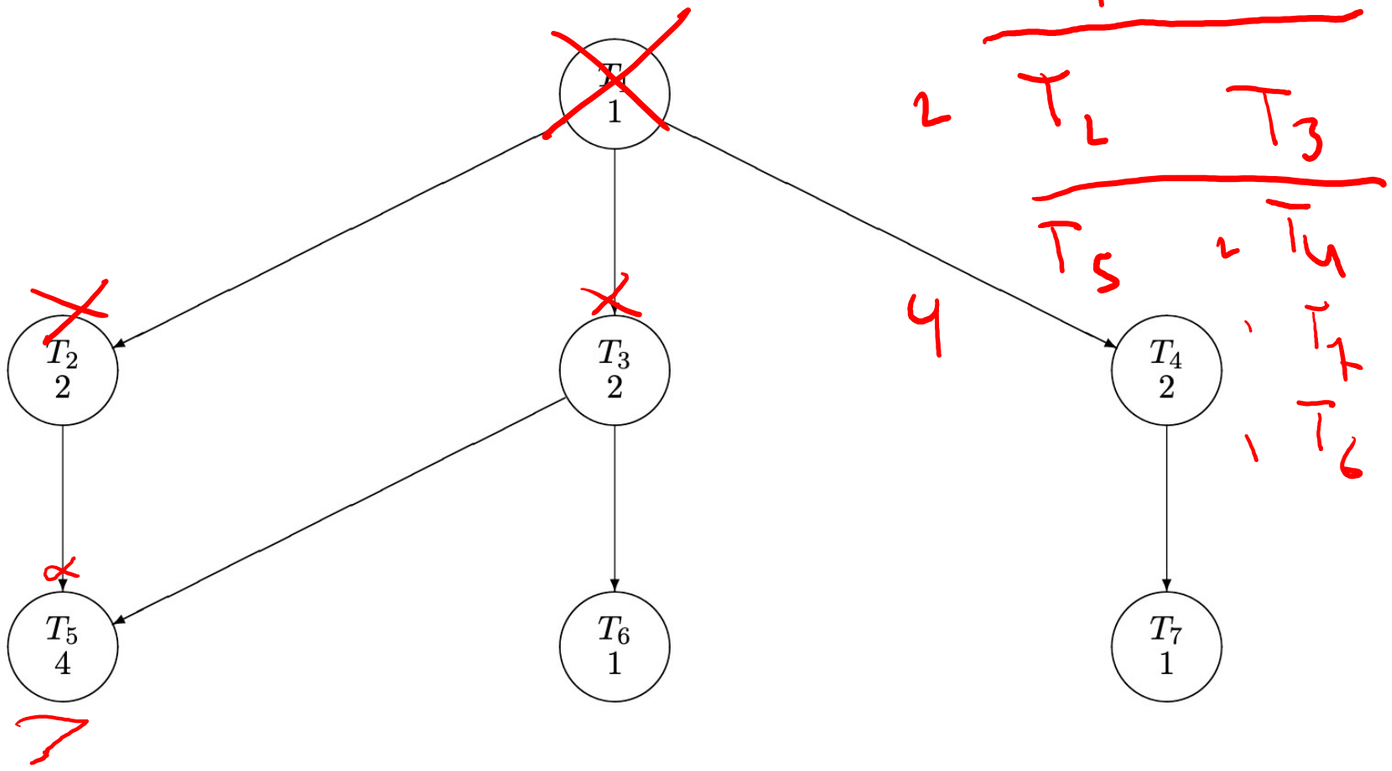
Poll Everywhere

pollev.com/artliu

Consider this graph

The numbers indicate the amount of time it takes for the task to execute

1. What is the work?
2. What is the span?
3. What is the minimum time it takes two processors to complete the tasks?



Definitions

$T_1 = 100$ sec $T_4 = 25$ sec

$\frac{100}{25} = 4$

A couple more terms:

- Speed-up on **P** processors: 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: T_1 / T_∞
  - At some point, adding processors won't help
  - What that point is depends on the span

Optimal T_P: Thanks ForkJoin library!

So we know T_1 and T_∞ but we want T_P (e.g., P=4)

- Ignoring memory-hierarchy issues (caching), T_P can't beat
  - T_1 / P why not?
  - T_∞ why not?

- So an *asymptotically* optimal execution would be:

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

  - 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

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

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

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

Suppose instead:

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

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

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

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

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

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

And the parallelism (infinite processors) is:

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

Amdahl's Law

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

Suppose our program takes 100 seconds.

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

What is the running time with

3 processors $33 + \frac{67}{3} = 55$

6 processors $33 + \frac{67}{6} = 44$

22 processors $33 + \frac{67}{22} = 36$

67 processors $33 + \frac{67}{67} = 34$

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

Amdahl's Law

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

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

The future and Amdahl's Law

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

- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 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?

$$100 \leq \frac{1}{S + \frac{1-S}{256}}$$

[wolframalpha says] $S \leq 0.0061.$

$100 \leq \frac{1}{S}$

$S \leq \frac{1}{100}$