Teaching Parallel Algorithms Using the Binary-Forking Model

Yihan Sun, EduPar'24

Guy Blelloch
Carnegie Mellon University

Yan Gu
UC Riverside

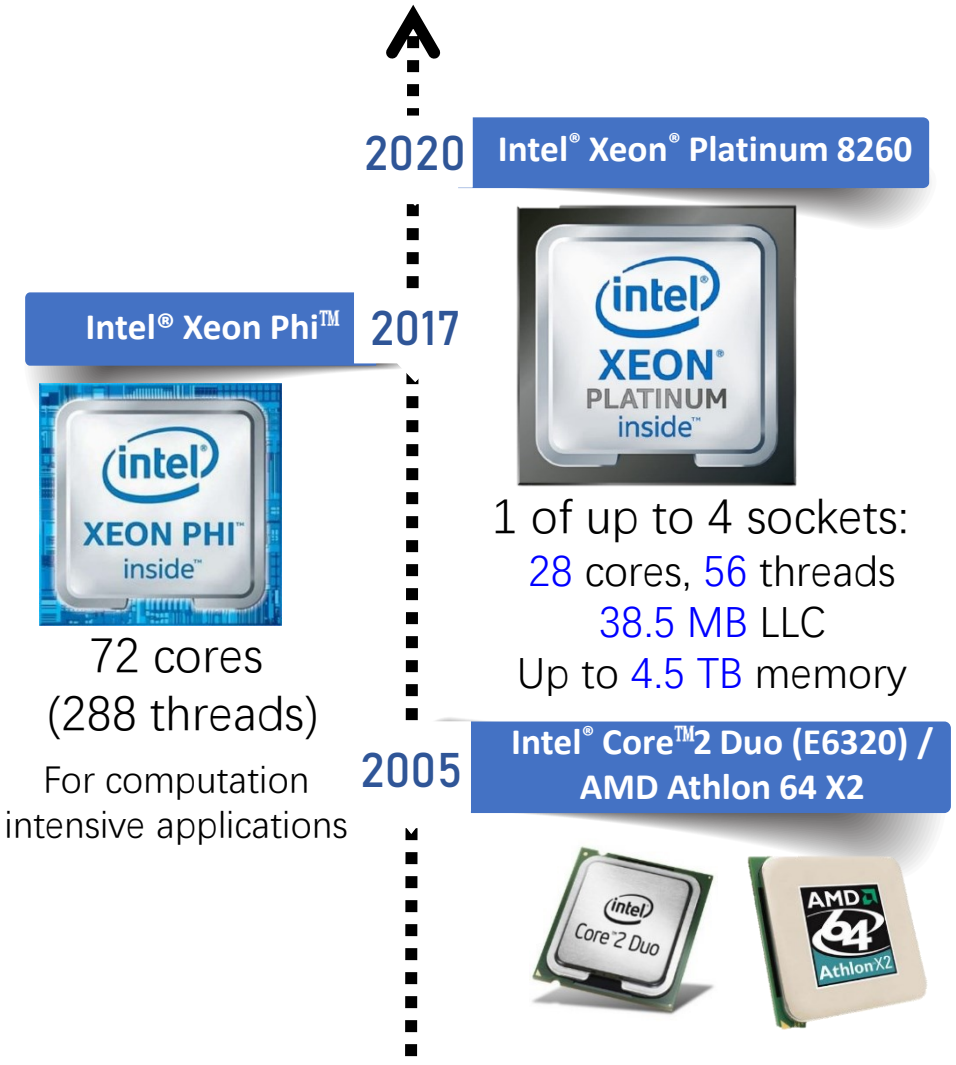
Yihan Sun
UC Riverside

Parallelism is everywhere!

- Parallelism moved to the mainstream from early this century…
- … and now it has become popular & powerful

- Almost all devices are multicore
- Running code in parallel will be the default setting in the future!

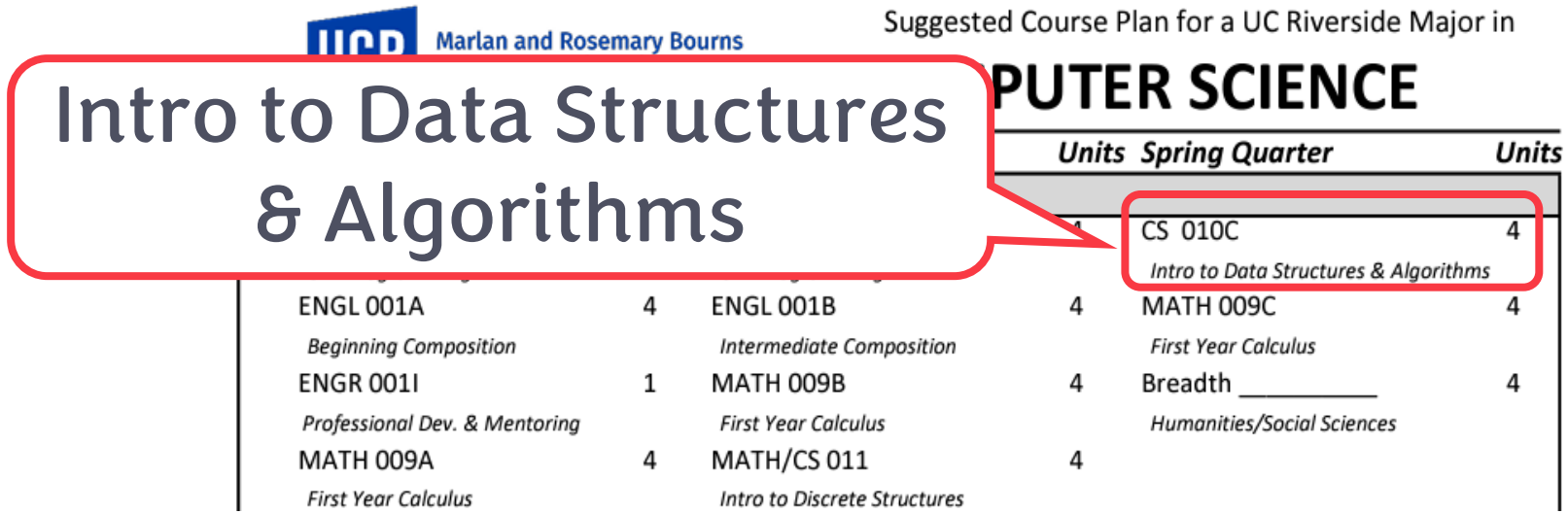
- It's important to introduce parallelism to young students … in the same way as sequential computing today



Introducing parallelism in classes?

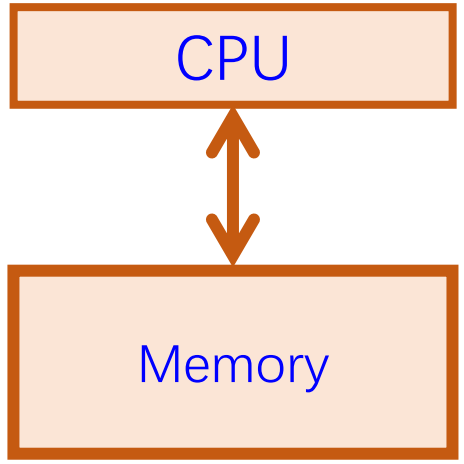
- **CS curriculum usually starts with basic programming + algorithms (theory)**

- **Sequential algorithms**
  - Very successful and popularized – not hard even for high school students
  - Theory is well-accepted: analyzing time complexity in big-O
  - Theoretically-efficient solutions are highly practical, easy to implement, simple to understand



What makes sequential algorithms successful?

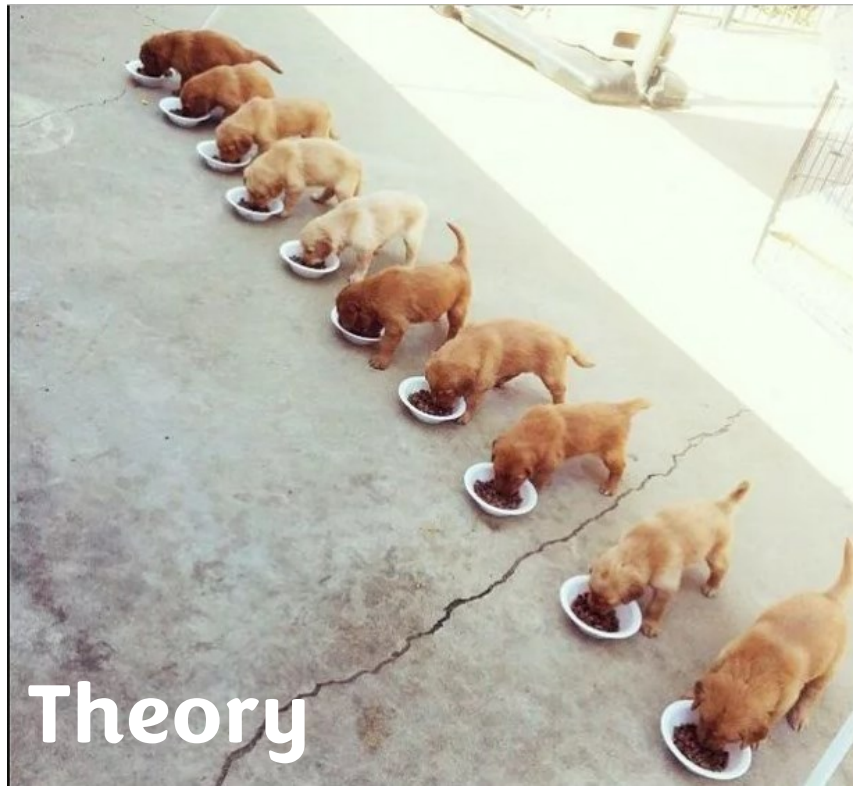
- **Random Access Machine (RAM) Model & time complexity**
  - Simple & elegant to understand
  - Close to the actual hardware
  - Reasonably accurate in estimating running time



- **Bridging theory and practice!**
  - Abstracts hardware (in practice) into a model – study advanced questions (in theory) on this model
  - Design algorithms (in theory) on this model – easily translate a RAM algorithm to actual code and get “predictable” performance (in practice)

- **But in parallel?**

Parallelism: theory and practice



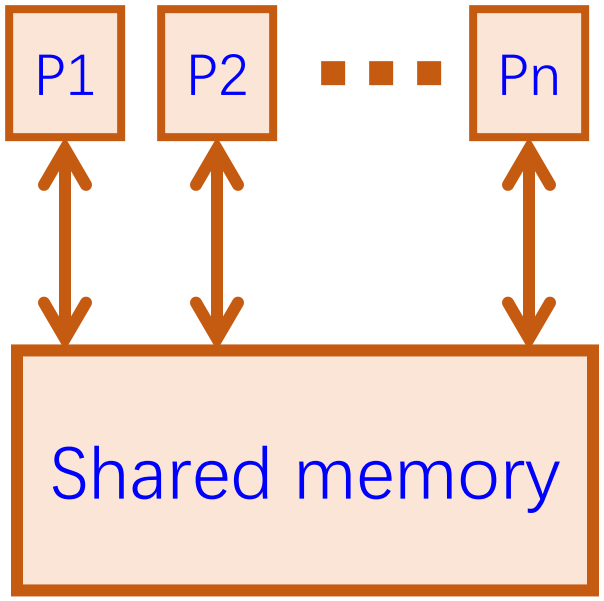
(Pictures from 9gag.com)

The Parallel RAM (PRAM) model

- P processors share a common memory
- Every processor acts like a RAM

- Proposed in the 70s [Fortune and Wyllie'78]
- Widely used in research – thousands of papers, a very hot topic during 80s-90s, many classic and great ideas

- The most important cost measure is the PRAM time – the longest time needed among all processors



PRAM
(~1980)

Is PRAM still the best model for teaching parallel algorithms today?

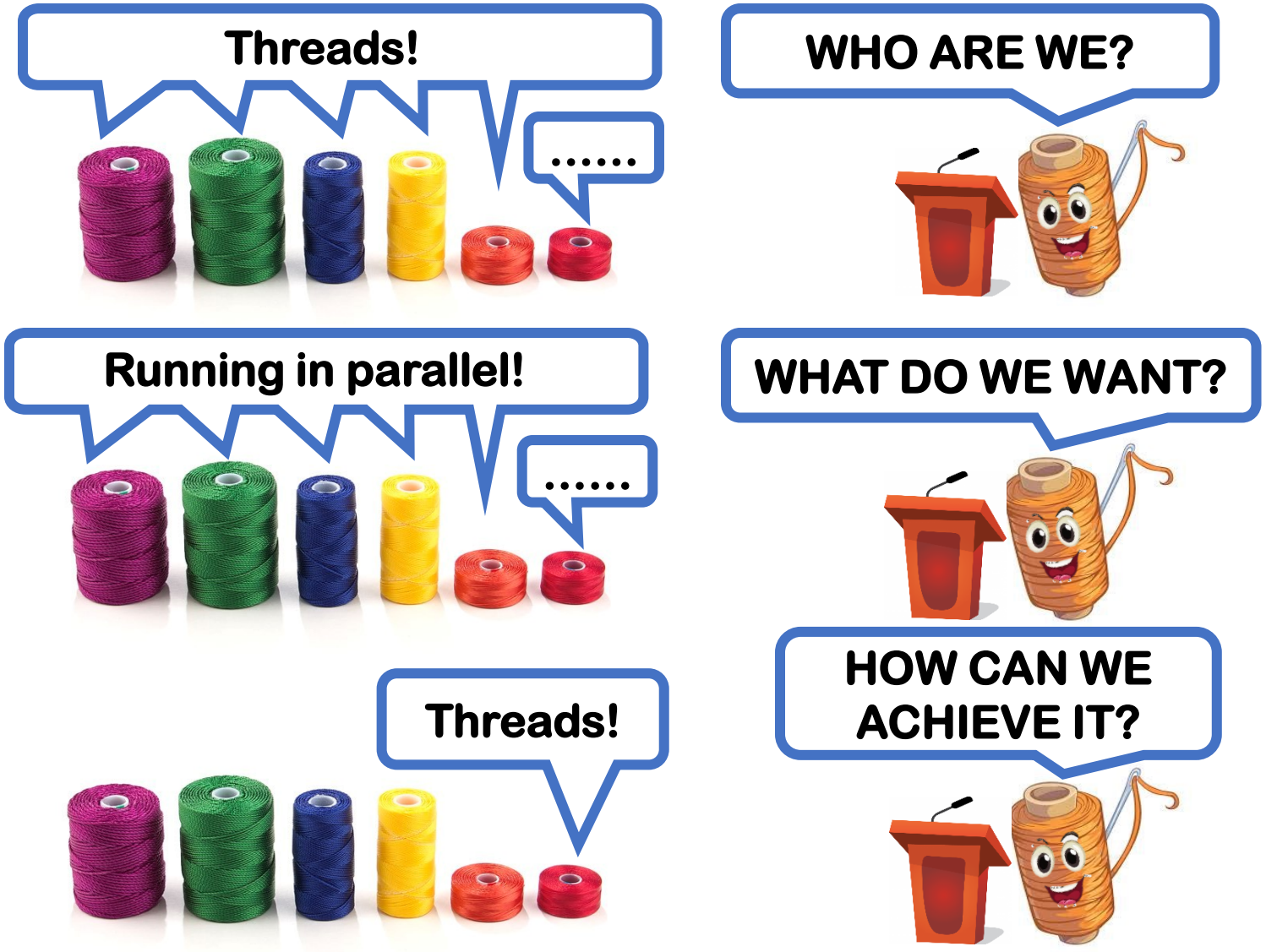
PRAM differs from modern multicore machine in a few important aspects…

- Usually assumes polynomial number of processors
- … and optimizes PRAM time, often overlooking total "work"
- … and assumes processors to be highly synchronized (run in lockstep)

[redacted] The main contribution of the present paper is [redacted] an EREW PRAM algorithm for the problem that runs in $O(n \log n)$ time and $O(n^2)$ space using n processors.

[redacted] This algorithm runs in $O(\log m)$ time using m processors and $O(m)$ space on an EREW PRAM. [redacted]

However, modern architecture is highly asynchronous



Real architecture: processor rates vary significantly due to

- cache effects
- processor pipelines
- ILP
- vectorization
- branch prediction
- hyper-threading
- overclocking
- interrupts
- .....

As you can see, threads are highly asynchronous.

Is PRAM still the best model for teaching parallel algorithms today?

PRAM differs from modern multicore machine in a few important aspects…

- Usually assumes polynomial number of processors
- … and optimizes PRAM time, often overlooking total "work"
- … and assumes processors to be highly synchronized (run in lockstep)

As a result, PRAM algorithms usually need non-trivial adaption and additional engineering effort to implement to get high-performance code

Is PRAM still the best model for teaching parallel algorithms today?

As a result, PRAM algorithms usually need non-trivial adaption and additional engineering effort to implement to get high-performance code

- **… which can be discouraging for students to learn theory about parallel algorithms ☹**
- **… and even for researchers to come up with and implement theoretically-efficient solutions in parallel ☹**

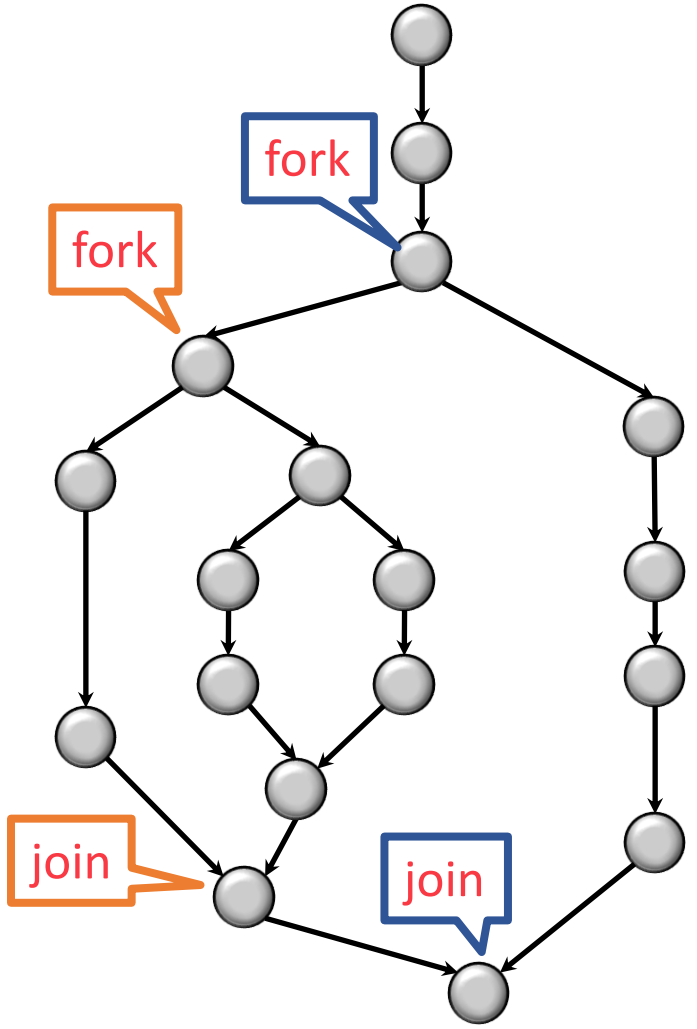
The binary-forking model

- **We formalized the binary-forking model in 2020** [Blelloch et al., SPAA'20]
  - Won outstanding paper award in SPAA'20
  - Built based on multithreaded models (already widely-used in various settings!)
  - The research paper includes simulation results about binary-forking and various existing modes, and new algorithms to achieve optimal bounds

- **… and the model is also extremely friendly for classroom teaching!**
  - Close to how (many) parallel programming tools are implemented
  - Easy to implement in code and get high performance
  - Simple cost analysis

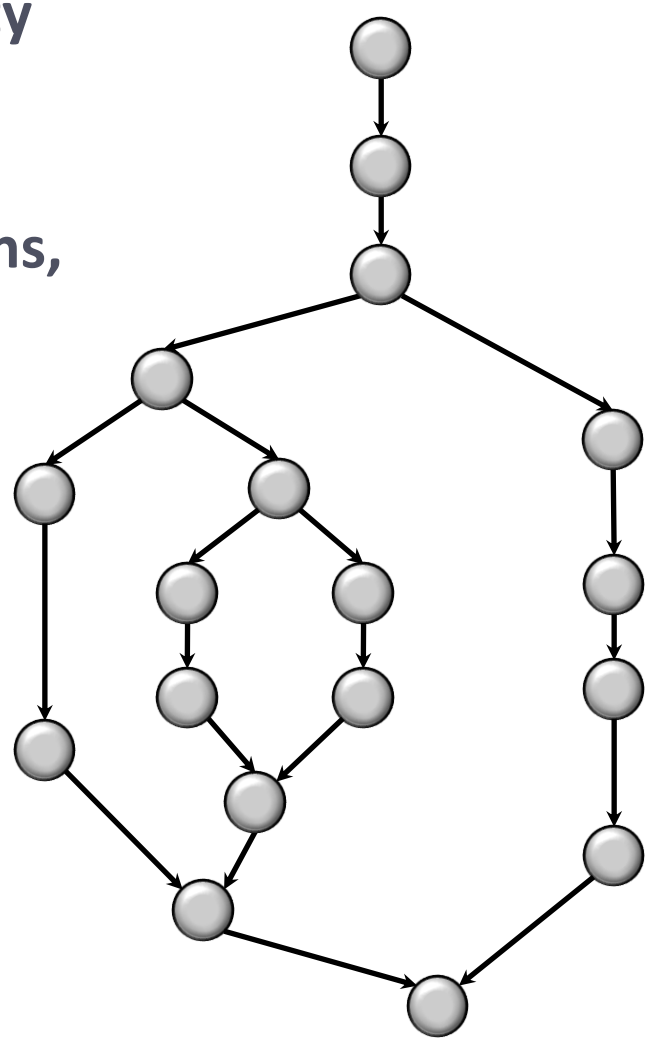
Binary Fork-Join Model

- Multiple processors share memory. A computation starts with one thread.
- The same RAM operations + fork/join
- `Fork:` creates two subtasks (threads) that run in parallel (and asynchronously)
- Fork-join can be nested
- After they finish, they `join` and continue
- Assumes race-free
  - Two logically parallel processors won't access the same memory location with at least one write
  - Or allows some atomic operations such as `Test&Set`, `Compare&Swap`, and `Fetch&Add`



Work-Span Analysis

- **Work W : total # of operations, sequential time complexity**
- **Span/depth D: the longest dependence chain of operations, similar to PRAM time if you have an infinite number of processors**
- **"Running time": using the randomized work-stealing scheduler, on P processors, the running time is $W/P + O(D)$ with high probability**



The binary fork-join model is based on multithreaded models [BL99, ABPO1]

- A class of models
- The computational models for many recent parallel algorithms (a short list: [ABB02, AFL+14, BFGS11, BG04, BGS10, BR98, BGSS20, BST12, BL99, CGTT17, CRSB13, CR17, DST16, TYK+15])
- The parallel model in [CLRS, 3rd and 4th edition]
- Supported by existing libraries such as Cilk/Open Cilk, OpenMP, TBB, Java Fork-Join, X10, TPL, Habanero

Basic Building Blocks to Design Parallel algorithms in the Binary-Forking Model

Design Parallel Algorithms in the Binary-Forking Model

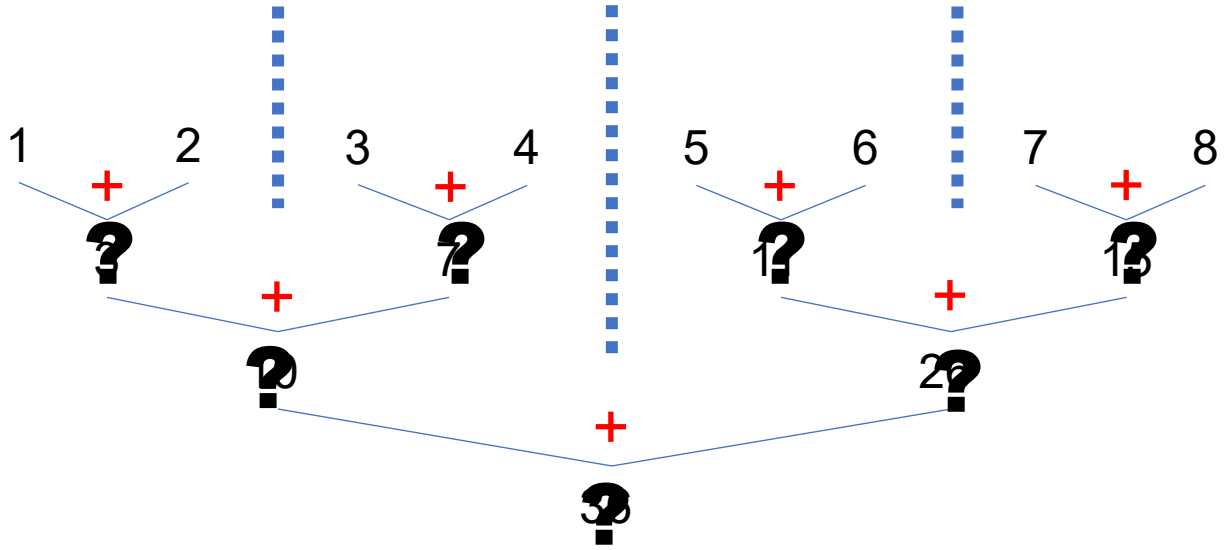
Simple RAM operations plus:

- **parallel_do (S1, S2): two statements running in parallel.**
  - Synchronize (join) when they both finish

- **parallel_for (i = 1 to n) f(i): function calls for all i running in parallel**
  - Synchronize (join) when they all finish
  - parallel_for can be simulated by log(n) levels of forks.

Reduce using divide-and-conquer

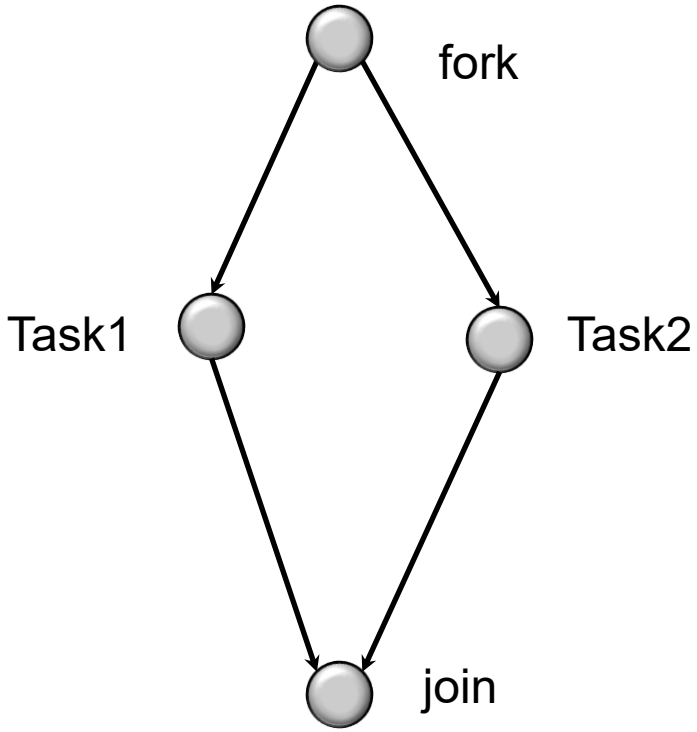
- Compute the sum (reduce) of all values in an array




```
reduce(A, n) {
    if (n == 1) return A[0];
    parallel_do {
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    }
    return L+R; }
```

Analysis: similar to sequential algorithms

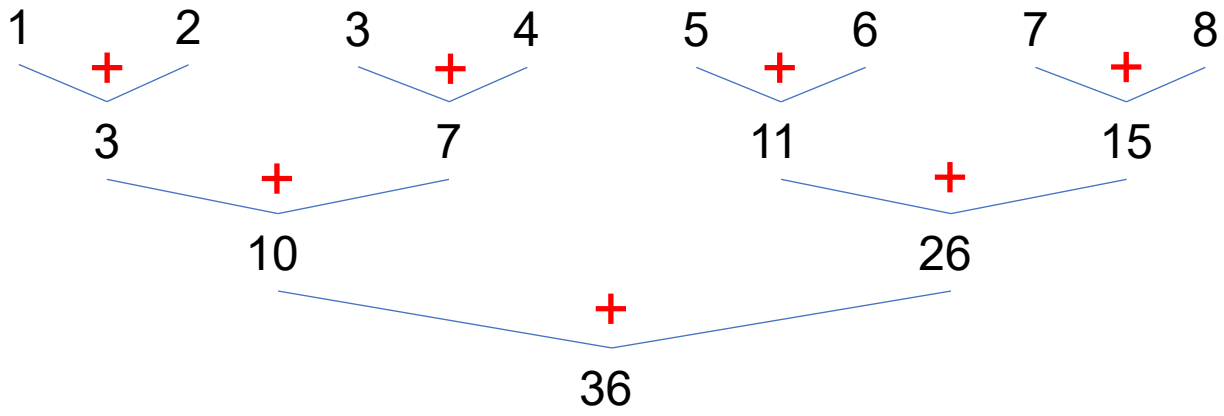
- When we see a fork-join:
  - Fork
    - Task1
    - Task2
  - Join
- Work = work of Task1 + work of Task2 + O(1)
- Span = max(span of Task1, span of Task2) +O(1)



	Work	Span
Executed sequentially	$O(1) + W_1 + W_2$	$O(1) + S_1 + S_2$
Executed in parallel	$O(1) + W_1 + W_2$	$O(1) + \max(S_1, S_2)$

Reduce using divide-and-conquer

- Compute the sum (reduce) of all values in an array




```
reduce(A, n) {
    if (n == 1) return A[0];
    parallel_do {
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    }
    return L+R; }
```

Recurrences:

Work: $W(n) = 2W(n/2) + O(1) \Rightarrow W(n) = \Theta(n)$

Span: $D(n) = \max(D(n/2), D(n/2)) + O(1) \Rightarrow D(n) = \Theta(\log n)$

Binary fork-join model

- Simple for theoretical analysis
- Simple for programming – almost exactly the code!
- Most of the state-of-the-art parallel languages support similar semantics and they know what to do with “fork” and “join”

```
reduce(A, n) {
    if (n == 1) return A[0];
    parallel_do {
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    }
    return L+R; }
```

```
1  int reduce(int* A, int n) {
2    if (n = 1) return A[0];
3    int x, y;
4    cilk_scope {
5      x = cilk_spawn reduce(A, n/2);
6      y = reduce(A + n/2, n - n/2); }
7    return x + y;
8  }
```

Other simple building blocks

Algorithm	Work	Span
Reduce (sum)	$O(n)$	$O(\log n)$
Scan (prefix sum)	$O(n)$	$O(\log n)$
Filter (pack)	$O(n)$	$O(\log n)$
Merge	$O(n)$	$O(\log n)$
Merge sort	$O(n \log n)$	$O(\log^2 n)$
Partition	$O(n)$	$O(\log n)$
Quicksort	$O(n \log n)$ (in expectation)	$O(\log^2 n)$ (with high probability)

If you want to include it in a regular algorithm class...

- **1-2 weeks of classes of overview of parallel algorithms**
- **Has been included in two UCR classes**
  - CS141: upper-level required undergrad algorithm class
  - CS142: “algorithm engineering”: tech elective class
  - (Also used in CMU 15-853 “algorithms in the real-world”)
- **CLRS Chapter 27 (fourth edition):**
  - The model (Fork-join, work, span, parallelism, etc.), race conditions, Matrix multiplication / Merge and merge sort
  - Various practice problems covering other parallel algorithms such as reduce and scan

Other topics to consider for a more advanced parallel algorithm class in 10-16 weeks…

- **Other building blocks:** list contraction, tree contraction, other sorting algorithms, random permutation, …
- **Data structures:** Hash tables, binary search trees, …
- **Graph algorithms:** connectivity, single-source shortest paths, minimum spanning tree, maximal independent set, strongly connected component, …
- **Concurrency:** atomic operations, linearizability, lock/wait-freedom, etc.
- **Scheduling results:** greedy scheduler, randomized work-stealing, etc.

- **In the paper we provided citations to the results that we think are classroom-ready, most of them have a theoretically-efficient solution that can be implemented without much coding effort & have open-source code available**

Other topics to consider for a more advanced parallel algorithm class in 10-16 weeks…

- Parallel computational geometry
- Parallel I/O Efficient algorithms
- Concurrent data structures
- Architecture-related topics
- …

- In the paper we provided citations to the results that we think are classroom-ready, most of them have a theoretically-efficient solution that can be implemented without much coding effort & have open-source code available
- A full algorithm class in the parallel setting have been taught in CS214 (graduate) at UCR and also a sophomore class at CMU

Example Assignments

- **Quicksort / mergesort**
  - A simple and fundamental problem. The codes can be compared to STL sort and show impressive speedup
  - Involves important building blocks such as scan, partition, merge, etc.
  - A few interesting engineering tricks can be explored

- **Breadth-First Search**
  - Fundamental graph processing problem; involves important building blocks such as flatten, pack, atomic operations, etc.
  - Can be tested on real-world graphs. Very different performance on different graph types
  - A few interesting optimizations can be integrated (many relevant research results)

Example Assignments

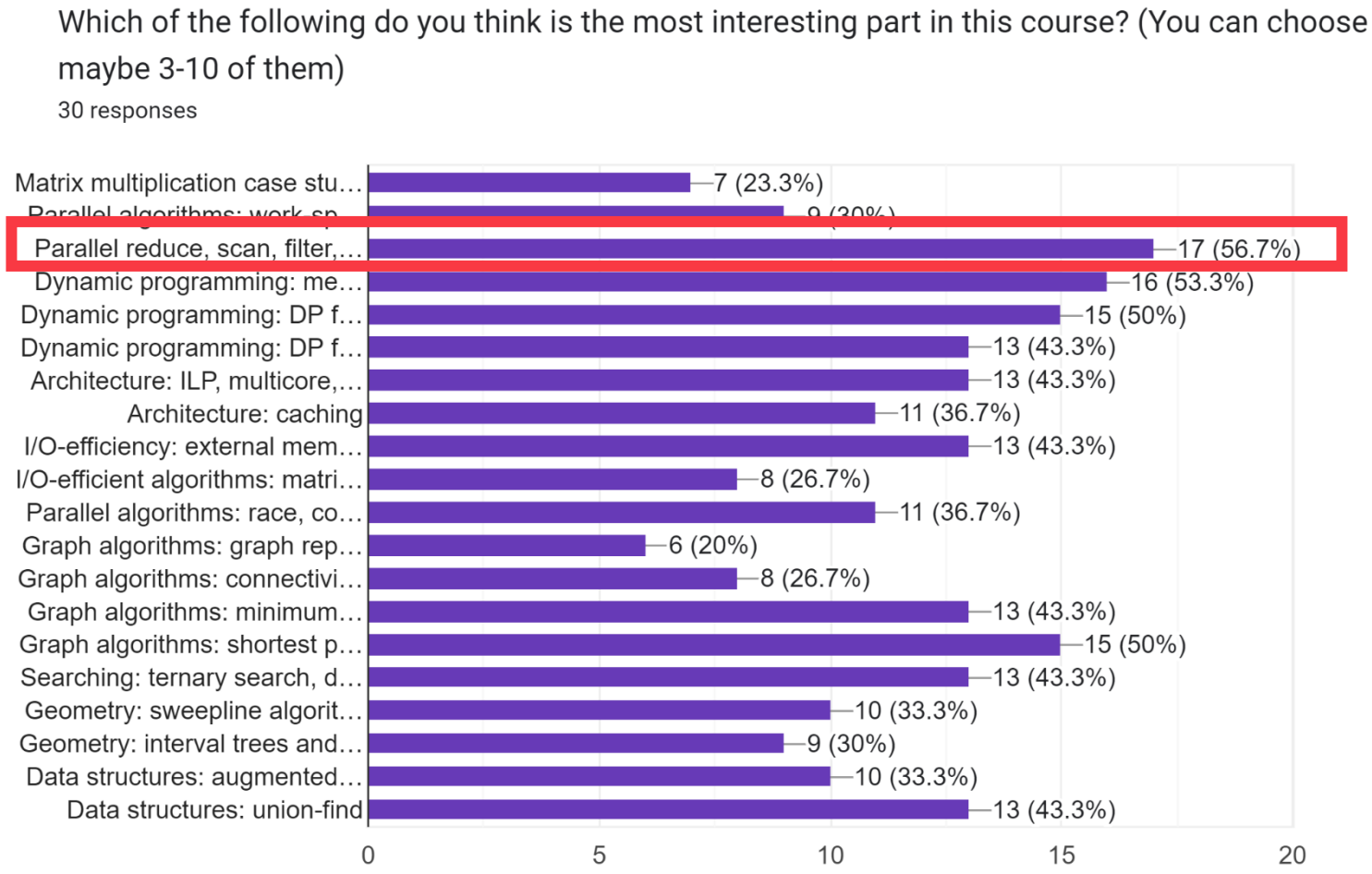
Other topics

- Parallel search tree (binary tree / B-tree)
- Random permutation / list ranking
- Minimum spanning tree
- Graph connectivity
- …

- (Almost all topics mentioned previously have a simple version for implementation with reasonably good performance)

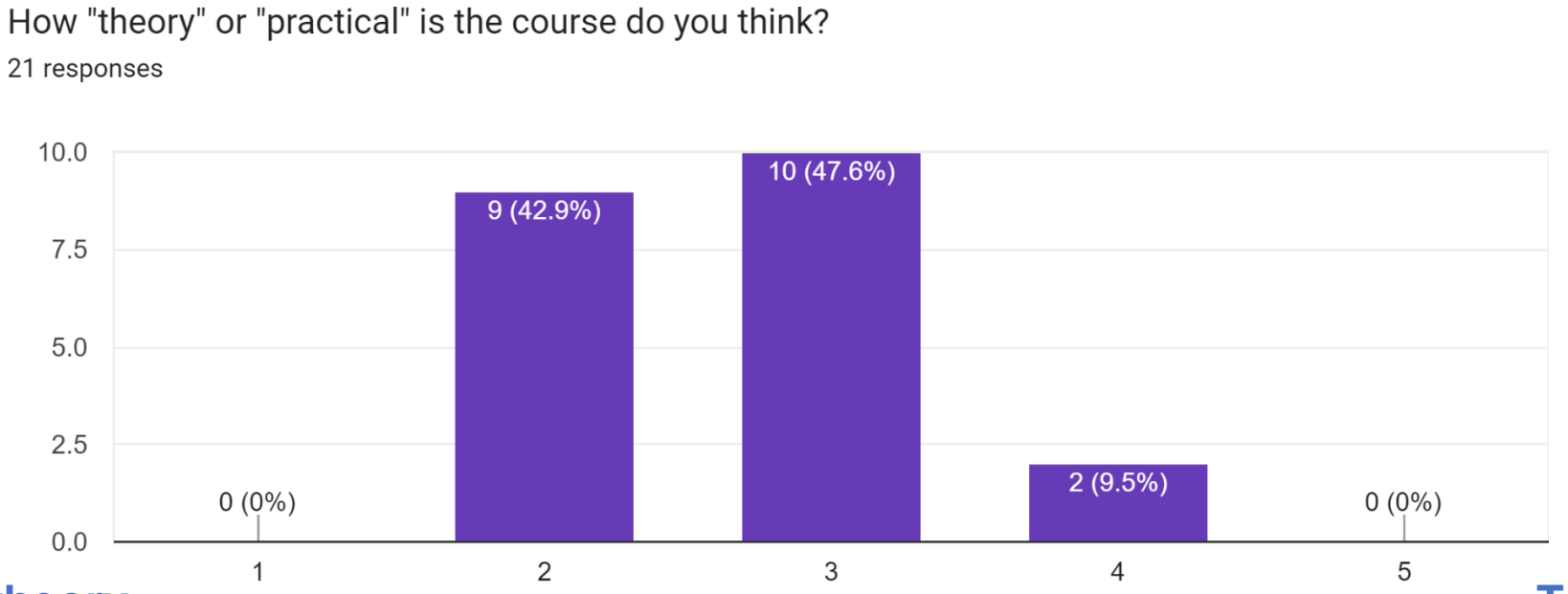
Student evaluation

- **In CS142 (algorithm engineering, upper-level undergrad class) Winter'23 at UCR, parallelism has been covered as one of the topics to improve performance**
  - Most students have no background in parallelism
  - Students were asked to choose 3-10 most interesting topics from ~20 topics covered in class.
  - The topic on parallel reduce/scan/filter/quicksort is the **most-liked topic**! (56.7%)



Student evaluation

- **In CS214 (parallel algorithms, entry-level graduate theory class) Fall'23 at UCR, all analysis/algorithms are based on the binary-forking model**
  - Parallelism is still new to most of the students
  - Students were asked to choose 1-5 from "too much theory" to "too much coding". All answers were in 2-4; 47.6% chose 3.



Too much theory

Too much coding

Summary

- If you want to add some theoretical components to your class about parallelism, binary-forking is a good choice!

- Simple cost analysis!
- Close to real hardware!
- Easy to convert to code!
- Many recent results are available on fundamental problems!

- You can also choose to add it as a brief introduction for parallelism in regular algorithm/theory classes

Teaching & Building Research Community for Software Performance Engineering (SPE)

- Fill in the short survey (30 seconds) to let us know you are interested!

Summary

- If you want to add some theoretical components to your class about parallelism, binary-forking is a good choice!

(SPE Survey)

- Simple cost analysis!
- Close to real hardware!
- Easy to convert to code!
- Many recent results are available on fundamental problems!

- You can also choose to add it as a brief introduction for parallelism in regular algorithm/theory classes