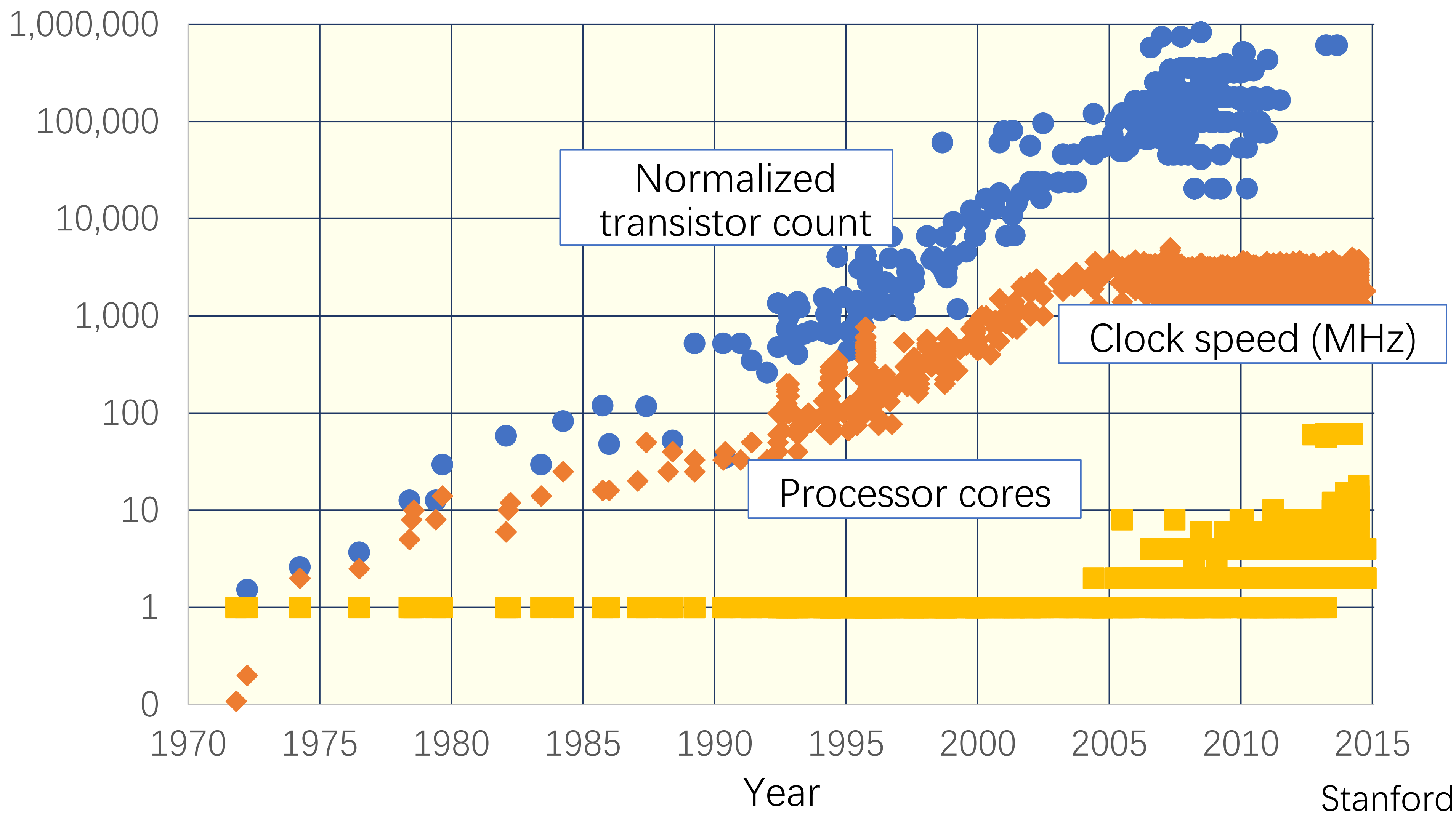
Analysis of Work-Stealing and Parallel Cache Complexity

Joint work with Zachary Napier and Yihan Sun

Yan Gu
UC Riverside
Jan 12, 2022

Technology Scaling

- **Nowadays, it's almost impossible to find a single-core processor**

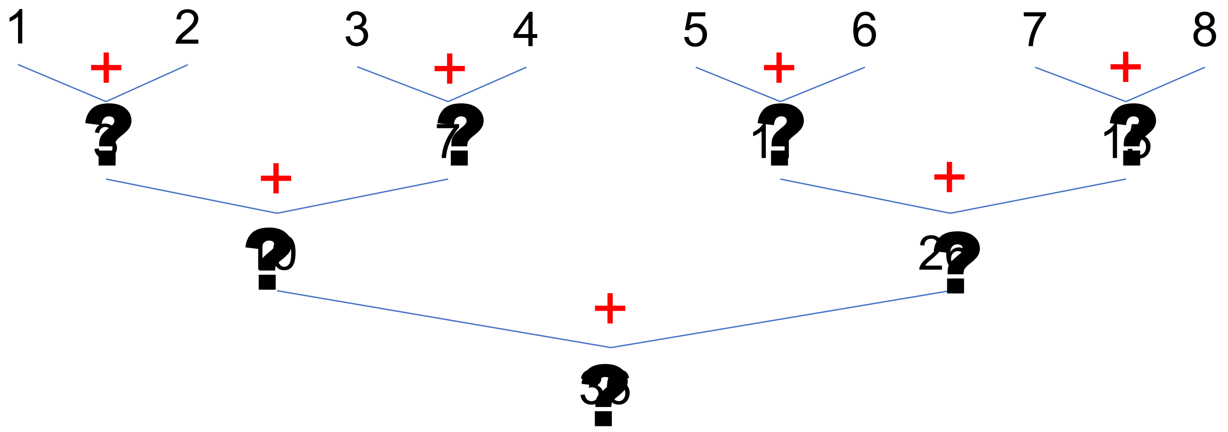


Stanford's CPU DB [DKM12]

Technology Scaling

- **Nowadays, it's almost impossible to find a single-core processor**
- **It is of great importance to understand parallelism and teach it in our CS curriculum**
- **The simplest form of parallelism is multicore CPU with shared-memory**
  - We should not consider each processor as independent distributed node, and program like MPI
  - Instead, we should base on a better abstraction that hides low-level details to algorithm designers and programmers

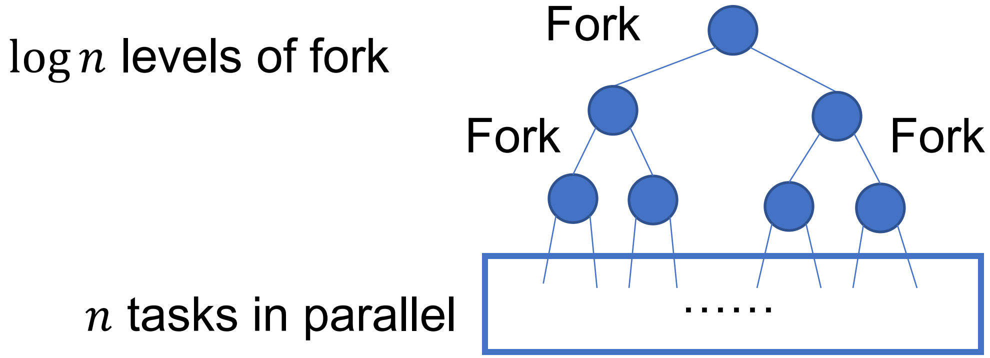
Fork-join parallelism

- Toy example: compute the sum (reduce) of all values in an array
- 


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

- Stay algorithmically: identify parallelism, without worrying any system-level concerns

Binary fork-join model



- Computation starts from one thread
- A thread can perform operation in standard RAM (arithmetic operation, memory access, …), or
  - Fork: start a new thread working on the next statement
  - Join: previous forked processors synchronize here
  - Parallel for: can be simulated by using $O(\log n)$ spawns, perform the computation of the for loops in parallel, and have a sync at the end
- No concurrent writes to the same memory location or needs to be atomic

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

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

Binary fork-join model

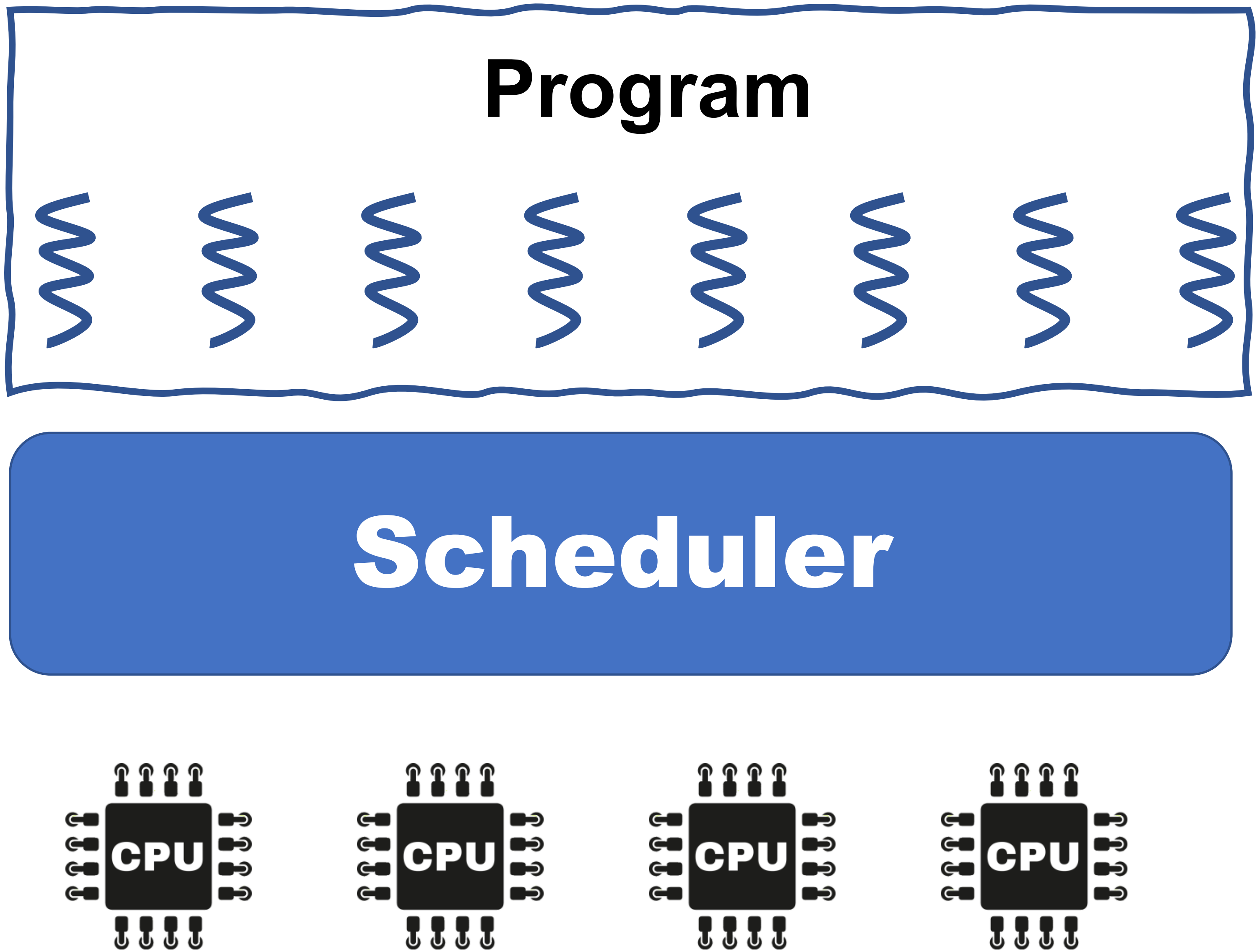
- Simple for theoretical analysis – we'll see in a while
- Simple for programming – almost exactly the code!

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

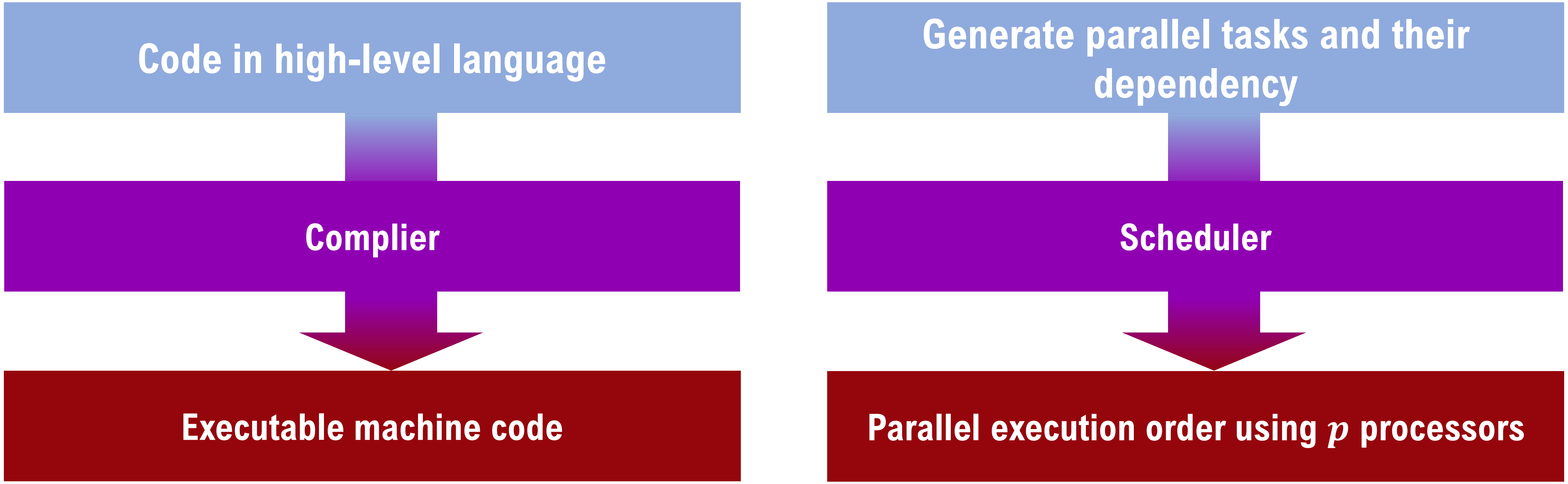
```
#include <iostream>
#include <cstdio>
#include <stdlib.h>
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
using namespace std;

int reduce(int* A, int n) {
    if (n == 1) return A[0];
    int L, R;
    L = cilk_spawn reduce(A, n/2);
    R = reduce(A+n/2, n-n/2);
    cilk_sync;
    return L+R;
}

int main() {
    int n = atoi(argv[1]);
    int* A = new int[n];
    cilk_for (int i = 0; i < n; i++) A[i] = i;
    cout << reduce(A, n) << endl;

    return 0;
}
```

- Other variants of this version available in [BFGS20]

Scheduler: map tasks to processors

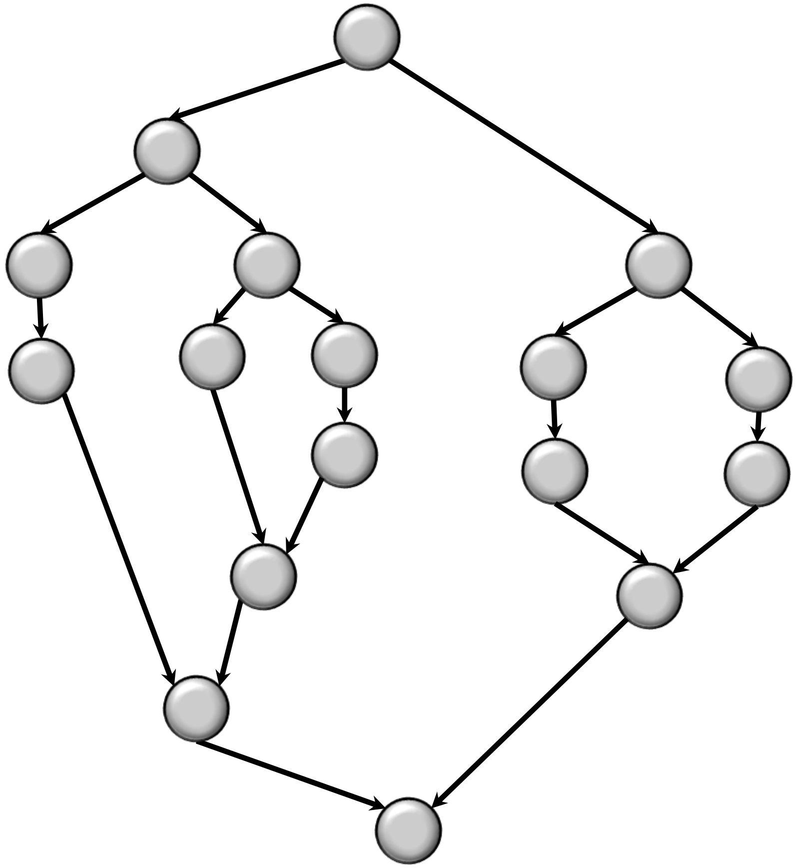


A scheduler plays the role of a compiler for the sequential code that hides all low-level details for algorithm designers



The scheduling problem

- Given a DAG that each node (corresponding to an instruction) has a constant fan-in and fan-out, the scheduler maps each node to one of the P processors, and minimizes the total idle time for all processors

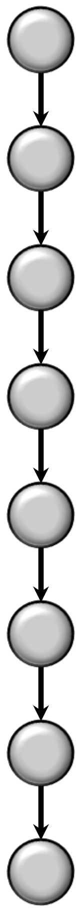


The scheduling problem

- Given a DAG that each node (corresponding to an instruction) has a constant fan-in and fan-out, the scheduler maps each node to one of the P processors, and minimizes the total idle time for all processors

- The best worst-case total idle time you can hope for: $(P-1)D$, where P is the number of processors, and D is the longest path length in the DAG

- The famous "randomized work-stealing (RWS) scheduler" achieves $O(PD)$ idle time whp [BlumofeLeiserson99], while achieve good performance in practice



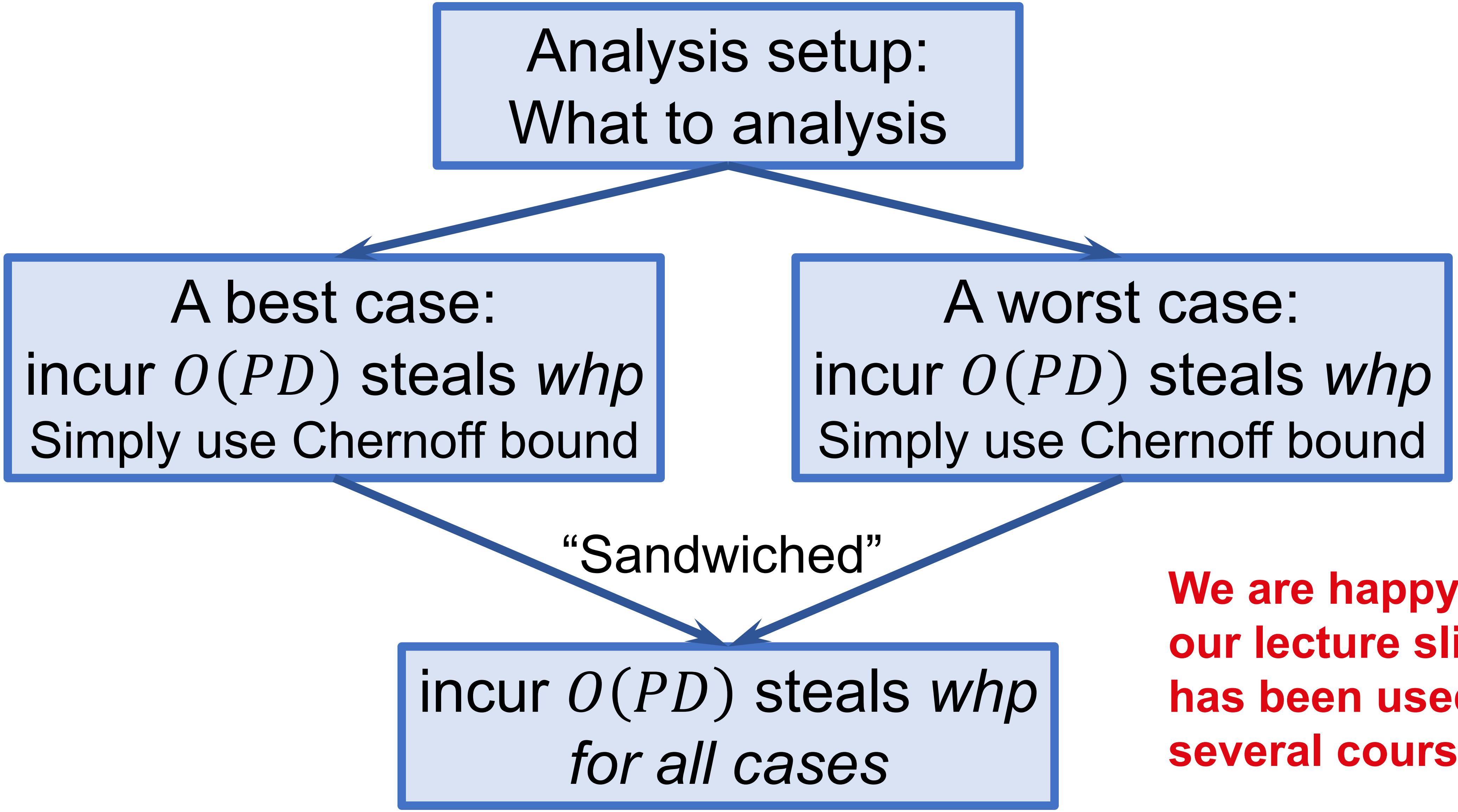
What we studied in this paper

- The analysis of the RWS scheduler is quite compilated, and most existing ones require mapping a potential function to each node in the DAG (hence most existing parallel algorithm courses treat it as a black box)
  - In this paper we showed a simplified analysis that (1) requires no potential function, (2) applies to a more general asynchronized setting, (3) separate math from the main idea, and (4) only uses Chernoff bound in a simple form

- For parallel I/O costs, there exists no tight analysis for many classic algorithms using fork-join parallelism and RWS scheduler
  - In this paper, we show a framework (based on [BlellochGu20]) to derive much tighter bounds for 10 classic problems on algebra and dynamic programming

A very brief proof outline for the RWS scheduler

Proof outline for work-stealing scheduler

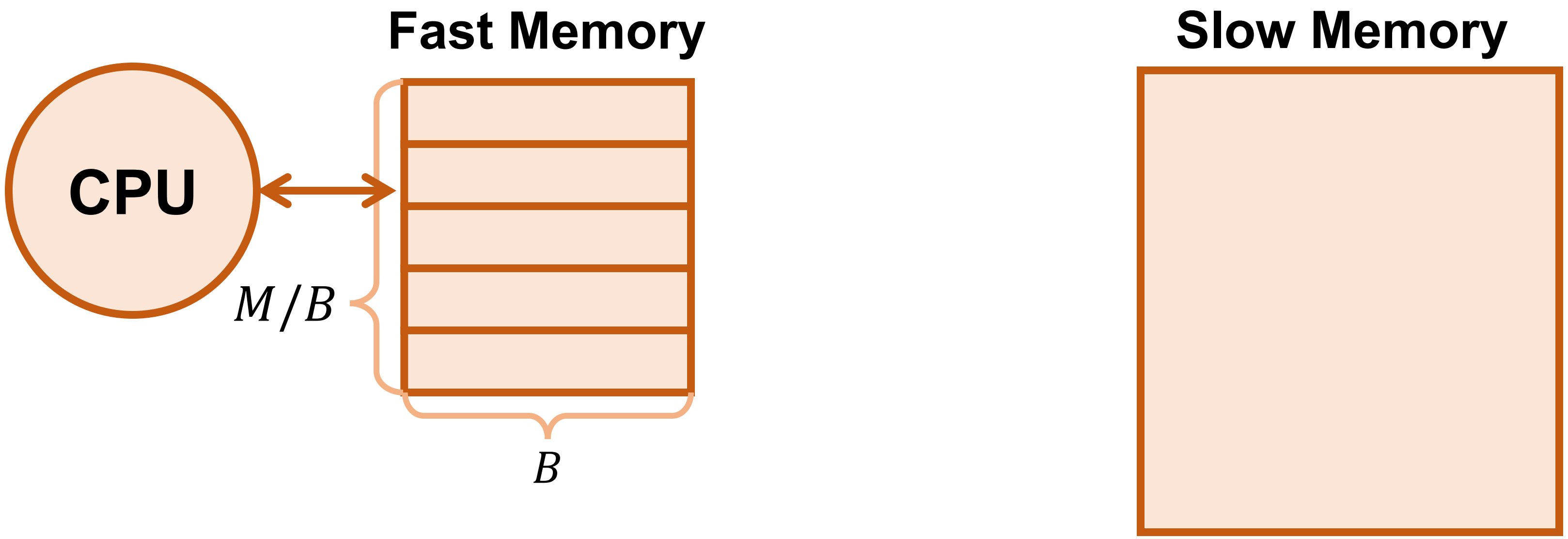


We are happy to share our lecture slides that has been used in several courses

Analyzing parallel I/O complexity

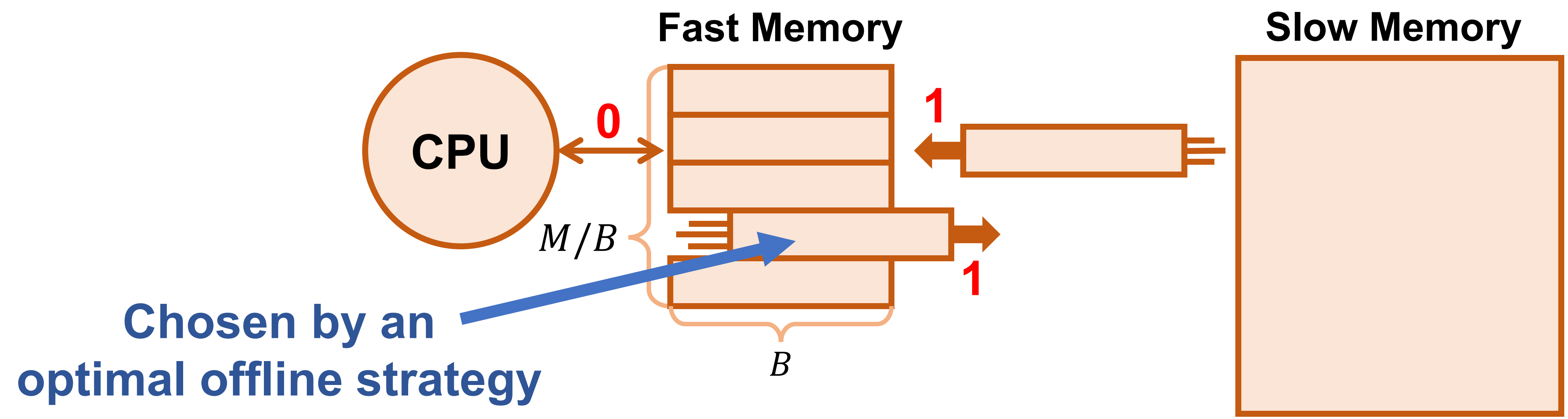
The (sequential) I/O Model (External Memory-, Ideal Cache-) [AV88], [FLPR12]

- **Two-level memory hierarchy:**
  - A small memory (fast memory, cache) of fixed size M
  - A large memory (slow memory) of unbounded size
- **Both are partitioned into blocks of size B**
- **Instructions can only apply to data in primary memory, and are free**



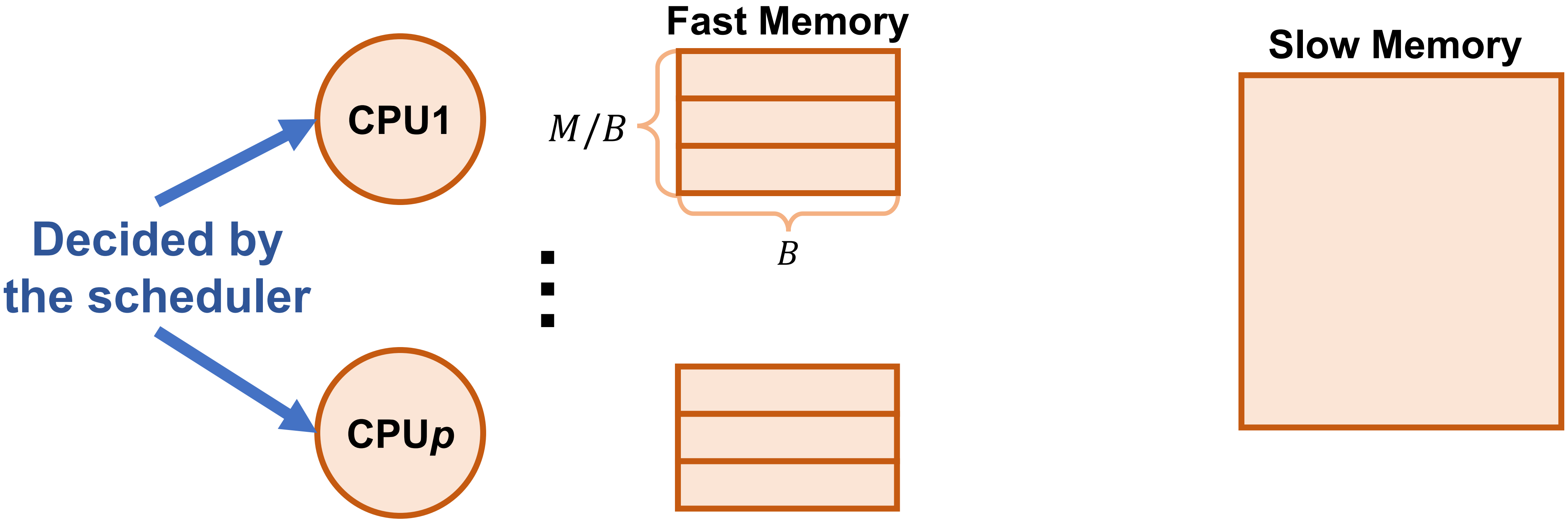
The (sequential) I/O Model (External Memory-, Ideal Cache-) [AV88], [FLPR12]

- We assume the cache is fully associative, and the it takes unit cost to load and evict a pair of blocks
- The complexity of an algorithm on the I/O model (I/O complexity) assumes an optimal cache replacement policy



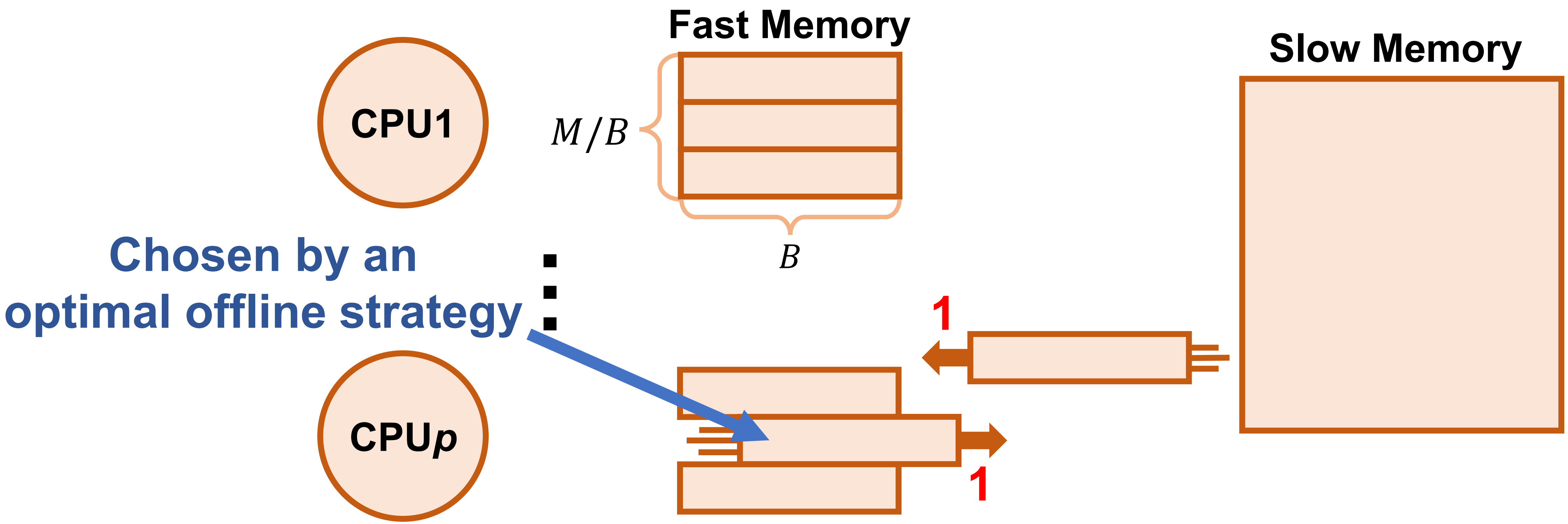
The parallel I/O Model (private-cache model) [ABB02],[FS09]

- Each of a total of p processors owns a private cache of size M
- Where each operation is executed is decided by the RWS scheduler



The parallel I/O Model (private-cache model) [ABB02],[FS09]

- Each of a total of p processors owns a private cache of size M
- Where each operation is executed is decided by the RWS scheduler
- What are in the caches is decided by optimal replacement policy
- The parallel I/O cost is the total loads/evicts to execute an algorithm



What we should analyze about parallel I/O (cache) bound

- When there is only one processor, the parallel I/O cost is the same as the sequential I/O cost
- When there are more processors, the parallel I/O cost can either be lower (since we have larger total cache size) or larger (when a "task" is scheduled to another processor, we lose all data in the cache)
- Usually we care about the worst-case guarantee, and want to bound the "parallel overhead", so:

parallel I/O cost $Q_p \leq$ sequential I/O cost Q_1 + parallel overhead Q'_p

- Trivial upper bound: $Q'_p = O(pD) \cdot \frac{M}{B}$, which is #steal multiplied by cache size [ABB02], but this is bound is very loose

Unfortunately, there was not much improvement on this topic, probably due to the complication

- **Frigo and Strumpen [2009] gave a general approach to analyze parallel I/O (cache) bounds for cache-oblivious algorithms**
  - Gave good bounds on matrix transpose, matrix multiplication, sorting
  - But the derived bounds are not optimal for many other algorithms
  - Some details are discussed in more details by Cole and Ramachadram **[2012]**

  - The parallel overhead is polynomially proportional to D, the span of the algorithm
  - For instance, for Gaussian Elimination, using the method in **[FS09]**, we have

$$Q_p = O\left(\frac{n^3}{B\sqrt{M}} + P^{\frac{1}{3}} n^{\frac{1}{3}} \cdot \frac{n^2}{B} + \mathrm{lower\_order\_terms}\right)$$

Using Gaussian Elimination as an example

- Using the method in **[FS09]**, we have

$$Q_p = O\left(\frac{n^3}{B\sqrt{M}} + P^{\frac{1}{3}} n^{\frac{1}{3}} \cdot \frac{n^2}{B} + \text{lower\_order\_terms}\right)$$

  where the overhead term dominates when $n = O\left(P^{\frac{1}{2}} M^{\frac{3}{4}}\right) \approx 10^7$

- Meanwhile, for matrix multiplication, the method in **[FS09]** gives:

$$Q_p = O\left(\frac{n^3}{B\sqrt{M}} + P^{\frac{1}{3}} \log^{\frac{2}{3}} n \cdot \frac{n^2}{B} + \text{lower\_order\_terms}\right)$$

- In practice, we can measure that the cacheline loads/evicts is similar for the cache-oblivious algorithms for both problems

- **Can we show the bound of MM for Gaussian Elimination as well?**

New results shown in this paper

Algorithm	Seq. Bound	Parallel Overhead: New in this paper	Parallel Overhead: Previous best [27, 40, 52]	Parallel Overhead: Lower Bound [11, 27]
Gaussian Elimination Kleene's algorithm for APSP	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{2/3} n \cdot \frac{n^2}{B} + Pn$	$P^{1/3} n^{1/3} \cdot \frac{n^2}{B} + Pn\log B$	$P^{1/3} \cdot \frac{n^2}{B}$
Triangular System Solver	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{5/3} n \cdot \frac{n^2}{B} + Pn$	$P^{1/3} n^{1/3} \cdot \frac{n^2}{B} + Pn\log B$	$P^{1/3} \cdot \frac{n^2}{B}$
Cholesky Factorization LU Decomposition	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{5/3} n \cdot \frac{n^2}{B} + Pn\log n$	$P^{1/3} n^{1/3}\log^{1/3} n \cdot \frac{n^2}{B} + Pn\log n$	$P^{1/3} \cdot \frac{n^2}{B}$
LWS Recurrence	$\frac{n^2}{BM} + \frac{n}{B} + 1$	$P^{1/2}\log^2 n \cdot \frac{n}{B} + Pn$	$P^{1/2} \cdot n^{1/2} \cdot \frac{n}{B} + Pn$	$P^{1/2} \cdot \frac{n}{B}$
GAP Recurrence	$\frac{n^3}{BM} + \frac{n^2\log M}{B} + 1$	$P^{1/2}\log^2 n \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2} n^{\kappa/2}\log M \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2} \cdot \frac{n^2}{B}$
Parenthesis Recurrence	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{5/3} n \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/3} n^{\kappa/3} \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/3} \cdot \frac{n^2}{B}$
RNA Recurrence	$\frac{n^4}{BM} + \frac{n^2}{B} + 1$	$P^{1/2}\log^2 n \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2} n^{\kappa/2} \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2} \cdot \frac{n^2}{B}$
Protein Accordion Folding	$\frac{n^3}{BM} + \frac{n^2}{B} + 1$	$P^{1/2}\log n \cdot \frac{n^2}{B} + Pn\log^2 n$	$P^{1/2} n^{1/2}\log n \cdot \frac{n^2}{B} + Pn\log^2 n$	$P^{1/2} \cdot \frac{n^2}{B}$

$\kappa = 1.58$

The new bounds in this paper is only polylogarithmically off the lower bounds

Matrix Multiplication
Gaussian Elimination
Triangular System Solver
LU Decomposition

LWS Recurrence
Parenthesis Recurrence
RNA Recurrence
GAP Recurrence
Protein accordion folding
2-Knapsack Recurrence

↕

k-d grid [BellllochGu20]

↕

I/O bound vs. Work bound

Sequential vs. Parallel

Lower vs. Upper bound (algorithms)

Symmetric vs. Asymmetric

In this paper, we show that we can decouple the span from the analysis of the parallel I/O (cache) bounds

Using Frigo-Strumpen's bounds for basic components: 2D-grid, 3D-grid, matrix transpose

The span (parallelism) does not show up in the analysis, and only affects the base-case bound

Defining (α, β, k, l, m)-recurrence

Plugging in base-case bounds for 2D-grid, 3D-grid, matrix transpose

Solve the recurrence and get tighter parallel cache bounds

Summary

Two main contributions of this paper

- We showed a simplified analysis for the randomized work-stealing (RWS) scheduler that (1) requires no potential function, (2) applies to a more general asynchronized setting, (3) separate math from the main idea, and (4) only uses Chernoff bound in a simple form

- We show a framework (based on [BlellochGu20]) to derive much tighter parallel I/O (cache) bounds for 10 classic problems on algebra and dynamic programming

- If you have any questions, please contact me via: ygu@cs.ucr.edu