Parallel Algorithms

Yan Gu

Course announcement

- **Problem-solving training 1 is available**
- **Start to do it soon!**
  - Hard to predict the amount of time you need
  - You don’t have other homework this week
- **5 of you have already solved some problems**
- **More have started**

Course announcement

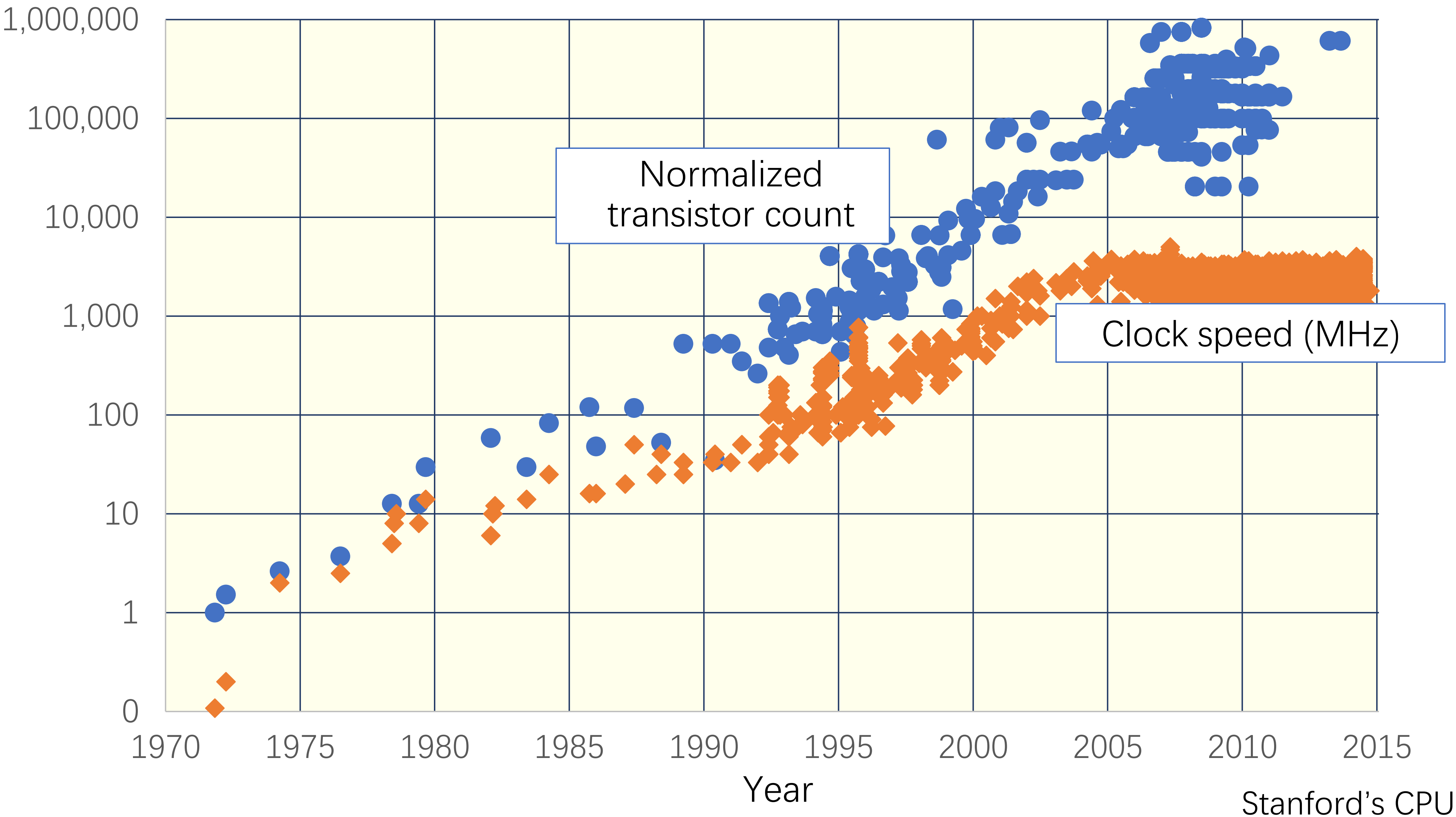
- **Office hour:**
  - Yan Gu : 1:00 - 2:00 PM Friday
  - Xiaojun Dong : 4:00 - 5:00 PM Tuesday

- **Recorded video:**
  - https://www.cs.ucr.edu/~ygu/teaching/142/W21/web/video/L1.mp4
  - For future courses, just replace “1” to the lecture label

Parallel Algorithms

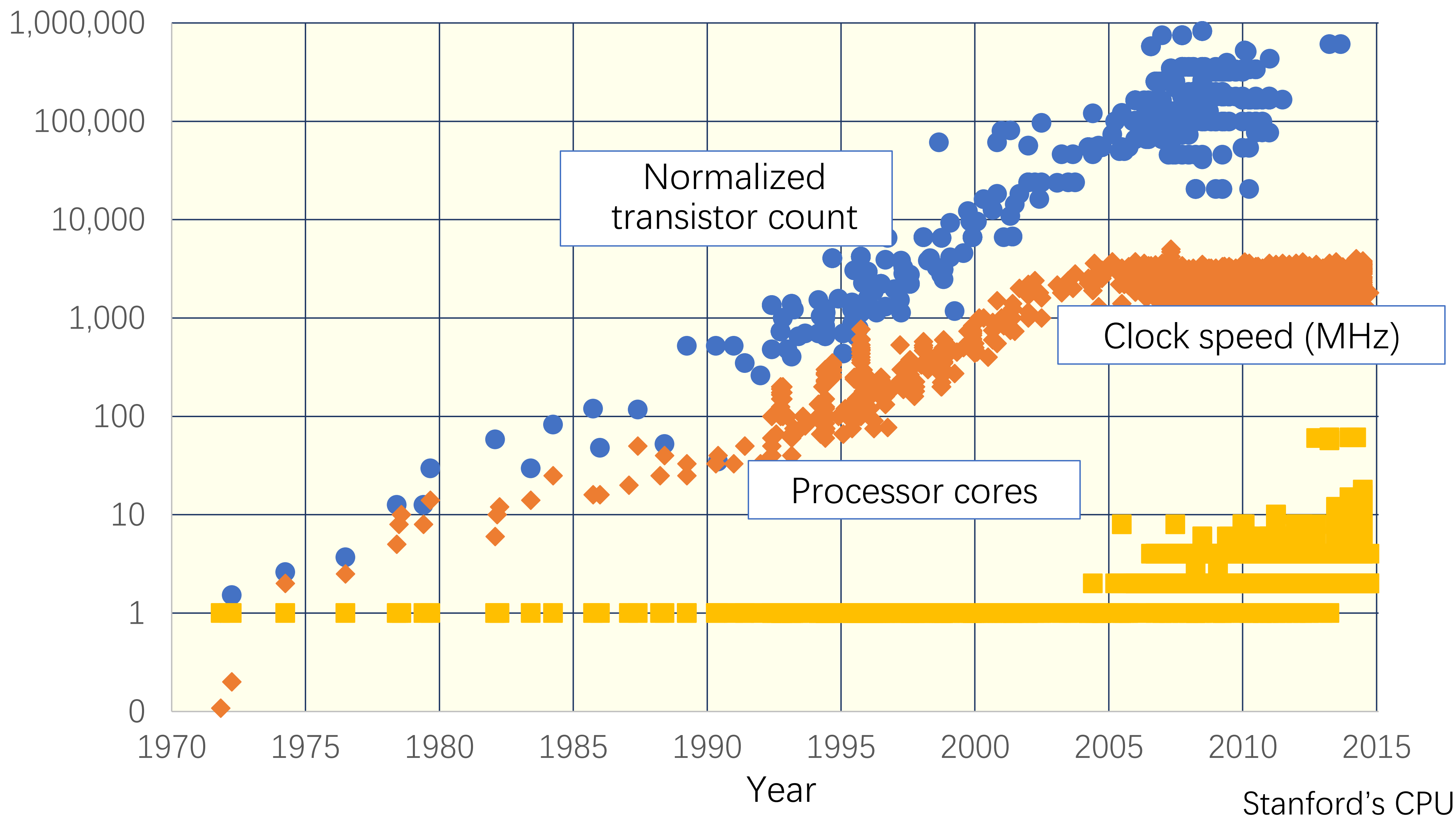
Yan Gu

Technology Scaling After 2004



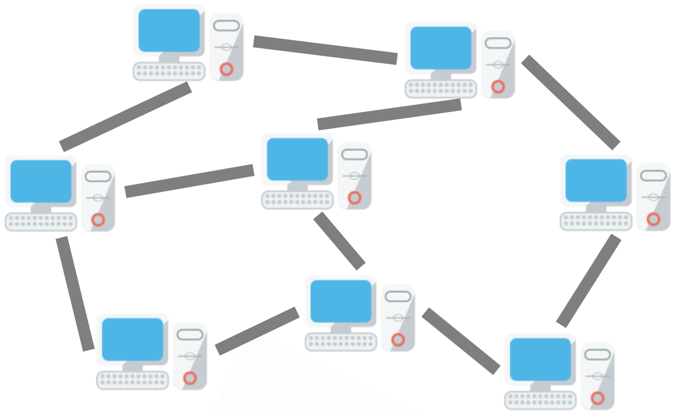
Stanford's CPU DB [DKM12]

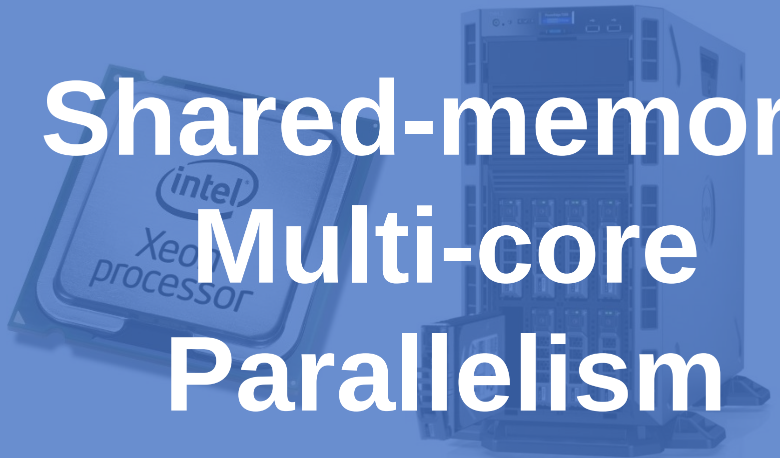
Technology Scaling After 2004



Stanford's CPU DB [DKM12]

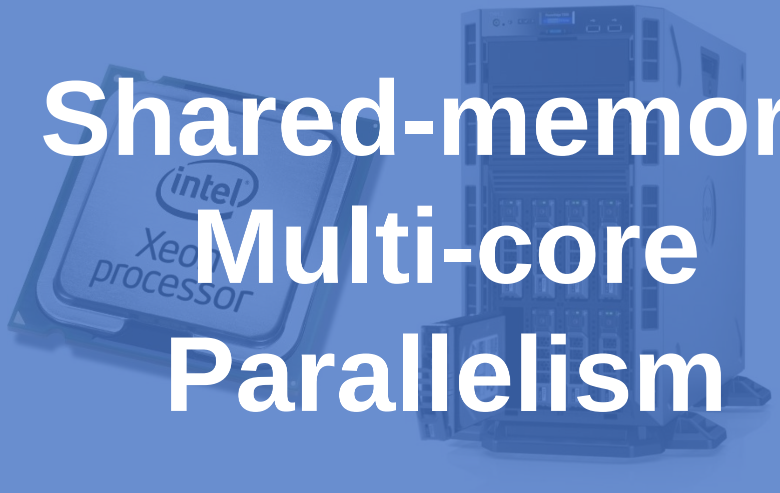
Ways to Make Code Faster: Parallelism





Shared-memory Multi-core Parallelism

What you will learn in this lecture



Shared-memory Multi-core Parallelism

Multiple processors collaborate to get a task done

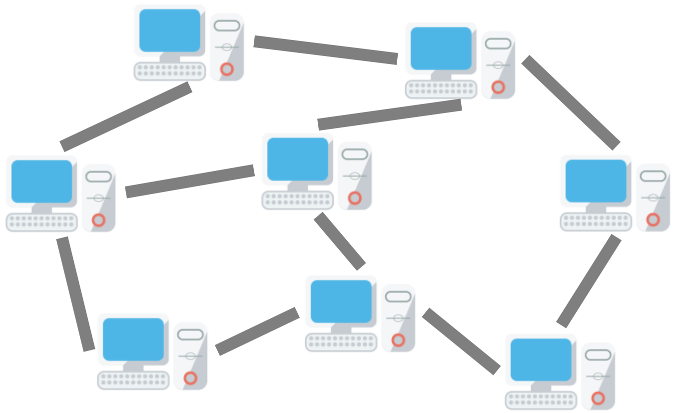
Parallel machines

4 cores, 8 hyperthreading
Usually $700-$1500

- 96-cores, 192 hyper-threading
- 1.5TB of main memory
- Cost: about 30k USD, mostly due to memory

AWS: 144 hyper-threads and 2TB of memory

0.01 to ~6 dollars per hour

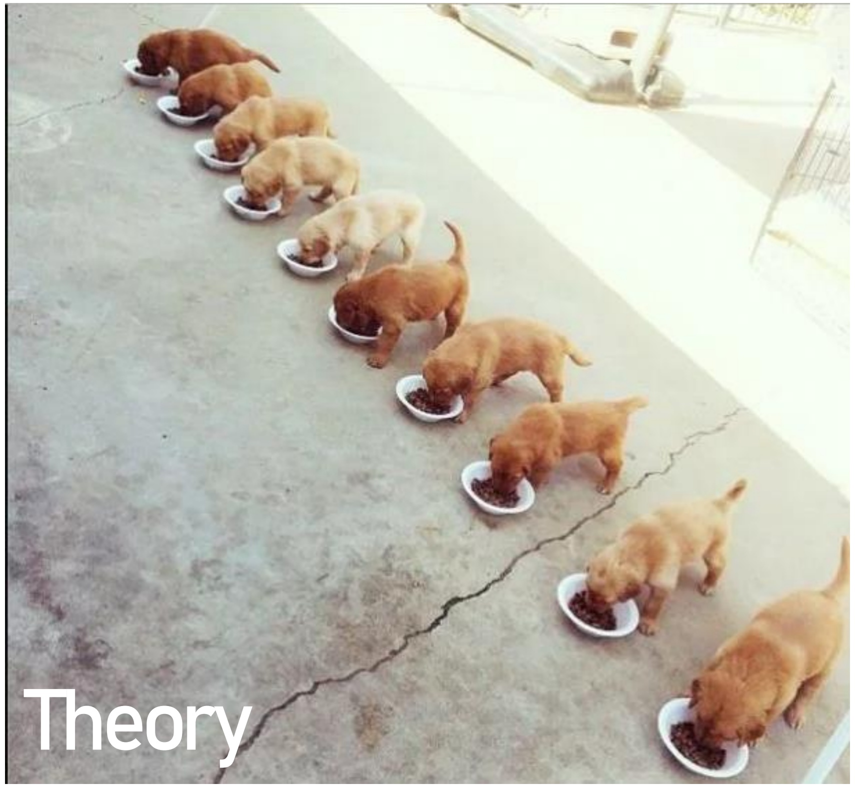


Each of them a multi-core machine

We need to consider parallelism in algorithm design!

Multi-core Programming: Theory and Practice

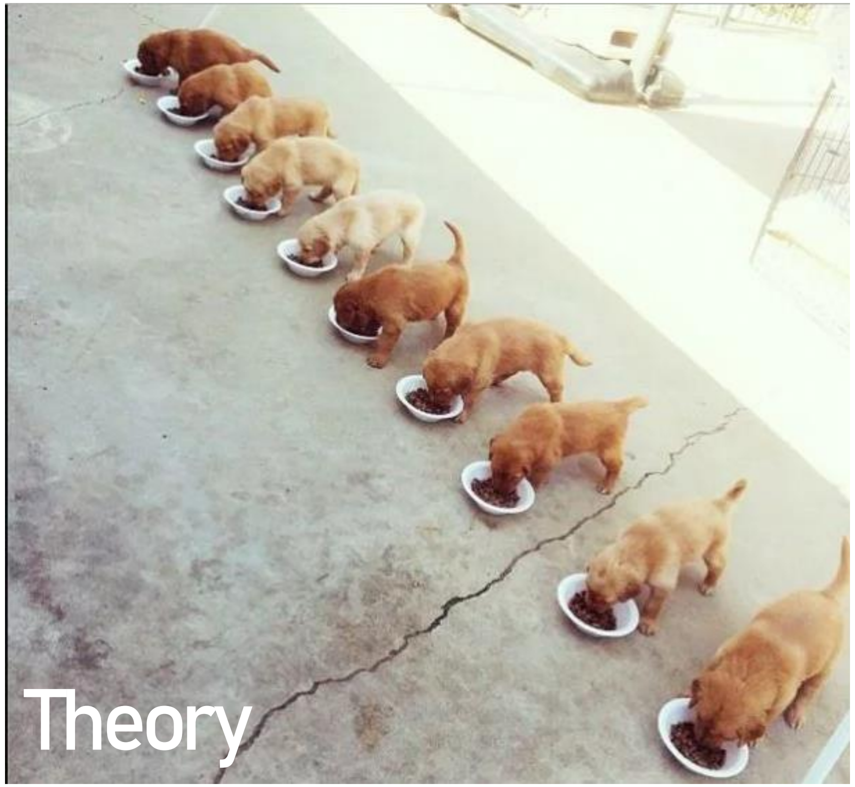
Memory leaking: memory which is no longer needed is not released

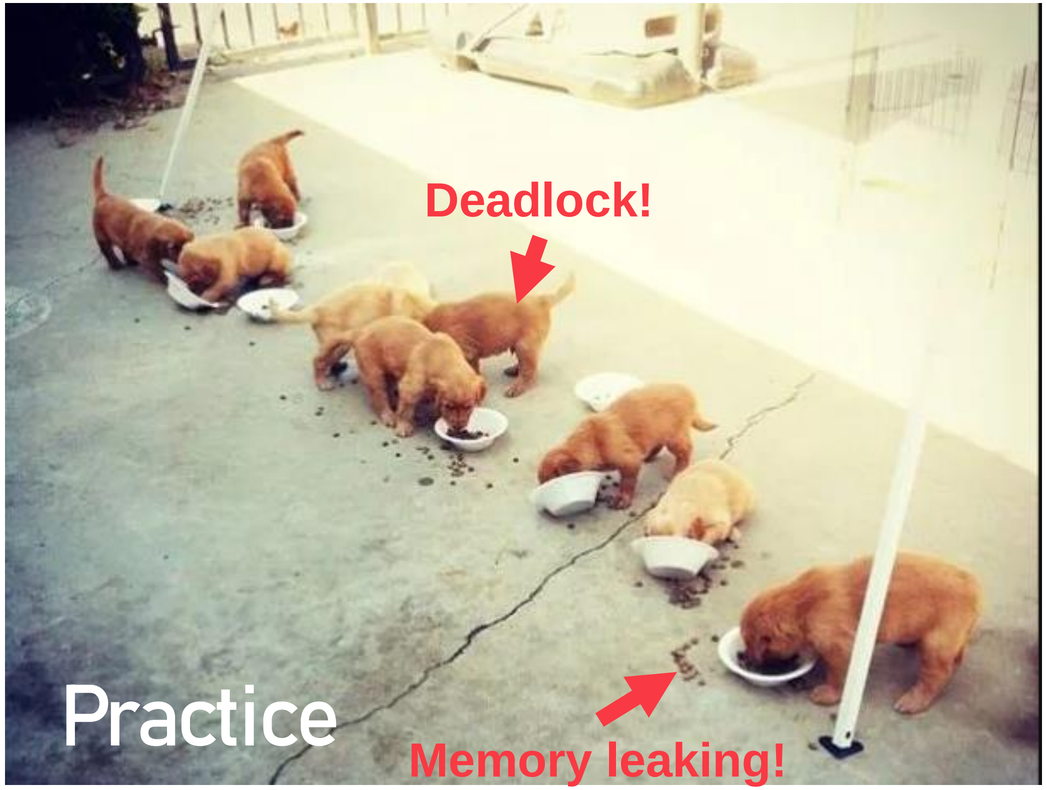


(Pictures from 9gag.com)

Multi-core Programming: Theory and Practice

Deadlock: a state in which each member of a group is waiting for another member, including itself, to take action, such as releasing a lock

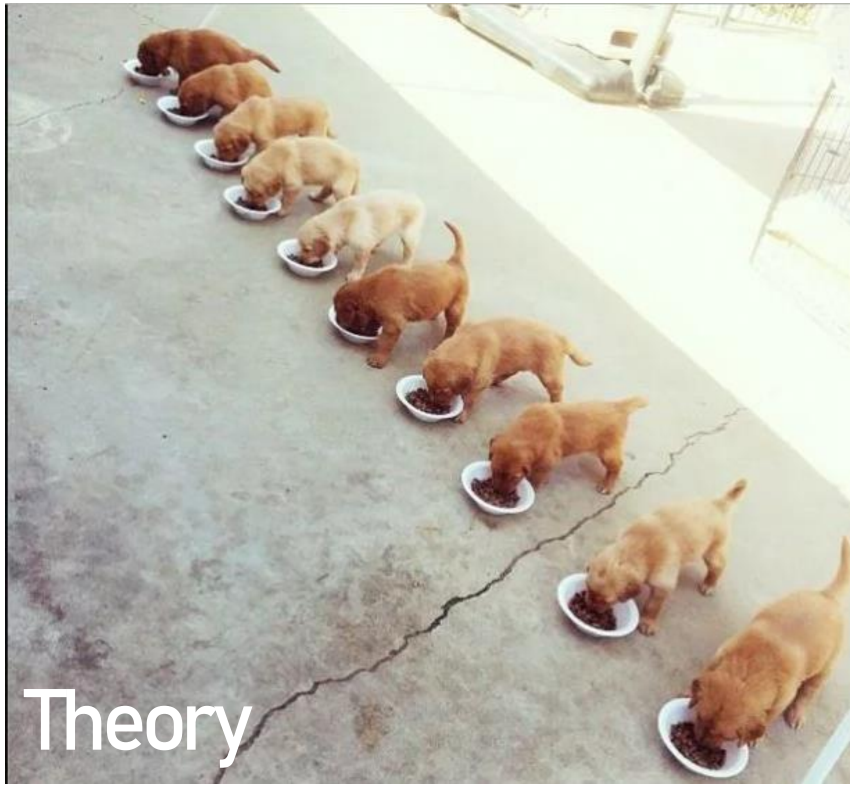




(Pictures from 9gag.com)

Multi-core Programming: Theory and Practice

Data Race: Two or more processors are accessing the same memory location, and at least one of them is writing

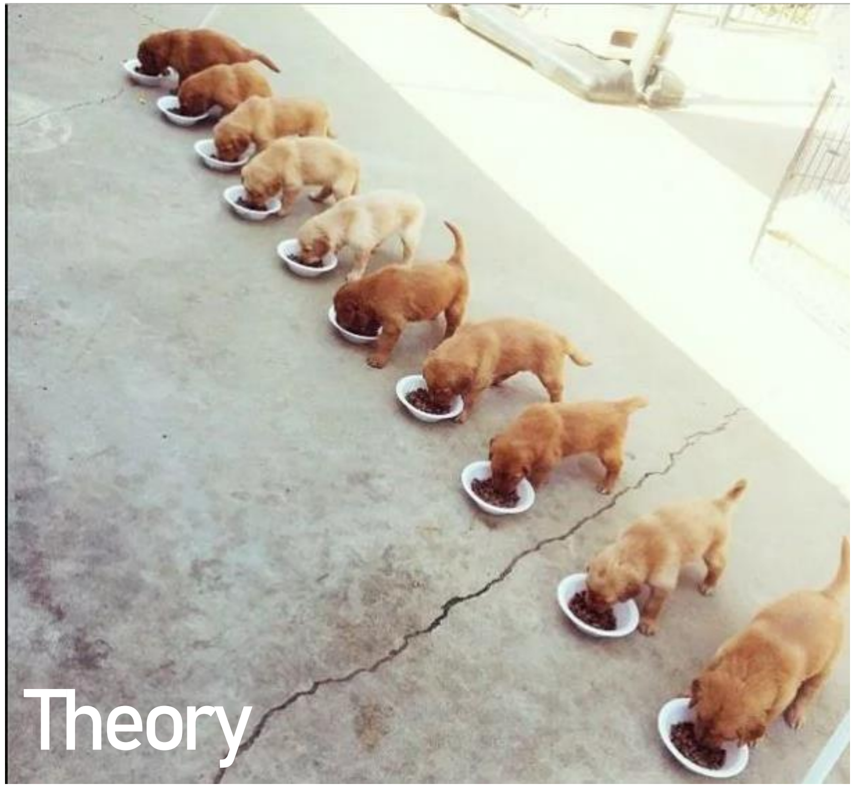


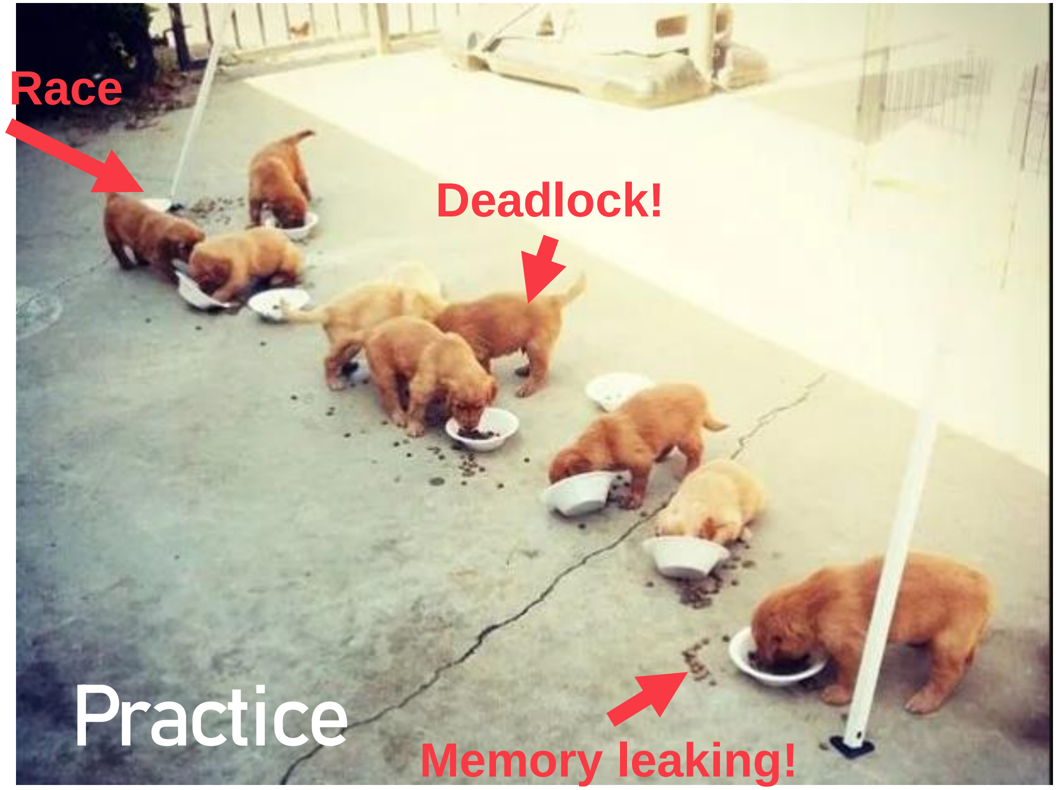
(Pictures from 9gag.com)

Multi-core Programming: Theory and Practice

Zombie process: a process that has completed execution but still has an entry in the process table

Missing the 10th dog! Did it become a zombie???





(Pictures from 9gag.com)

Multi-core Programming

- **We need to learn theory:**
  - Making performance predictable

- **Not let this to happen →**

Parallel algorithms

- **We'll learn some fundamental knowledge about parallel algorithm design**
- **We'll practice parallel programming on some simple applications**
- **If you are interested, take the course CS214 (parallel algorithms) in Spring**
  - Offered by Yihan Sun, tier-1 graduate course

Warm-up: reduce
(Compute the sum of values in an array)

6 + 15 + 15 = 36

A = 1 2 3 | 4 5 6 | 7 8

Sum(A): 36

- **Cut the input array into smaller segments, sum each up individually, and finally sum up the sums**

```
Sum(A, n) {
  int B[p];
  for processor i (i=0..p-1) {
    for (j=i*n/p to i*n/p+n/p) B[i] += A[j];
  }
  sync all processors;
  for (j = 0 to p) ret += B[i];
  return ret; }
```

6 + 15 + 15 = 36

A = 1 2 3 | 4 5 6 | 7 8

Sum(A): 36

- **Cut the input array into smaller segments, sum each up individually, and finally sum up the sums**
- **Picking the appropriate number of segments can be annoying**
  - Machine parameter, runtime environment, algorithmic details

6 + 15 + 15 = 36

A = 1 2 3 | 4 5 6 | 7 8

Sum(A): 36

- Cut the input array into smaller segments, sum each up individually, and finally sum up the sums

```
Sum(A, n) {
  int B[p];
  for processor i (i=0..p-1) {
    for (j=i*n/p to i*n/p+n/p) B[i] += A[j];
  }
  sync all processors;
  for (j = 0 to p) ret += B[i];
  return ret; }
```

What if you have $O(n)$ processors?

Problems

- Should not assume we know the number of processors p ahead of time
- Algorithm must have good performance (parallelism) for any given p (which even dynamically changes)
- Dealing with system-level issues is error-prone – makes parallel programming notoriously hard

Is there an easier way for parallel algorithm/programming?

Dynamic Multi-threading (task-parallel) + Scheduler

Dynamic Multi-threading

- **Specify parallelism for tasks**
  - Specify which tasks can be executed in parallel (parallel do, parallel for, …)

- **No worry about communication protocols, load balancing, system-level implementation, # of available processors, …**

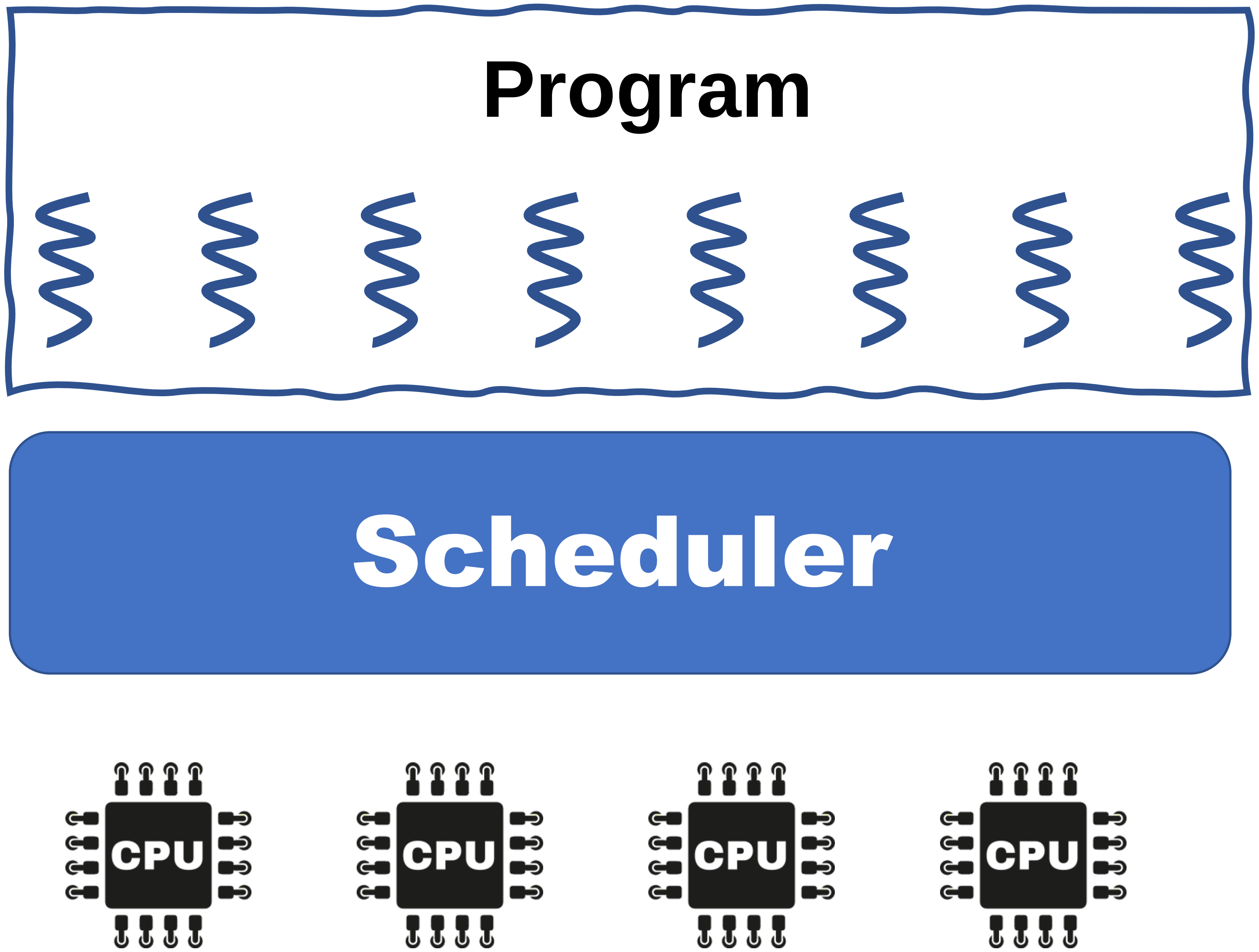
- **The actual execution will be done by a scheduler**

- **Greatly simplifies programming and theoretical analysis**

Scheduler

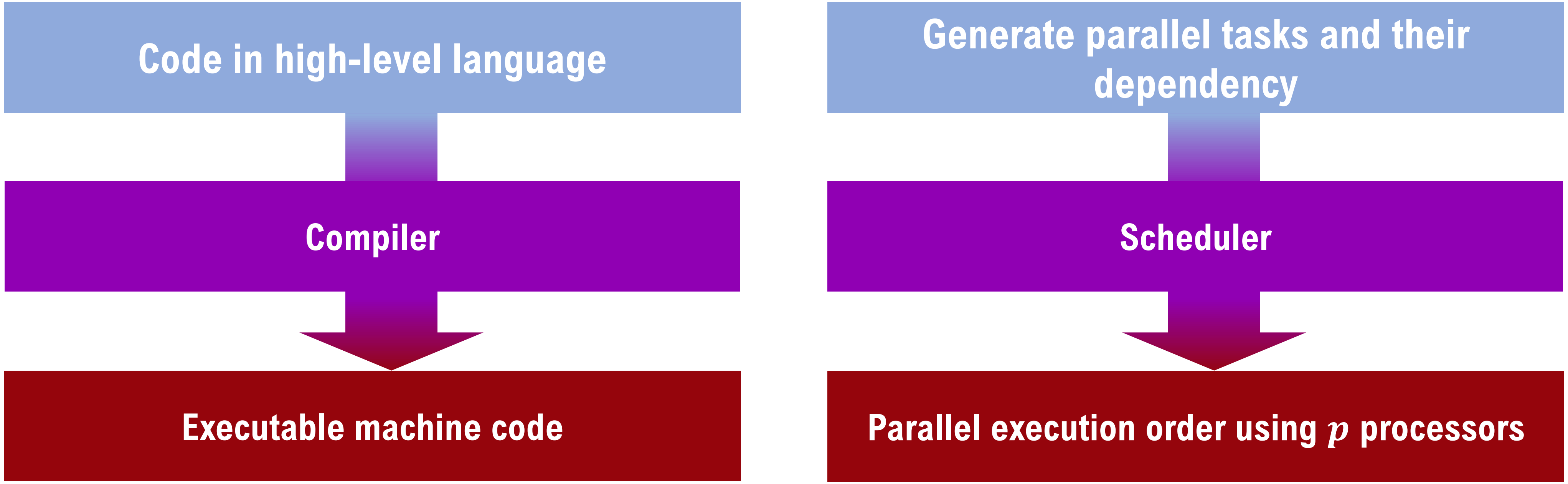
- The program generate tasks

- The scheduler maps each task to a processor (e.g., whenever a processor is available)



Scheduler

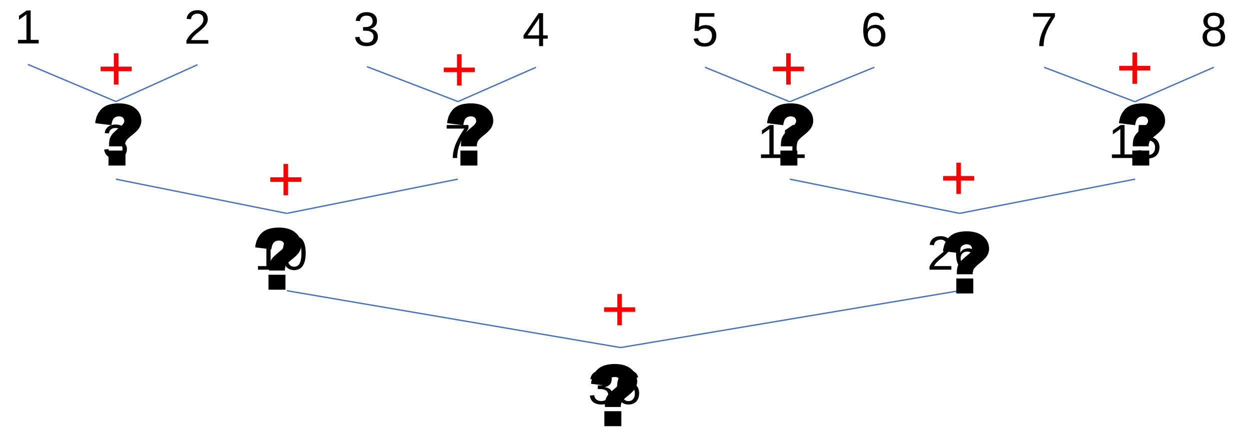
- Consider it as a complier. Programmers then only need to focus on high-level algorithm design



- We always assume an effective scheduler
- We design algorithms only focusing on generating parallel tasks

Back to the warm-up example

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

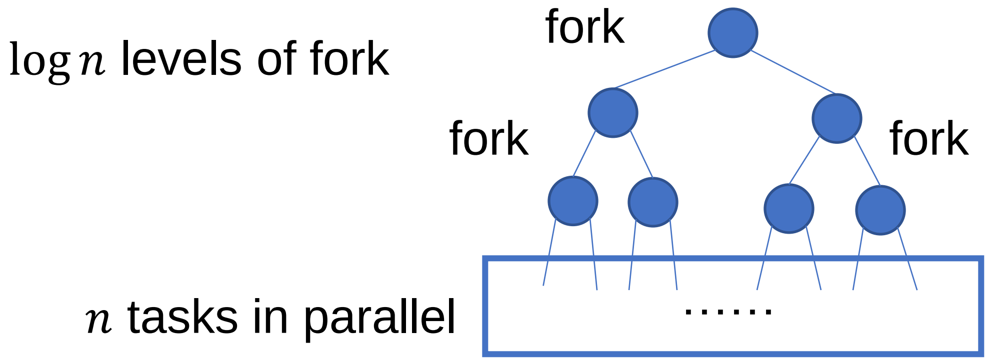



```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

How to evaluate the running time (time complexity) of a parallel algorithm

(without knowing how many processors can be used)

Binary Fork-Join Model



- **You write the code exactly the same as the sequential code, except that**
  - The "in parallel" instruction: fork two tasks (functions) and they can be run in parallel (but not necessarily run in parallel)
  - The "parallel for" instruction: all iterations in this for loop can be run in parallel

```
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;
}
```

```
copy(A, B, n) {
    parallel for (i=0; i<n; i++)
        B[i] = A[i];
}
```

It's extremely easy to implement such an algorithm

- **Cilk, PBBS, the Java fork-join framework, X10, Habanero, Intel Threading Building Blocks (TBB), and the Microsoft Task Parallel Library**

```
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;
}
```

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

It's extremely easy to implement such an algorithm

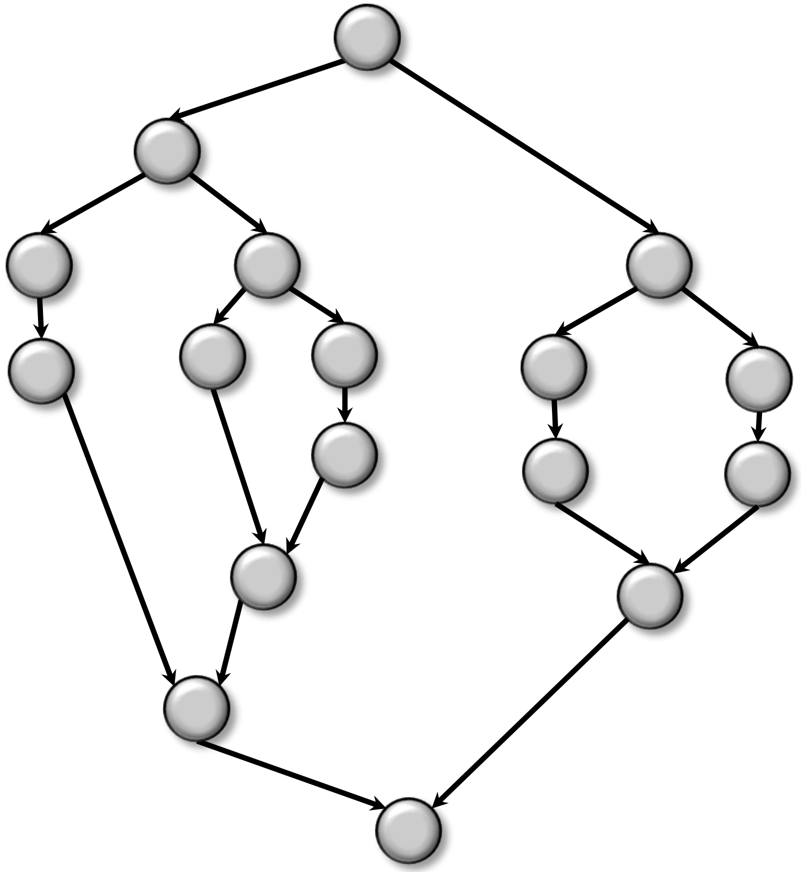
- Simple for theoretical analysis – we'll see in a while
- Simple for programming – almost exactly the code!

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

Cost model: work-span

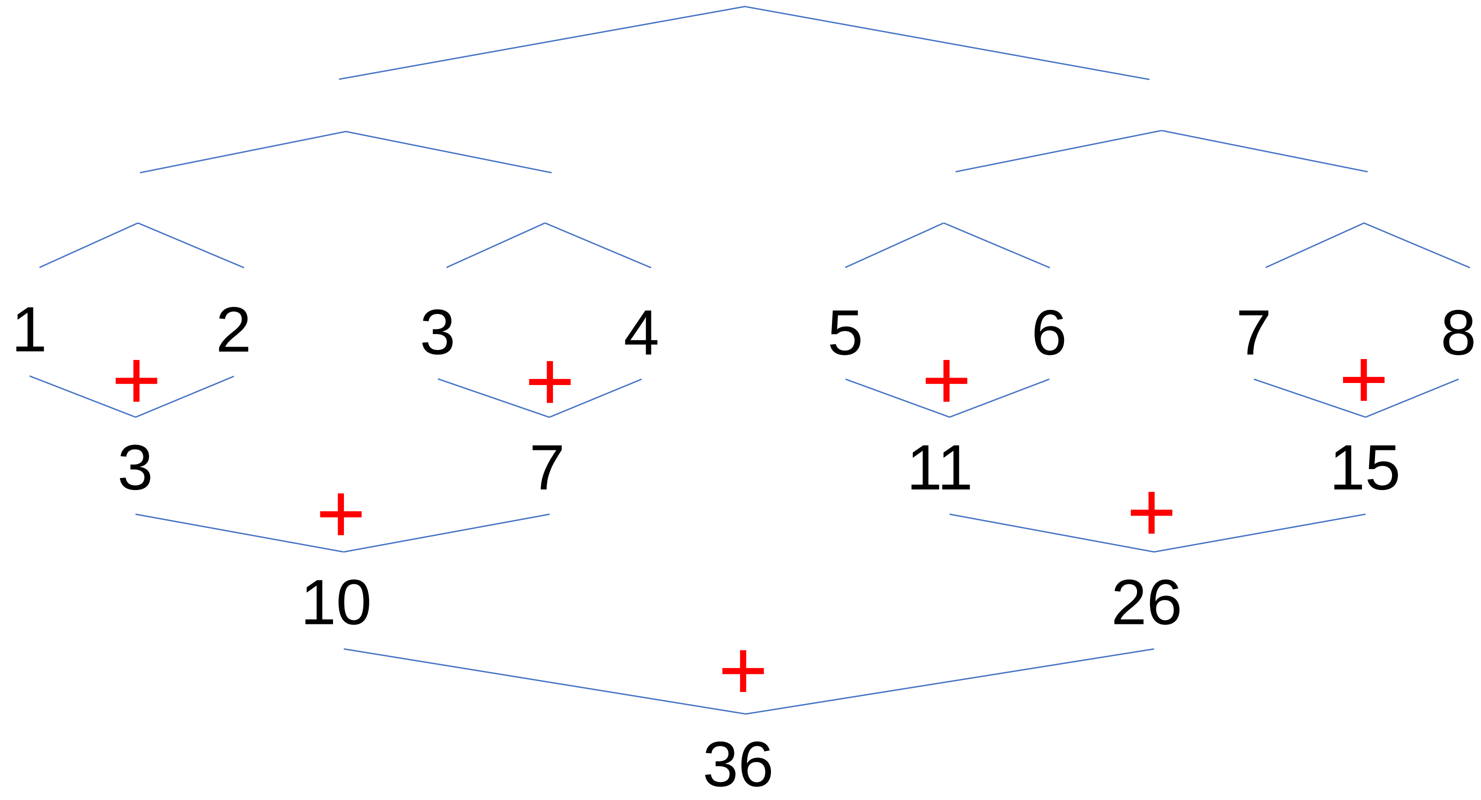
- **For all computations, draw a DAG**
  - A->B means that B can be performed only when A has been finished
- **Work: the total number of operations**
- **Span (depth): the longest length of chain**



- **It shows the dependency of operations in the algorithm**

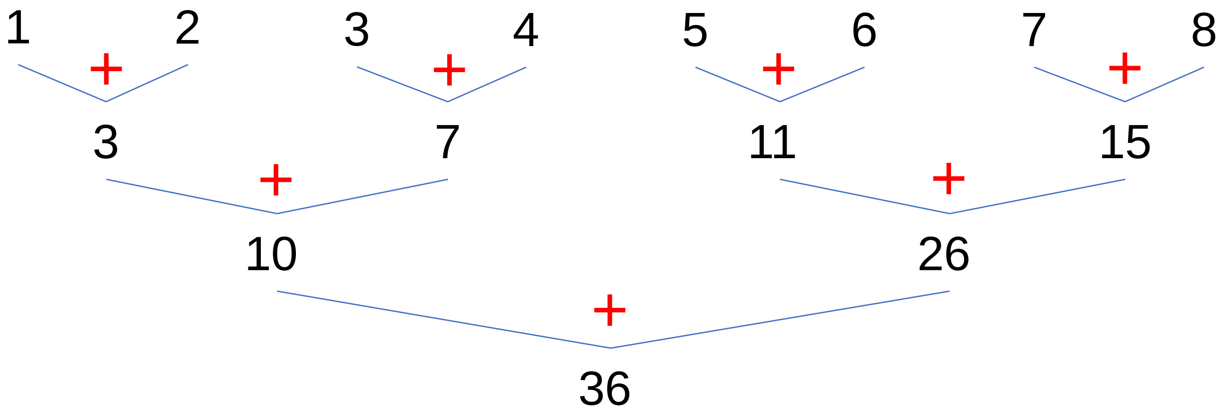
Computational DAG

```
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;
}
```



Cost model: work-span

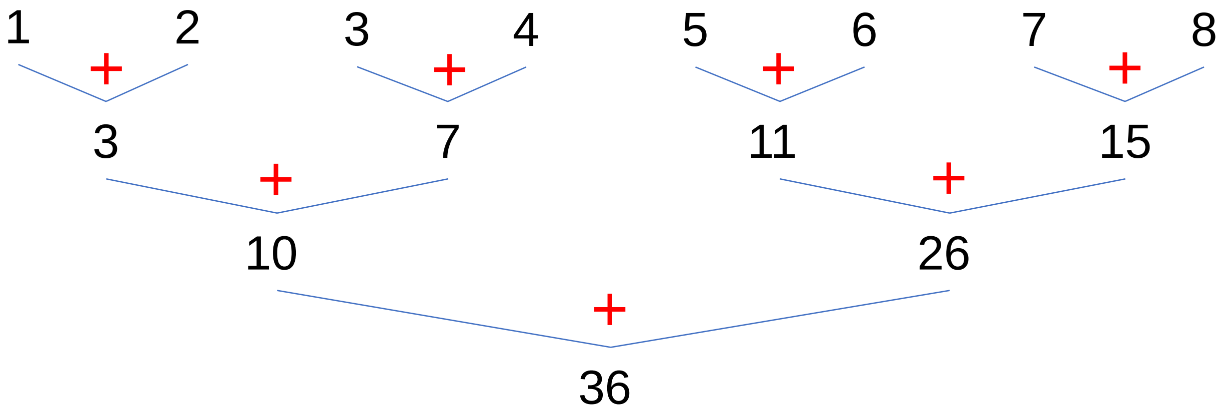
```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```



Work: $O(n)$

- **Work: The total number of operations in the algorithm**
  - Sequential running time when the algorithm runs on one processor
  - Work-efficiency: the work is (asymptotically) no more than the best (optimal) sequential algorithm
  - Goal: make the parallel algorithm efficient when a small number of processor are available

Cost model: work-span



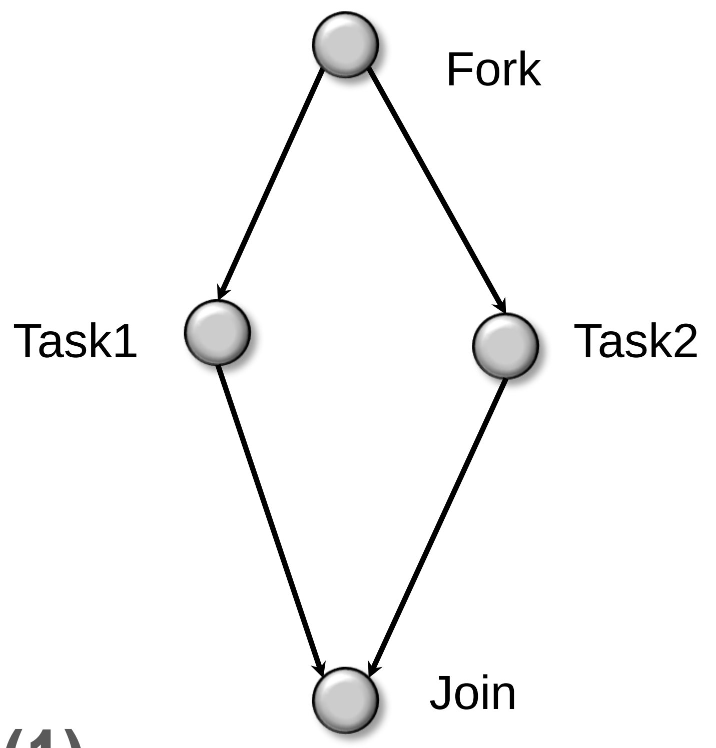

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

Span: $O(\log n)$

- **Span (depth): The longest dependency chain**
  - Total time required if there are infinite number of processors
  - Our goal is usually to make span polylogarithmic
  - Goal: make the parallel algorithm faster and faster when more and more processors are available (**scalability**)

Compute work and span

- When we see a in-parallel (fork-join, spawn-sync):
  - in-parallel
  - Task1
  - Task2
- Work = work of Task1 + work of Task2+O(1)
- Span = max(span of Task1, span of Task2)+O(1)

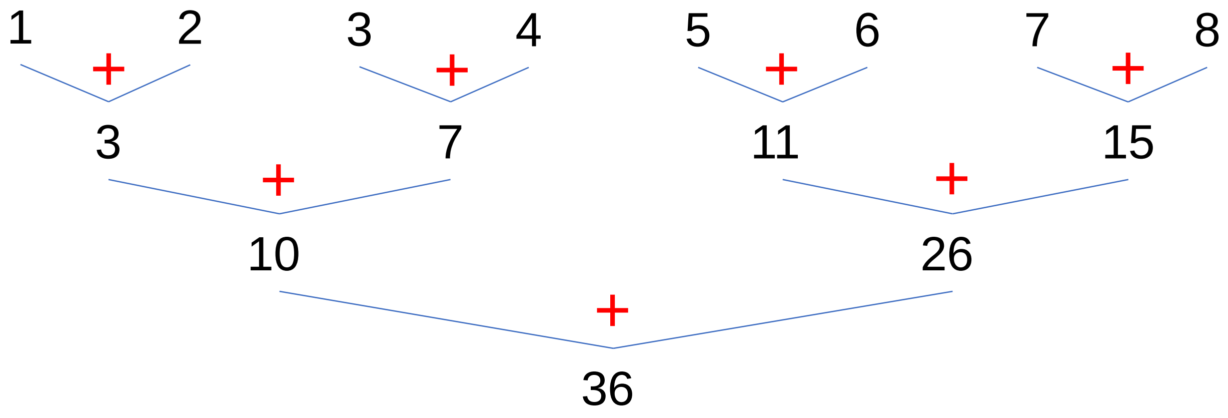


Compute work and span

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1)$
- $\Rightarrow W(n) = \Theta(n)$

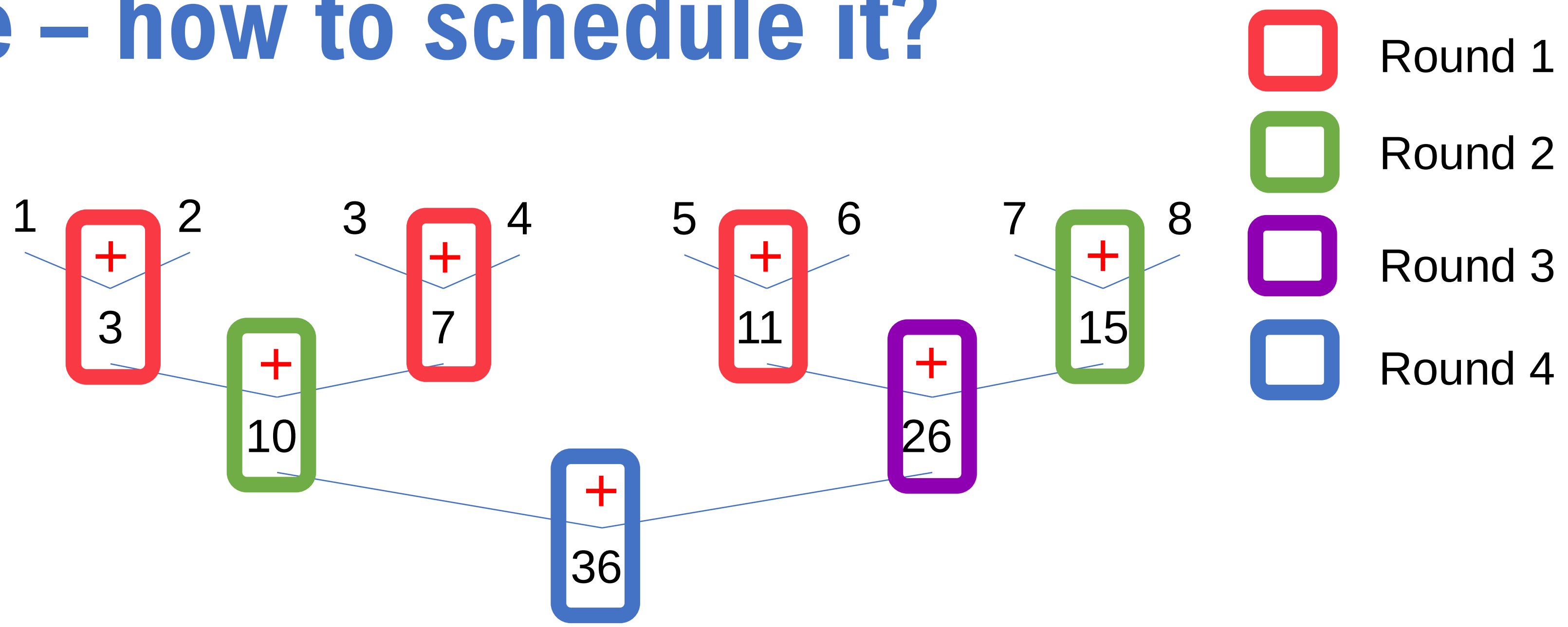
- $S(n) = S\left(\frac{n}{2}\right) + \Theta(1)$
- $\Rightarrow S(n) = \Theta(\log n)$

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



How do work and span relate to the real execution and running time?

Reduce – how to schedule it?



- **Find at most p tasks that do not depend on each other and execute them in parallel**
- **Can be executed in time $\frac{W}{p} + S$ using p processors for a DAG with work W and span S**
  - $\frac{W}{p} + O(S)$ in practice, usually a big constant in the big-O

Golden standard for a parallel algorithm

- **Simple**
- **Work-efficient**
  - (Asymptotically) Use no more work than the sequential algorithm
  - Fast or no (much) slower on one core
- **Low span**
  - Ideally logarithmic or polylogarithmic
  - Fast when there are lots of cores

Summary

- **Parallel algorithms**
  - Some theoretical results/tools, help you reason your parallel code/performance
- **Dynamic multi-threading**
  - Keep things simple – only focus on high-level parallelism and dependency
  - The actually execution will be done by a scheduler
- **Fork-join**
  - Fork (spawn): create a new thread working on a task in parallel
  - Join (sync): synchronous previously forked threads
- **Work-span model**
  - A parallel algorithm/computation can be viewed as a DAG
  - Work: the total number of operations. Running time using 1 processor
  - Span (depth): the longest dependency chain. Running time using an unlimited number of processors

Next lectures

- **How to program a parallel algorithm**
  - In a simple, efficient, and elegant way
  - Still some engineering work to do. What are they?
- **More parallel algorithms**
  - Scan, filter, pack, partition, sorting
  - Parallel thinking