

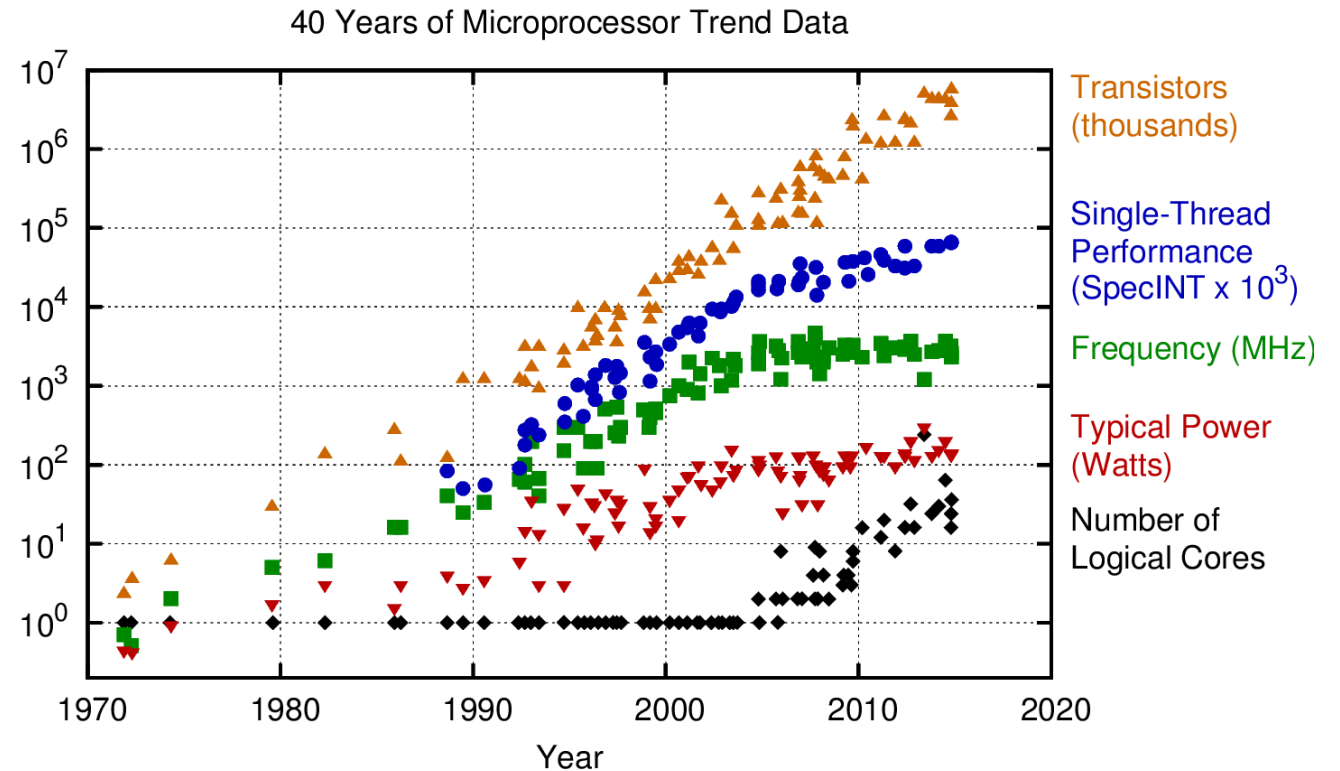
CS260 – Algorithmic
Engineering
Yihan Sun

Parallel Algorithms and Implementations

Algorithmic engineering
– make your code faster

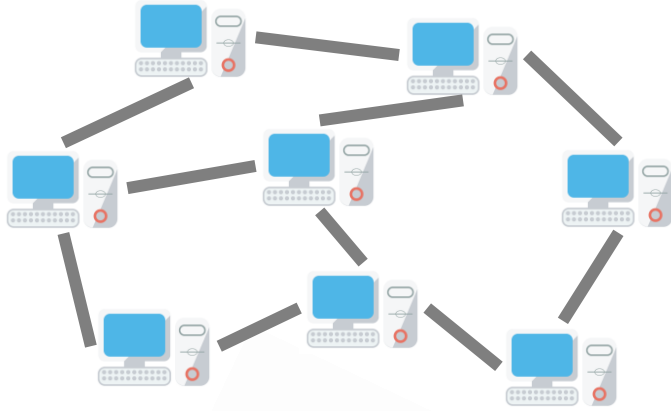
Ways to Make Code Faster

- Cannot rely on the improvement of hardware anymore
- Use multicores!

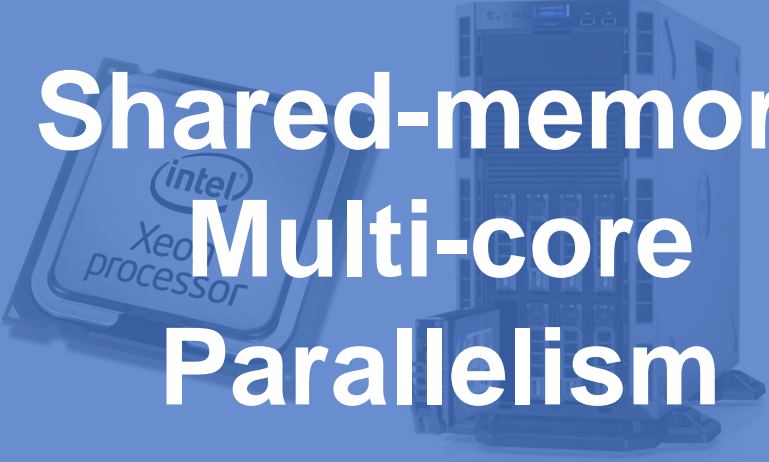


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Ways to Make Code Faster: Parallelism



Shared-memory
Multi-core
Parallelism

A blue rounded rectangle containing a faded image of an Intel Xeon processor and a server rack. The text "Shared-memory Multi-core Parallelism" is overlaid in white.

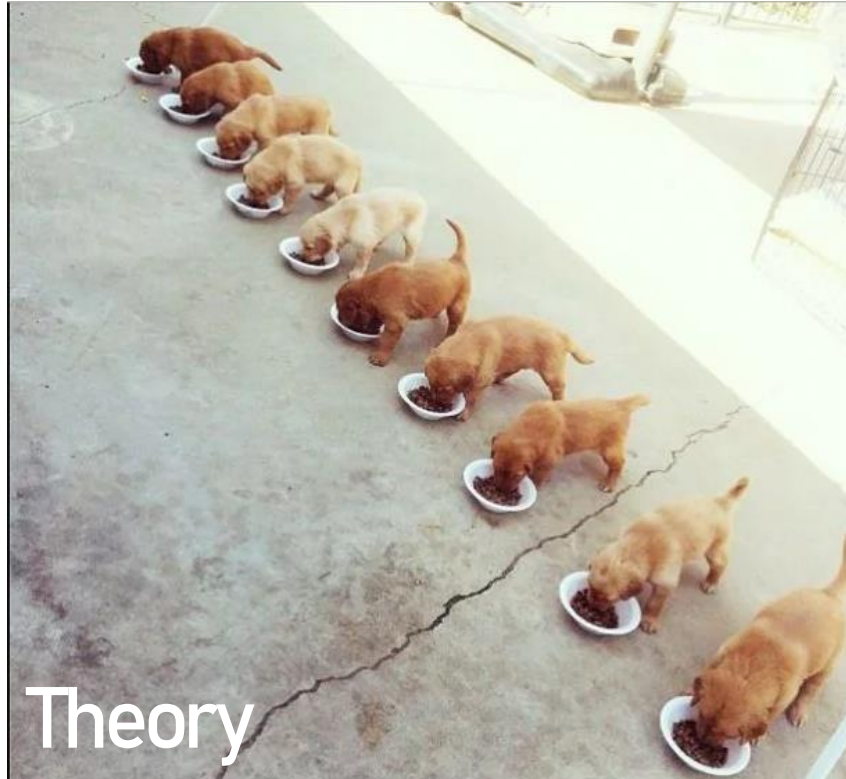
Shared-memory Multi-core Parallelism

Multiple processors **collaborate** to get a task done

(And avoid any contention between them)

Multi-core Programming: Theory and Practice

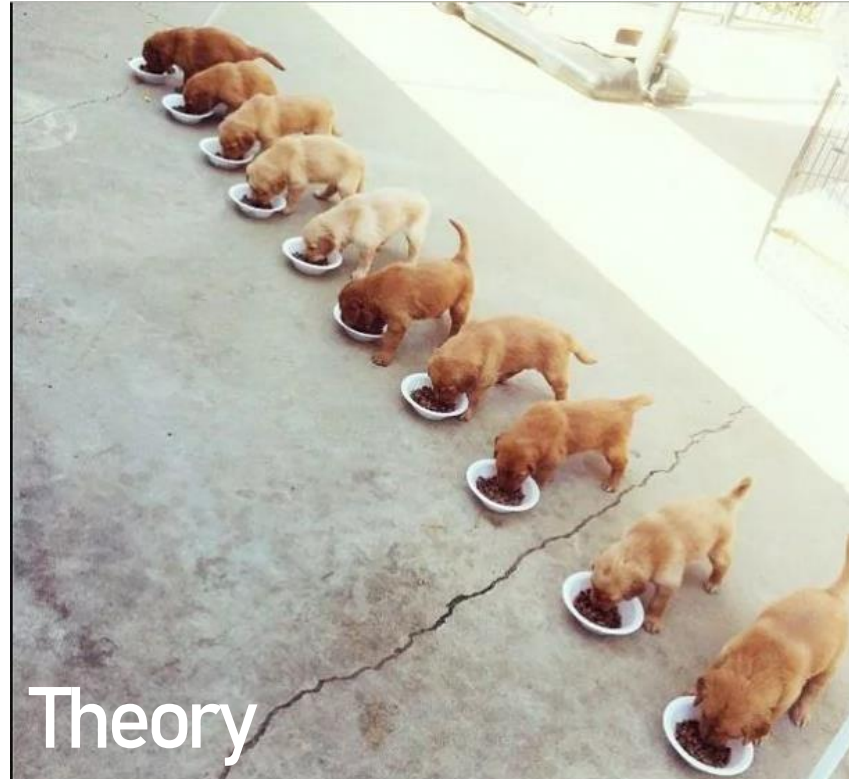
Memory leaking: memory which is no longer needed is not released



(Pictures from 9gag.com)

Multi-core Programming: Theory and Practice

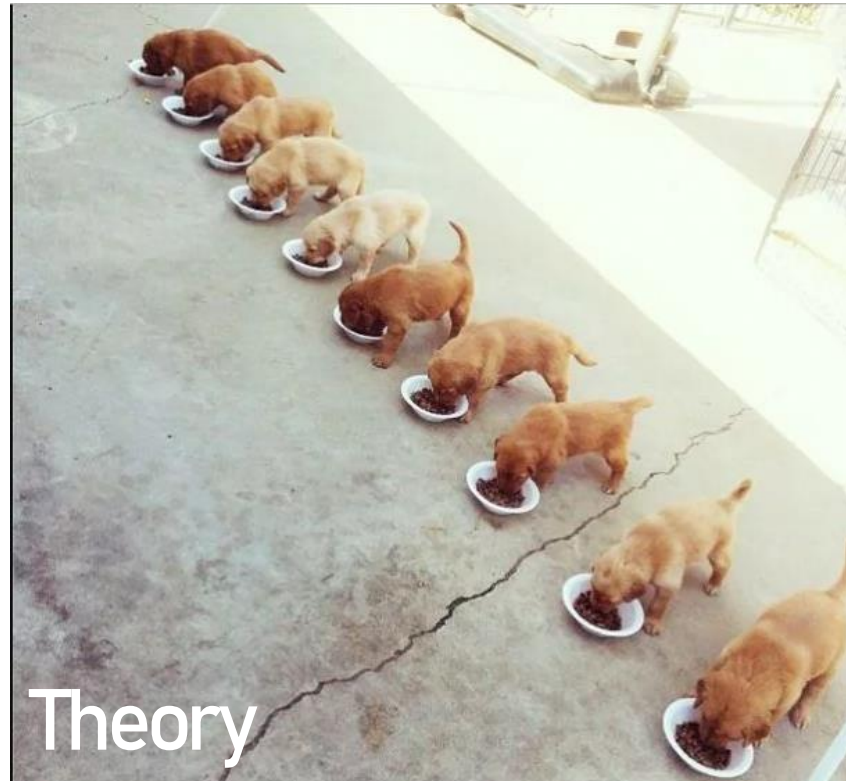
Deadlock: a state in which each member of a group is waiting for another member, including itself, to take action, such as releasing a lock



(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

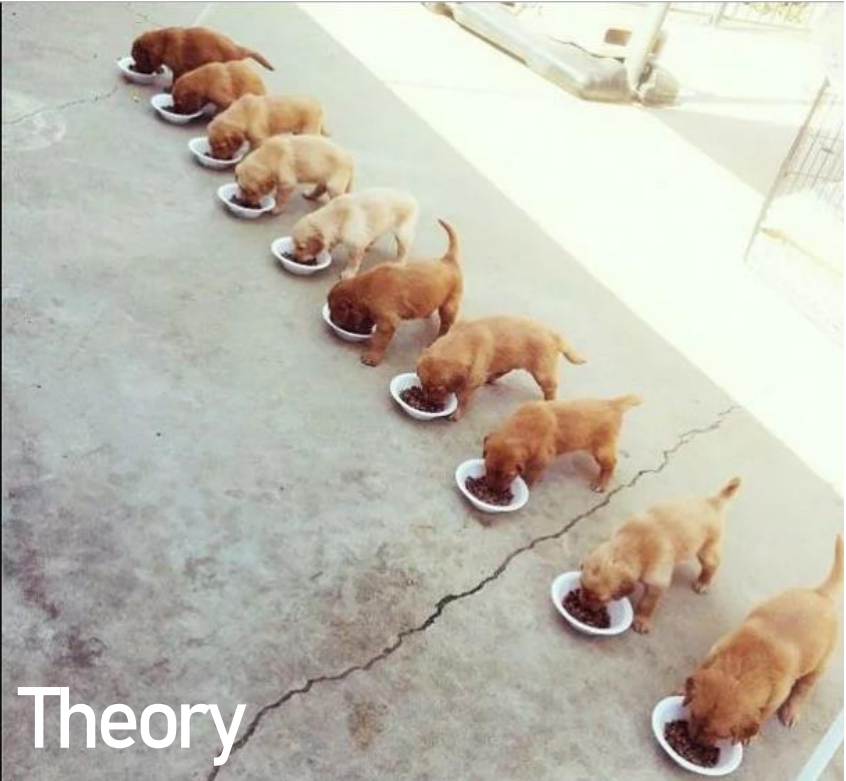


(Pictures from 9gag.com)

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)

Parallel programming

- **Not let this to happen** →
- **Write code that is**
 - High performance
 - Easy to debug



Make parallelism simple – some basic concepts

- **Shared memory**
 - All processors share the memory
 - They may or may not share caches – will be covered later
- **Design parallel algorithms without knowing the number of processors available**
 - It's generally hard to know # available processors
- **Scheduler: bridge your algorithm and the OS**
 - Your algorithm specifies the logical dependency of parallel tasks
 - The scheduler maps them to processors
 - Usually also dynamic

How can we write parallel programs?

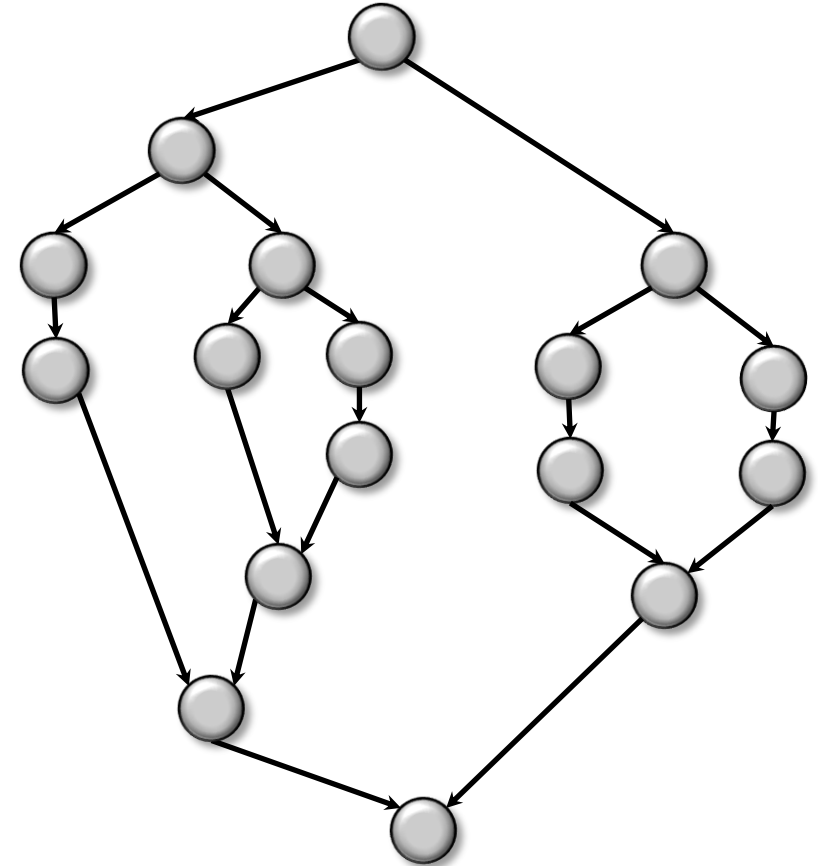
What your program tells the scheduler

- **Fork-join model**

- At any time, your program can **fork** a number of tasks and let some parallel threads execute them
- After they all return, they are synchronized by a **join** operation
- Fork-join can be nested

- **Most commonly used primitives**

- Execute two tasks in parallel (**parallel_do**)
- Parallel for-loop: execute n tasks in parallel (**parallel_for**)



Fork-join parallelism

As long as you can design a parallel algorithm in fork-join, implementing them requires very little work on top of your sequential C++ code

- **Supported by many programming languages**

```
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
```

- **Cilk/cilk+ (silk – thread)**

- Based on C++
- Execute two tasks in parallel
 - do_thing_1 can be done in parallel in another thread
 - do_thing_2 will be done by the current thread
- Parallel for-loop: execute n tasks in parallel
 - For cilk, it first forks two tasks, then four, then eight, ... in $O(\log n)$ rounds

Fork

Join

```
cilk_spawn do_thing_1;
do_thing_2;
cilk_sync;
```

```
cilk_for (int i = 0; i < n; i++) {
    do_something;
}
```

Cilk

- The name comes from silk because “**silk and thread**”
- A quick brain teaser: what is the difference/common things between *string* and *thread*?
 - If you don't know what am asking / find they have nothing in common, you must be a programmer
 - They are both thin, long cords

Fork-join parallelism

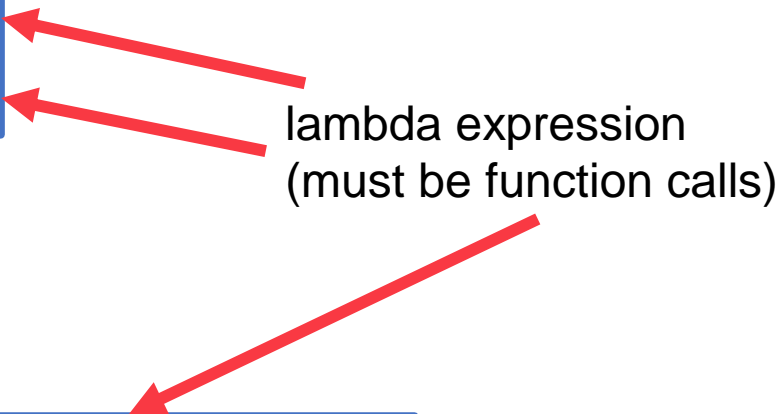
- A lightweight library: PBBS (Problem-based benchmark suite)
- Code available at: <https://github.com/cmuparlay/pbbslib>

```
#include "pbbslib/utilities.h"
```

You can also use cilk or openmp to compile your code

```
par_do([&] () {do_thing_1;},  
      [&] () {do_thing_2;});
```

lambda expression
(must be function calls)

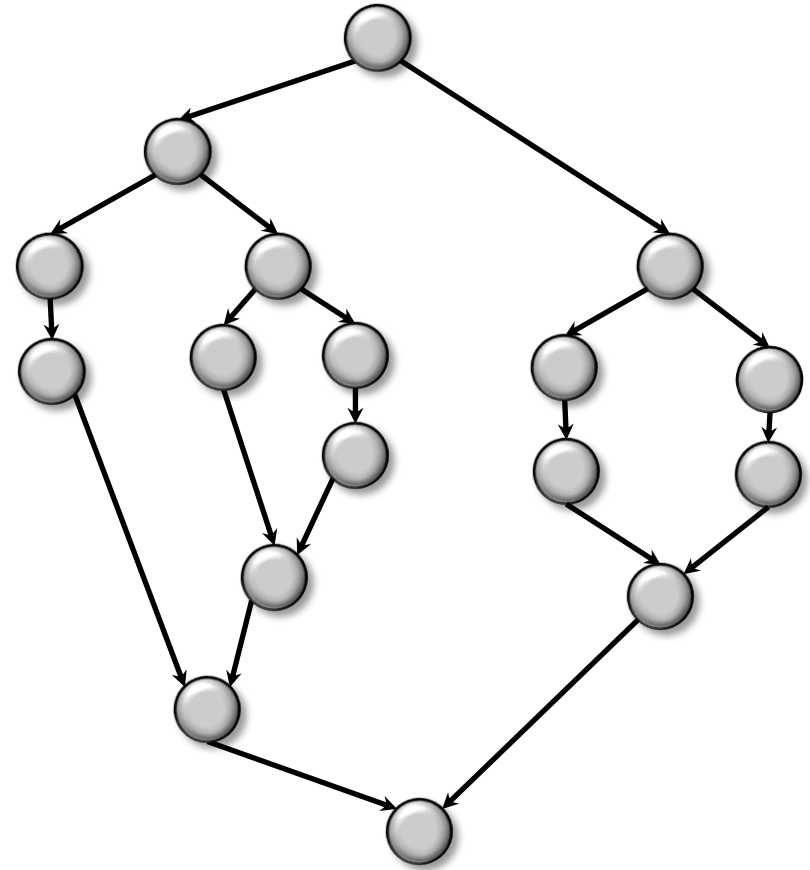


```
parallel_for (0, 100, [&] (int i) {Do_something});
```

Cost model work and span

Cost model: work-span

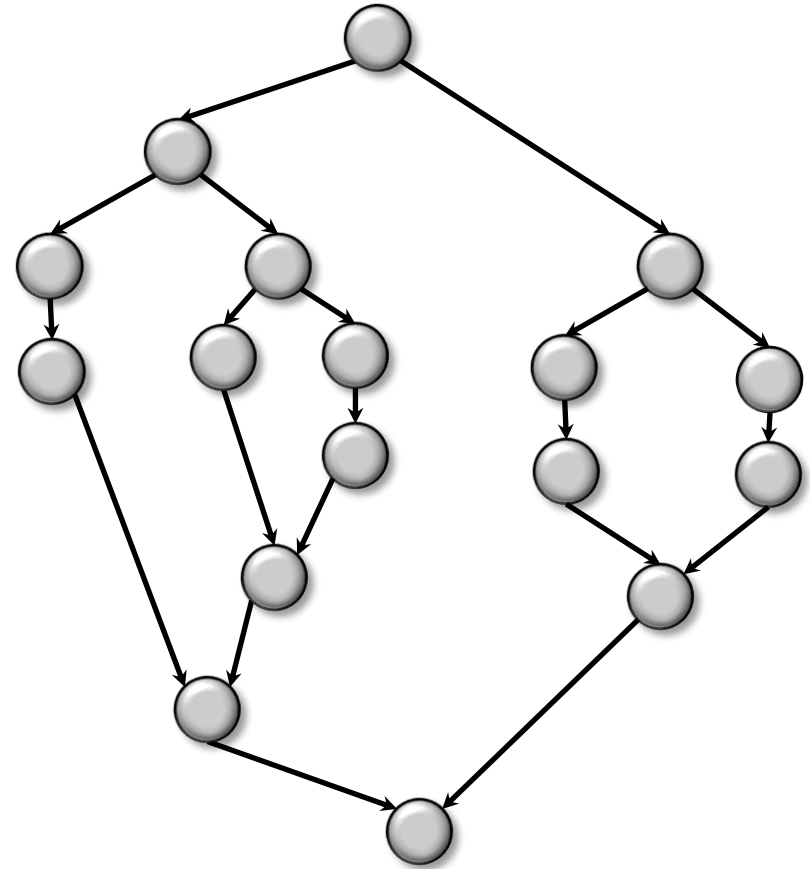
- **For all computations, draw a DAG**
 - A->B means that B can be performed only when A has been finished
 - It shows the dependency of operations in the algorithm
- **Work: the total number of operations**
- **Span (depth): the longest length of chain**



Work = 17
span = 8

Cost model: work-depth

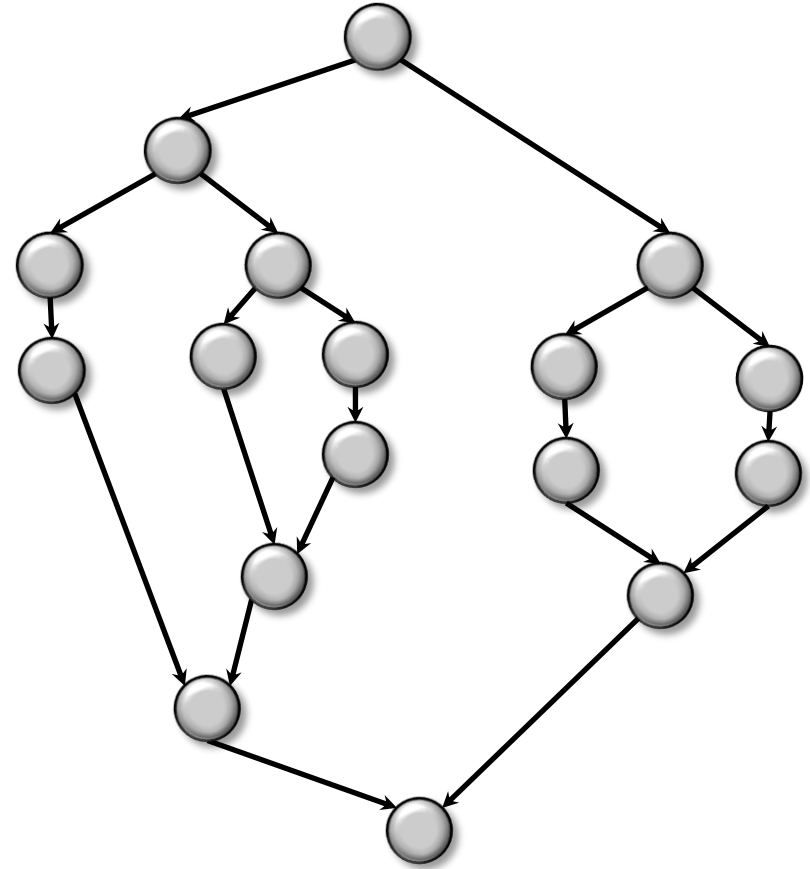
- **Work:** The total number of operations in the algorithm
 - Sequential running time when the algorithm runs on one processor
 - Work-efficiency: the work is (asymptotically) no more than the best (optimal) sequential algorithm
 - Goal: make the parallel algorithm efficient when a small number of processor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



T_1

Cost model: work-depth

- **Span (depth): The longest dependency chain**
 - Total time required if there are infinite number of processors
 - Make it polylog(n) or $O(n^\epsilon)$
 - Goal: make the parallel algorithm faster and faster when more and more processors are available - scalability



T_∞

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

Schedule a parallel algorithm with work W and span S

Can be scheduled in time $O\left(\frac{W}{p} + S\right)$ (w.h.p. for some randomized schedulers)

- p : number of processors
- Asymptotically, it is also the **lower bound**
- W/p term: even though all processors are **perfectly-balanced full-loaded**, we need this amount of time
- S term: even though we have **an infinite number** of processors, we need this amount of time
- More details will be given in later lectures

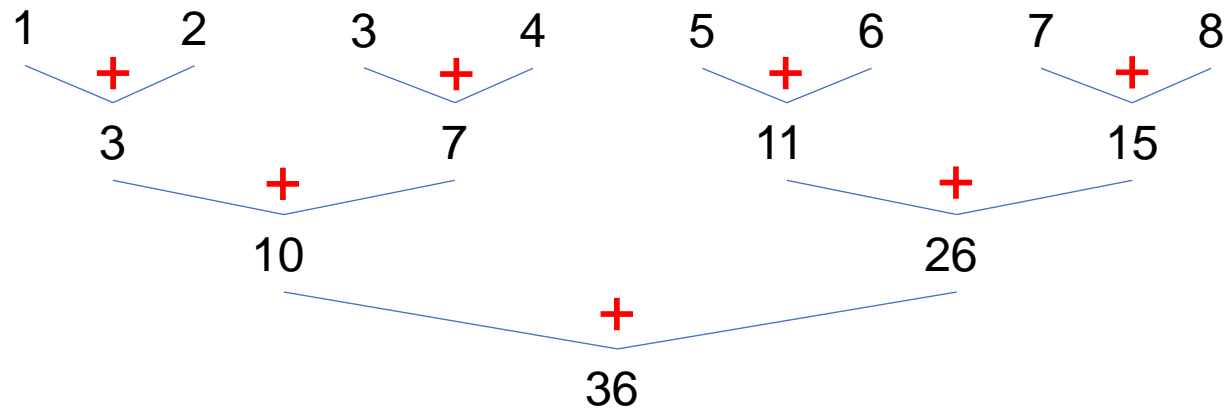
Parallelism / speedup

- T_1 : running time on one thread, work
- T_∞ : running time on unlimited number of processors, span
- **Parallelism** = $\frac{T_1}{T_\infty}$
- **Speedup:**
 - Sequential running time / parallel running time
 - Self-speedup: parallel code running on one processor / parallel code running on p processors

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

Warm-up

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



Work: $O(n)$
Span: $O(\log n)$

```
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;  
}
```

Implementing parallel reduce in cilk

Pseudocode

```
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;  
}
```

Code using Cilk

```
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; }
```

It is still valid is running sequentially,
i.e., by one processor

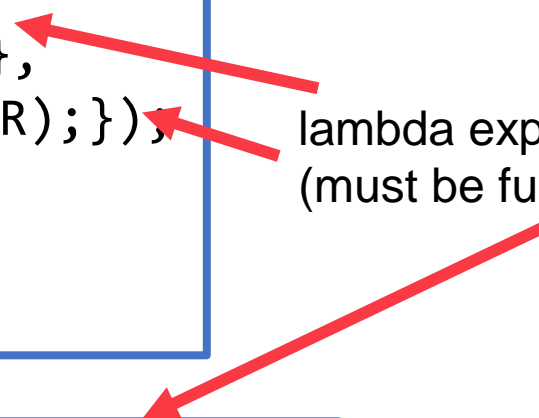
Implementing parallel reduce in PBBS

```
#include "pbbslib/utilities.h"
```

You can also use cilk or openmp to compile your code

```
void reduce(int* A, int n, int& ret) {  
    if (n == 1) ret = A[0]; else {  
        int L, R;  
        par_do([&] () {reduce(A, n/2, L);},  
              [&] () {reduce(A+n/2, n-n/2, R);});  
        ret = L+R;  
    }  
}
```

lambda expression
(must be function calls)



```
parallel_for (0, 100, [&] (int i) {A[i] = i;});
```

Testing parallel reduce

```
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; }
```

Input of 10^9 elements

Sequential running time

Parallel code on 24 threads*

Parallel code on 4 threads

Parallel code on 1 thread



Self-speedup:
13.29

Code was running on course server

*: 12 cores with 24 hyperthreads

Testing parallel reduce

```
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; }  
}
```

Input of 10^9 elements

Sequential running time	0.61 s
Parallel code on 24 threads*	4.51 s
Parallel code on 4 threads	17.14 s
Parallel code on 1 thread	59.95 s



Speedup:
??

Code was running on course server

*: 12 cores with 24 hyperthreads

Implementation trick 1: coarsening

Coarsening

- Forking and joining are costly – this is the overhead of using parallelism
- If each task is too small, the overhead will be significant
- Solution: let each parallel task get enough work to do!

```
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 reduce(int* A, int n) {  
    if (n < threshold) {  
        int ans = 0;  
        for (int i = 0; i < n; i++)  
            ans += A[i];  
        return ans; }  
    int L, R;  
    L = cilk_spawn reduce(A, n/2);  
        R = reduce(A+n/2, n-n/2);  
    cilk_sync;  
    return L+R; }
```


Testing parallel reduce with coarsening

Input of 10^9 elements

Algorithm	Threshold	Time
Sequential running time	–	0.61s
Parallel code on 24 threads	100	0.27s
Parallel code on 24 threads	10000	0.19s
Parallel code on 24 threads	1000000	0.19s
Parallel code on 24 threads	10000000	0.22s

Best threshold depends on the machine parameters and the problem

Testing parallel reduce with coarsening

Input of 10^9 elements

Algorithm	Threshold	Time
Sequential running time	–	0.61s
Parallel code on 24 threads	100	0.27s
Parallel code on 24 threads	10000	0.19s
Parallel code on 24 threads	1000000	0.19s
Parallel code on 24 threads	10000000	0.22s

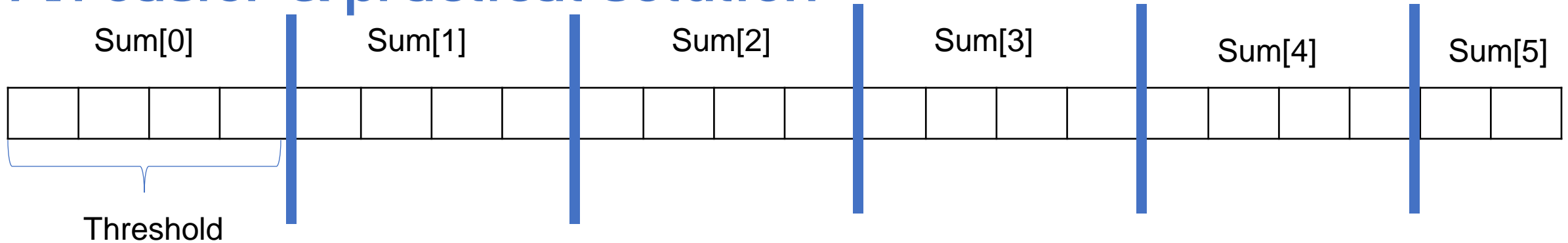
In the best case using 24 threads improves the performance by about 3 times.

- The reduce algorithm is I/O bounded (will be discussed in the course later)
- # threads is small
- Can expect better speedup in algorithms like matrix multiplication

Divide-and-conquer + coarsening

- **Coarsening means that we don't want each subtask running in parallel to be too small**
- **Is there an alternative way to make it simpler?**

An easier & practical solution



1. Divide the array into t blocks
2. In parallel, compute the sum of each block. Within each block the sum is computed sequentially.
3. Add all sums for all blocks together, sequentially

How many blocks should we use?

How many blocks should we use?

- **p as the number of available processors of the machine?**
 - May not be a good idea – load balancing
 - If any of these processors is unavailable, an extra round is needed
 - If any of them is blocked or is slow – the slowest one is the bottleneck
- **Usually we can use cp blocks, for some constant c**
 - E.g., for $c \approx 10 \sim 100$
 - Or using $c = \sqrt{n}$
 - Having more tasks allows for more flexibility in scheduling
 - State-of-the-art schedulers can do a good job

Testing parallel reduce with coarsening

Input of 10^9 elements

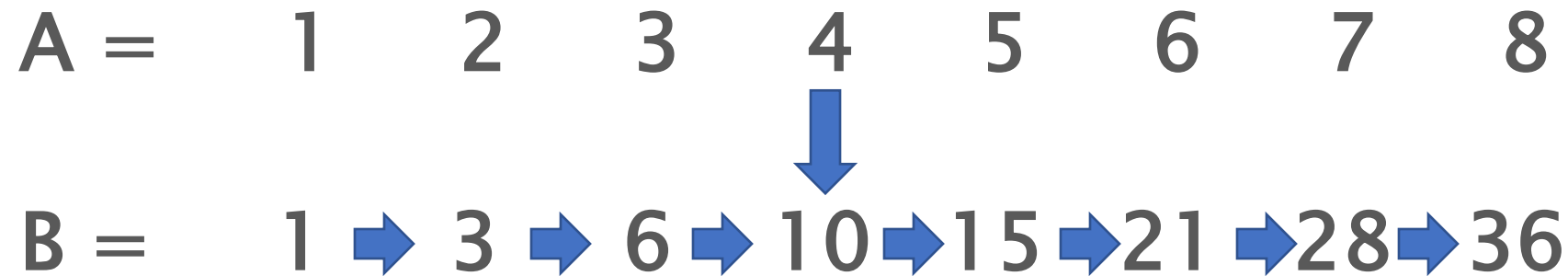
Algorithm	#blocks	Time
Sequential running time	–	0.61 s
Parallel code on 24 threads	100	0.26 s
Parallel code on 24 threads	1000	0.19 s
Parallel code on 24 threads	100000	0.19 s
Parallel code on 24 threads	10000000	0.21 s

For more complicated algorithms, the best #blocks can be different

Prefix Sum (Scan)

Prefix sum

The most widely-used building block in parallel algorithm design

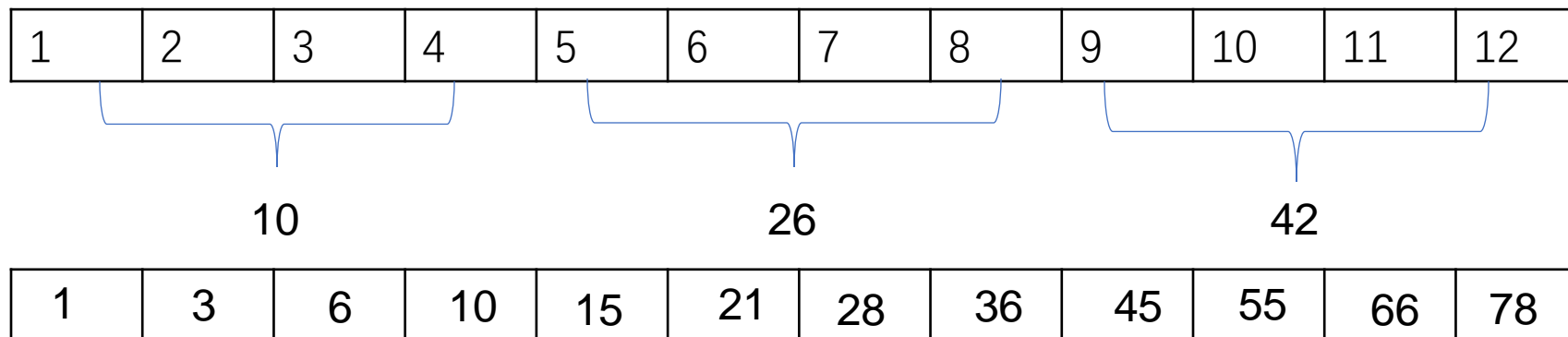


Prefix sum algorithms

- We can design algorithms to make it work-efficient with $O(\log n)$ depth
- But again we need coarsening to avoid small parallel tasks
- Can we also use the blocking idea?

A parallel scan algorithm

- Divide the array into t blocks, each with size about b
- Compute the sum of each block in an array B , in parallel (sequential within each block)
- Compute the prefix sum of B sequentially, and write the prefix sum of B to the b -th, $2b$ -th, ... slots in the output
- Fill in the rest of each block in parallel – run a sequential prefix sum for each block, with an offset decided by the prefix sum at the end of the previous block



Abstract reduce and scan

- **For both reduce and scan, the binary operation can be any associative operations**
 - Not necessary to be addition on integers
 - Real numbers, Boolean values, ...
 - Multiply, bit operation (and, or, xor, ...), ...
 - For a sequence of matrices, define the operation as matrix multiplication
 - Compute the product of multiple matrices
 - For a sequence of sets, define the operation as union/intersection

Summary

- **Scheduler:**
 - Help you map your parallel tasks to processors
- **Fork-join**
 - Fork: create several tasks that will be run in parallel
 - Join: after all forked threads finish, synchronize them
- **Work-span**
 - Work: total number of operations, sequential complexity
 - Span (depth): the longest chain in the dependence graph
- **Writing code in parallel**
 - `Parallel_do`: execute two tasks in parallel
 - `Parallel_for`: execute a for-loop in parallel
 - Cilk and PBBS based on C++

Summary

- **Reduce/scan algorithms**
 - Divide-and-conquer or blocking

- **Coarsening**
 - Avoid overhead of fork-join
 - Let each subtask large enough