

# Parallel Algorithms

Yan Gu

# Course announcement

- **Problem-solving training 1 is due on Wednesday**
- **Only 10 of you (less than half) have already solved some problems**
  - 3 solved all problems
  - Another 2 solved two problems
- **As a comparison, we also gave out 3 problems (but harder) in CS 218 with the same deadline**
  - Over 80% have solved problems already
  - Over 60% have solved all problems

# Modified score-to-grade mapping

- **A+: very top performance in the class**
- **A: 85%**
- **A-: 82%**
- **B+: 78%**
- **B: 75%**
- **C: 68%**
- **D: 60%**
- **F: <60%**

# Course announcement

- **Performance-engineering assignment 1 is out**
- **60% written, practice the basic concepts and design parallel algorithms**
- **40% programming, implement the algorithms and engineer performance**
- **Deadline: Jan 20 (Wednesday)**

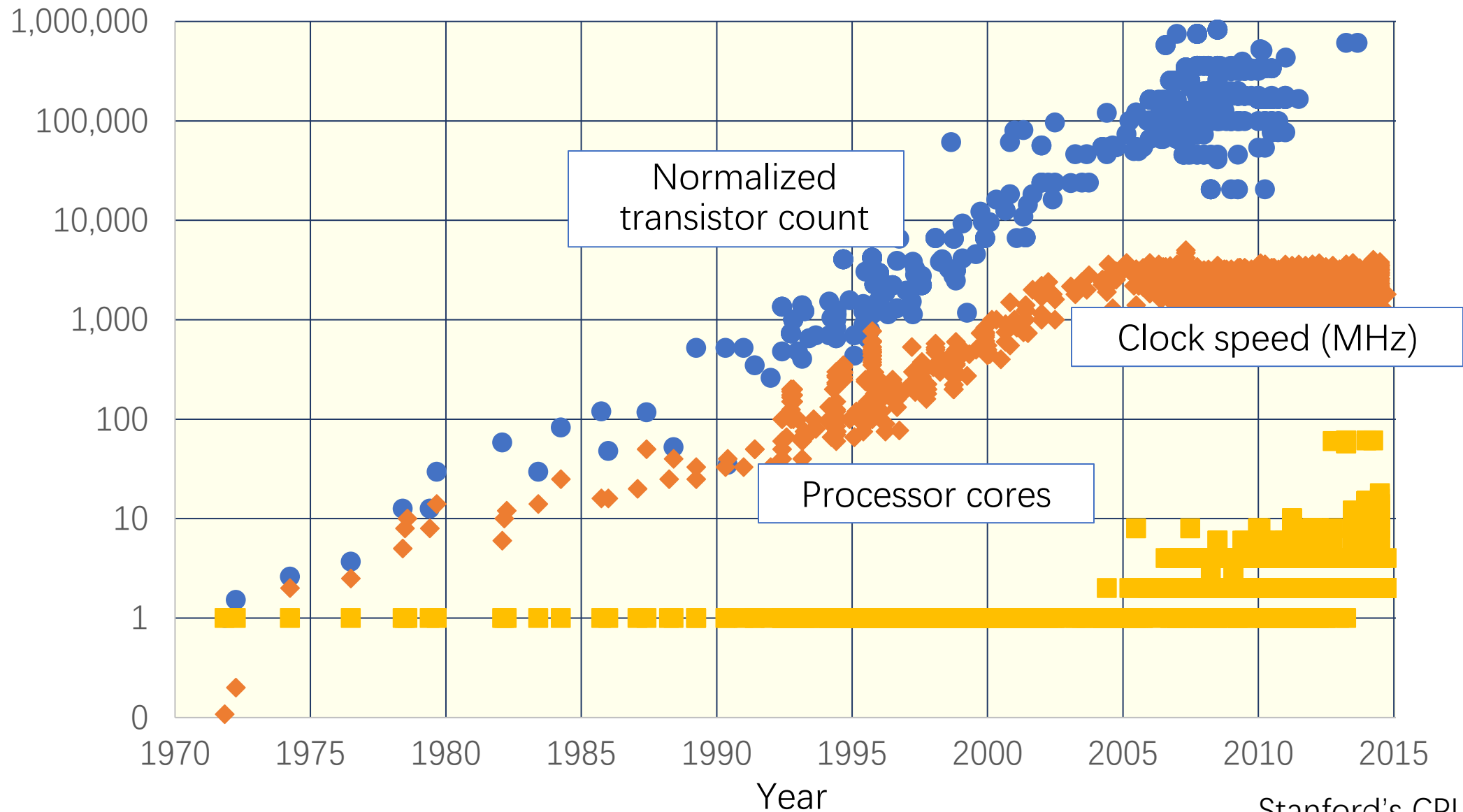
# For the programming part

- Each of you will be assigned to a lab machine WCH133-01~WCH133-47
- Reservation is via this link:  
[https://docs.google.com/spreadsheets/d/1QiDI8\\_IMRzlymicn6vQYwv-mauQxzDKMVgDryNVsGjo/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1QiDI8_IMRzlymicn6vQYwv-mauQxzDKMVgDryNVsGjo/edit?usp=sharing)
- **Final performance testing is on ti-05 (a 12-core machine)**
  - Only use ti-05 for reporting numbers, not for debugging
  - You get bonus candies if your solution is the fastest
- **Start earlier!!!**

# Parallel Algorithms

Yan Gu

# Technology Scaling After 2004



# Parallel machines



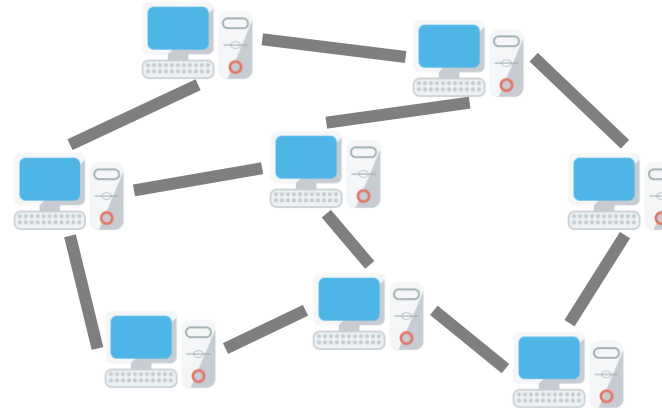
4 cores, 8 hyperthreading  
Usually \$700-\$1500



- ❖ 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



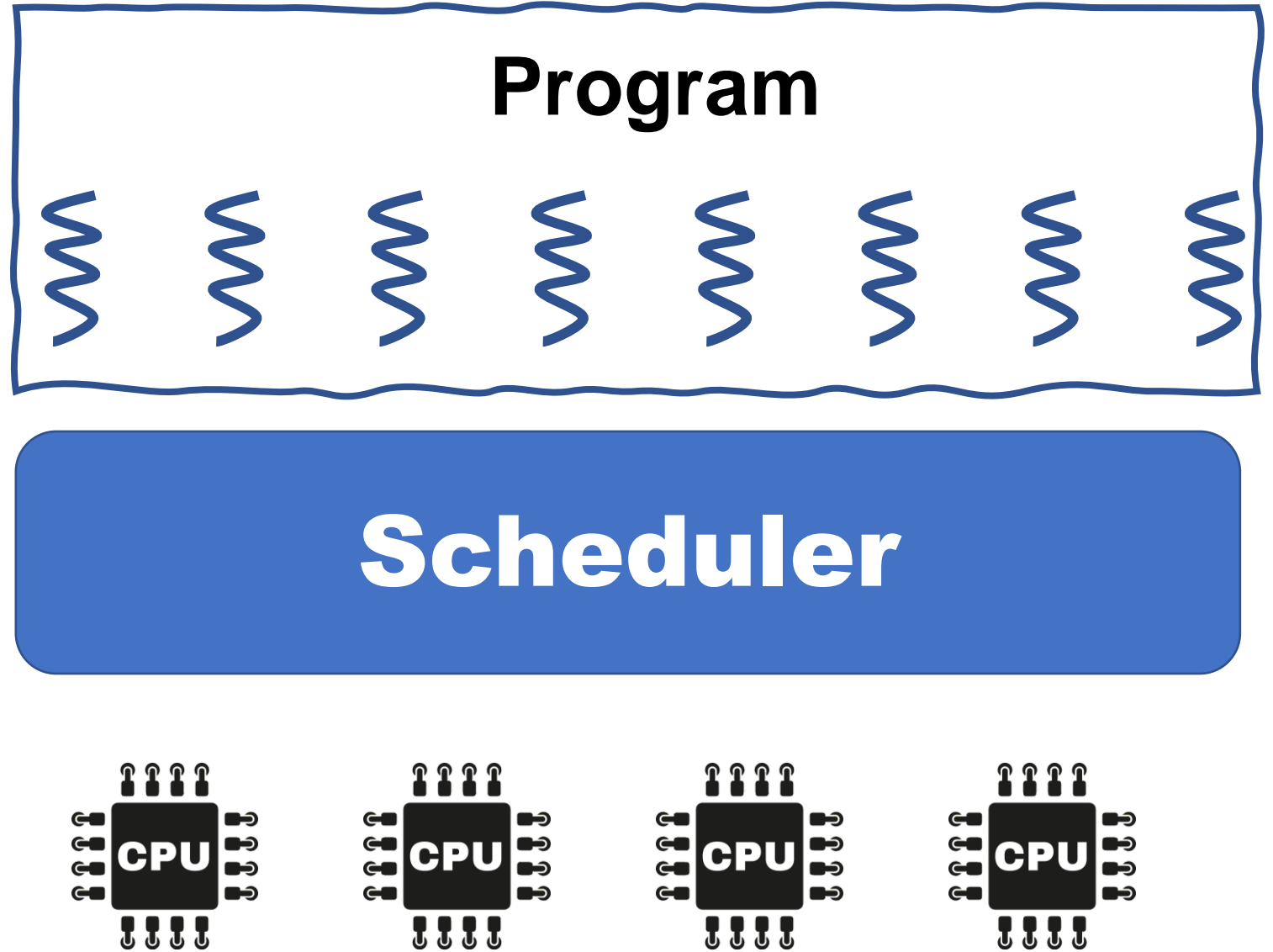
# Multi-core Programming

- **We need to learn theory:**
  - Making performance predictable
- **Not let this to happen →**



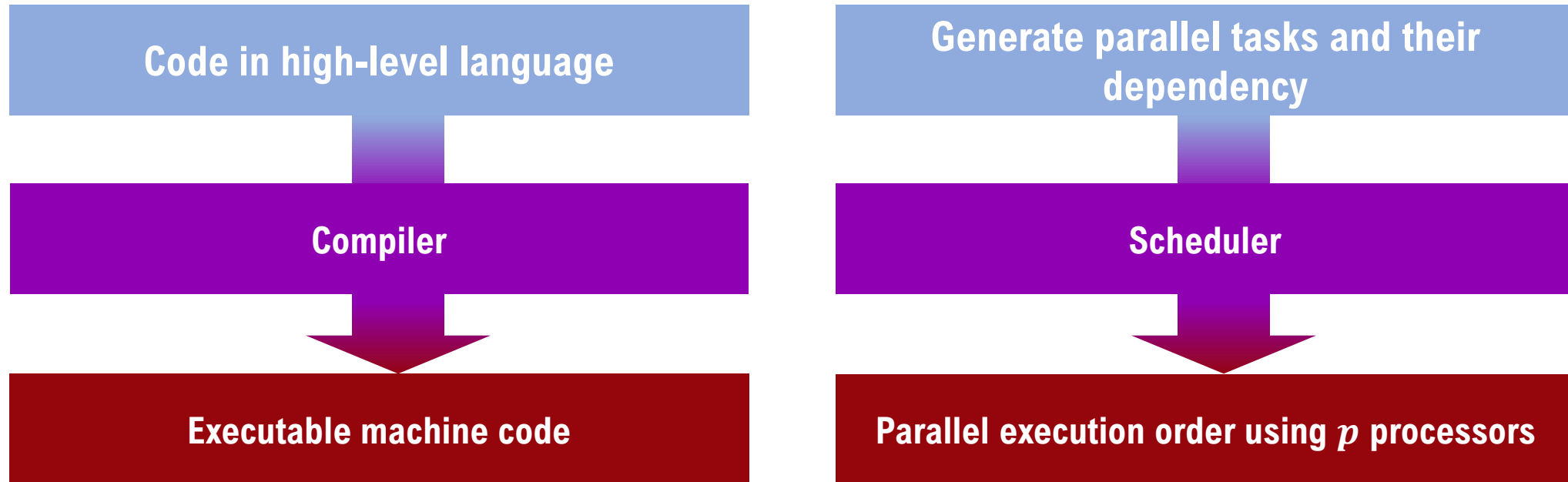
# Scheduler

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



# Scheduler

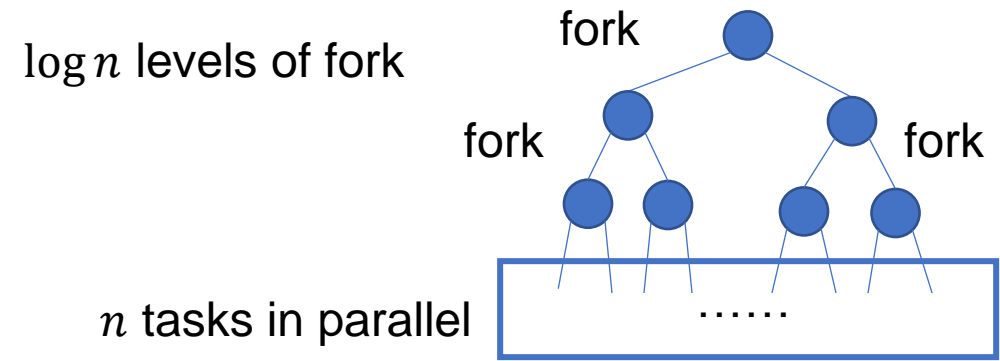
- Consider it as a compiler. Programmers then only need to focus on high-level algorithm design



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

# 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

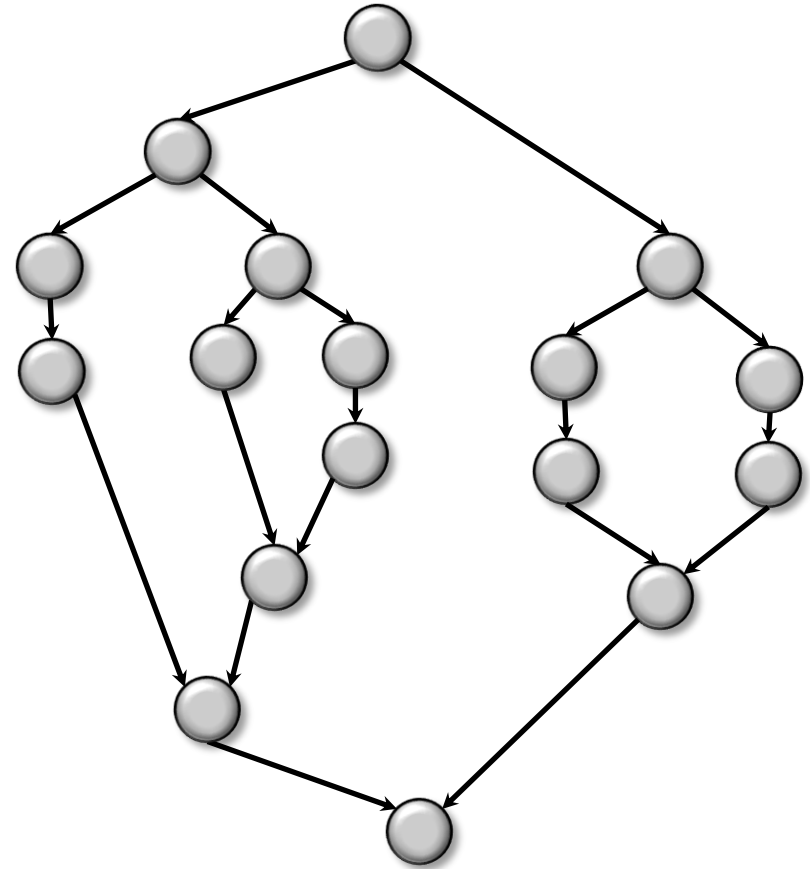
- Simple for theoretical analysis – we'll see in a while
- Simple for programming – almost exactly the code!

```
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;  
}
```

```
1  #include <iostream>  
2  #include <cstdio>  
3  #include <stdlib.h>  
4  #include <cilk/cilk.h>  
5  #include <cilk/cilk_api.h>  
6  using namespace std;  
7  
8  int reduce(int* A, int n) {  
9      if (n == 1) return A[0];  
10     int L, R;  
11     L = cilk_spawn reduce(A, n/2);  
12     R = reduce(A+n/2, n-n/2);  
13     cilk_sync;  
14     return L+R;  
15 }  
16  
17 int main() {  
18     int n = atoi(argv[1]);  
19     int* A = new int[n];  
20     cilk_for (int i = 0; i < n; i++) A[i] = i;  
21     cout << reduce(A, n) << endl;  
22  
23     return 0;  
24 }
```

# Cost model: work-span

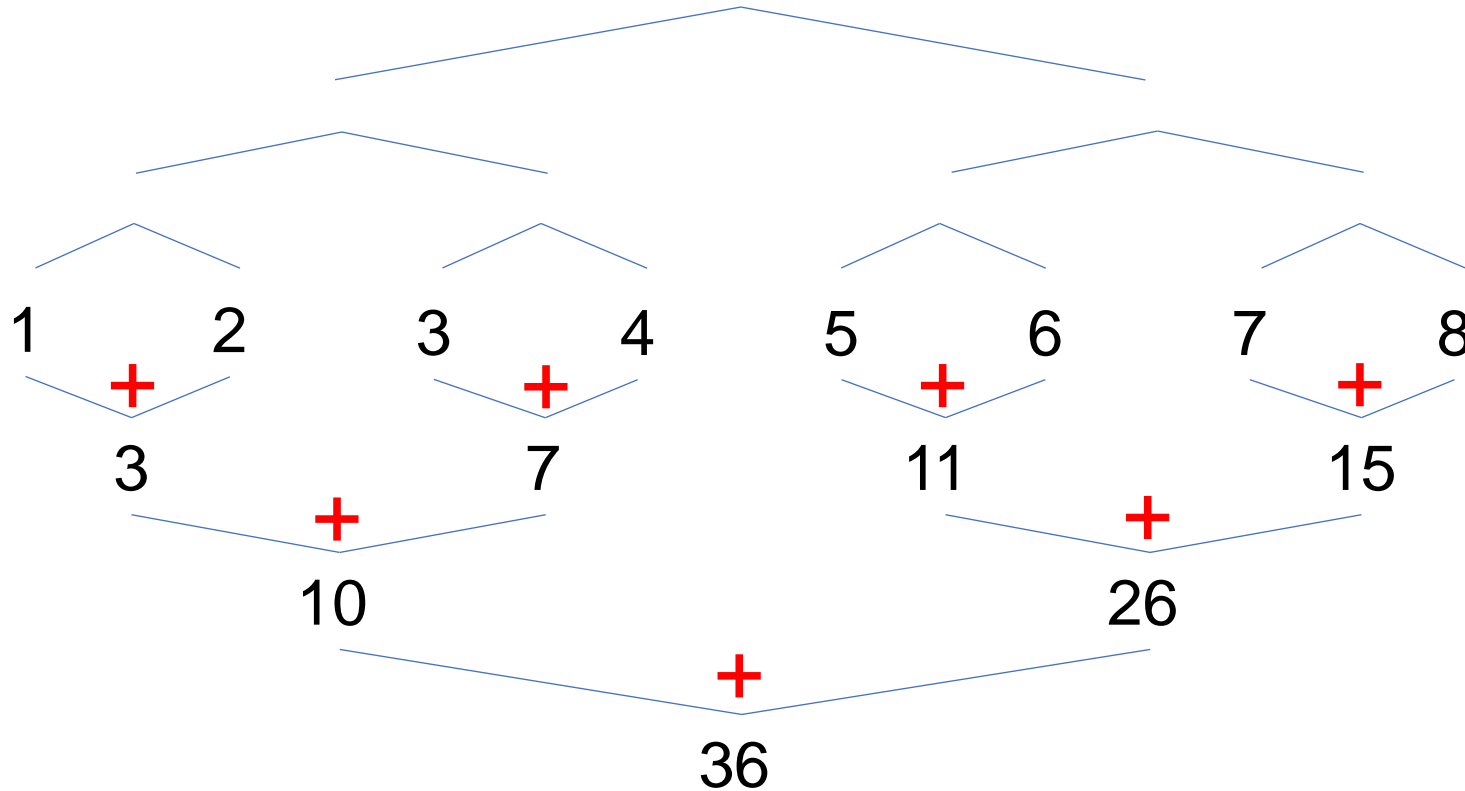
- **For all computations, draw a DAG**
  - $A \rightarrow B$  means that B can be performed only when A has been finished
- **Work: the total number of operations**
- **Span (depth): the longest length of chain**



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

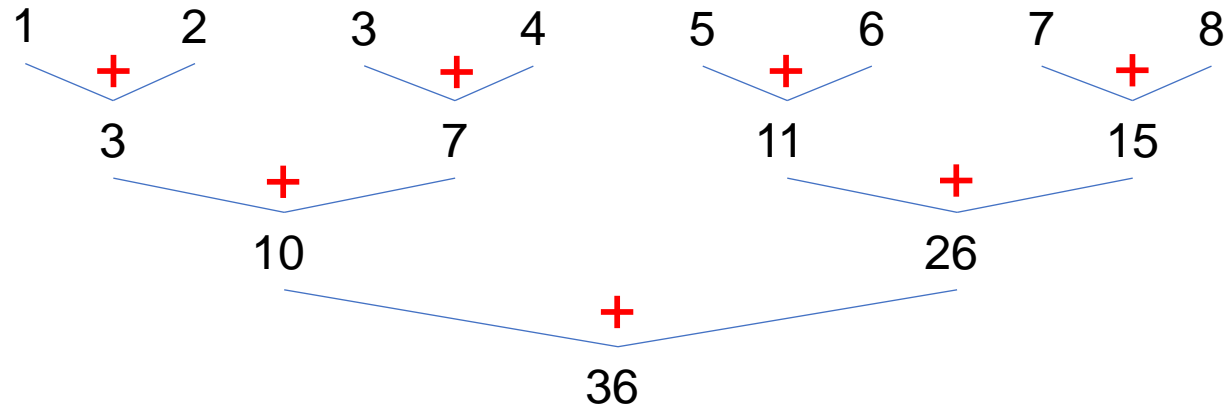
# Computational DAG

```
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

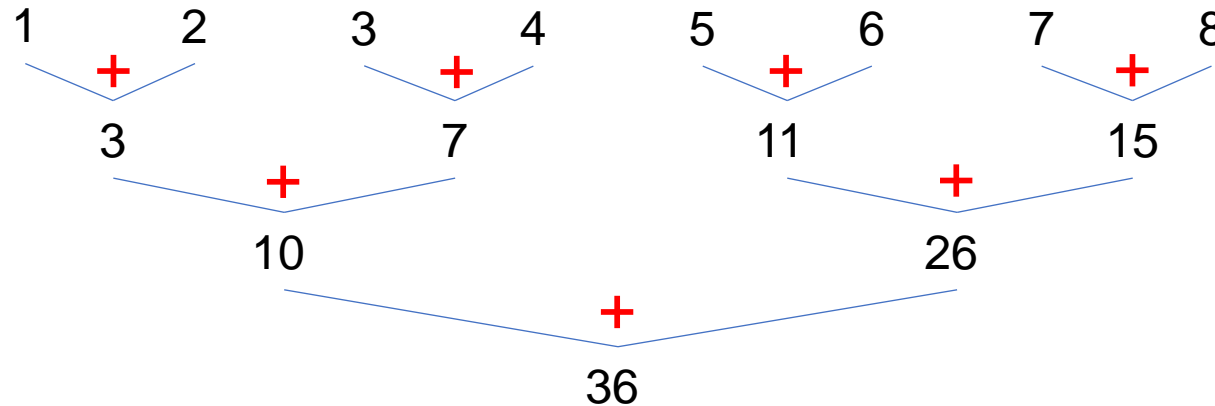


```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

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;  
}
```

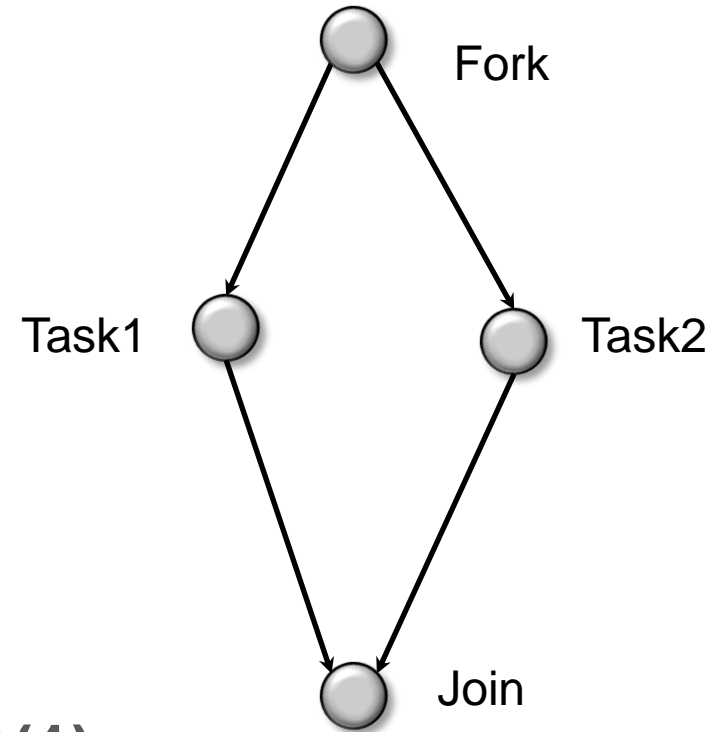
Span:  $O(\log n)$

- **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
  - $\text{Work} = \text{work of Task1} + \text{work of Task2} + O(1)$
  - $\text{Span} = \max(\text{span of Task1}, \text{span of Task2}) + O(1)$

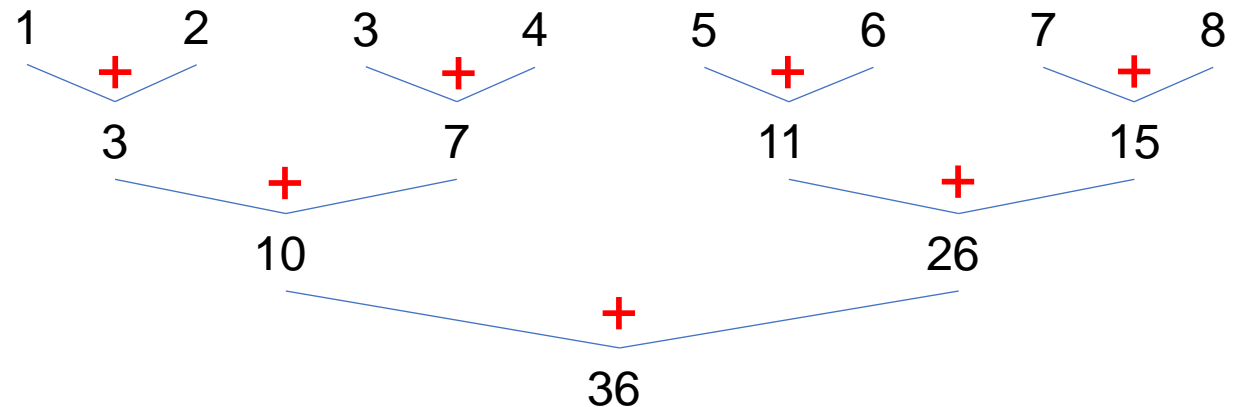


# Compute work and span

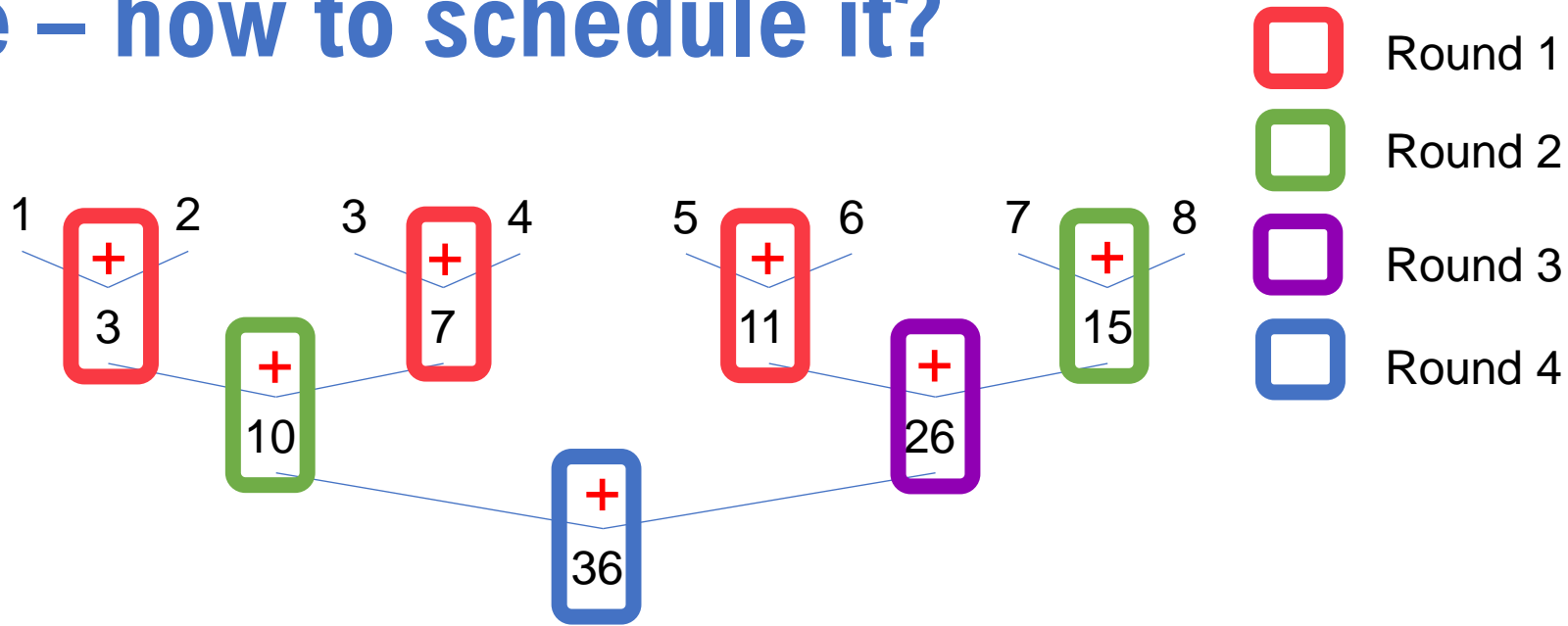
- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1)$
- $\Rightarrow W(n) = \Theta(n)$

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

```
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;  
}
```



# 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

# Programming fork-join parallelism

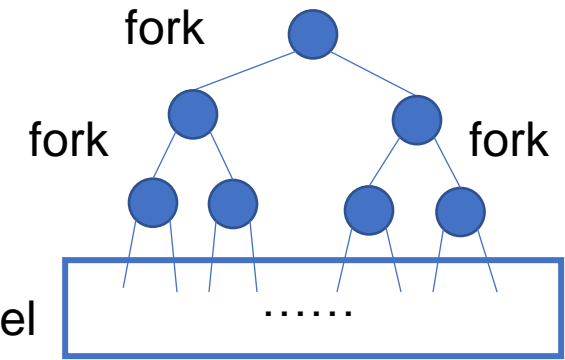
# Binary Fork-Join Model

- 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

$\log n$  levels of fork

$n$  tasks in parallel



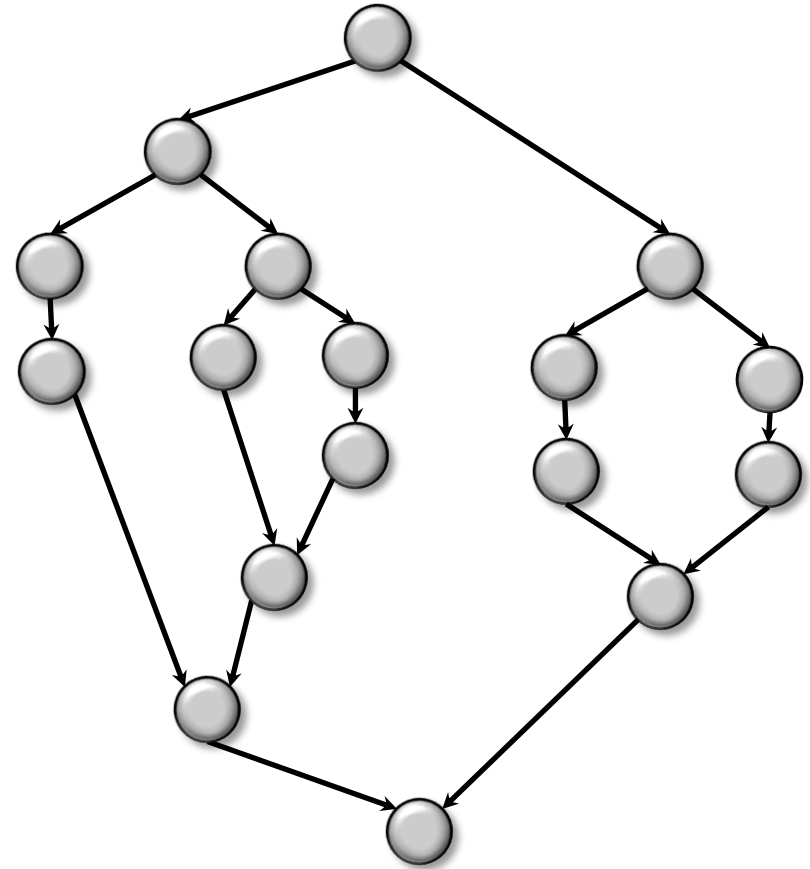
```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

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



# Cost model: work-span

- **For all computations, draw a DAG**
  - $A \rightarrow B$  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**

# 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
- 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

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

Fork →

```
cilk_spawn do_thing_1;
do_thing_2;
Join → cilk_sync;
```

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

# Cilk

- The name comes from silk because “**silk 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



File:Spool of string.jpg - Wikipedia  
en.wikipedia.org



Pack of 100 Yo-Yo Strings 50...  
latiadadelyoyo.com



Jute Twine String Natural 250...  
officemax.co.nz · In stock



Irish Linen Bookbinding Thread...  
talasonline.com · In stock



Amazon.com: GOELX Silk Thread Shiny and ...  
amazon.com



Browse Threads - WonderFil  
wonderfil.ca

# Fork-join parallelism

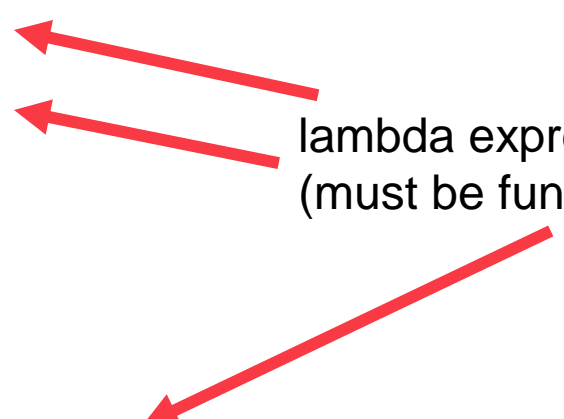
- A lightweight library: PBBS (Problem-based benchmark suite)
- Code available at: <https://github.com/cmuparlay/pbbslib>

```
#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});
```

# 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

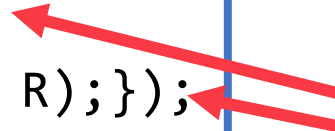
# Implementing parallel reduce in PBBS

```
#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	0.61s
Parallel code on 24 threads*	4.51s
Parallel code on 4 threads	17.14s
Parallel code on 1 thread	59.95s

Speedup:  
??



Code was running on course server

\*: 12 cores with 24 hyperthreads

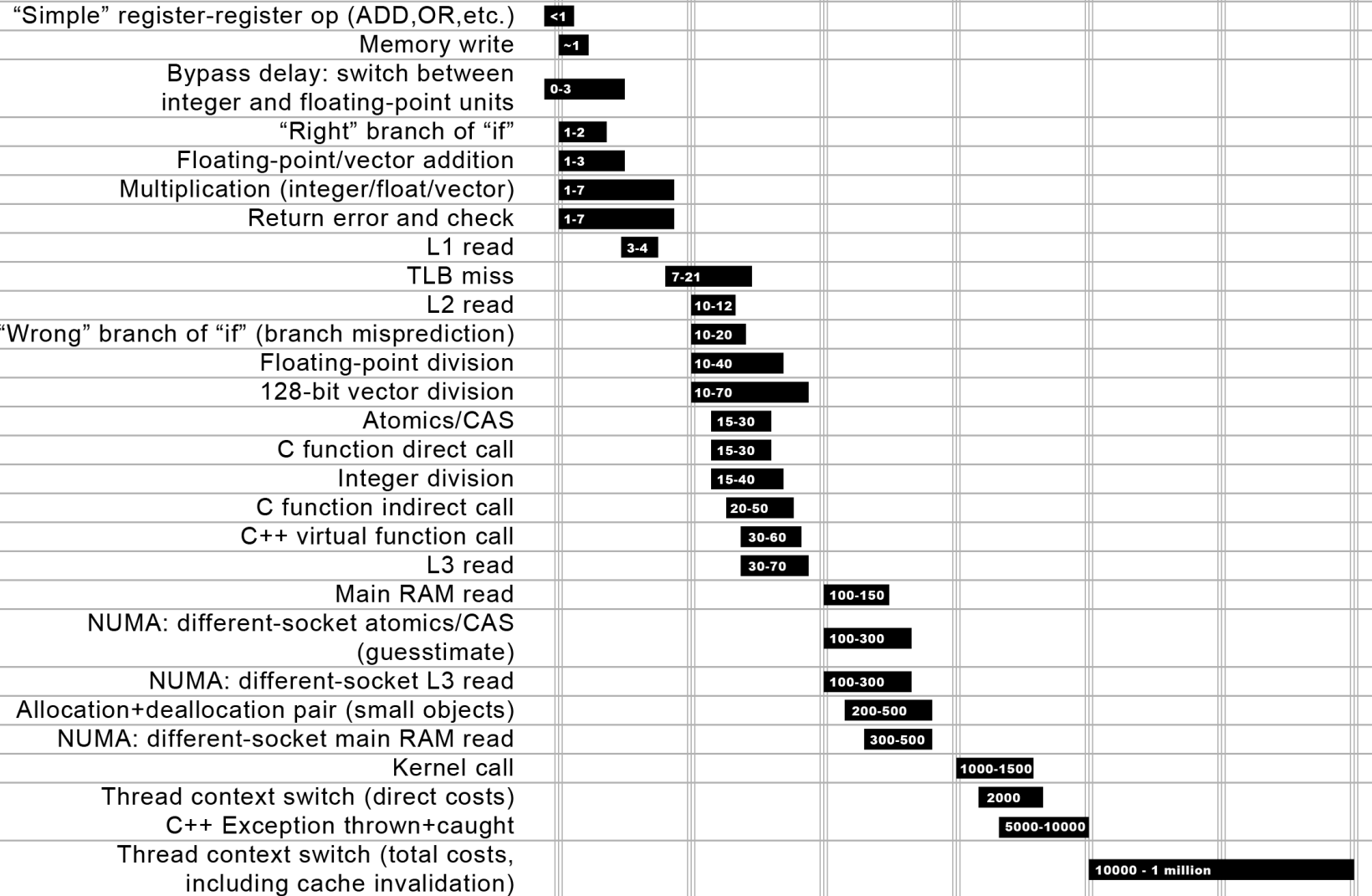




# Not all CPU operations are created equal

Operation Cost in CPU Cycles

$10^0$   $10^1$   $10^2$   $10^3$   $10^4$   $10^5$   $10^6$



- A cilk-spawn is about 100 cycles

Distance which light travels while the operation is performed

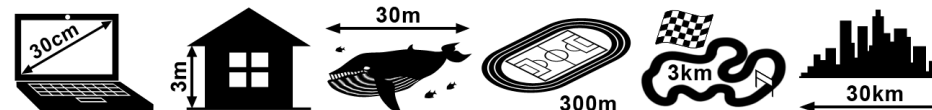


Image from ithare.com:

<http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/>

# **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.

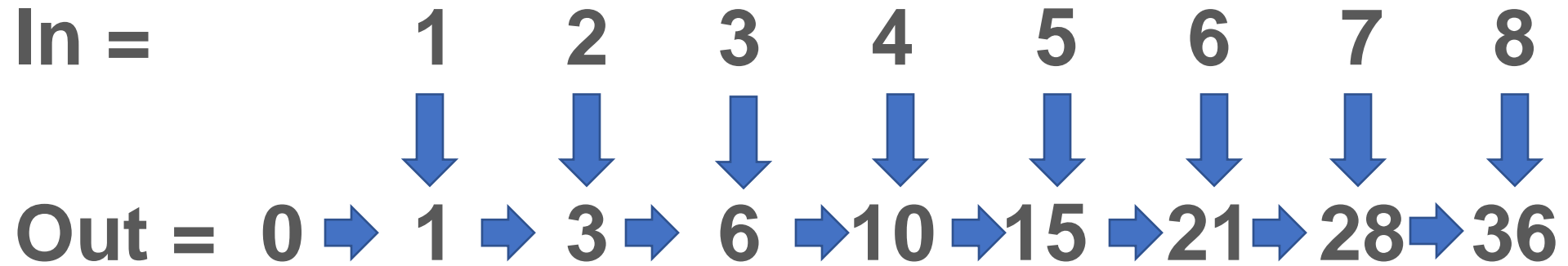
- The reduce algorithm is I/O bounded (will be discussed in the course later)
- # threads is small
- Can expect better speedup in algorithms like matrix multiplication

# 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?**

# Prefix Sum (Scan)

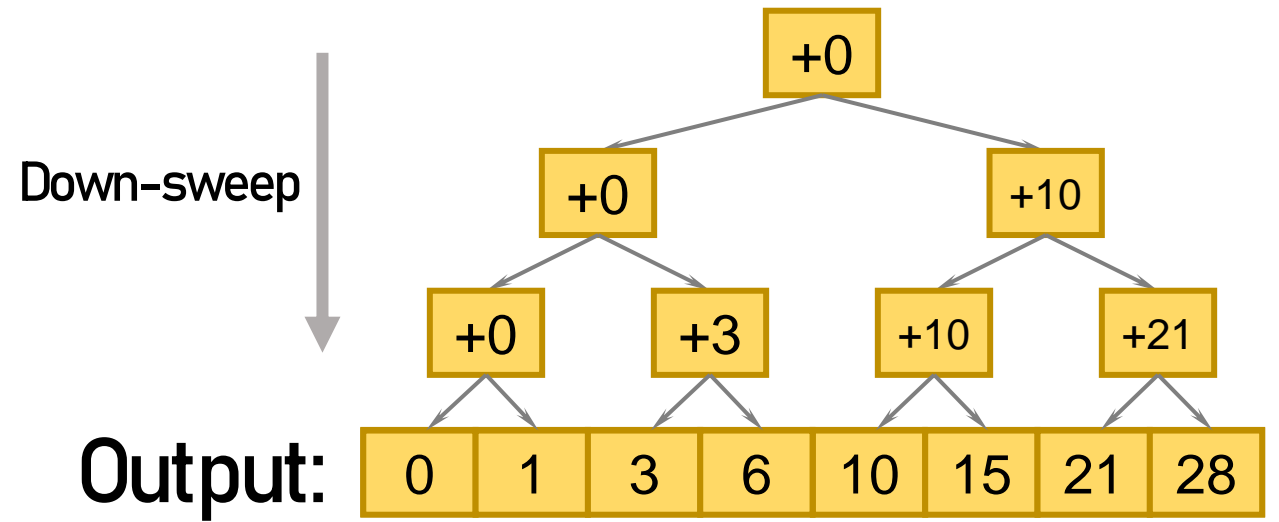
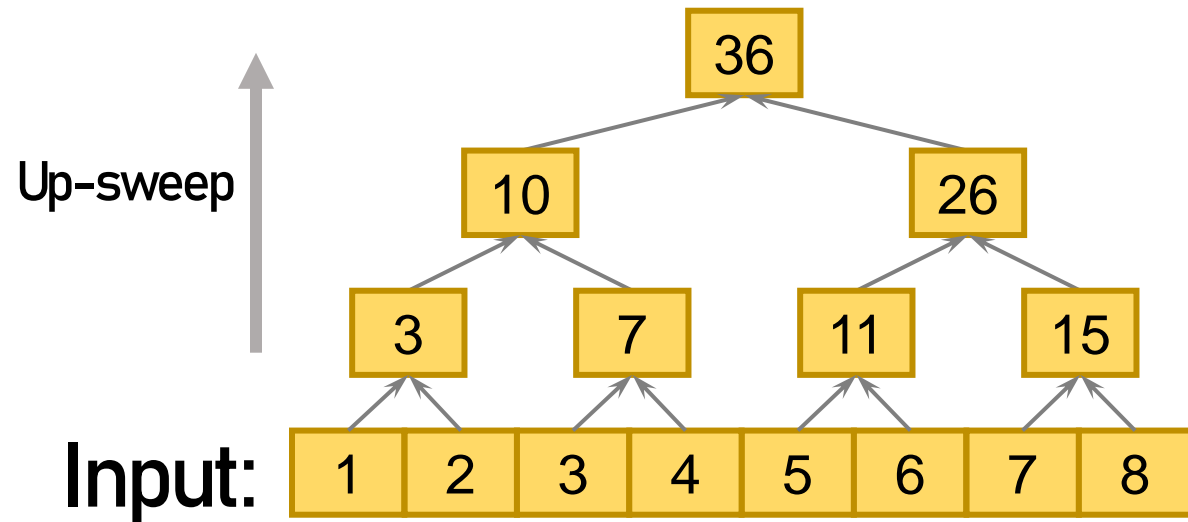
# Prefix sum



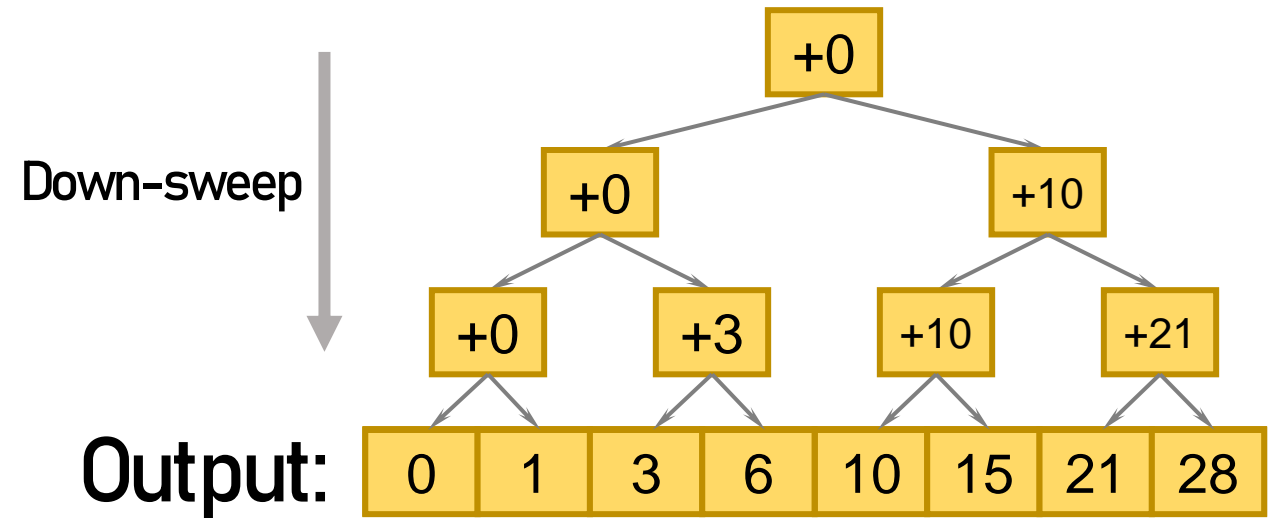
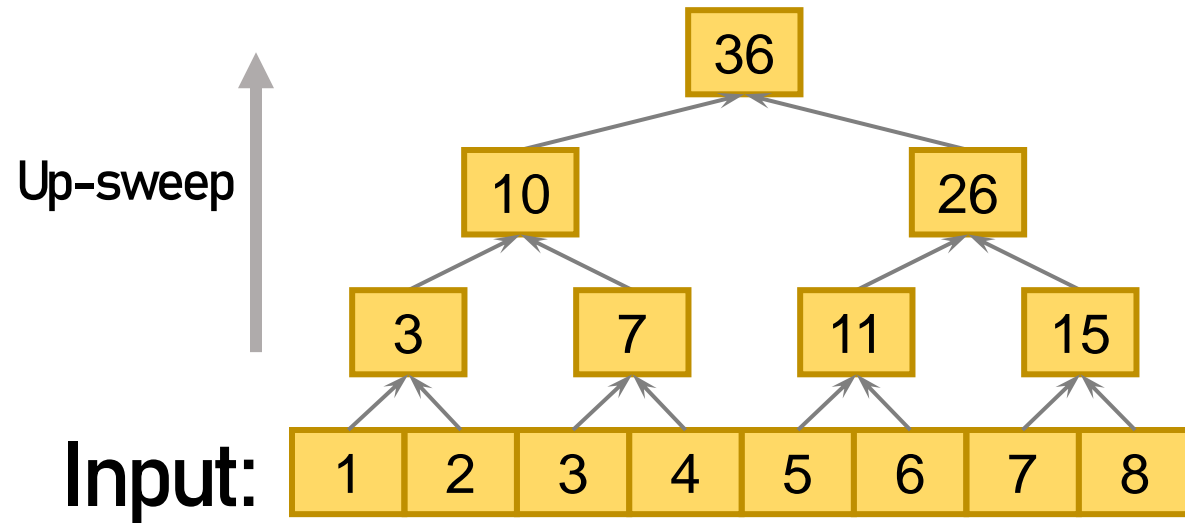
The most widely-used building block in parallel algorithm design



# A divide-and-conquer algorithm



# Pseudocode for scan



```

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 A[0..n]=L+R;
}
    
```

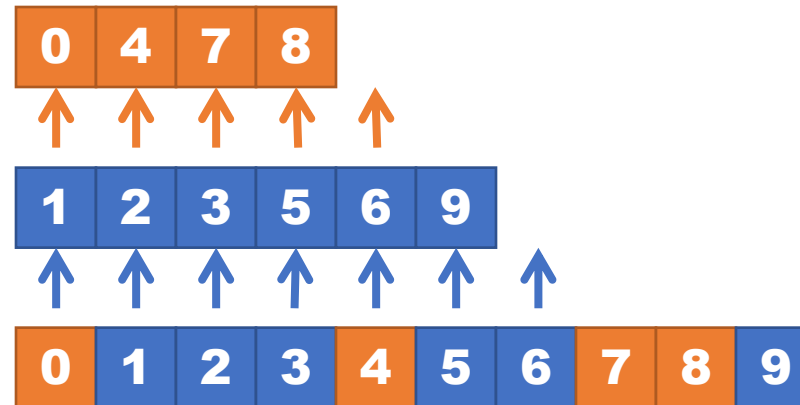
```

scan(A, n, ps) {
    if (n == 1) { A[0]=ps; return;}
    In parallel:
        scan(A, n/2, ps);
        scan(A+n/2, n-n/2, ps+LeftSum);
    }
}
    
```

# Parallel merge

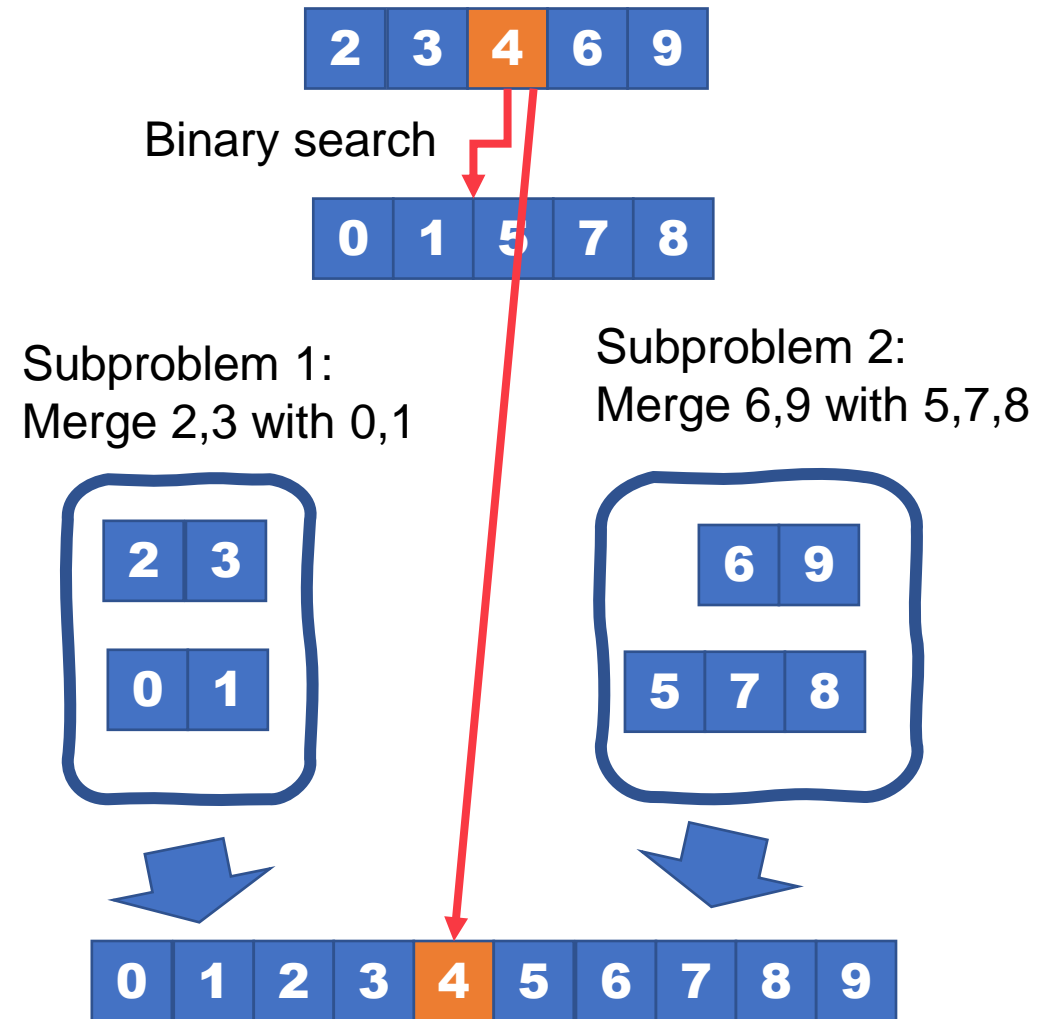
# Parallel merging

- Given two sorted arrays, merge them into one sorted array
- Sequentially, use two moving pointers



# A parallel merge algorithm

- Find the median  $m$  of one array
- Binary search it in the other array
- Put  $m$  in the correct slot
- Recursively, in parallel do:
  - Merge the left two sub-arrays into the left half of the output
  - Merge the right ones into the right half of the output



# A parallel merge algorithm

```
//merge array A of length n1 and array B of length n2 into array C.  
Merge(A', n1, B', n2, C) {  
    if (A' is empty or B' is empty) base_case;  
    m = n1/2;  
    m2 = binary_search(B', A'[m]);  
    C[m+m2+1] = A'[m];  
    in parallel:  
        merge(A', m, B', m2, C);  
        merge(A'+m+1, n1-m-1, B'+m2+1, n2-m2-1, C+m+m2);  
    return C;  
}
```

