Parallel Programming

Divide and Conquer, Cilk-style bounds

About Me

Rafael Wampfler
PhD Student at Computer Graphics Lab
Research interests

Affective Computing
Machine Learning

Augmented Reality / Telepresence
Medical Applications

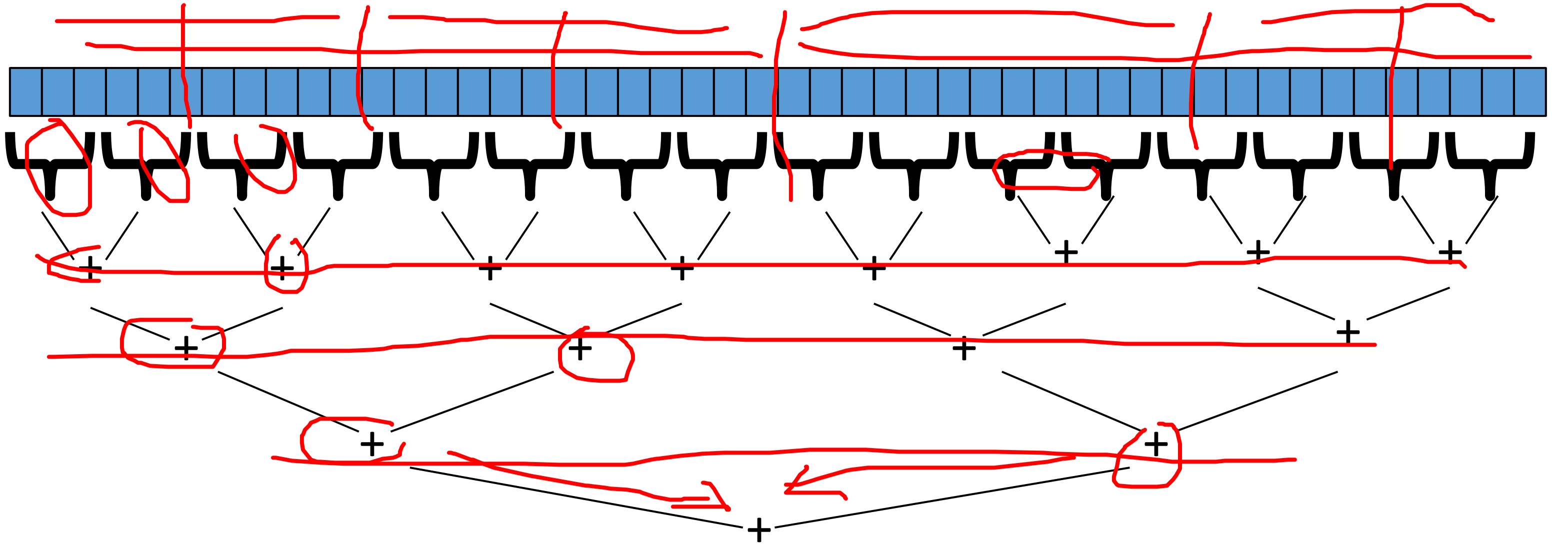
Divide-and-conquer really works – (but it's hard work)

The key is divide-and-conquer parallelizes the result-combining

- *If* you have enough processors, total time is height of the tree: $O(\texttt{log}\ n)$ (optimal, exponentially faster than sequential $O(n)$)
- Often relies on operations being associative (like +)

Will write all our parallel algorithms in this style

- But using <u>special libraries engineered for this style</u>
  - Takes care of scheduling the computation well



Divide-and-conquer – with manual fixes (Pt. I)

```
public void run(){
        int size = h-l;
        if (size < SEQ_CUTOFF)
                for (int i=l; i<h; i++)
                        result += xs[i];
        else {
           int mid = size / 2;
           SumThread t1 = new SumThread(xs, l, l + mid);
           SumThread t2 = new SumThread(xs, l + mid, h);
           t1.start();
           t2.start();
           t1.join();
           t2.join();
           result = t1.result + t2.result;
        }
}
```

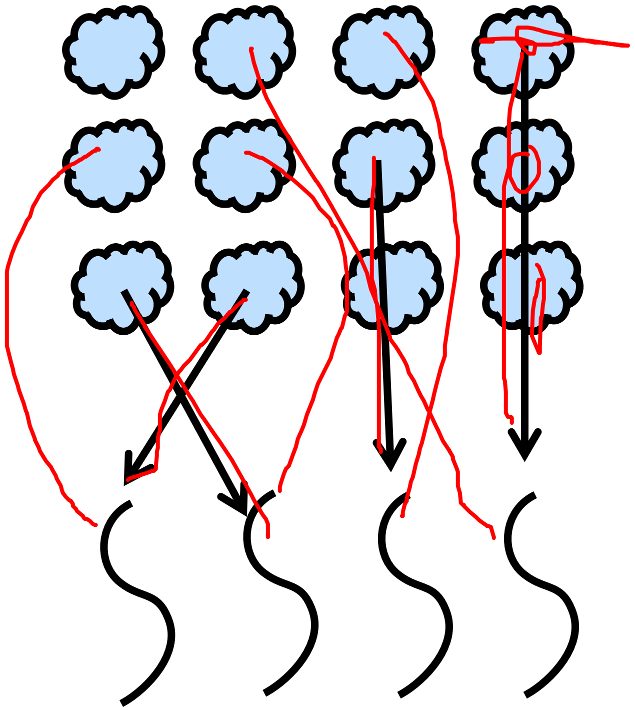
Recap: One thread per task model

Java threads are actually quite heavyweight

Java threads are mapped to OS threads

In general: using one thread per (small tasks) is highly inefficient

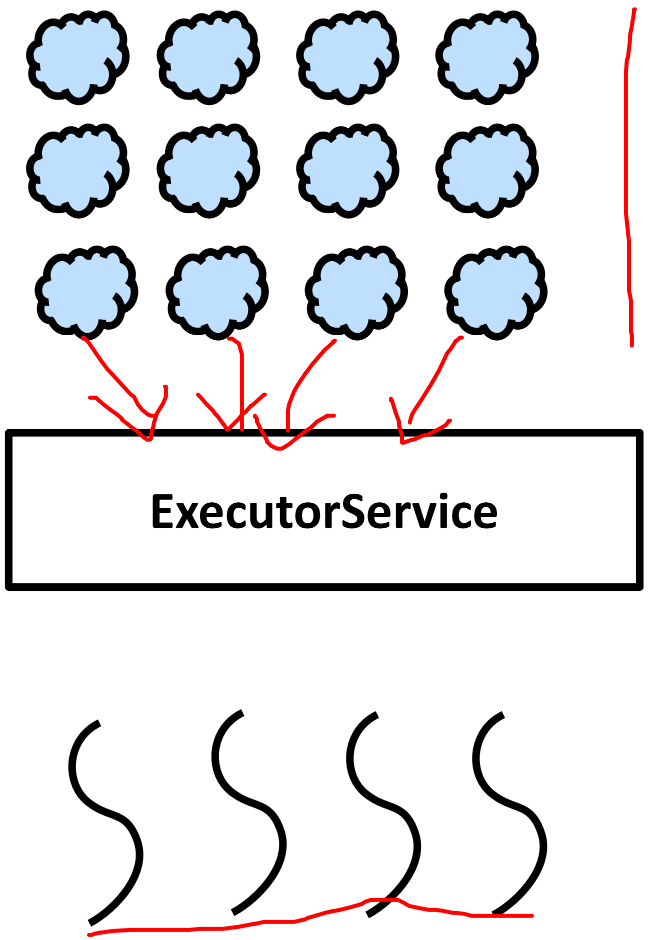
Alternative approach: schedule tasks on threads



(Thread pool)

How many threads would you use?

Java's executor service: managing asynchronous tasks



Tasks

Interface

(Thread pool)

Implementation
e.g.: **ThreadPoolExecutor**

Java's executor service:managing asynchronous tasks

User submits tasks

ExecutorService

gets back a **Future**

```
.submit(Callable<T> task) → Future<T>
.submit(Runnable task)    → Future<?>
```

Note: Callable vs Runnable

ExecutorService can handle “Runnable” or “Callable” tasks:

Interface Runnable:

→ void run() ⟶ Does not return result

Interface Callable<T>:

→ T call() ⟶ Returns result

Using executor service: Hello World (task)

```
static class HelloTask implements Runnable {

    String msg;

    public HelloTask(String msg) {
        this.msg = msg;
    }

    public void run() {
        long id = Thread.currentThread().getId();
        System.out.println(msg + " from thread:" + id);
    }
}
```

Using executor service: Hello World (creating executor, submitting)

```java
int ntasks = 1000;
ExecutorService exs = Executors.newFixedThreadPool(4);

for (int i=0; i<ntasks; i++) {
  HelloTask t = new HelloTask("Hello from task " + i);
  exs.submit(t);
}

exs.shutdown(); // initiate shutdown, does not wait, but can’t submit more tasks
```

Using executor service: Hello World (output)

```
…
Hello from task 803 from thread:8
Hello from task 802 from thread:10
Hello from task 807 from thread:8
Hello from task 806 from thread:9
Hello from task 805 from thread:11
Hello from task 810 from thread:9
Hello from task 809 from thread:8
Hello from task 808 from thread:10
Hello from task 813 from thread:8
Hello from task 812 from thread:9
Hello from task 811 from thread:11
...
```

Recursive Sum with ExecutorService

```
public Integer call() throws Exception {
  int size = h - l;
  if (size == 1)
    return xs[l];

  int mid = size / 2;
  sumRecCall c1 = new sumRecCall(ex, xs, l, l + mid);
  sumRecCall c2 = new sumRecCall(ex, xs, l + mid, h);

  Future<Integer> f1 = ex.submit(c1);
  Future<Integer> f2 = ex.submit(c2);

  return f1.get() + f2.get();
}
```

Simple! – But does this work?

If you execute the code, you will observe that it never returns (i.e., the computation is not completed)

Why does this happen?

1

```
sum(0,100):
  t1 = spawn sum(0,50)
  t2 = spawn sum(50,100)
  t1.wait(); t2.wait()
```

2

```
sum(0,50):
  t1 = spawn sum(0,25)
  t2 = spawn sum(25,50)
  t1.wait(); t2.wait()
```

3

```
sum(50,100):
  t1 = spawn sum(50,75)
  t2 = spawn sum(75,100)
  t1.wait(); t2.wait()
```

4

```
sum(0,25):
  t1 = spawn sum(0,12)
  t2 = spawn sum(12,25)
  t1.wait(); t2.wait()
```

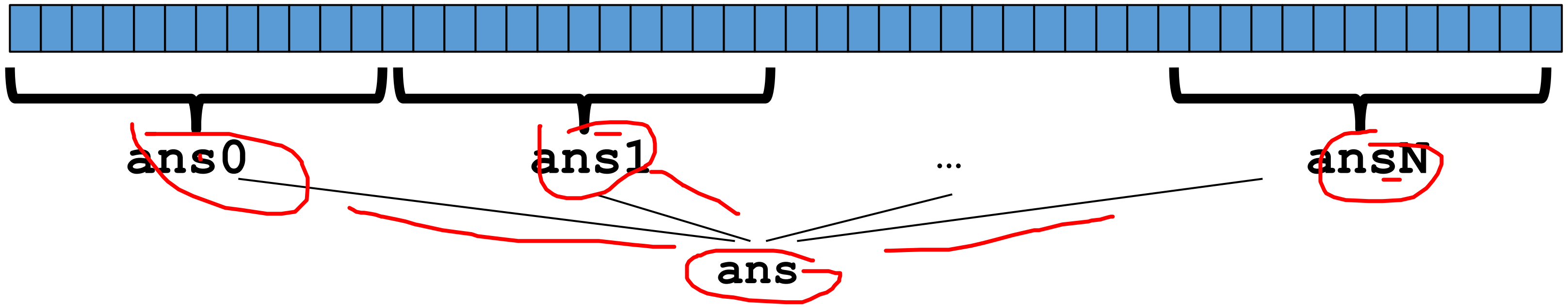
tasks will end up waiting

eventually we will run out of threads

Adding Numbers ExecutorService: another approach

Problem with the divide and conquer approach is that tasks create other tasks and work partitioning (splitting up work) is part of the task.

A possible approach is to decouple work partitioning from solving the problem. That is we split the array into chunks (how many?) and create a task per chunk. Then, we submit tasks into ExecutorService and combine results (e.g., sum). It can be tricky to do the initial partitioning of work and final summing in parallel.

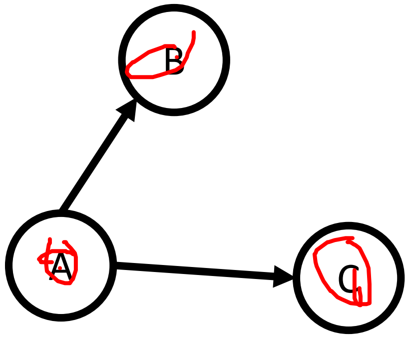


Task Parallel Programming [Cilk-style]

Tasks:

- execute code
- spawn other tasks
- wait for results from other tasks

A graph is formed based on spawning tasks



The edges mean that Task B was created by Task A and that Task C was created by Task A

fib() Function

$$fib(n) = \begin{cases} n & n < 2 \\ fib(n-1) + fib(n-2) & n >= 2 \end{cases}$$

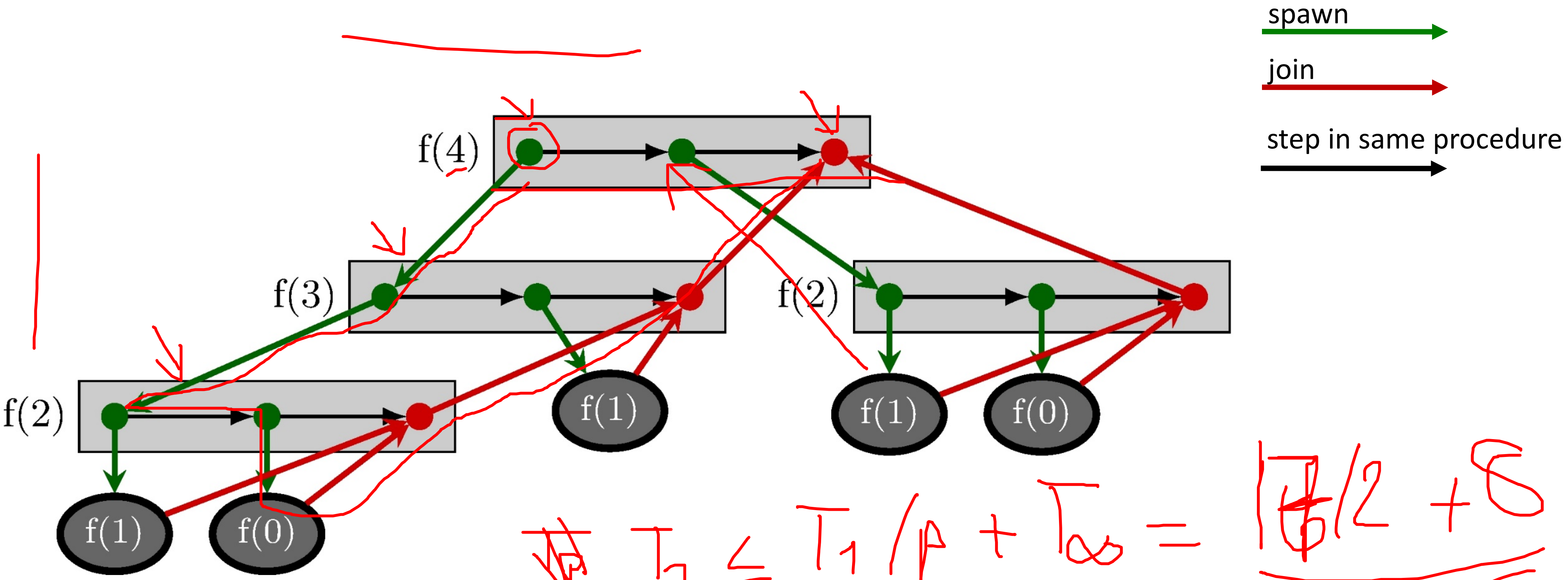
Sequential Version

```
public class Fibonacci {
 public static long fib(int n){
        if (n < 2)
            return n;
        long x1 = fib(n-1);
        long x2 = fib(n-2);
        return x1 + x2;
    }
}
```

Parallel Version

```
public class Fibonacci {
 public static long fib(int n) {
        if (n < 2)
            return n;
        spawn task for fib(n-1);
        spawn task for fib(n-2);
        wait for tasks to complete
        return addition of task results
}}}
```

`fib(4)` task graph



$T_2 \le T_1/p + T_\infty = 18/2 + 8$

The task graph is a directed acyclic graph (DAG)

Task parallelism discussion

- Tasks can execute in parallel
  - but they don't have to
  - assignment of tasks to CPUs/cores is up to the scheduler
- Task graph is **dynamic**
  - unfolds as execution proceeds
- Intuition: wide task graph → more parallelism

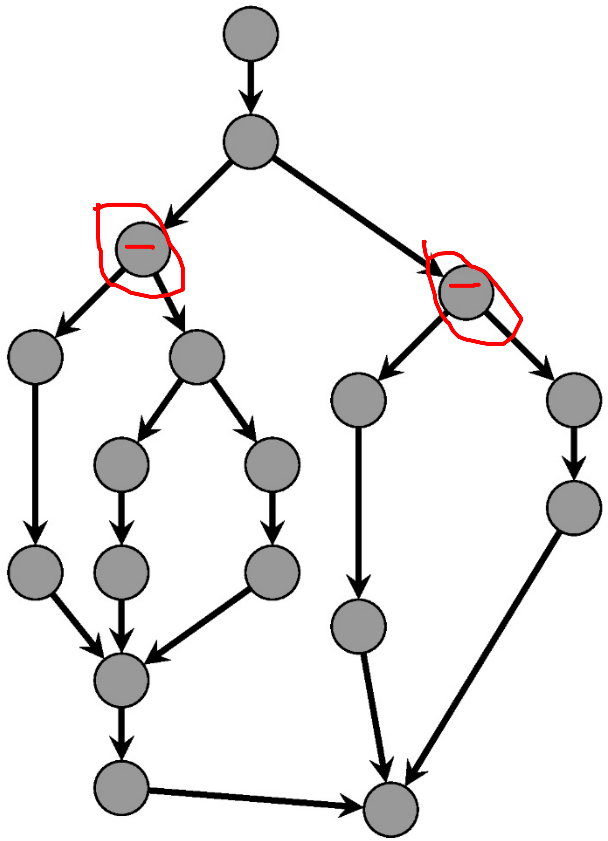
Task parallelism: performance model

- Task graph: tasks become available as computation progresses
- We can execute the graph on **p** processors
  Scheduler assign tasks to processors
- $\mathbf{T_p}$: execution time on **p** processors

T_1

T_2

T_3



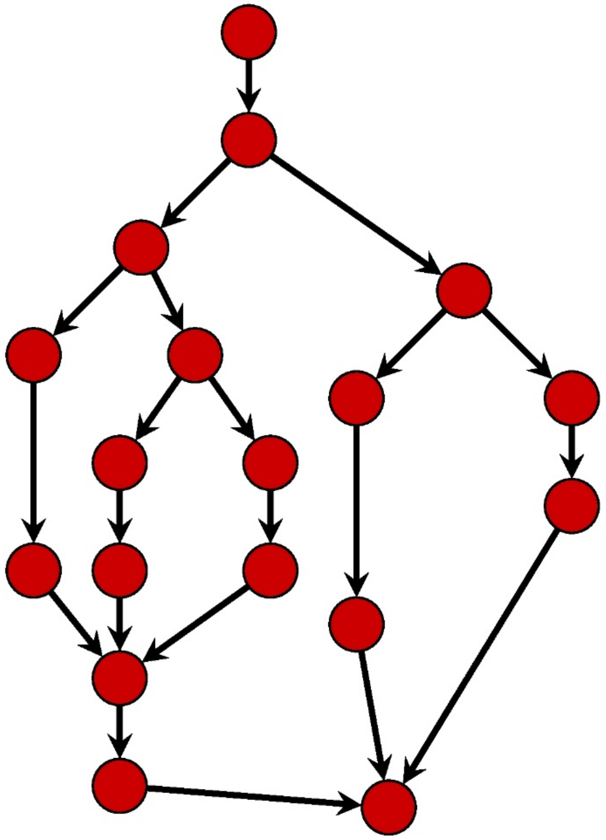
Task parallelism: performance model [some reminders]

- **T_p**: execution time on **p** processors
- **T_1: work** (total amount of work)
  - the sum of the time cost of all nodes in graph
  - as if we executed graph sequentially (p=1)
- **T_1 / T_p → speedup**

$T_1 = 10$

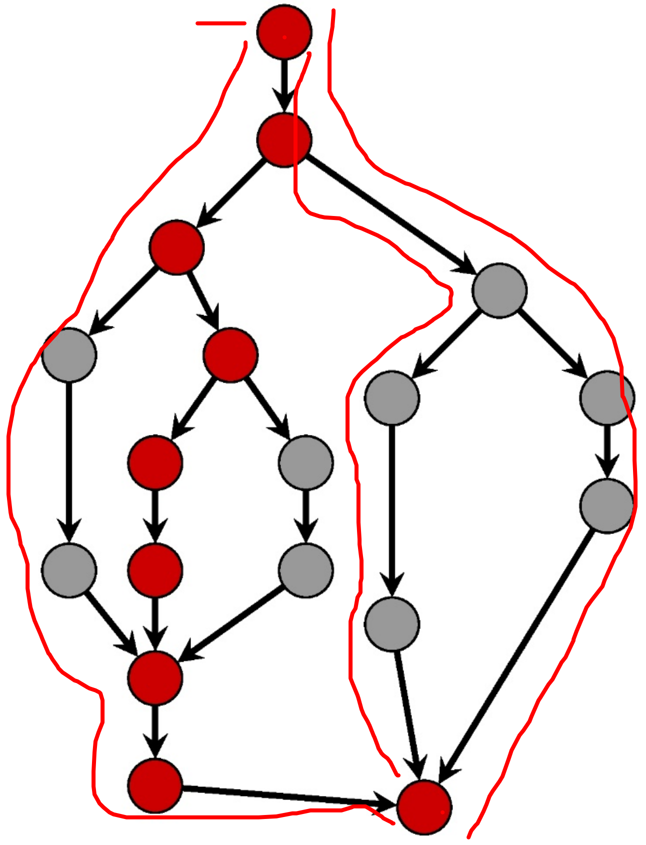
$T_2 = 5$

$T_1 / T_2 = 2$



Task parallelism: performance model (Bounds)

- **T_∞: span**, **critical path**
  - Time it takes on infinite processors
  - longest path from root to sink
- **$T_1 / T_\infty \rightarrow$ parallelism**
  - "wider" is better
- Lower Bounds:
  - $\mathbf{T_p \geq T_1 / P}$
  - $\mathbf{T_p \geq T_\infty}$



On this graph, T_∞ is 9

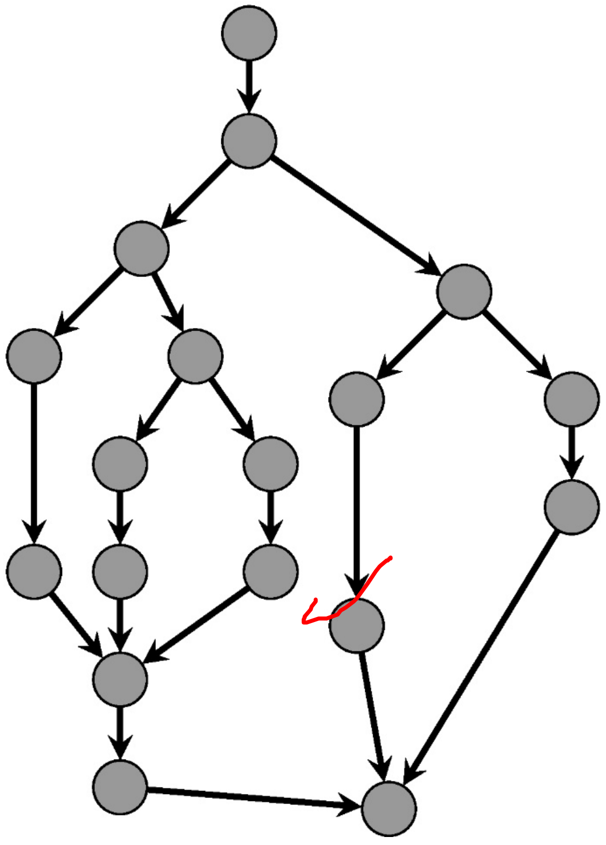
Scheduling of task graphs

Scheduler is an algorithm for assigning **tasks** to **processors**

Note that:

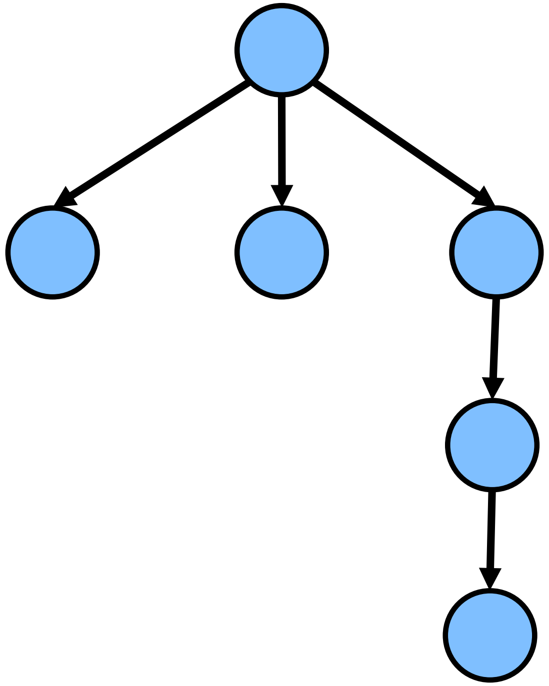
$\mathbf{T_p}$ depends on scheduler

$\mathbf{T_1 / P}$ and $\mathbf{T_\infty}$ are fixed

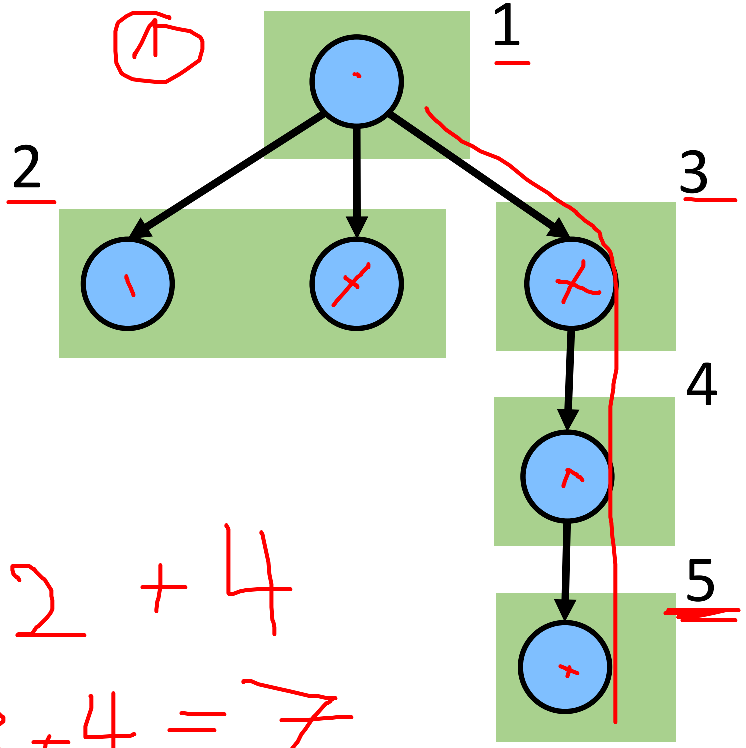


What is T_2 for this graph?

That is, we have 2 processors.

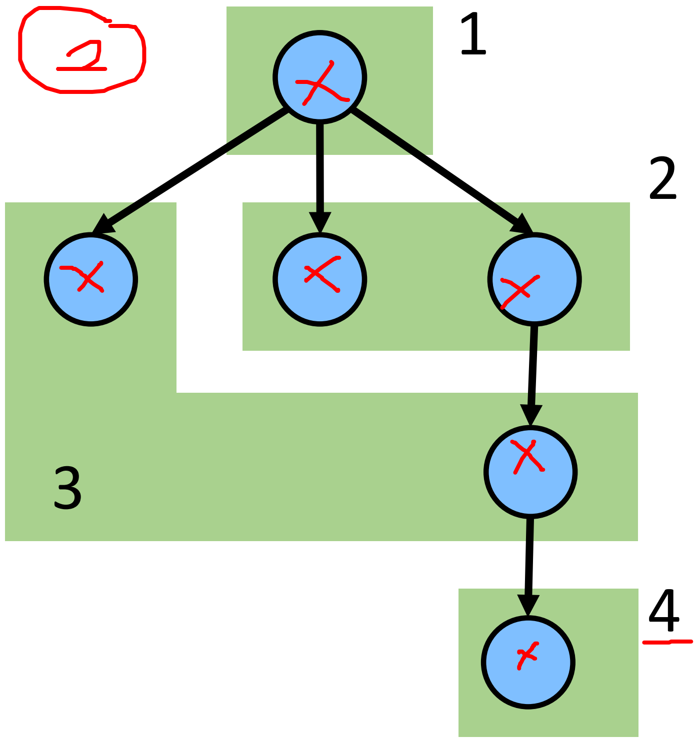


$$T_2 \leq 6/2 + 4$$
$$\leq 3 + 4 = 7$$



T_2 will be 5 with this scheduling

(we have 5 time steps)



T_2 will be 4 with this scheduling

(we have 4 time steps)

a bound on how fast you can get on p processors
with a greedy scheduler: $T_p \leq T_1 / P + T_\infty$

Work stealing scheduler

First used in MIT's Cilk, now a standard method

Provably: $T_p = T_1 / P + O(T_\infty)$

Empirically: $T_p \approx T_1 / P + T_\infty$

Guideline for parallel programs => "Scheduling Multithreaded Computations by Work Stealing", Blumfoe & Leiserson, MIT

Summary

Divide and conquer for parallel programming

Cilk-style task graphs, scheduling and bounds