Parallel Programming

Divide and Conquer, Cilk-style bounds

Lets look at a code example: sum the elements of a list

Sequential Version

The first step of writing a parallel program is writing a sequential version:

- Helps validate our eventual parallel program is correct
  - by comparing results with the simpler, sequential version
- Evaluate the performance of our parallel program
  - we write parallel programs to improve performance!

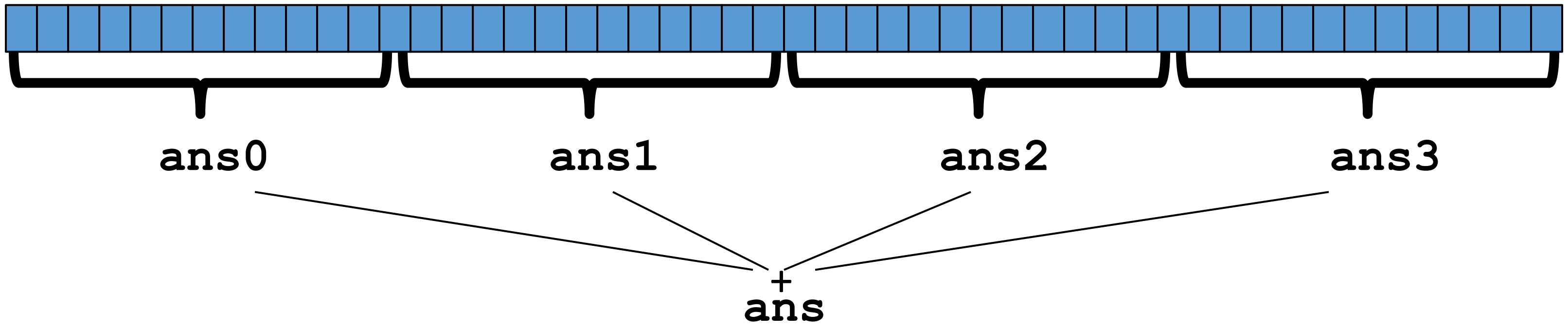
Adding Numbers - Sequentially

```java
public static int sum(int[] input){
    int sum = 0;
    for(int i=0; i<input.length; i++){
        sum += input[i];
    }
    return sum;
}
```

Parallelism idea

Idea: Have 4 threads simultaneously sum 1/4 of the array

- Warning: This is an inferior first approach



- Create 4 *thread objects*, each given a portion of the work
- Call **`start()`** on each thread object to actually *run* it in parallel
- *Wait* for threads to finish using **`join()`**
- Add together their 4 answers for the *final result*

First attempt, part 1

```
class SumThread extends java.lang.Thread {

  int lo; // arguments
  int hi;
  int[] arr;

  int ans = 0; // result

  SumThread(int[] a, int l, int h) {
    lo=l; hi=h; arr=a;
  }


  public void run() { //override must have this type
    for(int i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

Because we must override a no-arguments/no-result `run`, we use fields to communicate across threads

First attempt, continued (wrong)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}


int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++) // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);

  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

Second attempt (still wrong)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}


int sum(int[] arr){// can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start(); // start actually runs the thread in parallel
  }
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

Third attempt (correct in spirit)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}

int sum(int[] arr){// can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();
  }
  for(int i=0; i < 4; i++) { // combine results
    ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
  }
  return ans;
}
```

Discussion

The `Thread` class defines various methods you could not implement on your own

- For example: `start`, which calls `run` in a new thread

The `join` method is valuable for coordinating this kind of computation

- Caller blocks until/unless the receiver is done executing (meaning the call to `run` finishes)
- Else we would have a race condition on `ts[i].ans`

This style of parallel programming is called fork/join

Java detail: code has 1 compile error because `join` may throw `java.lang.InterruptedException`

- In basic parallel code, should be fine to catch-and-exit

Shared memory?

Fork-join programs (thankfully) do not require much focus on sharing memory among threads

But in languages like Java, there is memory being shared.

In our example:

- **`lo`**, **`hi`**, **`arr`** fields written by “main” thread, read by helper thread
- **`ans`** field written by helper thread, read by “main” thread

When using shared memory, you must avoid race conditions (we will see a more formal definition of data races, later)

Issues with this approach (and some workarounds)

Several reasons why this is a poor parallel algorithm

Reason 1: want code to be reusable and efficient across platforms

- "Forward-portable" as core count grows
- So at the *very* least, **parameterize by the number of threads**

```
int sum(int[] arr, int numTs){
  int ans = 0;
  SumThread[] ts = new SumThread[numTs];
  for(int i=0; i < numTs; i++){
   ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                             ((i+1)*arr.length)/numTs);
   ts[i].start();
  }
  for(int i=0; i < numTs; i++) {
    ts[i].join();
    ans += ts[i].ans;
  }
  return ans;
}
```

Issues with this approach (and some workarounds)

Reason 2: want to use (only) processors "available to you *now*"

- Not used by other programs or threads in your program
  - Maybe caller is also using parallelism
  - Available cores can change even while your threads run

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numTs){
  …
}
```

Issues with this approach (and some workarounds)

Reason 3: Though unlikely for **`sum`**, in general subproblems may take significantly different amounts of time

Example: Apply method **`f`** to every array element, but maybe **`f`** is much slower for some data items, e.g.: is a large integer prime?

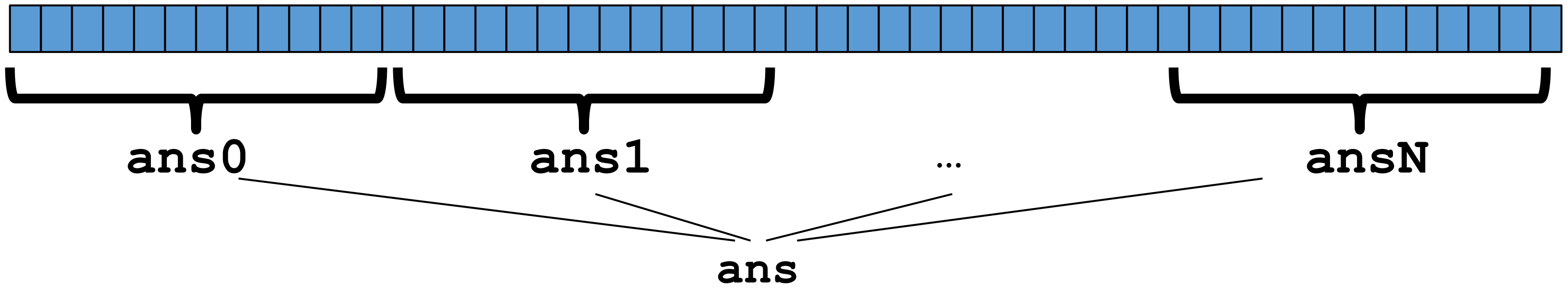
If we create 4 threads and all slow data is processed by 1 of them, we won't get nearly a 4x speedup

- Example of a load imbalance

A Better Approach

The counterintuitive (?) solution to all these problems is to use lots of threads, far more than the number of processors

- But this will require changing our algorithm
- And for constant-factor reasons, abandoning Java's threads

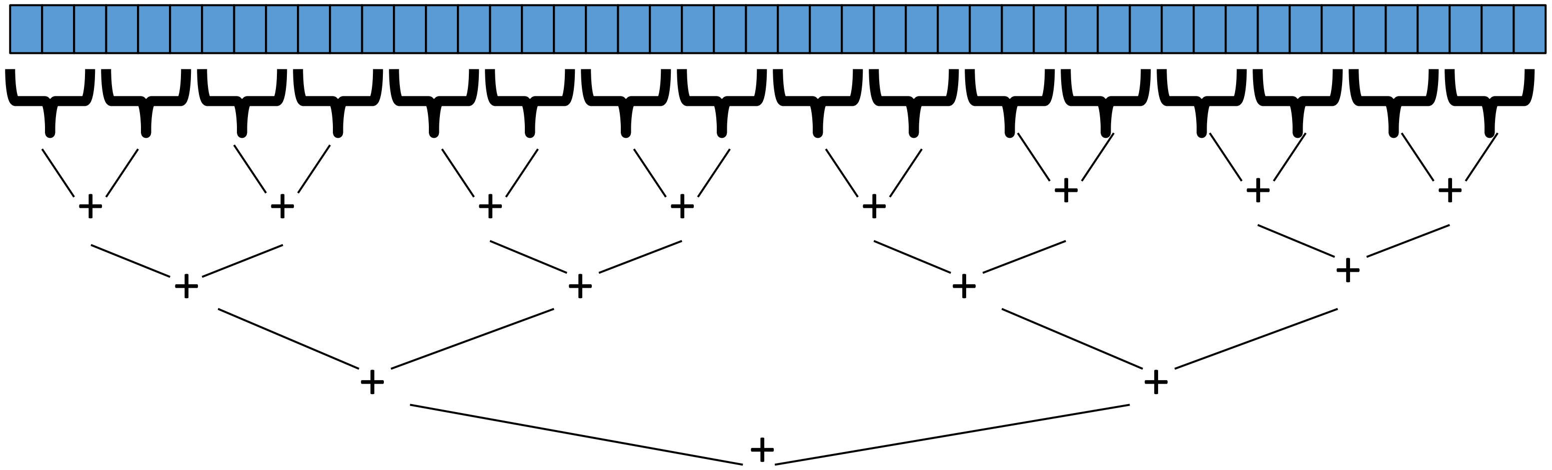


1. Forward-portable: Lots of helpers each doing a small piece
2. Processors available: Hand out "work chunks" as you go
3. Load imbalance: No problem if slow thread scheduled early enough
   - Variation probably small anyway if pieces of work are small

Divide and Conquer to the Rescue!

This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls



Divide and Conquer

Fundamental pattern in parallel programming, also called **recursive splitting**

```
Divide and Conquer:
  if cannot divide:
    return unitary solution (stop recursion)
  divide problem in two
  solve first (recursively)
  solve second (recursively)
  combine solutions
  return result
```

Sequential Version: Recursive Sum

```
public static int do_sum_rec(int[] xs, int l, int h) {
     int size = h-l;
     if (size == 1)  /*check for termination criteria*/
          return xs[l];

     /* split array in half and call self recursively*/
     int mid  = size / 2;
     int sum1 = do_sum_rec(xs, l, l + mid);
     int sum2 = do_sum_rec(xs, l + mid, h);
     return sum1 + sum2;
}
```

Parallel Recursive Sum (with Threads)

```
public class SumThread extends Thread {
int[] xs;
int h, l;
int result;

public SumThread(int[] xs, int l, int h){
        super();
        this.xs = xs;
        this.h = h;
        this.l =l;
}

public void run(){
        /*Do computation and write to result*/
        return;
}
```

Parallel Recursive Sum (with Threads)

```
public void run(){
    int size = h-l;
    if (size == 1) {
        result = xs[l];
        return;
    }
    int mid = size / 2;
    SumThread t1 = new SumThread(xs, l, l + mid);
    SumThread t2 = new SumThread(xs, l + mid, h);

    t1.start();
    t1.join();

    t2.start();
    t2.join();

    result = t1.result + t2.result;
    return;
}
```

Is this OK?

Parallel Recursive Sum (with Threads)

```
public void run(){
    int size = l-h;
    if (size == 1) {
          result = xs[l];
          return;
    }
    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;
    return;
}
```

Remark: This doesn't compile because join() can throw exceptions. In reality we need a try-catch block here.

Result

Java.lang.OutOfMemoryError: unable to create new native thread

One thread per parallel task model

Java threads are actually quite heavyweight

Java threads are mapped to OS threads (in the Oracle and most real-world implementations)

In general: <u>using one thread per (small tasks) is highly inefficient</u>

Divide-and-Conquer works – (really, we'll get there)

In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup

In practice, creating all those threads and communicating swamps the savings, so:

- Use a *sequential cutoff*, typically around 500-1000
  - Eliminates *almost all* the recursive thread creation (bottom levels of tree)
- Do not create two recursive threads; create one and do the other "yourself"
  - Cuts the number of threads created by another 2x

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;
        }
}
```

Half the threads

```
// wasteful: don't
SumThread t1 = …
SumThread t2 = …
t1.start();
t2.start();
t1.join();
t2.join();
result=t1.result+t2.result;
```

```
// better: do
// order of next 4 lines
// essential – why?
t1.start();
t2.run();
t1.join();
result=t1.result+t2.result;
```

If a *language* had built-in support for fork-join parallelism, we would expect this hand-optimization to be unnecessary

But the *library* we are using expects you to do it yourself (and the difference is surprisingly substantial)

Again, no difference in theory

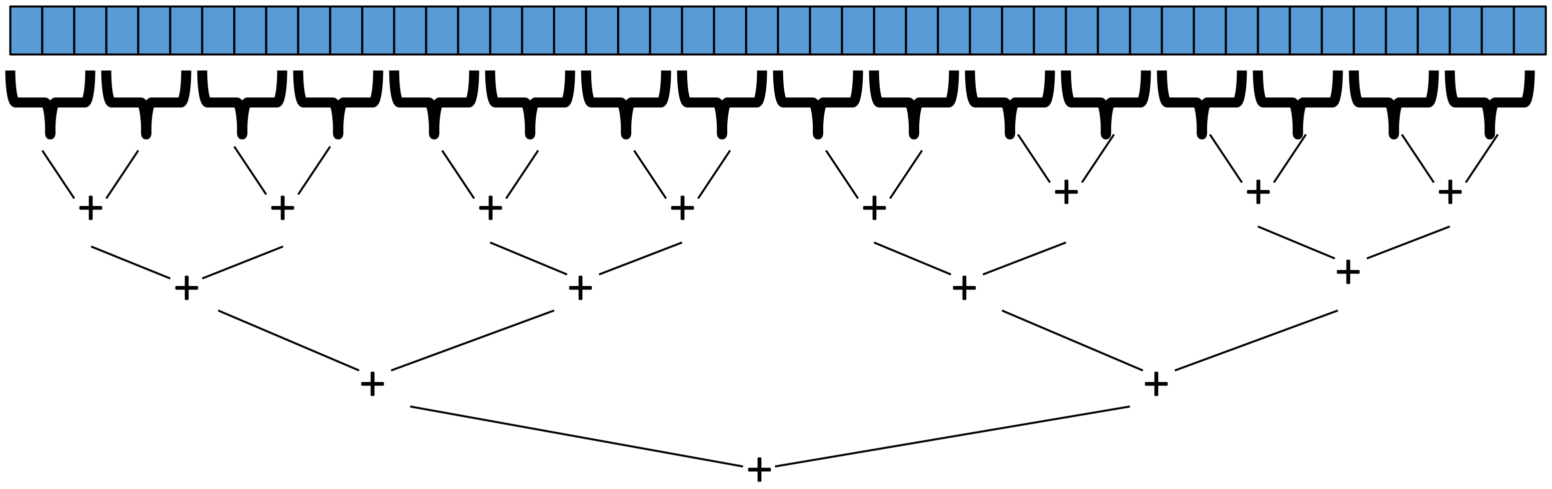
Divide-and-conquer really works – (but it's hard work)

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

- *If* you have enough processors, total time is height of the tree: $O(\mathtt{log}\ n)$ (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



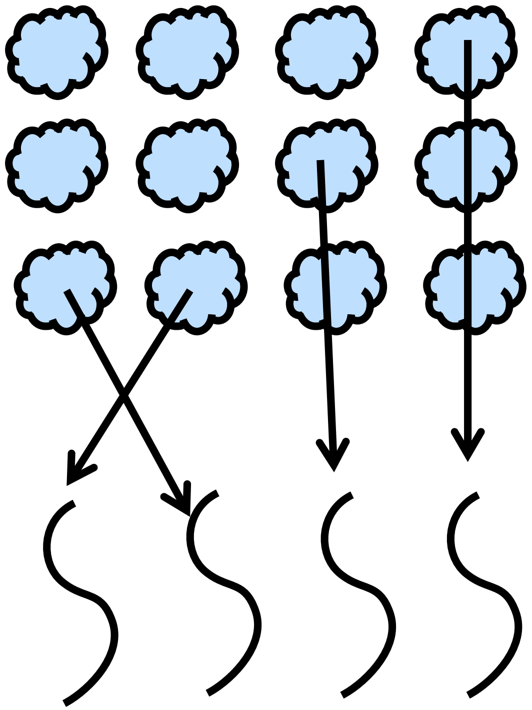
Recap: One thread per task model

Java threads are actually quite heavyweight

Java threads are mapped to OS threads

In general: <u>using one thread per (small tasks) is highly inefficient</u>

Alternative approach: schedule tasks on threads



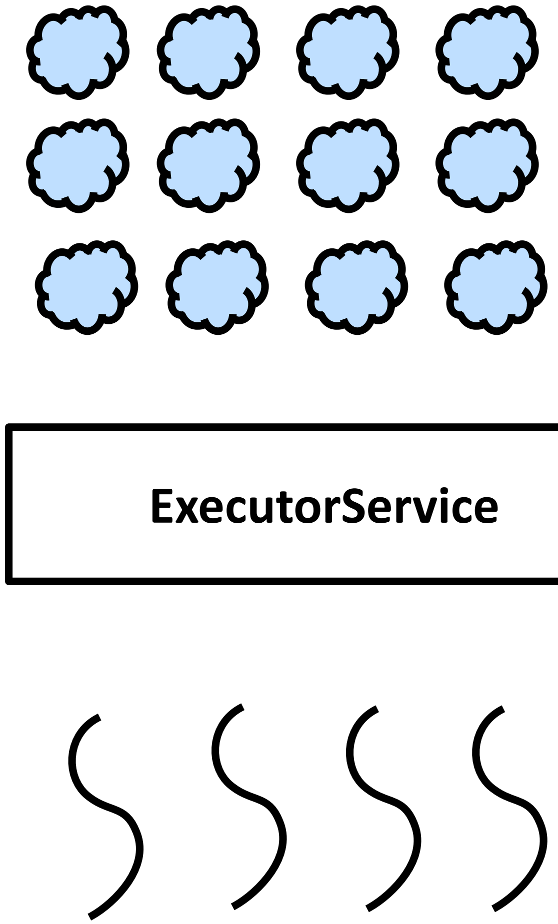
Tasks

(Thread pool)

Threads

How many threads would you use?

Java's executor service: managing asynchronous tasks



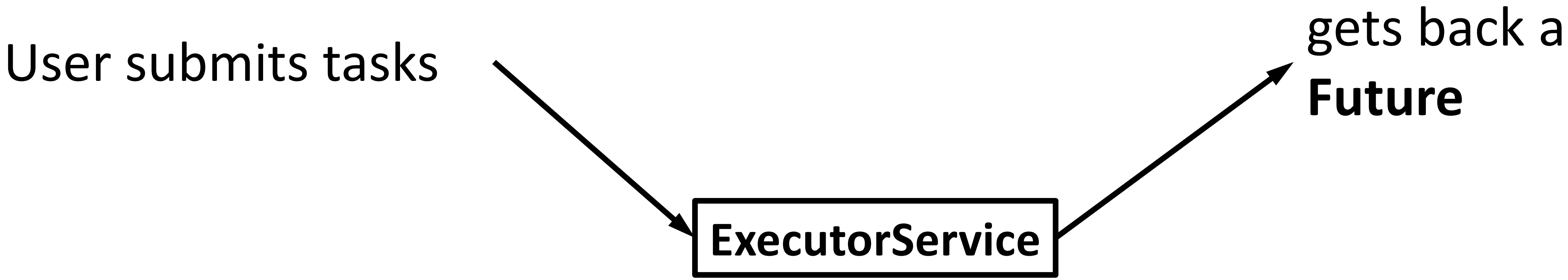
Tasks

Interface

(Thread pool)

Implementation
e.g.: **ThreadPoolExecutor**

Java's executor service:managing asynchronous tasks




```
.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)

```java
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)

```
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?

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

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

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

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

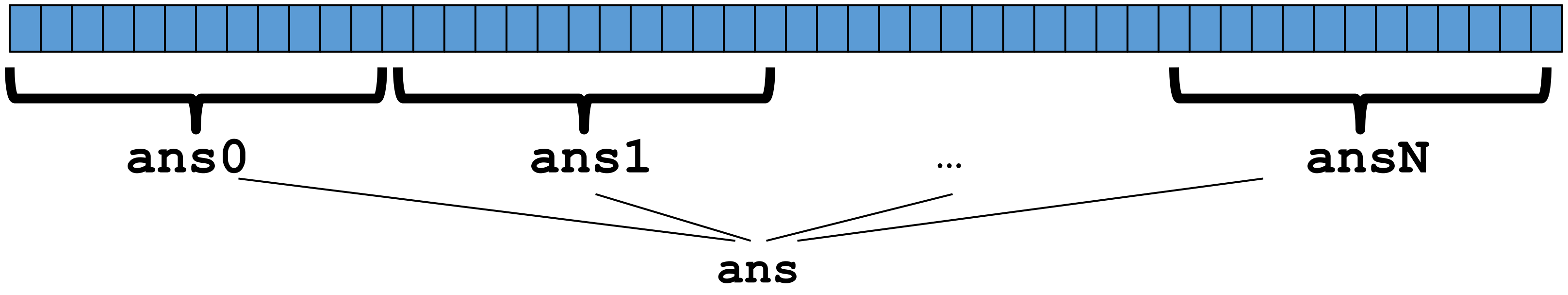
tasks will end up waiting

eventually we will <u>run out of threads</u>

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.

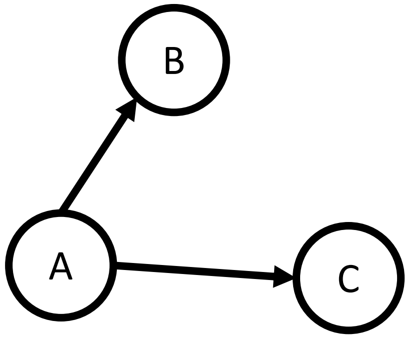


Task Parallel Programming [Cilk-style]

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

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

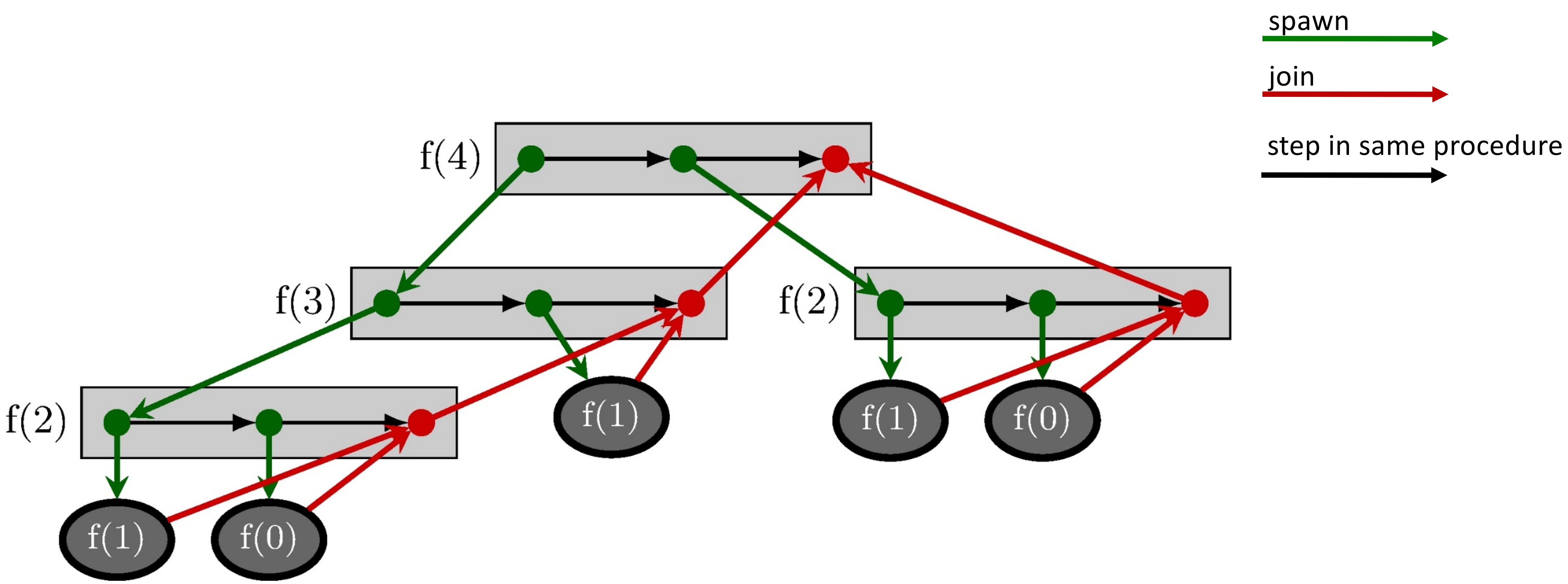
Sequential Version

```
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



The task graph is a directed acyclic graph (DAG)

Task parallelism discussion

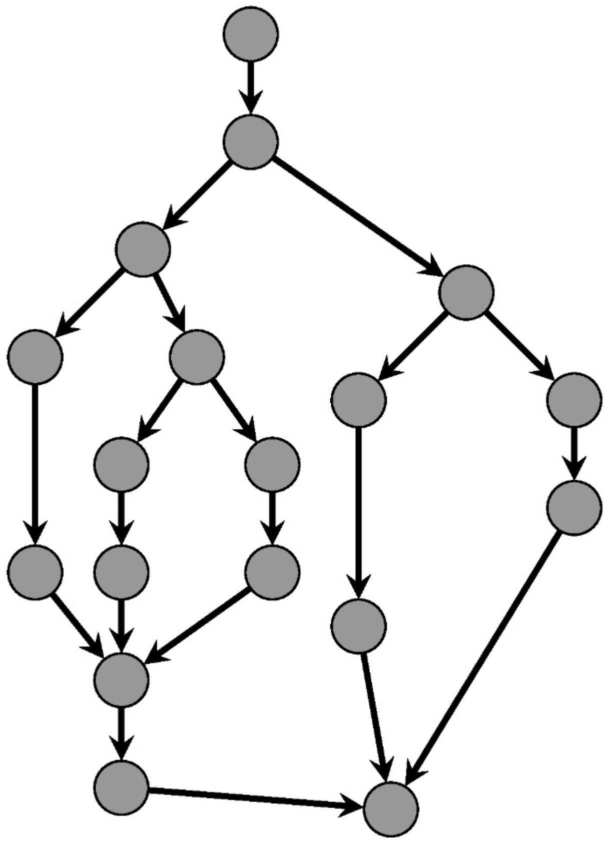
- Tasks can execute in parallel
  - but they don't have to
  - assignment of tasks to CPUs/cores is up to the scheduler

- 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

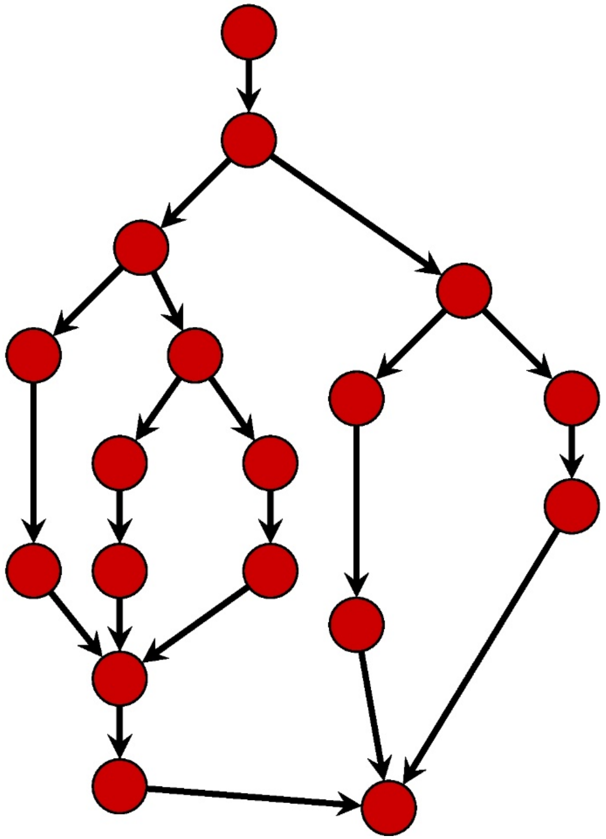


Task parallelism: performance model [some reminders]

- $\mathbf{T_p}$: execution time on **p** processors

- $\mathbf{T_1}$**: work** (total amount of work)
  - the sum of the time cost of all nodes in graph
  - as if we executed graph sequentially (p=1)

- $\mathbf{T_1}$ / $\mathbf{T_p}$ → **speedup**

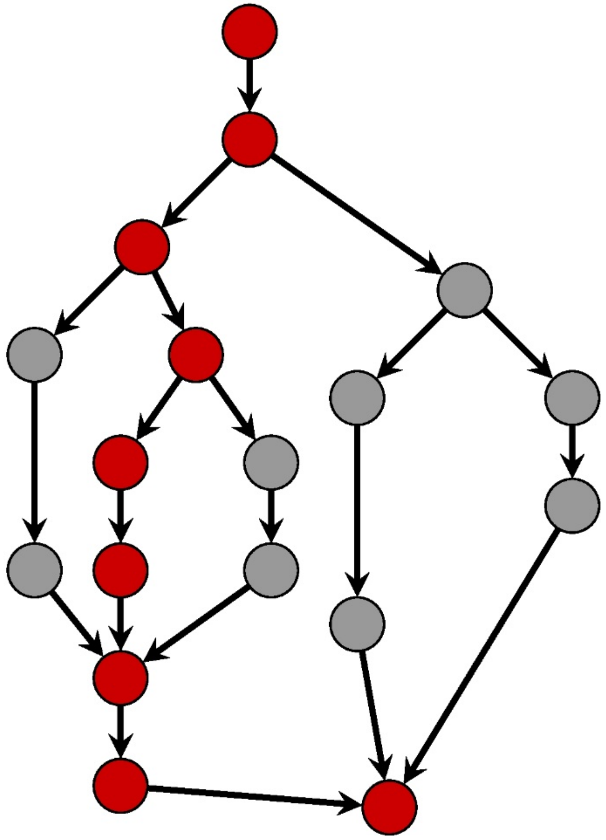


Task parallelism: performance model (Bounds)

- **T_∞: span**, **critical path**
  - Time it takes on infinite processors
  - longest path from root to sink

- **T_1 / T_∞ → parallelism**
  - "wider" is better

- Lower Bounds:
  - **$T_p \geq T_1 / P$**
  - **$T_p \geq T_\infty$**



On this graph, T_∞ is 8

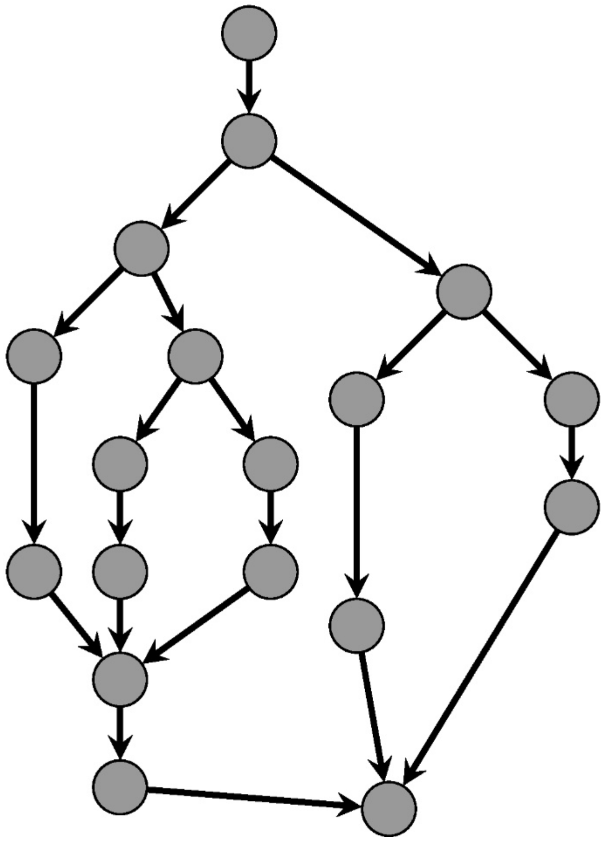
Scheduling of task graphs

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

Note that:

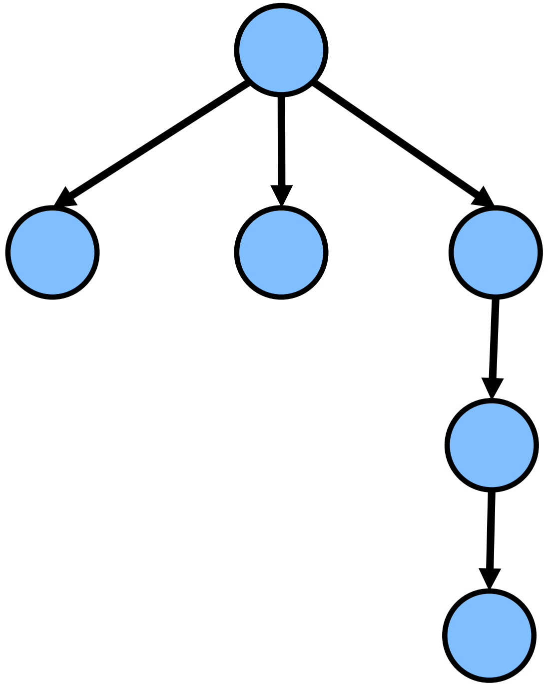
$\mathbf{T_p}$ depends on scheduler

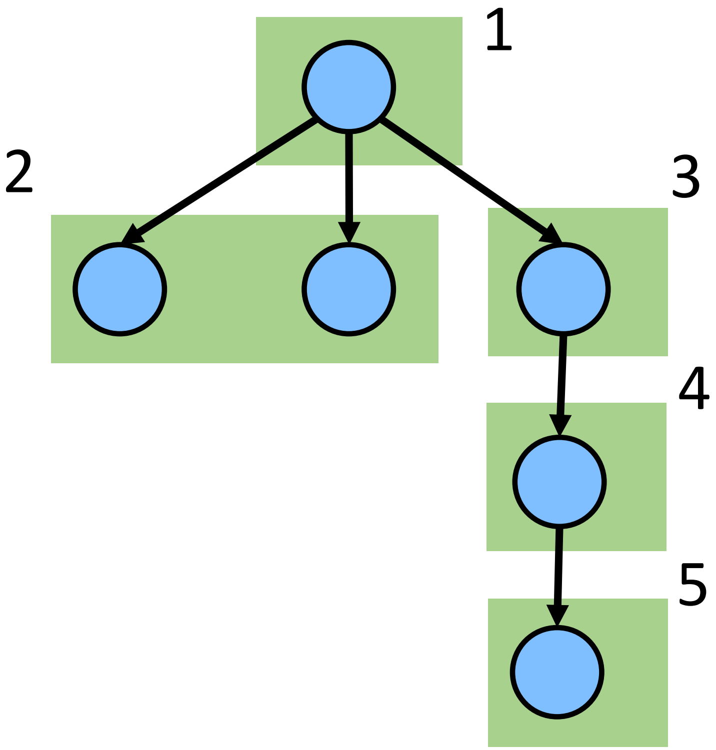
$\mathbf{T_1 / P}$ and $\mathbf{T_\infty}$ are fixed



What is T_2 for this graph?

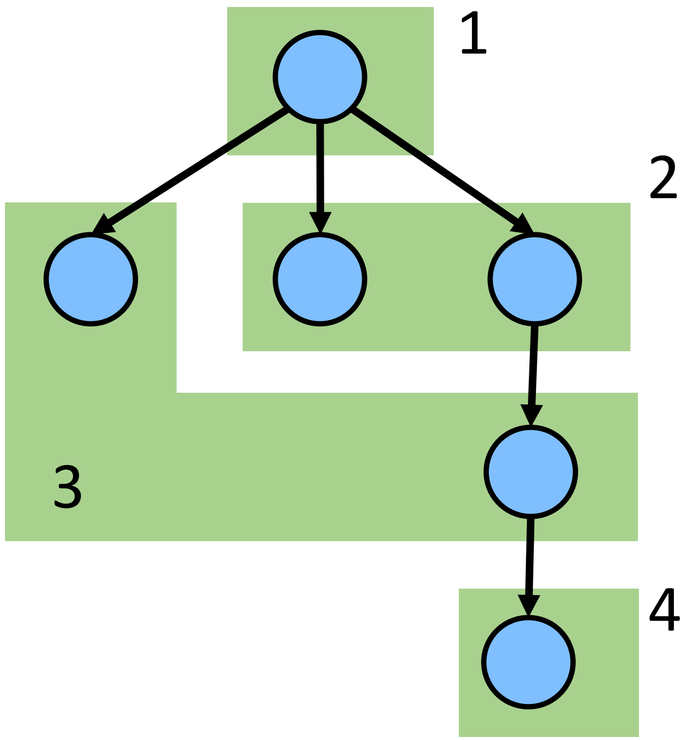
That is, we have 2 processors.





T_2 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