# Parallel Programming
# Exercise Session 6

Spring 2020

# Feedback: Exercise 5

# Recall: Amdahl's vs Gustafson's Law

The key is goal to:

➔ Understand the main difference and implications (i.e., when to use which formula)

➔ Know how to derive formula based on the understanding, Not because you memorized them for the exam

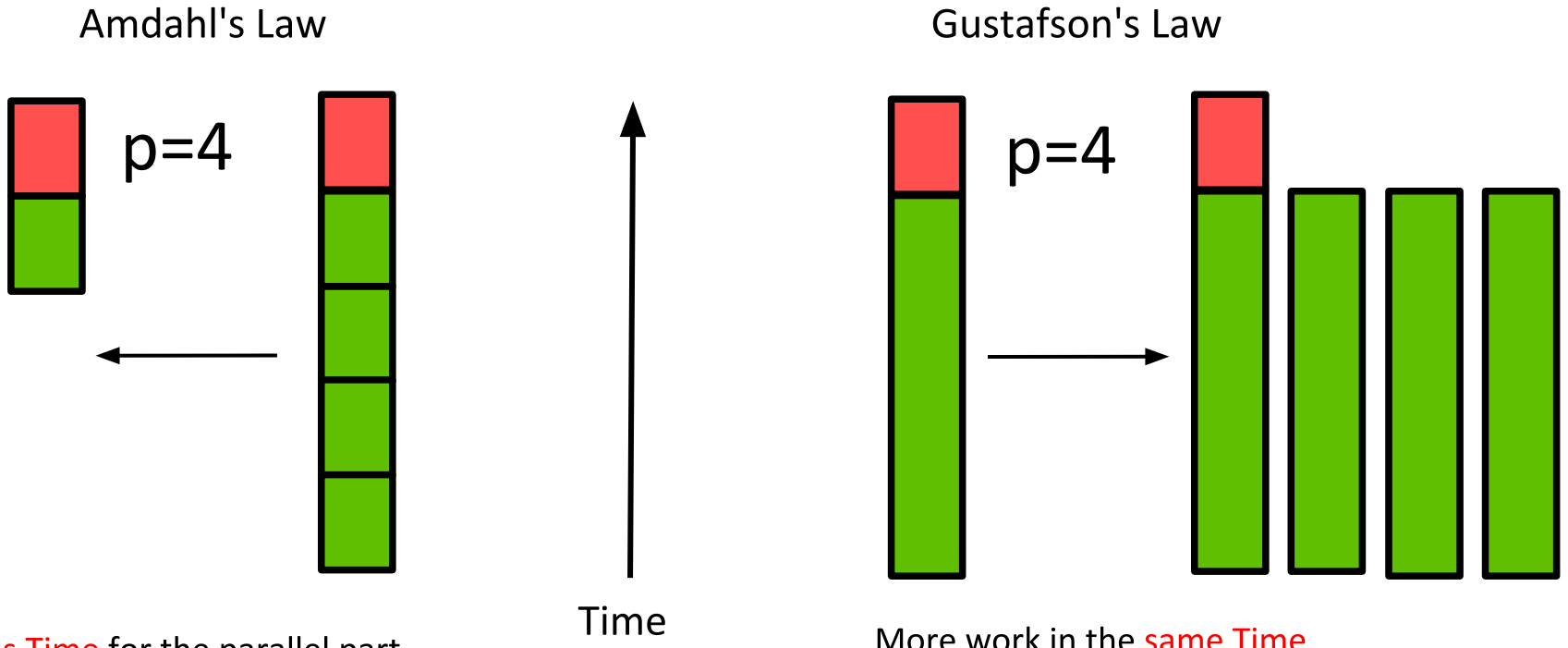
# Recall: Amdahl's vs Gustafson's Law

The key is goal to:

➔ **Understand the main difference and implications (i.e., when to use which formula)**

➔ Know how to derive formula based on the understanding, Not because you memorized them for the exam

# Recall: Amdahl's vs Gustafson's Law



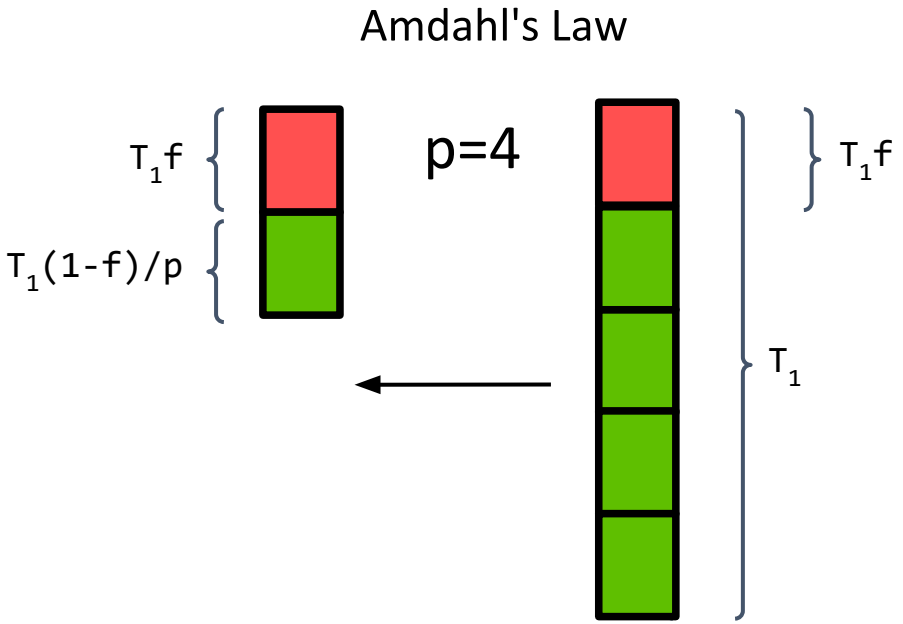
# Recall: Amdahl's vs Gustafson's Law

The key is goal to:

➔ Understand the main difference and implications (i.e., when to use which formula)

➔ **Know how to derive formula based on the understanding, Not because you memorized them for the exam**

# Amdahl's Law Derivation




Less Time for the parallel part

$T_1$ - sequential time

$f$ - sequential fraction

$T_p$ - parallel time on p processors

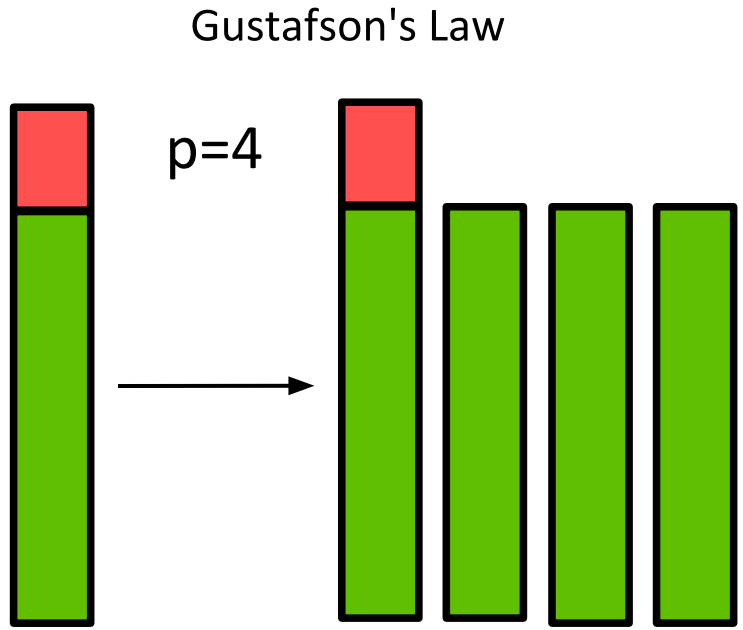
$T_p = T_1 f + T_1(1-f)/p$

$S_p$ - speedup

$S_p = T_1/T_p$

$S_p = 1/(f + (1-f)/p)$

# Gustafson's Law Derivation




More work in the same Time

$T$ - sequential time of original work

$T_1$ - sequential time with work*p

$f$ - sequential fraction

$T_1 = ?$

$T_p$ - parallel time on p processors

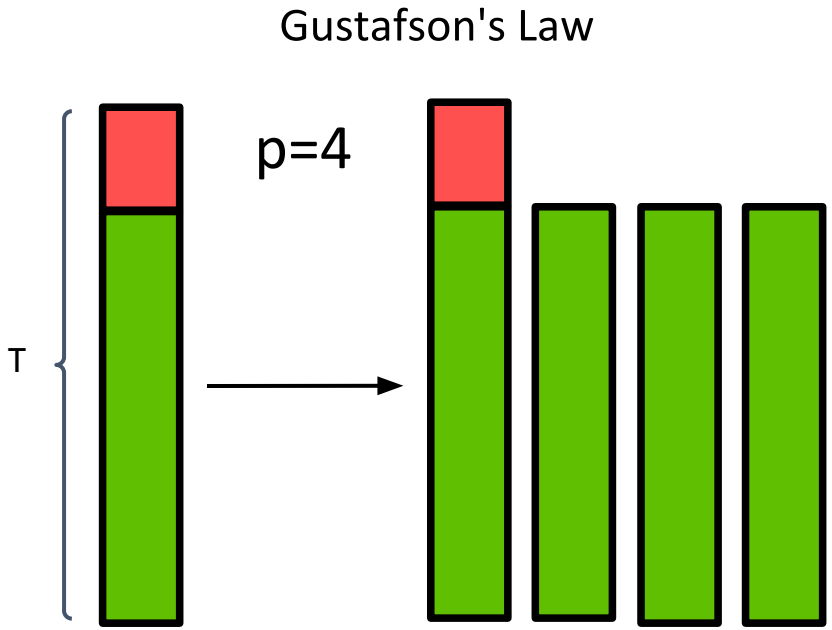
$T_p = ?$

$S_p$ - speedup

$S_p = T_1/T_p$

$S_p = ?$

# Gustafson's Law Derivation




More work in the same Time

$T$ - sequential time of original work

$T_1$ - sequential time with work*p

$f$ - sequential fraction

$T_1 = Tf + T(1-f)p$

$T_p$ - parallel time on p processors

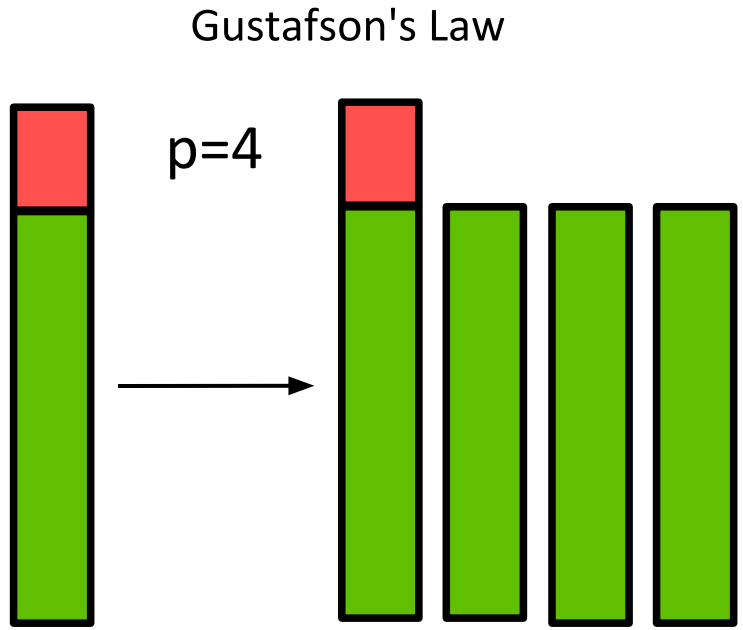
$T_p$ = **?**

$S_p$ - speedup

$S_p = T_1/T_p$

$S_p$ = **?**

# Gustafson's Law Derivation




More work in the same Time

$T$ - sequential time of original work

$T_1$ - sequential time with work*p

$f$ - sequential fraction

$$T_1 = Tf + T(1-f)p$$

$T_p$ - parallel time on p processors

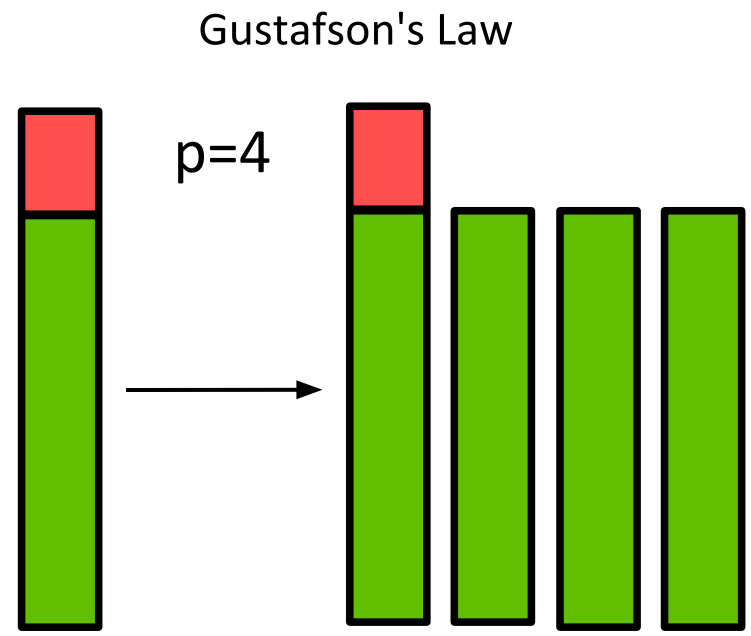
$$T_p = Tf + T(1-f)p/p = T$$

$S_p$ - speedup

$$S_p = T_1/T_p$$

$$S_p = \textbf{?}$$

# Gustafson's Law Derivation




More work in the same Time

$T$ - sequential time of original work

$T_1$ - sequential time with work*p

$f$ - sequential fraction

$$T_1 = Tf + T(1-f)p$$

$T_p$ - parallel time on p processors

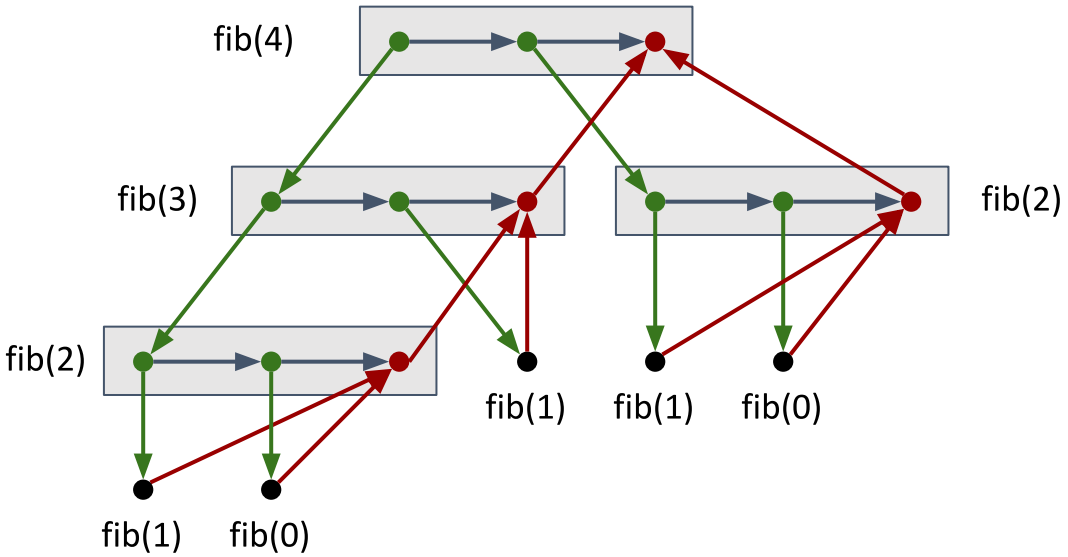
$$T_p = Tf + T(1-f)p/p = T$$

$S_p$ - speedup

$$S_p = T_1/T_p$$

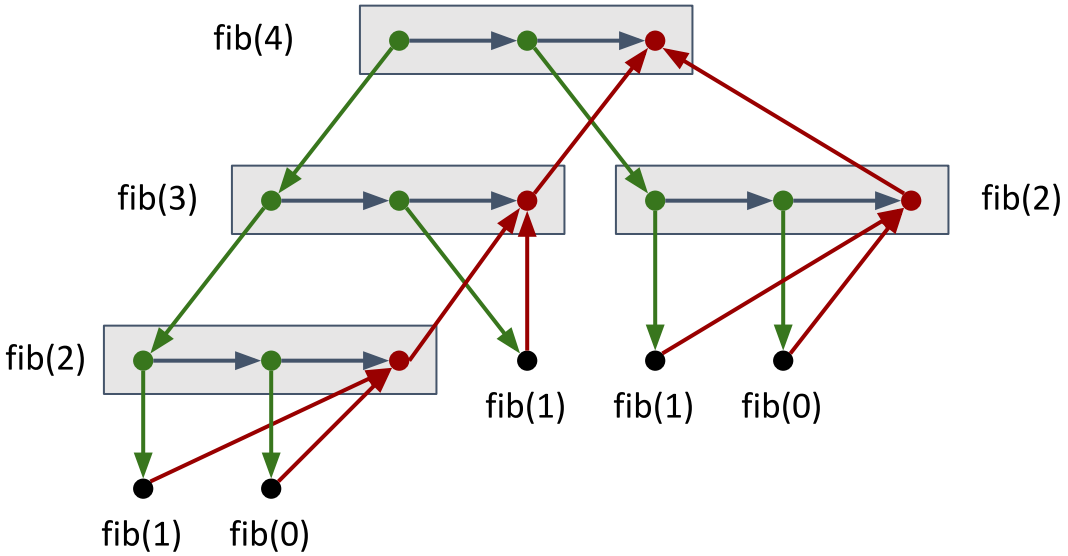
$$S_p = f + (1-f)p$$

# **fib(4)** task graph




```
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




```
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
    }
}
```

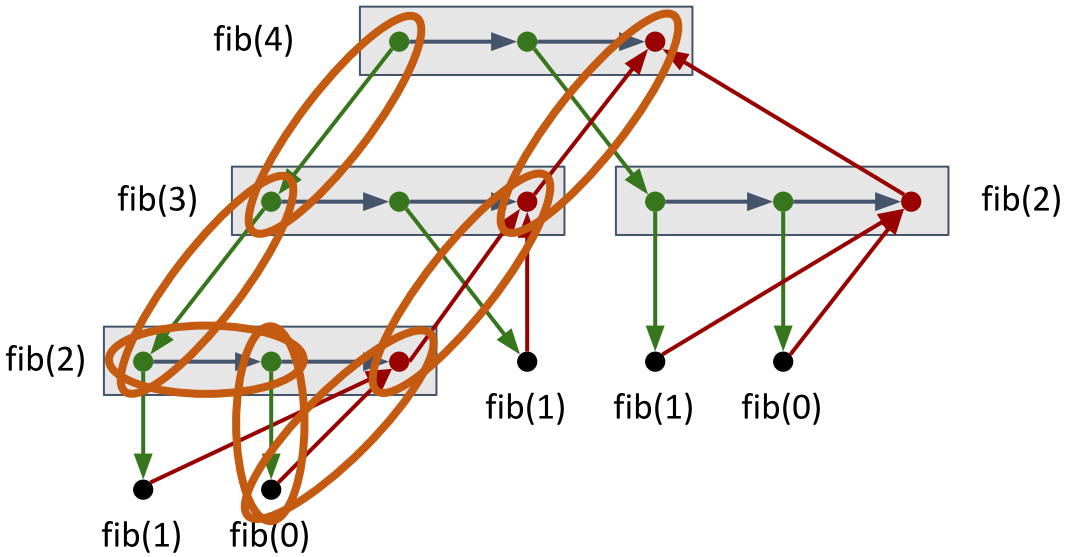
## What is a task?

base case  spawn  wait

## What is an edge?

same procedure  spawn  wait

# **fib(4)** task graph




**critical path length is 8 tasks**

```
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
    }
}
```

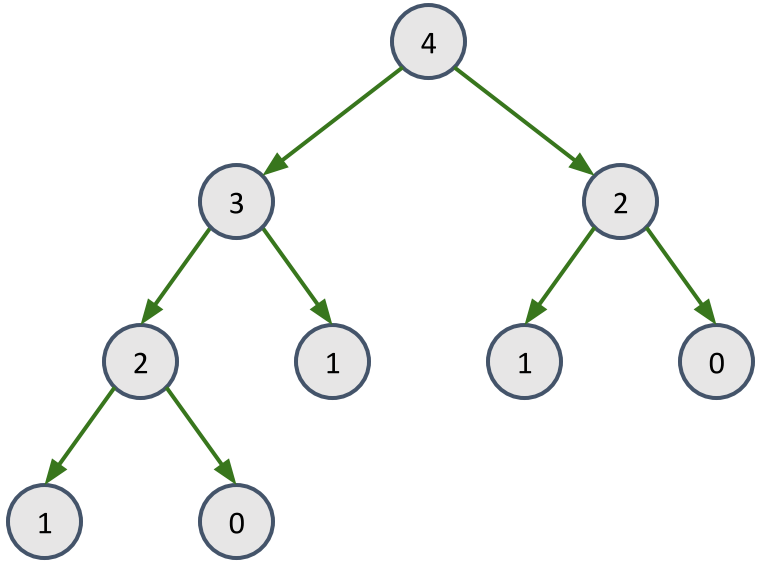
**What is a task?**

**base case** **spawn** **wait**

**What is an edge?**

**same procedure** **spawn** **wait**

# **fib(4)** simplified task graph




**Simpler at the expense of not modelling joins and inter-process dependencies**

```
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
    }
}
```

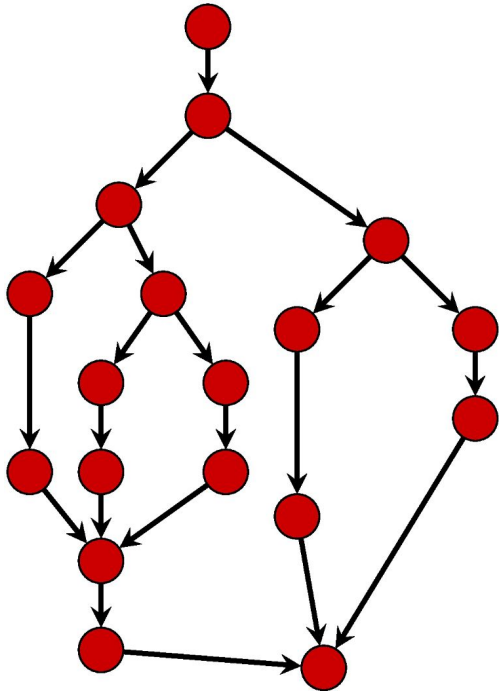
**What is a task?**

**Call to Fibonacci**

**What is an edge?**

**spawn**
**(no dependency within same procedure)**

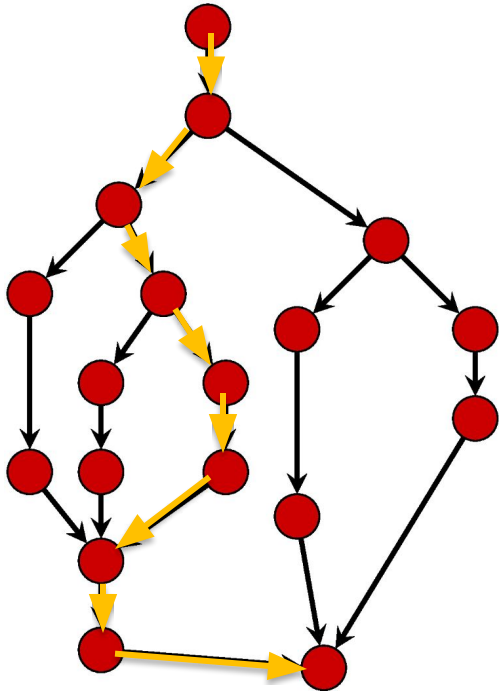
# Task Graphs



Critical path: path from start to end that takes the longest (for some metric)

Example: #nodes

# Task Graphs

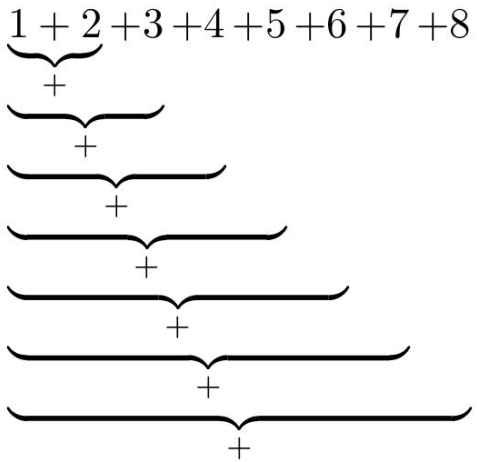


Critical path: path from start to end that takes the longest (for some metric)

Example: #nodes

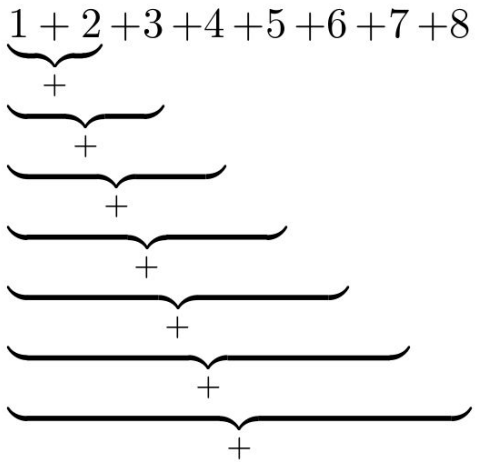
# Task Graphs

Adding eight numbers:



# Task Graphs

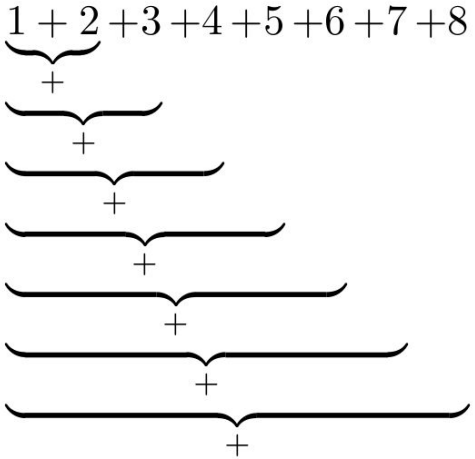
Adding eight numbers:



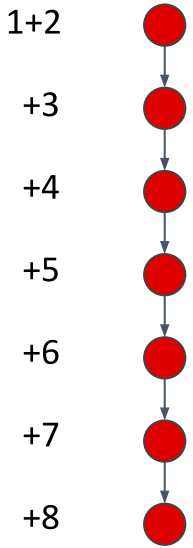

What is the corresponding task graph?

# Task Graphs

Adding eight numbers:

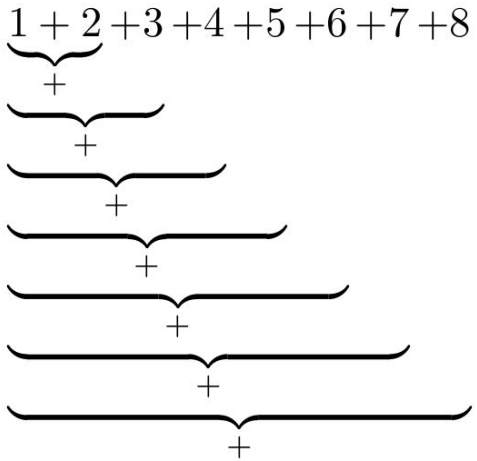



What is the corresponding task graph?

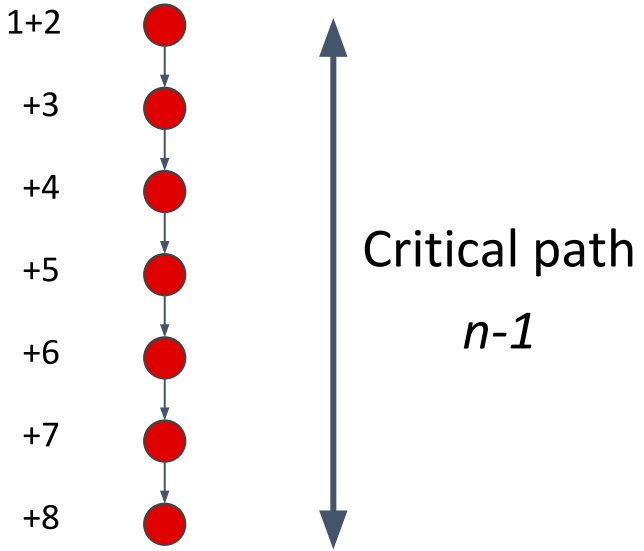


# Task Graphs

Adding eight numbers:

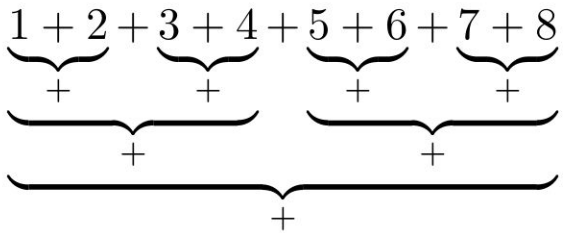



What is the corresponding task graph?



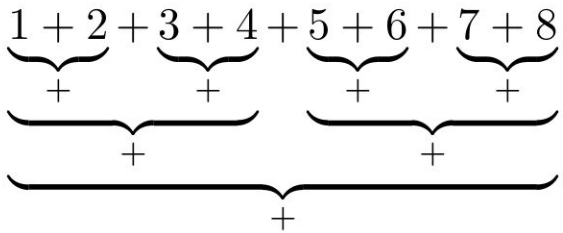
# Task Graphs

Adding eight numbers:



# Task Graphs

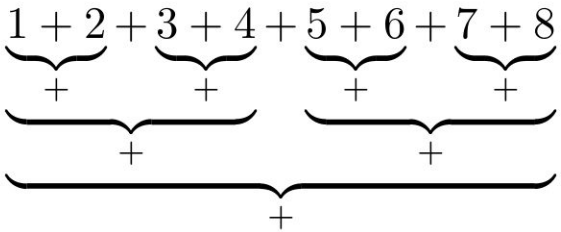
Adding eight numbers:



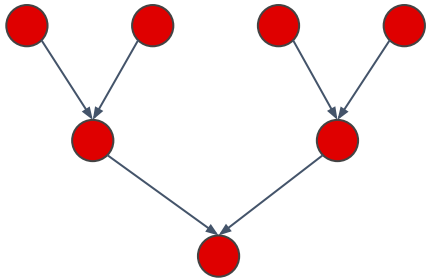

What is the corresponding task graph?

# Task Graphs

Adding eight numbers:

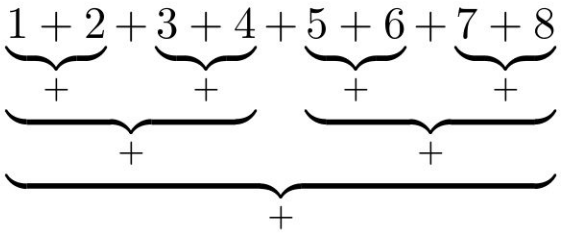



What is the corresponding task graph?

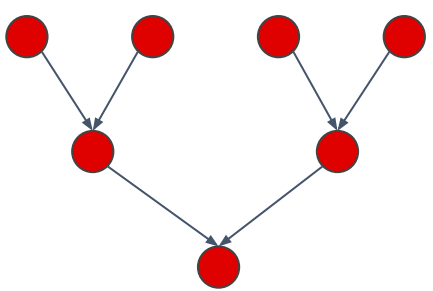


# Task Graphs

Adding eight numbers:




What is the corresponding task graph?



Critical path

*log(n)*

# Search And Count

Search an array of integers for a certain feature and count integers that have this feature:

- Light workload: count number of non-zero values.
- Heavy workload: count how many integers are prime numbers.

We will study single threaded and multi-threaded implementation of the problem.

# Search And Count - Sequential

```
public class SearchAndCountSingle {
  private int[] input;
  private Workload.Type type;

  private SearchAndCountSingle(int[] input, Workload.Type wt) {
    this.input = input;
    this.type = wt;
  }

  private int count() {
    int count = 0;
    for (int i = 0; i < input.length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  }
}
```

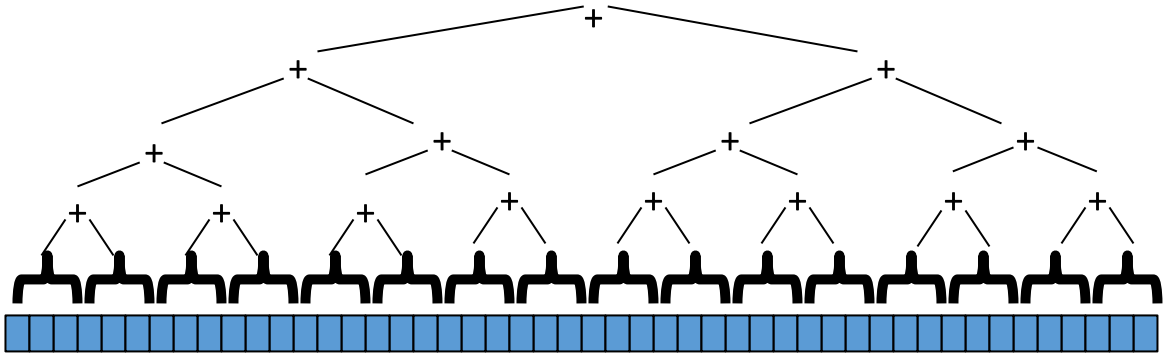
Straightforward implementation. Simply iterate through the input array and count how many times given event occurs.

# Divide and Conquer

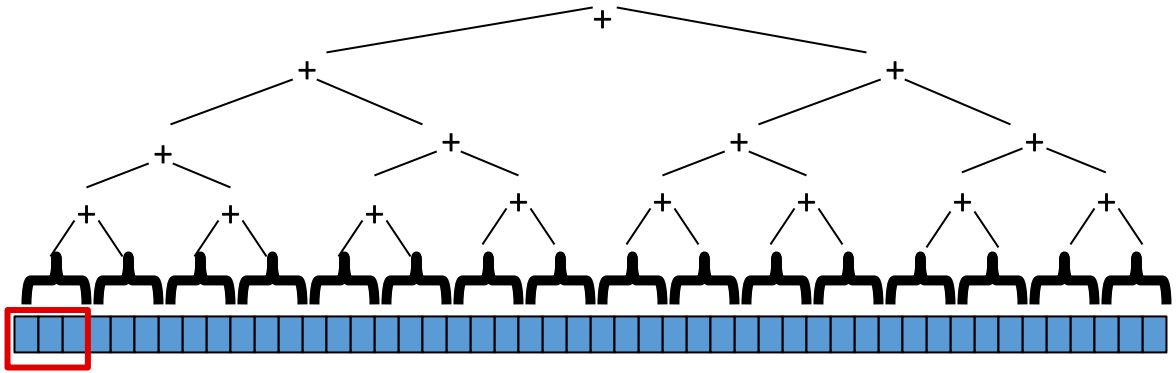
Basic structure of a divide-and-conquer algorithm:

1. If problem is small enough, solve it directly
2. Otherwise
    a. Break problem into subproblems
    b. Solve subproblems recursively
    c. Assemble solutions of subproblems into overall solution

# Divide and Conquer



# Divide and Conquer

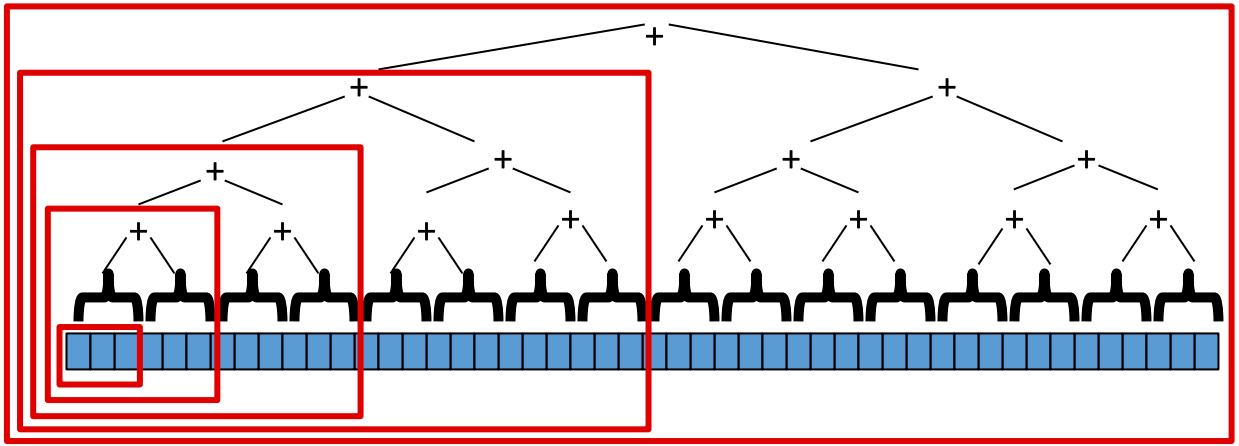


base case

no further split

# Divide and Conquer

Tasks at different
levels of granularity



What determines a task?

i) input array ii) start index iii) length/end index

These are fields we want to store in the task

# Feedback: Tasks A-D

# ExecutorService

## TPS01-J. Do not execute interdependent tasks in a bounded thread pool

Created by Dhruv Mohindra, last modified by Carol J. Lallier on Jun 22, 2015

Bounded thread pools allow the programmer to specify an upper limit on the number of threads that can concurrently execute in a thread pool. Programs must not use threads from a bounded thread pool to execute tasks that depend on the completion of other tasks in the pool.

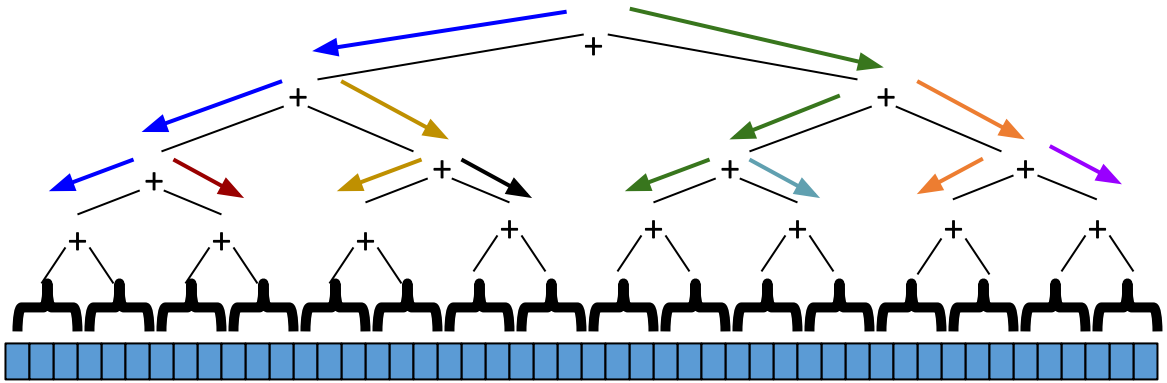
A form of deadlock called *thread-starvation deadlock* arises when all the threads executing in the pool are blocked on tasks that are waiting on an internal queue for an available thread in which to execute. Thread-starvation deadlock occurs when currently executing tasks submit other tasks to a thread pool and wait for them to complete and the thread pool lacks the capacity to accommodate all the tasks at once.

This problem can be confusing because the program can function correctly when fewer threads are needed. The issue can be mitigated, in some cases, by choosing a larger pool size. However, determining a suitable size may be difficult or even impossible.

Similarly, threads in a thread pool may fail to be recycled when two executing tasks each require the other to complete before they can terminate. A blocking operation within a subtask can also lead to unbounded queue growth [Goetz 2006].

# Divide and Conquer Parallelization

thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
thread 8
...

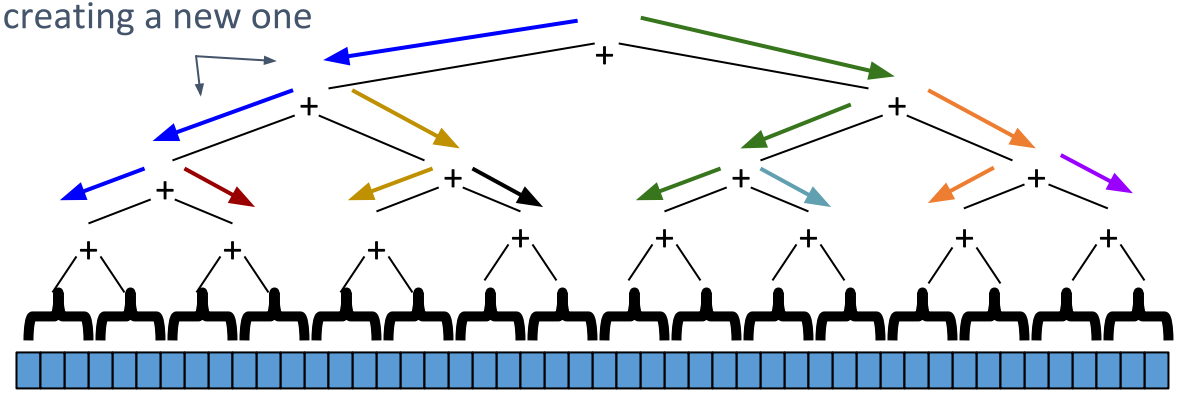


# Divide and Conquer Parallelization

Performance optimization

Same thread is reused instead of creating a new one

thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
thread 8
...

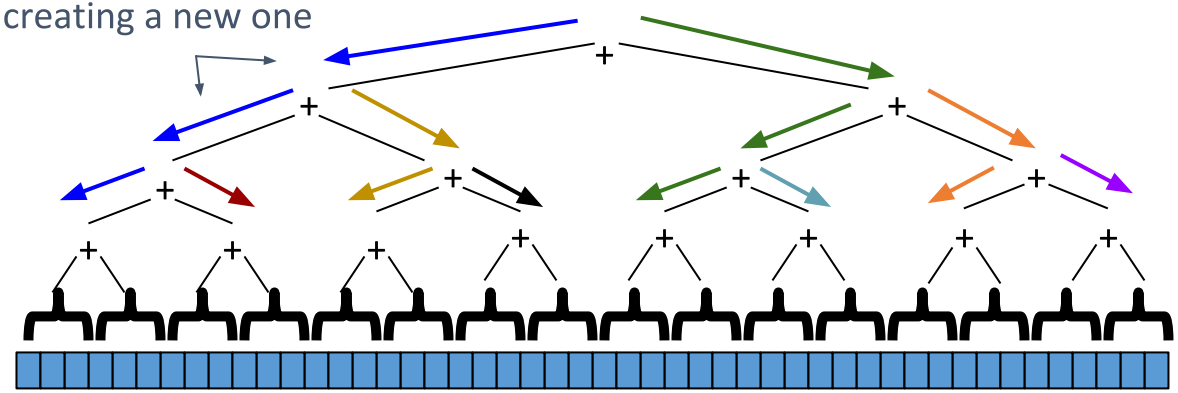


# Divide and Conquer Parallelization

Performance optimization

Same thread is reused instead of creating a new one

thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
thread 8
...

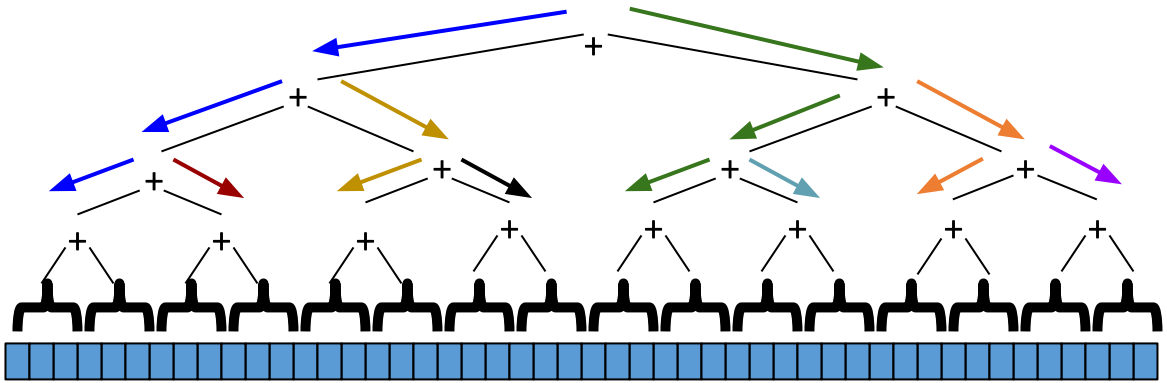


**Task B:**

Extend your implementation such that it creates only a fixed number of threads. Make sure that your solution is properly synchronized when checking whether to create a new thread

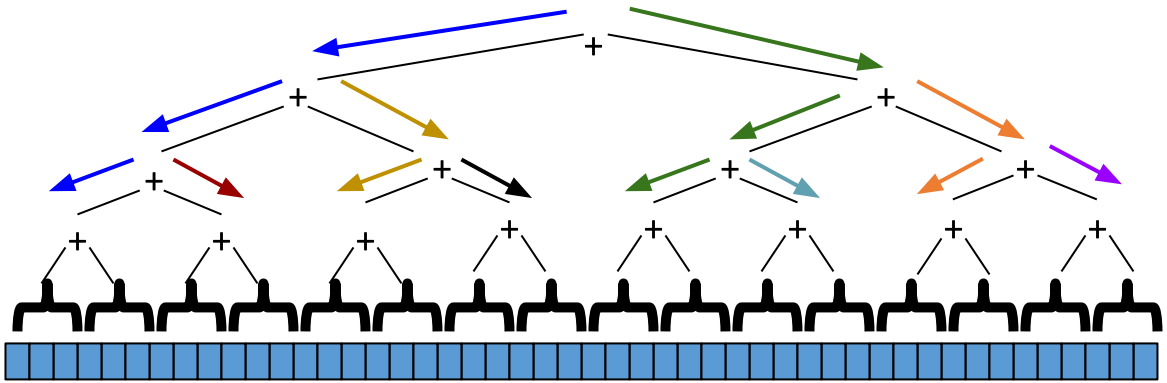
**How to achieve this?**

# Divide and Conquer Parallelization



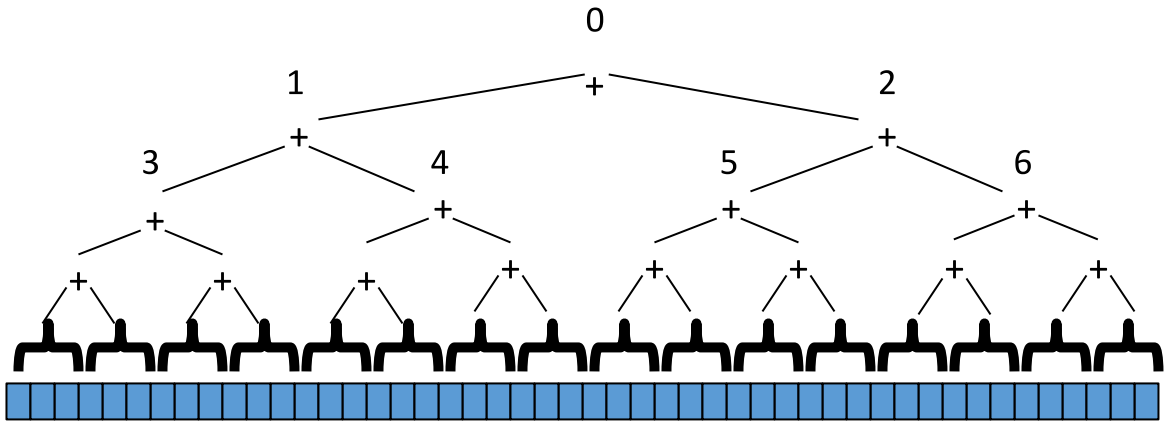
**Option 1:**
Shared counter with
synchronized/atomic access

# Divide and Conquer Parallelization



**Option 1:**
Shared counter with
synchronized/atomic access

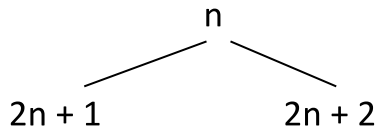
# Divide and Conquer Parallelization

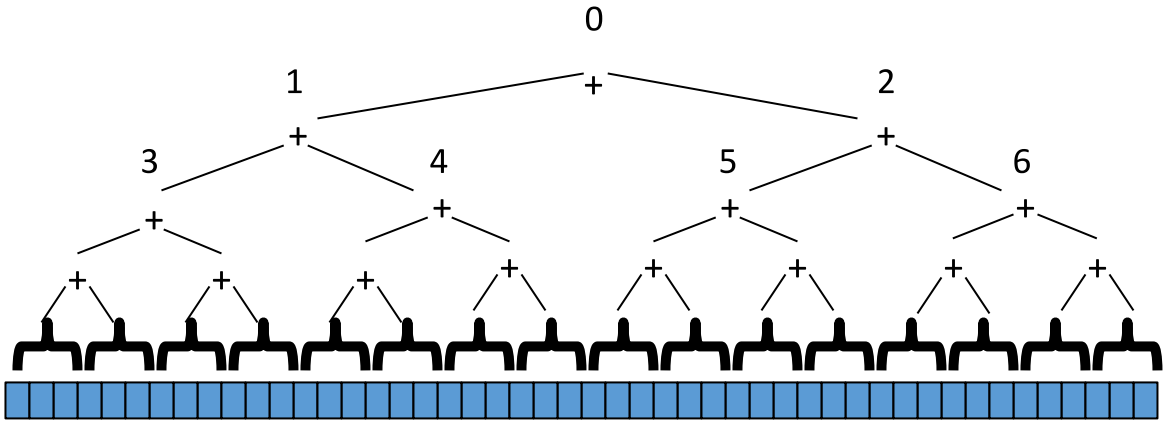



**Option 1:**
Shared counter with synchronized/atomic access

**Option 2:**
Assign unique sequential id to each task. Spawn threads for first N tasks.

# Divide and Conquer Parallelization

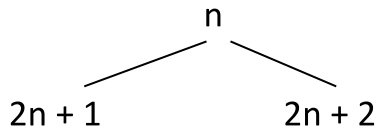


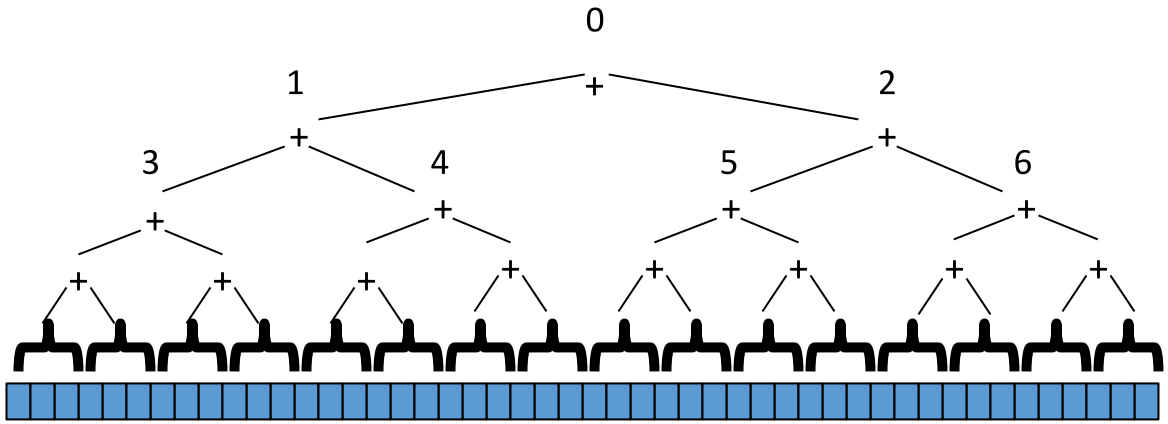




**Option 1:**
Shared counter with synchronized/atomic access

**Option 2:**
Assign unique sequential id to each task. Spawn threads for first N tasks.

# Divide and Conquer Parallelization




**Option 1:**
Shared counter with synchronized/atomic access

**Option 2:**
Assign unique sequential id to each task. Spawn threads for first N tasks.

+ no synchronization required
- imbalanced amount of work

# Divide and Conquer vs Fork/Join

**Divide And Conquer**

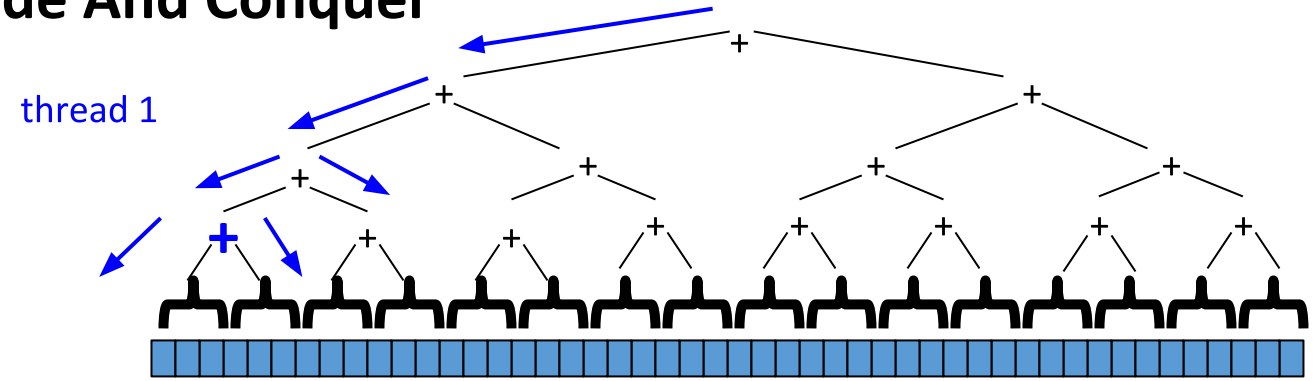
Fundamental design pattern based on recursively breaking down a problem into smaller problems that can be combined to give a solution to the original problem

**Fork/Join**

A framework that supports Divide and Conquer style parallelism

# Divide and Conquer vs Fork/Join

**Divide And Conquer**

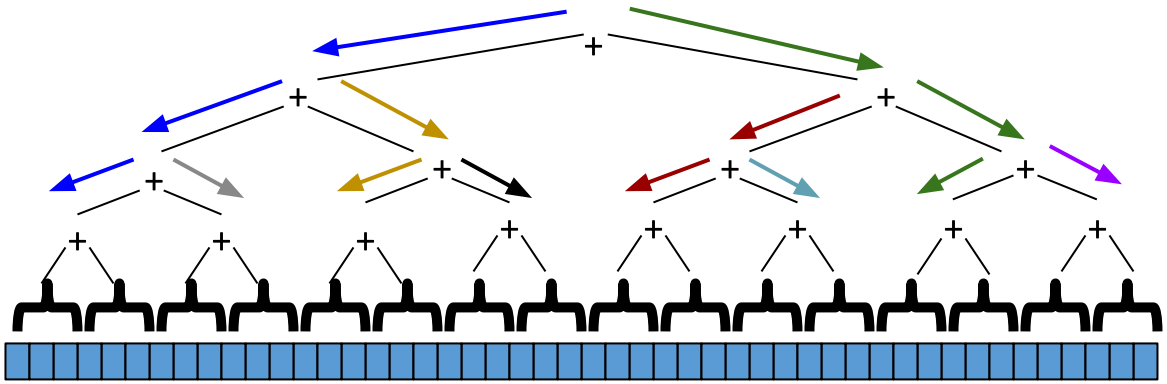



recursively breaking down a problem into smaller problems

*problems are solved sequentially*

# Divide and Conquer vs Fork/Join

thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
...



**Fork/Join**

a framework that supports Divide and Conquer style parallelism

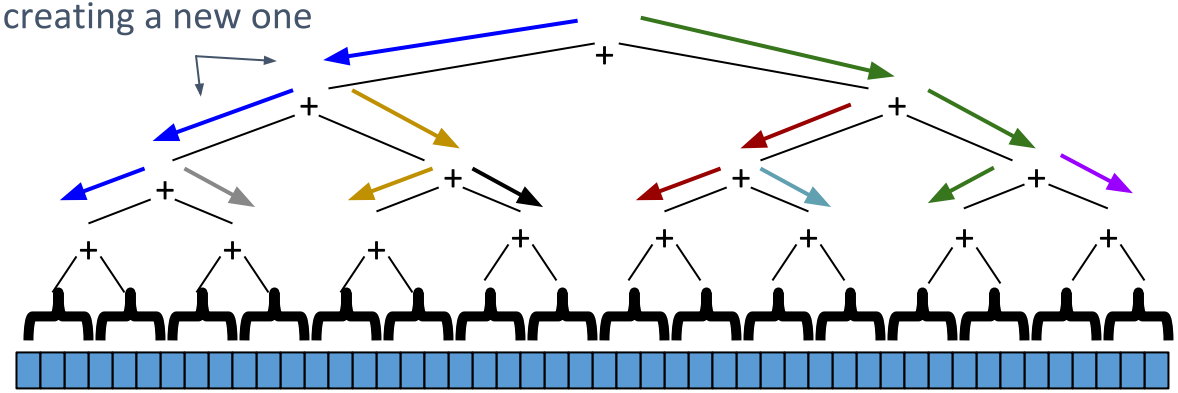
*problems are solved in parallel*

# Divide and Conquer vs Fork/Join

thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
...

Performance optimization

Same thread is reused instead of creating a new one



**Fork/Join**

a framework that supports Divide and Conquer style parallelism

*problems are solved in parallel*

# Search And Count - Task Parallel

Define the task structure:

```
public class SearchAndCountMultiple extends RecursiveTask<Integer> {

	private int[] input;
	private int start;
	private int length;
	private int cutOff;
	private Workload.Type workloadType;

}
```

# Search And Count

```
protected Integer compute() {

}
```

Recall the template for divide and conquer task parallelism

# Search And Count

```
protected Integer compute() {
  if (// work is small) {


    // do the work directly


  else {
    // split work into pieces




    // invoke the pieces and
       wait for the results


    // combine the results
  }
}
```



Recall the template for divide and conquer task parallelism

# Search And Count

```
protected Integer compute() {
  if (// work is small) {


    // do the work directly


  else {
    // split work into pieces




    // invoke the pieces and
       wait for the results


    // combine the results
  }
}
```



Recall the template for divide and conquer task parallelism

Let fill in the template for the search and count task

# Search And Count

```
protected Integer compute() {
  if (// work is small) {

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (// work is small) {

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

Same as sequential implementation

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);

    sc1.fork();
    int count2 = sc2.compute();
    int count1 = sc1.join();

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);

    sc1.fork();
    int count2 = sc2.compute();
    int count1 = sc1.join();

    // combine the results
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)

    // do the work directly

  else {
    // split work into pieces

    // invoke the pieces and
       wait for the results

    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);

    sc1.fork();
    int count2 = sc2.compute();
    int count1 = sc1.join();

    return count1 + count2;
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Search And Count

```
protected Integer compute() {
  if (// work is small)


    // do the work directly


  else {
    // split work into pieces




    // invoke the pieces and
       wait for the results


    // combine the results
  }
}
```

```
protected Integer compute() {
  if (length <= cutOff) {
    int count = 0;
    for (int i = start; i < start + length; i++) {
      if (Workload.doWork(input[i], type)) count++;
    }
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);

    sc1.fork();
    int count2 = sc2.compute();
    int count1 = sc1.join();

    return count1 + count2;
  }
}
```

```
public class SearchAndCountMultiple
   extends RecursiveTask<Integer> {
      private int[] input;
      private int start;
      private int length;
      private int cutOff;
      private Workload.Type type;
}
```

# Exercise 6

# Assignment 6

Task Parallelism:

- Merge Sort
- Longest Sequence

# Merge sort algorithm

In this exercise you will implement the merge sort algorithm using task parallelism.

The merge sort algorithm partitions the array into smaller arrays, sorts each one separately and then merges the sorted arrays.

- By default, the partitioning of the array continues recursively until the array size is 1 or 2, which then is sorted trivially.
- Try larger cutoff values (e.g partition arrays down to minimum size 4 instead of 2) and see how this affects the algorithm performance.
- Discuss the asymptotic running time of the algorithm and the obtained speedup.

# Longest Sequence

Given a sequence of numbers:

[1, 9, 4, 3, 3, 8, 7, 7, 7, 0]

find the longest sequence of the same consecutive number

# Longest Sequence

Given a sequence of numbers:

[1, 9, 4, 3, 3, 8, 7, 7, 7, 0]

find the longest sequence of the same consecutive number

# Longest Sequence

Given a sequence of numbers:

[1, 9, 4, 3, 3, 8, 7, 7, 7, 0]

find the longest sequence of the same consecutive number.

If multiple sequences have the same length, return the first one (the one with lowest starting index)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[1, 1, 0, 0]

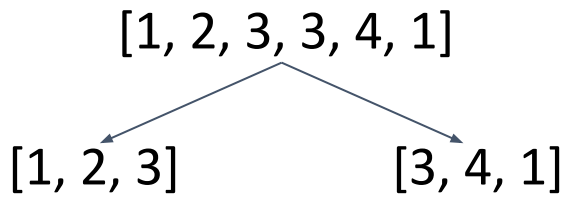
# Longest Sequence

***Task:***

Implement task parallel version that finds the longest sequence of the same consecutive number.

***Challenge:***

The input array cannot be divided arbitrarily. For example:



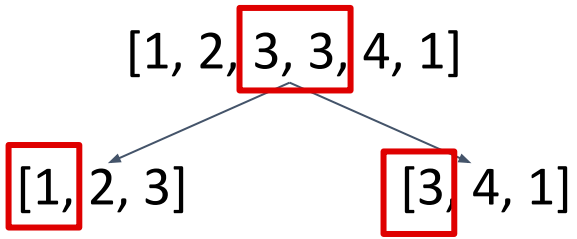
# Longest Sequence

***Task:***

Implement task parallel version that finds the longest sequence of the same consecutive number.

***Challenge:***

The input array cannot be divided arbitrarily. For example:




Combining results of subtasks does not give the correct answer!