Parallele Programmierung FS25

Exercise Session 6

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- Theory Recap
- Intro Exercise 6
- Exam Questions
- Kahoot

Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too

Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too
- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9

Organisation

- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9
- Falls ihr Feedback möchtet kommt bitte zu mir

Exam Preparation Session

- Monday, March 31, 11:15 – 12:00
- Tuesday, April 1, 10:15 – 12:00

- HG F 5 / HG F 7

- Hosted by Vera Schubert and Jackson Stanhope

Organisation

- Wo sind wir jetzt?

Date	Title
Feb 17	Introduction & Course Overview
Feb 18	Java Recap and JVM Overview
Feb 24	Introduction to Threads and Synchronization (Part I)
Feb 25	Introduction to Threads and Synchronization (Part II)
Mar 3	Introduction to Threads and Synchronization (Part III)
Mar 4	Parallel Architectures: Parallelism on the Hardware Level
Mar 10	Basic Concepts in Parallelism
Mar 11	Divide & Conquer and Executor Service
Mar 17	DAG and ForkJoin Framework
Mar 18	Parallel Algorithms (Part I)
Mar 24	Parallel Algorithms (Part II)
Mar 25	Shared Memory Concurrency, Locks and Data Races
Mar 31	Virtual Threads
Apr 01	Exam Preparation (First Half)

Plan für heute

- Organisation
- **Nachbesprechung Exercise 5**
- Theory Recap
- Intro Exercise 6
- Exam Questions
- Kahoot

Recall: Amdahl's vs Gustafson's Law

The key goal is to:

- ➔ Understand the main difference and implications (i.e., when to use which formula)
- ➔ Know how to derive the formulas based on your understanding, not because you memorized them for the exam

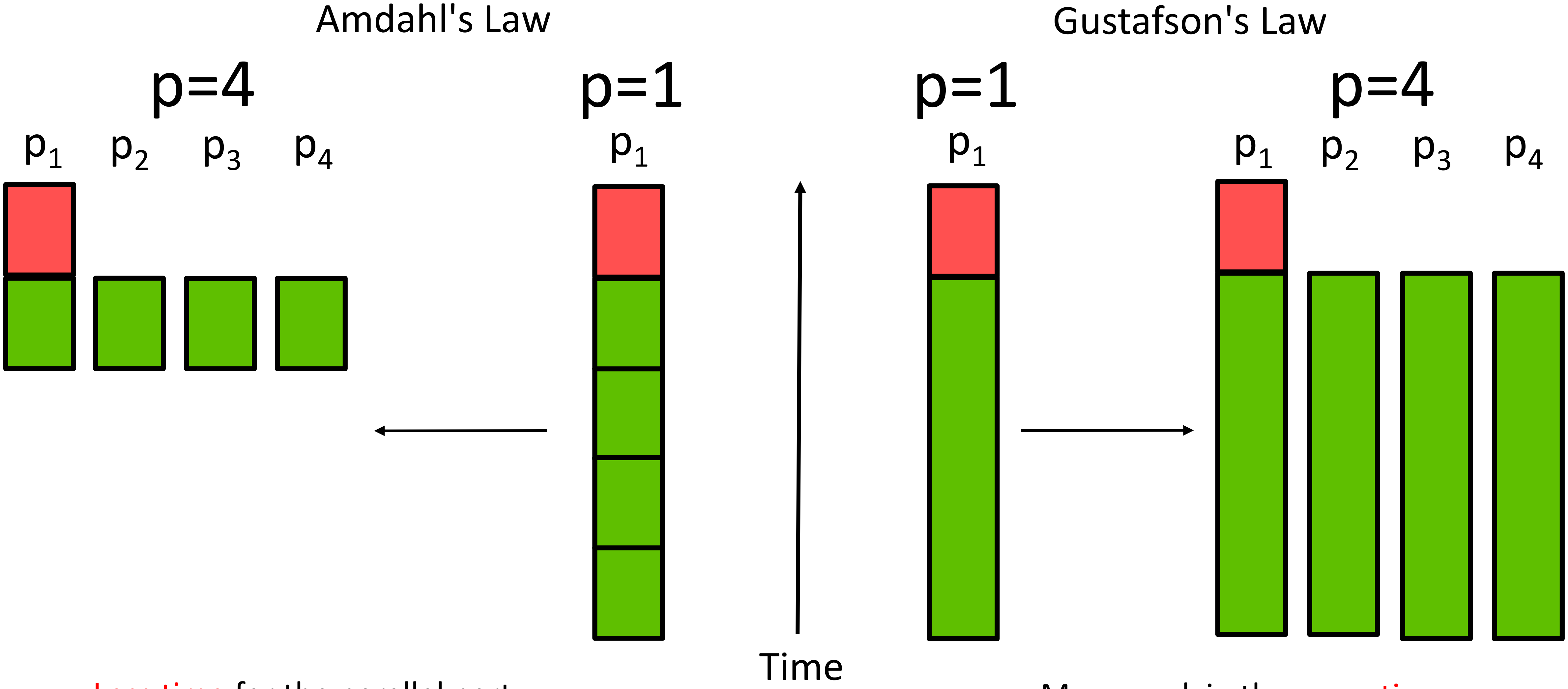
Recall: Amdahl's vs Gustafson's Law

The key goal is to:

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

➔ Know how to derive the formulas based on your understanding, not because you memorized them for the exam

Recall: Amdahl's vs Gustafson's Law



Less time for the parallel part

More work in the same time

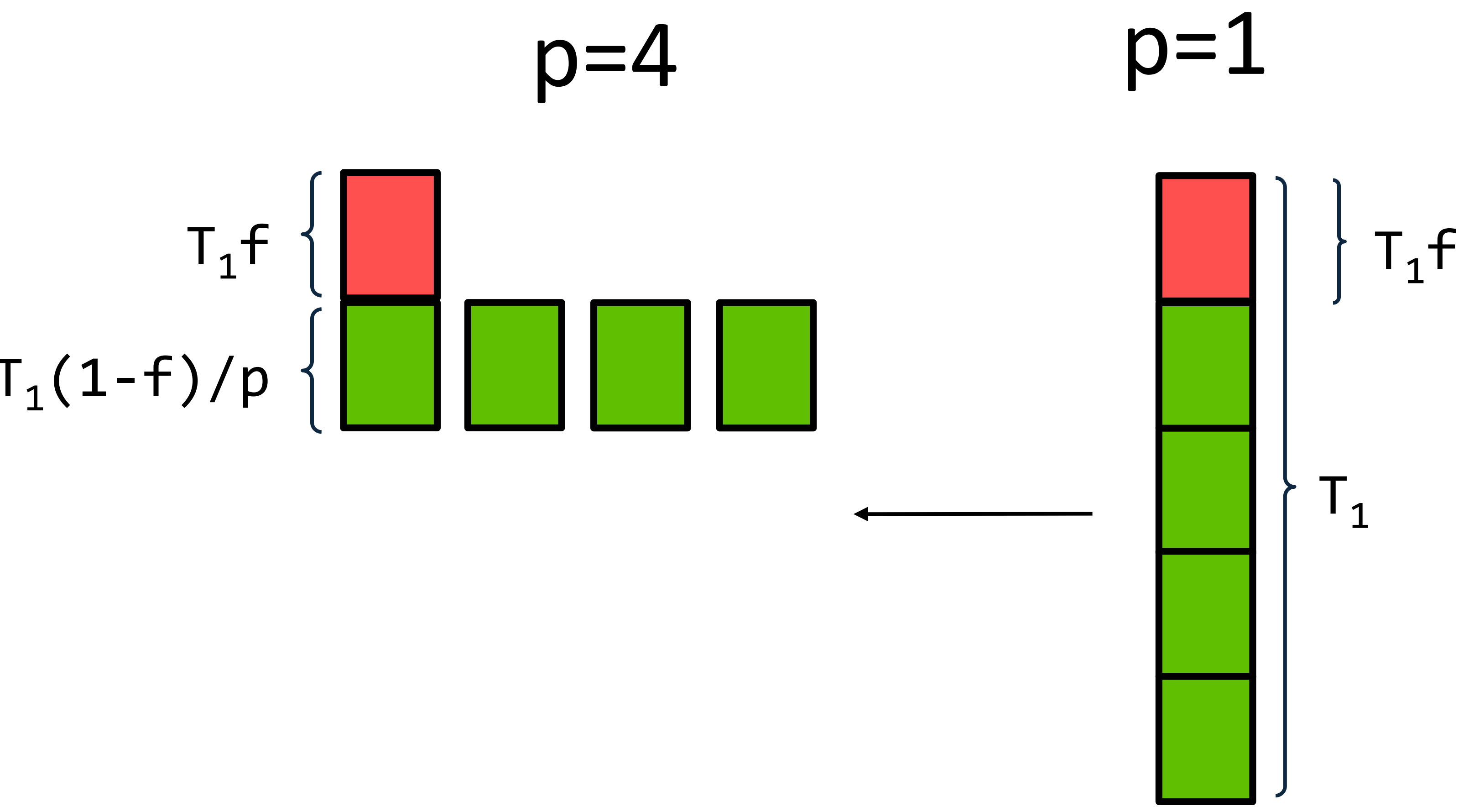
Recall: Amdahl's vs Gustafson's Law

The key goal is to:

- ➔ Understand the main difference and implications (i.e., when to use which formula)
- ➔ **Know how to derive the formulas based on your understanding, not because you memorized them for the exam**

Amdahl's Law Derivation

Amdahl's Law



Less time for the parallel part

T_1 - sequential time

f - sequential fraction

T_p - parallel time on p processors

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

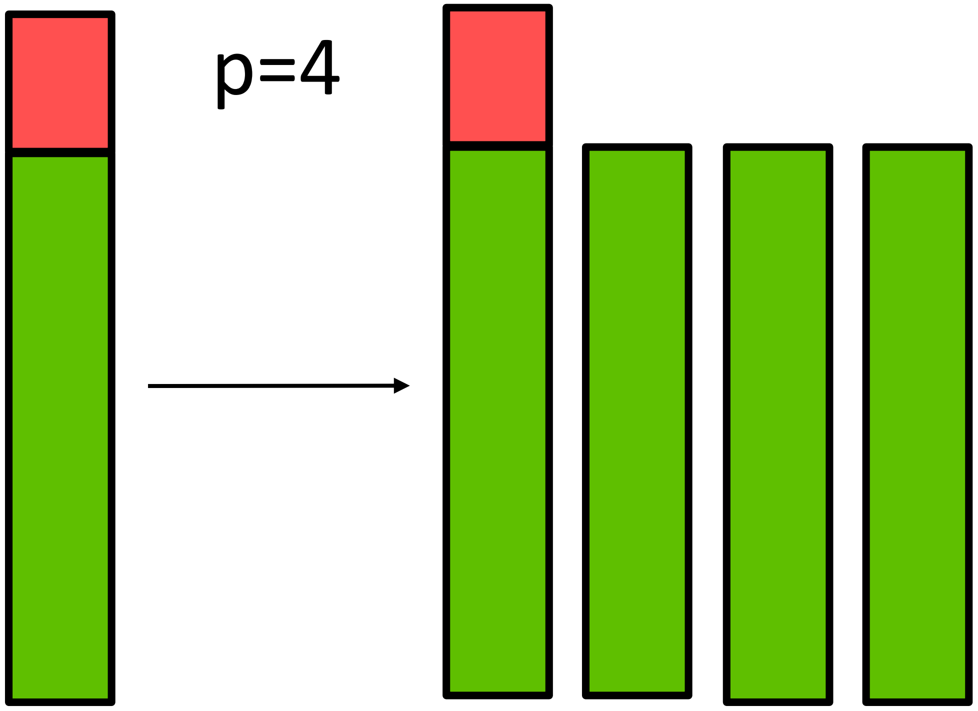
S_p - speedup

$$S_p <= T_1/T_p$$

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

Gustafson's Law Derivation

Gustafson's Law



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 = ?

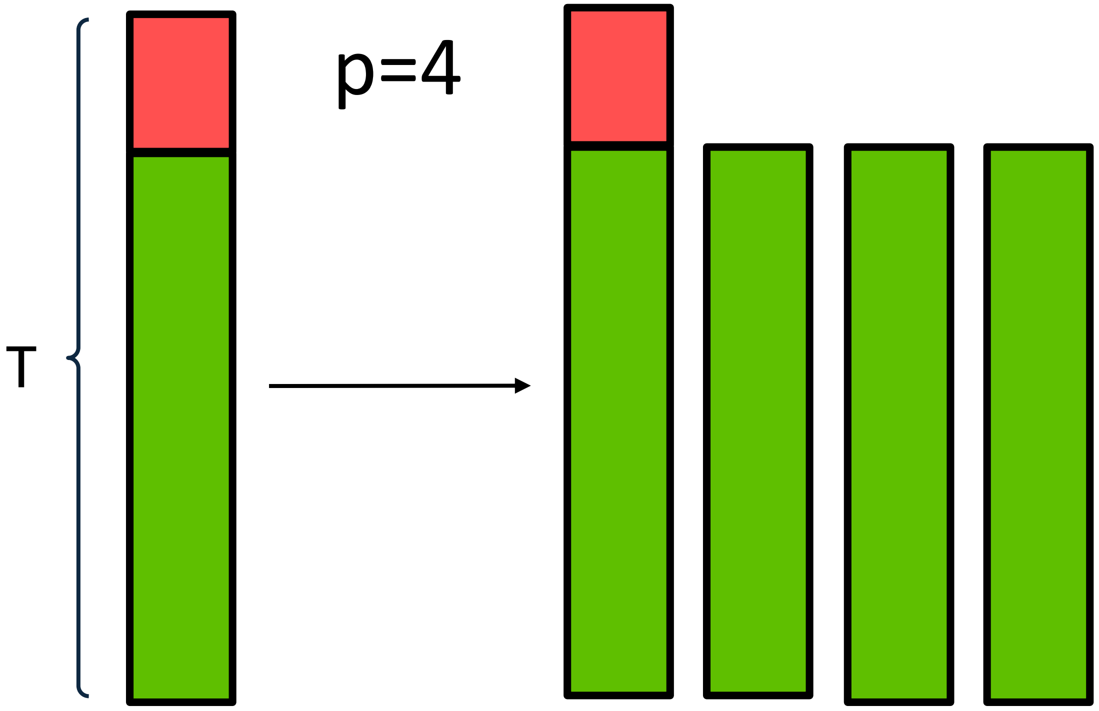
S_p - speedup

$S_p = T_1/T_p$

S_p = ?

Gustafson's Law Derivation

Gustafson's Law



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 = ?

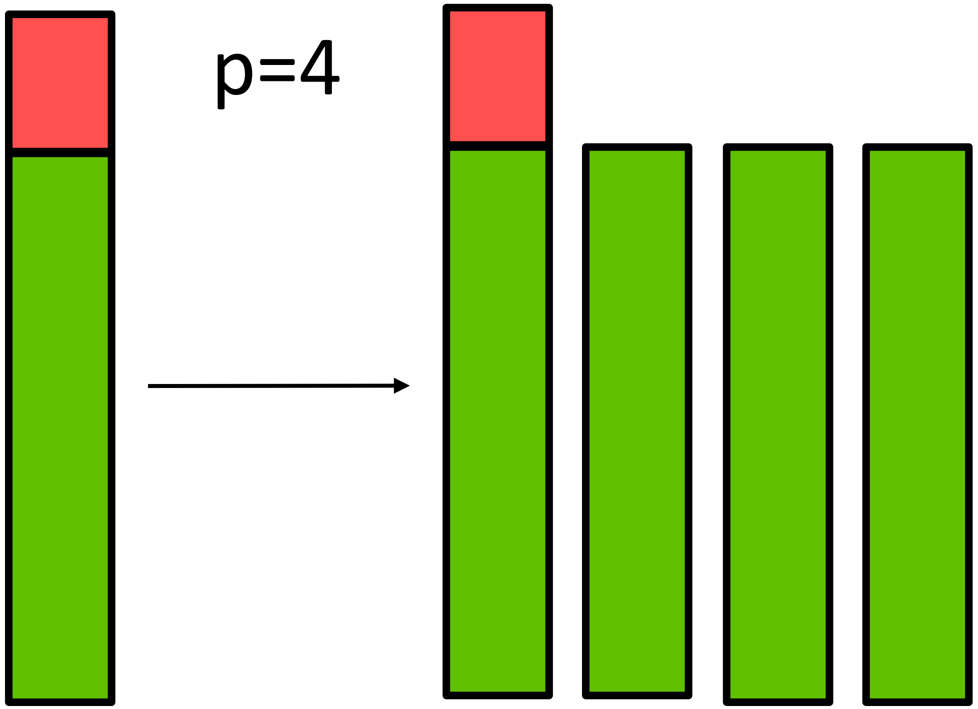
S_p - speedup

$S_p = T_1/T_p$

S_p = ?

Gustafson's Law Derivation

Gustafson's Law



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

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

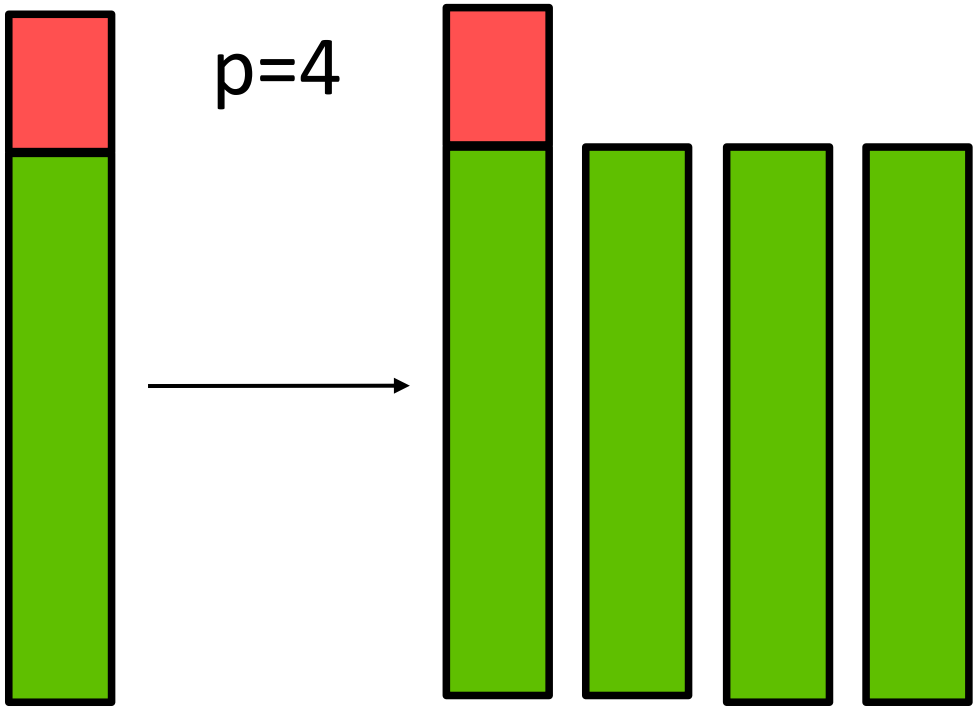
S_p - speedup

$$S_p = T_1/T_p$$

$$S_p = ?$$

Gustafson's Law Derivation

Gustafson's Law



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

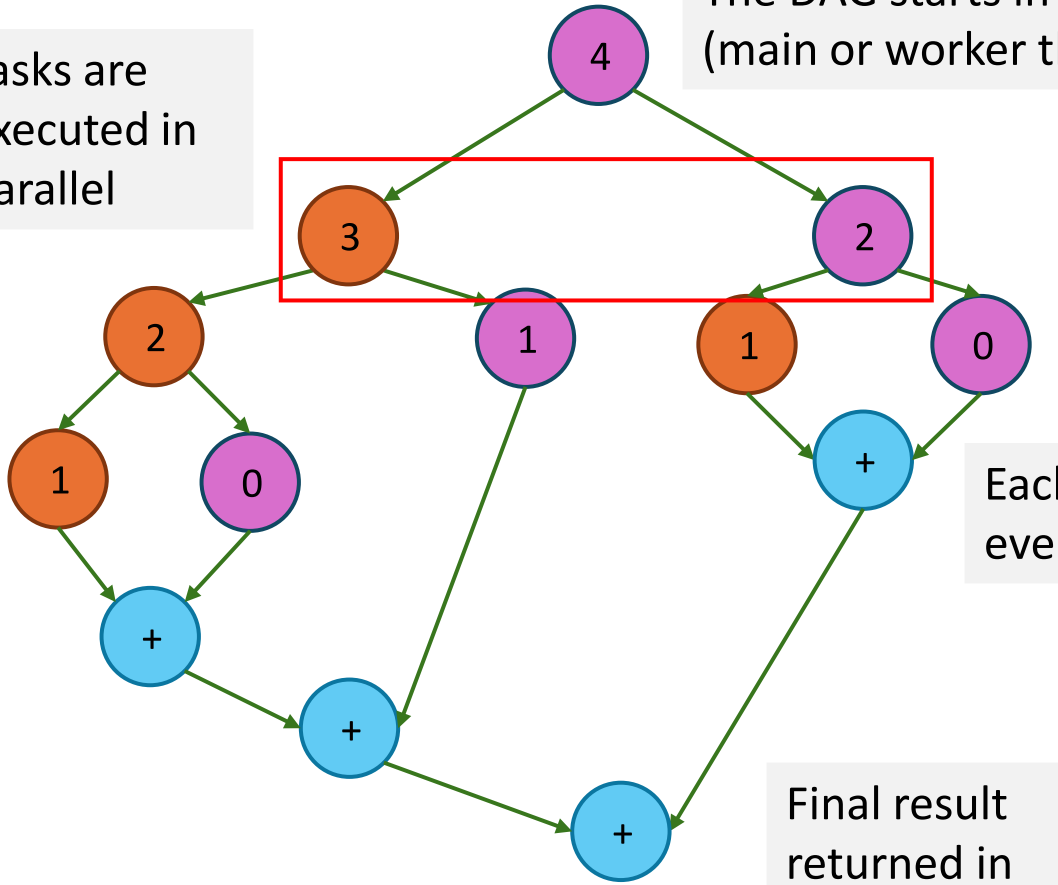
$$S_p = T_1/T_p$$

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

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 FJ



Tasks are executed in parallel

The DAG starts in a single thread (main or worker thread)

Each forked task eventually joins

Final result returned in single thread

```
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
    }
}
```

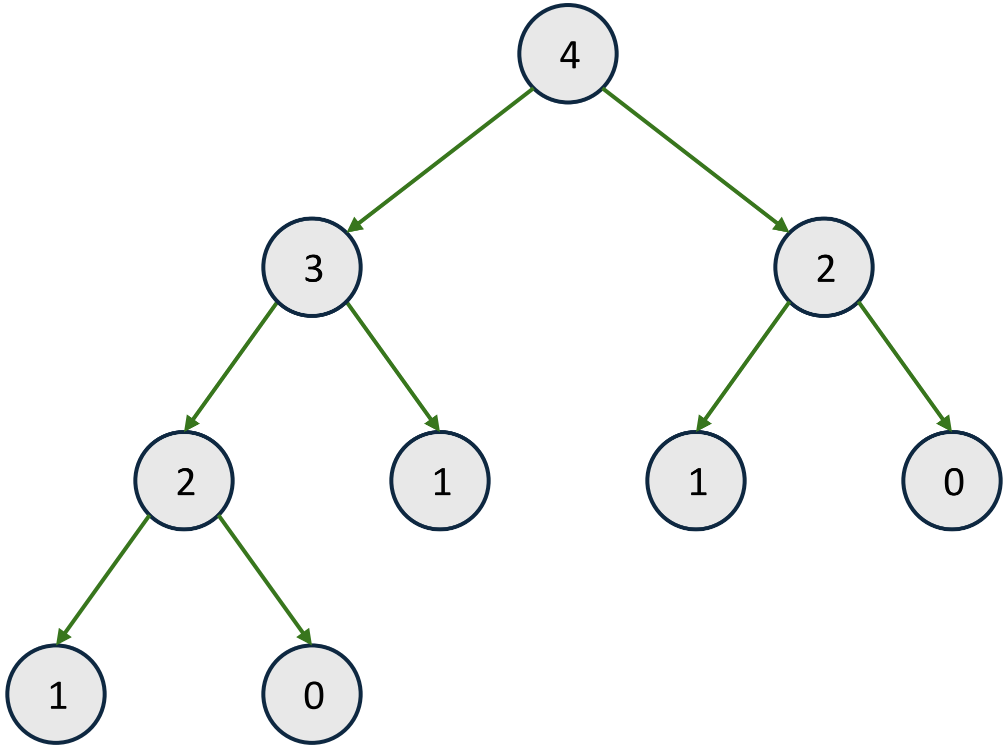
What is a task?

`new forked task`, `continuation of current task`, `join`

What is an edge?

`spawn, same procedure, wait`

fib(4) simplified 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
    }
}
```

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

What is a task?

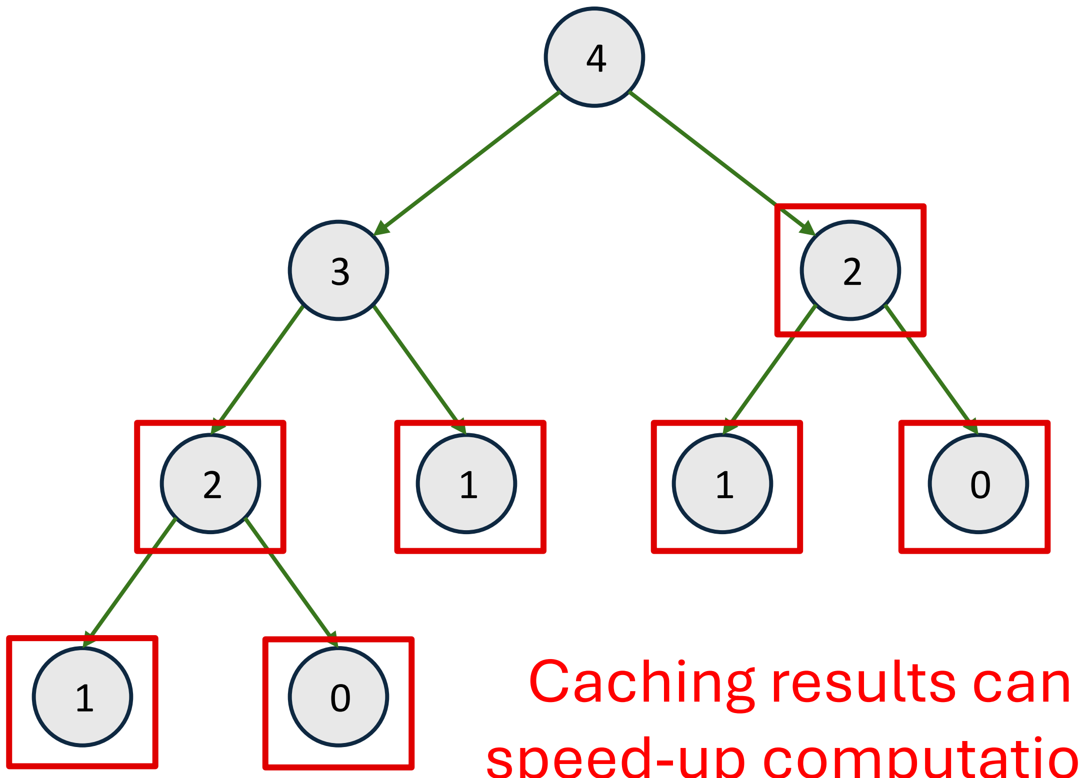
`Call to Fibonacci`

What is an edge?

→

spawn
(no dependency within same procedure)

fib(4) simplified task graph



Caching results can speed-up computation

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
    }
}
```

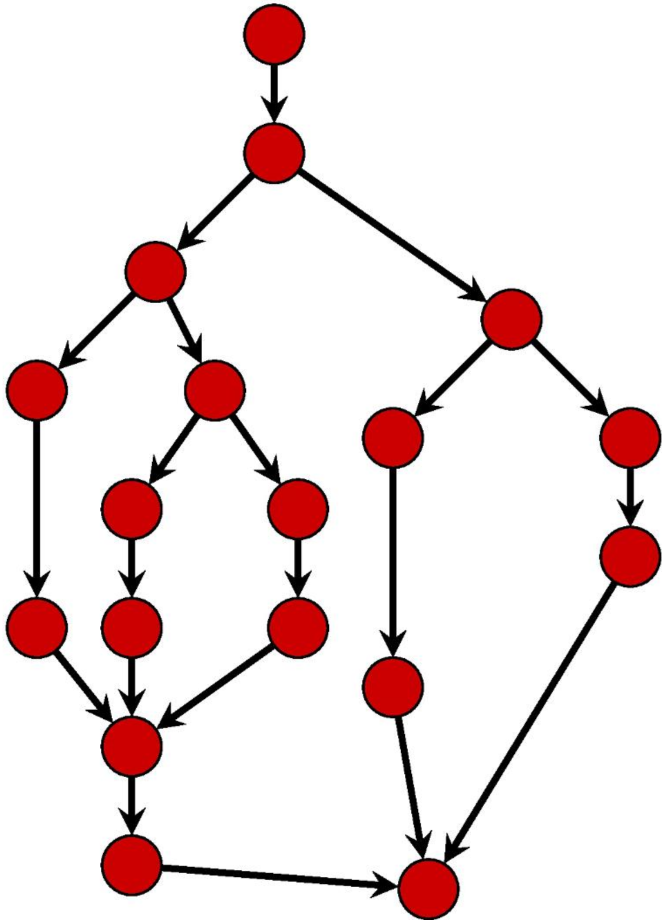
What is a task?

`Call to Fibonacci`

What is an edge?

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

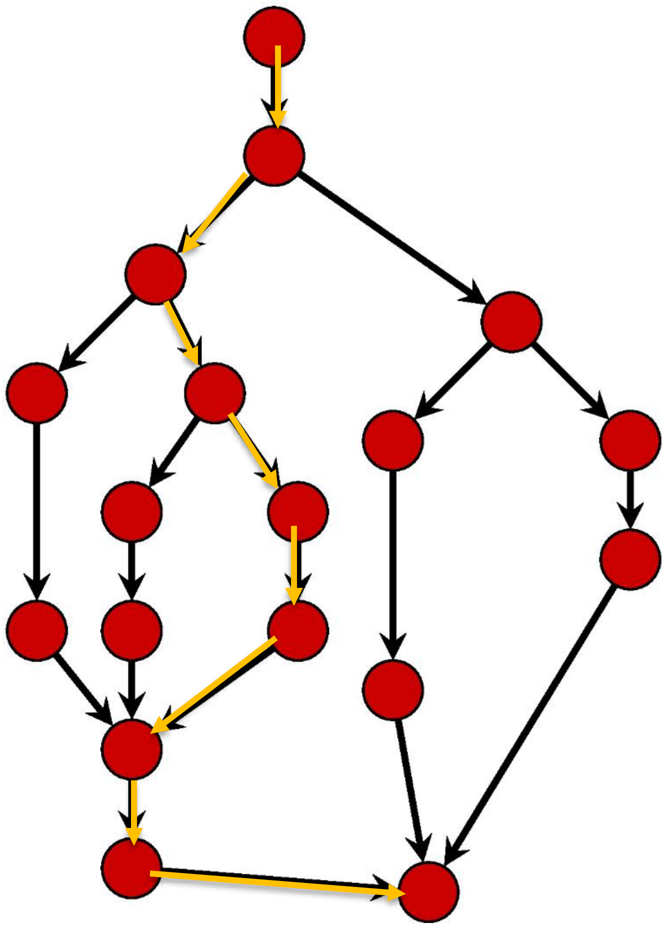
Task Graphs



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

Example: #nodes

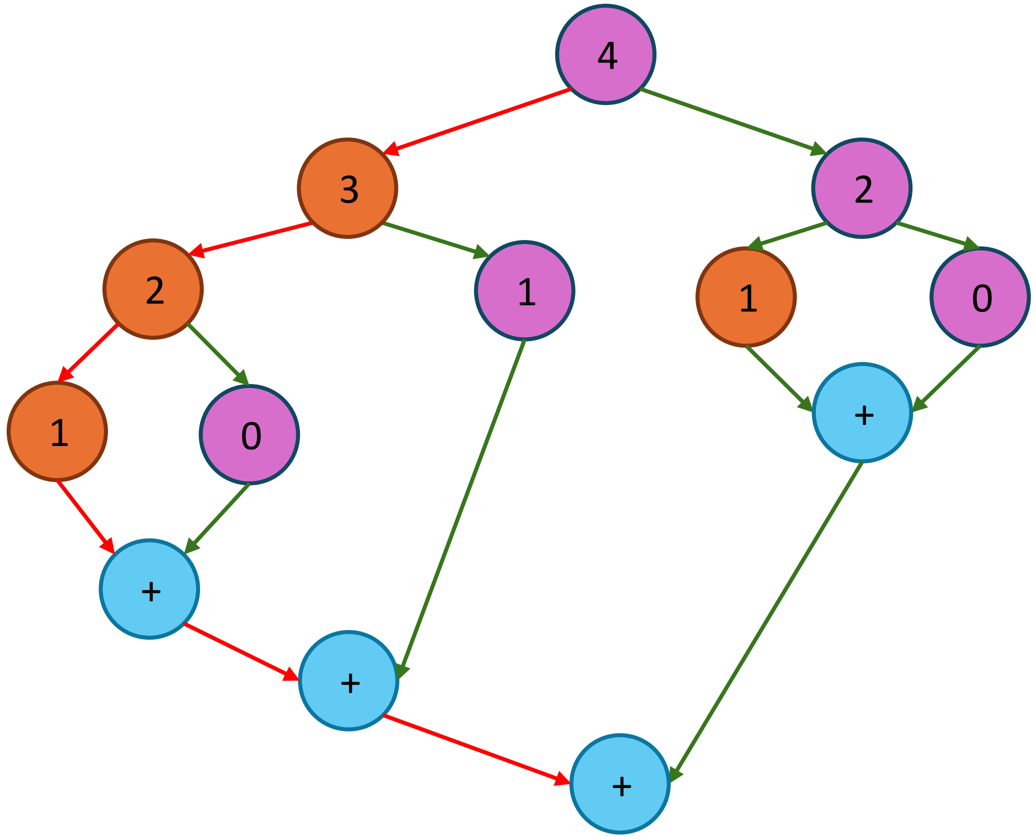
Task Graphs



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

Example: #nodes

fib(4) task graph FJ



critical path length is 7 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
    }
}
```

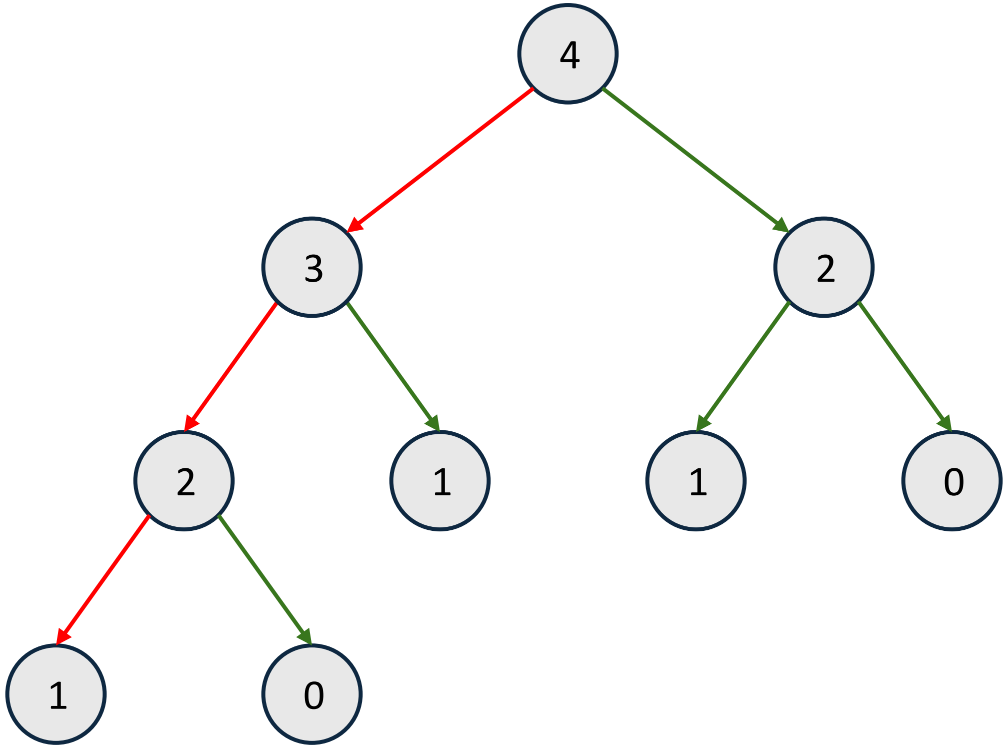
What is a task?

new forked task, continuation of current task, join

What is an edge?

spawn, same procedure, wait

fib(4) simplified task graph



critical path length is 4 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?

Call to Fibonacci

What is an edge?

spawn
(no dependency within same procedure)

Task Graph Simplified

Task: Call to add()

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{1+2}_{+}+3}_{+}+4}_{+}+5}_{+}+6}_{+}+7}_{+}+8}_{+}$$

Task Graph Simplified

Task: Call to add()

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{1+2}_{+}+3}_{+}+4}_{+}+5}_{+}+6}_{+}+7}_{+}+8}_{+}$$

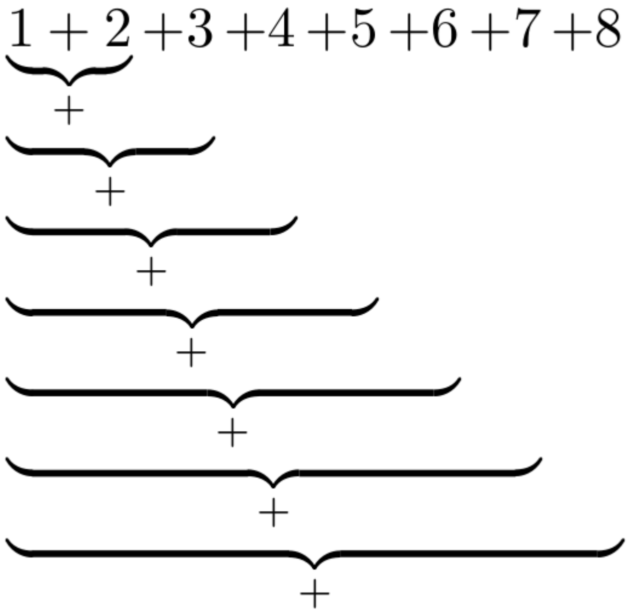
What is the corresponding task graph?

Task Graph Simplified

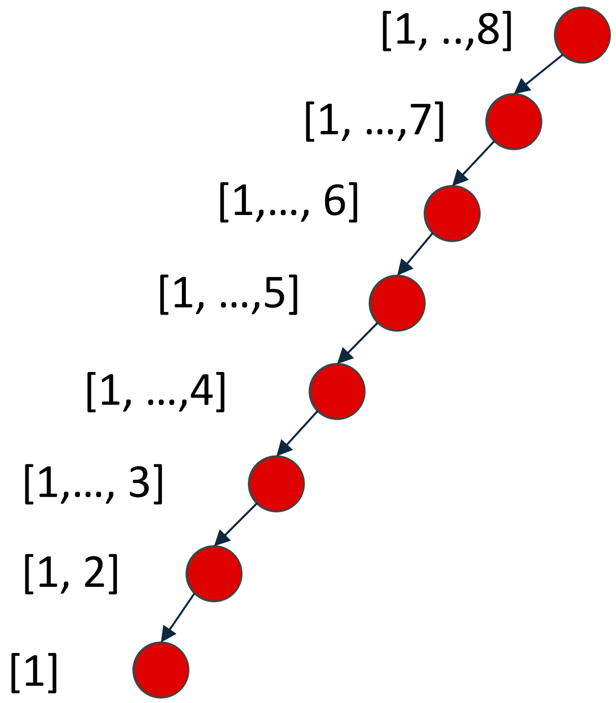
Task: Call to add()

Cut-off: 1

Adding eight numbers:



What is the corresponding task graph?



essential

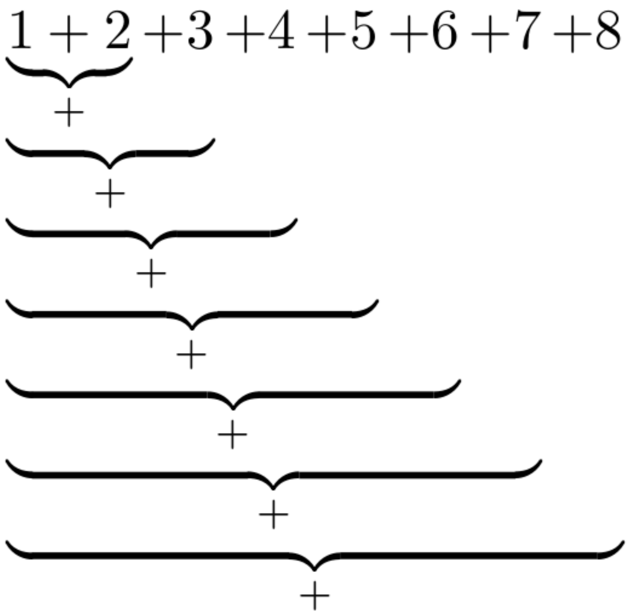
Task Graph Simplified

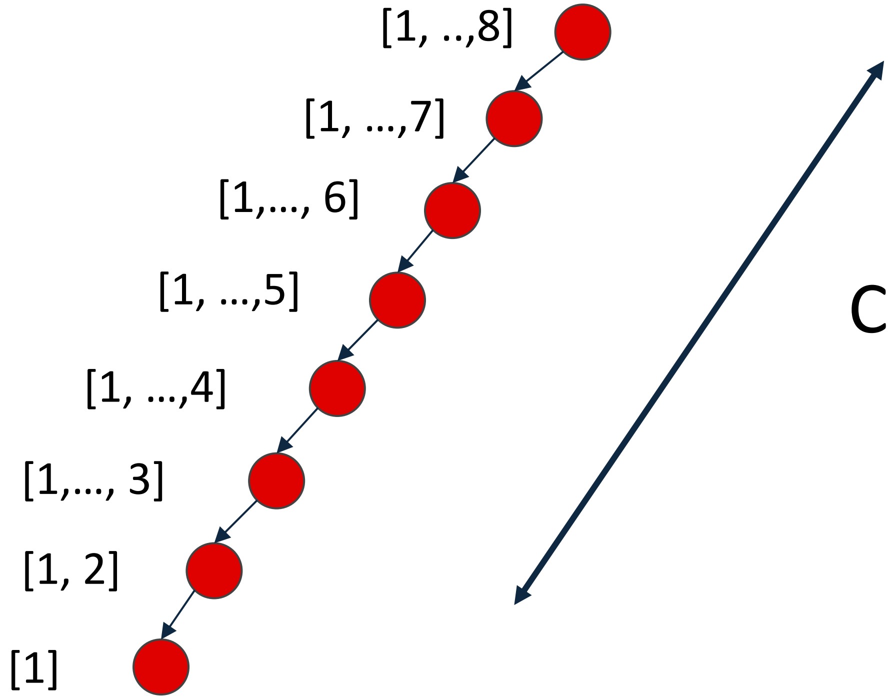
Task: Call to add()

Cut-off: 1

What is the corresponding task graph?

Adding eight numbers:





Critical path

8

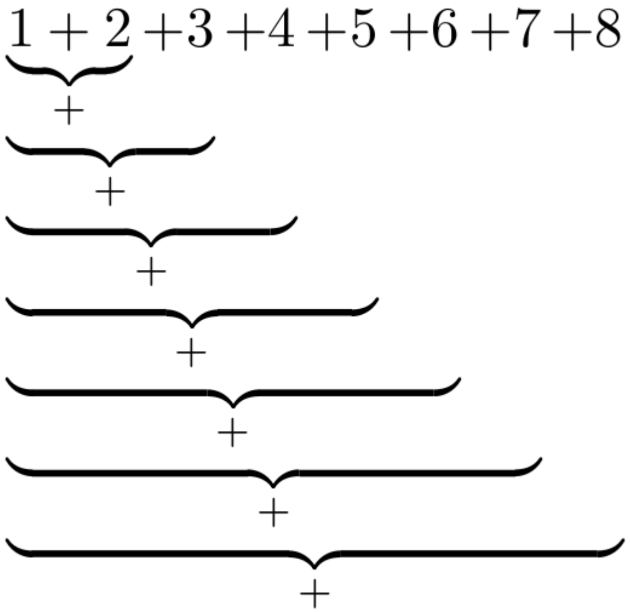
essential

Task Graph FJ

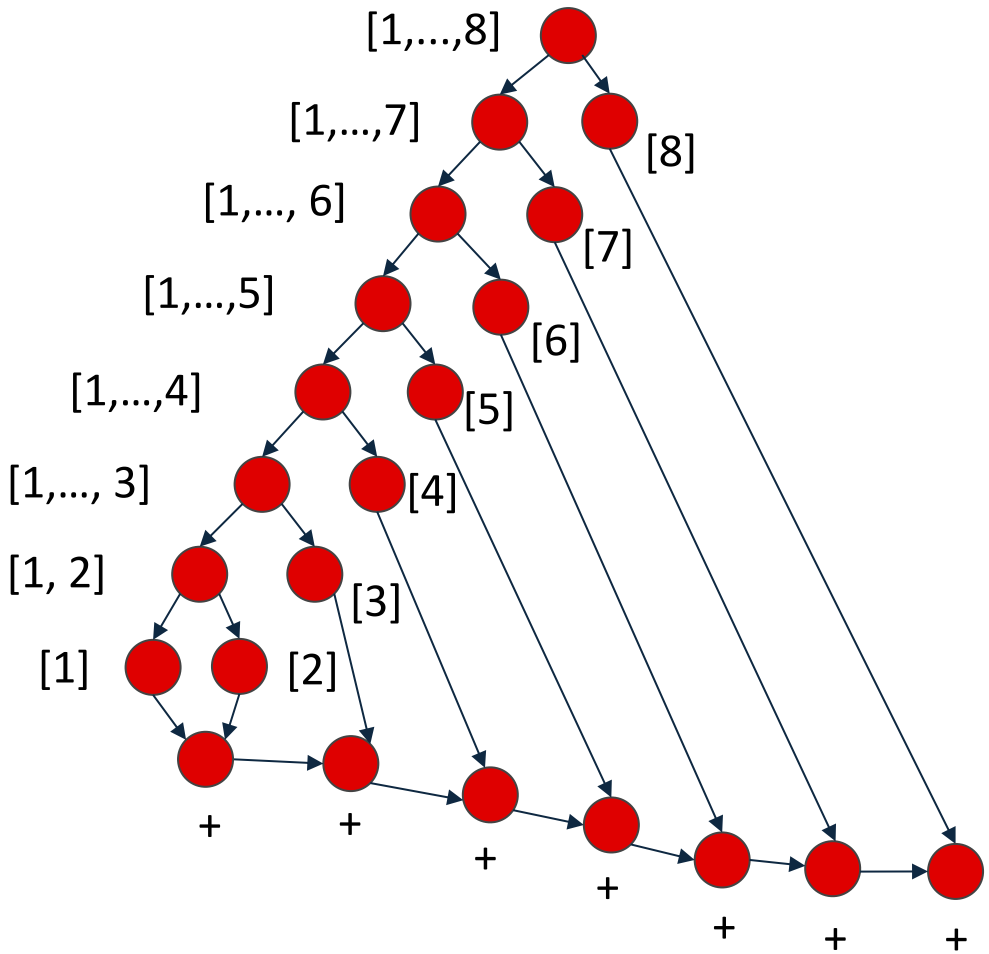
Task: fork, join, continuation

Cut-off: 1

Adding eight numbers:



What is the corresponding task graph?

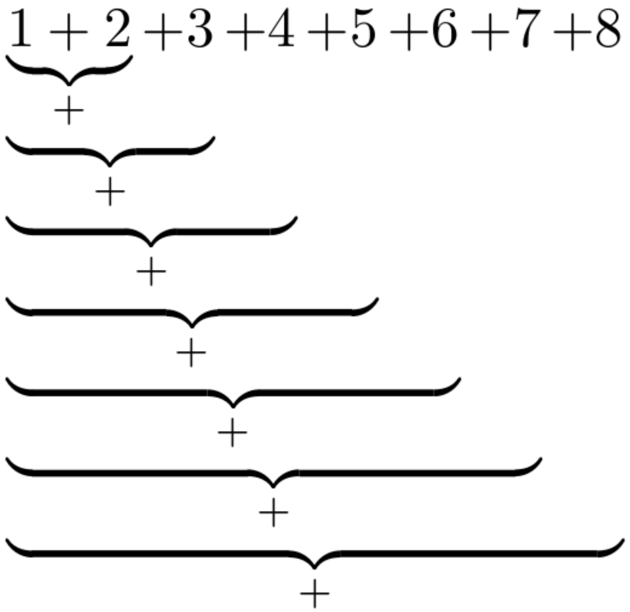


Task Graph FJ

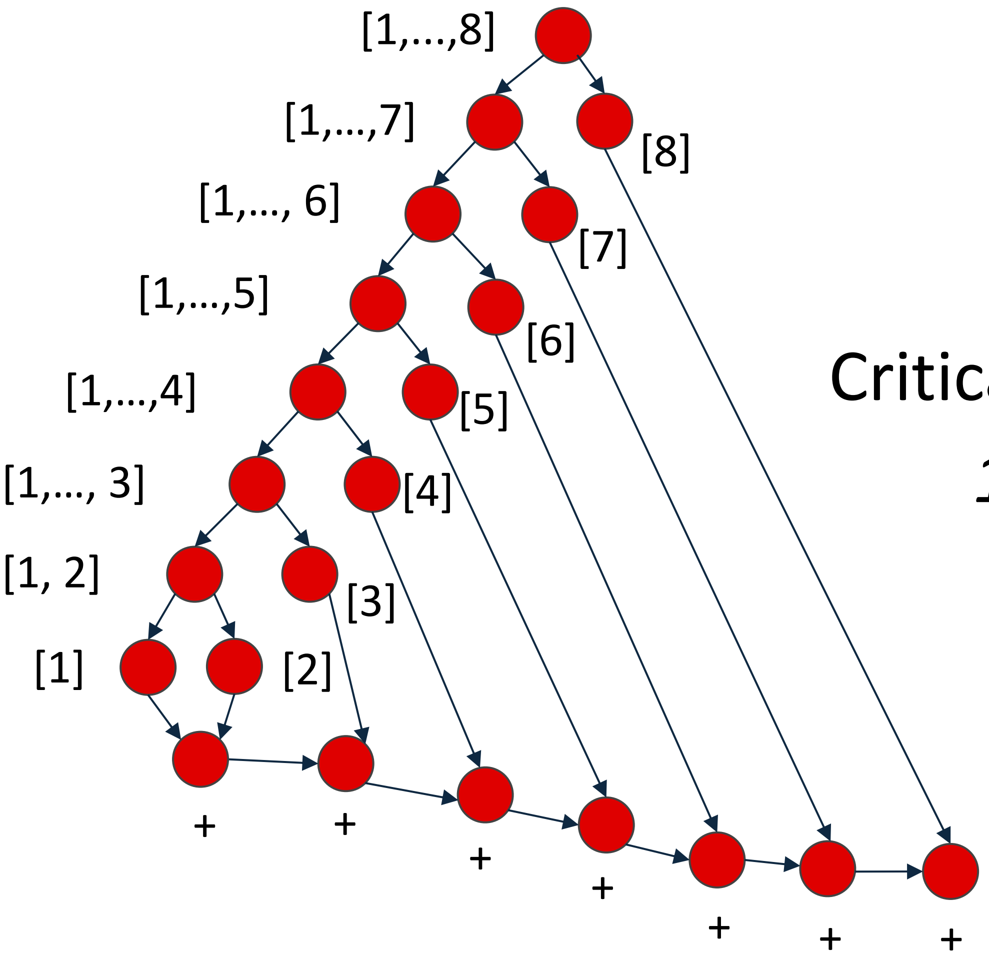
Task: fork, join, continuation

Cut-off: 1

Adding eight numbers:



What is the corresponding task graph?



Critical path

15

Task Graph Simplified

Task: Call to add()

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{1+2}_{+}+\underbrace{3+4}_{+}}_{+}+\underbrace{\underbrace{5+6}_{+}+\underbrace{7+8}_{+}}_{+}}_{+}$$

Task Graph Simplified

Task: Call to add()

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{1+2}_{+}+\underbrace{3+4}_{+}}_{+}+\underbrace{\underbrace{5+6}_{+}+\underbrace{7+8}_{+}}_{+}}_{+}$$

What is the corresponding task graph?

Task Graph Simplified

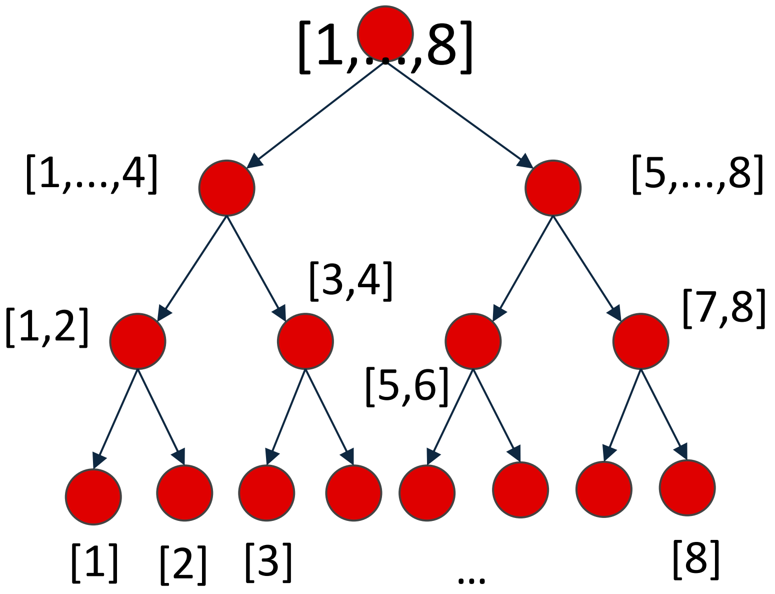
Task: Call to add()

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{1+2}_{+}+\underbrace{3+4}_{+}}_{+}+\underbrace{\underbrace{5+6}_{+}+\underbrace{7+8}_{+}}_{+}}_{+}$$

What is the corresponding task graph?



essential

Task Graph Simplified

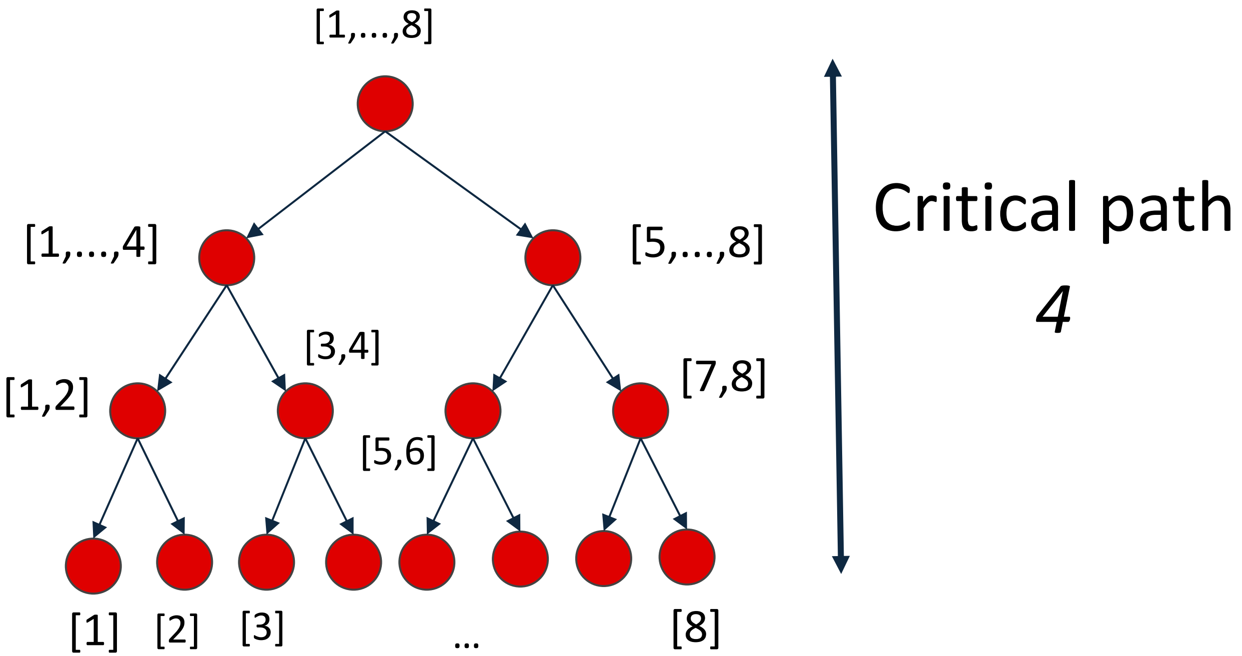
Task: Call to add()

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{1+2}_{+}+\underbrace{3+4}_{+}}_{+}+\underbrace{\underbrace{5+6}_{+}+\underbrace{7+8}_{+}}_{+}}_{+}$$

What is the corresponding task graph?



Task Graph FJ

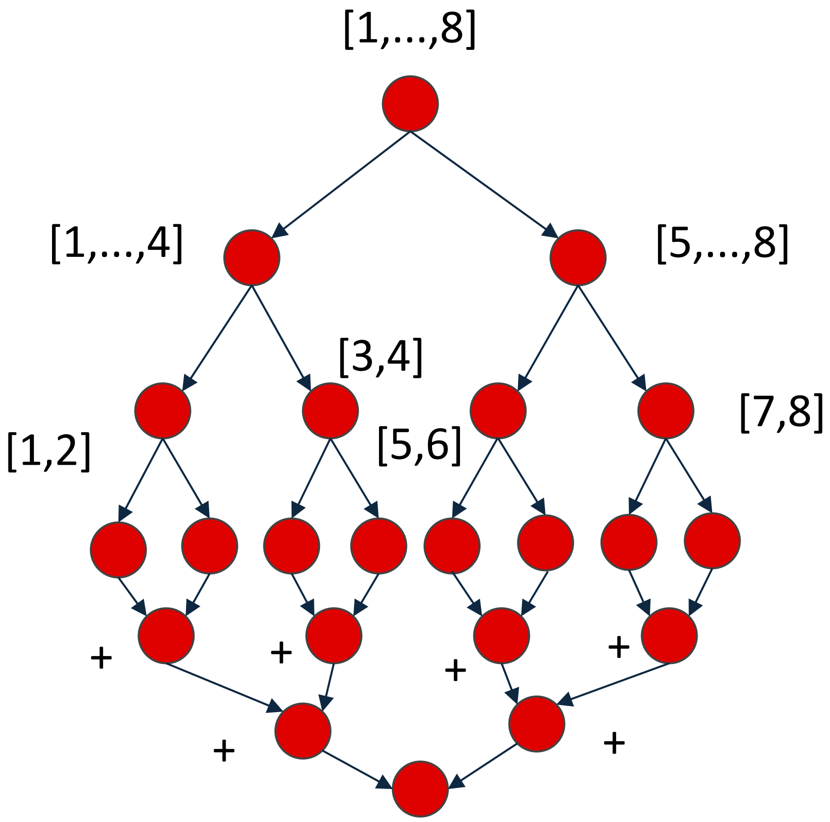
Task: fork, join, continuation

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{1+2}_{+}+\underbrace{3+4}_{+}}_{+}+\underbrace{\underbrace{5+6}_{+}+\underbrace{7+8}_{+}}_{+}}_{+}$$

What is the corresponding task graph?



Task Graph FJ

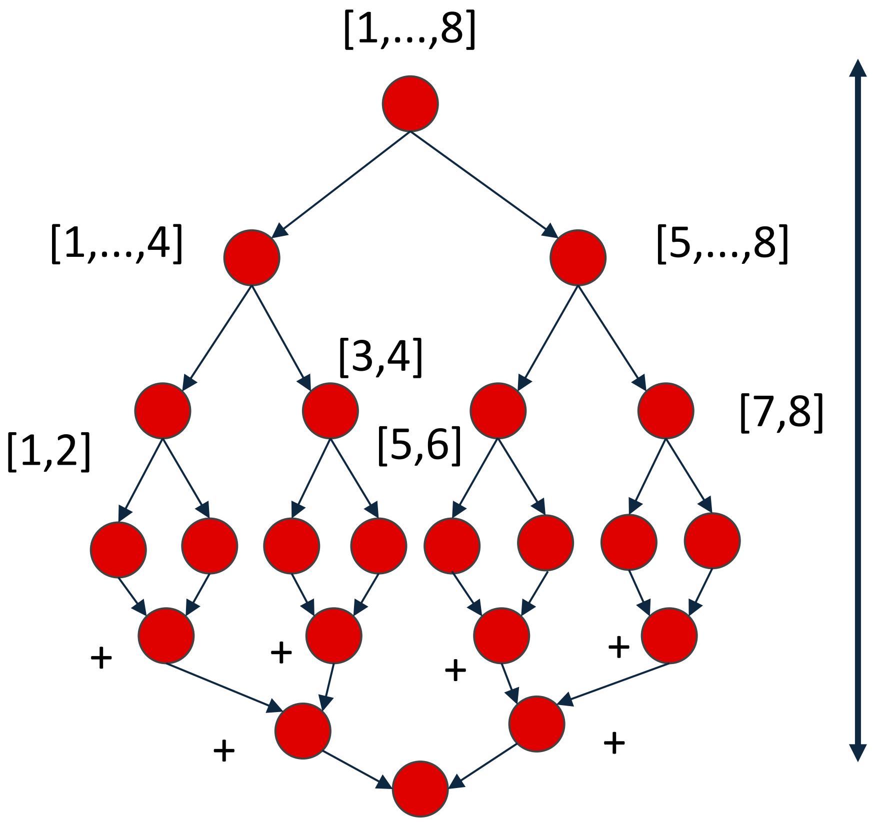
Task: fork, join, continuation

Cut-off: 1

Adding eight numbers:

$$\underbrace{\underbrace{\underbrace{1+2}_{+}+\underbrace{3+4}_{+}}_{+}+\underbrace{\underbrace{5+6}_{+}+\underbrace{7+8}_{+}}_{+}}_{+}$$

What is the corresponding task graph?



Critical path

7

Task Graphs

A wide task graph → higher potential parallelism

A deep task graph → more sequential dependencies

- There is a master solution , feel free to take a look if you had trouble with these theory task
- I’ll also upload my code solution

Coding Part

Task 1: 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.

Task 1 A: Search And Count - Sequential

```java
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;
  }
}
```

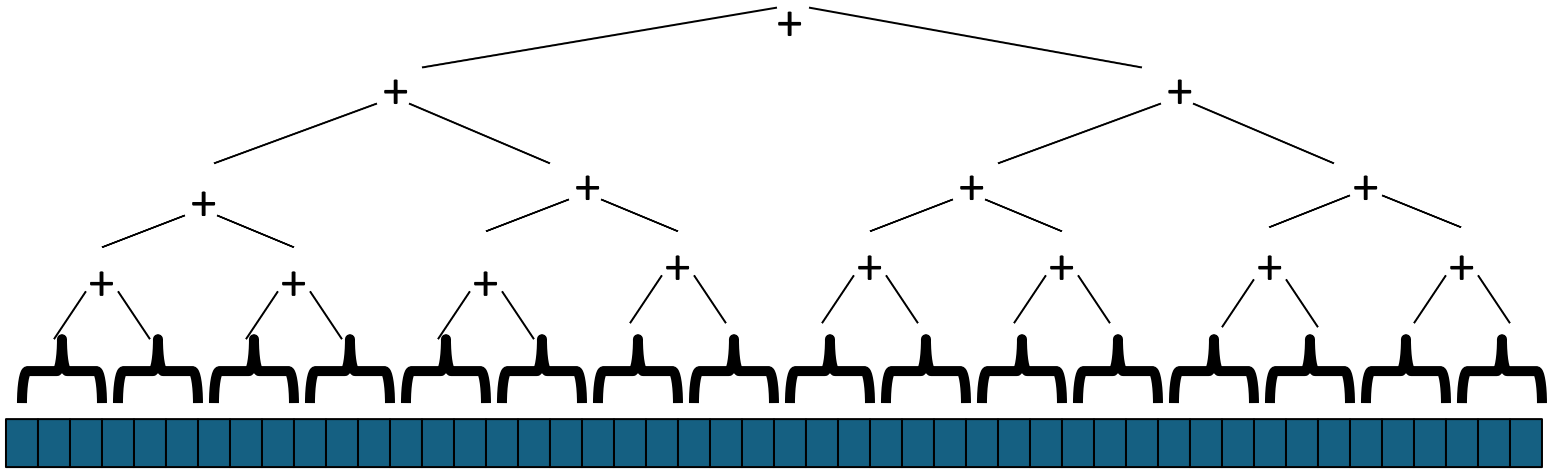
Straightforward implementation. Simply iterate through the input array and count how many times given event occurs.

Divide and Conquer

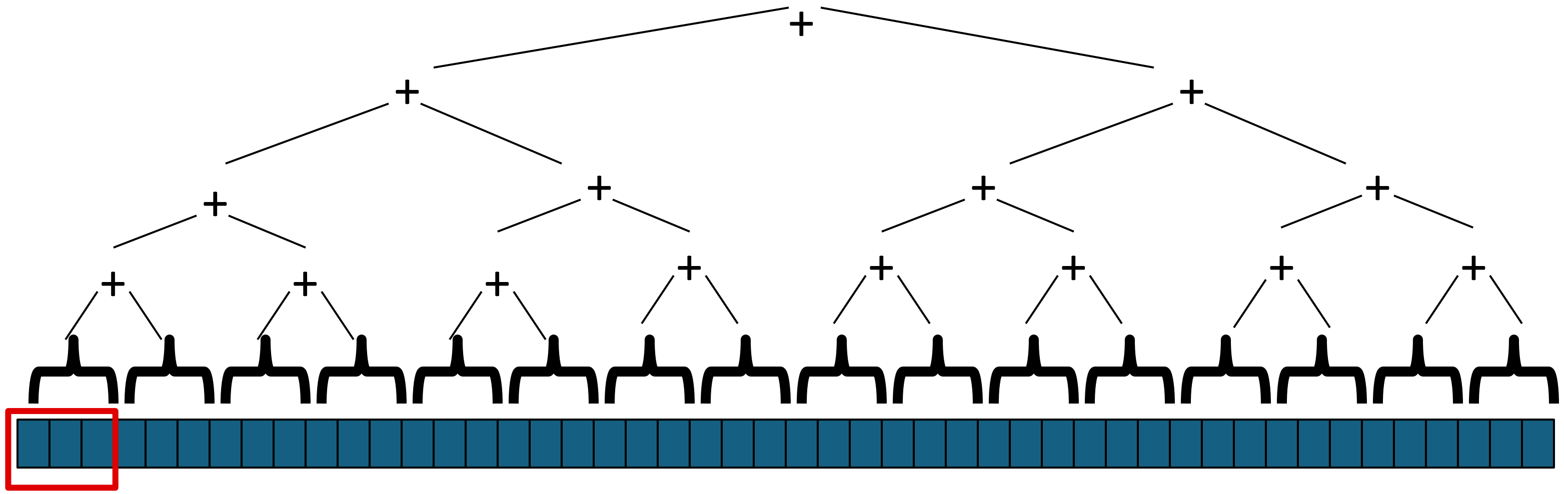
Basic structure of a divide-and-conquer algorithm:

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



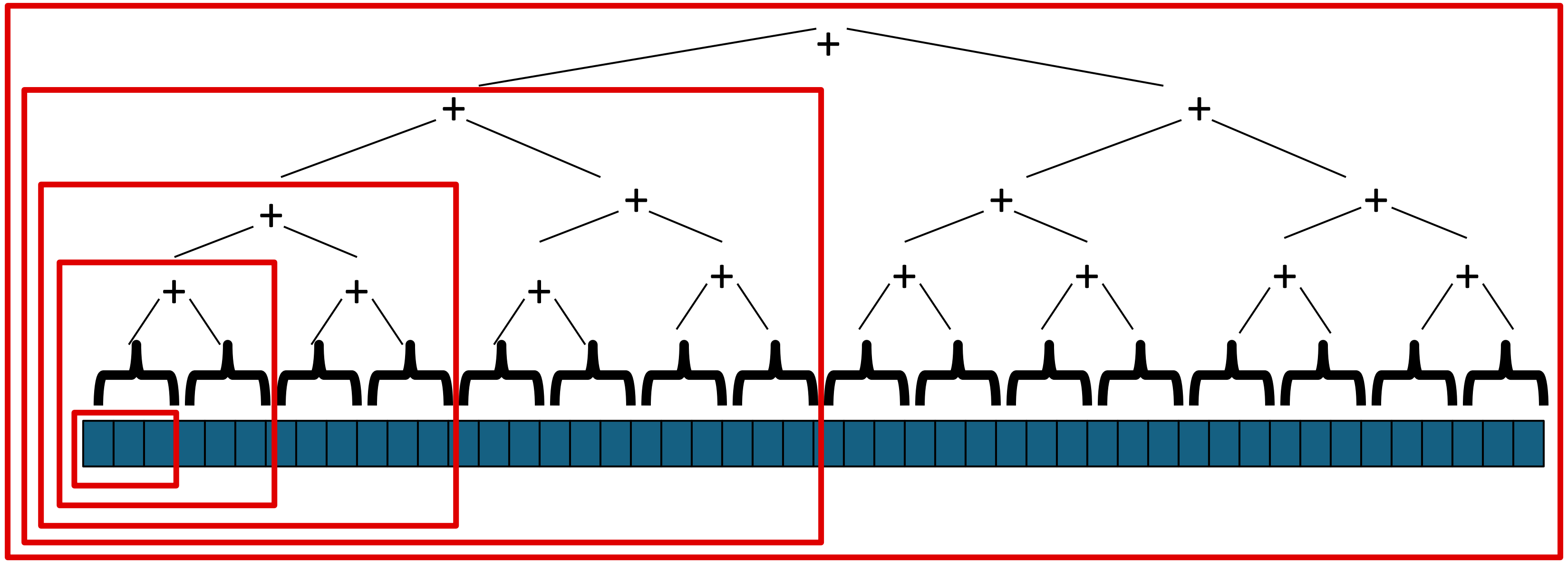
Divide and Conquer



base case
no further split

Divide and Conquer

Tasks at different
levels of granularity



What determines a task?

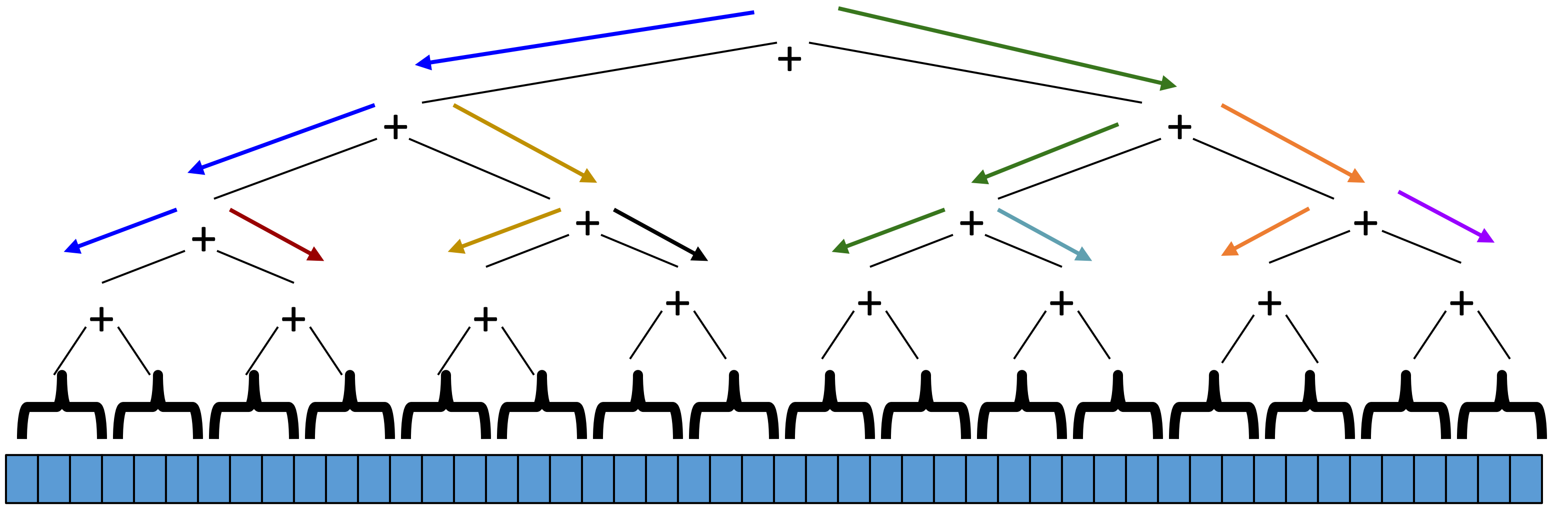
i) input array ii) start index iii) length/end index

These are fields we want to store in the task

Feedback: Tasks 1 B-D

Divide and Conquer Parallelization

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

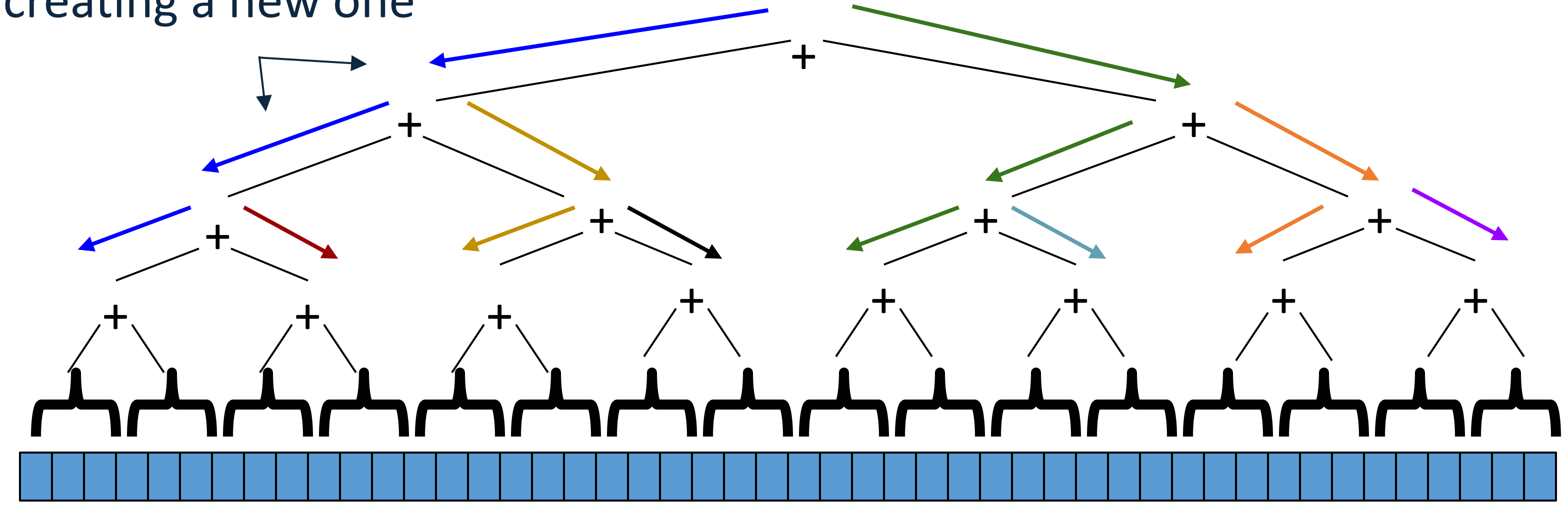


Divide and Conquer Parallelization

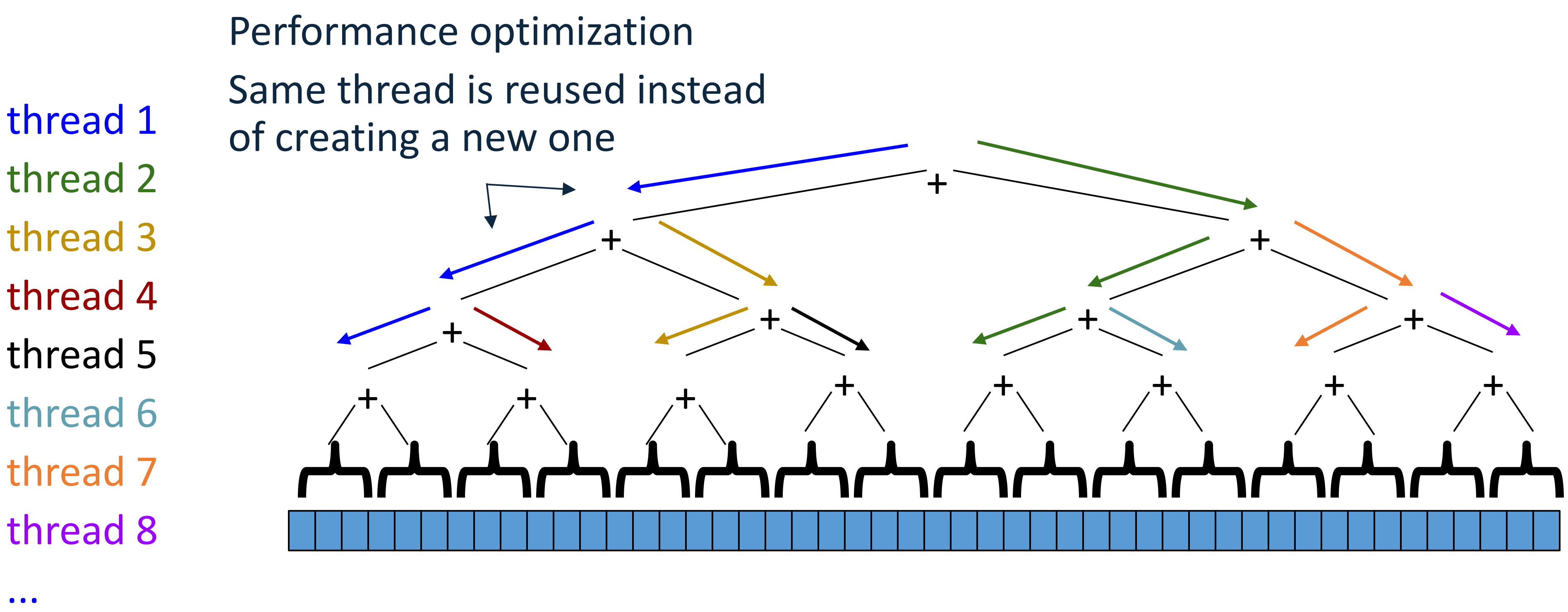
thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
thread 8
...

Performance optimization

Same thread is reused instead of creating a new one



Divide and Conquer Parallelization

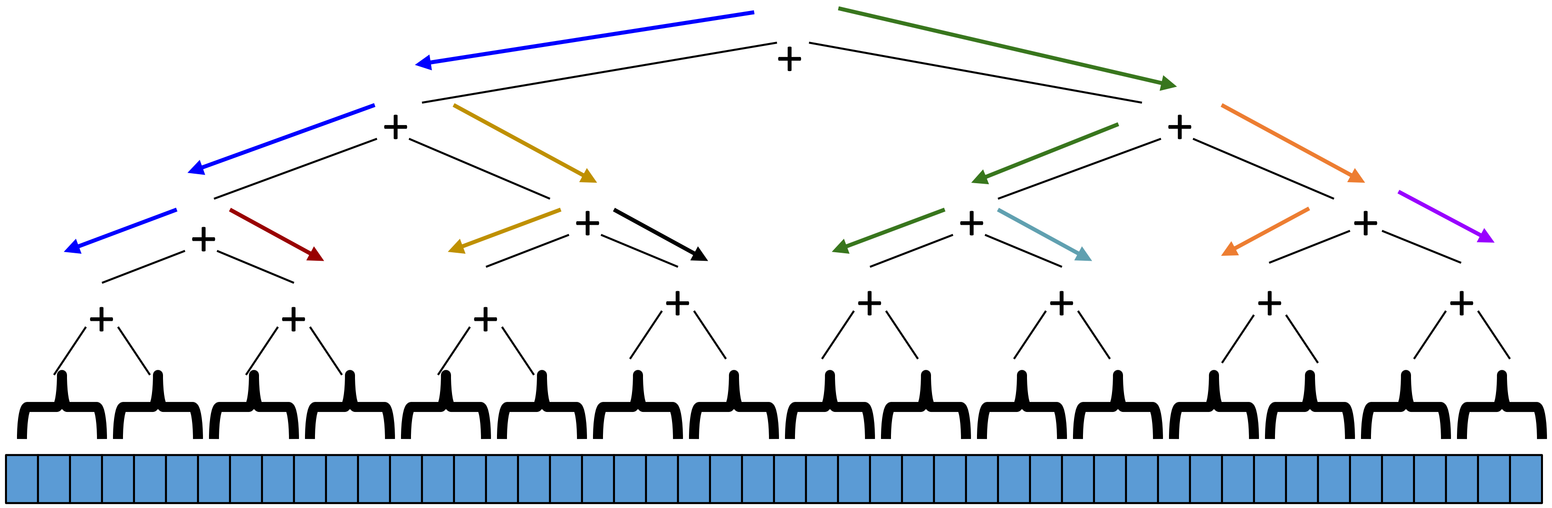


Task B:

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

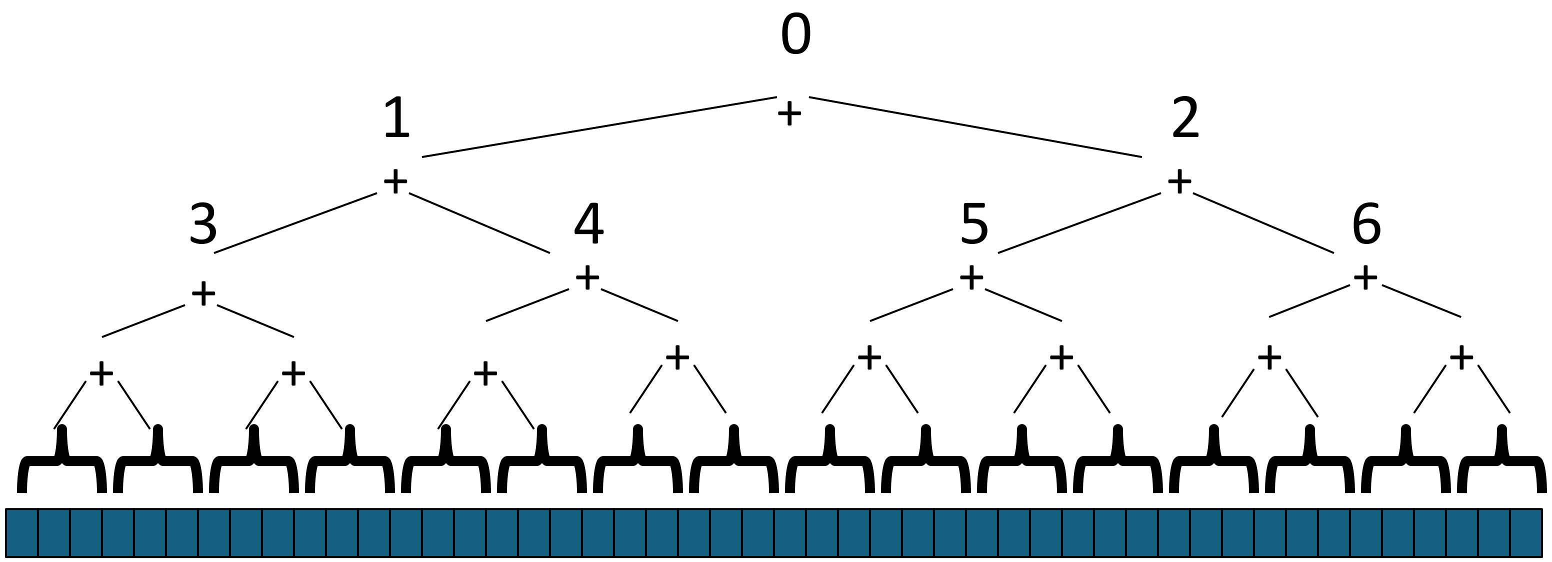
How to achieve this?

Divide and Conquer Parallelization



Option 1:
Shared counter with
synchronized/atomic access

Divide and Conquer Parallelization

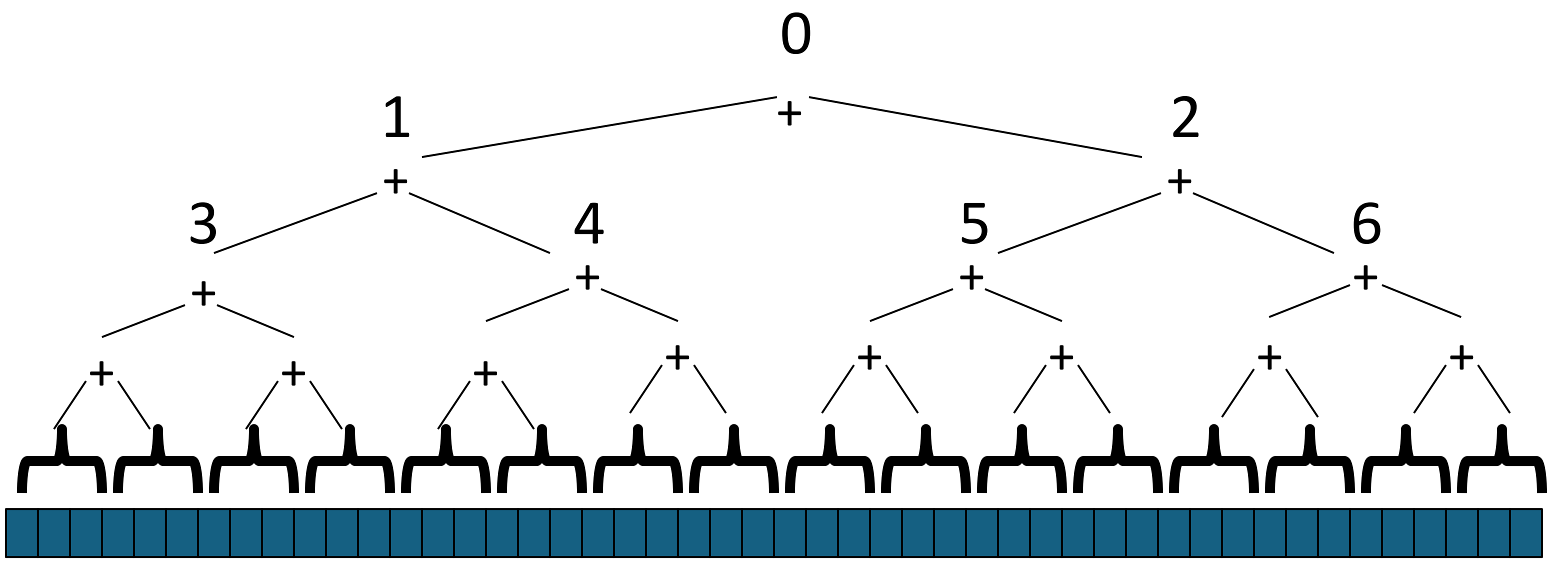


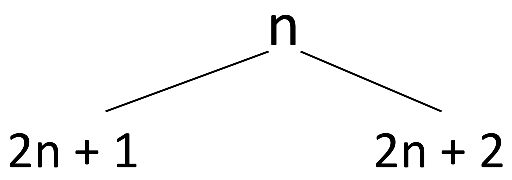
Option 1:
Shared counter with synchronized/atomic access

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

essential

Divide and Conquer Parallelization

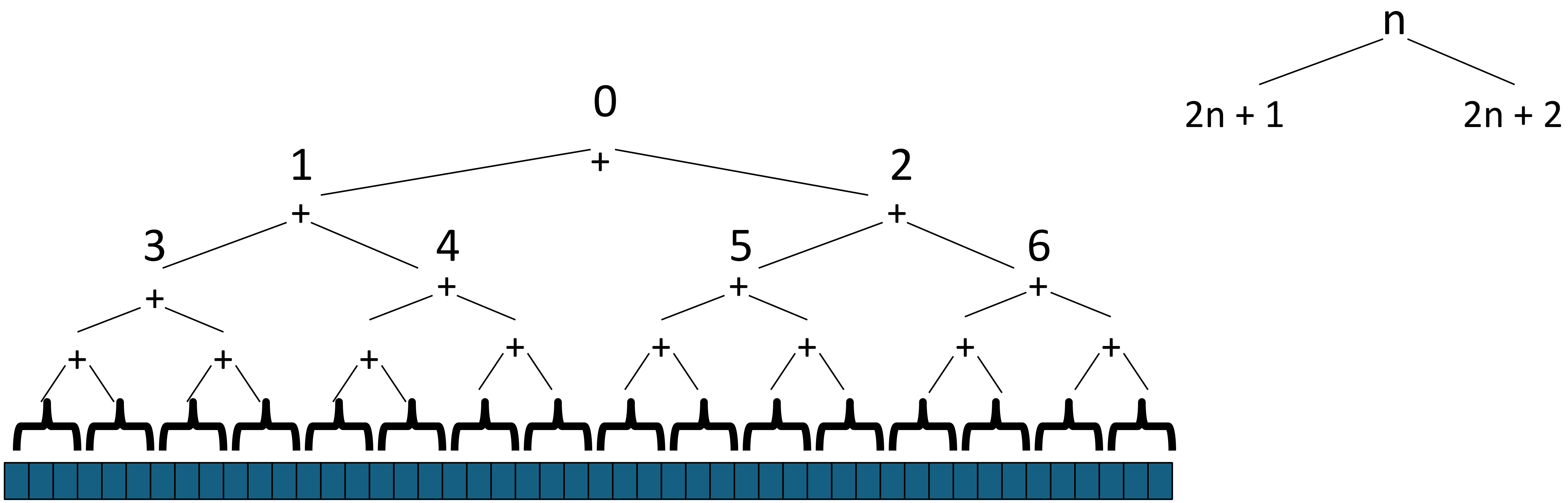




Option 1:
Shared counter with synchronized/atomic access

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

essential

Divide and Conquer vs Fork/Join

Divide And Conquer

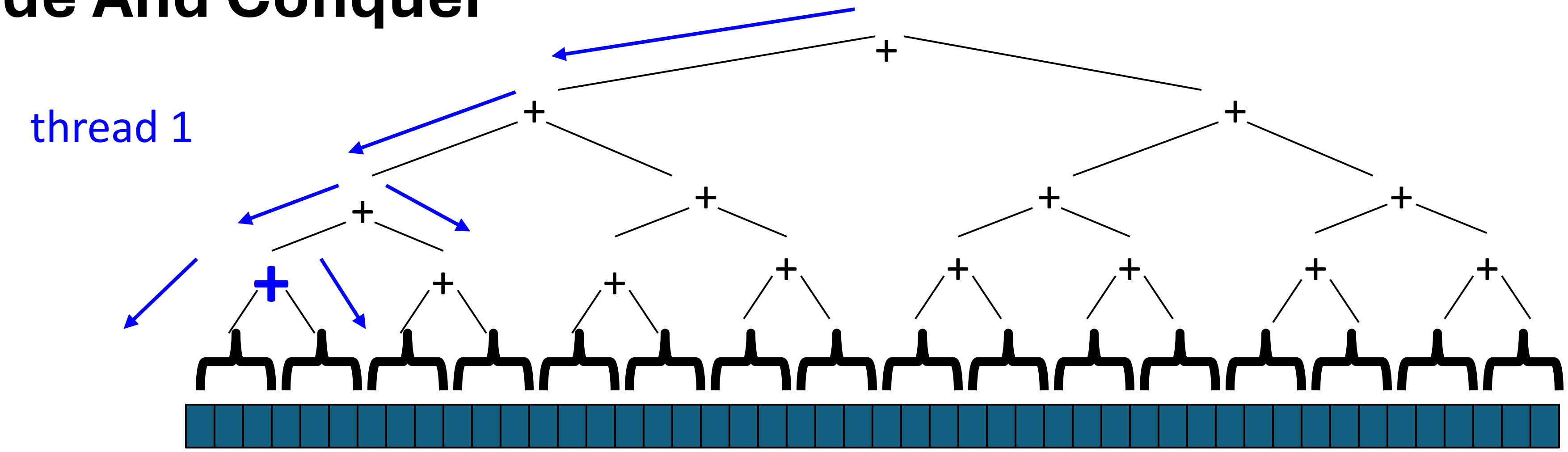
Fundamental design pattern based on recursively breaking down a problem into smaller problems that can be combined to give a solution to the original problem

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

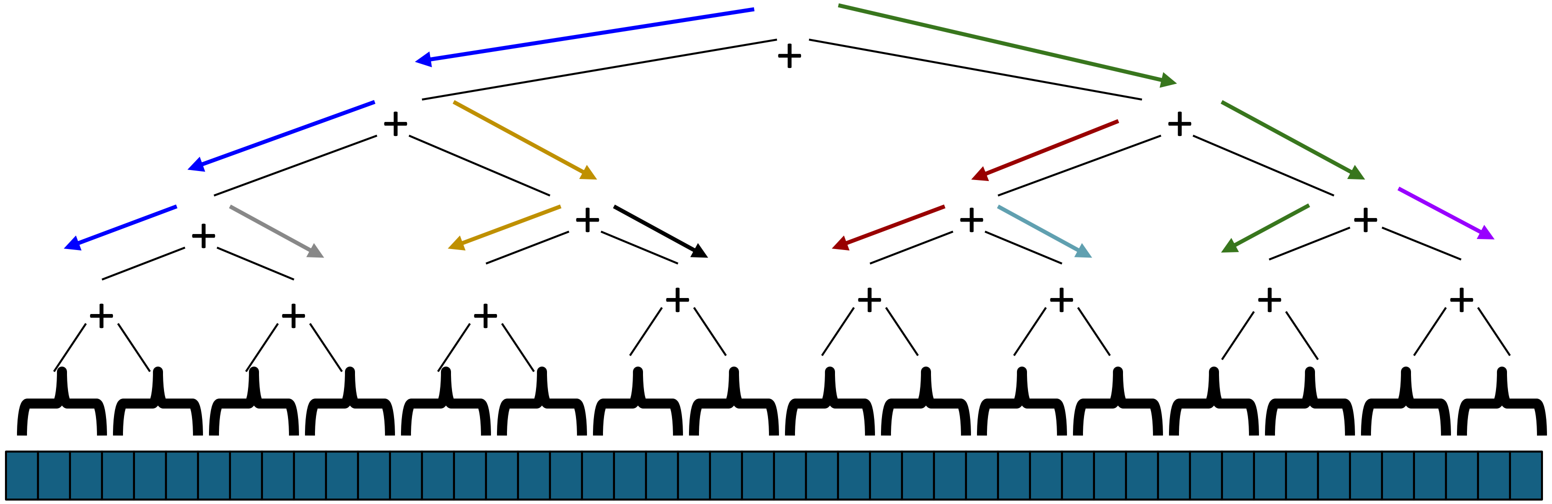
problems are solved sequentially

essential

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

essential

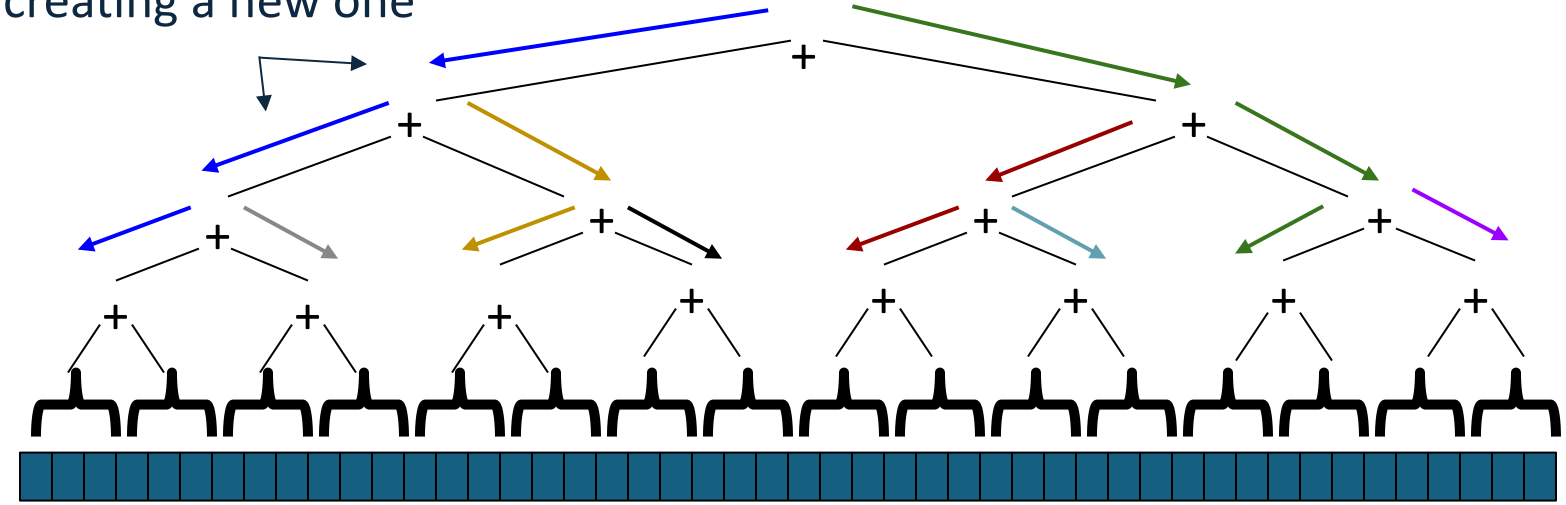
Divide and Conquer vs Fork/Join

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's fill in the template for the search and count task

Search And Count

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

```java
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
  }
}
```

```java
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
  }
}
```

Search And Count

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

```java
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
  }
}
```

```java
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
  }
}
```

essential

Search And Count

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

```
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
  }
}
```

essential

Search And Count

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

```java
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
  }
}
```

```java
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
  }
}
```

Same as sequential implementation

essential

```java
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

```java
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
  }
}
```

```java
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
  }
}
```

essential

Search And Count

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

```java
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
  }
}
```

```java
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
  }
}
```

essential

Search And Count

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

```java
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
  }
}
```

```java
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
  }
}
```

essential

Search And Count

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

```java
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
  }
}
```

```java
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();
    sc2.fork();
    int count1 = sc1.join();
    int count2 = sc2.join();
    // combine the results
  }
}
```

essential

```java
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

```java
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
  }
}
```

```java
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();
    sc2.fork();
    int count1 = sc1.join();
    int count2 = sc2.join();
    // combine the results
  }
}
```

essential

Search And Count

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

```java
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
  }
}
```

```java
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();
    sc2.fork();
    int count1 = sc1.join();
    int count2 = sc2.join();
    return count1 + count2;
  }
}
```

Search And Count

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

```java
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
  }
}
```

```java
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();
    sc2.fork();
    int count1 = sc1.join();
    int count2 = sc2.join();
    return count1 + count2;
  }
}
```

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].

Fork/Join: recommended for Divide and Conquer tasks as they have strong task interdependency

ExecutorService: for handling many independent requests where tasks are standalone

- See code

Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- **Theory Recap**
- Intro Exercise 6
- Exam Questions
- Kahoot

Summary

T_1: Time on one processor
T_P: Time on P processors
T_∞: Time on "infinite" processors ($\lim_{P\to\infty} T_P$)

$S_P = \frac{T_1}{T_P}$: Speedup generated using P processors

- **Amdahls Law** assumes a fixed workload in variable time it states:

$$S_P = \frac{T_1}{T_P} \leq \frac{1}{f + \frac{1-f}{P}}$$

- **Gustafsons Law** assumes a fixed time window, but measures the speedup in workload terms:

$$S_P \leq f + P(1 - f)$$

Lock Object

Shared object that satisfies the following interface

```
public interface Lock{
      public void lock();      // entering CS
      public void unlock();    // leaving CS
}
```

providing the following semantics

new Lock make a new lock, initially *"not held"*

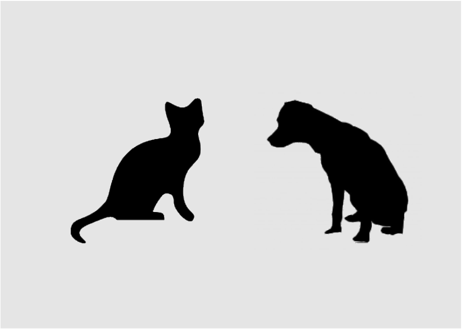
acquire blocks (only) if this lock is already currently *"held"*
Once *"not held"*, makes lock *"held"* [all at once!]

release makes this lock *"not held"*
If >= 1 threads are blocked on it, exactly 1 will acquire it

Required Properties of Mutual Exclusion

Safety Property

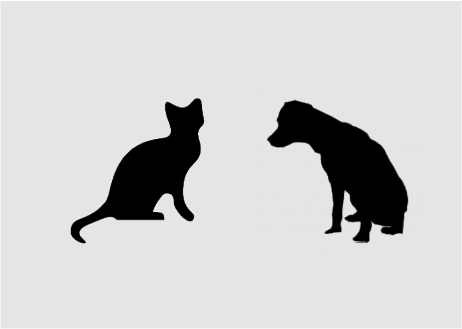
- At most one process executes the critical section code



Required Properties of Mutual Exclusion

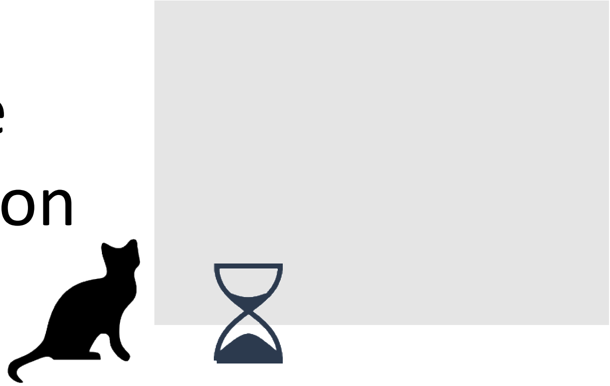
Safety Property

- At most one process executes the critical section code



Liveness

- *Minimally*: acquire_mutex must terminate in finite time when no process executes in the critical section



Almost-correct pseudocode

```
class BankAccount {
   private int balance = 0;
   private Lock lk = new Lock();
   …
   void withdraw(int amount) {
      lk.lock(); // may block
      int b = getBalance();
      if(amount > b)
         throw new WithdrawTooLargeException();
      setBalance(b – amount);
      lk.unlock();
   }
   // deposit would also acquire/release lk
}
```

One lock for each account

Almost-correct pseudocode

```
class BankAccount {
   private int balance = 0;
   private Lock lk = new Lock();
   …
   void withdraw(int amount) {
      lk.lock(); // may block
      int b = getBalance();
      if(amount > b)
         throw new WithdrawTooLargeException();
      setBalance(b – amount);
      lk.unlock();
   }
   // deposit would also acquire/release lk
}
```

One lock for each account

Lock won't be released if exception is thrown!

Solution: Use try/finally block!

```java
Lock lk = new ReentrantLock();

public static long criticalWork() {
  lk.lock();
  try {
    //do some work
    return result;
  } finally {
    lk.release();
  }
}
```

Always gets executed
(even after exception
or return)

Possible mistakes

Incorrect: Use different locks for **`withdraw`** and **`deposit`**

- Mutual exclusion works only when using same lock
- **`balance`** field is the shared resource being protected

Poor performance: Use same lock for every bank account

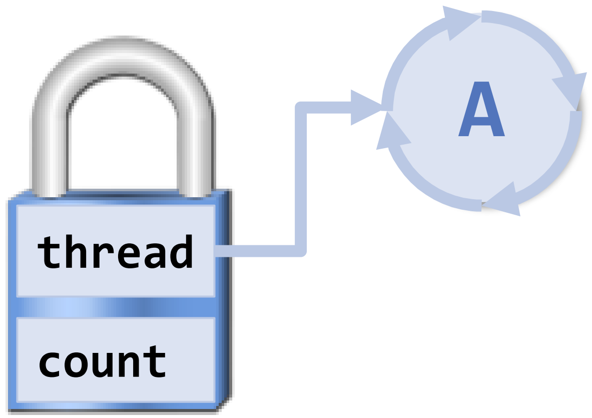
- No simultaneous operations on different accounts

Incorrect: Forget to release a lock (blocks other threads forever!)

- Previous slide is wrong because of the exception possibility!

```
if(amount > b) {
  lk.unlock(); // hard to remember!
  throw new WithdrawTooLargeException();
}
```

Re-entrant lock



A **re-entrant lock** (a.k.a. **recursive lock**)

"remembers"

- the thread (if any) that currently holds it
- a *count*

When the lock goes from *not-held* to *held*, the count is set to 0

If (code running in) the current holder calls **`lock(acquire)`**:

- it does not block
- it increments the count

On **`unlock(release)`**:

- if the count is > 0, the count is decremented
- if the count is 0, the lock becomes *not-held*

Re-entrant locks work

- This simple code works fine provided **`lk`** is a reentrant lock
- Okay to call **`setBalance`** directly
- Okay to call **`withdraw`** (won't block forever)

```
int setBalance(int x) {
  lk.lock();
  balance = x;
  lk.unlock();
}

void withdraw(int amount) {
  lk.lock();
  …
  setBalance(b – amount);
  lk.unlock();
}
```

Waiting for lock

lock()

- Acquires the lock and **blocks indefinitely** until it is available

tryLock()

- Attempts to acquire the lock **without blocking**
- Returns true if successful, false if the lock is already held
- Supports timeouts

```
if (lk.tryLock(100, TimeUnit.MILLISECONDS)) { // Wait up to 100ms
    try {
        // Critical section
    } finally {
        lk.unlock();
    }
} else {
    // Lock not acquired, handle accordingly
}
```

Avoiding long waits or deadlocks, implementing fallback strategies

Race condition

A **Race Condition** occurs in concurrent programming when the correctness of the system depends on the specific interleaving or ordering of operations executed by multiple threads or processes.

Typically, problem is some *intermediate state* that "messes up" a concurrent thread that "sees" that state

Note: This lecture makes a big distinction between *data races* and *bad interleavings*, both instances of race-condition bugs

- Confusion often results from not distinguishing these or using the ambiguous "race condition" to mean only one

The distinction

Data Race [aka *Low Level Race Condition, low semantic level*]
Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. Simultaneous read/write or write/write of the same memory location

(for mortals) **always** an error, due to compiler & HW

Bad Interleaving [aka *High Level Race Condition, high semantic level*]
Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources.

"Bad" depends on your specification

On low- and high-level data races

Shared data **balance**, access protected by synchronized

Forgot **synchronized** in **withdraw**:

- **withdraw** accesses **balance** only under lock (via **setBalance** / **getBalance**)
- No concurrent read / write or write / write accesses of **balance**
  → No low-level data race
- Two **withdraw** operations can be interleaved – if this is a problem depends on the specification of our bank account
- We might get unwanted interleavings (intermediate states that should not be observed / violating invariants)
  → We can still have a high-level data race

```
public synchronized void setBalance(int x) {
..  }

public synchronized int getBalance() { ..  }

public ~~synchronized~~ void withdraw(int amount)
{
    ..
    b = getBalance()
    ..
    setBalance(b - amount);
    ..
}
public synchronized void deposit(int amount){
    ..
    b = getBalance()
    ..
    setBalance(b + amount);
    ..
}
```

Why Locks?

- See code examples

Why Locks?

- example models a bank where multiple threads transfer money between accounts
- Problem:
  - Locking Order Issue: Each transfer call synchronizes first on this and then on target
  - If two threads try to transfer in opposite directions at the same time, they will deadlock
  - e.g. if Thread-1 locks accountA and waits for accountB, while Thread-2 locks accountB and waits for accountA, both will be stuck indefinitely

Why Locks?

- To prevent deadlocks, it ensures that locks are always acquired in a consistent order
- Prevent deadlock by introducing order with System.identityHashCode() to determine which account should be locked first (or any kind of unique ID)

Parallel Patterns

Credits to Erxuan Li PProg FS25

The prefix-sum problem

Given **`int[] input`**,

produce **`int[] output`** where:

`output[i]` = **`input[0]+input[1]+…+input[i]`**

Sequential prefix-sum

```
int[] prefix_sum(int[] input){
    int[] output = new int[input.length];
    output[0] = input[0];

    for(int i = 1; i < input.length; i++)
        output[i] = output[i-1] + input[i];

    return output;
}
```

Does not seem parallelizable

- Work: O(n), Span: O(n)
- This algorithm is sequential, but a **different algorithm** has: Work O(n), Span O(`log` n)

Parallel prefix-sum

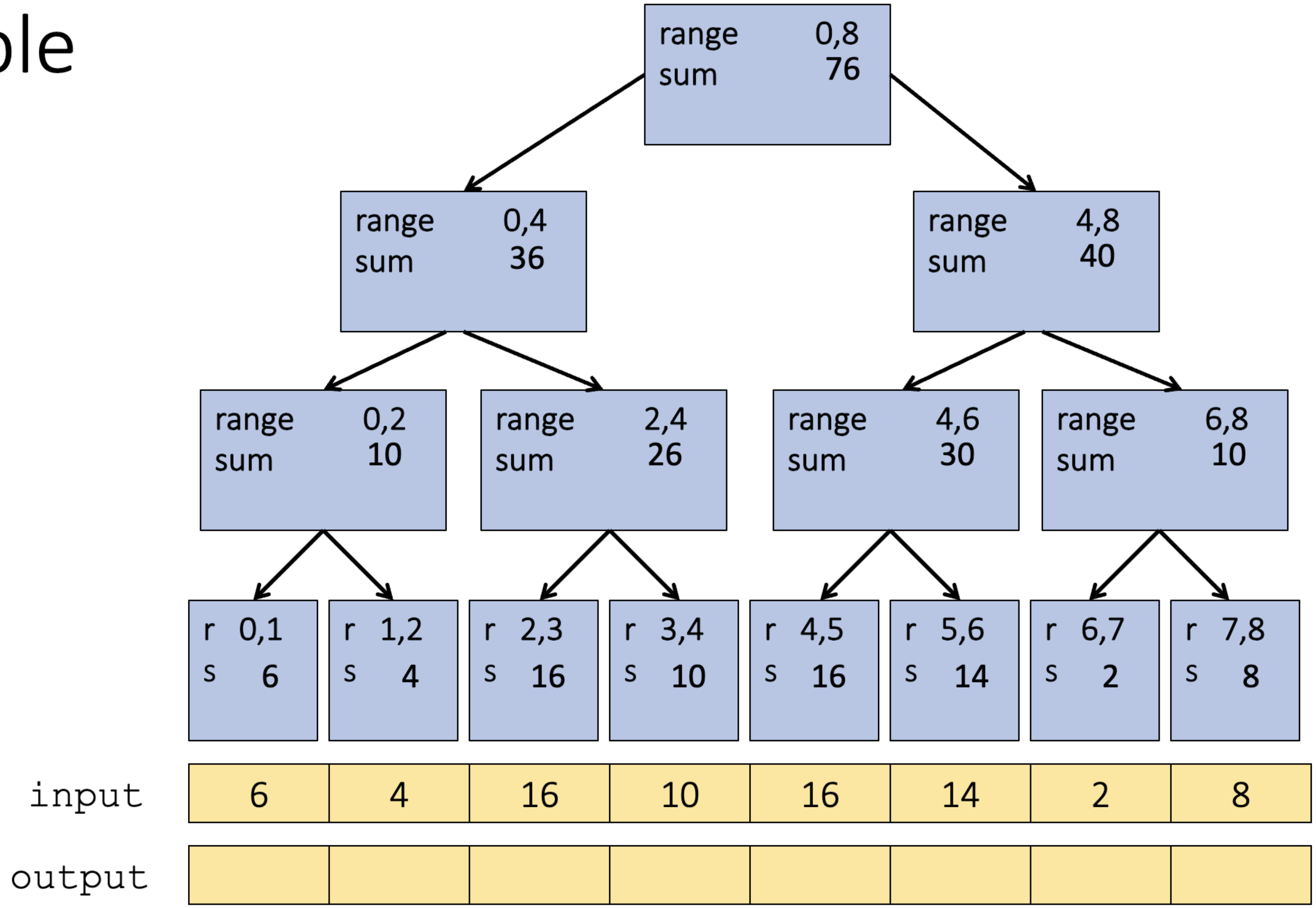
The parallel-prefix algorithm does two passes

- Each pass has O(n) work and O(`log` n) span
- So in total there is O(n) work and O(`log` n) span
- So like with array summing, parallelism is n/`log` n

First pass builds a tree bottom-up: the “up” pass

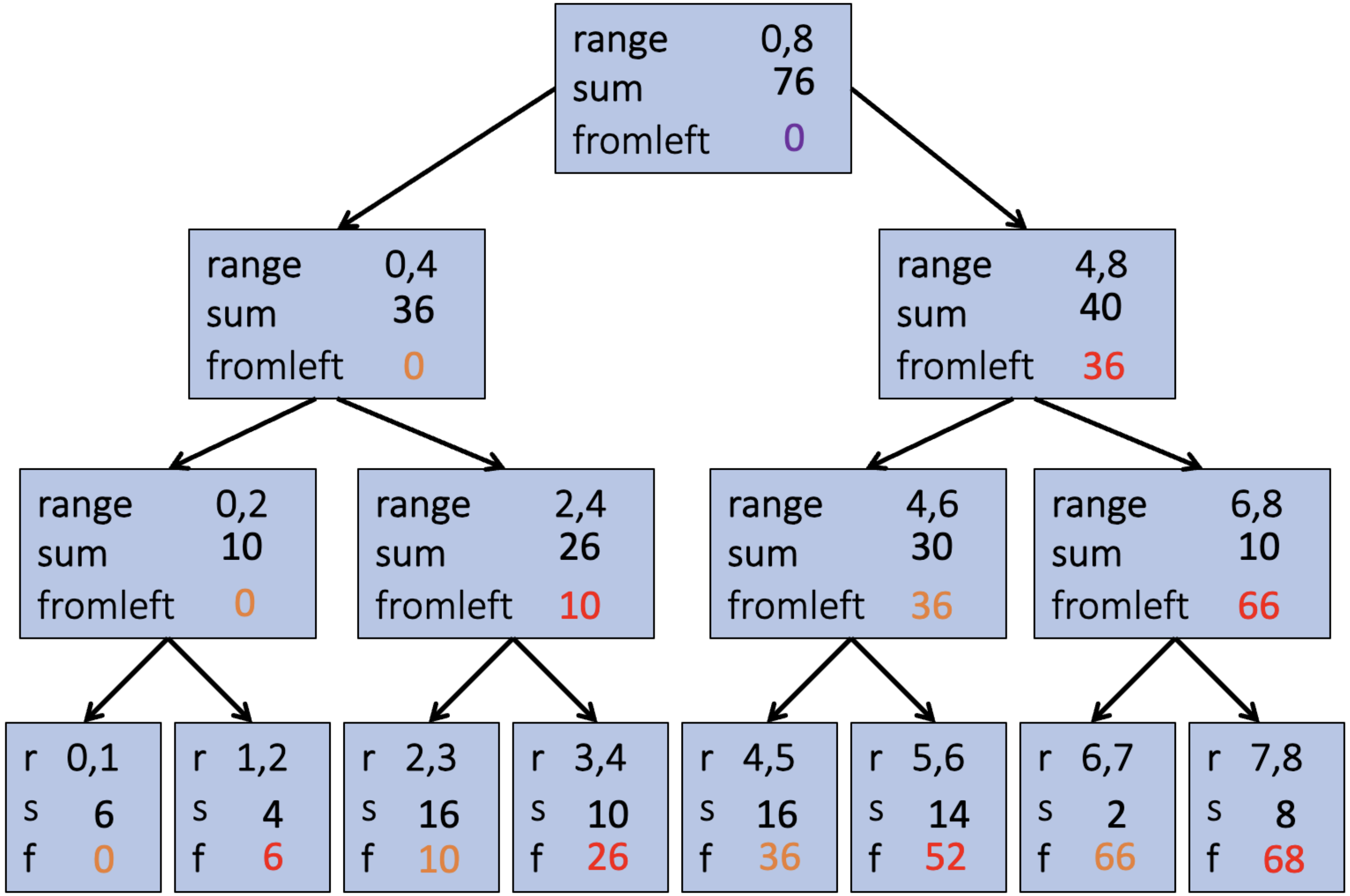
Second pass traverses the tree top-down: the “down” pass

Example



range 0,8
sum 76
range 0,4
sum 36
range 4,8
sum 40
range 0,2
sum 10
range 2,4
sum 26
range 4,6
sum 30
range 6,8
sum 10
r 0,1
s 6
r 1,2
s 4
r 2,3
s 16
r 3,4
s 10
r 4,5
s 16
r 5,6
s 14
r 6,7
s 2
r 7,8
s 8
input
6
4
16
10
16
14
2
8
output

Example



input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

Parallel Patterns

- We are now quite familiar with how to parallize algorithms
- There are a few recurring patterns that are important to know

Map, Reduction, Stencil, Scan, Pack

Reduction

- A reduction is an operation that produces a single answer from a collection (array etc) via an **associative** operator.
- Needs to be associative. Otherwise divide-and-conquer won‘t work

Example: array sum

Map

- Operates on each element of the input data indenpendently (each array element)
- Output is the same size → no size reduction
- Doesn‘t have to be the same operation on each element

Example: add two arrays

Stencil

- Like map but can take more than one element as input
- Generalization of map and thus also no size reduction

Example:

Image → apply averaging filter on each pixel

Update a value based on its neighbors

Never do it in-place because you would then take values that are already output values.

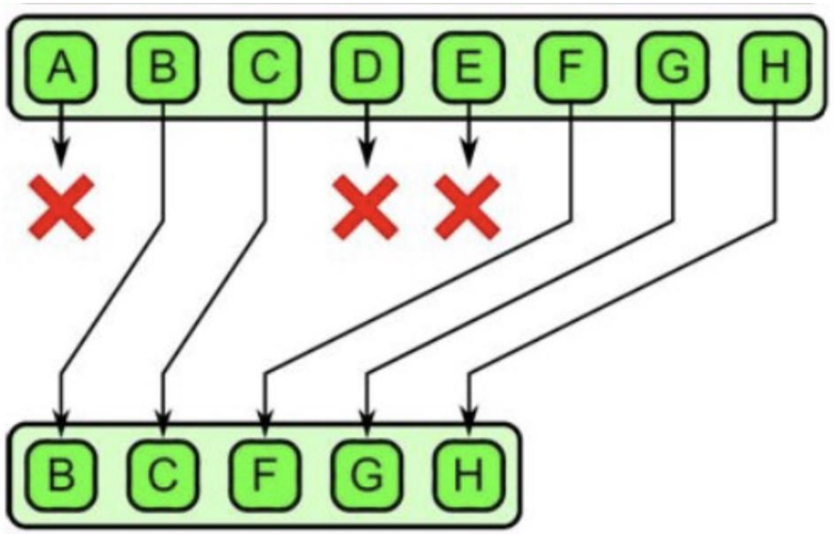
Scan

- Collection of data X → return collection of data Y
- Y(i) = functionOf(Y(i - 1) & X(i))
- Seems sequential because of dependencies
- Can parallelize if function is associative → O(log(n)) span

Example: parallel prefix sum

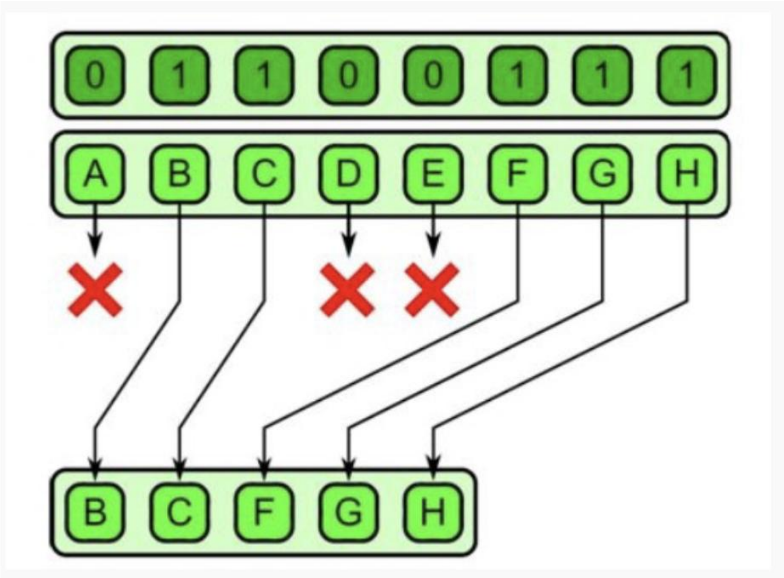
Pack

- Collection of data X → return collection of data X if fulfill condition



Pack

- First compute bit vector
- Then find index in result array (prefix sum on bit vector)



Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- Theory Recap
- **Intro Exercise 6**
- Exam Questions
- Kahoot

Assignment 6

Task Parallelism:

- Merge Sort
- Longest Sequence

essential

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:

[1, 2, 3, 3, 4, 1]

[1, 2, 3]　　　　[3, 4, 1]

Longest Sequence

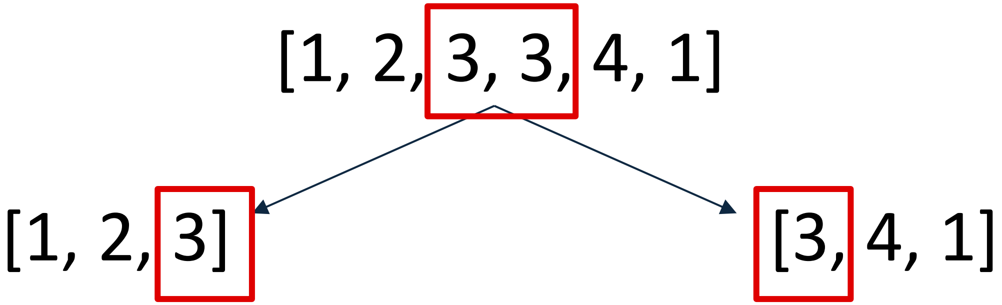
Task:

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!



Hint

- ExecutorService ex = Executors.newWorkStealingPool();

Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- Theory Recap
- Intro Exercise 6
- **Exam Questions**
- Kahoot

```java
public class Main {
  public static Thread CreateThread(int start) {
    return new Thread(new Runnable() {

      @Override
      public void run() {
        for (int i = start; i < 7; i+=2) {
            System.out.println("Number " + i);
        }
      }
    });
  }

  public static void main(String[] args) throws InterruptedException {
    CreateThread(1).start();
    CreateThread(2).start();
  }
}
```

Markieren Sie alle Ausgaben, welche durch den Codeausschnitt ausgegeben werden können.

Mark all the print sequences that can be produced by running the program shown above.

☐

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
```

☐

```
Number 1
Number 6
Number 3
Number 4
Number 5
Number 2
```

☐

```
Number 6
Number 5
Number 4
Number 3
Number 2
Number 1
```

☐

```
Number 2
Number 4
Number 6
Number 1
Number 3
Number 5
```

```java
public class Main {
  public static Thread CreateThread(int start) {
    return new Thread(new Runnable() {

      @Override
      public void run() {
        for (int i = start; i < 7; i+=2) {
            System.out.println("Number " + i);
        }
      }
    });
  }

  public static void main(String[] args) throws InterruptedException {
    CreateThread(1).start();
    CreateThread(2).start();
  }
}
```

Markieren Sie alle Ausgaben, welche durch den Codeausschnitt ausgegeben werden können.

Mark all the print sequences that can be produced by running the program shown above.

☐

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
```

☐

```
Number 1
Number 6
Number 3
Number 4
Number 5
Number 2
```

☐

```
Number 6
Number 5
Number 4
Number 3
Number 2
Number 1
```

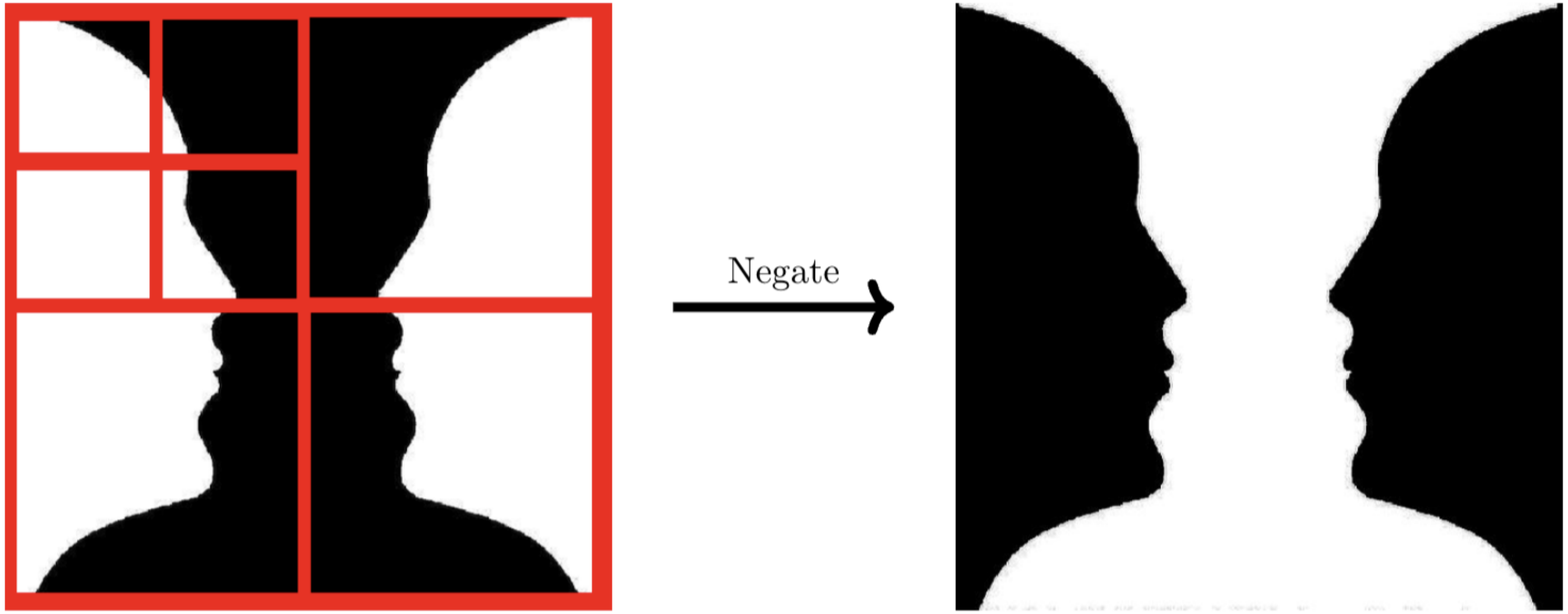
☐

```
Number 2
Number 4
Number 6
Number 1
Number 3
Number 5
```

Fork/Join Framework (16 points)

3. Der folgende Code zielt darauf ab, ein Bild zu negieren, indem es mithilfe des Fork/Join-Frameworks rekursiv in mehrere Unterfenster (vier pro Rekursionsschritt) unterteilt wird. Die Unterfenster können dann parallel negiert werden. Das folgende Beispiel verdeutlicht die Unterteilung des Bildes und die Negierung der einzelnen Unterfenster.

   The following code aims to negate an image by recursively subdividing it into multiple subwindows (four per recursion step) using the Fork/Join framework. The subwindows can then be negated in parallel. The example below illustrates the subdivision of the image and negation of the individual subwindows.



Bitte lesen Sie den Code sorgfältig durch und beantworten Sie dann die Fragen zum Code:

Please read the code carefully and then answer the questions regarding the code:

```java
public class ImageNegationFJ extends RecursiveAction {
    final static int CUTOFF = 32;
    double[][] image, invertedImage;
    int startx, starty;
    int length;

    public ImageNegationFJ(double[][] image, double[][] invertedImage,
            int startx, int starty, int length) {
        this.image = image;
        this.invertedImage = invertedImage;
        this.startx = startx;
        this.starty = starty;
        this.length = length;
    }

    @Override
    protected void compute() {
```

```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```java
final static int CUTOFF = 32;
```

```java
double[][] image, invertedImage;
```

```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

(a) Welche Annahme trifft der Code bezüglich der Abmessungen des Arrays, das das Eingabebild darstellt?

What assumption does the code make concerning the dimensions of the array representing the input image? (2)

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

13

The image should be square $s \times s$ and we should have $s = d2^k$, where $d \leq 32$. This is necessary, because we want `length` to be divisible by 2 in the case `length > 32`. If this would not be the case, we would do floor division and leave pixels unprocessed.

+ Add Comment ··· More

```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

(b) Parallelisiert der Code die beabsichtigte Aufgabe korrekt oder gibt es weitere Optimierungsmöglichkeiten? Wenn ja, welche Optimierung würden Sie vorschlagen und warum?

Does the code correctly parallelize the intended task or is there further optimization that could be done? If so, which optimization would you propose and why? (4)

Not good:

```
upperLeft.fork();
upperLeft.join();
upperRight.fork();
upperRight.join();
lowerLeft.fork();
lowerLeft.join();
lowerRight.compute();
```

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 7 months ago

8

No, the parallelization is incorrect, as we have subsequent `fork()` and `join()` calls, which means that we wait for the corresponding subproblem to be finished, before calling `fork()` on the next one. To fix this, we should do the following:

```
upperLeft.fork();
upperRight.fork();
lowerLeft.fork();
lowerRight.compute();
upperLeft.join();
upperRight.join();
lowerLeft.join();
```

```java
public class ImageNegationFJ extends RecursiveAction {
    final static int CUTOFF = 32;
    double[][] image, invertedImage;
    int startx, starty;
    int length;

    public ImageNegationFJ(double[][] image, double[][] invertedImage,
            int startx, int starty, int length) {
        this.image = image;
        this.invertedImage = invertedImage;
        this.startx = startx;
        this.starty = starty;
        this.length = length;
    }

    @Override
    protected void compute() {
```

(c) Vervollständigen Sie das folgende Codegerüst, indem Sie die oben implementierte `ImageNegationFJ` Klasse und die `ForkJoinPool` Klasse verwenden, um die Variable `negatedImage` mit den negierten Werten zu füllen.

Complete the following code skeleton by using the above implemented `ImageNegationFJ` class and the `ForkJoinPool` class to fill the variable `negatedImage` with the negated values. (4)

```java
double[][] image = {{0, 1}, {1, 0}};
int imageSize = image.length;
double[][] negatedImage = new double[imageSize][imageSize];
```

....

....

....

....

....

....

Tobias Steinbrecher @tsteinbreche · 8 months ago

10

```
double[][] image = {{0,1}, {1,0}};
int imageSize = image.length;
double[][] negatedImage = new double[imageSize][imageSize];
ForkJoinPool fjp = new ForkJoinPool();
ForkJoinTask t = new ImageNegationFJ(image, negatedImage, 0, 0 ,imageSize);
fjp.invoke(t);
```

+ Add Comment ··· More

(d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der ForkJoin-Pool effektiv, um das 2×2 `negatedImage` Array aus Aufgabe 3c) zu füllen?

Assuming that the `ImageNegationFJ` *class is correctly parallelized, how many threads does the ForkJoinPool effectively use to fill the 2×2* `negatedImage` *array from task 3c)?* (2)

```java
double[][] image = {{0,1}, {1,0}};
```

```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                    - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```java
final static int CUTOFF = 32;
```

(d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der ForkJoin-Pool effektiv, um das 2×2 `negatedImage` Array aus Aufgabe 3c) zu füllen?

Assuming that the `ImageNegationFJ` *class is correctly parallelized, how many threads does the ForkJoinPool effectively use to fill the* 2×2 `negatedImage` *array from task 3c)?* (2)

```java
double[][] image = {{0,1}, {1,0}};
```

```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```java
final static int CUTOFF = 32;
```

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

Because of the sequential cutoff, only **one** Thread would be used *effectively*.

(e) Gehen Sie von einem konstanten Overhead von 16 MB = 2^4 MB pro Thread aus und dass pro Split immer vier neue Threads erstellt werden. Dies bedeutet, dass die Anzahl der Threads nicht durch den ForkJoinPool festgelegt wird, sodass kein Thread wiederverwendet wird und es zu keinem Work Stealing zwischen den Threads kommt. Was ist der niedrigste Wert für `CUTOFF`, wenn Sie ein Bild der Größe 4000×4000 eingeben, bevor Ihnen bei einem RAM der Größe 10 GB der Speicher ausgeht? Hinweis: 1 GB = 2^{10} MB.

Assume a fixed overhead of 16 MB = 2^4 MB per thread and that there are always four new threads created per split. This means that the number of threads is not fixed by the ForkJoinPool, so no thread is re-used and there is no work stealing among the threads. What is the lowest value for `CUTOFF` *if you input an image of size 4000×4000 before you run out of memory using a RAM of size 10 GB? Hint:* 1 GB = 2^{10} MB. (4)

(e) Gehen Sie von einem konstanten Overhead von 16 MB $= 2^4$ MB pro Thread aus und dass pro Split immer vier neue Threads erstellt werden. Dies bedeutet, dass die Anzahl der Threads nicht durch den ForkJoinPool festgelegt wird, sodass kein Thread wiederverwendet wird und es zu keinem Work Stealing zwischen den Threads kommt. Was ist der niedrigste Wert für `CUTOFF`, wenn Sie ein Bild der Größe 4000×4000 eingeben, bevor Ihnen bei einem RAM der Größe 10 GB der Speicher ausgeht? Hinweis: 1 GB $= 2^{10}$ MB.

Assume a fixed overhead of 16 MB $= 2^4$ MB per thread and that there are always four new threads created per split. This means that the number of threads is not fixed by the ForkJoinPool, so no thread is re-used and there is no work stealing among the threads. What is the lowest value for `CUTOFF` if you input an image of size 4000×4000 before you run out of memory using a RAM of size 10 GB? Hint: 1 GB $= 2^{10}$ MB. (4)

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

14

Number of threads, which we can use:

$$N = \frac{10 \cdot 2^{10}}{2^4} = 10 \cdot 2^6 = 10 \cdot 4^3$$

In each recursive call, we will use 4 new threads (under given assumptions). Thereby, we have the constraint ($i :=$ number of divisions)

$$4^i \leq 10 \cdot 4^3 \iff i \leq \log_4(10) + 3 \iff i \leq 4$$

and the smallest possible value is $\text{CUTOFF} = 4000/2^4 = \mathbf{250}$ to avoid a fifth division.

Add Comment ··· More

Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- Theory Recap
- Intro Exercise 6
- Exam Questions
- **Kahoot**

QUIZ

Feedback

- Falls ihr Feedback möchtet sagt mir bitte Bescheid!
- Schreibt mir eine Mail oder auf Discord

Danke

- Bis nächste Woche!