Parallel Programming

Divide and Conquer, Executor Service, ForkJoin Framework

What we have seen so far

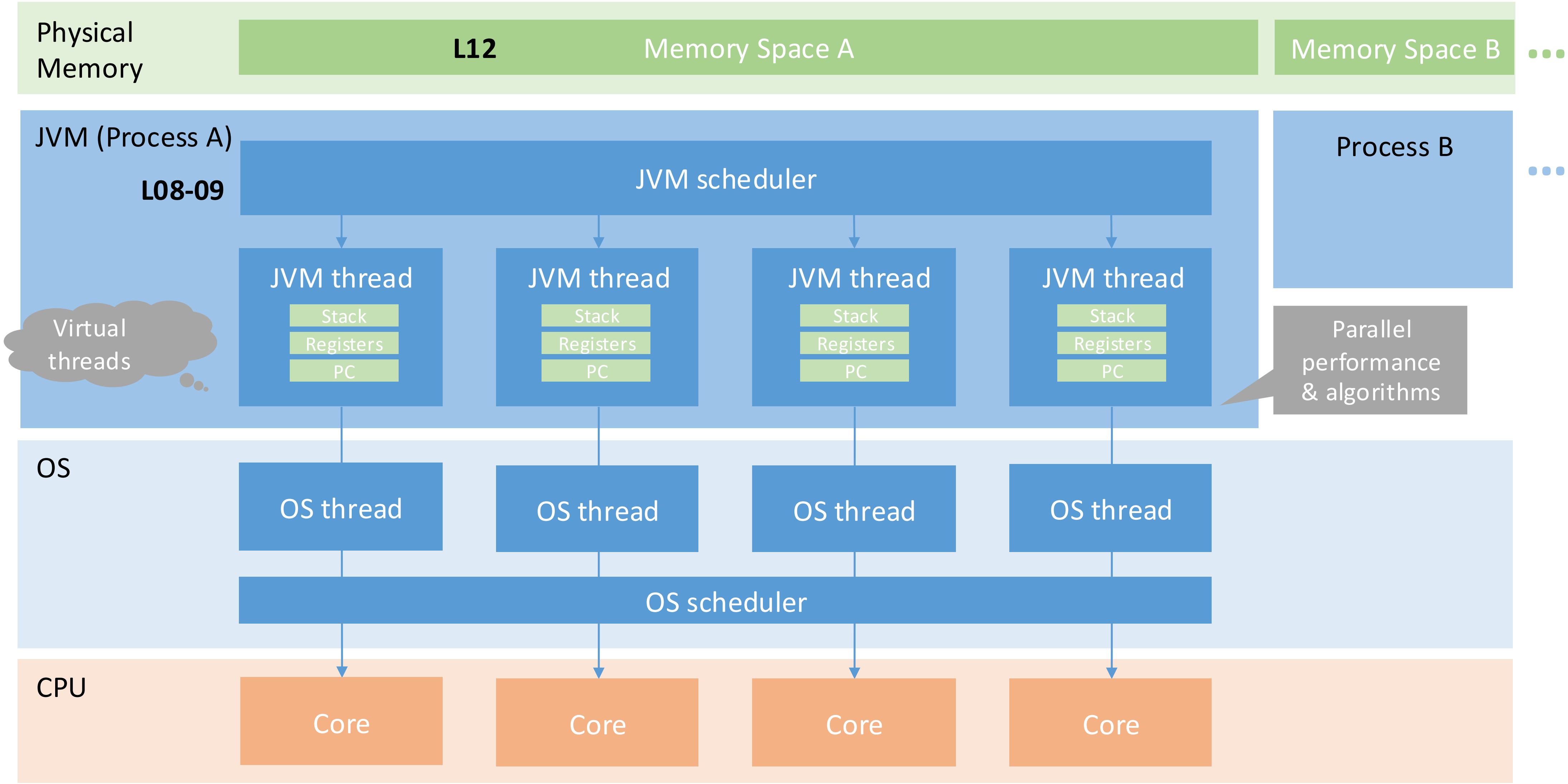
Optimizing sequential or parallel code by considering hardware design (L06)

Using Java threads to take advantage of multiple cores (L03-05)

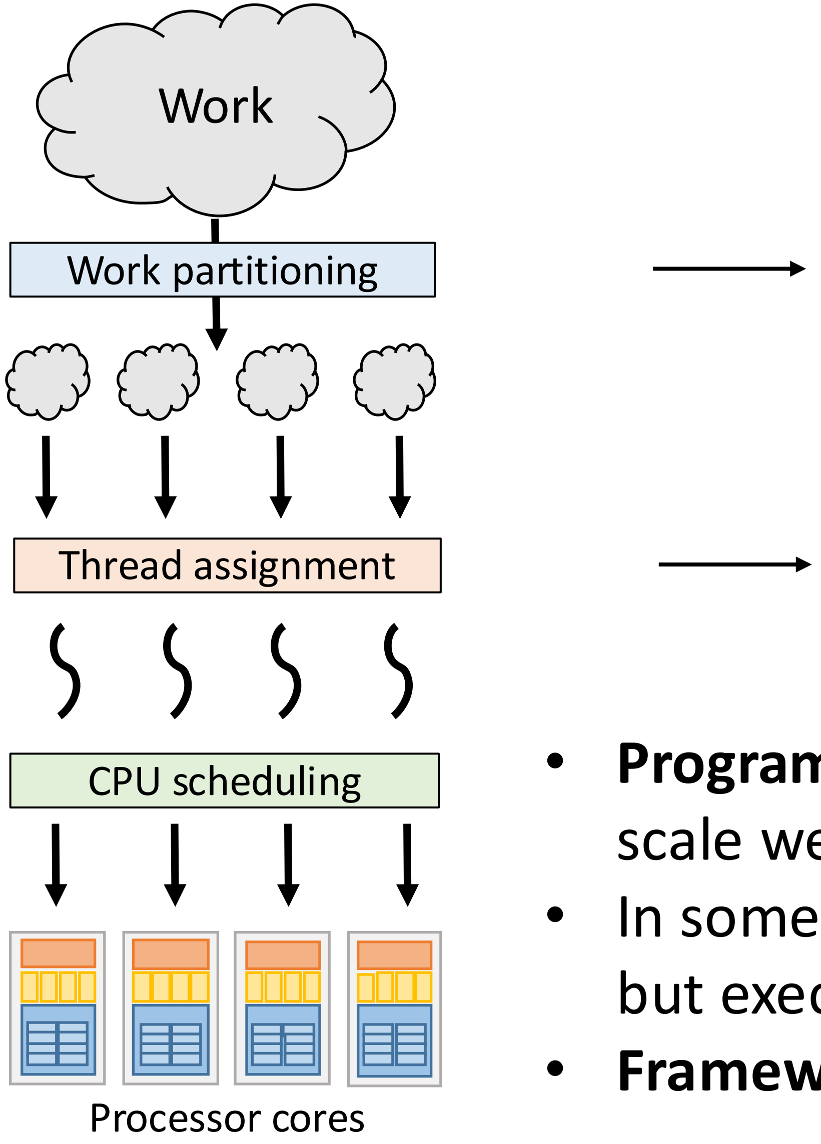
Scalability, metrics, and Amdahl's Law (L07) (preview: the importance of algorithms and data structures / L10)

Today: How work distribution and thread assignment / user-level scheduling strategies impact performance (L08-09)

Big picture (part I)



Java Threads: Suboptimal by default



Manually partitioning work is tedious, can lead to load imbalance, and does not scale well

Manual management of threads (relying on explicit start/join mechanisms) makes achieving optimal thread assignment difficult

- **Programmer** should focus on designing parallel algorithms that scale well
- In some cases, work partitioning is guided by the **programmer** but executed automated
- **Framework** handles thread assignment / user-level scheduling

Structure of next lectures

Back to Java Threads

Divide and Conquer

Executor Service

Fork-Join Framework

Back to Java Threads
Code example: Summing elements of an array

Sequential version

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

- Helps validate our eventual parallel program is correct
  - By comparing results with the simpler, sequential version

- Evaluate the performance of our parallel program
  - We write parallel programs to improve performance
  - But recall: What should T1 (baseline) be

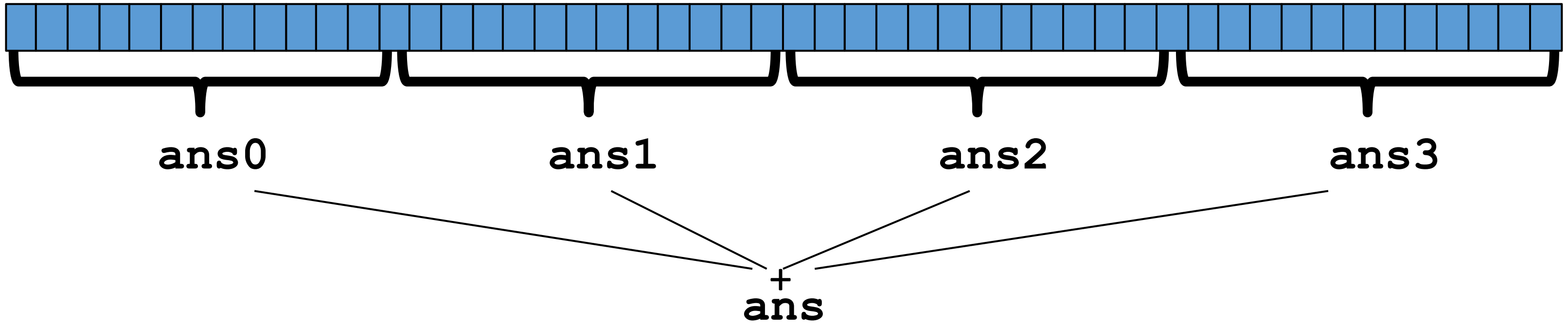
Adding numbers - sequentially

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

Parallelism idea

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

- This is an inferior first approach



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

Code example: PP-L08-01ArraySum

(additional material)

Sum with Java Threads

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

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

    int ans = 0; // result

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

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

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

Sum with Java Threads

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

int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){ // do parallel computations
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start(); // start the threads
    }
    for(int i=0; i < 4; i++) { // combine results
        ts[i].join(); // wait for helper to finish! -> try catch
        ans += ts[i].ans;
    }
    return ans;
}
```

Shared memory?

This style of parallel programming is called **fork-join**

Fork-join programming helps reduce race conditions by encouraging independent tasks

When using shared memory, you must avoid race conditions

Issues with this approach (and some workarounds)

Several reasons why this is a poor parallel algorithm

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

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

Issues with this approach (and some workarounds)

Several reasons why this is a poor parallel algorithm

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

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

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

Code example: PP-L08-02ParameterizedThreads

(additional material)

Issues with this approach (and some workarounds)

Reason 2: We still need to commit to a number of threads a priori

- Number of threads = number of cores might not be optimal
- Available cores can change even while your threads run
- We want this to be **dynamic**

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

Issues with this approach (and some workarounds)

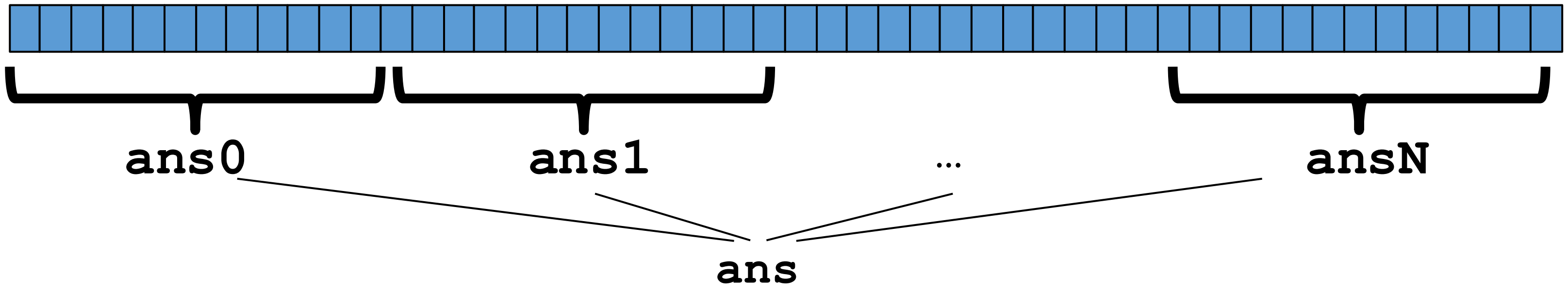
Reason 3: Subproblems may take significantly different amounts of time (though unlikely for `sum`)

- Problem of **load imbalance**
- Example: Apply method **`f`** to every array element, but maybe **`f`** is much slower for some data items, e.g.: is a large integer prime?
- If we create 4 threads and all slow data is processed by 1 of them, we won’t get nearly a 4x speedup

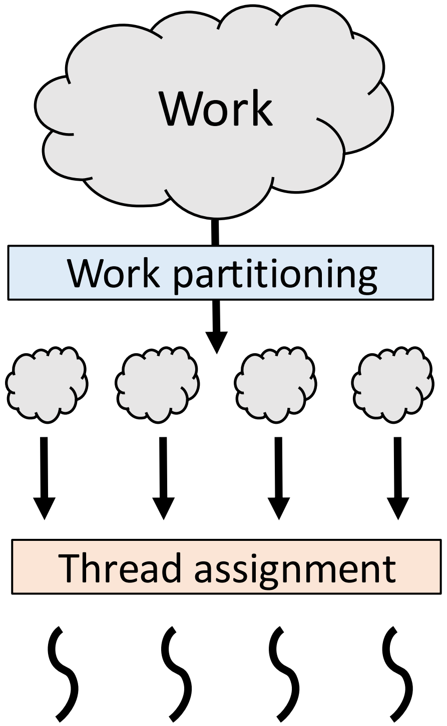
A better approach

The **better strategy** is to expose as much parallelism as possible and let the JVM runtime efficiently distribute the work across threads

- But this will require changing our algorithm (→ now)



- And for constant-factor reasons, abandoning Java Threads (→ afterwards)

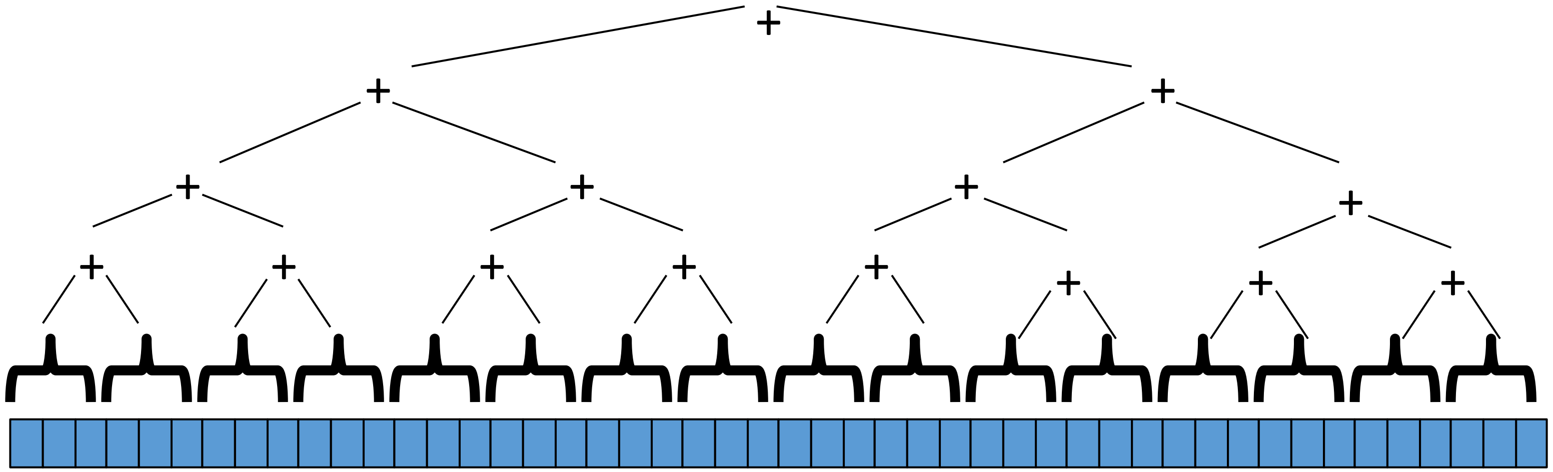


Divide and Conquer

Divide and Conquer to the rescue!

This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls



Divide and Conquer

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

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

Sequential version: Recursive sum

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

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

Code example: PP-L08-03ParallelRecursiveSum

Parallel recursive sum (with Threads)

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


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


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

Parallel recursive sum (with Threads)

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

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

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

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

Is this OK? -> no parallelism!

Parallel recursive sum (with Threads)

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

    t1.start();
    t2.start();

    t1.join();
    t2.join();

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

Remark: This doesn’t compile because join() can throw exceptions. We need a try-catch block here.

Result

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

One thread per parallel task model

Java threads are actually quite heavyweight

One-to-one mapping from Java threads to OS threads
(in the Oracle and most real-world implementations – exception:
Project Loom / virtual threads (L13))

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

Manual fixes (not optimal)

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

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

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

Code example: PP-L08-04RecursiveSumOpt

Manual fixes (not optimal) - Cutoff

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

cutoff, eliminating bottom levels of tree

Manual fixes (not optimal) - Half the threads

```
int mid = size / 2;
SumThread t1 = new SumThread(xs, l, l + mid);
SumThread t2 = new SumThread(xs, l + mid, h);

t1.start();
t2.start();

t1.join();
t2.join();

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

start two threads in each recursive step
-> wasteful

```
int mid = size / 2;
SumThread t1 = new SumThread(xs, l, l + mid);
SumThread t2 = new SumThread(xs, l + mid, h);

t1.start();
t2.run();
t1.join();

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

only start one thread, call run() directly on t2 object
-> better performance, but we once said that we should never call run() directly…

Java Threads - Discussion

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

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

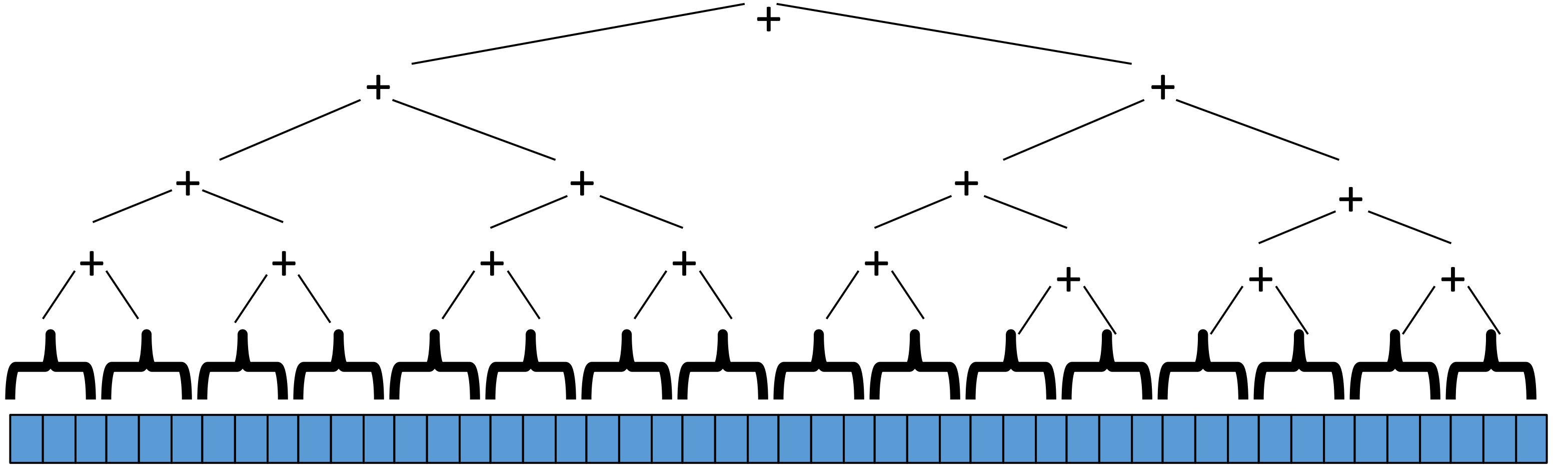
Divide-and-Conquer really works (but it is hard work)

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

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

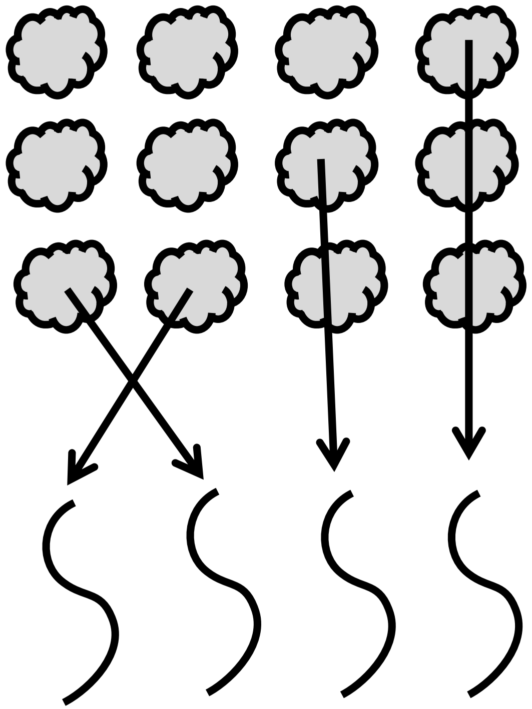
Will write all our parallel algorithms in this style

- But using special libraries engineered for this style (→ Java's Fork Join)
  - Takes care of scheduling the computation well



Executor Service

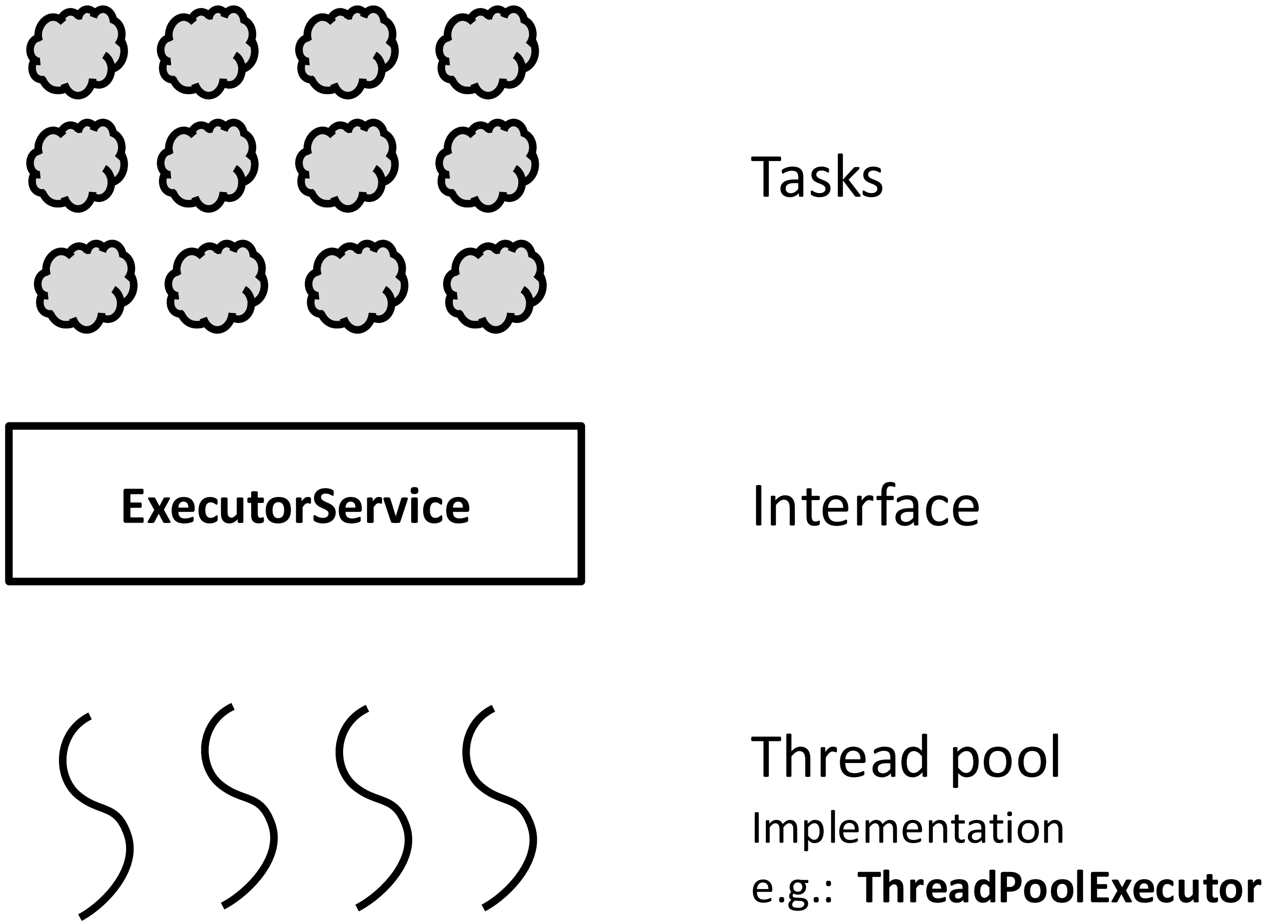
Alternative approach: Schedule tasks on threads



Tasks

Thread pool

Java's executor service: Managing asynchronous tasks



ExecutorService

Tasks

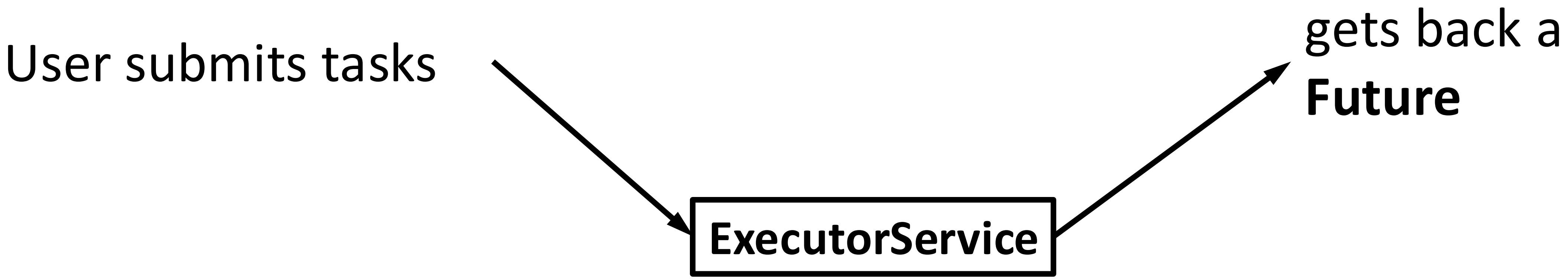
Interface

Thread pool

Implementation

e.g.: **ThreadPoolExecutor**

Java's executor service:managing asynchronous tasks



User submits tasks

ExecutorService

gets back a **Future**

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

Note: Callable vs Runnable

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

Interface Runnable:

→ void run() → Does not return result

Interface Callable<T>:

→ T call() → Returns result

Code example: PP-L09-01ExecutorHelloTask

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

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

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

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

Using executor service: Hello World (task)

```java
static class HelloTask implements Runnable {

    String msg;

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

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

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

Code example: PP-L09-02RecursiveSumExecutor

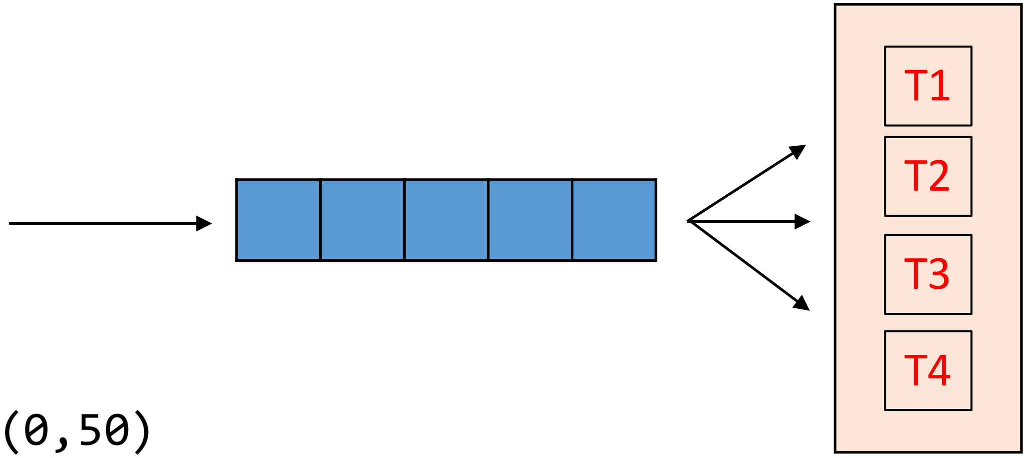
Recursive sum with ExecutorService

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

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

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

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

Simple! – But does this work?

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

Why does this happen?



T1

```
sum(0,100):
  f1 = submit sum(0,50)
  f2 = submit sum(50,100)
  return f1.get() + f2.get()
```

T2

```
sum(0,50):
  f1 = submit sum(0,25)
  f2 = submit sum(25,50)
  return f1.get() + f2.get()
```

T3

```
sum(50,100):
  f1 = submit sum(50,75)
  f2 = submit sum(75,100)
  return f1.get() + f2.get()
```

T4

```
sum(0,25):
...
```

?

```
sum(25,50):
...
```

tasks will end up waiting

eventually we will **run out of threads**

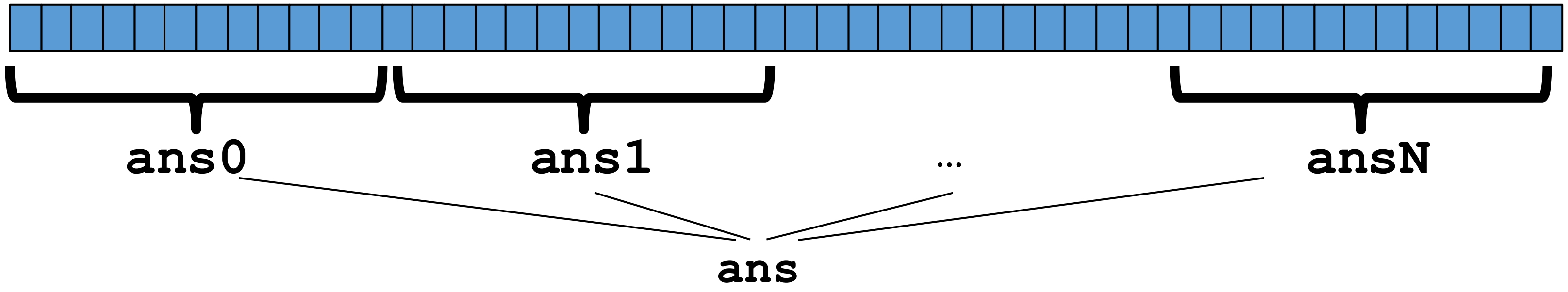
Executor Service - Discussion

Not suitable for **recursive problems** where deep structures require waiting for partial results

Well-suited for **flat structures** or tasks that can run independently in parallel (e.g., transactions)

Adding Numbers ExecutorService: Another approach

- Problem with the divide and conquer approach: tasks create other tasks and work partitioning (splitting up work) is part of the task
- A possible approach is to decouple work partitioning from solving the problem
  - Split the array into chunks (how many?) and create a task per chunk
  - Then, we submit tasks into ExecutorService and combine results (e.g., sum)
  - It can be tricky to do the initial partitioning of work and final summing in parallel



Fork Join Framework

Java’s ForkJoin framework

We need a new **framework that supports Divide-and-Conquer style parallelism**

That is, when a task is waiting, it is suspended and other tasks are allowed to run!

That library, finally

The **ForkJoin Framework** is designed to meet the needs of divide-and-conquer fork-join parallelism

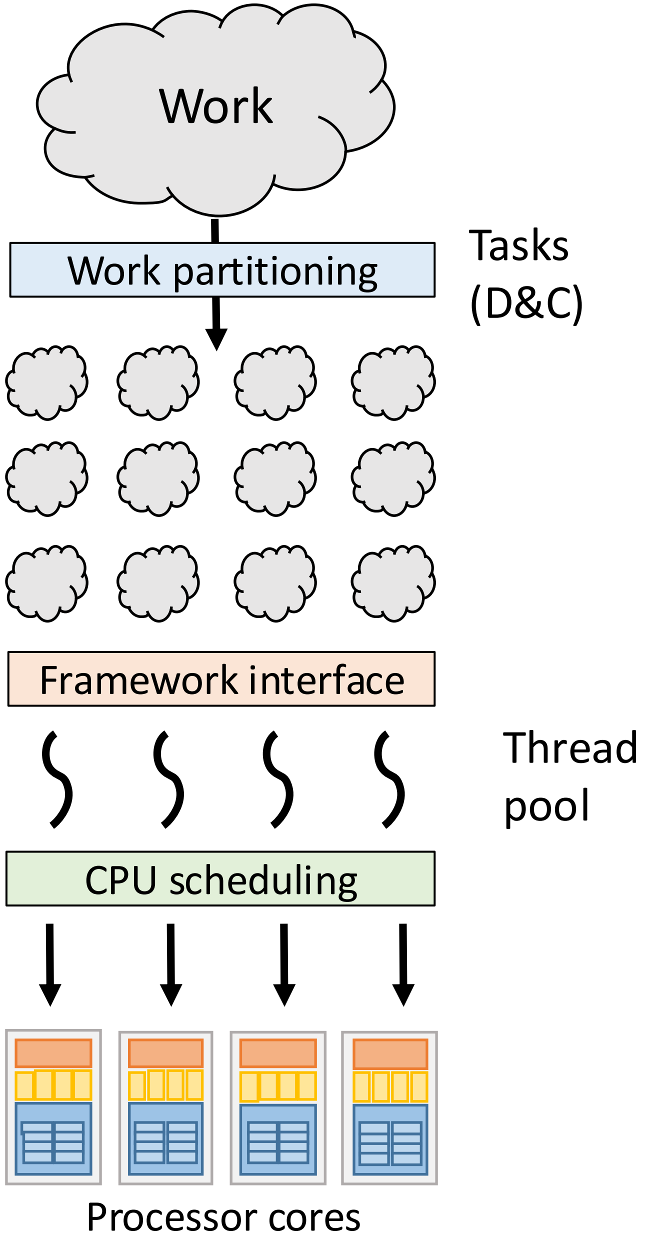
- In the Java standard libraries

Lecture will focus on pragmatics/logistics

- Similar libraries available for other languages
  - C/C++: Cilk (inventors), Intel's Thread Building Blocks
  - C#: Task Parallel Library
  - …
- Library's implementation is also fascinating

Fork Join

- Partition the work into tasks with D&C
- Submit the FJ task to the FJ interface
- Framework assigns tasks to FJ thread pool



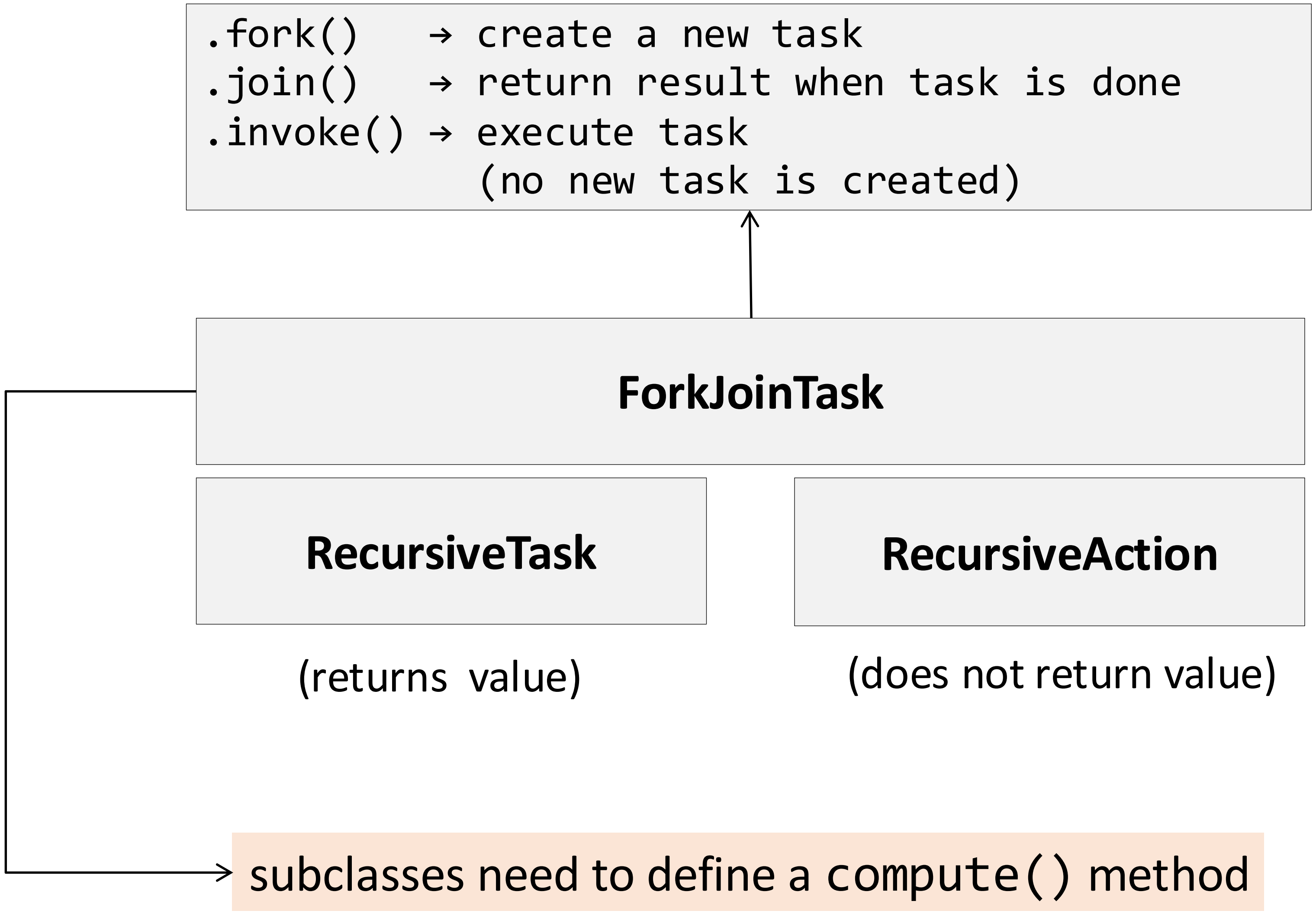
Different terms, same basic idea

To use the ForkJoin Framework:

A little standard set-up code (e.g., create a **ForkJoinPool**)

Don’t subclass **Thread**	↔	Do subclass **RecursiveTask<V>**
Don’t override **run**	↔	Do override **compute**
Do not use an **ans** field	↔	Do return a **V** from **compute**
Don’t call **start**	↔	Do call **fork**
Don’t just call **join**	↔	Do call **join** which returns answer
Don’t call **run** to hand-optimize	↔	Do call **compute** to hand-optimize
Don’t have a topmost call to **run**	↔	Do create a pool and call **invoke**

Tasks in Fork/Join framework



```
.fork()   → create a new task
.join()   → return result when task is done
.invoke() → execute task
            (no new task is created)
```

ForkJoinTask

RecursiveTask

(returns value)

RecursiveAction

(does not return value)

subclasses need to define a `compute()` method

Tasks in Fork/Join framework – Usual procedure

`t.fork()` → spawn a new task

… → possibly do other things

`res = t.join()` → wait for task result

`t.invoke()` → execute the task computation without spawning a task (in-place)

Code example: PP-L10-03ForkJoinRecursiveSum

Recursive sum with ForkJoin (use)

Default # of processors

```
class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}


static long sumArray(int[] array) {
    return Globals.fjPool.invoke(new SumForkJoin(array,0,array.length));
}
```

ForkJoinPool:

- constructor creates a number of threads equal to the number of available processors
- .invoke() submits task and waits until it is completed
- .submit() submits task (receives a Future)

Recursive sum with ForkJoin (RecursiveTask Class)

```
class SumForkJoin extends RecursiveTask<Long> {
    int low;
    int high;
    int[] array;


    SumForkJoin(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }


    protected Long compute() { /*…*/ }
```

Recursive sum with ForkJoin (compute)

```
protected Long compute() {
        if(high - low <= 1)
            return array[high];
        else {
            int mid = low + (high - low) / 2;
            SumForkJoin left  = new SumForkJoin(array, low, mid);
            SumForkJoin right = new SumForkJoin(array, mid, high);
            left.fork();
            right.fork();
            return left.join() + right.join();
        }
    }
```

Fixes/work-around – cont’d

We are splitting array to size=1, overhead

Make sure each task has ‘enough’ to do!

Framework tries to perform well even if the tasks do not have so much computation

```java
protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            SumForkJoin left  = new SumForkJoin(array, low, mid);
            SumForkJoin right = new SumForkJoin(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }
```

Pushes task to the local queue, making it stealable

Optimize, call compute: Executes task immediately in the current thread (avoids queuing)

Make sure each task has “enough” do do

Parallelism only helps if each task does enough work to compensate for the overhead of creating and scheduling tasks

- In our example: sum += a[i] -> real work per element is tiny, too many tasks -> framework spends more time managing tasks than computing sums

Tasks should perform approx. 100-10’000 basic operations before being split

- Sum array: sequential threshold

Summary

Java Threads: The programmer manually handles both partitioning and scheduling

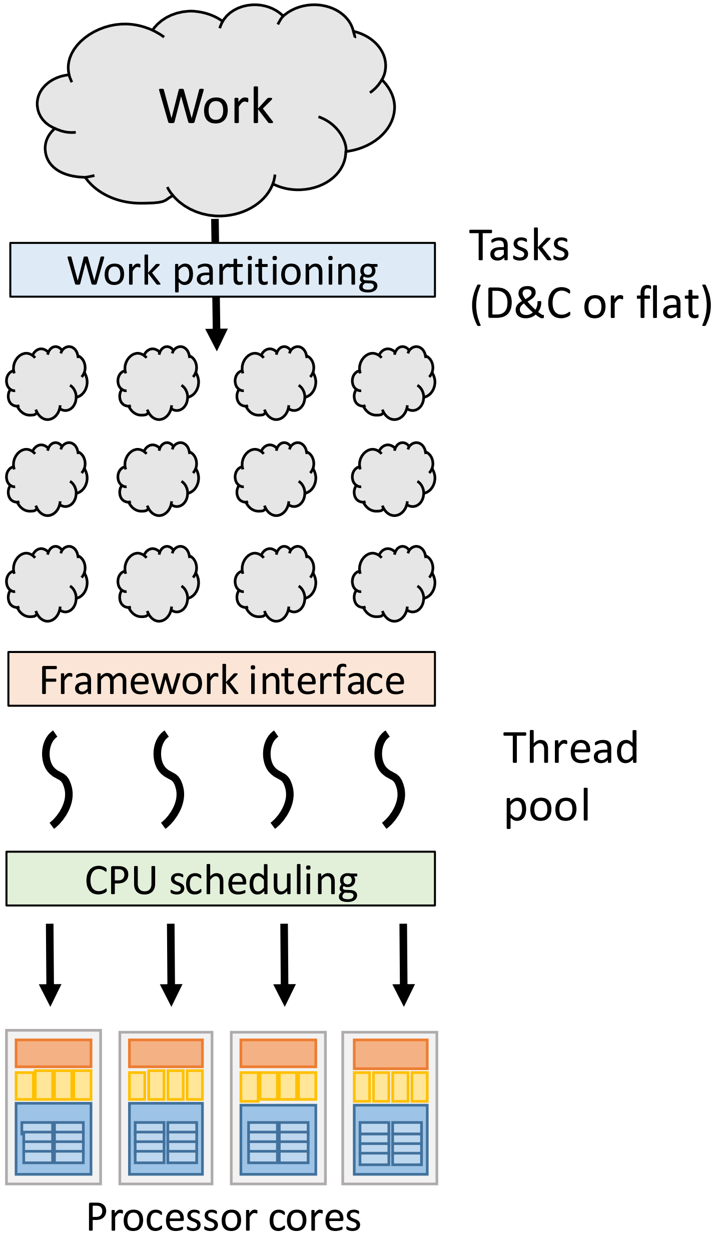
- Suboptimal (error-prone, inefficient, hard to scale)

Executors: The programmer partitions the work, but scheduling is handled automatically

- Well-suited for **flat structures** or tasks that can run independently in parallel (e.g., transactions)

Fork/Join Framework: Both partitioning and scheduling are automated

- Well-suited for **recursive problems** where deep structures require waiting for partial results


Work
Work partitioning
Tasks
(D&C or flat)
Framework interface
Thread
pool
CPU scheduling
Processor cores