Advanced Java Programming Course

Parallel Programming

Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

- Introduction
- Task parallelism
  - Task Creation and Termination
  - Tasks in Java's Fork/Join Framework
- Functional parallelism
  - Futures: Tasks with Return Values
  - Futures in Java's Fork/Join Framework
  - Java Streams
- Loop parallelism

Introduction

- Parallel Programming is a process of breaking down a complex problem into smaller and simpler tasks, which can be executed at the same time using a number of computer resources.
- In the process of Parallel Programming, the tasks which are independent of each other are executed parallelly using different computers or multiple crores present in a CPU of a computer.

Introduction

- Parallel Programming is an essential necessity for firms to handle heavy and large-scale projects as they need to maintain their economic standards. Due to Parallel Programming, the speed of the project is increased, and its probability of error is decreased.
- Parallel Programming is quite different from multithreading as its tasks have no need to follow an order of execution. Each task of Parallel Programming is designed according to the function they need to perform. Hence it is widely known as Functional Parallelism or Data Parallelism.

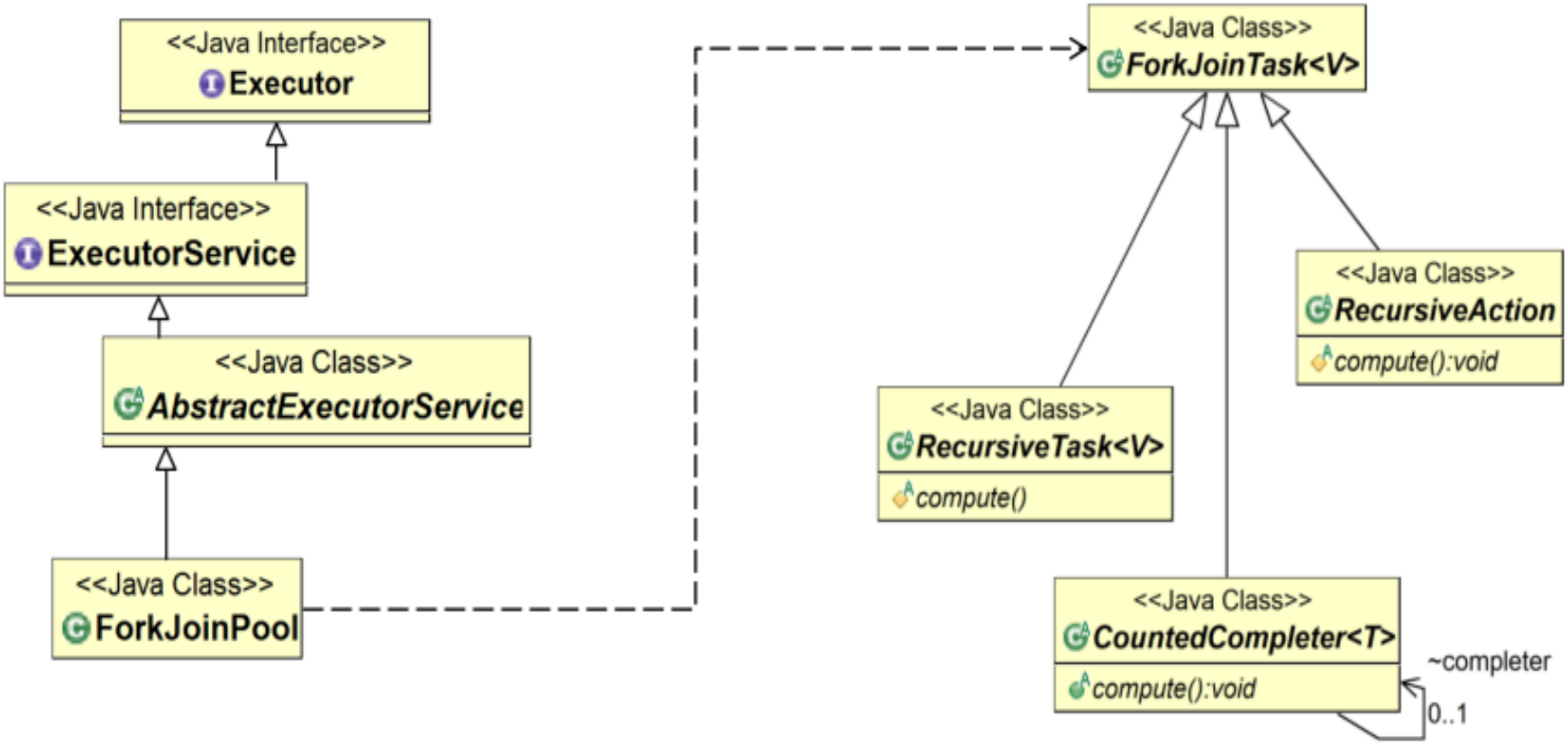
Task parallelism

- Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e., threads), as opposed to the data (data parallelism).
- We will use the fork/join framework to implement task parallelism.
- The fork/join framework helps to take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively.
- The goal is to use all the available processing power to enhance the performance of your application.

The fork/join framework

- The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors.
- It is designed for work that can be broken into smaller pieces recursively.
- The fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The fork/join framework



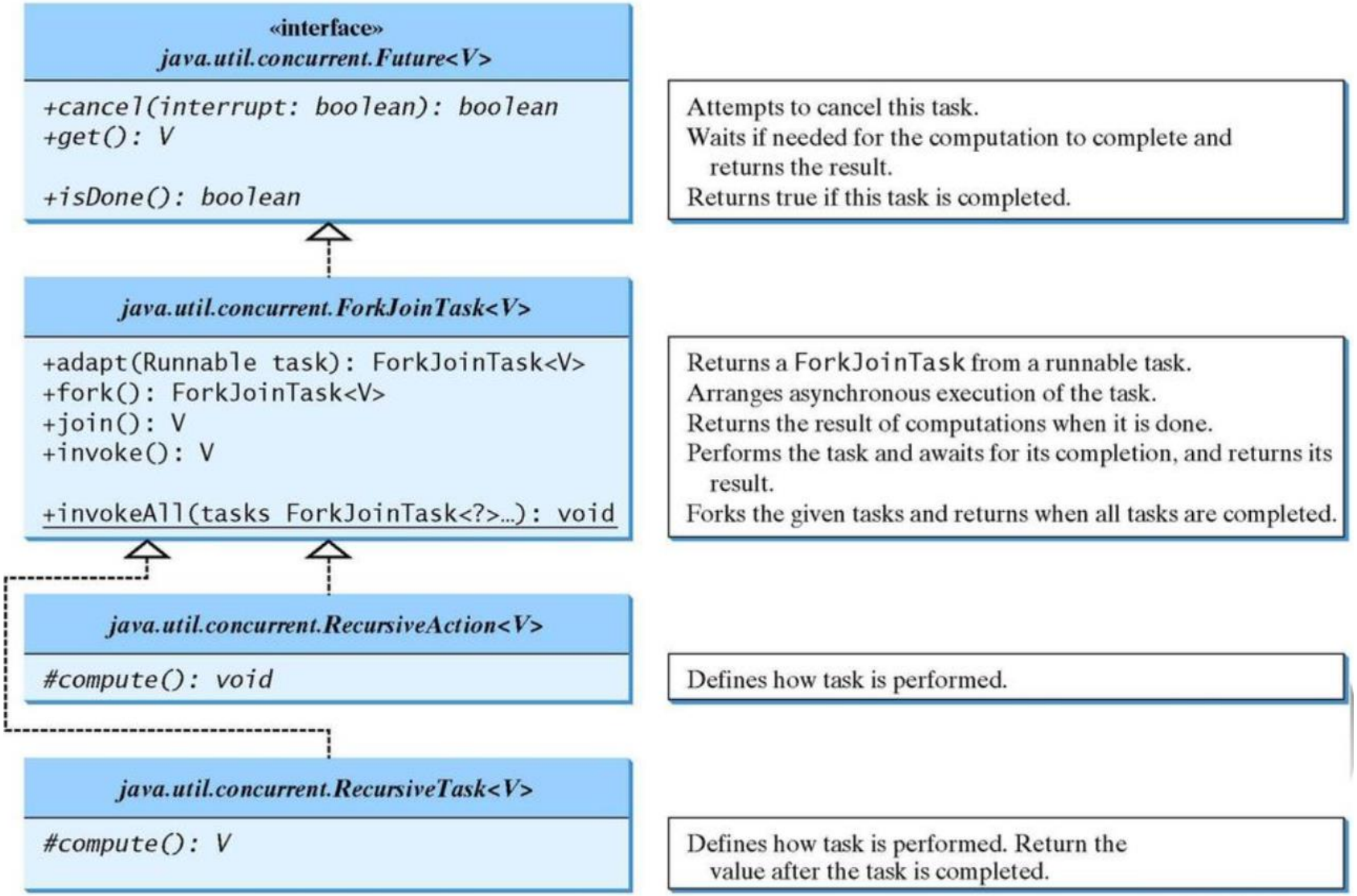
The fork/join framework

- The center of the fork/join framework is the ForkJoinPool class, an extension of the AbstractExecutorService class.
- ForkJoinPool implements the core work-stealing algorithm and can execute ForkJoinTask processes.
- ForkJoinTask has subclasses: RecursiveAction, RecursiveTask and ForkJoinTask.

The fork/join framework

- ForkJoinPool class invokes a task of type ForkJoinTask, which you have to implement by extending one of its subclasses:
  - RecursiveAction: which represents tasks that do not yield a return value, like a Runnable.
  - RecursiveTask: which represents tasks that yield return values, like a Callable.
- ForkJoinTask subclasses also contain the following methods:
  - fork(): which allows a ForkJoinTask to be scheduled for asynchronous executiond.
  - join(): which returns the result of the computation when it is done, allowing a task to wait for the completion of another one.

The fork/join framework



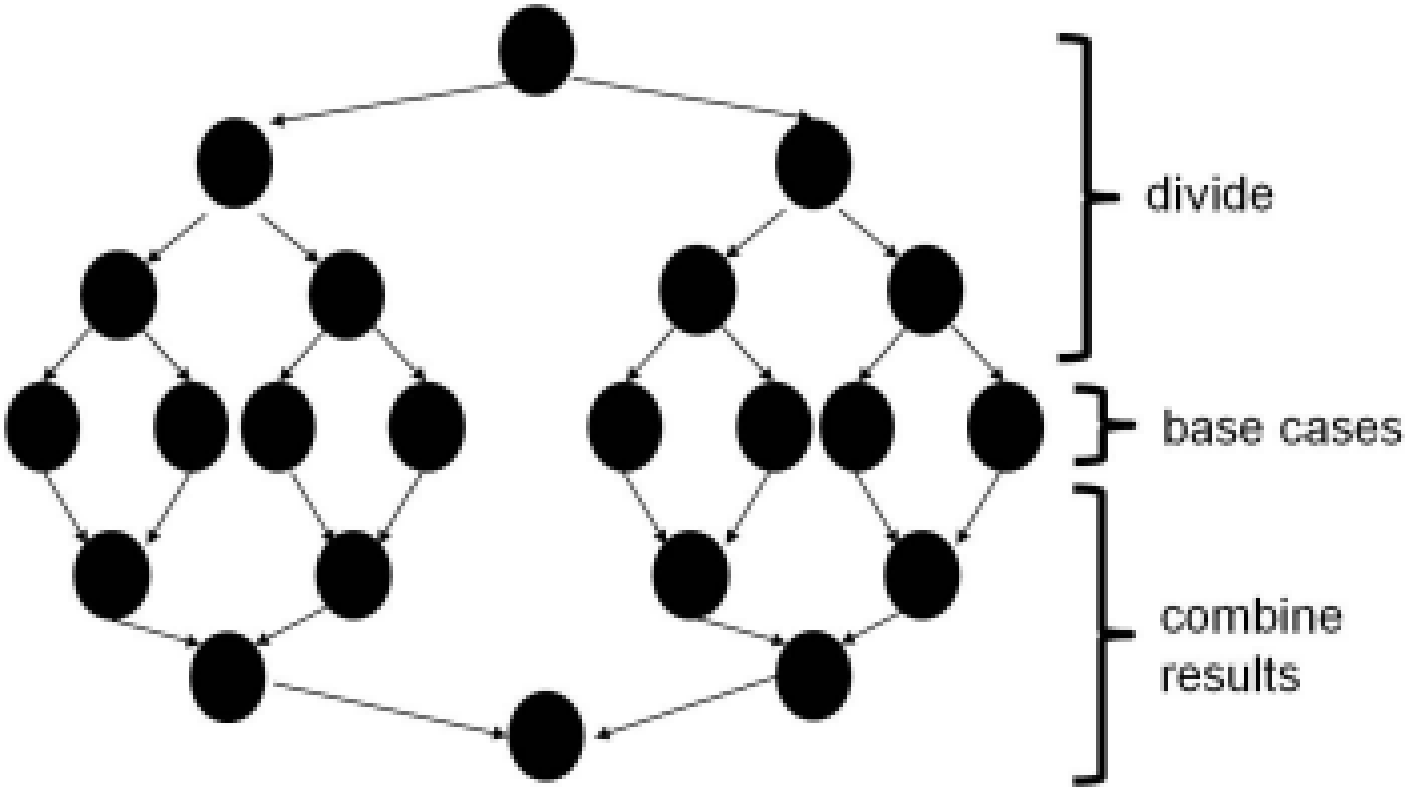
The fork/join framework

- Steps of execution
  - First, you have to decide when the problem is small enough to solve directly. This acts as the base case. A big task is divided into smaller tasks recursively until the base case is reached.
  - Each time a task is divided, you call the fork() method to place the first subtask in the current thread's deque, and then you call the compute() method on the second subtask to recursively process it.
  - Finally, to get the result of the first subtask you call the join() method on this first subtask. This should be the last step because join() will block the next program from being processed until the result is returned.

The fork/join framework

- Steps of execution
  - Thus, the order in which you call the methods is important. If you don't call fork() before join(), there won't be any result to retrieve. If you call join() before compute(), the program will perform like if it was executed in one thread and you'll be wasting time.
  - If you follow the right order, while the second subtask is recursively calculating the value, the first one can be stolen by another thread to process it. This way, when join() is finally called, either the result is ready or you don't have to wait a long time to get it.
  - You can also call the method invokeAll(ForkJoinTask<?>… tasks) to fork and join the task in the right order

The fork/join framework



The fork/join framework

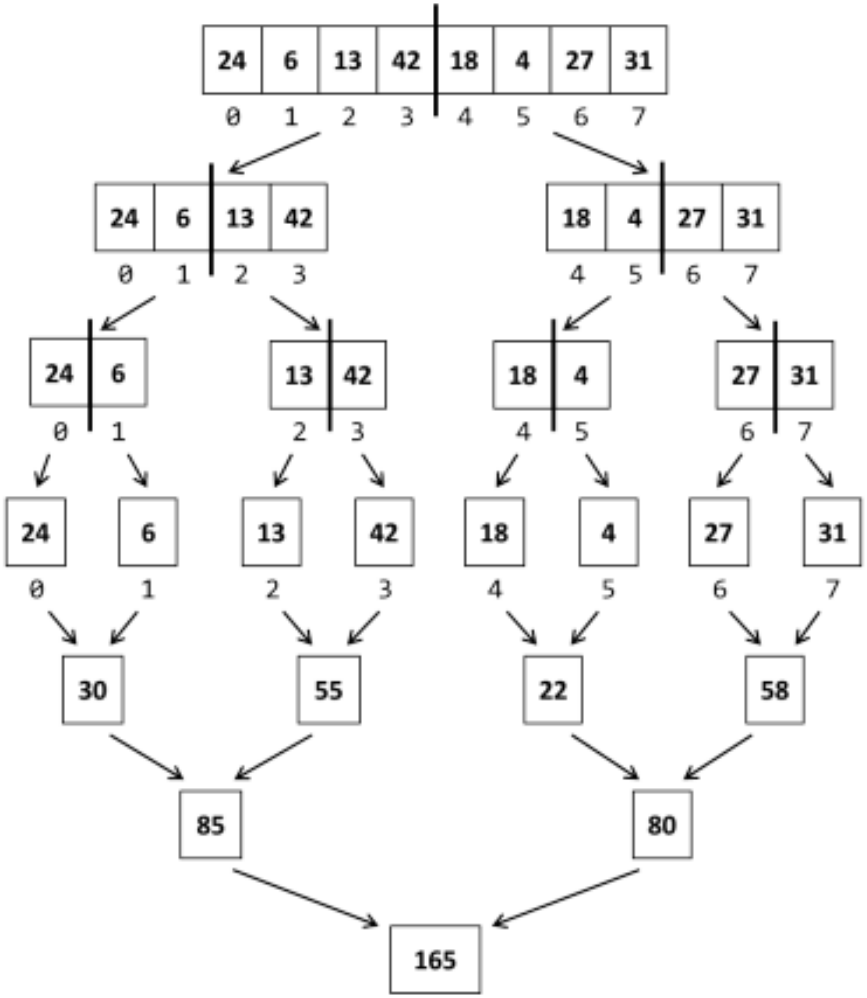
◊ With this framework, you will implement tasks whose main method will be something like this:

```
if ( problem.size() > DEFAULT_SIZE) {
        divideTasks();
        executeTask();
        taskResults = joinTasksResult();
        return taskResults;
} else {
        taskResults=solveBasicProblem();
        return taskResults;
}
```

The fork/join framework

- Implementation with RecursiveAction Interface
- Implementation with RecursiveTask Interface

The fork/join framework



Functional Parallelism

- **Futures: Tasks with Return Values**
- **Task Creation and Termination (Async, Finish)**

```
finish {
  async S1; // asynchronously compute sum of the lower half of the array
  S2;       // compute sum of the upper half of the array in parallel with S1
}
S3; // combine the two partial sums after both S1 and S2 have finished
```

Functional Parallelism

- **Parallel streams**
  - Streams and lambda expressions were the two most important new features of the Java 8 version.
  - Streams have been added as a method in the Collection interface and other data sources and allow the processing of all elements of a data structure generating new structures, filtering data, and implementing algorithms using the map and reduce technique.
  - A special kind of stream is a parallel stream that realizes its operations in a parallel way.

Functional Parallelism

- Sequential Streams:
  - Process elements in a stream in a serial manner, one after another.
  - Operations are executed in a single thread, following the order of the stream.
  - Suitable for small to medium-sized data sets or operations where the order of execution is important.
  - Can be created by default when working with streams.

Functional Parallelism

- Parallel Streams:
  - Process elements in a stream concurrently, utilizing multiple threads.
  - Operations are divided into multiple tasks that can be executed simultaneously on different threads.
  - Suitable for large data sets or operations that can be performed independently.
  - Created by converting a sequential stream to a parallel stream using the parallel() method.

Functional Parallelism

- A parallel stream is split into multiple substreams that are processed in parallel by multiple instances of the stream pipeline being executed by multiple threads, and their intermediate results are combined to create the final result.
- There are two ways we can create which are listed below and described later as follows:
  - Using parallel() method on a stream
  - Using parallelStream() on a Collection

Functional Parallelism

◊ **Parallel streams**

✧ The most important elements involved in the use of parallel streams are:

- The Stream interface: This is an interface that defines all the operations that you can perform on a stream.
- Optional: This is a container object that may or may not contain a non-null value.
- Collectors: This is a class that implements reduction operations that can be used as part of a stream sequence of operations.
- Lambda expressions: Streams have been thought of to work with Lambda expressions.

Loop Parallelism

- **Parallel Loops**
  - We learned different ways of expressing parallel loops. The most general way is to think of each iteration of a parallel loop as an async task, with a finish construct encompassing all iterations.

```
finish {
for (p = head; p != null ; p = p.next)
   async compute(p);
}
```

  - Java streams can be an elegant way of specifying parallel loop computations

```
a = IntStream.rangeClosed(0, N-1).parallel().toArray(i -> b[i] + c[i]);
```