Java Programming Course

Parallel Programming with Java

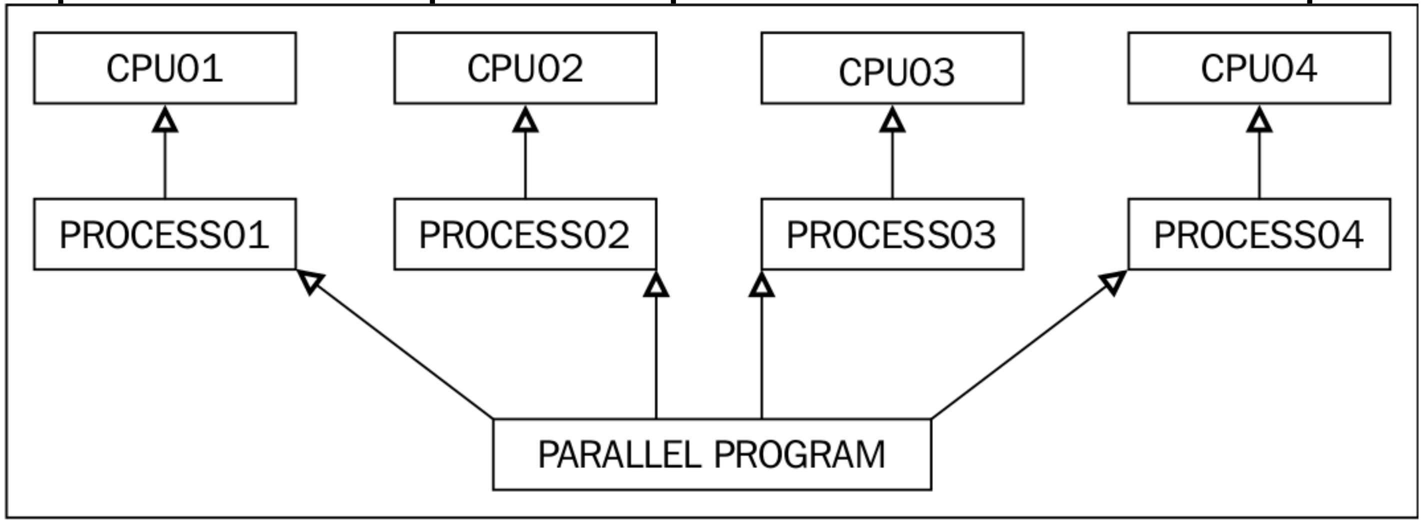
Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

- Introduction
- Fork/Join Framework
  - ForkJoinPool
  - ForkJoinTask
    - RecursiveTask
    - RecursiveAction
- Task Parallelism
- Functional Parallelism
- Loop Parallelism
- Pipelining 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 cores present in a CPU of a computer.



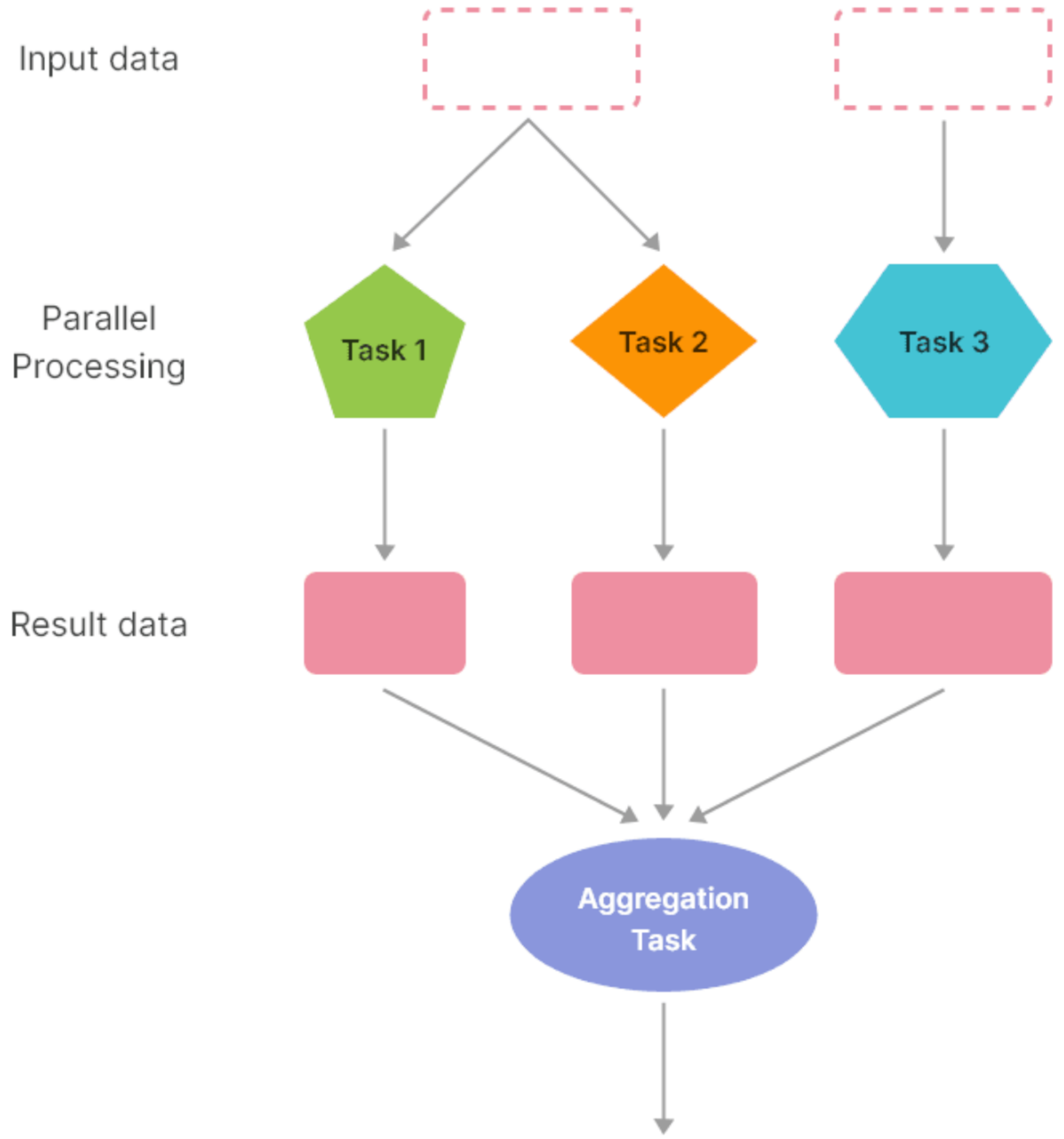
Introduction (cont'd)

- ◇ Parallelism is a form of computation in which many instructions are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel).
- ◇ Styles of parallelism:
  - Task parallelism
  - Data parallelism

Task Parallelism

- Also know as function parallelism.
- Approach that focuses on distributing task rather than data across different processing units. Each of these tasks can be a separate function or a method operating on different data or performing different computations. This type of parallelism is a great for problems where different operations can be performed concurrently on the same or different data.

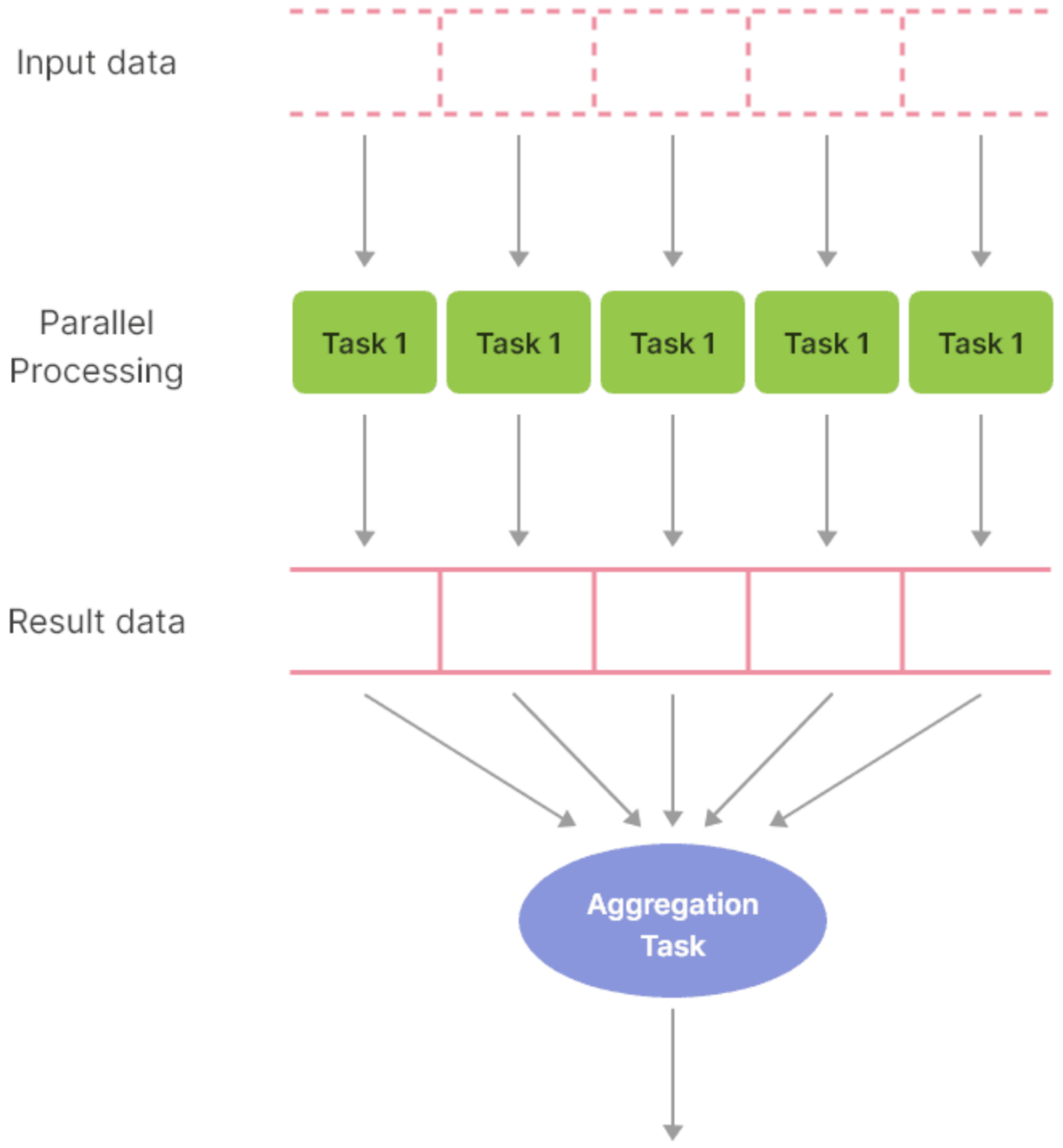
Task Parallelism (cont'd)



Data Parallelism

- The dataset is split into smaller, more manageable chunks, or partitions. Each partition is processed independently by separate tasks running the same operation. This distribution is done in a way that each task operates on a different core or processor, enabling high-level parallel computation.

Data Parallelism (cont'd)



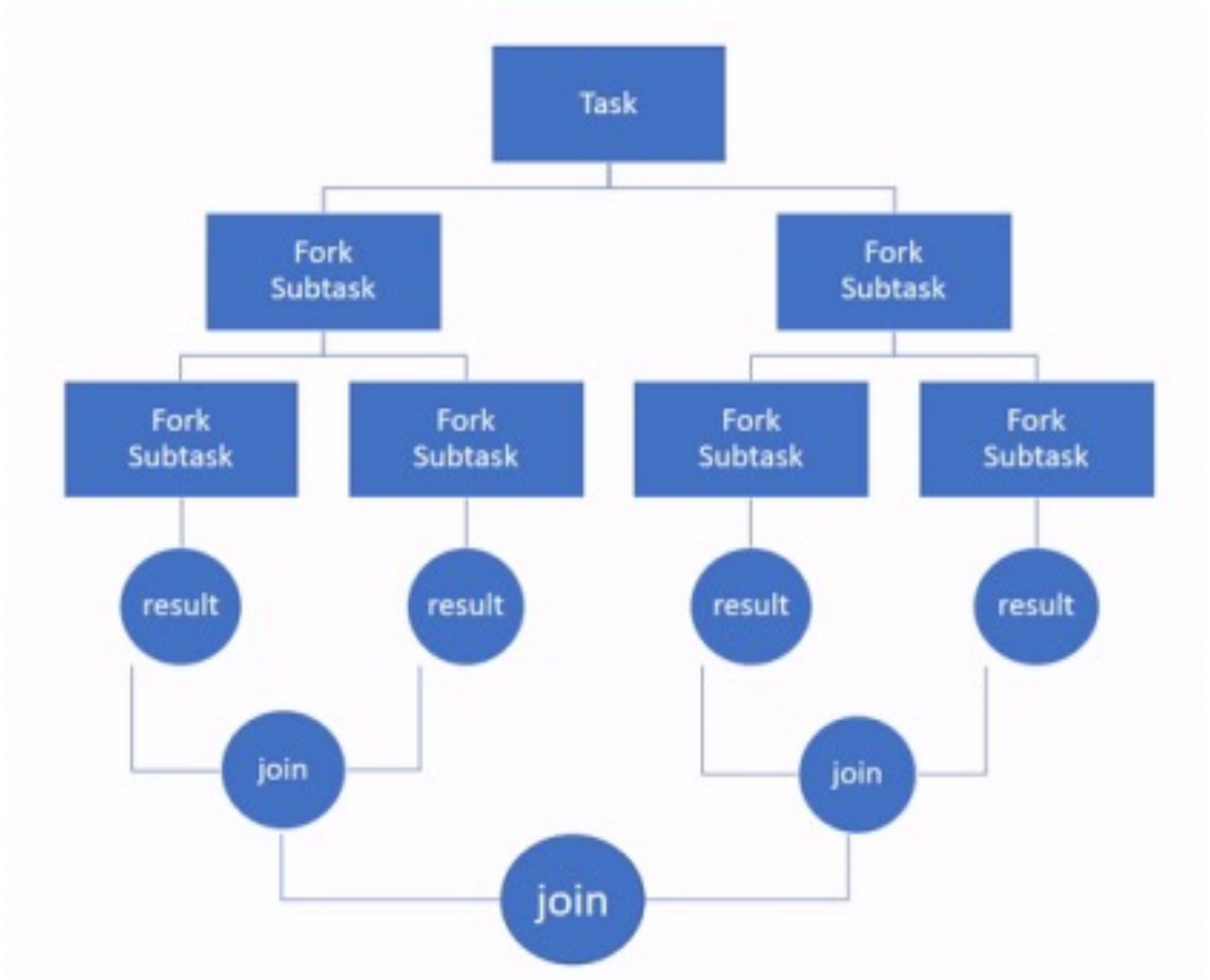
Why is Parallel Programming used?

- Basically, Parallel Programming refers to the parallel execution of processes due to the availability of multiple resources such as processing cores.
- Parallel Programming is considered to be a much more efficient method than multithreading. The parallel tasks which are solved simultaneously are combined to give the final solution to a larger problem.
- The Java Standard Environment (Java SE) provides a programmer with the " Fork/Join Framework ". The Fork/Join Framework helps a programmer to implement Parallel Programming easily in their applications.

Fork/Join Framework
An overview of Java ForkJoinPool

- Since Java version 1.7
- The ForkJoinPool is similar to the Java ExecutorService but with one difference that it easily split the working task into smaller tasks which are then submitted to the ForkJoinPool (this pool manages worker threads of type ForkJoinWorkerThread); and finally joins the results (if any).

Fork/Join Framework
An overview of Java ForkJoinPool (1)



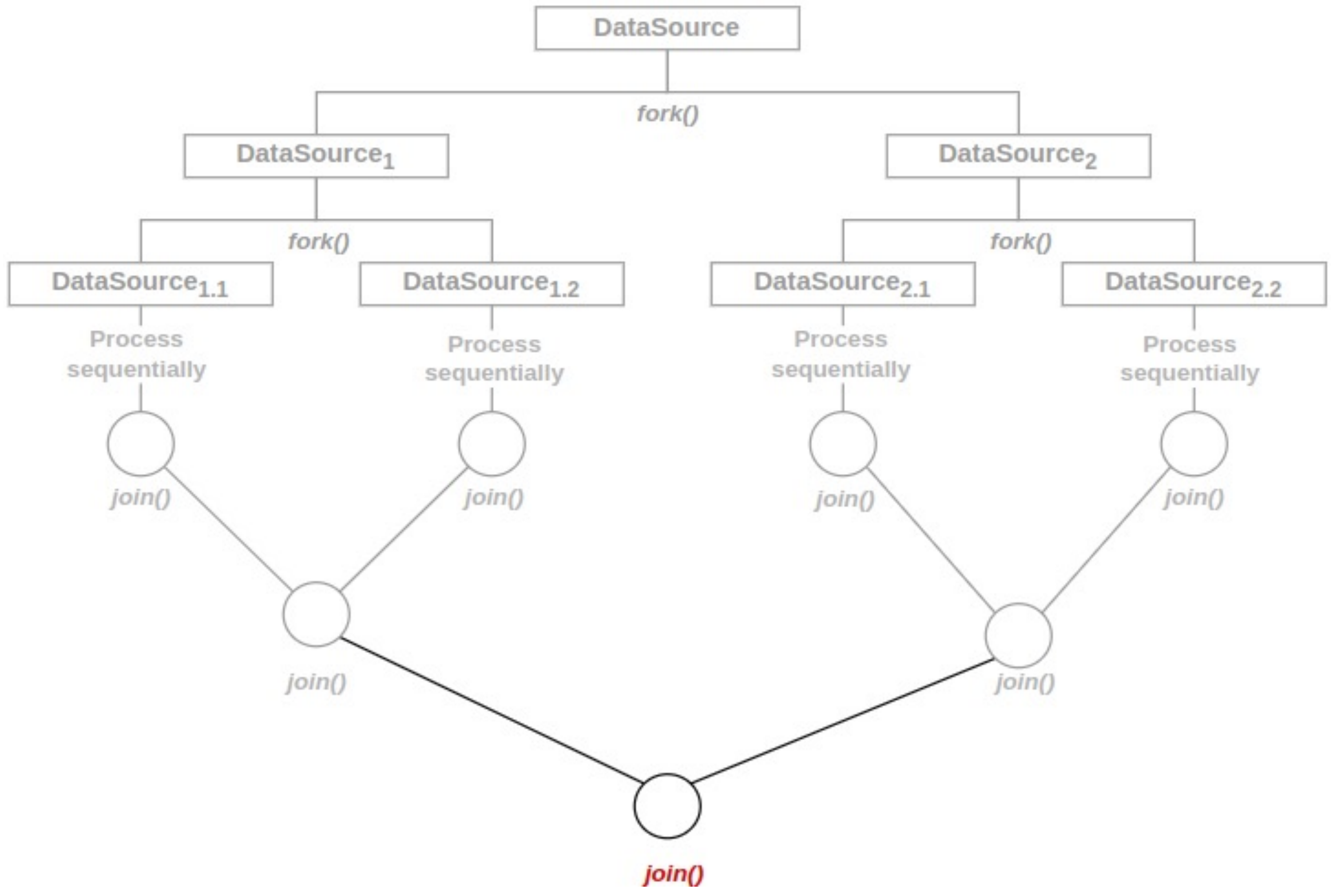
Fork/Join Framework
An overview of Java ForkJoinPool (2)

- ◇ The fork-join pool supports a style of parallel programming that solves problems by "divide & conquer".
- ◇ Worker threads can execute only one task at a time, but the ForkJoinPool doesn't create a separate thread for every single subtask. Instead, each thread in the pool has its own double-ended queue (or deque) that stores tasks.
- ◇ This architecture is vital for balancing the thread's workload with the help of the work-stealing algorithm.

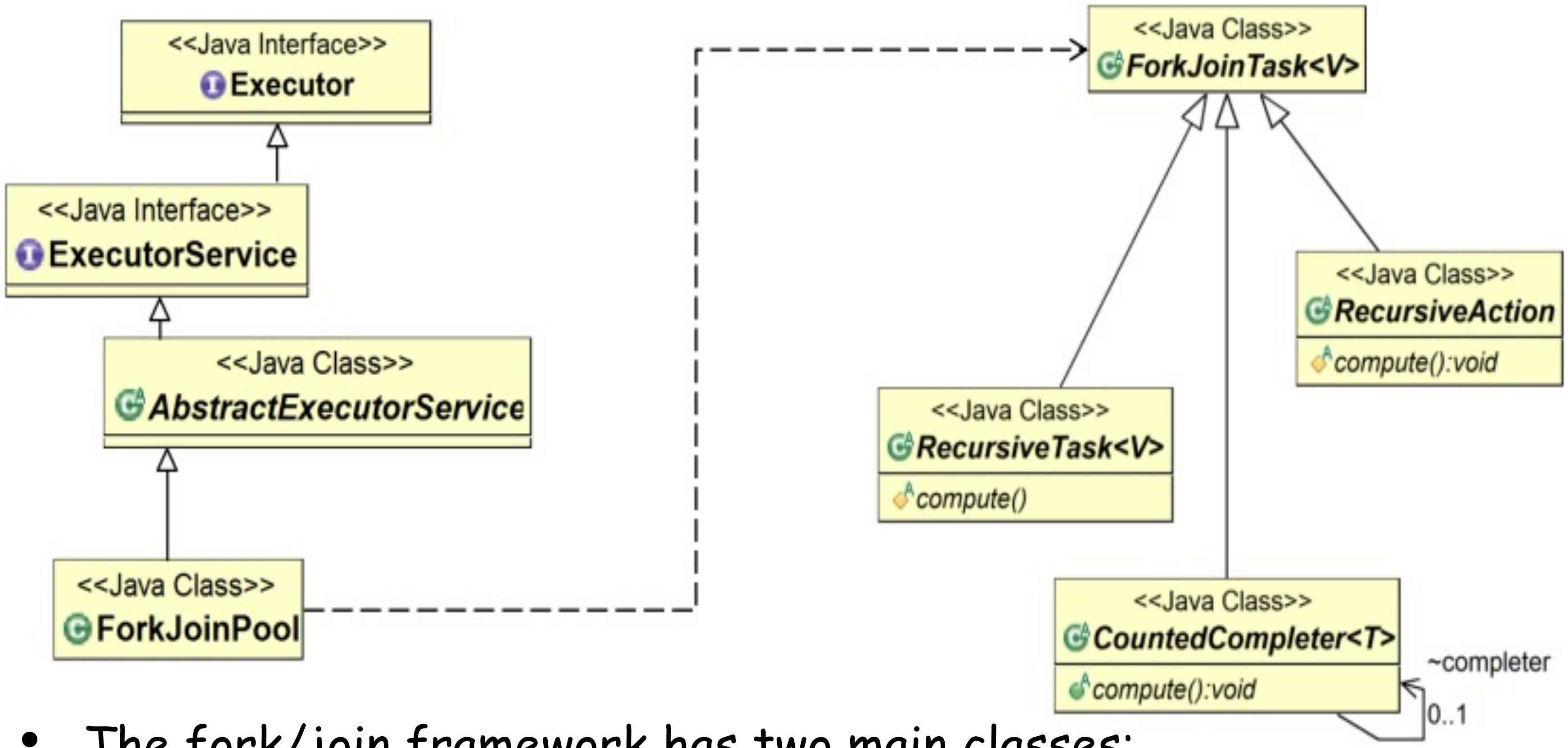
Fork/Join Framework
An overview of Java ForkJoinPool (3)

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve(part)
        join all subtasks spawned in previous loop
        return combined results
```

Fork/Join Framework
An overview of Java ForkJoinPool (4)



Fork/Join Framework
Structure & Functionality



- The fork/join framework has two main classes:
  - ForkJoinPool
  - ForkJoinTask

Fork/Join Framework
ForkJoinPool class

- ◇ Is an implementation of the interface ExecutorService.
- ◇ In general, executors provide an easier way to manage concurrent tasks than plain old threads. The main feature of this implementation is the work-stealing algorithm.
- ◇ There's a common ForkJoinPool instance available to all applications that you can get with the static method commonPool()
  - ForkJoinPool commonPool = ForkJoinPool.commonPool();
- ◇ Constructors:
  - ForkJoinPool()
  - ForkJoinPool(int parallelism)
  - …

Fork/Join Framework
ForkJoinPool class (1)

- ◇ Key methods:
  - execute(): Desired asynchronous execution; call its fork method to split the work between multiple threads.
  - invoke(): Await to obtain the result; call the invoke method on the pool.
  - submit(): Returns a Future object that you can use for checking the status and getting the result on its completion.

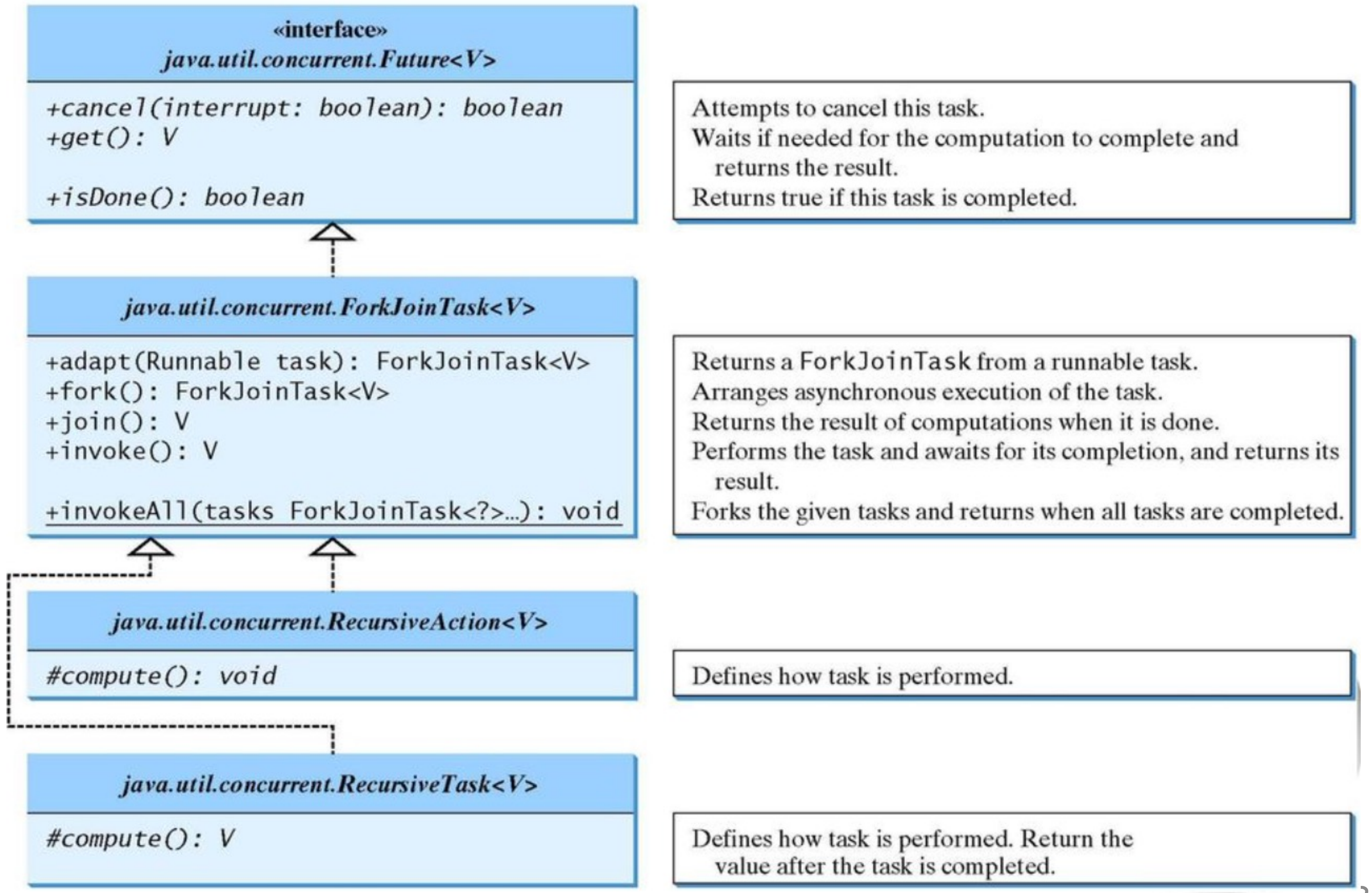
Fork/Join Framework
ForkJoinTask class

- ◇ ForkJoinPool class invokes a task of type ForkJoinTask, which you have to implement by extending one of its two 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 execution (launching a new subtask from an existing one).
  - join(): which returns the result of the computation when it is done, allowing a task to wait for the completion of another one.

Fork/Join Framework
ForkJoinTask class (1)

- ◇ The ForkJoinTask class provides several methods for checking the execution status of a task.
  - isDone(): returns true if a task completes in any way.
  - isCompletedNormally(): returns true if a task completes without cancellation or encountering an exception.
  - isCancelled(): returns true if the task was cancelled.
  - isCompletedabnormally(): returns true if the task was either cancelled or throw an exception.

Fork/Join Framework
ForkJoinTask class (1)


«interface»
java.util.concurrent.Future<V>
+cancel(interrupt: boolean): boolean
+get(): V
+isDone(): boolean
java.util.concurrent.ForkJoinTask<V>
+adapt(Runnable task): ForkJoinTask<V>
+fork(): ForkJoinTask<V>
+join(): V
+invoke(): V
+invokeAll(tasks ForkJoinTask<?>...): void
java.util.concurrent.RecursiveAction<V>
#compute(): void
java.util.concurrent.RecursiveTask<V>
#compute(): V

Attempts to cancel this task.
Waits if needed for the computation to complete and returns the result.
Returns true if this task is completed.

Returns a ForkJoinTask from a runnable task.
Arranges asynchronous execution of the task.
Returns the result of computations when it is done.
Performs the task and awaits for its completion, and returns its result.
Forks the given tasks and returns when all tasks are completed.

Defines how task is performed.

Defines how task is performed. Return the value after the task is completed.

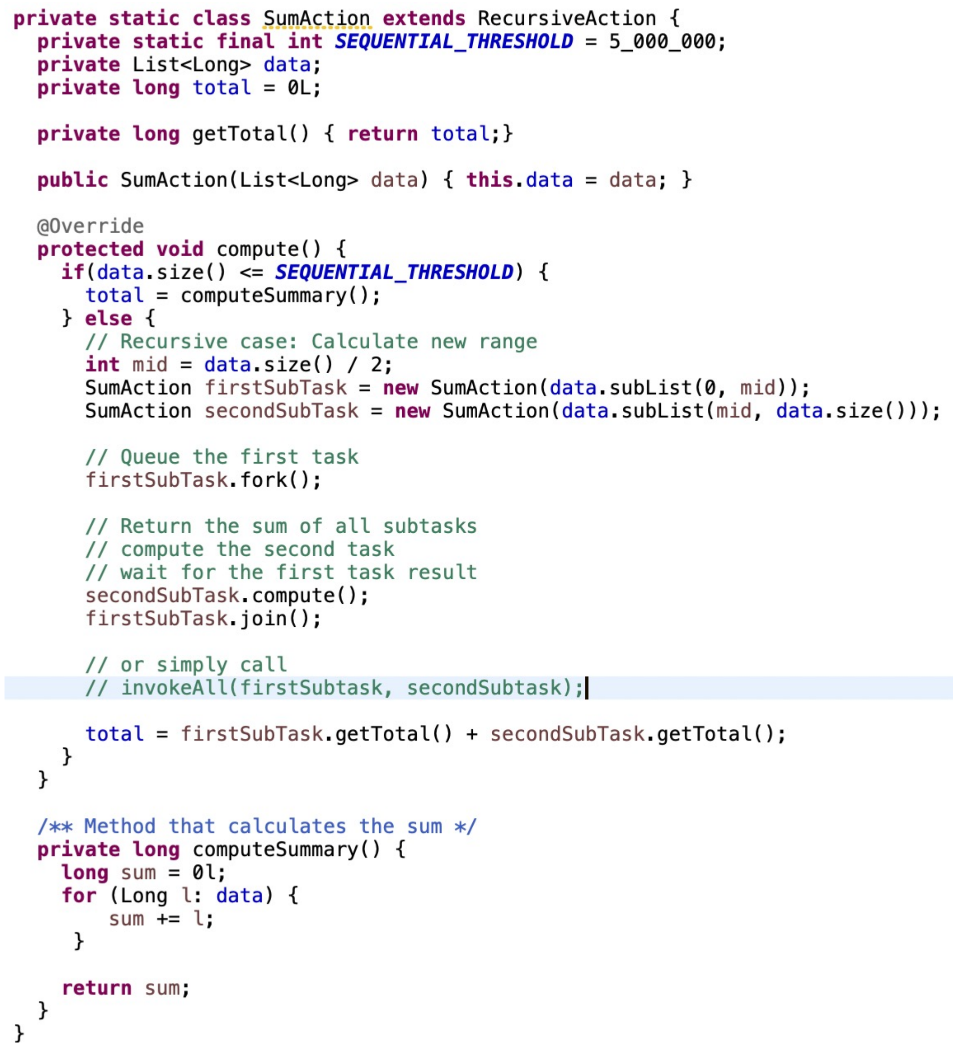
Fork/Join Framework
Standard setup code

1. Create a ForkJoinPool (create first thread)
   - ForkJoinPool pool = ForkJoinPool.commonPool();
2. Do subclass RecursiveTask<V> or RecursiveAction
   - SumArray extends RecursiveAction
   - SumArray extends RecursiveTask<V>
3. Do override compute()
4. Do return a V from compute
5. Do call fork() -> Start a new thread
6. Do call join (which returns answer) -> Wait for a thread to finish
7. Do call compute() to hand-optimize
8. Do create a pool and call invoke
   - SumArray task = new SumArray(data);
   - pool.invoke(task);

Task Parallelism
Creating Tasks with Fork/Join FW

- ◇ Tasks are the most basic unit of parallel programming,
- ◇ Modern task-based approaches for parallel programming:
  - Task creation
  - Task terminations
  - And the "computation graph": work, span, ideal parallelism, …
- ◇ In this framework, a task can be specified in the `compute()` method of a user-defined class that extends the standard RecursiveAction class in the FJ framework.

Task Parallelism
Example - SumAction



```java
private static class SumAction extends RecursiveAction {
  private static final int SEQUENTIAL_THRESHOLD = 5_000_000;
  private List<Long> data;
  private long total = 0L;

  private long getTotal() { return total;}

  public SumAction(List<Long> data) { this.data = data; }

  @Override
  protected void compute() {
    if(data.size() <= SEQUENTIAL_THRESHOLD) {
      total = computeSummary();
    } else {
      // Recursive case: Calculate new range
      int mid = data.size() / 2;
      SumAction firstSubTask = new SumAction(data.subList(0, mid));
      SumAction secondSubTask = new SumAction(data.subList(mid, data.size()));

      // Queue the first task
      firstSubTask.fork();

      // Return the sum of all subtasks
      // compute the second task
      // wait for the first task result
      secondSubTask.compute();
      firstSubTask.join();

      // or simply call
      // invokeAll(firstSubtask, secondSubtask);

      total = firstSubTask.getTotal() + secondSubTask.getTotal();
    }
  }

  /** Method that calculates the sum */
  private long computeSummary() {
    long sum = 0l;
    for (Long l: data) {
        sum += l;
     }

    return sum;
  }
}
```

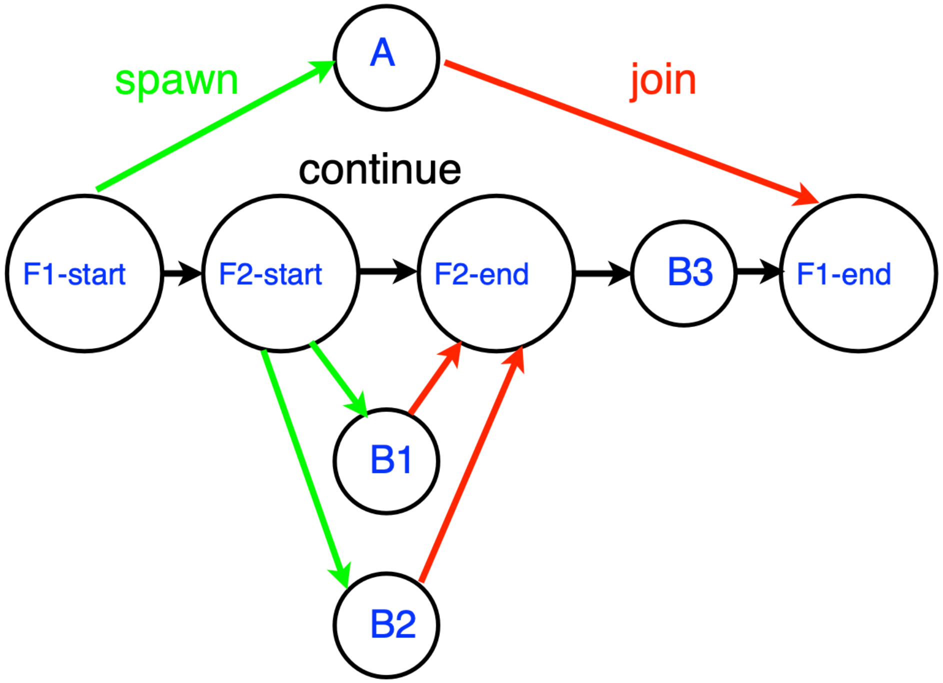
Task Parallelism
Computation Graphs

- A Computation Graph (CG) captures the dynamic execution of a parallel program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential sub-computation without any spawned, begin-finish or end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn/fork" edges connect parent tasks to child spawned tasks
  - "Sync/Join" edges connect the end of each spawned task to all join operations on that task.
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (DAGs)

Task Parallelism
Computation Graphs (1)

```
1. must finish { //F1
2.    spawn { A; }
3.    must finish { //F2
4.       spawn { B1; }
5.       spawn { B2; }
6.    } //F2
7.    B3;
8. } //F1
```

Computation Graph



Key idea: if two statements, X and Y have **no path of directed edges** form one to the other, then they can run in parallel with each other.

Functional Parallelism

- Will learn about futures, stream, …
- Learn Java API for functional parallelism including Fork/Join framework and the Stream API's

Functional Parallelism
Creating Tasks with Fork/Join FW

- ◇ Extends the RecursiveTask
- ◇ compute() method return type (non-void)
- ◇ A method call like `left.join()` waits for the task referred to by object `left` in both cases, but also provides the task's return value in the case of future tasks.

Functional Parallelism Example SumTask

```java
class SumTask extends RecursiveTask<Long> {
  private static final int SEQUENTIAL_THRESHOLD = 5_000_000;

  private List<Long> data;

  public SumTask(List<Long> data) {
    this.data = data;
  }

  @Override
  protected Long compute() {
    if(data.size() <= SEQUENTIAL_THRESHOLD) {
      long sum = computeSummary();
      System.out.format("Sum of %s: %d\n", data.toString(), sum);
      return sum;
    } else {
      // Recursive case
      // Calculate new range
      int mid = data.size() / 2;
      SumTask firstSubTask = new SumTask(data.subList(0, mid));
      SumTask secondSubTask = new SumTask(data.subList(mid, data.size()));

      // Queue the first task
      firstSubTask.fork();

      // Return the sum of all subtasks
      // compute the second task
      // wait for the first task result
      return secondSubTask.compute() + firstSubTask.join();
    }
  }
```

Functional Parallelism
Java Stream

- Java Stream can provide a functional approach to operating on collections of data.
- As an example, the following pipeline can be used to compute the average age of all active students using Java streams:

```
students.stream()
        .filter(s -> s.checkIsCuurent())
        .mapToDoblue(a -> a.getAge())
        .average();
```

Functional Parallelism
Java Stream

◇ An important benefit of using Java streams when possible is that the pipeline can be made to execute in parallel by designating the source to be a parallel stream:

- simply replacing:

  `students.stream()`

- in the above code by:

  `students.parallelStream()`

  or

  `Stream.of(students).parallel()`

Loop Parallelism
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);
}
-> forall (i : [0:n-1]) a[i] = b[i] + c[i]
```

- 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]);
```

FAQ

That's all for this session!

Thank you all for your attention and patient !