Parallel Function Evaluator

Note

- This is an individual assignment. Use of ChatGPT is allowed, but it **MUST** be specified as inline comments to prevent being misinterpreted as plagiarism. Cheating and plagiarism will be caught and punished HEAVILY! https://registry.hkust.edu.hk/resource-library/what-happens-if-you-are-caught-cheating
- Download the project skeleton from Canvas and complete the tasks based on it. Run tests with `gradle test`.
- The due time is 11:59 pm on **November 30**. Compress only the `src` folder in the project directory as a zip file and submit it on Canvas. Write the Github repository link on the Canvas submission comment as in lab assignments.
- The Github repository should be private and have the TA account (https://github. com/COMP3021-2023Fall) invited as a collaborator. The Github repository should have at least three commits separated in two days to prevent obvious plagiarism. The final commit on the repository should contain at least the same files as the submitted files. Failing this will result in 20% mark penalty on top of other penalties.
- Late submissions (regardless of technical issues such as Canvas or Github downtime) will be penalized by 20% of the total attainable marks in this assignment per day. This assignment accounts for 25% of the total course grade.
- Email to cklamaq@connect.ust.hk if you have any questions. You should also CC sliew@connect.ust.hk and yzhongbd@connect.ust.hk in your email.

1 Prerequisites, Goals, and Outcomes

1.1 Prerequisites

For this programming assignment, you should have learned:

- Knowledge of functional programming: lambda expressions, method references, and functional interfaces.
- Understanding of event-driven programming: event handling, event listeners, and event sources.
- Familiarity with parallel and concurrent programming: threads, synchronization, and thread-safe data structures.

1.2 Goals

1. Develop efficient parallel and concurrent programs using Java's built-in tools and libraries.
2. Implement event-driven programming techniques to decouple data dependencies.

3. Apply functional programming concepts and lambda expressions in Java to simplify code and improve readability.

1.3 Outcomes

Upon completion, you should be able to:

1. Design and implement Java code using functional programming concepts, such as lambda expressions and method references.
2. Create event-driven applications by effectively handling events, registering event listeners, and managing events.
3. Develop parallel and concurrent programs using threads, synchronization techniques, and thread-safe data structures to optimize performance and resource utilization.

2 Task Description

In this assignment, you will design and implement a simple parallel function evaluator. In computing, it is common to have many tasks (pure functions in our case) that depends on the output of some other tasks. These tasks together form a dependency graph, and tasks that are not dependent on each other can be executed in parallel.

As an overview, the system consists of: function data dependency graph node (Section 2.2), sequential function evaluator (Section 2.3), parallel task pool (Section 2.4), parallel function evaluator (Section 2.5), parallel array task (Section 2.6).

We have provided skeleton for the project. You will be asked to fill in some methods whose signatures are already defined in the skeleton. Feel free to define additional methods as needed.

When in doubt, look at the tests that use the code before asking questions.

2.1 Base Score (20 marks)

If your program can compile, you will automatically get 20 base score. However, to encourage the use of lambda functions and avoid introducing unnecessary synchronization, marks will be deducted from the base score if you

1. Defined additional class attributes/classes. (-5 per attribute/class defined)
2. Used mutable local variable. (-5 per mutable local variable) Note that loop variables are also considered mutable local variables. Try to use the `stream` API as much as possible.
3. Used more than 6 `synchronized` keywords. (-5 per additional `synchronized`)

Note that the base score is non-negative, so even if you use 20 `synchronized` keywords, you base score can only go to 0.

You should not use `synchronized` keyword to your custom methods. Only add them to pre-defined methods.

Basically, we have a list of provided methods for you to fill in the implementation. You have to decide which methods need the synchronized keyword. If you missed some of the methods,

marks will be deducted (Section 2.4 and Section 2.5). If you added too many, marks will also be deducted (quota mentioned above).

The `synchronized` keyword is needed when the same object/variable is accessed by multiple threads, where one of the accesses is a *write* access. For example, if you run both the `set` and `get` method with the same `ArrayList`, you need to synchronize the accesses. However, if both accesses are `get`, you *don't* have to synchronize them. This means that if you add `synchronized` keywords appropriately, your program will be data race free. Note that data race freedom does *not* imply your program is correct when run in parallel. You still have to make sure your use of semaphore, for example, is correct under various schedules.

Examples

```
public class SeqEvaluator<T> implements Evaluator<T> {
  // This is predefined in the skeleton, so this is fine
  private HashMap<FunNode<T>, List<Consumer<T>>> listeners = new HashMap<>();

  // -5 because you cannot add fields to classes (rule 1)
  private ArrayList<T> something;

  // Custom method is OK
  private ArrayList<T> somethingelse() {
    // ...
  }

  // the rest of the skeleton
}

// assume this is in the skeleton...
public class Foo {
  // If this was not in the skeleton:
  // -5 because you cannot add custom classes (rule 1)
  private class InnerClass {
    // ...
  }

  private void foobar() {
    // It is fine to use local variables, as long as you don't mutate them
    int a = 0;

    // -5 because loop variables are mutable (rule 2)
    for (int i = 0; i < 10; i++)
      foo();
  }
}
```

API Hints

Here are some APIs that may be useful for you.

- `Optional<T>`: There are two kinds of values: `Optional.of(v)` and `Optional.empty()`. The former one means that we have a value `v` of type `T`, and the latter one means we don't have a value here. This is an alternative to using `null`, and has the advantange of letting the type system check for null for you. There are several methods that are useful when dealing with optionals:

  - `Optional::get()`: Returns the value inside the optional, and throw an error if there is no such value.
    ```
    assert Optional.of(1).get(1) == 1;
    ```
  - `Optional::isEmpty()` and `Optional::isPresent()`: Returns if the optional is empty or contains a value.
    ```
    assert Optional.of(1).isEmpty() == !Optional.of(1).isPresent();
    assert Optional.of(1).isPresent();
    assert Optional.empty().isEmpty();
    ```
  - `Optional::ifPresent(action)`: Run `action` with the content of the optional when the optional contains something.
    ```
    // instead of this
    if (v.isPresent())
      action(v.get());
    // do this
    v.ifPresent(action);
    ```
- `IntStream.range(int start, int end)`: return an `IntStream` that starts at `start` and ends at `end` (exclusive). For example, the following two are equivalent:
  ```
  for (int i = 0; i < 5; i++) foo();
  IntStream.range(0, 5).forEach(i -> foo());
  ```
- `IntStream::boxed()`: turns an `IntStream` into `Stream<Integer>`, in which you can `map` to other types.
  ```
  // the result of the expression is Stream<Optional<Integer>>
  IntStream.range(0, 5).boxed().map(java.util.Optional::of)
  ```
- `Stream::reduce(op)` or `IntStream::reduce(op)`: Apply the binary operator `op` on the stream, and return an optional result (empty if the stream is empty).
  ```
  // ((1 + 2) + 3) + 4
  assert IntStream.range(1, 5).boxed().reduce((a, b) -> a + b).get() == 10;
  ```
- `Stream::collect(collector)`: Allows you to collect the stream into some collection. `java.util.stream.Collectors` contains various collector implementations.
  ```
  assert IntStream .range(1, 5).boxed().collect(Collectors.toList())
    .equals(List.of(1, 2, 3, 4));
  ```

2.2 Function Data Dependency Graph Node (10 marks)

A function data dependency graph (over values of type `T`) consists of nodes `FunNode<T>` and edges between nodes. A `FunNode<T>` represents a function with specific arity. The inputs and output of the function are cached in class attributes.

Tasks

You should implement the methods in `FunNode.java`. Test cases using `FunNode<T>` can be found in `SimpleScalar.java`.

You will get 10 marks if your program can pass the hidden test cases. There is no partial credit for this part.

Specification

1. The `inputs` field contains cached inputs. The size of this list should be equal to the arity of the function. Inputs not yet available are represented by `Optional.empty()`.
2. The `output` field contains the cached output. If the function is not yet evaluated, the `output` should be set to `Optional.empty()`.

3. The `f` field contains the Java function used for computing the result.
4. The `setInput` method sets the input `i` to `inputs`, and return `Optional.of(this)` when the function can be evaluated, i.e. all inputs are available. Otherwise, it returns `Optional.empty()`.
5. The `getResult` method returns the result of the current node.
6. The `eval` method evaluates the function and set the `result` field.

You may assume the following:

1. $0 \leq$ arity in the constructor.
2. $0 \leq i <$ arity in `setInput`.
3. `eval` is only called when the inputs are ready, i.e. after calling `setInput` for all the arguments.
4. `getResult` is only called after `eval`.
5. The function `f` will not modify its inputs.

2.3 Sequential Function Evaluator (20 marks)

In Section 2.2, we implemented the nodes for a function data dependency graph. Function evaluators implementing the `Evaluator` interface contains the edges of the data dependency graph, as well as evaluates all functions that are ready.

As a starter, we will now implement two sequential versions of the function evaluator. `SeqEvaluator` is the simplest version, which directly uses recursion to evaluate the graph. `SeqContEvaluator` uses an explicit queue instead of doing recursion, which allows it to evaluate long dependency chains without risking stack overflow.

Tasks

You should implement the methods in `SeqEvaluator.java` and `SeqContEvaluator.java`. Test cases can be found in `EvaluatorTest.java`.

If you want to test a certain evaluator, change the `EvaluatorTest::evaluators` function in `EvaluatorTest.java` to return a list containing only the evaluator that you want to test.

You will get 15 marks for each evaluator implementation if your implementation can pass the hidden test cases. There is partial credit if some hidden test cases fail, depending on the number of failures.

Specification

- `Evaluator`:
  - `public void addDependency(FunNode<T> a, FunNode<T> b, int i)`
    Register an edge of the data dependency graph. This sets the input `i` of `b` as the result of `a`.
  - `public void start(List<FunNode<T>> nodes)`
    Start evaluating nodes in `nodes`. Returns when all functions are evaluated.
- `SeqEvaluator`:
  - `HashMap<FunNode<T>, List<Consumer<T>>> listeners`
    Something used for implementing `addDependency`.
- `SeqContEvaluator`:

- `HashMap<FunNode<T>, List<Consumer<T>>> listeners`
  Something used for implementing `addDependency`.
- `ArrayDeque<FunNode<T>> queue`
  A queue that is used to store `FunNode<T>` that are not yet evaluated.

You may assume that
1. `addDependency` calls have valid `i`.
2. `start` calls have valid `nodes`, i.e. they have their inputs available and are not dependent on each other.
3. Evaluation order does not matter.
4. The dependency graph is acyclic.

Note that according to Section 2.1, you should not define additional attributes for the classes, or you will get your points deducted.

2.4 Parallel Task Pool (20 marks)

To facilitate the implementation of parallel function evaluator, we will implement a custom thread pool called `TaskPool`.

The pool contains a number of worker threads, and a parallel task queue. Each task is a `Runnable`, which is a function that takes no parameter and returns nothing. Users of the pool can submit tasks to the pool **in parallel** from different threads. The tasks are put into the task queue, and pick up by worker threads when they finished executing their previous task. Users can also add multiple tasks at once, and wait for all the tasks to complete, including tasks added when running other tasks.

Tasks

You should implement the methods in `TaskPool.java`, including methods in the nested class. You should also add the `synchronized` to functions whenever appropriate.

Test cases for `TaskPool.java` can be found in `TaskPoolTest.java`.

Points will be deducted when

1. Some hidden tests failed.
2. Missed `synchronized` in some methods. (-5 each)
3. `addTasks` method does not return after the pool turns idle. (-5)
4. `addTasks` method returns when the pool is still busy. (-5)
5. `terminate` fails to terminate all the threads. (-5)
6. `addTasks` method does not return after calling terminate. (-5)

Note that for 5 and 6, we assume the tasks are interruptible, e.g. running `Thread.sleep`.

Specification

- `TaskPool(int numThreads)`
  Initialize the task pool with `numThreads` worker threads.
- `void addTask(Runnable task)`
  Add one task to the queue for execution.

Note that this can be called while the task pool is busy, and from different threads as well as inside a task.

- `void addTasks(List<Runnable> tasks)`
  Add mutliple tasks to the queue for execution, and wait for the pool to become idle or terminate.

  Note that `addTasks` can be called multiple times throughout the lifetime of the task pool, but **not concurrently**.

- `void terminate()`
  Stop all threads and return after all threads are being joined.

  We will not call any method of the `TaskPool` after calling `terminate`.

2.5 Parallel Function Evaluator (20 marks)

Tasks

Using the parallel task pool implemented in Section 2.4, implement the parallel function evaluator in `ParEvaluator.java`. Test cases can be found in `EvaluatorTest.java`, similar to the sequential evaluators.

You should also add `synchronized` keyword to methods in `FunNode` as appropriate.

Points will be deducted when

1. Some hidden tests failed.
2. Missed `synchronized` in some methods. (-5 each) Hint: If you want to read the value set by other threads, you need `synchronized`.
3. Less than 2x speedup when run with 4 threads under appropriate settings. (-10)

Note: No points will be awarded if you use existing Java thread pools, such as `java.util.concurrent.Executors`.

2.6 Parallel Array Task (10 marks)

This bonus task is about using the thread pool above to implement two simple parallel array processing algorithms: parallel map and parallel prefix sum.

Parallel Map

A parallel map function is just to apply the map function to each element of the array in parallel. In practice, due to synchronization overhead, we typically split the array into chunks and each task will perform the map sequentially over that chunk.

Parallel Prefix Sum

Consider an input sequence $x_0, x_1, ...$, its (inclusive) prefix sum is the sequence

$$
\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2 \\
&\vdots
\end{aligned}
$$

In general, the summation operator can be replaced with any *stateless*, *associative* binary operators. There are multiple implementations of parallel prefix sum, and we shall discuss one approach that uses *two-level* scans.

Because the operator is associative, one can group the array elements in the following way:

$$y_{kc+i} = (x_0 + ... + x_{c-1}) + (x_c + ... + x_{2c-1}) + ... + x_{kc} + ... + x_{kc+i}$$

Define $z_i = x_{ic} + x_{ic+1} + ... + x_{(i+1)c-1}$. These z_i can be computed in parallel. Once these z_i are evaluated, we can compute the prefix sum of z_i, call it z'_i. By the previous equation, we know that $y_{kc+i} = z'_{k-1} + x_{kc} + ... + x_{kc+i}$. Note that c is a constant integer, representing the chunk size that we chose empirically.

In your implementation, you should implement the algorithm as follows:

1. Compute z_i in parallel.
2. Compute the prefix sum z'_i of z_i.
3. Compute y_i in parallel using the equation defined above.

Tasks

Implement `parMap` and `parInclusivePrefixSum` in `ArrayUtils.java`. Tests can be found in `ArrayUtilsTest.java`.

For parallel map, if your function takes 50% less time than the sequential variant when run under 4 threads, you will get 5 points.

For parallel prefix sum, if your function takes 50% less time than the sequential variant when run under 4 threads, you will get 5 points.

No point will be awarded if you use other Java parallelization APIs, such as `Arrays.parallelPrefix`.