Lab 4: Creating Parallelism using Futures

Instructors: Zoran Budimlić, Mack Joyner

Course Wiki: http://comp322.rice.edu

Staff Email: comp322-staff@mailman.rice.edu

Goals for this lab

- Creating parallelism using Futures, using Abstract Metrics for evaluation

Lab Projects

Today, we will solve a simple problem of computing a sum of reciprocal values of an array of doubles, by using futures to create parallelism. We will evaluate the performance in this lab using abstract metrics.

The GitHub classroom signup for this lab is located here:

https://classroom.github.com/a/Z_LpScWw

For instructions on checking out this repository through IntelliJ or through the command-line, please see the Lab 1 handout. The below instructions will assume that you have already checked out the lab4 folder, and that you have imported it as a Maven Project if you are using IntelliJ.

As in Lab 1, you will get a "WARNING: javaagent path does not equal Habanero-Java jar path." warning if you don't set up the javaagent path correctly to point to your local installation of the HJLib .jar file. As in Lab 1, resolve this by editing the run configuration and setting the correct -javaagent option in the VM. For Zoran's setup, this VM option looks like this:

```
-javaagent:"/Users/zoran/.m2/repository/com/github/RiceParProgCourse/hjlib-elog-aedancullen/
master-d2108184ec-1/hjlib-elog-aedancullen-master-d2108184ec-1.jar"
```

For your setup, it will look very similar, but with a different path.

Also, just as with Lab 1, don't forget to pick JDK11 as the JDK for the lab.

1 Abstract Performance Metrics

While Computation Graphs provide a useful abstraction for reasoning about performance, it is not practical to build Computation Graphs by hand for large programs. The Habanero-Java (HJ) library used in our course includes the following utilities to help programmers reason about the CGs for the programs they write:

- *Insertion of calls to* `doWork()`. The programmer can insert a call of the form `doWork(N)` anywhere in a future task code indicate execution of N units of work (application-specific abstract operations). Multiple calls to `doWork()` are permitted within the same future task. They have the effect of adding to the abstract execution time of that future task. The performance metrics will be the same regardless of which physical machine the HJ program is executed on, and provides a convenient theoretical way to reason about the parallelism in your program. However, the abstraction may not be representative of actual performance on a given machine, and measuring abstract metrics actually slows down your program.

In this lab, you will need to call `doWork(1)` for every double-precision addition you perform in your program.

- *Printout of abstract metrics.* If an HJlib program is executed with a specified option, abstract metrics are printed at the end of program execution that capture the total number of operations executed (*WORK*) and the critical path length (*CPL*) of the CG generated by the program execution. The ratio, *WORK*/*CPL* is also printed as a measure of *ideal parallelism*.

2 Parallelization of Reciprocal Array Sum using Futures

We will now work with the simple parallel array sum program introduced in the Demonstration Video for Topic 1.1. Edit the `ReciprocalArraySum.java` program provided in your GitHub repository. There are `TODO`s in the `ReciprocalArraySum.java` file guiding you on where to place your edits.

The goal of this exercise is to create an array of `N` random doubles, and compute the sum of their reciprocals in several ways, then compare the benefits and disadvantages of each. Performance in this lab will be measured using *abstract metrics* that accumulate *WORK* and *CPL* values based on calls to *doWork(1)*. The ways in which you will implement reciprocal sum are listed below:

- Sequentially in method seqArraySum().
- In parallel using **two** futures in method parArraySum2Futures(). It is important to add the calls to doWork(1) for every addition operation performed in your program on double values to keep track of abstract metrics. We only want you to call doWork(1) for additions on double values. Integer additions (for example loop indices or array indexing) and other double precision operations should be ignored. For the default input size, our solution achieved an ideal parallelism of *just under* 2.
- In parallel using **four** futures in method parArraySum4Futures(). You are essentially creating a version of parArraySum2Futures that uses 4 futures instead of 2. For the default input size, our solution achieved an ideal parallelism of *just under* 4.
- In parallel using **eight** futures in method `parArraySum8Futures()`. You are essentially creating a version of `parArraySum2Futures` that uses 8 futures instead of 2. You can probably recognize a pattern here, and instead of manually implementing code to create 2, 4, and 8 futures to solve these three problems, you should probably think of a way to generalize parallelization using `N` futures, where `N` is an arbitrary number, in order to avoid copying and pasting code. You will pass the test if you implement the first three problems manually, but if you implement a general parallelization function using `N` futures, the code for the first three problems will be straightforward calls to this function. For the default input size, our solution achieved an ideal parallelism of *just under* 8.
- In parallel with the goal of achieving *Maximum Parallelism*. Think about how to structure your code in order to execute as many as possible `doWork(1)` calls in parallel. You might be tempted to call your general parallelization function using `N` futures you wrote above (if you implemented it that way, of course) with `DEFAULT_N` or `DEFAULT_N/2` as the number of futures to use to compute the result, but that is probably not going to work. Think about how you can decompose the work needed to be done in order to achieve maximum parallelism. What would be the CPL of such a computation graph? How should you structure your future task creation in order to achieve this CPL?

  Our reference implementation achieved the ideal parallelism of 52428.75 for the default input size.
- When you complete your last task and achieve maximum parallelism, you will probably notice that the testReciprocalMaxParallelism test takes a lot longer to execute than the tests using 2, 4, or 8 futures. Can you explain why is that the case?

3 Turning in your lab work

For lab4, you will need to turn in your work before Wednesday, Feb 9, 2022 at 4:30PM, as follows:

1. Show your tests passing locally to an instructor or TA to get credit for this lab.
2. Commit and push your work to your `lab4` GitHub Classroom repository. The only changes that must be committed are your modifications to `ReciprocalArraySum.java`. Check that all the work for today's lab is in your `lab4` directory by opening https://classroom.github.com/a/Z_LpScWw in your web browser and checking that your changes have appeared.