Lab 1: Async-Finish Parallel Programming with Abstract Metrics

Instructor: Dr. Mackale Joyner

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

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

Goals for this lab

- Three HJlib APIs: `launchHabaneroApp`, `async`, and `finish`.
- Abstract metrics with calls to `doWork()`.

NOTE: The instructions below are written for Mac OS and Linux computers, but should be easily adaptable to Windows with minor changes e.g., you may need to use \ instead of / in some commands.

Note that all commands below are CaSe-SeNsItIvE. For example, be sure to use "S20" instead of "s20".

1 Lab 1 Exercises

1.1 HelloWorld program

The first exercise is to familiarize yourself with the kind of code you will see and be expected to write in your assignments. The `HelloWorldError.java` program does not have any interesting parallelism, but introduces you to the starter set for HJlib, which consists of three method calls[1]:

- `launchHabaneroApp()` Launches the fragment of code to be run by the Habanero runtime. All your code that uses any of the Habanero constructs must be (transitively) nested inside this method call. For example,

  ```
  launchHabaneroApp(() -> {S1; ...});
  ```

  executes S1, ..., within an implicit `finish`. You are welcome to add `finish` statements explicitly in your code in statements S1, ... While most assignments will not require that you write `launchHabaneroApp` explicitly (it will be included in the testing harness), it is good to be aware of.

- `async` contains the API for executing a Java 8 lambda asynchronously. For example,

  ```
  async(() -> {S1; ...});
  ```

  spawns a new child task to execute statements S1, ... asynchronously.

- `finish` contains the API for executing a Java 8 lambda in a finish scope. For example,

  ```
  finish(() -> {S1; ...});
  ```

  executes statements S1, ..., but waits until all (transitively) spawned `async`s in the statements' scope have terminated.

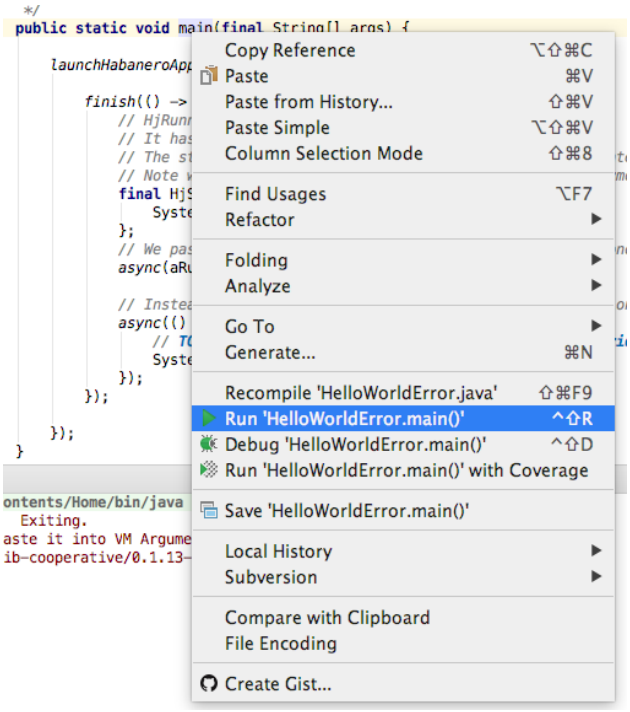
Uncomment line 44 in `HelloWorldError.java`. Compiling with your IDE or `mvn clean compile` should give you a compilation error similar to:

[1] Note that these and other HJlib APIs make extensive use of Java 8 lambda expressions.

```
HelloWorldError.java:[44,53] cannot find symbol
  symbol:   variable ss
  location: class edu.rice.comp322.HelloWorldError
```

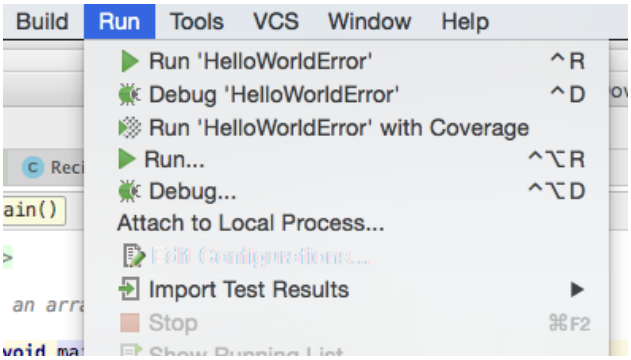
Your task is to fix this error. Replace "ss" by "s" in HelloWorldError.java and rebuild, verifying a successful compilation.

Next, we can try running the simple HelloWorldError project. From IntelliJ, that should be as simple as right-clicking on the `main` method and selecting Run:



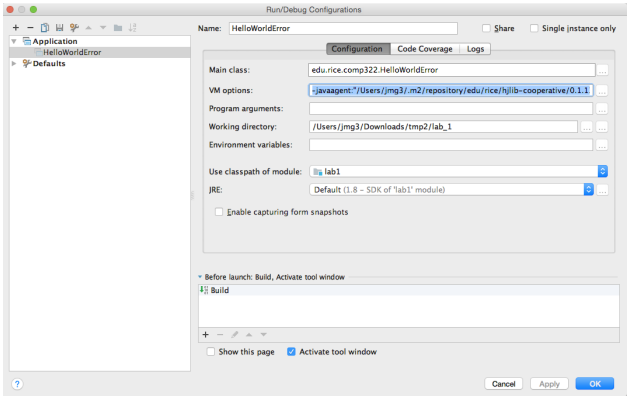
We expect this to produce some error output in the console:

HJlib requires what is called a "Java Agent" to be added to the command line when launching programs. If HJlib discovers during startup that no Java Agent has been provided, it will 1) print the above error message, and 2) place the needed command-line argument in your clipboard for convenience. In IntelliJ, the simplest way to resolve this for the HelloWorldError example is through Run → Edit Configurations...



In the popup, you can then paste the `-javaagent` from the error output into the VM options textbox and hit OK.

For Windows, the error may not show the `-javaagent` option in the output. You can paste the following: `-javaagent:"C:\Users\yourusername\.m2\repository\edu\rice\hjlib-cooperative\0.1.13-SNAPSHOT\hjlib-cooperative-0.1.13-20170111.022154-1.jar"`



If you try re-running HelloWorldError, the program should now complete successfully with two prints.

2 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 the course includes the following utilities to help programmers reason about the CGs for their programs:

- *Insertion of calls to* `doWork()`. The programmer can insert a call of the form `perf.doWork(N)` anywhere in a step to indicate execution of N application-specific abstract operations. Multiple calls to `doWork()` are permitted within the same step. They have the effect of adding to the abstract execution time of that step. 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.
- *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*.

3 ReciprocalArraySum Program

We will now work with the simple two-way parallel array sum program introduced in the Demonstration Video for Topic 1.1. Edit the `ReciprocalArraySum.java` program provided in your svn 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 comparing the benefits and disadvantages of each. As with Homework 1, 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** asyncs in method parArraySum_2asyncs(). It is important to add the calls to doWork() as seen in the seqArraySum() method to keep track of abstract metrics. For the default input size, our solution achieved an ideal parallelism of *just under* 2.
  - In parallel using **four** asyncs in method parArraySum_4asyncs(). You are essentially creating a version of parArraySum_2asyncs that uses 4 asyncs instead of 2. Think about the following question: How do you want to split up the work among the 4 tasks? For the default input size, our solution achieved an ideal parallelism of *just under* 4.
  - Lastly, in parallel using **eight** asyncs in method parArraySum_8asyncs(). You are essentially creating a version of parArraySum_2asyncs that uses 8 asyncs instead of 2. Think about the following questions: Do you really want to have to manually create 8 asyncs manually? Is there a better way you could write this function? Remember that copying and pasting code is generally discouraged. For the default input size, our solution achieved an ideal parallelism of *just under* 8.
- Compile and run the program in IntelliJ to ensure that the program runs correctly without your changes. Follow the instructions for "Step 4: Your first project" in https://wiki.rice.edu/confluence/pages/viewpage.action?pageId=14433124. If you're not using IntelliJ, you can do this by running the `mvn clean compile exec:exec -Preciprocal` command as specified in the README file.

  Be sure you **run the Lab1CorrectnessTest** file, not the ReciprocalArraySum file.
- Compare the abstract metric results and the actual speedup metric results and be able to explain the discrepancies before leaving lab. Note that the actual speedups depend on the input array size, which is 10^6 for today's lab, as well as the characteristics of your laptop.

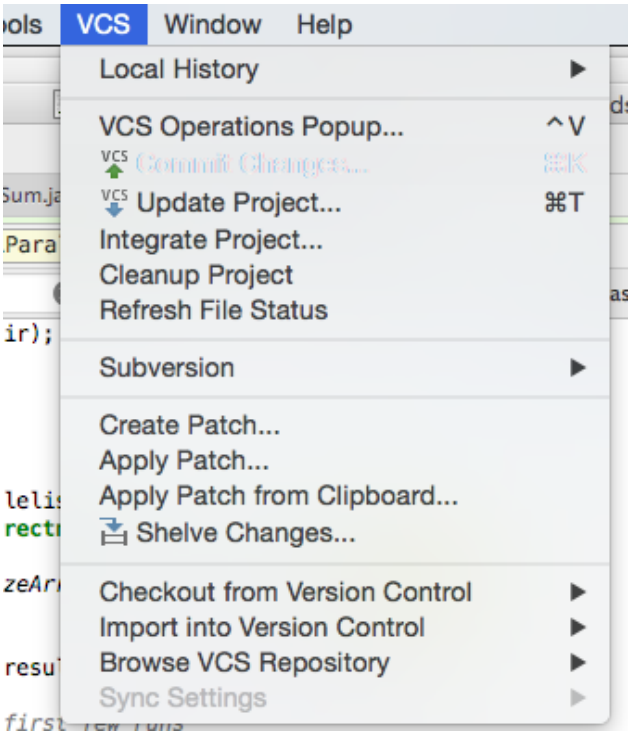
3.1 Submitting to your SVN repo

SVN supports committing changes from your local repo back to the SVN cloud. This can be down in IntelliJ or on the command line. If it's the first time you're committing the fine, you need to first perform an `svn add`. To commit from the command line, this is possible using the `svn commit` command from your project directory:

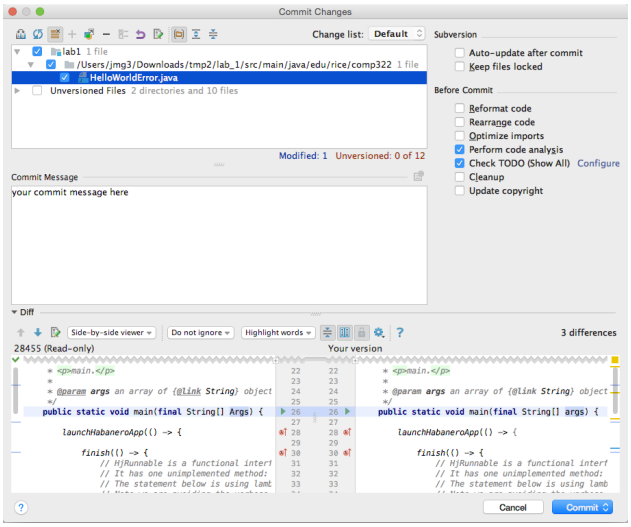
```
$ svn commit -m "your commit msg here"
```

Where "your commit msg here" can be any informational message you like.

From IntelliJ commits can be done through the VCS ¿ Commit Changes... selection:



The pop-up window will then allow you to fill in a commit message and preview the differences between the versions of the code in the cloud and on your laptop. After providing a commit message, hit Commit (and feel free to ignore any warnings for now).



You can confirm that your commit went through using your web browser. For example, by navigating to:

`https://svn.rice.edu/r/comp322/turnin/S20/NETID/lab_1/src/main/java/edu/rice/comp322/ReciprocalArraySum`

with NETID replaced by your Net ID, you should see an updated version of ReciprocalArraySum.java with your changes.

While the concept of SVN may be new to you, using `svn commit` to save your changes to the SVN server can be very useful. Frequently committing your code protects you from an accidental deletion or modification of your source blowing away days worth of work, as all changes will be saved in SVN. All of your commits to SVN are also visible to the teaching staff, and when asking for help on an assignment it can sometimes be simple to just point them to your code in SVN to ensure everyone is looking at the same version.

If you're having trouble running svn in IntelliJ on Windows, see if the following link helps to fix your problem: https://intellij-support.jetbrains.com/hc/en-us/community/posts/206320249-Checking-out-project-from-svn Try the uncheck the "use the command line" checkbox suggestion.

4 Demonstrating and submitting in your lab work

Show your work to an instructor or TA to get credit for this lab (as in COMP 215). They will want to see your updated files committed to Subversion in your web browser, and the passing/failing unit tests on your laptop. Labs must be checked off by a TA by the following Wednesday at 11:59pm.