Lab 1: Async-Finish Parallel Programming with Abstract Metrics

Instructors: Dr. Mackale Joyner and Dr. Zoran Budimlić

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()`.
- Autograder submission.

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 "S18" instead of "s18".

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.

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

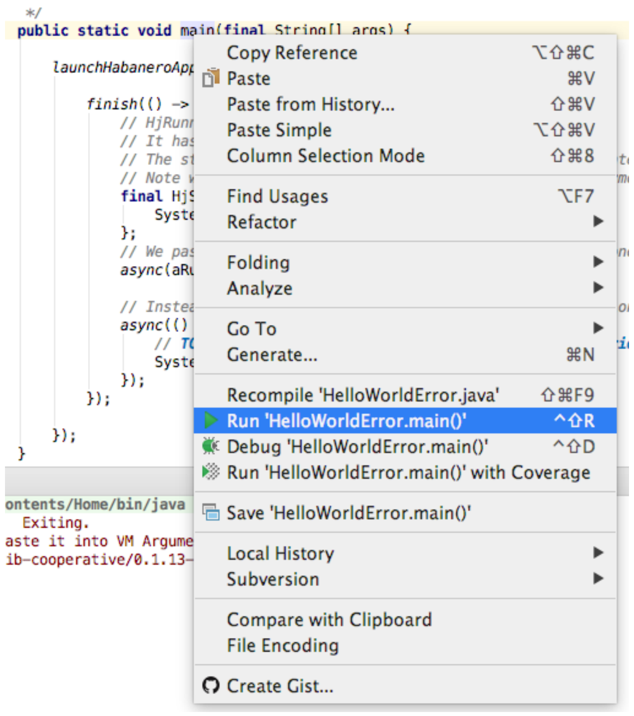
Run `mvn clean compile` from the top-level directory, or using IntelliJ to build it. On the command line you should receive an error from `HelloWorldError.java`:

```
[INFO] --- maven-checkstyle-plugin:2.17:check (validate) @ lab1 ---
[INFO] Starting audit...
HelloWorldError.java:24:8: error: Unused @param tag for 'args'.
HelloWorldError.java:26:44: error: Parameter name 'Args' must match pattern '^[a-z][a-zA-Z0-9]*$'.
```

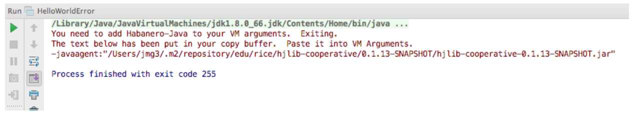
The above is a style error, reported by the Checkstyle tool. In COMP 322, you will be graded on the style of your code based on the number of errors reported by Checkstyle. Find the `Args` parameter to `main` and change it to `args`. Per Checkstyle, variable names should not start with a capital letter. Compiling again with your IDE or `mvn clean compile` should give you a new compilation error similar to:

```
cannot find symbol variable ss
```

Your next task is to fix this error, which is an inconsistency between the javadocs and method arguments. 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:


Copy Reference ⌥⇧⌘C
Paste ⌘V
Paste from History... ⇧⌘V
Paste Simple ⌥⇧⌘V
Column Selection Mode ⇧⌘8
Find Usages ⌥F7
Refactor
Folding
Analyze
Go To
Generate... ⌘N
Recompile 'HelloWorldError.java' ⇧⌘F9
Run 'HelloWorldError.main()' ^⇧R
Debug 'HelloWorldError.main()' ^⇧D
Run 'HelloWorldError.main()' with Coverage
Save 'HelloWorldError.main()'
Local History
Subversion
Compare with Clipboard
File Encoding
Create Gist...

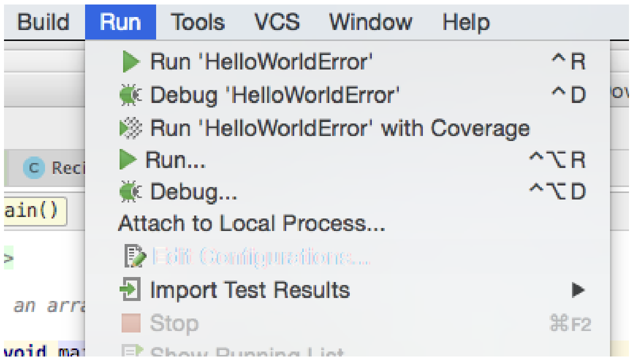
We expect this to produce some error output in the console:



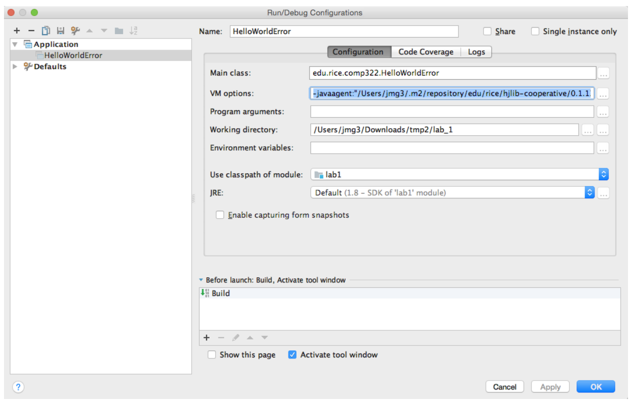
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java ...
You need to add Habanero-Java to your VM arguments.  Exiting.
The text below has been put in your copy buffer.  Paste it into VM Arguments.
-javaagent:"/Users/jmg3/.m2/repository/edu/rice/hjlib-cooperative/0.1.13-SNAPSHOT/hjlib-cooperative-0.1.13-SNAPSHOT.jar"

Process finished with exit code 255
```

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:



Name: HelloWorldError
Share
Single instance only
Main class: edu.rice.comp322.HelloWorldError
VM options: -javaagent:"/Users/jmg3/.m2/repository/edu/rice/hjlib-cooperative/0.1.1
Program arguments:
Working directory: /Users/jmg3/Downloads/tmp2/lab_1
Environment variables:
Use classpath of module: lab1
JRE: Default (1.8 – SDK of 'lab1' module)
Enable capturing form snapshots
Before launch: Build, Activate tool window
Build
Show this page
[✓ Activate tool window]

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 questions: How do you want to split up the work among the 4 tasks? Equally? Is this the best way? 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 the Habanero AutoGrader

In COMP 322 we will use an auto-grading system to test and evaluate lab and homework assignments. Today, you will do a test submission to the Habanero AutoGrader to become familiar with the submission process. The goal is not to produce a submission that passes all tests (in fact, the tests as written will intrinsically not pass when run through the Autograder) but to simply complete a submission. More information is available on the Autograder here.

You should have received an e-mail prior to the start of lab with your login credentials for the autograder. In a web browser, navigate to `http://ananke.cs.rice.edu`. We recommend using either Firefox or Chrome, IE may cause issues. Enter the provided login credentials and you should be greeted with an empty Overview page.

You will be submitting the completed Maven project that you used to complete the previous sections. The submission process consists of the following steps:

1. Create a zip file containing your full Maven project, including the src/ directory, README, and pom.xml. For example, you might do this from the file browser by creating a ZIP from the lab_1/ folder or on the command line by running `zip -r lab_1.zip lab_1`.

2. In your browser, select "COMP322-S18-Lab1" in the "Select Assignment" dropdown.
3. From the file dialog created by clicking the "Browse..." button, navigate to your .zip file and select it for upload to the autograder.
4. Finally, click the "Run" button and wait for your submission to be uploaded. Your run may take some time to complete, particularly if many students submit to the autograder at the same time. Be patient, but please alert a TA if you have been waiting for more than a few minutes or receive any error messages you do not understand.

Once this process is completed, you should see a new entry in the list of "Past Runs" on the Overview page. In the leftmost column is a unique run ID. The center column lists the assignment for the run. The rightmost column provides a status for the run. For today's lab, that status can be `TESTING CORRECTNESS`, `FAILED`, or `FINISHED`.

`TESTING CORRECTNESS` implies that your run is currently being processed. The autograder will test your submission with unit tests, run your submission through a style checker, and use static tools to check for bugs in your submission. For labs and homework, only the correctness, style, and performance of your submission may be counted towards your grade. The various static code checking tools the autograder supplies are purely for your benefit to help you improve your submission.

`FAILED` implies that a failure has occurred while processing your submission. This may indicate a compilation error, an unexpected exit condition from a third-party tool, or an internal error in the autograder. The autograder will not mark your submission as `FAILED` simply because a unit test failed, `FAILED` indicates that a more critical error prevented complete processing of your submission.

`FINISHED` indicates that your run has completed execution and the results can be viewed. To view the results, click on the hyperlink for that run in the "Run ID" column. Among other things, the opened page will list:

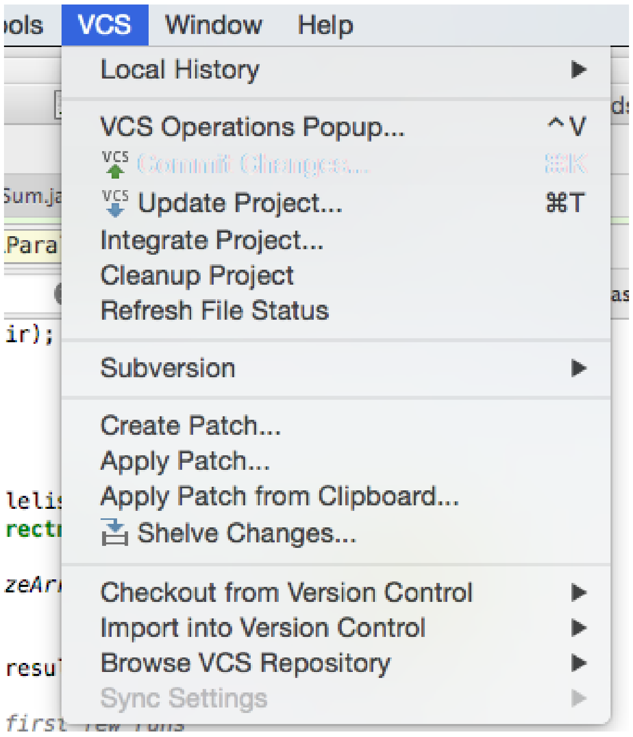
1. Your score and information on points deducted.
2. The output of the checkstyle tool when run on your program.
3. The output of compiling your program using Maven.
4. The output of your unit tests.
5. The output of the FindBugs static checker.

Recall that SVN also supports committing changes from your local repo back to the SVN cloud. On the command line, this is possible using the `svn commit` command from your project directory:

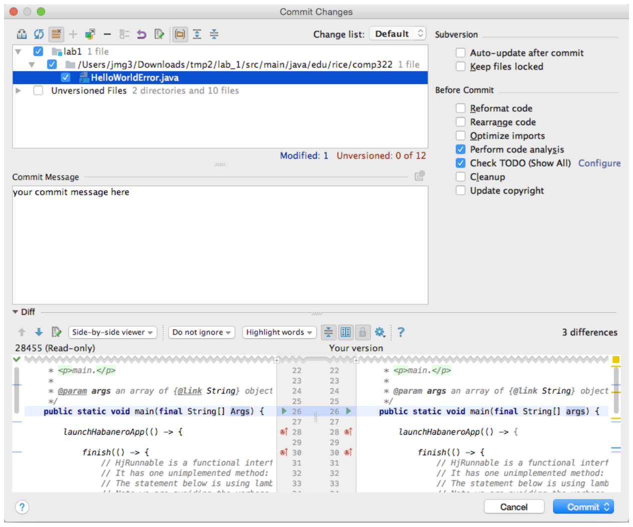
```
$ svn commit -m "your commit msg here"
```

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).


Commit Changes

Change list: Default

- [✓] lab1 1 file
  - [✓] /Users/jmg3/Downloads/tmp2/lab_1/src/main/java/edu/rice/comp322 1 file
    - [✓] HelloWorldError.java
- [ ] Unversioned Files 2 directories and 10 files

Modified: 1 Unversioned: 0 of 12

Commit Message

your commit message here

Subversion
- [ ] Auto-update after commit
- [ ] Keep files locked

Before Commit
- [ ] Reformat code
- [ ] Rearrange code
- [ ] Optimize imports
- [✓] Perform code analysis
- [✓] Check TODO (Show All) Configure
- [ ] Cleanup
- [ ] Update copyright

Diff

Side-by-side viewer | Do not ignore | Highlight words | 3 differences

28455 (Read-only)
```
     * <p>main.</p>
     *
     * @param args an array of {@link String} object
     */
    public static void main(final String[] Args) {

        launchHabaneroApp(() -> {

            finish(() -> {
                // HjRunnable is a functional interf
                // It has one unimplemented method:
                // The statement below is using lamb
```
Your version
```
     * <p>main.</p>
     *
     * @param args an array of {@link String} object
     */
    public static void main(final String[] args) {

        launchHabaneroApp(() -> {

            finish(() -> {
                // HjRunnable is a functional interf
                // It has one unimplemented method:
                // The statement below is using lamb
```

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

`https://svn.rice.edu/r/comp322/turnin/S18/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.

The Autograder also supports submissions through SVN rather than using ZIP files, which many students find more convenient. The process for this is the same, except that you paste the SVN URL for your root project directory in the `SVN URL` textbox during submission, rather than using the ZIP upload. For example, for Lab 1 you would use the following URL:

```
https://svn.rice.edu/r/comp322/turnin/S18/NETID/lab_1
```

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.

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 or in the autograder UI. Labs must be checked off by a TA by the following Monday at 11:59pm.