Lab 1: Async-Finish Parallel Programming with Abstract Metrics

Instructor: Vivek Sarkar. Co-Instructor: Dr. Mackale Joyner

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

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

Goals for this lab

- Subversion integration
- Introduction to HJlib installation and setup
- Three HJlib APIs: `launchHabaneroApp`, `async`, and `finish`.
- Abstract metrics with calls to `doWork()`.
- Autograder submission.

Importants tips and links

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 "S17" instead of "s17".

1 Subversion Setup

Subversion is a software versioning and revision control system. A revision control system is a system that tracks incremental revisions of files. It manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data or examine the history of how your data changed.

Subversion implements the concept of a version control repository - the core storage mechanism for versioned data. Each of you has a private repository for COMP 322 allocated in a "cloud" hosted by Rice's subversion (svn) server, svn.rice.edu. During COMP 322 we will use your private repository to distribute code to you. You can always examine the most recent contents of your svn repository by visiting `https://svn.rice.edu/r/comp322/turnin/S17/`*your-netid. It is possible that your svn account is not properly set up as yet. If you are unable to access the above URL, please send email to helpdesk@rice.edu cc'ing comp322-staff@mailman.rice.edu and requesting that they fix your access. After that, you can ignore this section for now (till you get access) and move on to the next section.*

The svn repository is empty to begin with, but will be populated with folders for homeworks and labs. We have a strict naming convention for these folders — "`hw_1`", "`hw_2`'', ... for homeworks and "`lab_1`", "`lab_2`", ... for labs.

There are primarily three SVN commands you will need to be familiar with for COMP 322: `checkout`, `commit`, and `add`. There are many other commands; if you are curious a full overview can be found here.

An SVN `checkout` downloads files from the SVN cloud to your local laptop. For example, you will frequently use `checkout` in COMP 322 to download the provided source code template for labs and homeworks from your private SVN repository.

An SVN `commit` uploads files from a checked-out version of the SVN repo on your local laptop back to the SVN cloud. SVN `commit` is smart, in that it will only upload the changes you have made to those files. SVN

will automatically keep a version history of those changes for you so that you can look back through your commits to see the changes you have made over time. You will likely regularly use `commit` in COMP 322 to update the cloud SVN repo with changes you have made to your local copy.

However, SVN only keeps track of the files on your local machine that you ask it to. If you create a new file inside a local, checked-out SVN repo, SVN will not automatically commit it and any changes you have made to it back up to the SVN cloud. Instead, you will first have to SVN `add` it to inform SVN that there is a new file it should start uploading with your commits.

There are a few ways you can interact with your subversion repository:

1. You can integrate subversion with your individual IntelliJ projects; after the integration you can use IntelliJ's GUI to publish your data to svn. Further details are available in online tutorials at `https://www.jetbrains.com/idea/help/version-control-with-intellij-idea.html`. In particular, pay attention to the "Enabling Version Control" and "Common Version Control Procedures" articles.
2. Eclipse users can consider using the Subclipse plugin.
3. If you are familiar with subversion and have your own svn client on your local machine, you are welcome to use that instead to commit your files. A common standalone SVN client for Windows is Tortoise SVN.
4. Another option is to use the command line for svn, i.e. the `svn commit`, `svn checkout`, `svn add` commands. Please refer to SVN manual for more information on each command.

2 Lab 1 Exercises

The files for this lab should already be available in your subversion repo for this course. Please visit the url at `https://svn.rice.edu/r/comp322/turnin/S17/`*your-netid*`/lab_1` to ensure that the files are available. The directory structure for the lab should look like the following:

```
pom.xml
src
    main
     java
         edu
             rice
                 comp322
                     HelloWorldError.java
                     ReciprocalArraySum.java
    test
        java
            edu
                rice
                    comp322
                        Lab1CorrectnessTest.java
```

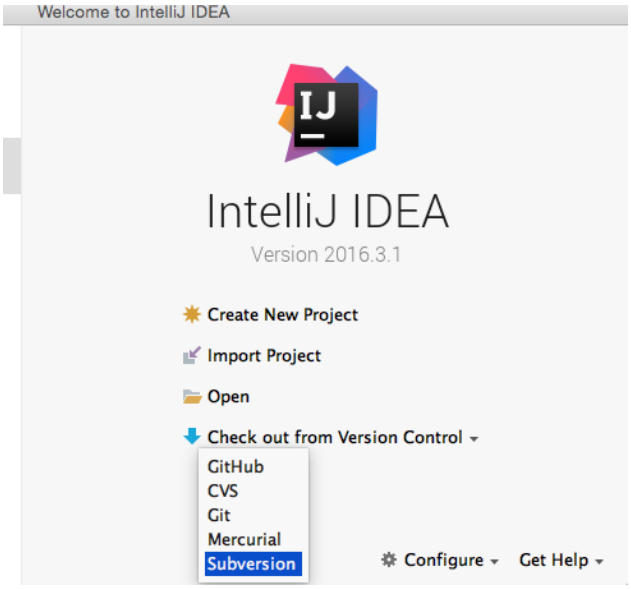
For this lab, you will be required to only edit the `HelloWorldError.java` and `ReciprocalArraySum.java` files.

You will need to download the project from the subversion server into your machine, set up the project on IntelliJ, and submit your changes back to the Rice subversion server. Note that you will need to have Java 8, Maven, and IntelliJ installed for this step. You can download the `lab_1` project on your machine through either the command line or using IntelliJ's SVN support. The two sections below contain instructions on both approaches, you only need to follow one.

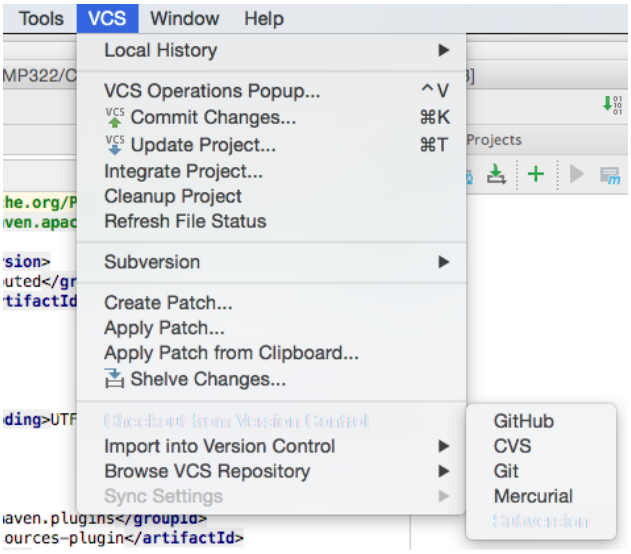
2.1 Using IntelliJ to checkout your SVN repo

Checking out your Lab 1 SVN folder from the remote SVN server is straightforward in IntelliJ.

You can start from either the "Check out from Version Control" option on the welcome screen:



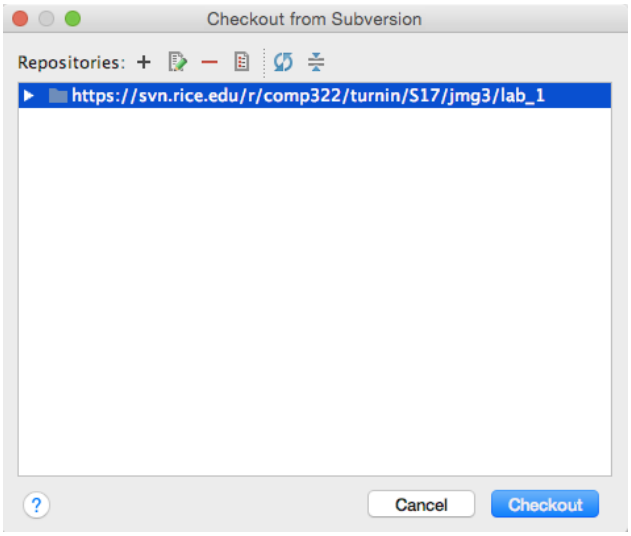
or use the "Checkout from Version Control" option accessible under the top menu's VCS tab:



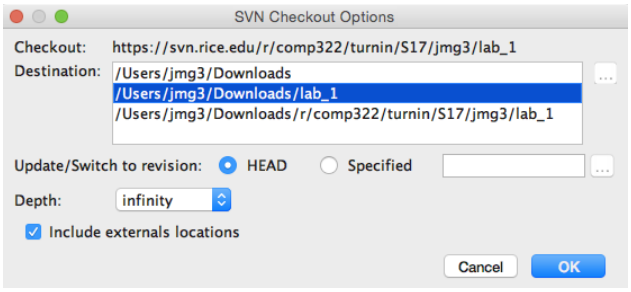
Use the plus button in the pop-up window to add the following SVN folder:

`https://svn.rice.edu/r/comp322/turnin/S17/NETID/lab_1`

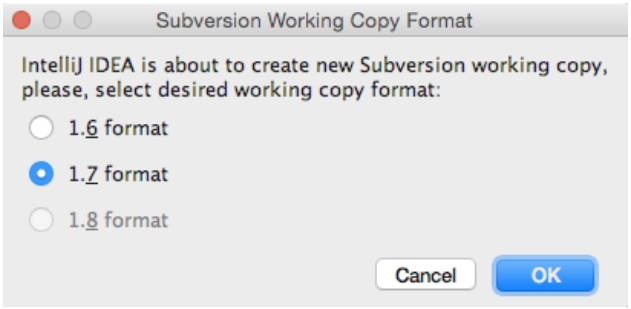
with NETID replaced by your Net ID. Click the Checkout button to start downloading this repo.



The next prompt will query you for the folder name to checkout the local copy of the SVN folder to. The choice of folder is up to you. Once you have made a choice, click OK.

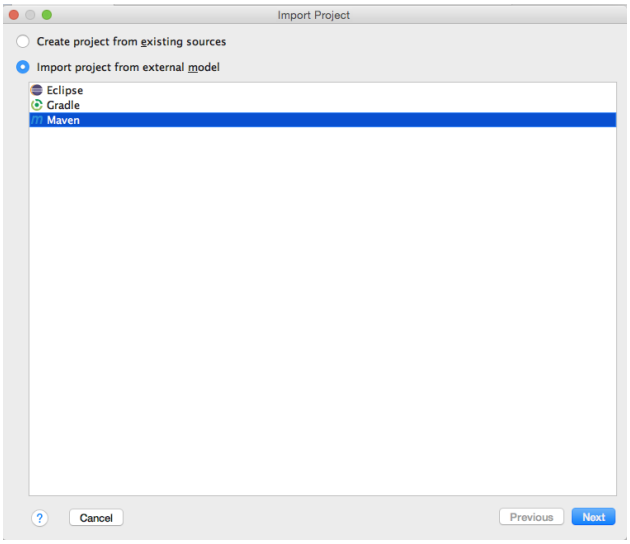


IntelliJ will likely then ask you to choose the SVN format to use for the local copy of the remote folder. We recommend you simply pick the latest option available and hit OK:



IntelliJ will now ask if you would like to open the newly checked-out project. Hit Yes:

On the next screen, you may be asked if you would like to import this project as a Maven project (your IDE may also skip this step and auto-detect it as a Maven project, in which case you can skip ahead too). By importing as a Maven project, IntelliJ will automatically download and manage dependencies for you:



Finally, IntelliJ should open your project and may ask you if you would "like to schedule the following file for addition to Subversion?". Select No. You are now ready to get started!

2.2 Using the command line to checkout your SVN repo

Note that you do not need to follow these instructions if you have already checked the project out through IntelliJ. These instructions are for those who would prefer to work on the command line.

The command-line SVN client can be installed by following the instructions at Apache Subversion Binary Packages. Once installed, you can verify that the command works by running the following command:

```
$ svn --version
  svn, version 1.8.11 (r1643975)
  compiled Jan  5 2015, 13:33:40 on x86_64-apple-darwin13.4.0
  ...
```

To checkout your SVN repo for Lab 1 on the command line, navigate in a terminal to the directory you would like to checkout into and issue the following command:

```
$ svn checkout https://svn.rice.edu/r/comp322/turnin/S17/NETID/lab_1
```

You should see a lab_1 directory created in your working directory with the source code for this lab.

2.3 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.

For details on all of your favorite HJlib APIs, we recommend also bookmarking the HJlib documentation website, hosted at `http://pasiphae.cs.rice.edu`.

Start by trying to compile the project by running `mvn clean compile` from the top-level directory, or using IntelliJ to build it. On the command line you should receive an error similar to the following 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]*$'.
```

In the IDE the error may be more subtle, just a highlighting of the above error as red text in HelloWorldError.java.

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. Consistent and clean style is an important part of writing production-level code in your future career, and a good practice to get started with early!

It should be straightforward to fix the above error in HelloWorldError.java by finding the `Args` parameter to `main` and changing it to `args`. In this case, Checkstyle was informing us that variable names should not start with a capital letter and reporting an inconsistency between the provided javadocs and the method arguments.

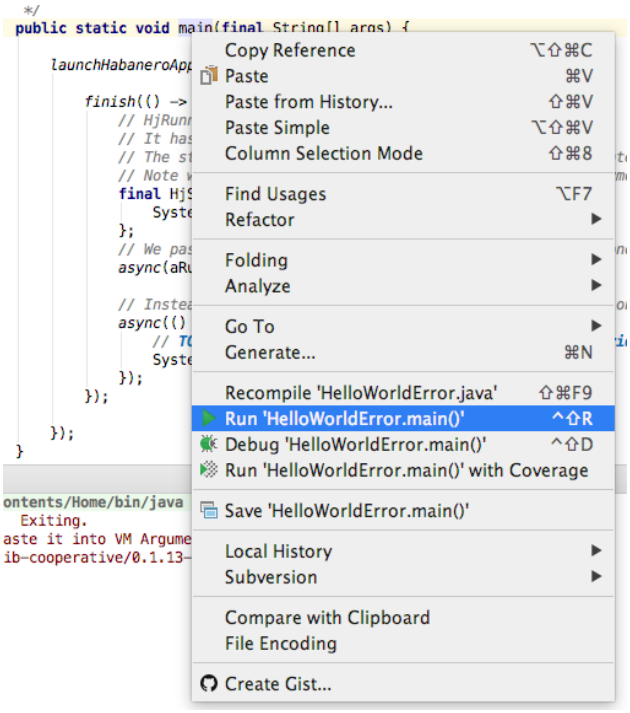
With that style error fixed, compiling again with your IDE or `mvn clean compile` should give you a new compilation error similar to:

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

```
cannot find symbol variable ss
```

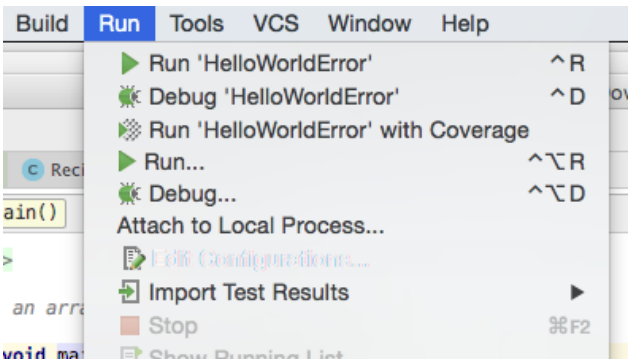
Your next task is to fix this error. You can replace "ss" by "s" in HelloWorldError.java and rebuild, verifying that you can now successfully compile the whole project.

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:

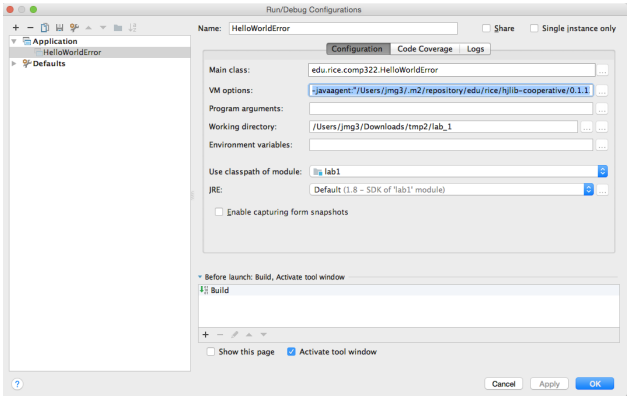


We expect this to produce some error output in the console:

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:



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

3 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 *e.g.,* floating-point operations, comparison operations, stencil operations, or any other data structure 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 main advantage of using abstract execution times is that 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. The main disadvantage is that 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*.

4 ReciprocalArraySum Program

We will now work with the simple two-way parallel array sum program introduced in the Demonstration Video for Topic 1.1. Your assignment is to edit the `ReciprocalArraySum.java` program provided in your svn repository for this exercise. 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.

4.1 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-S17-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:

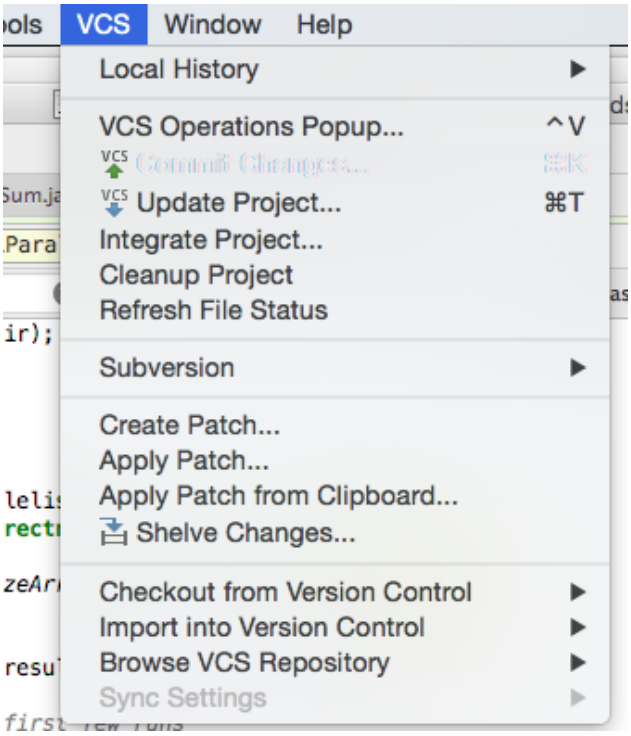
1. Your score and information on points deducted.
2. The output of the checkstyle tool when run on your program.
3. The output of compiling your program using Maven.
4. The output of your unit tests.
5. The output of the FindBugs static checker.

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:

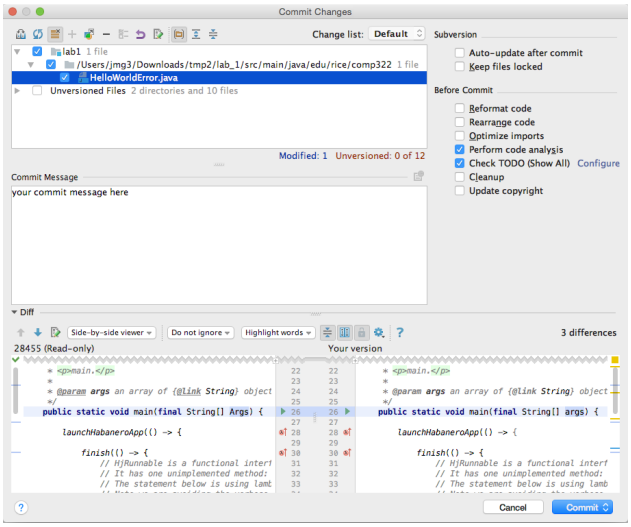
```
$ svn commit -m "your commit msg here"
```

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).



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

`https://svn.rice.edu/r/comp322/turnin/S17/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/S17/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.

5 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 prior to the start of the lab the following week.