Lab 6: Futures Vs Data-Driven Futures

Instructor: Vivek Sarkar

Course wiki : `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`

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

Goals for this lab

- STIC setup
- Experiment with Data-Driven Futures
- Experiment with Futures
- Understand the difference between DDF and Futures
- Compare running time for the sequential and parallel programs

Importants tips and links

edX site : `https://edge.edx.org/courses/RiceX/COMP322/1T2014R`

Piazza site : `https://piazza.com/rice/spring2015/comp322/home`

Java 8 Download : `https://jdk8.java.net/download.html`

Maven Download : `http://maven.apache.org/download.cgi`

IntelliJ IDEA : `http://www.jetbrains.com/idea/download/`

HJ-lib Jar File : `http://www.cs.rice.edu/~vs3/hjlib/code/maven-repo/habanero-java-lib/hjlib-cooperative/0.1.4-SNAPSHOT/hjlib-cooperative-0.1.4-SNAPSHOT.jar`

HJ-lib API Documentation : `https://wiki.rice.edu/confluence/display/PARPROG/API+Documentation`

HelloWorld Project : `https://wiki.rice.edu/confluence/pages/viewpage.action?pageId=14433124`

Lab Projects

The Maven project for this lab is located in the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S15/`*`NETID`*`/lab_6_ddfs_and_futures/`

Please use the subversion command-line client to checkout the project into appropriate directories locally. For example, you can use the following commands from a shell:

```
$ cd ~/comp322
$ svn checkout https://svn.rice.edu/r/comp322/turnin/S15/NETID/lab\_6\_ddfs\_and\_futures/ lab\_6
```

1 STIC Usage

As the previous lab, you are going to use STIC to run your programs. If you have not set up STIC, please refer to Appendix A for STIC set up instructions.

- Please note that when you are logged in STIC, you are logged in a `login node`. These nodes are intended for users to compile software and prepare data files. You should run your program on a `compute node` by submitting a job into the job queue. To specify the number of nodes you want to use, please change `myjob.slurm` file accordingly.
- If you have an out-of-memory error on STIC, you could decrease your input size. For this lab, you should not have this issue.
- After your job finished running, a `slurm-[job number].out` file will appear under the directory where you have submitted your job(that is, the directory under which you ran `sbatch myjob.slurm` command).
- You have to run the `source /home/smi1/dev/hjLibSource.txt` command `every time` you log in on STIC.
- `Important`: If you want to make changes locally and update these changes on STIC, you could check out the lab on your local machine and commit all changed files to svn. Then do a `svn up` from your lab folder on STIC.

2 Matrix Expression Evaluation Using Data-Driven Futures

2.1 Matrix Expression Language

We have provided a sequential program, `MatrixEval.java`, to evaluate matrix expressions consisting of the following terms and operators:

- The only leaf terms supported are identifiers which can be of two forms:

  Identity Matrix: An identifier of the form $m\langle num1\rangle$ represents a square identity matrix of size $\langle num1\rangle \times \langle num1\rangle$. For example, *m100* represents the 100×100 identity matrix. (The expression language has no variable declarations, so there's no significance to the name m other than the fact that it denotes a matrix.)

  Random Matrix: An identifier of the form $m\langle num1\rangle x\langle num2\rangle s\langle seed\rangle$ represents a random matrix of size $\langle num1\rangle \times \langle num2\rangle$, for which the elements are generated using `java.util.Random` starting with an integer (long) *seed*, and calling `nextInt()` to generate successive elements of the matrix. For example, *m100x200s5* represents the 100×200 random matrix generated using 5 as the initial seed.
- The + operator represents matrix addition. An exception is thrown if the matrices don't have the same dimension sizes i.e., if they are not conformable. Otherwise, the matrix sum is returned.
- The − operator represents matrix subtraction. An exception is thrown if the matrices don't have the same dimension sizes i.e., if they are not conformable. Otherwise, the matrix difference is returned.
- The $*$ operator represents matrix multiplication. An exception is thrown if the number of columns in the first matrix operand does not equal the number of rows in the second matrix operand i.e., if they are not compatible for matrix multiplication. Otherwise, the matrix product is returned.
- Usual precedence and evaluation rules apply for the above operators, and parentheses can also be used.

As an example, "*m3 + m3 * m3*", will be evaluated as follows:

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} + \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} = \begin{bmatrix}2 & 0 & 0\\0 & 2 & 0\\0 & 0 & 2\end{bmatrix}$$

2.2 Sequential Matrix Expression Evaluation

1. Download the lab6 files from the Code Examples column for Lab 6 in the course web page.The code in `MatrixEval.java` performs sequential evaluation of Matrix expressions presented in Section **??**. `test.txt` is an input file containing a simple expression in the matrix expression language.

2. Run the program on STIC. You can compile and run the sequential program as follows:

   ```
   mvn clean compile exec:exec -PmatrixSeq
   ```

   To run the program using 8 cores, change the parameter in myjob.slurm accordingly and submit the job to compute node(as described in the previous section). We just run the program on 8 cores for consistency with the procedure used for other results. We only use 1 core for sequential program.

3. Record in `lab_6_written.txt` the best execution time observed.

2.3 Parallelization of MatrixEval using Futures

The sequential code in `MatrixEval.java` parses the input expression, and then calls different `eval()` methods to evaluate unary and binary operators in the expression. The major potential for parallelism is in the `eval(HjDataDrivenFuture<MatrixEval.Matrix> d)` method in class `Binary`, shown in Listing **??**. Given the semantics of expression evaluation, the calls to `lft.eval(lft_ddf)` and `rgt.eval(rgt_ddf)` can execute in parallel.

Note that the sequential implementation uses `HjDataDrivenFuture` as a container to return the result of evaluating the expression. Your assignment for this section is to use futures with get() operations to parallelize the sequential program.

1. Write a parallel version of MatrixEval.java in MatrixEvalFuture.java using futures. You can replace HjDataDrivenFuture by HjFuture wherever needed.

2. Run the program on STIC. You can compile and run the sequential program as follows:

   ```
   mvn clean compile exec:exec -PmatrixFuture
   ```

   To run the program using 8 cores, change the parameter in myjob.slurm accordingly and submit the job to compute node(as described in the previous section).

3. Record in `lab_6_written.txt` the best execution time observed and the speedup w.r.t to MatrixEval.java

2.4 Parallelization of MatrixEval using Data-Driven Tasks

An alternative approach is to parallelize the evaluation of `lft` and `rgt` using data-driven tasks. The fact that the sequential version uses `HjDataDrivenFuture` containers will simplify this conversion.

1. Write a parallel version of MatrixEval.java in MatrixEvalDDF.java using data-driven tasks with calls to asyncAwait().

2. Run the program on STIC. You can compile and run the sequential program as follows:

   ```
   mvn clean compile exec:exec -PmatrixDdf
   ```

```
1  public void eval(HjDataDrivenFuture<MatrixEval.Matrix> d) {
2    HjDataDrivenFuture<Matrix> lft_ddf = newDataDrivenFuture();
3    HjDataDrivenFuture<Matrix> rgt_ddf = newDataDrivenFuture();
4    lft.eval(lft_ddf);
5    rgt.eval(rgt_ddf);
6    Matrix result = null;
7
8    switch (opr) {
9      case Lexical.plus:
10       result = MatrixEval.matrixAdd(lft_ddf.get(), rgt_ddf.get());
11       d.put(result);
12       break;
13     case Lexical.minus:
14       result = MatrixEval.matrixMinus(lft_ddf.get(), rgt_ddf.get());
15       d.put(result);
16       break;
17     case Lexical.times:
18       result = MatrixEval.matrixMultiply(lft_ddf.get(), rgt_ddf.get());
19       d.put(result);
20       break;
21     default:
22       error("Unhandled binary operator");
23   }
24 }
```

Listing 1: Sequential implementation of eval() method in class Binary

To run the program using 8 cores, change the parameter in myjob.slurm accordingly and submit the job to compute node(as described in the previous section).

3. Record in `lab_6_written.txt` the best execution time observed and the speedup w.r.t to MatrixEval.java and MatrixEvalFuture.java

3 Turning in your lab work

For each lab, you will need to turn in your work before leaving, as follows.

1. Check that all the work for today's lab is in the `lab_6_ddfs_and_futures` directory. If not, make a copy of any missing files/folders there. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.

2. Use the turn-in script to submit the `lab_6` directory to your turnin directory as explained in the first handout: *turnin comp322-S15:lab_6*. Note that you should *not* turn in a zip file.

   NOTE: Turnin should work for everyone now. If the turnin command does not work for you, please talk to a TA. As a last resort, you can create and email a `lab_6.zip` *file to comp322-staff@mailman.rice.edu.*

Appendix: STIC setup

STIC(Shared Tightly-Integrated Cluster) is designed to run large multi-node jobs over a fast interconnect. The main difference between STIC and CLEAR is that STIC allows you to gain access to compute nodes to obtain reliable performance timings for your programming assignments. On CLEAR, you have no control over who else may be using a compute node at the same time as you.

- Login to STIC.

  ```
  ssh ⟨your-netid⟩@stic.rice.edu
  ⟨your-password⟩
  ```

  Your password should be the same as the one you have used to login CLEAR. Note that this login connects you to a *login* node.

- After you have logged in STIC, run the following command to setup the JDK8 and Maven path.

  ```
  source /home/smi1/dev/hjLibSource.txt
  ```

  Note: You will have to run this command each time you log on STIC. You could choose to add the command in /.bash_profile so that it will run automatically each time you log in.

- Check your installation by running the following commands:

  ```
  which java
  ```

  You should see the following: `/home/smi1/dev/jdk1.8.0_31/bin/java`

  Check java installation:

  ```
  java -version
  ```

  You should see the following:

  ```
  java version "1.8.0_31"
  Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
  Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
  ```

  Check maven installation:

  ```
  mvn --version
  ```

  You should see the following:

  ```
  Apache Maven 3.1.1 (0728685237757ffbf44136acec0402957f723d9a; 2013-09-17 10:22:22-0500)
  Maven home: /home/smi1/dev/apache-maven-3.1.1
  Java version: 1.8.0_31, vendor: Oracle Corporation
  Java home: /home/smi1/dev/jdk1.8.0_31/jre
  Default locale: en_US, platform encoding: UTF-8
  OS name: "linux", version: "2.6.18-371.perfctr.el5", arch: "amd64", family:
  "unix"
  ```

- When you log on to STIC, you will be connected to a *login node* along with many other users. Once you have an executable program, and are ready to run it on the compute nodes, you must create a job script that uses commands to prepare for execution of your program. We have provided a script template in

  ```
  lab5/src/main/resources/myjob.slurm
  ```

Figure 1: myjob.slurm

```
1  #!/bin/bash
2  #SBATCH --job-name=lab5
3  #SBATCH --nodes=[TODO: No. of nodes you want to test your program on]
4  #SBATCH --ntasks-per-node=1
5  #SBATCH --mem=1000m
6  #SBATCH --time=00:30:00
7  #SBATCH --mail-user=[TODO:your email address]
8  #SBATCH --mail-type=ALL
9  #SBATCH --export=ALL
10 #SBATCH --partition=classroom
11
12 echo "My job ran on:"
13 pwd
14 echo $SLURM_NODELIST
15 echo $USER
16 echo $SHARED_SCRATCH
17 cat $SLURM_NODELIST
18 if [[ -d /home/$USER && -w /home/$USER ]]
19 then
20     cd /home/$USER/[TODO:Path to your lab5 folder]
21     source  /home/smi1/dev/hjLibSource.txt
22     java -version
23     mvn --version
24     mvn clean compile exec:exec -PmatrixDdf
25 fi
```

You only need to change the lines marked "TODO". For example, on line 3, change the TODO to the number of nodes you want to run your parallel program on.

- To submit the job, run the following command in the same directory where you put myjob.slurm(in this case, it was place under `lab5/src/main/resources/myjob.slurm`):

  ```
  sbatch myjob.slurm
  ```

  After you have submitted the job, you should see the following:

  ```
  Submitted batch job [job number]
  ```

- To check the status of a submitted job, use the following command:

  ```
  squeue -u [your-net-id]
  ```

- To cancel a submitted job, use the following command:

  ```
  scancel [job-id]
  ```

  When your job finished running, your should see an output file titled `slurm-[job-id].out` in the same directory where you have submitted the job.

- To transfer a project folder to STIC, you can use one of two methods:
  - Use Subversion: You can commit your local changes to SVN. Then you can checkout or update the project on your STIC account using one of the following:

    ```
    svn checkout https://svn.rice.edu/r/comp322/turnin/S15/NETID/lab_5_ddfs_and_futures/
    ```

    or, if you have already checked out the SVN project on your account,

    ```
    svn update
    ```

  - Use SCP: Use the following command on your local machine:

    ```
    scp -r [folder-name] [your-net-id]@stic.rice.edu:[path to the storage location]
    ```

    For example, if I have a folder named "lab5" on my local machine, and I want to store it in "./comp322" on STIC, I would type the following command:

    ```
    scp -r lab5 [net-id]@stic.rice.edu:./comp322
    ```