# Lab 5: Futures Vs Data-Driven Futures

Instructor: Vivek Sarkar

**Resource Summary**

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

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

**Clear Login:** ssh *your-netid*@ssh.clear.rice.edu and then login with your password

## Important tips and links:

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

**Piazza site :** `https://piazza.com/rice/spring2014/comp322/home`

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

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

**HJ-lib Jar File :** `http://www.cs.rice.edu/~vs3/hjlib/habanero-java-lib.jar`

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

**HelloWorld Project :** `https://wiki.rice.edu/confluence/display/PARPROG/Download+and+Set+Up`

**Sugar Login:** ssh *your-netid*@sugar.rice.edu and then login with your password

**Linux Tutorial** visit `http://www.rcsg.rice.edu/tutorials/`

As indicated earlier in a lecture, adding this call in your program before the call to `initializeHabanero()` will increase the limit on blocked threads:
**System.setProperty(HjSystemProperty.maxThreads.propertyKey(), "200");**

On SUGAR, JDK8 is already available at `/users/COMP322/jdk1.8.0` and HJ-Lib is already installed at `/users/COMP322/habanero-java-lib.jar`. Run the following command to setup the JDK8 path.

```
source /users/COMP322/hjLibSetup.txt
```

When you log on to Sugar, you will be connected to a *login node* along with many other users. To request a dedicated *compute node*, you should use the following command from a SUGAR login node:

```
qsub -q commons -I -V -l nodes=1:ppn=8,walltime=00:30:00
```

*Note that all commands below are CaSe-SeNsItIvE. For example, be sure to use "S14" instead of "s14".*

*IMPORTANT: please refer to the tutorial on Linux and SUGAR, before staring this lab. Also, if you and others experience long waiting times with the "qsub" command, please ask the TAs to announce to everyone that they should use ppn=4 instead of ppn=8 in their qsub command (to request 4 cores instead of 8 cores).*

# 1 Matrix Expression Evaluation Using Data-Driven Futures

## 1.1 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 $100 \times 100$ 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 $100 \times 200$ 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:

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

## 1.2 Sequential Matrix Expression Evaluation

1. Download the files `MatrixEval.java` and `test.txt` from the Code Examples column for Lab 5 in the course web page, `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`.The code in `MatrixEval.java` performs sequential evaluation of Matrix expressions presented in Section 1.1. `test.txt` is an input file containing a simple expression in the matrix expression language.

2. Run the program on SUGAR. You can compile the program as follows:

   ```
   javac -cp /users/COMP322/habanero-java-lib.jar MatrixEval.java
   ```

   To run the program using 8 cores, use the following command on a *compute node*:

   ```
   java -cp /users/COMP322/habanero-java-lib.jar:.  -Dhj.numWorkers=8 MatrixEval
   test.txt
   ```

   (We just run the program on 8 cores for consistency with the procedure used for other results. Since this is a sequential program, it will only use 1 core.)

3. Record in `lab_5_written.txt` the best execution time observed.

```java
public void eval(HjDataDrivenFuture<MatrixEval.Matrix> d) {
  HjDataDrivenFuture<Matrix> lft_ddf = newDataDrivenFuture();
  HjDataDrivenFuture<Matrix> rgt_ddf = newDataDrivenFuture();
  lft.eval(lft_ddf);
  rgt.eval(rgt_ddf);
  Matrix result = null;

  switch (opr) {
    case Lexical.plus:
      result = MatrixEval.matrixAdd(lft_ddf.get(), rgt_ddf.get());
      d.put(result);
      break;
    case Lexical.minus:
      result = MatrixEval.matrixMinus(lft_ddf.get(), rgt_ddf.get());
      d.put(result);
      break;
    case Lexical.times:
      result = MatrixEval.matrixMultiply(lft_ddf.get(), rgt_ddf.get());
      d.put(result);
      break;
    default:
      error("Unhandled␣binary␣operator");
  }
}
```

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

### 1.3 Parallelization of MatrixEval using Data-Driven Tasks

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 1. 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 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 SUGAR. You can compile the program as follows:

   ```
   javac -cp /users/COMP322/habanero-java-lib.jar MatrixEvalDDF.java
   ```

   To run the program using 8 cores, use the following command on a *compute node*:

   ```
   java -cp /users/COMP322/habanero-java-lib.jar:.  -Dhj.numWorkers=8 MatrixEvalDDF
   test.txt
   ```

3. Record in `lab_5_written.txt` the best execution time observed and the speedup w.r.t to MatrixEval.java.

### 1.4 Parallelization of MatrixEval using Futures

An alternate approach to data-driven tasks is to use futures with get() operations instead of asyncAwait().

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 SUGAR. You can compile the program as follows:

   ```
   javac -cp /users/COMP322/habanero-java-lib.jar MatrixEvalFuture.java
   ```

   To run the program using 8 cores, use the following command on a *compute node*:

   ```
   java -cp /users/COMP322/habanero-java-lib.jar:.  -Dhj.numWorkers=8 MatrixEvalFuture
   test.txt
   ```

3. Record in `lab_5_written.txt` the best execution time observed and the speedup w.r.t to MatrixEval.java

## 2 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_5` 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_5` directory to your turnin directory as explained in the first handout: *turnin comp322-S14:lab_5*. 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_5.zip` *file to comp322-staff@mailman.rice.edu.*