Lab 5: Tiny Matlab

Instructor: Vivek Sarkar

1 Matrix Expression Language

We have provided a sequential program, `MatrixEval.hj`, 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⟨num1⟩* represents a square identity matrix of size ⟨*num1*⟩×⟨*num1*⟩. 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⟨num1⟩x⟨num2⟩s⟨seed⟩* represents a random matrix of size ⟨*num1*⟩×⟨*num2*⟩, 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 Parallelization using Data-Driven Tasks

The code in `MatrixEval.hj` parses the input expression, and then calls the `eval()` methods to evaluate the expression. The major potential for parallelism is in the `eval()` method in class `Binary`, shown in Listing 1. Given the semantics of expression evaluation, the calls to `lft.eval()` and `rgt.eval()` can execute in parallel.

Your assignment today is to use the `async await` feature in HJ to parallelize the evaluation of these two calls using *data-driven tasks (DDTs)* and *data-driven futures (DDFs)* (Lecture 8). HJ's `DataDrivenFuture` class now accepts type parameters, so you can use the `DataDrivenFuture<Matrix>` type for DDFs in this assignment.

You should run your program on SUGAR, to evaluate the parallelization. As before, you can compile the program as follows, after repeating the setup from Lab 4:

```
hjc MatrixEval.hj
```

To run the program, use the following command on a compute node (obtained using the "qsub -I ..." command discussed in Lab 4):

```
hj -places 1:8 MatrixEval test.txt
```

where `test1.txt` is a text file containing the input expression. What speedups do you see with parallelization?

You're welcome to test your code with other input expressions, both for correctness (with small matrices) and for performance (with larger matrices). There is a `PrintMatrix()` method included that you may choose to use when debugging your code with small inputs such as `test0.txt`.

```
1         public MatrixEval.Matrix eval() {
2             switch (opr) {
3             case Lexical.plus:
4                 return MatrixEval.matrixAdd(lft.eval(), rgt.eval());
5             case Lexical.minus:
6                 return MatrixEval.matrixMinus(lft.eval(), rgt.eval());
7             case Lexical.times:
8                 return MatrixEval.matrixMultiply(lft.eval(), rgt.eval());
9             default:
10                error("Unhandled binary operator");
11            }
12            return null;
13        }
```

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