Lab 6: Barriers, Data-Driven Tasks

Instructor: Vivek Sarkar

Resource Summary

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

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

Coursera Login: visit `http://rice.coursera.org` and log in via Shibboleth

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

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

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

IMPORTANT: please refer to the tutorial on Linux and SUGAR from Lab 5, 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).

As in past labs, create a text file named `lab_6_written.txt` in the `lab_6` directory, and enter your timings and observations there.

1 One-Dimensional Iterative Averaging Example Revisited with Barriers

1. Download the `OneDimAveraging.hj` program from Lab 5 by typing the wget command, `wget http://www.cs.rice.edu/~vs3/downloads/OneDimAveraging.hj`.
2. The code in `OneDimAveraging.hj` performs the iterative averaging computation discussed in the lectures. This code performs a sequential version of the computation in method `runSeq()` and a parallel chunked `for-finish-forasync–for` version in method `runChunkedForkJoin`.
3. *Your task is to create a more efficient SPMD version of* `runChunkedForkJoin()` *by using a* `forall` *loop with a barrier (*`next`*) operation instead.* Call this version `runSPMD()`. See slide 8 in Lecture 13 for the general approach.
4. The input arguments for the main method in this program are as follows:
   (a) `tasks` = number of chunks to be used for chunked parallelism. The default value for `tasks` is `Runtime.getNumOfWorkers()`, which is the number of workers w specified with the "`-places` $1 : w$" option (default is $w = 8$ on SUGAR).
   (b) `n` = problem size. Iterative averaging is performed on a one-dimensional array of size `(n+2)` with elements 0 and `n+1` initialized to 0 and 1 respectively. The final value expected for each element i is $i/(n + 1)$. The default value for n is 1,000,000.
   (c) `iterations` = number of iterations needed for convergence. The default value is 2,000. This default was set for expediency. For this synthetic problem, you typically many more iterations to guarantee convergence.
   (d) `rounds` = number of repetitions for the entire computation. As discussed earlier, these repetitions are needed for timing accuracy. The default value is 3. For 3 repetitions, a reasonable approach is to just report the minimum time observed.

5. You should run your program on SUGAR, to evaluate the parallelization. As before, you can compile the program as follows:

   ```
   hjc OneDimAveraging.hj
   ```

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

   ```
   hj -places 1:8 OneDimAveraging
   ```

6. Record in `lab_5_written.txt` the best sequential and SPMD-parallel times observed for the default inputs (using 8 cores), and then compute their ratio as the speedup. Compare your results for runSPMD() with the results that you obtained in Lab 5 for runChunkedForkJoin().

2 Data-Driven Tasks

Download the following files to prepare for this section of the lab:

1. `wget http://www.cs.rice.edu/~vs3/downloads/MatrixEval.hj`
2. `wget http://www.cs.rice.edu/~vs3/downloads/test.txt`
3. `wget http://www.cs.rice.edu/~vs3/downloads/test0.txt`

2.1 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 $m\langle num1\rangle$ 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 Recap of Data-Driven Tasks

Data-driven tasks were covered in Lecture 13. To use this feature, be sure to include the following import statement at the start of your program: "`import hj.lang.DataDrivenFuture;`"

This extension is enabled by adding an `await` clause to the async statement as follows:

`async await` $(DDF_a, DDF_b, \cdots)$ ⟨ `statement` ⟩

Each of $DDF_a, DDF_b, \cdots$ is an instance of the standard `DataDrivenFuture` class in HJ. A DDF acts as a container for a single-assignment value, like regular `future` objects. However, unlike `future` objects, DDF's can be used in an `await` clause, and any async task can be a potential producer for a DDF (though only one task can be the actual producer at runtime because of the single-assignment property).

The example HJ code fragment in Figure 1 shows five logically parallel tasks and how they are synchronized through DDFs. Initially, two DDFs are created as containers for data items `left` and `right`. Then a `finish` is created with five `async` tasks. The tasks, `leftReader` and `rightReader`, include `left` or `right` in their `await` clauses respectively. The fifth task, `bothReader`, includes both both `left` and `right` in its `await` clause. Regardless of the underlying scheduler, the first two `asyncs` are guaranteed to execute before the fifth `async`.

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish { // begin parallel region
    async left.put(leftBuilder()); // Task1
    async right.put(rightBuilder()); // Task2
    async await (left) leftReader(left.get()); // Task3
    async await (right) rightReader(right.get());// Task4
    async await (left, right) bothReader(left.get(), right.get());  //Task5
} // end parallel region
```

Figure 1: Example Habanero Java code fragment with Data-Driven Futures.

2.3 Parallelizing MatrixEval 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 13). HJ's `DataDrivenFuture` class now accepts type parameters, so you can use the `DataDrivenFuture<MatrixEval.Matrix>` type for DDFs in this assignment.

WARNINGS:

1. *You may need to modify method call interfaces e.g., adding a DDF parameter to eval(), to complete this assignment.*
2. *Be sure to add "break;" statements in "switch" statements if needed.*

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? Enter your results in `lab_6_written.txt`.

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`.

```java
        public MatrixEval.Matrix eval() {
            switch (opr) {
            case Lexical.plus:
                return MatrixEval.matrixAdd(lft.eval(), rgt.eval());
            case Lexical.minus:
                return MatrixEval.matrixMinus(lft.eval(), rgt.eval());
            case Lexical.times:
                return MatrixEval.matrixMultiply(lft.eval(), rgt.eval());
            default:
                error("Unhandled binary operator");
            }
            return null;
        }
```

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

3 Turning in your lab work and quiz

As in previous labs, you will need to complete a quiz on Coursera and turn in your work before leaving, as follows:

1. Visit rice.coursera.org, select "Fundamentals of Parallel Programming" course, and take the Lab 6 quiz.
2. Check that all the work for today's lab is in the `lab_6` 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.
3. Before you leave, create a zip file of your work by changing to the parent directory for `lab_6/` and issuing the following command, "`zip -r lab_6.zip lab_6`".
4. Use the turn-in script to submit the contents of the `lab_6.zip` file as a new `lab_6` directory in your turnin repository as explained in Lab 1. You can always examine the most recent contents of your svn repository by visiting `https://svn.rice.edu/r/comp322/turnin/S13/`*your-netid*.