Homework 3: due by 12noon on Friday, March 18, 2016

Instructor: Vivek Sarkar, Co-Instructor: Shams Imam.

Checkpoint 1 due by 12noon on Friday, February 26, 2016

Checkpoint 2 due by 12noon on Friday, March 11, 2016

Total score: 100 points

All homeworks should be submitted in a directory named "hw_3" in your svn repository for this course. In case of problems committing your files, please contact the teaching staff at `comp322-staff@rice.edu` before the deadline to get help resolving for your issues. No late submissions will be accepted unless you are using your slip days.

The slip day policy for COMP 322 is similar to that of COMP 321. All students will be given 3 slip days to use throughout the semester. When you use a slip day, you will receive up to 24 additional hours to complete the assignment. You may use these slip days in any way you see fit (3 days on one assignment, 1 day each on 3 assignments, etc.). If you use slip days, you must submit a README.txt file in your svn homework folder before the actual submission deadline indicating the number of slip days that you plan to use.

Please note the deadlines for Checkpoints 1 and 2. Only the code (not the written report) for those checkpoints need to be submitted by the respective checkpoint deadlines. Slip days can be used for individual checkpoints if needed, following the slip day policy outlined above. The written report and and the final project are due by the final deadline for this homework.

Other than slip days, no extensions will be given unless there are exceptional circumstances (such as severe sickness, not because you have too much other work). Such extensions must be requested and approved by the instructor (via e-mail, phone, or in person) before the due date for the assignment. Last minute requests are likely to be denied.

If you see an ambiguity or inconsistency in any homework question, please seek a clarification on Piazza or from the teaching staff. If it is not resolved through those channels, you should state the ambiguity/inconsistency that you see, as well as any assumptions that you make to resolve it.

Finally, please note that the programming project for this homework is significantly more challenging than in the past homeworks. It is important for you to start early, and to meet the intermediate checkpoints to ensure that you are on track to complete the entire homework before the final deadline.

Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates, the teaching assistants and the professor, but you should never misrepresent someone else's work as your own. If you use any material from external sources, you must provide proper attribution.

1 Written Assignments (25 points total)

Submit your solutions to the written assignments as a PDF file named `hw_3_written.pdf` *in the hw_3 directory. Please note that you be penalized 10 points if you misplace the file in some other folder or if you submit the report in some other format.*

1.1 Future Tasks and Data-Driven Futures

1. (10 points) *Summarize the similarities and differences between futures and data-driven futures in HJ-lib.* In your summary, you should state if it is possible to create a race condition or a deadlock when accessing the value in the future container when using either construct.

2. (15 points) Consider the HJ-lib code fragment below that operates on an array of DataDrivenFutures. (Note that there are no `get()` operations in this example, so the only purpose of the `put()` operation is to synchronize with `await` clauses. For simplicity, we just use an empty string `""` as the object being put into a DataDrivenFuture.)

   Is it possible for any instance of the async statement in line 6 to be indefinitely blocked on its await clause? If so, explain how. If not, explain why not.

```
1.      // Assume n > 1
2.      HjDataDrivenFuture<Void>[][] A = new HjDataDrivenFuture[n][n];
3.      // Initialization
4.      for (int i = 0; i < n; i++)
5.        for (int j = 0; j < n; j++)
6.          A[i][j] = newDataDrivenFuture();
7.      // Main computation
8.      for (int i = 1; i < n; i++)
9.        for (int j = 0; j < n-1; j++)
10.         asyncAwait(A[i-1][j+1], () -> {
11.               . . .
12.           A[i][j].put(null);
13.         }); // asyncAwait
14.     // Boundary conditions
15.     for (int j = 1; j < n; j++) A[0][j].put(null);
16.     for (int i = 1; i < n-1; i++) A[i][n-1].put(null);
```

2 Programming Assignment (75 points)

2.1 Pairwise Sequence Alignment

In this homework, we will focus on the *pairwise sequence alignment* problem in evolutionary and molecular biology, and how parallelism can help in solving this problem. (This homework is adapted from Homework 7 from the Spring 2011 offering of COMP 182 by Prof. Luay Nakhleh.)

Let X and Y be two sequences over alphabet Σ (for DNA sequences, $\Sigma = \{A, C, T, G\}$). An *alignment* of X and Y is two sequences X' and Y' over the alphabet $\Sigma \cup \{-\}$, where X' is formed from X by adding only dashes to it, and Y' is formed from Y by adding only dashes to it, such that

1 $|X'| = |Y'|$ *i.e.,* X' and Y' have the same size,

2 there does not exist an i such that $X'[i] = Y'[i] = -$, and

3 X is a subsequence of X', and Y is a subsequence of Y'. A is a subsequence of B, if you can obtain B from A by adding a (possibly empty) prefix string and a (possibly empty) suffix string.

This alignment is also referred to as *global pairwise alignment* (as opposed to *local pairwise alignment*, which is used to align selected regions of sequences X and Y).

Sequence alignment helps biologists make inferences about the evolutionary relationship between two DNA sequences. Aligning two sequences amounts to "reverse engineering" the evolutionary process that acted upon the two sequences and modified them so that their characters and their lengths differ. As an example, one possible alignment of the two sequences $X = ACCT$ and $Y = TACGGT$ is as follows:

$$\begin{array}{lcccccc} X' & = & - & A & C & - & C & T \\ Y' & = & T & A & C & G & G & T \end{array}$$

As you may imagine, there may be multiple alignments for the same pair of sequences. For example, a trivial alternate alignment for X and Y is as follows:

$$\begin{array}{lcccccccccc} X'' & = A & C & C & T & - & - & - & - & - & - \\ Y'' & = - & - & - & - & T & A & C & G & G & T \end{array}$$

2.2 Scoring in Pairwise Sequence Alignment: Optimality Criterion

As discussed above, a number of alignments exist for a given pair of sequences; therefore, we define a *scoring scheme* that gives higher scores to "better" alignments. Once the scoring scheme is defined, we seek an alignment with the highest score (among all feasible alignments). For DNA, a scoring scheme is given by a 5×5 matrix M, where for $p, q \in \{A, C, T, G\}$, $M_{p,q}$ specifies the score for aligning p in sequence X' with q in sequence Y', $M_{p,-}$ denotes the penalty for aligning p in sequence X' with a dash in sequence Y', and $M_{-,q}$ denotes the penalty for aligning q in sequence Y' with a dash in sequence X'. Assuming $|X'| = |Y'| = k$, the score of the alignment is

$$\sum_{i=1}^{k} M_{X'[i],Y'[i]}. \tag{1}$$

For this assignment, we will assume the following scoring scheme: $M_{p,p} = 5$, $M_{p,q} = 2$ (for $p \neq q$), $M_{p,-} = -2$ and $M_{-,q} = -4$.

For this scoring scheme. the score of the (X', Y') alignment in Section 2.1 is

$$M_{-,T} + M_{A,A} + M_{C,C} + M_{-,G} + M_{C,G} + M_{T,T} = (-4) + 5 + 5 + (-4) + 2 + 5 = 9$$

and the score of the (X'', Y'') alignment is $4 \times M_{p,-} + 6 \times M_{-,q} = -32$.

2.3 Sequential Algorithm to compute the Optimal Scoring for Pairwise Sequence Alignment

In this problem, we introduce a sequential dynamic programming algorithm (called the Smith-Waterman algorithm) to compute the Optimal Scoring for Pairwise Sequence Alignment. For two sequences X and Y of lengths m and n, respectively, denote by $S[i, j]$, $0 \leq i \leq m$ and $0 \leq j \leq n$, the score of the best alignment of the first i characters of X with the first j characters of Y. The boundary values are, $S[i, 0] = i * M_{p,-}$ and $S[0, j] = j * M_{-,p}$. It has been shown that this optimal scoring can be defined as follows $\forall i, j \geq 1$:

$$S[i, j] = \max \begin{cases} S[i-1, j-1] + M_{X[i],Y[j]} \\ S[i-1, j] + M_{X[i],-} \\ S[i, j-1] + M_{-,Y[j]} \end{cases}. \tag{2}$$

The above definition directly leads to a sequential dynamic programming algorithm that can be implemented as shown in Listing 1. Assume that the input sequences are represented as Java strings, and the scoring matrix, S, is represented as a 2-dimensional array of size (*X.length()+1*) $\times$ (*Y.length()+1*). After the algorithm terminates, the final score is available in *S[X.length()][Y.length()]*.

The dependence structure of the iterations in Listing 1 is shown in Figure 1. The cells in the figure correspond to $S[i, j]$ values, and the arrows show the dependences among the $S[i, j]$ computations.

```
1  for ( i = 1; i <= xLength; i++)
2   for ( j = 1; j <= yLength; i++) {
3     int i = point.get(0);
4     int j = point.get(1);
5     char xChar = X.charAt(i-1);
6     char YChar = Y.charAt(j-1);
7     int diagScore = S[i-1][j-1] + M[charMap(xChar)][charMap(YChar)];
8     int topColScore = S[i-1][j] + M[charMap(xChar)][0];
9     int leftRowScore = S[i][j-1] + M[0][charMap(YChar)];
10    S[i][j] = Math.max(diagScore, Math.max(leftRowScore, topColScore));
11  }
12 int finalScore = S[xLength][yLength];
```

Listing 1: Sequential implementation of Smith-Waterman Algorithm for Optimal Scoring for Pairwise Sequence Alignment

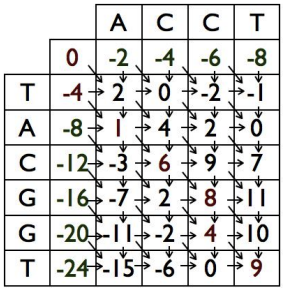

Figure 1: Dependences in Pairwise Sequence Alignment

This homework focuses on computing the optimal score for pairwise sequence alignment, not on the alignment itself. Though a biologist is ultimately interested in seeing the alignment, there are many applications where the score alone is of interest. For example, in multiple sequence alignment, the most commonly used approach is called progressive alignment, where an evolutionary tree is first built based on the scores of pairwise alignments, and then the tree is used as a guide for doing the multiple sequence alignment. In this case, the pairwise alignments are performed solely for the sake of obtaining scores, and the alignments themselves are not needed. However, it is important to compute the scores as quickly as possible when exploring alignments of large DNA sequences.

2.4 Your Assignment: Parallel Optimal Scoring for Pairwise Sequence Alignment

Your assignment is to design and implement parallel algorithms for optimal scoring for pairwise sequence alignment. We have provided a sequential implementation of the algorithm in `SeqScoring.java` that you can use as a starting point. Remember to commit all your source code into the subversion repository during your assignment submission. Your homework deliverables are as follows:

1. **[Checkpoint 1 due on February 26, 2016: Ideal parallelism with abstract execution metrics (10 points)]** Examine the dependence structure for $S[i, j]$ defined in Section 2.3 and create an ideal parallel version called `IdealParScoring.hj` that computes the same output as `SeqScoring.hj`, and delivers the maximum ideal parallelism ignoring all overheads. For analysis of ideal parallelism, assume that computing a single element of $S[i, j]$ takes one unit of time, *e.g.,* by inserting a call to doWork(1) between lines 9 and 10 of Listing 1. You will need to insert this doWork() call at the appropriate location in the parallel solution you create in `IdealParScoring`. Evaluate your code using HJlib's abstract metrics. We recommend testing your solution on pairs of strings of length ≤ 10 for this part

of the assignment. These abstract metric costs will be used in Homework 1, and will not include the async spawn cost from Homework 2.

For this checkpoint, we have provided a small set of correctness tests in `Homework3Checkpoint1CorrectnessTest.java`. Additional tests will be provided on the Autograder.

2. **[Checkpoint 2 due on March 11, 2016: Useful parallelism on NOTS compute nodes (20 points)]** Create a new parallel version of `SeqScoring.java` that is designed to achieve the smallest execution time using 16 cores on a dedicated NOTS compute node. Note that this is real execution time, not abstract metrics. Your code for this part will need to go into the `UsefulParScoring.java` file.

   For this part of the assignment, we recommend first debugging your solution on small strings for correctness (which can be done on any platform), and then evaluating the performance of your implementation with pairs of strings of length $O(10^4)$ on dedicated NOTS compute nodes. Lab 5 provided instructions on submitting jobs to the NOTS cluster.

   Even though each NOTS compute node has 32GB (or more) of memory, we will evaluate all homeworks using a maximum heap size of 8GB. The JVM heap size for tests running on NOTS is set in the provided pom.xml. Your submission will be evaluated with 8GB of heap, so changing this value in your pom.xml may result in incorrect test results. If you are running the JUnit tests locally through IntelliJ rather than using the provided pom.xml, you will want to add the following JVM command line argument to your Run Configurations to ensure that the JVM launched by IntelliJ is allowed to allocate up to 8GB of memory (in addition to the `-javaagent` argument that should already be there): `-Xmx8192m`

   However, note that most laptops do not have 8GB of physical memory, so running some of the larger tests locally may be prohibitively slow as your machine swaps memory pages out to disk as you exceed physical memory capacity.

   NOTE: a solution to the sparse memory version below is also acceptable as a solution for Checkpoint 2. However, we feel that many students will benefit from first completing Checkpoint 2 for a dense memory version before starting on the sparse memory version below.

   For this checkpoint, we have provided a small set of correctness tests in `Homework3Checkpoint2CorrectnessTest.java`. The `testUsefulParScoring` test in `Homework3PerformanceTest.java` also evaluates the performance of your `UsefulParScoring` implementation against the sequential version. We have provided a SLURM file under `src/main/resources` that can be used on NOTS to submit `Homework3PerformanceTest` for testing on a compute node. Note that you will need to edit this SLURM file to supply your e-mail for notification and to provide the correct path to your hw_3 folder on NOTS. Additional correctness and performance tests will also be provided on the Autograder for the `UsefulParScoring` implementation.

3. **[Final submission: Sparse memory version and useful parallelism on NOTS compute nodes (30 points)]**

   The sequential algorithm outlined in Listing 1 and `SeqScoring.java` allocates and uses a dense two-dimensional matrix which requires $O(n^2)$ space when processing strings of size $O(n)$. The goal of this part of the assignment is to create a *sparse* memory version of the program that can process strings of length $O(10^5)$ or greater by using space that is less than $O(n^2)$. The key idea to think about is what data really needs to be retained as the computation advances. For example, in the sequential version, row 1 of the S matrix can be freed (set to null and garbage-collected) when the computation reaches row 3, since computation of row 3 only needs row 2 and not row 1. As a reminder, since the JVM uses automatic memory management through garbage collection, an object cannot be freed unless there are no remaining references to it. To produce a space-efficient version, you will need to ensure that no unnecessary data is retained, i.e. that it is not referenced by any variables in your implementation.

   You will need to design and implement an analogous approach to reducing the space requirements of the parallel version. This will require reworking the data structure for matrix S, and may even require

using a different algorithm (with different parallel constructs) from `UsefulParScoring.java`. Your code for this part will need to go into the `SparseParScoring.java` file.

As before, we recommend first debugging your solution on small strings for correctness, and then testing with pairs of strings of length $O(10^5)$ on dedicated NOTS compute nodes, with the heap size set to 8GB. For reliable timing measurements, *be sure to run these computations only on NOTS compute nodes, and not on the login node.* Also, there may be be some impact of Java's garbage collection (GC) on the performance you observe. Please contact a teaching staff member if you believe that GC overheads are interfering with your performance measurements.

For this checkpoint, we have provided a small set of correctness tests in `Homework3Checkpoint3CorrectnessTest.java`. The `testSparseParScoring` test in `Homework3PerformanceTest.java` also evaluates the performance of your `SparseParScoring` implementation against the sequential version. We have provided a SLURM file under `src/main/resources` that can be used on NOTS to submit `Homework3PerformanceTest` for testing on a compute node. Additional correctness and performance tests will also be provided on the Autograder for the `SparseParScoring` implementation.

4. [**Homework report (15 points)**] With the final checkpoint you should submit a written report summarizing the design of your parallel algorithms in `IdealParScoring.java`, `UsefulParScoring.java`, and `SparseParScoring.java`, explaining why you believe that each implementation is correct and data-race-free.

   Your report should also include the following measurements for `UsefulParScoring.java` and `SparseParScoring.java`:

   (a) Execution time of `SeqScoring.java` and `UsefulParScoring.java` on a NOTS compute node with inputs of length (approximately) 10,000. You can get these numbers from a run of the `Homework3PerformanceTest.testUsefulParScoring` test on NOTS (through the Autograder or manually).

   (b) Execution time of sequential and parallel versions of `SparseParScoring.java` with inputs of length (approximately) 100,000. Note that you will need a single-threaded execution of `SparseParScoring` for this item, not a run of `SeqScoring` as it will run out of memory. You can get these measurements from the `Homework3PerformanceTest.testSparseParScoring` test on NOTS (through the Autograder or manually).

2.5 A Note on the Autograder

In previous labs and assignments, we have only used the Autograder to test correctness of your implementation and abstract performance metrics. However, the autograder can also handle automatic performance testing on the NOTS cluster. This handles the transfer of your submission to NOTS, the launch of a SLURM compute job on a NOTS compute node, and the transfer of the test results back to the Autograder for display. Hence, on this assignment you will start to see a new Performance section in your runs, which includes tests exclusively focused on the real world performance of your implementation running on a NOTS compute node for checkpoints 2 and 3.

2.6 Generation of Test Data

You are welcome to generate any test data that you choose to debug your programs. Just keep in mind that they need to be strings of characters in $\{A, C, T, G\}$.

We have provided some sample input files for your convenience. They are available in the `src/main/resources` directory.