Homework 3: due by 11:59pm on Wednesday, March 22, 2017

Instructor: Vivek Sarkar, Co-Instructor: Mackale Joyner.

Programming Checkpoint 1 due by 11:59pm on Friday, February 24, 2017

Programming Checkpoint 2 due by 11:59pm on Wednesday, March 8, 2017

Total score: 100 points

All homeworks should be submitted through the Autograder, and also committed in the svn repository at `https://svn.rice.edu/r/comp322/turnin/S17/`*your-netid*`/hw_3` that we will create for you. In case of problems committing your files, please contact the teaching staff at `comp322-staff@mailman.rice.edu` before the deadline to get help resolving for your issues.

Your solution to the written assignment should be submitted as a PDF file named `hw_3_written.pdf` in the hw_3 directory. This is important — you will be penalized 10 points if you place the file in some other folder or with some other name. The PDF file can be created however you choose. If you scan handwritten text, make sure that you use a proper scanner (not a digital camera) to create the PDF file. Your solution to the programming assignment should be submitted in the appropriate location in the hw_3 directory.

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.). Slip days will be automatically tracked through the Autograder.

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 a question, please seek a clarification on Piazza (remember not to share homework solutions in public posts) 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. The written portion is due along with the final submission by March 22, 2017.*

1.1 Parallel Topological Sort

Recall from COMP 182 that a *directed, acyclic graph* (DAG) is a directed graph with no cycles. A *topological sort* of a DAG $g = (V, E)$ is a linear ordering of all its nodes such that if g contains an edge (u, v), then u appears before v in the ordering. For this question, assume that the DAG is restricted so that only node 0 is a *root node* (i.e., a node with no incoming edges), and all other nodes have at least one incoming edge. Algorithm **DFS** below shows how to perform a sequential depth-first search traversal for such a DAG, which prints node indices in a preorder traversal of the DAG.

Algorithm 1: DFS of DAG with single root node

Input: Graph $g = (V, E)$, $V = \{0, 1, \ldots, n-1\}$, parent map p_i, $\forall i \in V$ (node 0 is the only root node).
Output: None.
Modifies: Parent map, p.

$p_0 \leftarrow -1$; `// We designate the parent of the root node to be '-1'`
Visit$(g, 0, p)$;

Algorithm 2: Visit

Input: Graph $g = (V, E)$, node $i \in V$, and p_j, $\forall j \in V$.
Output: None.
Modifies: p.

Print i ; `// Use any parallel constructs that you've learned in Module 1 to ensure that these print statements are performed in a topological sort order. You may also use Atomic Variables from Module 2, if you like, but no other constructs from Module 2.`
foreach *outgoing neighbor* h *of* i *(with an edge from* h *to* i*)* **do**
  if $p_h = null$ **then**
    $p_h \leftarrow i$;
    Visit(g, h, p);

1. (15 points) Modify algorithm **DFS** above by *adding parallel constructs* to ensure that the print statements occur in a *topological sort* order for the given DAG. You may add any parallel constructs that you've learned in Module 1 (e.g., async tasks, future tasks, data-driven tasks, etc.) and also add auxiliary objects as needed (e.g., futures, data-driven futures, etc.), but the basic depth-first traversal in the original algorithm cannot be modified. *You may also use Atomic Variables from Module 2, if you like, but no other constructs from Module 2. You can also assume that incoming neighbors are available as part of the DAG data structure, in addition to the outgoing neighbors used above.* Explain why your modified algorithm prints the node indices in a topological sort order.
2. (10 points) Explain if your modified algorithm in part 1 above is guaranteed to be *functionally deterministic* and/or *structurally deterministic*, and why or why not.

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:

X'	=	-	A	C	-	C	T
Y'	=	T	A	C	G	G	T

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:

X''	= A	C	C	T	-	-	-	-	-	-
Y''	= -	-	-	-	T	A	C	G	G	T

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)* × *(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

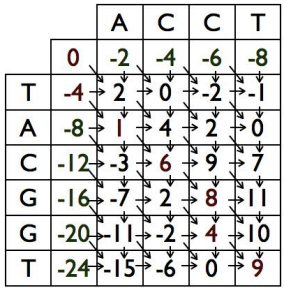

Figure 1: Dependences in Pairwise Sequence Alignment

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. Your homework deliverables are as follows:

1. **[Checkpoint 1 due on February 24, 2017: 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.java` that computes the same output as `SeqScoring.java`, 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`. Your solution will be evaluated using HJlib's abstract metrics. The abstract metric costs used will be those from Homework 1, i.e. they will not include the async spawn cost from Homework 2.

   For this checkpoint, we have some unit tests in `Homework3Checkpoint1CorrectnessTest.java` and on the autograder.

   Hints based on common errors/omissions from past years: Remember to check that your solution passes all unit tests, and that you don't have any checkstyle errors.

2. **[Checkpoint 2 due on March 8, 2017: 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) using `Homework3Checkpoint2CorrectnessTest`, and then evaluating the performance of your implementation with pairs of strings of length $O(10^4)$ on dedicated NOTS compute nodes using `Homework3PerformanceTest`. Lab 5 provides instructions on submitting jobs to the NOTS cluster. You may also use the autograder to automatically perform performance tests on NOTS.

   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 solution for the final homework submission 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 set of unit tests in `Homework3Checkpoint2CorrectnessTest.java`. The `testUsefulParScoring` and `testUsefulParScoring2` tests in `Homework3PerformanceTest.java` also evaluate 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.

Hints based on common errors/omissions from past years: Remember to check that your solution passes all unit tests, and that you don't have any checkstyle errors. Also, you should aim to get a speedup of $\geq 10\times$ for this part; you will get a 1-point deduction if the speedup is in the $[9, 10)$ range, a 2-point deduction if it is in the $[8, 9)$ range, etc.

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.*

   For this checkpoint, we have provided a small set of unit tests in `Homework3Checkpoint3CorrectnessTest.java`. The `testSparseParScoring` and `testSparseParScoring2` tests in `Homework3PerformanceTest.java` also evaluate 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.

   Hints based on common errors/omissions from past years: Remember to check that your solution passes all unit tests, and that you don't have any checkstyle errors. Also, you should aim to get a speedup of $\geq 5\times$ for this part; you will get a 1-point deduction if the speedup is in the $[4.5, 5)$ range, a 2-point deduction if it is in the $[4, 4.5)$ range, etc. Finally, remember to include all the information listed below (summarize design of all three parallel versions, include performance results, etc.)

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 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 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).