Homework 2: due by 11:59pm on Tuesday, September 25, 2012 (Total: 100 points)

Instructor: Vivek Sarkar
Co-Instructor: Ran Libeskind-Hadas

All homeworks should be submitted using the submission system at http://cs.hmc.edu/submit. In case of problems using the system, please email your homework to cs181ehelp@cs.hmc.edu.

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 elses work as your own. If you use any material from external sources, you must provide proper attribution.

1 Written Assignment (50 points total)

Please submit your solution to this assignment in a plain text file named `hw2_written.txt` *in the submission system.*

Optimal Sorting on a 2-Dimensional Mesh!

1. Recall that we argued in class that every sorting algorithm on a $\sqrt{n} \times \sqrt{n}$ mesh has worst case running time of $\Omega(\sqrt{n})$ (that is, asymptotically $\sqrt{n}$ or worse.) Explain briefly why this is true.

2. In class we developed the Shearsort Algorithm which sorts n numbers on a mesh in time $O(\sqrt{n} \log n)$. Now we'll develop an asymptotically optimal sorting algorithm for a $\sqrt{n} \times \sqrt{n}$ mesh. That is, we'll find an algorithm that runs in time $O(\sqrt{n})$. We'll call this algorithm "Niftysort". The Niftysort algorithm will sort the input into a snakelike order (just like the ordering that Shearsort gave us). For simplicity, we'll assume that $\sqrt{n}$ is a power of 2.

   Niftysort starts by calling itself to recursively Niftysort each the four quadrants of the mesh in a snakelike order. Second, Niftysort sorts each entire row of the mesh. The rows are sorted in alternating order (increasing, decreasing, etc. as in Shearsort). Third, the columns of the mesh are sorted from top to bottom (again like in Shearsort). Finally, we treat the overall snakelike path of length n as one long array and we do $4\sqrt{n}$ steps of odd-even sorting on this array. (Actually, only $2\sqrt{n}$ steps are needed, but this makes the proof more complicated. If it makes your life easier, you can replace the 4 with a 6 or any other fixed constant that you like and you can justify.) Use the Knuth 0-1 Sorting Theorem to prove that Niftysort correctly sorts any input set.

3. Finally, show that the running time of Niftysort is indeed $O(\sqrt{n})$. Show your work in detail. Certifiably nifty!

2 Programming Assignment (50 points total)

2.1 Habanero-Java Setup

See Section 2.1 of the Homework 0 handout for instructions on setting up a Habanero-Java installation for use in this homework.

2.2 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. 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 $|X'| = |Y'|$,

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'.

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

2.3 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 2.2 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 $4 \times M_{p,-} + 6 \times M_{-,q} = -32$.

2.4 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 $\forall i,j \geq 1$:

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

The above definition directly leads to a sequential dynamic programming algorithm that can be implemented as shown in Listing 1 (using HJ's sequential `for` loop construct for iterating over a rectangular region). 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 (point[i,j] : [1:X.length(),1:Y.length()] ){
2     char XChar = X.charAt(i-1);
3     char YChar = Y.charAt(j-1);
4     int diagScore = S[i-1][j-1] + M[charMap(XChar)][charMap(YChar)];
5     int topColScore = S[i-1][j] + M[charMap(XChar)][0];
6     int leftRowScore = S[i][j-1] + M[0][charMap(YChar)];
7     S[i][j] = Math.max(diagScore, Math.max(leftRowScore, topColScore));
8    }
9  }
10 int finalScore = S[X.length()][Y.length()];
```

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

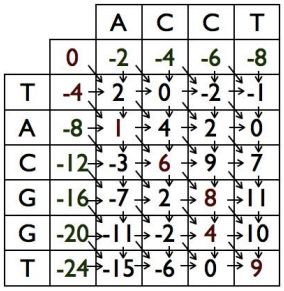

Figure 1: Dependences in Pairwise Sequence Alignment

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.

2.5 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.hj. that you can

use as a starting point. All programs that you submit should take two command-line arguments, *string1* and *string2*, as in `SeqScoring.hj`. We have also provided two small sample inputs in X.txt and Y.txt. Section 2.6 contains suggestions for generating larger test data.

Your homework deliverables are as follows:

1. **[Ideal parallel version using forall loop and barrier (10 points)]** *For this part, your assignment is to examine the dependence structure for $S[i, j]$ defined in Section 2.4 and create an ideal parallel version called* `IdealForallScoring.hj` *that computes the same output as* `SeqScoring.hj`*, and uses a* `forall` *loop with barriers*[1] *to deliver 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. A call to `perf.addLocalOps(1)` has already been added at the appropriate point in IdealParScoring.hj. However, note that the current HJ implementation only supports abstract metric for the finish, async, forall, and future constructs (but not for data-driven futures). We recommend testing your solution on pairs of strings of length ≤ 10 for this part of the assignment.

   A key challenge is that neither the i-loop nor the j-loop in Listing 1 can legally be converted to a `forall` loop because of the dependence structure shown in Figure 1. However, a technique known as *loop skewing* can be applied to transform the loop in Listing 1 to the loop nest shown below in Listing 2. (The *iLow* $\leq$ *iHigh* test in line 4 is technically redundant, but is included to work around a limitation in the current HJ compiler support for forall loops.)

   This skewed loop nest iterates over diagonals rather than rows and columns. As a result, all iterations in the for-i loop are independent, and that loop can be converted to a forall-i loop.

```
1  for (point [jj] : [2:xLength+yLength] ) {
2     int iLow = Math.max(1,jj-yLength);
3     int iHigh = Math.min(xLength,jj-1);
4     if (iLow <= iHigh)
5        for (point [i] : [iLow:iHigh] ){
6           int j = jj-i;
7           char XChar = X.charAt(i-1);
8           char YChar = Y.charAt(j-1);
9           int diagScore = S[i-1][j-1] + M[charMap(XChar)][charMap(YChar)];
10          int topColScore = S[i-1][j] + M[charMap(XChar)][0];
11          int leftRowScore = S[i][j-1] + M[0][charMap(YChar)];
12          S[i][j] = Math.max(diagScore, Math.max(leftRowScore,topColScore));
13       } // for-i
14 } // for-j
```

Listing 2: Skewed version of loop nest from Listing 1

   The submission system will ask for a file named `IdealForallScoring.hj`*, which should contain your solution to this part of the homework. Include ideal parallel times for the sequential and parallel versions (using inputs X.txt and Y.txt), as comments at the top. Also include one or two sentences as comments to indicate you expect the CPL complexity to be as a function of Xlength and Ylength in general.*

2. **[Ideal parallel version using futures (15 points)]** *For this part, your assignment is to examine the dependence structure for $S[i, j]$ defined in Section 2.4 and create an ideal parallel version called* `IdealFutureScoring.hj` *that computes the same output as* `SeqScoring.hj`*, and uses* futures *instead of a* `forall` *loop, to again deliver the maximum ideal parallelism ignoring all overheads.*

   To use futures in this part, you will need to replace the two-dimensional array of `int`, `S[ ][ ]`, by a two-dimensional array of `future`⟨`int`⟩.

[1] See Sections 9.1, 10.1 and 10.2 of the Module 1 handout for a summary of `forall` loops and barriers.

The submission system will ask for a file named `IdealFutureScoring.hj`*, which should contain your solution to this part of the homework. Include ideal parallel times for the this version using inputs X.txt and Y.txt, as a comments at the top, along with one or two sentences as comments as to what you expect the CPL complexity to be as a function of Xlength and Ylength in general. Also, add a comment as to which version (forall vs. futures) is likely to have a smaller value of CPL if the time taken to compute each element was variable instead of constant. For example, if* `addLocalOps(1)` *was replaced by* `addLocalOps(i+j)`*.*

3. **[Useful parallelism with data-driven tasks (25 points)]** *For this part, your assignment is to create a new parallel version of* `SeqScoring.hj` *that is designed to achieve the smallest execution time on a real multicore processor with 2 or more cores by using data-driven tasks and data-driven futures.* You may use any machine that you like for this purpose. FYI, the link LabMachines identifies a set of lab machines that were installed at the start of this semester and should have identical configurations (so executions times for the same program and input should be reproducible across these machines).

   The reason for using data-driven tasks is that the forall and future versions created in the previous parts of this homework will incur too much overhead to be practical for use in this part. We recommend first debugging your solution on small strings for correctness, and then evaluating the performance of your implementation with pairs of strings of length $O(10^4)$ for performance evaluations.

   Finally, the original `SeqParScoring.hj` program accepts a third command-line argument that specifies the number of times the entire computation should be performed. A value of 5 (or larger) is recommended for this argument. The rationale for the repetitions is to report the minimum time of 5 runs so as to reduce the impact that JVM services (e.g., JIT compilation, garbage collection) may have on your measurements. Since the lab machines have 4GB of memory, it is recommended that you increase the maximum heap size to 2GB by using the -mx option when running HJ programs on these machines: "hj -mx 2000m UsefulParScoring *string1 string2* 5".

   The submission system will ask for a file named `UsefulParScoring.hj`*, which should contain your solution to this part of the homework. Include (in comments at the top) the execution times (minimum time of 5 runs) for the* `SeqParScoring.hj` *and* `UsefulParScoring.hj` *programs for inputs that are approximately* 10^4 *in length. Also comment on whether or not your parallel program has obtained a speedup relative to the sequential version*[2]*.*

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 $\{A, C, T, G\}$.

This is optional, but if you are interested in generating pairs of DNA sequences under realistic models of evolution, you can use a free web service available at `http://bibiserv.techfak.uni-bielefeld.de/rose/submission.html`. You should use the following options for this web service (with default values for everything else):

- *Sequence type:* Select DNA.
- *Number of sequences:* Enter 2.
- *Average length of sequences:* Enter whatever length you need e.g., 1000. Earlier this year, the web site would accept larger lengths than 1000. If it doesn't, you can create synthetic inputs of length 10^4 by concatenating 10 strings of length 10^3.
- *Average pairwise distance:* This parameter impacts the number of gaps that you are likely to see in the alignment (a larger alignment will lead to more gaps). We recommend entering 5 for sequences of length $O(10^3)$.

[2]Note that the `-perf` option should *not* be used for these measurements.