Homework #1

(Due: March 20)

Task 1. [80 Points] Pairwise Sequence Alignment with Affine Gap Costs. Sequence alignment plays a central role in biological sequence comparison, and can reveal important relationships among organisms. Given two strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ over a finite alphabet Σ, an *alignment* of X and Y is a matching M of sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ such that if $(i, j), (i', j') \in M$ and $i < i'$ hold then $j < j'$ must also hold. The i-th letter of X or Y is said to be in a *gap* if it does not appear in any pair in M. Given a *gap penalty* g and a mismatch cost $s(a, b)$ for each pair $a, b \in \Sigma$, the *basic (global) pairwise sequence alignment problem* asks for a matching M_{opt} for which $(m + n - |M_{opt}|) \times g + \sum_{(a,b)\in M_{opt}} s(a, b)$ is minimized.

The formulation of the basic sequence alignment problem favors a large number of small gaps while real biological processes favor the opposite. The alignment can be made more realistic by using an *affine gap penalty* which has two parameters: a *gap introduction cost* g_i and a *gap extension cost* g_e. A run of t gaps incurs a total cost of $g_i + g_e \times t$. Such an alignment with minimum cost can be found by solving the following dynamic programming recurrences (observe the similarity between the recurrence for G and the LCS recurrence we saw in the class).

$$D(i,j) = \begin{cases} G(0,j) & \text{if } i = 0 \wedge j > 0, \\ \min \begin{Bmatrix} D(i-1,j), \\ G(i-1,j) + g_i \end{Bmatrix} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \qquad I(i,j) = \begin{cases} G(i,0) & \text{if } i > 0 \wedge j = 0, \\ \min \begin{Bmatrix} I(i,j-1), \\ G(i,j-1) + g_i \end{Bmatrix} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases}$$

$$G(i,j) = \begin{cases} 0 & \text{if } i = 0 \wedge j = 0, \\ g_i + g_e \times j & \text{if } i = 0 \wedge j > 0, \\ g_i + g_e \times i & \text{if } i > 0 \wedge j = 0, \\ \min \begin{Bmatrix} D(i,j), \\ I(i,j), \\ G(i-1,j-1) + s(x_i, y_j) \end{Bmatrix} & \text{if } i > 0 \wedge j > 0. \end{cases}$$

The optimal alignment cost is $\min\{G(m, n), D(m, n), I(m, n)\}$ and an optimal alignment can be traced back from the smallest of $G(m, n)$, $D(m, n)$ and $I(m, n)$.

For simplicity, in the rest of Task 1 we will assume that $m = n = 2^k$ for some integer $k \geq 0$. We will also assume that $\Sigma = \{A, C, G, T\}$, $g_i = 2$, $g_e = 1$, and $s(x_i, y_j) = 1$ if $x_i \neq y_j$ and 0 otherwise.

(*a*) [**5 Points**] Implement the serial algorithm that naïvely fills out the $n \times n$ cost arrays of I, D and G in the forward pass in $\Theta\left(n^2\right)$ time using $\Theta\left(n^2\right)$ space. Remember to save the backward pointers so that the traceback path can be extracted in the backward pass.

(*b*) [**10 Points**] Parallelize the forward pass in part (a) using `cilk_for`. Analyze the parallelism in the resulting algorithm (include both forward pass and backward pass).

(*c*) [**15 Points**] Observe that in the serial divide-and-conquer algorithm for LCS discussed in the class some of the recursive calls can be made in parallel (i.e., the processing of quadrants Q_{12} and Q_{21} in the forward pass). Implement a divide-and-conquer parallel algorithm for pairwise sequence alignment based on that idea. It is very important that you do not recurse down to sequences of length 1, as that will make the overhead of recursion too high. Find the base case size that gives you the smallest running time.

(*d*) [**5 Points**] Analyze the parallelism and space complexity S_p (i.e., space usage on p processing elements) of the algorithm in part (c).

(*e*) [**20 Points**] Implement a divide-and-conquer parallel algorithm following the parallel LCS algorithm we saw in the class. In the boundary generation agorithm divide the problem into q^2 subproblems, where $q = 2^l$ ($l > 0$) is a user-defined parameter. Optimize base case size.

(*f*) [**5 Points**] Analyze the parallelism and space usage S_p of the algorithm in part (e) assuming $q = p'$, where p' is the largest power of 2 such that $p' \leq p$. Here p is the maximum number of processing elements available on the machine.

(*g*) [**10 Points**] Find the largest value of n (say, n_{max}) for which your implementation in part (c) terminates in less than 30 minutes on a single core (i.e., using `-cilk_set_worker_count=1` as a parameter). Generate a Cilkview scalability plot (see slides 32–33 of lecture 3) for part (c) and one for part (e) using sequences of length n_{max}. Compare the two plots and explain the differences.

(*h*) [**10 Points**] Find the largest value of n (say, n'_{max}) for which your implementation in part (a) does not run out of RAM space and terminates in less than 30 minutes. Use `PAPI`[1] to find the number of L1 cache misses incurred by that implementation on sequences of length n'_{max}. Now remove all `spawn` and `sync` keywords from your implementation in part (c), and replace each `cilk_for` keyword with `for`. Then find the number of L1 cache misses incurred by the resulting serial implementation for $n = n'_{max}$. Explain your findings.

Task 2. [**30 Points**] **Pairwise Alignment for Highly Uneven Sequence Lengths.** Consider a scenario where $m \ll n$ and $m \ll p$ (see Task 1 for the definitions of m, n and p). More specifically, assume m to be a small constant for this task. Suppose we are no longer concerned about space usage or cache performance. We simply want to design an algorithm with high parallelism for computing only the cost of the optimal alignment. We do not need to extract the optimal alignment (so no need to store the backward pointers for extracting the traceback path).

(*a*) [**5 Points**] Analyze the parallelism of your implementation in part (b) of Task 1 for the current scenario.

(*b*) [**20 Points**] Let us simplify the problem by removing the "affine gap penalty" assumption, i.e., assuming $g_i = 0$. Then solving only the following recurrence suffices:

[1] In order to load `PAPI` use "module load papi/3.6.0" on `Ranger`, and "module load papi/4.1.2.1" on `Loanstar`. Check `http://icl.cs.utk.edu/papi/` for users' guide.

$$G(i,j) = \begin{cases} 0 & \text{if } i = 0 \wedge j = 0, \\ g_e \times j & \text{if } i = 0 \wedge j > 0, \\ g_e \times i & \text{if } i > 0 \wedge j = 0, \\ \min \left\{ \begin{matrix} G(i-1,j) + g_e, \\ G(i,j-1) + g_e, \\ G(i-1,j-1) + s(x_i, y_j) \end{matrix} \right\} & \text{if } i > 0 \wedge j > 0. \end{cases}$$

Design a parallel algorithm that fills out the cost matrix row by row. Analyze its parallelism.

(*c*) [**5 Points**] Extend your algorithm in part (b) to solve the original version of the problem, that is, with affine gap penalty (i.e., $g_i \neq 0$) as specified in Task 1 with recurrences for D, I and G. What is the parallelism of the extended algorithm?

```
PAR-RANDOMIZED-LOOPING-QUICKSORT(A[q : r])
(Input is an array of distinct elements.)
 1. n ← r − q + 1
 2. if n > 1 then
 3.     k ← 0
 4.     while max{r − k, k − q} > (3/4)n do
 5.         select a random element x from A[q : r]
 6.         k ← PAR-PARTITION(A[q : r], x)
 7.     endwhile
 8.     parallel : PAR-RANDOMIZED-LOOPING-QUICKSORT(A[q : k − 1])
                   PAR-RANDOMIZED-LOOPING-QUICKSORT(A[k + 1 : r])
```

Figure 1: Parallel randomized quicksort that loops until it finds a suitable partition.

Task 3. [**40 Points**] **Randomized Parallel Quicksort with Looping.** Consider the parallel randomized quicksort algorithm given in Figure 1 which loops until it finds a pivot element with rank between $\frac{1}{4}n$ and $\frac{3}{4}n$. It uses the $\Theta(n)$ work and $\Theta\left(\log^2 n\right)$ depth parallel partition algorithm discussed in the class.

(*a*) [**5 Points**] Show that $D < 8 \ln n$, where D is the recursion depth of the algorithm (ignoring the recursion depth of PAR-PARTITION).

(*b*) [**5 Points**] Prove that the algorithm terminates in $\mathcal{O}\left(\log^3 n\right)$ expected time and performs $\mathcal{O}(n \log n)$ expected work.

(*c*) [**25 Points**] Fix any element v of the original array A containing n elements, and track its location in A during the entire running time of PAR-RANDOMIZED-LOOPING-QUICKSORT. Let R_v be the number of times v ends up in $A[q : r]$ during a call to PAR-PARTITION in step 6 of the algorithm. Prove that $R_v \geq 32 \ln n$ with probability at most $\frac{1}{n^2}$.

(*d*) [**5 Points**] Prove that w.h.p. PAR-RANDOMIZED-LOOPING-QUICKSORT terminates in $\mathcal{O}\left(\log^3 n\right)$ time and performs $\mathcal{O}(n \log n)$ work.

APPENDIX 1: Input/Output Format for Task 1

Your code must read from standard input and write to standard output.

- **Input Format:** The first line of the input will contain a single integer giving the value of n (guaranteed to be a power of 2). Each of the next two lines will contain a sequence (strings of A, C, G, T) of length at least n. You must read only the first n symbols/characters of each sequence.
- **Output Format:** The first line of the output will contain a single integer giving the optimal alignment cost. The next two lines will contain the aligned sequences. Each output sequence will contain the original n symbols from the input possibly separated by gaps. Use "minus" ("-") to represent gaps.
- **Sample Input/Output:** Please check the folder "/work/01905/rezaul/CSE613/HW1/samples" on Lonestar.

APPENDIX 2: What to Turn in

Please email one compressed archive file (e.g., zip, tar.gz) containing the following items to `cse613hw@cs.stonybrook.edu`.

- Source code, makefiles and job scripts for Task 1.
- A PDF document containing all answers and plots.
- Output generated for the input files under "/work/01905/rezaul/CSE613/HW1/samples/turn-in/" on Lonestar. If the name of the input file is "xxxxx-in.txt", please name the output files as "xxxxx-1b-out.txt", "xxxxx-1c-out.txt" and "xxxxx-1e-out.txt" for tasks 1(b), 1(c) and 1(e), respectively. No need to generate an output if it takes more than an hour or overflows the RAM. In such cases simply state the cause of failure in the output file.
- Output files generated by Cilkview.

APPENDIX 3: Things to Remember

- **PLEASE NEVER RUN ANYTHING THAT TAKES MORE THAN A MINUTE OR USES MULTIPLE CORES ON TACC LOGIN NODES.** TACC policy strictly prohibits such usage. They reserve the right to suspend your account if you do so. All runs must be submitted as jobs to compute nodes (even when you use Cilkview or PAPI).
- Please store all data in your work folder ($WORK), and not in your home folder ($HOME).
- When measuring running times please exclude the time needed for reading the input and writing the output. Measure only the time needed by the algorithm. Do the same thing when you run Cilkview or measure cache misses with PAPI.
- Please make sure that speedup values for trial runs (or measured speedups) are included in the Cilkview plots you generate.