Problem Set 10

Due: Wednesday, November 12, 2014.

Collaboration policy: collaboration is *strongly encouraged.* However, remember that

1. You must write up your own solutions, independently.
2. You must record the name of every collaborator.
3. You must actually participate in solving all the problems. This is difficult in very large groups, so you should keep your collaboration groups limited to 3 people in a given week.
4. **No bibles. This includes solutions posted to problems in previous years.**

Problem 1. The following problem arises in both VLSI design (wiring together terminals on a chip) and in network routing (reserving bandwidth for a set of network connections). You are given a graph with unit capacity edges (channel on a chip, or links in a network) and a collection of pairs of vertices. Each pair must be connected by a single path, such that the paths are disjoint—at most one path per edge. Finding a feasible solution is NP-complete, so instead we seek an almost-feasible solution, in which there are not too many paths per edge.

(a) Devise an integer linear program capturing this routing problem. Write it in terms of f_{ij}^k, an indicator variable for whether the path from between the k^{th} demand pair uses edge ij. **Hint:** think of a unit-value flow between each demand pair.

(b) Consider the relaxation of this ILP. Argue that it can be seen as defining a collection of fractional paths between each demand pair, of total capacity 1, and that these paths can be read out of the solution in polynomial time.

(c) Given these fractional paths, devise a randomized rounding scheme that gives an integer solution. Argue that if each edge has capacity 1, and the path-finding problem has a feasible solution, then the rounding scheme can find a solution in polynomial time such that every edge carries $O(\log n)$ paths, where n is the number of paths to be routed. If you aren't familiar with the Chernoff bound, you may use the following fact: if $X = \sum X_i$ is a sum of independent indicator random variables, and $E[X] \leq 1$, then the $\Pr(X > O(\log n)) < 1/n^{10}$.

Problem 2. Suppose you are given a linked list: there is an array of N nodes in arbitrary order, and each node stores a pointer to the next element (successor) in the list. In this problem, you will solve the *list ranking* problem: compute the *rank* or *index* of each node in the list. You may assume a black box that sorts using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers in the cache-oblivious model.

(a) Show how to add to each node a pointer to its successor's successor using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers in the cache-oblivious model. (**Hint**: Use sorting.)

(b) Suppose we scan the nodes and, for each node, label the node with the outcome of an independent fair coin flip. What local rule can we use to select αN nodes in expectation, where $0 < \alpha < 1$ is a constant of your choice, with the property that no two selected nodes are adjacent in the list? Show how to label each node accordingly as selected or unselected using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers in the cache-oblivious model.

(c) Show how to unlink the selected nodes from the list, leaving just the unselected nodes in the list, using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers in the cache-oblivious model.

(d) Suppose that we recursively solve the list-ranking problem on the list of unselected nodes. Show how to then solve the original list-ranking instance using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers in the cache-oblivious model.

(e) What is the total number of memory transfers used by the recursive algorithm outlined for list ranking?

NONCOLLABORATIVE Problem 3. A paging strategy incurs *Belady's anomaly* if, for some sequence of page accesses and two cache sizes $k < k'$, the strategy makes fewer page faults on the smaller cache size k compared to the larger cache size k'. Prove that LRU (Least Recently Used) never incurs Belady's anomaly but that FIFO (First In First Out) incurs Belady's anomaly.

Problem 4. Here we will generalize from the Double Coverage algorithm for k-server on the line to k-server on trees, where each server resides at some vertex of a tree. We will define the length of a path on the tree as the sum of the lengths of the edges on the paths, where the lengths are positive reals. We say that a server s_i is a "neighbor" of a request if there is no other server on the path from s_i to the request.

Algorithm DC-TREE: At each time, all the servers neighboring the request are moving at a constant speed toward the request.

For analysis, we will use the same potential function used for the line: $\Phi = kM_{\min} + \sum_{\text{DC}}$, where $M_{\min}$ is the minimum cost matching between OPT's and DC's servers, and $\sum_{\text{DC}} = \sum_{i<j} d(s_i, s_j)$. Note that the set of neighbors may decrease during the service process since a moving server may become "blocked" by other servers.

(a) Show that **DC-TREE** is k-competitive.

Hint: Break the algorithm into phases, where in each phase the number of neighbors is fixed. Now consider the changes to each of the parts of the potential function within a phase.

(b) Show that any algorithm for k-server on a tree can be used to solve the paging problem by modeling paging as k-server on a particularly simple tree. Note that

k-server algorithm can place servers midway along edges, which doesn't make sense for paging, so you have to "reinterpret" it a bit to get a paging algorithm.

(c) What standard paging algorithm do you get when you apply the above reduction using DC-TREE?

Problem 5. Online snoopy caching. (Note: This problem is longer, but not harder, than usual.) In a snoopy cache multiprocessor system, we have n processors $P_1, \ldots, P_n$, where each P_i has a cache C_i, and the caches communicate with each other over a broadcast channel (bus). Each C_i is an array of size L that needs to provide to P_i a virtual interface to a shared memory of size L. Even though they are "caches", we will assume they never run out of space. Instead we explore the cost of synchronizing them, which is measured in rounds of communication over the bus. Each P_i can make two types of calls to C_i:

- $READ(\ell)$: Read the value at location ℓ of the shared memory.
- $WRITE(\ell, v)$: Write value v to location ℓ of the shared memory.

A location is **valid** for a cache if its local copy is guaranteed to contain a correct value that can be reported to the processor. If a location is valid in exactly one cache then it is **unique**, and otherwise it is **shared**. Initially, all locations are unique to C_1. Then, each cache can broadcast the following messages over the bus:

- *request*(ℓ): Ask what is the value at location ℓ. In response, one of the caches where ℓ is valid broadcasts the value. All caches hear the response (hence "snooping"), mark ℓ as valid and update its value at their local copy.
- *update*(ℓ, v): Announce that the value at location ℓ has been changed to v. Again, all caches hear the broadcast and update their local copies.
- *invalidate*(ℓ): Ask that all other caches mark location ℓ as invalid (they have to obey).

Each message costs one communication round (for *request*(ℓ), one round covers the cost of both the request and the response). Our goal is to devise a competitive synchronization scheme for the caches. For simplicity we will assume that processors call $WRITE$ only on locations that are valid in their cache (even though this somewhat breaks the interface abstraction). When P_i calls $WRITE(\ell, v)$ on a shared location, it has to either *update*(ℓ, v) so the other caches have the correct value, or to *invalidate*(ℓ) so they know their local value is no longer correct. If P_i is only doing one write then it makes sense to *update*, but if it is doing a sequence of writes it is wasteful for C_i to *update* on each write when it can just *invalidate* once. The challange is to decide when to *update* and when to *invalidate* in an online setting.

(a) Consider the scheme SHARE-ALL: Upon $WRITE(\ell, v)$, send *update*(ℓ, v) if ℓ is shared. Upon $READ(\ell)$, send *request*(ℓ) if ℓ is invalid, and then report its value. In this scheme, all location are shared once they are accessed for the first time. Show this scheme does not have a constant competitive ratio. In particular, devise a sequence of calls by the processors, such that the cost ratio of SHARE-ALL to an all-knowing algorithm is unbounded.

(b) Consider the scheme SHARE-NONE: $WRITE(\ell, v)$, update local copy without communication. Upon $READ(\ell)$, send $request(\ell)$ if ℓ is invalid, then report its value, and then immediately $invalidate(\ell)$. In this scheme, only one cache ever has a valid copy of each location, so no locations are shared. Show this scheme does not have a constant competitive ratio.

(c) Consider the scheme SHARE-SOME: Upon $WRITE(\ell, v)$, send $invalidate(\ell)$ if ℓ is shared. Upon $READ(\ell)$, send $request(\ell)$ if ℓ is invalid, and then report its value. We will prove this scheme is 2-competitive. Fix an arbitrary sequence of calls, let OPT be the optimal algorithm for that sequence, and let A be our SHARE-SOME algorithm. At each time t, each location can be in one of four states: (S, S) – shared in A and in OPT; (S, U) – shared in A and unique in OPT; (U, S) – unique in A and shared in OPT; (U, U) – unique in A and in OPT. Define the following potential function:

$$\Phi(t) := \#\{\text{locations in state } (U, U)\} - \#\{\text{locations in state } (U, S)\}.$$

1. Show that in time t we have $C_A(t) \leq 2 \cdot C_{OPT}(t) + \Delta\Phi$. **Hint**: Use case analysis on the two types of calls and on the 4 possible states of the location.
2. Conclude that over the whole sequence, $C_A \leq 2 \cdot C_{OPT} + \Phi(T) - \Phi(0)$, where T is the sequence length. Argue this implies 2-competitiveness.

(d) Now we will show that SHARE-SOME has the optimal competitive ratio for deterministic schemes. Consider two processors and one fixed location ℓ. Fix an arbitrary algorithm A. Consider the following adversarial sequence: in each time step,

- If ℓ is unique to C_1 in A, then P_2 calls $READ(\ell)$.
- If ℓ is unique to C_2 in A, then P_1 calls $READ(\ell)$.
- If ℓ is shared in A, then the last processor to call $READ(\ell)$ now calls $WRITE(\ell, v)$. (v is any value.)

1. What is the cost of A on this sequence after T steps?
2. Devise an all-knowing algorithm $PEEK$ for the above sequence and prove that $C_A \geq 2 \cdot C_{PEEK} - 1$. This means A cannot have competitive ratio better than 2. **Hint**: It is enough for $PEEK$ to peek only one time step ahead.

(e) **(optional)** Consider an alternative model in which only transmitting values counts for communication cost. Specifically, $update(\ell, v)$ costs 1 round of communication; $request(\ell)$ costs $p > 1$ rounds, to account for the delay in waiting for the response; $invalidate(\ell)$ is free, as well as any other communication that does not transmit values. Show that parts (c) extends to this setting: a natural extension of SHARE-SOME is 2-competitive.

OPTIONAL Problem 6. This problem is optional this week and will be required next week.

Read through the handout on the final project, which is linked from the course website. Submit a paragraph-long (or so) proposal for the final project detailing the topic and scope of the proposed project. Include citations to relevant papers. If you are collaborating with other students, submit just one copy with all of your names on it. Working alone is permitted but is not a good idea. Please limit your groups to 3 students. Email your proposal to Professor Karger at `karger@mit.edu` and the TAs. Make the subject line of your "6.854 project" of your email (without quotes).

Problem 7. How long did you spend on this problem set? Please answer this question using the Google form that is located on the course website. This problem is mandatory, and thus counts towards your final grade. It is due by the Monday 2:30pm after the pset due date.