Homework 1

Course: Algorithms for Memory Hierarchy

WS 2012/2013

Problem 1: External Memory Sorting

(a) **Recurrence solving.**

For the EM sorting algorithms we have seen the recurrence

$$Q(N) = \begin{cases} M/B \cdot Q(\frac{N}{M/B}) + O(N/B) & \text{if } N > B \\ O(1) & \text{if } N \leq B \end{cases}$$

Prove that this recurrence solves to $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$. If you need, you can brush up on your recurrence solving skills by reading chapter 4 of the third edition of "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein.

(b) **Work-efficient merge-sort.**

When designing I/O-efficient algorithms so far we concentrated only on the I/O complexity of the algorithms. In particular, we have not described how to process data in internal memory. In this problem we address the running time of the internal memory computations.

Design a mergesort algorithm which is I/O-efficient, i.e. obtains $O(\frac{N}{B} \log_{M/B} N/B)$ I/O complexity, and is work-efficient, i.e. obtains $O(N \log N)$ work in internal memory. Analyze your algorithm.

Hint: Think of how to perform an M/B-way merge work-efficiently in internal memory.

Problem 2: Duplicate Removal

Design an I/O-efficient algorithm for removing duplicates from a multiset of N elements (you cannot assume the range of the elements is known). The output from the algorithm should be the K distinct elements among the N input elements in sorted order. The algorithm should run in $O\left(\max\left\{\frac{N}{B}\log_{M/B}\frac{N}{B} - \sum_{i=1}^{K}\frac{N_i}{B}\log_{M/B}N_i, \frac{N}{B}\right\}\right)$ I/Os, where N_i is the number of copies of the i-th element in the input set.

(Hint: Use merge-sort and remove duplicates as soon as they are found. Analyze the algorithm by considering how many of the N_i copies of an element can be present after j merge steps.)

Problem 3: Selection

In class we stated that the running time of the deterministic selection algorithm is defined by the following recursion:

$$T(N) = \begin{cases} T(N/5) + T(3N/4) + O(N) = T(19N/20) + O(N) & \text{if } N > c \\ O(1) & \text{if } N \leq c \end{cases}$$

for some constant c.

(a) Solve this recurrence. Show your work.

(b) A more precise recurrence for the selection algorithm presented in class is

$$T(N) = \begin{cases} T(9N/10) + O(N) & \text{if } N > c \\ O(1) & \text{if } N \leq c \end{cases}$$

for some constant c.

Show how to obtain this recurrence and solve it.

Problem 4: Cache-oblivious algorithms

1. **Searching.** Prove that the following recurrence for cache-oblivious searching solves to $O(\log_B N)$.

$$Q(N) = \begin{cases} 2Q(\sqrt{N}) + O(1) & \text{if } N > B \\ O(1) & \text{if } N \leq B \end{cases}$$

Show your work.

2. **Sorting.** Prove that the following recurrence for cache-oblivious sorting solves to $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$, assuming that $M = \Omega(B^2)$.

$$Q(N) = \begin{cases} 2\sqrt{N} \cdot Q(\sqrt{N}) + O(N/B) & \text{if } N > M \\ O(N/B) & \text{if } N \leq M \end{cases}$$

Show your work.

Problem 5: PRAM Prefix sums

Write-down the pseudo-code for the PRAM prefix sums using $P = N$ processors with all the details. Formally prove the correctness of the algorithm.

Problem 6: Practical project: sequential vs random access

In this problem you will write some code to understand the effects of caches.

Create a linked list data structure, using a single array. Each entry in the array is a record of two integers: the value of current element and the index of the successor element. Using this data structure, create a circular linked list of size N, where the next pointer of the record in location i points to the record in location $(i + 1) \bmod N$. For the experiments, vary the size N from very small (less than L1 cache size on your system) to very large (e.g. a third or a half of the RAM on your system). Make sure for experiments with large N, N is definitely larger than the size of the highest level cache (L2 or L3, depending on your processor).

(a) Sequentially traverse the linked list following the successor pointers. To make sure the compiler does not optimize out your traversal code, perform some non-trivial operation on the values of the entries as you are traversing the list. For example, you can add up the values of each entry and print it out at the end, or perform list ranking and store the ranks in the values of each entries. Note the runtimes of your experiments. If the runtimes are faster than the resolution of your timer (very likely for small N), traverse the list multiple times.

(b) Now, randomly permute the entries of the array. You can use any way to permute the array, but make sure that you will generate any permutation with equal probability. The simplest (but not the fastest) way to do this is to assign a random number to the value of each entry and sort the array entries by these random numbers. Now traverse the linked list following the successor pointers performing the same non-trivial operation as in part a. Again, perform this for various sizes of N. Note the runtimes of your experiments.

Write up your observations comparing the runtimes of the different experiments you ran. In your measurements, don't count the time it takes to permute the array. In your writeup, describe the method you used to perform the permutation If you used your own method for performing the permutation, prove that it generates every permutation with equal probability.