6.854 Advanced Algorithms

Lecture 22: 11/26/2003 Lecturer: Erik Demaine

Scribes: Grant Wang

22.1 Cache-oblivious algorithms

22.1.1 Introduction

The cache-oblivious model (see Frigo, et al) is almost the same as the external memory model; there is a cache that is fast, and a disk that is much slower. We wish to measure the number of memory transfers an algorithms makes from disk to cache. The key difference between the external memory model and the cache-oblivious model is that an algorithm in the external memory model "knows" the block size B and the size M of the cache. That is, these parameters are part of the algorithm, and the algorithm can make specific memory transfers.

In the cache-oblivious model, the algorithm still runs on a system with a cache and a disk; however, it is unaware of the parameters B and M. In particular, memory transfers are determined by the system using the optimal offline strategy. If the system is using LRU or FIFO online algorithms for memory management, our assumption is valid in the sense that these algorithms, running on a cache size of $2M$ are 2-competitive with the optimal offline strategy on cache of size M. So if we are willing to double the cache size, the number of memory transfers of our algorithm are just a factor of two off the bounds that we will prove.

In practice, algorithms do not necessarily know the cache size M or block size B. Furthermore, it may be the case that the memory hierarchy is more than just two levels deep. This is where the cache-oblivious model proves helpful; our algorithm does not know M or B, yet the bounds we prove are still for any M or B. In addition, even if the memory hierarchy is more than just two levels deep, since our algorithm is optimal for all values of M and B, we can apply our same analysis to all levels of the memory hierarchy. As a result, the algorithm will be optimal on memory hierarchies more than two levels deep. The cache-oblivious model also simplifies the implementation of algorithms; only our analysis must worry about M and B. The algorithm is "oblivious" to these parameters.

In this lecture, we will study some cache-oblivious algorithms; the main tool in developing these algorithms is divide-and-conquer. We divide the problem into subproblems of size $O(1)$. However, in the analysis we will consider the point when the problem is less than or equal to the size M of memory, or the size B of a block. Therefore, at this size, the number of memory accesses is small and easy to compute.

22.1.2 Scanning: an introductory example

Suppose we wish to scan N sequential elements in memory. In the cache-oblivious model, we require $\lceil N/B \rceil + 1$ memory transfers. The extra memory transfer occurs because of a possible misalignment of the first few elements, with respect to block alignment. The external-memory model would be $\lceil N/B \rceil$, because here we could ensure that the first element occurs at the start of a block. Even though this is a simple result, we will use this in the following problems.

22.1.3 Matrix multiplication

Let $A, B \in R^{N \times N}$, and assume that $N = 2^k$. We wish to compute $C = AB$. First consider the most obvious algorithm to compute C. We compute each entry of C_{ij} by computing the dot-product of the ith row of A and the jth row of B:

$$C_{ij} = A^{(i)} B_{(j)}.$$

The number of memory transfers here is $O(\lceil N/B \rceil)$ for the computation of C_{ij} (since we must read $A^{(i)}$ and $B_{(j)}$ from memory). It follows that the total number of memory transfers is then $O(\lceil N^3/B \rceil)$.

Algorithm

We can do much better by considering a divide-and-conquer approach to matrix multiplication. Let us assume that $A, B \in R^{2N}$, so that we can write A, B and C as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

One can verify the following formulas for $C_{11} \ldots C_{22}$:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Analysis

We can write a recurrence for the total number of memory transfers. Note that the addition of two matrices is just scanning, and so can be done in $O(\lceil N^2/B \rceil)$ memory transfers. Therefore, we have the following recurrence for memory transfers:

$$T(N) = 8T(N/2) + O(\lceil N^2/B \rceil)$$

Furthermore, we have the following base cases for values of M and B:

$$T(c\sqrt{M}) = O(\lceil M/B \rceil)$$

That is, when the matrices can fit in cache, there is no need to do more memory transfers than just reading the matrices into cache.

Let's look at the recursion tree to solve this recurrence: at the first level of recursion, the number of memory transfers we do is CN^2/B. In the second level of recursion, the number of memory transfers we do is $8C\frac{N^2}{4B}$. Generalizing, in the ith level of recursion, we do 2^iCN^2 memory transfers. Our base case is when $\frac{N^2}{2^{2i}} = M$, i.e. when $i = \frac{\log N^2/M}{2}$. Therefore, the total number of memory transfers is:

$$\sum_{i=0}^{\frac{\log N^2/M}{2}} 2^iCN^2/B = O\left(\frac{N^3}{B\sqrt{M}}\right).$$

Even though we assumed that N is a power of two and that A, B were square, we can adapt the algorithm to the general case to obtain the same bounds.

22.1.4 Binary search

Suppose we wish to binary search over some sorted array of size N with height of 2^k: how should we lay out the array in memory to reduce the number of memory transfers? We will also use the technique of divide-and-conquer to lay out the memory so that a binary search will minimize the number of memory transfers.

Laying out the array

Instead of looking at the array itself, let's consider the binary tree T on the data. The median element r of the array is at the root of the tree. The root's left child will be a subtree with the elements that come before r in the array, and the root's right child will be a subtree with the elements that come after r in the array.

We wish to lay out T in memory: the key is that we wish to preserve locality. Consider a path down the tree to a leaf: we would like the accesses down this path to be adjacent. To this end, we recursively lay out T as follows:

> Let T_1 be the top half of the tree. It is easily seen that T_1 contains $\sqrt{N}$ entries. The bottom half of the tree contains subtrees $U_1 \ldots U_{\sqrt{N}}$; each subtree U_i contains $\sqrt{N}$ entries.
>
> Recursively layout each subtree $T_1, U_1 \ldots U_{\sqrt{N}}$ until the size of the tree is $O(1)$, and concatenate the layouts in the order above.

Analysis

Let us consider the analysis for a certain value of M and B. Actually, it will turn out that the value of M does not matter.

At some level of the recursion, the subtree we are considering to recursively carve up is of size at most B, which implies that its height is at most $\log B$. At this level of recursion, each subtree has height between $\frac{1}{2}\log B$ and $\log B$. On a path down from the root to a leaf, we visit $\log N$ nodes. The key property is that we achieve locality in the following sense: when we enter a subtree we stay in it and exit, but we never re-enter. Therefore, the total number of subtrees at this level of recursion we visit on a root to leaf path is $2\frac{\log N}{\log B}$. Since each of these subtrees fit in at most 2 blocks of memory (because of misalignment), the total number of memory accesses is:

$$4\frac{\log N}{\log B} \leq 4\log_B N + 2$$

Both the algorithm and analysis generalize when the height is not a power of two, and the node degree is greater than 2, but at most a constant. Furthermore, this can be made dynamic. We can get amortized $O(\log_B N)$ memory transfers for insert, delete, and find. In fact, this layout, known as the van Emde Boas layout (See Prokop 1999, Bender, Demaine, Farach-Colton 2000) can be applied to buffer trees as well.

22.1.5 Linked lists

The data structure problem we just considered was a static one: we were concerned only with how to lay out the data to minimize memory transfers on searches. It is significantly more difficult, in general, to obtain cache-oblivious algorithms for dynamic data structure problems where we may insert and delete entries in our data structure.

Here, we describe a solution to a simple linked lists data structure problem. We wish to derive a cache-oblivious bounds on a linked list that supports the following operations:

- INSERT(x): insert x at the end of the linked list
- DELETE(x): delete x from the linked list
- TRAVERSE(K, x): read off the next k elements in the linked list, starting with x.

Ideally, we would like the first two operations to have $O(1)$ memory transfers, and the last one to have $O(\lceil K/B \rceil)$ memory transfers.

External memory model

It is helpful to first consider how we'd solve this problem in the external memory model. A natural solution would be to break up the list into sublists each of size at most B, so that each sublist would fit in one block. To insert, we just insert into a sublist, and to delete, we just delete from a sublist. To make sure the size of each sublist stays within a constant fraction of B in spite of inserts and deletes, we can just merge or split occasionally. The details are not difficult, and are almost exactly the same as how splitting and merging works for B-trees. It is clear that if the size of each sublist stays within a constant fraction of B, traversing from a particular node takes $O(\lceil K/B \rceil)$ memory transfers.

Cache-oblivious memory model

Here, we develop algorithms for insert, delete and traverse that achieve the desired bounds for memory transfers, but in an *amortized* sense. We maintain the linked list as a contiguous segment, and implement the operations as follows:

- INSERT(x): Just add x to the end of the segment. This takes worst-case $O(1)$ memory transfers (not amortized).
- DELETE(x): Unlink the element x and fix the pointers, leaving a hole. Again, it is clear that this takes worst-case $O(1)$ memory transfers.
- TRAVERSE(K, x): We just follow the pointers K times. Let us count the number of *runs* during such a traversal. A *run* is a sequence of adjacent elements. In the worst case, we may end up with K runs; in the very best case, we end up with just 1 run. Let r be the number of runs during a traversal. The the number of memory transfers is just $r + O(\lceil K/B \rceil)$. To improve upon the number of runs in a future access, we can copy as we traverse, and merge $r - 2$ runs (all but the first and last) into one contiguous run at the end of memory

  Now we look at the amortized cost. Any run we come across in a traversal is the result of an update (either an insert or delete). Therefore, we can charge $r - 3$ of the $r + O(\lceil K/B \rceil)$ cost to the updates. The amortized cost of a traversal, then is $O(\lceil K/B \rceil)$.

One problem is that traversals increase the size of the data structure due to this recopying stage, which may leave long gaps between elements. A simple solution to this is to compactify by traversing the entire list. This will take $O(R + \lceil N/B \rceil)$ memory transfers, but we can amortize this to $O(1)$, because we can charge:

- R to the $R - 1$ runs that we just discarded
- $\lceil N/B \rceil$ to the updates that modified the data structure.