I/O Avoiding Algorithms

Introduction

1. Given a machine with a two-level memory hierarchy, what does an efficient algorithm look like?
    - I/O: Transfers of data between slow and fast memory
    - Assuming these transfers are the dominant cost, how do we minimize them?
    - How to argue lower bounds on I/O

A Sense of Scale

1. Lower-bound on the number of slow-fast memory transfers for a comparison-based sort:
    - Q(n;Z,L) = Omega(n/L log(n/L)/log(Z/L))
2. Given:
    - Volume of data to sort: 1 PiB (2ˆ50 bytes)
    - Record (item) size: 256 bytes
    - Fast memory size: 64 GiB = 2ˆ36 bytes
    - Memory transfer size: 32 KiB = 2ˆ15 bytes
3. Calculate the following numbers of transfer operations in units of trillions of operations (Tops)
    - n = 2ˆ50 / 2ˆ8 = 2ˆ42 records
    - Z = $2^{36/2}8$ = 2ˆ28 records
    - L = $2^{15/2}8$ = 2ˆ7 records

Complexity	Number of Transfers	Speedup vs nlog(n)/log(2)
nlog(n)/log(2)	185	1.00
nlog(n/L)/log(2)	154	1.20
n	4.40	42.0
(n/L)log(n/L)/log(2)	1.20	154
(n/L)log(n/Z)/log(2)	0.481	672
(n/L)log(n/Z)/log(Z/L)	0.0573	3530

External Memory Mergesort

1. Logically divide the input into chunks no greater than Z so that a single chunk fits in fast memory
    - n/(f * Z) where f = [0, 1)
2. Read a chunk of the input from the slow memory into fast memory, producing a (sorted) run
3. Write the run back to slow memory
4. Repeat until all n/(f * Z) chunks are sorted runs
5. Merge the n/(f * Z) runs into a single run

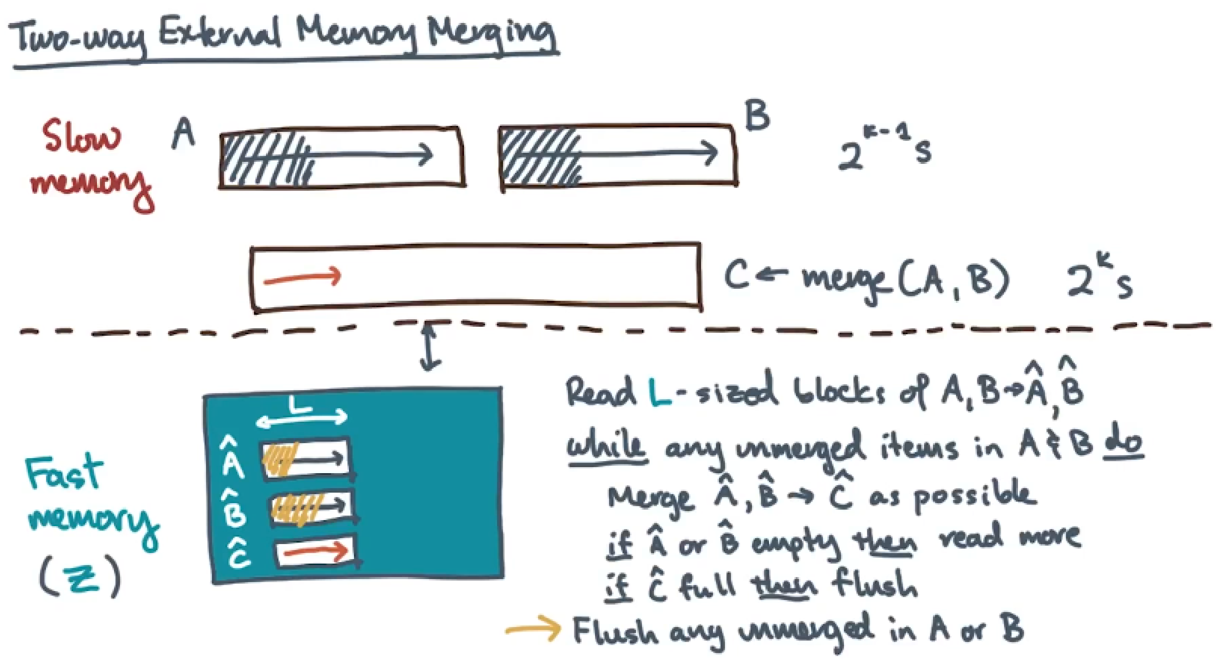
Partitioned Sorting Step Analysis

1. Phase 1:
    - Partition input in n/fZ chunks
    - for each chunk i to n/fZ do
        - Read chunk i
        - Sort it into a (sorted) run
        - Write run i
2. Assumptions:
    - L | f * Z and f * Z | n
    - Comparison sort is optimal
3. What is the algorithmic complexity of each step in phase 1?
    - Read chunk i: fZ/L * n/FZ = n/L transfers
    - Sort it into a (sorted) run: f * Z * log(fZ) * (n/(f * Z)) = nlog(Z) comparisons

- Write run i: fZ/L * n/FZ = n/L transfers

External Memory Mergesort Quiz

1. At each level k, run size is s * 2 ^ k
    - m is number of sorted runs
    - s is number of elements in each run
    - n = m * s
2. Phase 2:
    - Real L-sized blocks of A and B (Ahat and Bhat)
    - while any unmerged items in A and B do
        - Merge Ahat, Bhat -> Chat as possible
        - if Ahat or Bhat empty then read more
        - If Chat full then flush to slow memory
    - Flush any unmerged elements in A or B
3. Algorithmic analysis
    - Transfers: 2^(k-1)/L + 2^(k-1)/L + 2^(k)/L = 2^(k+1)s/L
    - Comparisons: O(s * 2^k)
    - Number of pairs merged at level k = n / (s * s^k)
    - Number of levels = log2(n/s)
4. Totals:
    - Transfers: O(2n/L * log2(n/s))
    - Comparisons: O(n * log2(n/s))



Two-way External Memory Merging

What's Wrong with 2 Way Merging?

1. Phase 1:
    - Partition input into n/Z chunks
    - Sort each chunk, producting n/Z runs of size Z each

2. Phase 2:
    - Merge all runs
3. In terms of n, Z, and L, determine aymptotic costs:
    - Comparisons: O(nlog(n))
    - Transfers: O(n/L * log(n/Z))
4. What's wrong with naive two-way merging algorithm?
    - Two-way merge doesn't utilize all of the fast memory Z
    - Can be off by a factor of 10x to 100x on modern hardware

---

# of transfers in external memory mergesort with 2-way merging:

$$Q(n; Z, L) = O\left(\frac{n}{L} \log_2 \frac{n}{Z}\right) = O\left(\frac{n}{L} \cdot \left[\log_2 \frac{n}{L} - \log_2 \frac{Z}{L}\right]\right)$$

Lower bound:

$$Q(n; Z, L) = \Omega\left(\frac{n}{L} \log_{\frac{Z}{L}} \frac{n}{L}\right) = \Omega\left(\frac{n}{L} \frac{\log_2 \frac{n}{L}}{\log_2 \frac{Z}{L}}\right)$$

Remaining factor of potential improvement:

$$O\left(\log_2 \frac{Z}{L} \cdot \left[1 - \frac{\log_2 \frac{Z}{L}}{\log_2 \frac{n}{L}}\right]\right)$$

Optimal Lower-Bound for Merge Sort

Multiway Merging

1. Instead of merging two blocks at a time, merge all k blocks at once with an additional buffer k+1 for the output
    - Read L blocks of each of k runs into fast memory
    - At each point, pick the next smallest value
    - If an input buffer is empty, refill it
    - If the output buffer is full, write it to slow memory
2. How do we choose the next smallest efficiently?
    - Linear scan (small k)
    - Priority queue (min-heap)
        - Build: O(k)
        - Extract min: O(logK)
        - Insert: O(logK)
3. Cost of 1 k-way merge:
    - Transfers: 2 * k * s / L
    - Comparisons: O(k + k * s * logK)

Cost of Multiway Merge

1. What is the total cost of multiway merging?
    - Assume k = O(Z/L) < Z/L
    - Maximum number of levels l in a merge tree is O(log(n/L)/log(Z/L))

- This makes intuitive sense because we're merging chunks of Z/L instead of chunks of 2, which changes the base of the log
    - Additionally, we have to merge n/L total chunks
2. Total number of comparisons: O(nlog(n))
3. Total number of transfers: O(n/L * log(n/L)/log(Z/L))

A Lower Bound on External Memory Sorting

1. Merge sort with multiway merge complexity is as good as we can hope for from an asymptotic perspective
2. There are two approximations involved:
    - Stirling's approximation: log(x!) ~ xlog(x)
    - log(a choose b) ~ b * log(a/b)

How Many Transfers in Binary Search

1. How many transfers does binary search do?
    - Assumptions:
        - L | n
        - A is aligned
        - n, L, Z are powers of 2
2. Q(n;Z,L) = log(n/L)
    - If n > L, Q = 1 + Q(n/Z; Z, L)
    - If n <= L, Q = 1
    - Using the Master Theorem, Q(n;Z,L) = O(log(n/L))

Lower Bounds for Search

1. Size of index i = floor(log(n)) + 1 bits = O(log(n)) bits
2. Let x(L) = maximum number of bits "learned" per L-sized read
3. Q(n;Z,L) = O(log(n) / x(L))
4. What is an asymptotic upper bound on x(L)?
    - O(log(L))
5. Lower bound: O(log(n)/log(L))
    - Approximately log(L) speedup compared to naive binary search

I/O Efficient Data Structures

1. Using which of these can search attain the I/O lower bound?
    - Ordered doubly-linked list (false)
    - Binary search tree (false)
    - Red-black tree (false)
    - Skip list (false)
    - B-tree (true)
2. B-tree can be made I/O optimal only if the correct branching factor B is chosen
    - B should be equal to L
    - B must be specific to the machine

Conclusion

1. Optimizing transfers requires making memory accesses contiguous and exploit fast memory capacity to the greatest extent possible
2. Our model assumes that data movement dominates, but this may not be true
    - Computational intensity and machine balance