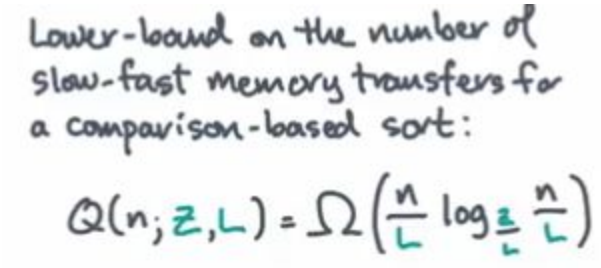
Lesson 3-3 I/O Avoiding Algorithms

Sense of Scale

Goal of Lesson: develop a lower bound for the amount of communication to sort on a machine with slow and fast memory.



n is the number of data points
Z is the size of the fast memory (in words)
L is the number of words per transfer

Quiz Information:

Given:
-volume of data to sort: r*n = 1 PiB (2^{50} Bytes)

-record (item) size: r = 256 Bytes

-Fast memory sze: r*Z = 64 GiB (2^{36} Bytes)

-Memory transfer size: r*L = 32 KiB (2^{15} Bytes)

The number of transfer operations:
Use: r = 2^8B, n = 2^{43} records, Z = 2^{28} records, L = 2^7 records

n $\log_2$ n = 185 T_{ops}
n $\log_2$ (n/L) = 154 T_{ops}
n = 4.40 T_{ops}
n/L $\log_2$ (n/L) = 1.20 T_{ops}
n/L $\log_2$ (n/Z) = 0.275 T_{ops}
n/L $\log_{Z/L}$ (n/L) = 0.0523 T_{ops}

Note the improvements relative to the baseline (n $\log_2$ n) :

A big improvement comes from reducing n to n/L. This means transferring the data in L sized fragments.

Another big improvement comes from going from base 2 to base Z/L. This improvement comes from the capacity of fast memory (Z).

Handy log factoid:
$\log_{Z/L} x = (\log_2 x)/(\log_2 Z/L)$

External Memory Merge Sort

Let's sort n elements in a two level memory hierarchy.

Phase 1:
Assume processor is sequential.

1. Divide the n elements into chunks of size Z, so that each chunk fits into fast memory.
2. Read a chunk from slow memory to fast memory.
3. Now sort this chunk, call it a run.
4. Write the chunk back to slow memory.
5. Repeat for each chunk of data.
6. You will end up with n/(fZ) chunks of sorted runs.

Phase 1:

```
Partition input into n/(fZ) chunks
foreach chunk i ← 1 to n/(fZ) do
        Read chunk i
        Sort it into a run
        Write run i
```

Phase 2:

```
Merge the n/(fZ) runs into a single run
```

Partitioned Sorting Step Analysis
(Phase 1 from above)

Count the number of asymptotic transfers at each step to obtain a total asymptotic cost.

Read chunk i	→	O(n/L) transfers
Sort it into a run	→	O(n*log(Z)) transfers
Write run i	→	O(n/L) transfers

To derive the values:
Assume: L |(fZ) and (fZ) | n + optimal comparison sort

Read chunk i → (fZ/L) * (n/(f*Z)) = O(n/L)
Sort it (the number of comparisons) → $O(fZ \log(fZ)) * n/fZ = O(n \log_2 Z)$
Write run i → O(n/L)

The scheme is giving us something proportional to n/Z transactions

Two Way External Memory Merging

Assume ‘m’ runs of size ‘s’.
The number of items is n = m * s
Now we must merge all of the sorted runs into a single sorted run

One method: merger pairs of runs, then pairs of the pairs, etc.
If we follow this method ….
For each merge, the run size grows…. s, 2s, … ,2^{k-1}s, 2^ks

To visualize this:

We have two runs each of size 2^{k-1}s in slow memory.
Goal… to merge the two runs into a new run ‘C’ of size 2^ks
To do this:
In fast memory there will be three buffers, each holding ‘L’ items (recall ‘L’ is the transaction size)
We’ll use the first two buffers will hold the two runs ’A’ and ‘B’. The third buffer, ‘C’, will hold the combined, sorted run.

To begin:

1. Move an ‘L’ size block from ‘A’ and one from ‘B’. Store them in the fast memory A^ and B^.
2. Sort A^ and B^ into C^

```
Read L sized blocks of A,B and store in A^ and B^
while any unmerged items in A & B do
        merge A^, B^ → C^ as possible
        if A^ or B^ empty then read more
        if C^ full then flush
Flush any unmerged in A or B
```

What is the cost of this?
This scheme only loads items from A or B once
Transfers = (2^{k-1}s) /L + (2^{k-1}s) /L

It only writes a given output block once (2^ks) /L

So … Transfers = (2^{k-1}s) /L + (2^{k-1}s) /L + (2^ks) /L = (2^{k+1}s) /L

Number of comparisons $\theta(2^k s)$

Number of pairs merged at level k = $n/(2^k s)$
Number of levels = $\log_2(n/s)$

Total number of transfers is: $2(n/L)\log_2(n/s)$
Total number of comparisons is: $\theta(n \log_2 n/s)$

The question to ask is … is this good or bad?

External Memory Merge Sort

Merge Sort in External Memory:

Phase 1:
Partition input into $\theta(n/Z)$ chunks
Sort each chunk, producing $\theta(n/Z)$ runs of size Z each

Phase 2:
Merge all runs using a 2-way merge

What are the asymptotic costs:

Phase 1: Comparisons = $O(n \log_2 Z)$
Transfers = $O(n/L)$

Phase 2: Comparisons = $O(n \log_2 n/Z)$
Transfers = $O(n/L \log_2 n/Z)$

Total:
Comparisons: O(n log (n)) = n log n (this is good news, the scheme is work optimal)
Transfers: O(n/L * log(n/Z))

The lower bound is: ~$n/L \log_{Z/L}(n/L)$

What's Wrong with 2 Way Merging

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

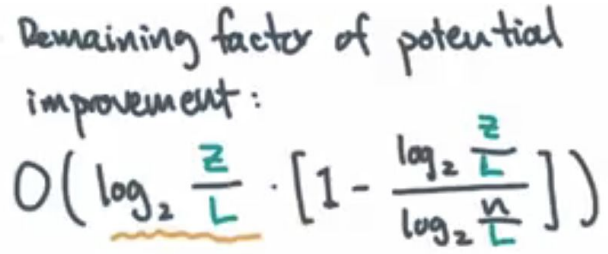
$Q(n;Z,L) = O(n/L \log_2(n/Z)) = O(n/L [\log_2(n/L) - \log_2(Z/L)])$

The lower bound:

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

Lower bound:

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



This difference can be quite high, depending on the system.

Why doesn't 2-way merge do a better at achieving lower costs?
Two-way merging is not good at utilizing fast memory ('Z') capacity. The merging procedure only works on pairs of arrays at a time and uses a block of size 'L'. So two-way merge is sensitive to 'L' but not sensitive to 'Z'.

You should be able to fix this issue.

Multiway Merging
To do better than 2-way merging, let's merge a lot of runs at once.

Assume:

K runs, stored in slow memory and sorted in ascending order
K + 1 blocks will fit in fast memory, $(K+1)L \leq Z$ (Z is size of fast memory)
(this will allow one block for each input and 1 block for the output)

1.Load the fast memory with k blocks.
2.Now find the smallest value of all the blocks, move it to the output block (the k +1 block).
The smallest value can be found a number of ways … a linear scan or min heap are two possibilities. The linear scan will work if K is small.
To use a priority queue (or min heap)
1.load the blocks
2.build the heap (cost O(K) operations)
3.extract Min (cost O(log K) operations)
4. insert (cost O(log k) operations)
These are all fast memory operations - so we can count them as comparisons.
3.When the output block if filled, flush it by writing it to slow memory.
4. If an input block is empty, refill it.

What is the cost of a K-way merge?

Transfers: read input blocks once, write blocks once: 2Ks/L

Comparisons: initial cost to build the heap, then each item is either inserted or extracted

O(K + Ks log K) .. for a single k-way merge

Cost of Multiway Merge

Initial input has n elements, divided into sorted runs, with z items each.
If you do k-way merges, the number of comparisons = n log(n)

What is the total number of asymptotic memory transfers?
Assume k = θ (z/L) < z/L

l = θ ($\log_{z/L}$ n/L) (l is maximum number of merge trees)

Transfers per run at i = θ (K^is/L)
# of runs at level i = n/K^is
Total transfers at level i = θ (n/L)
# of levels = θ ($\log_{z/L}$ n/L)

What is the total number of asymptotic memory transfers?
Total transfers at level i * # of levels = O(n/L * log(n/L, Z/L)

A Lower Bound on External Memory Sorting

Mergesort with $\theta(Z/L)$ -way merges: Q(n;Z,L) = θ (n/L $\log_{Z/L}$(n/L))

This is very good.

# of possible orderings: n!
orderings after t-1 transfers: K(t-1)
so K(0) = n!

Now, after a certain number of orderings, you real from slow memory to fast memory L items.
So there are L! ways of ordering this new items.
So now you have Z-L old items (already ordered) and L new items.
How many ways can these be ordered? $\leq$ $(Z \; choose \; L)L!$

After t reads, the lower bound on the number of orderings: K(t) $\geq$ K(t-1)/[(Z choose L)L!]^t
This is conservative, it assumes L is unordered.
If L has been read before ….

$$K(t) \geq \frac{n!}{\left(\frac{z}{L}\right)^t \cdot (L!)^{\frac{n}{L}}}$$

Now answer the question, when does only 1 ordering remain?
This is the lower bound on the number of transfers: