# 15853 Final Project Report

Rohan Yadav Rahul Jaisingh

## 1 Introduction

The RAM model of computation is the model where a processor can access any cell in memory in constant time. This model is the standard setting for algorithm design, and cost analysis under this model generally translates well to runtime on moden machines. However, as the data we run our algorithms on is quickly becoming huge, the standard RAM model of analysis diverges from reality. Modern machines have non uniform memory access times, in both onboard memories (cache heirarchies) and external memory (RAM vs disk). On these machines, accessing data that is not currently in cache, or in RAM can be order of magnitudes slower than accessing data that does reside in cache or RAM. As data becomes much larger than the size of the cache, or the size of RAM, these slowdowns can dominate the running time. To better analyze the running time of algorithms on these data sets, as well as to design algorithms to take advantage of this difference in memory access times, we analyze programs in the I/O model.

In the I/O model, we assume that our input has size $N$, and there exists the fast memory, which has size $M$, and is chunked into pieces of size $B$. We don't assume anything about the size of the slow memory. We consider all operations on data currently residing in $M$ to have cost 0. The only operations that have a cost are memory loads in $M$ from the slow memory. Data is loaded in $M$ from the slow memory in blocks of size $B$. We assume that the algorithm can choose which data to pull into $M$, and which blocks get evicted as well. This model more accurately captures the effect of different memory access times on modern machines, especially the discrepacny between RAM and disk access. Additionally, the I/O model is a cache-aware model, which is different from the cache-oblivious model, because in the I/O model we can assume that the algorithm has access to the parameters $M$ and $B$, where in the cache-oblivious model the algorithm does not.

In this report, we implemented and experimented with two I/O efficient sorting algorithms, and ran them on data that did not fit in RAM.

## 2 Sorting

Optimal sorting algorithms in the RAM model translate poorly to the I/O model, as they generally perform too much random access into the data, or do

not take advantage of the fast access of data residing in fast memory. The lower bound for sorting in the I/O model is $\Theta((N/B)\log_{M/B}(N/M))$ time, which is much faster than the lower bound for sorting in the RAM model. This bound is proven in lecture, and is a similar argument to the standard information theoretic argument for the RAM model lower bound. Standard sorting algorithms like Mergesort or Quicksort are able to achieve only $\Theta((N/B)\log_2(N/M))$ time in the I/O model, which is a large factor slower than the lower bound. To achieve the optimal bounds for the I/O model, we implement a K-way mergesort algorithm, as well as a buffer tree based sort.

# 3 K-way Mergesort

## 3.1 Introduction

The basic idea of the K-way Mergesort algorithm is to break the input sequence into more than two peices in the recursive calls. Doing this allows us to have the base of the logarithm be larger than 2. Specifically, we break our input sequence into $M/B$ peices, and perform a recursive I/O sort on each one. This allows the depth of the recursion to be $\Theta(\log_{M/B}(N/M))$, because we recurse until the data is size $M$. The merging of all $M/B$ sequences is also efficient, due to the number of sequences we are merging together. Because we can load $B$ elements into $M$ for each sequence, and we only need to read any element from each sequence once, we can have a block from each sequence in $M$ at once, and have fast access to a portion of every sequence that is being merged. Therefore, this entire merge step costs $N/B$, since every element must be loaded, and all other operations are free in memory. The recurrence generated by this algorithm is balanced, and results in the final bounds of $\Theta((N/B)\log_{M/B}(N/M))$.

## 3.2 Algorithm and Optimizations

More detailed psuedocode for our algorithm can be found in Figure 1. We made a few optimizations while implementing this algorithm. The first was to use a priority queue to hold an element from each of the $M/B$ sequences, rather than looping over all of them to find the minimum every time. While theoretically this operation is free, in practice $M/B$ can be large, and accessing all of these elements can be costly. One detail that we couldn't avoid was that we had to allocate extra output arrays, because the multi-way merge couldn't be done in place. This extra allocation in a sense pulls extra memory into $M$ and forces extra blocks to be evicted from $M$ during the merges. While this only affects the number of loads by a constant factor, there is slowdown during the runtime due to the extra memory usage.

```
fun kmergesort S =
  (* base case in memory *)
  if |S| <= M then return quicksort S else

  (* recursive case *)
  P = split into M/B pieces
  for p in P:
    kmergesort p

  O = allocate output sequence
  (* current index at each piece *)
  F = {0 for 0 ≤ i < M/B}

  for i in range(0, |S|):
    (m, j) = min element from P at each index
    O[i] = m
    F[j]++

  return O
```

Figure 1: K-way Mergesort Psuedocode

# 4 Buffer Tree Sort

## 4.1 Introduction

B-trees are I/O efficient data structures that provide good bounds on searching in the I/O model, and used heavily in practice. B-trees are trees that have a higher fanout that two, and also store more than a single element at each node. In the case of I/O efficient B-trees, the trees store $B$ elements at each node, and have $B$ children. This leads to fast lookups, as searching for an element takes $\log_B N$ time, and the search through every node takes only constant time, as the block has already been loaded into memory. However, if a standard B-tree is directly used in sorting, sub-optimal bounds are achieved — $\Theta(N \log_B(N))$. The reason for this is that insertions into the tree do not take advantage of the caching in the I/O model.

To achieve the optimal bounds, we use a data structure called a Buffer Tree, or a Buffered B-tree. A Buffer Tree is similar to a B-tree, but every internal node holds a buffer of size $M$, and every leaf node has a buffer of size $B$. Additionally, every internal node has $M/B$ children. Upon an insertion, an element is just added to the appropriate buffer, as opposed to moving down the tree. Once a buffer fills up, all of the elements are inserted into the appropriate children. In this way, operations are batched to be more efficient, and now a sequence of $N$ insertions costs $\Theta(N/B \log_{M/B}(N/B))$. To recover a sorted sequence, we can simply walk across the leaves of the tree. Because this final step takes only $N/B$

```
fun rec_insert(T, e) =
  insert e into T->buffer
  if T is a leaf: return
  if T->buffer is full:
    sort(T->buffer)
    for each d in T->buffer:
      c = child of T that d belongs in
      rec_insert(c, d)
      if c is too large:
        split(c) into c1 and c2
        insert c1 and c2 into T's children

fun insert(T, e) =
  T' = rec_insert(T->root, e)
  if T'->root is too big:
    R = new root
    R->children = split(T)
    return R
  return T'
```

Figure 2: Buffer Tree insertion algorithm

time, we can use a Buffer Tree to construct an I/O optimal sorting algorithm.

## 4.2 Algorithm and Implementation

The main difficulty in our implementation was the insertion function for our buffer trees. Searching and writing out the sorted values to the output sequence were straightforward. The difficulty in insertion was mainly the internal balancing steps. Psueodocode for our insertion algorithm can be found in Figure 2. To effectively sort, we also needed to add in a flush style operation. The main issue without an operation like this is that the buffers themselves are not sorted, and elements can be stuck in the buffers instead of being sorted at the leaves, by the time that all $N$ elements have been inserted. To recover a sorted sequence, we need to flush all of the buffers to make sure that all of the elements that in the leaves, so that we can just walk over the leaves to output the sorted elements. To do the flush, we just sort the current buffer, and then recursively insert each element into its corresponding child, and then flush each child. After this operation, all of the buffers will be flushed, and all of the elements will be sorted at the leaves.

| Input Size | Standard | K-Merge | Buffer Tree |
|---|---|---|---|
| 2 GB | 6309 | 3919 | 6625 |
| 4 GB | 130721 | 78673 | 249098 |
| 8 GB | 252504 | 148170 | 493306 |

Figure 3: Results of I/O sorts (Time in Seconds)

# 5 Results

## 5.1 Experimental Details

We implemented our algorithms in C++, and compiled with `O3` level optimization. We compared against the sorting algorithm provided by the `<algorithm>` library in the STL. We additionally used the same sorting algorithm as the base case for our sorting algorithms when the data fit into memory. We tested on an AWS EC2 instance with a single core Intel Xeon CPU @3.30 GHz and 0.5 GB of RAM. Additionally, we outfitted the instance with a 50 GB swap file, so allocations larger than RAM are allocated onto the swap file, which resides in external memory. Allowing the instance to have only 0.5 GB of RAM allowed for easier testing, as we were able to allocate much smaller data and have it still spill out of on board memory.

## 5.2 K-way Mergesort Results

The results of our k-way mergesort algorithm can be found in Figure 3. We ran these experiments with $M = 0.5$ GB and $B = 16$ KB. While the page size on the instance was 4 KB, we saw better results with a larger $B$ value. We postulate that this was the case due to the fewer number of recursive calls that must be made by the sort, which could incur a large amount of overhead, as $M/B$ is large in these cases. We can see that our implementation performs well, consistently out-performing the sort provided in the STL, and even doubling the performance on the 2 and 4 GB input sizes.

## 5.3 Buffer Tree Sort

The results of our buffer tree sort can be found in Figure 3. We ran these experiments with $M = 0.5$ GB and $B = 16$ KB. As with the K-way merge algorithm, we ran with $B$ being larger than the actual page size. We did not see good results with the buffer trees, as can be seen in the table above. For the tests we ran on, the buffer tree sort was at least as slow as the standard sort, or much worse. We believe that this slowdown comes from the overly strong assumptions of the I/O model. The model assumes that all accesses in fast memory are free, and thus the buffer tree can do operations like searching, sorting and insertions for free. However, on a real machine, these are still non-negligible costs, and we believe that large number of sorted insertions, buffer sorts, and searches the buffer tree must do leads to a slowdown in practice.

# 6 Conclusions

In conclusion, we can see that for at least k-merge, algorithms that are optimal in the standard model do not accurately account for non uniform memory access costs. By analyzing algorithms even in the I/O model, which assumes a simple memory hierarchy, and user control over memory, we can write algorithms that heavily outperform even the standard libraries of C++. In the case of buffer trees, we can see where the theoretical model of I/O efficiency makes assumptions that are too strong, and that an optimal data structure in the I/O model did not translate well to real machines. Our code can be found at `https://github.com/rohany/15853-final-project`.

# 7 References

For all references we consulted the course notes of 15853.