Examples of External Memory Analysis

Sam McCauley

Written Oct 5 2021; Last Edited October 5, 2021

The purpose of this document is to give examples of how to analyze algorithms in the external memory model. While you've seen a number of these examples in class, this is a version of those (and other) examples in a more formal written form.

1 Reviewing the External Memory Model

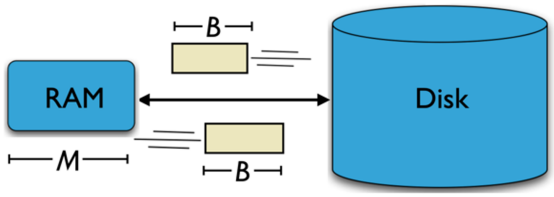
In modern computing, cache performance is often *more important* than number of operations This is at odds with classic algorithmic analysis, which measures number of operations. The external memory model is a way to algorithmically analyze the cache performance of a given approach.

The idea of the external memory model is that it should carry over many of the advantages of classic algorithmic running time analysis. The analysis should be relatively simple, and should be platform-independent. External memory analysis provides a performance guarantee—that is to say, it is a worst-case analysis. We will use asymptotic notation in our analysis; as with classic algorithmic analysis this allows us to give a high-level, platform-independent view of performance without getting bogged down by details that don't affect bottom-line performance.

However, cache parameters differ massively between computers, and these parameters can have a massive impact on performance.[1] With this in mind, we need to use two parameters in our analysis:

- We use M to denote the size of the cache.
- We use B to denote the size of a cache line. We always have $B \ll M$.

With this in mind, we are ready to define our model. The basic unit of cost in our model is a *cache miss*. A cache miss transfers B *consecutive* items to or from the disk, for a cost of 1.



In the external memory model, *all computation in cache is free*. You can perform arbitrary computations on all items in cache. You can also rearrange them however you want. The only restriction is that data must be read from, and written to, disk in cache lines of size B—where each such read or write costs 1.

Let M be the number of items we can store simultaneously in our cache. That means that when we bring a new cache line of B items into cache, we must write B items back to disk. (We can bring them in later if we need them again.) Note that these B items must be written *consecutively* to disk.

[1] For example, we will see that whether or not a problem entirely fits in cache can have an enormous impact on cache performance.

Choosing What Fits in Memory With the above in mind, the decision of what's stored in cache—that is to say, when a new cache line comes in, the decision of what B items to evict and what $M - B$ items to retain—is crucially important to cache performance. In the external memory model, we assume that the computer does this *optimally*. That means that when we analyze an algorithm, we can state exactly what items we think the computer ought to store in cache, and we can assume that the computer follows through on this. In practice, computers use an incredibly effective low-resource algorithm called CLOCK (which is very similar to the Least Recently Used (LRU) policy, which evicts the B least recently accessed items); this algorithm is very close to optimal for most practical workloads.

Vocabulary

- "Cache" of size M; "disk" of unlimited size
- With the cost of one "cache miss" can bring in B consecutive items
  - (Sometimes called "memory access" or "I/Os" but I will try not to use those terms.)
- These B items are called a "block" or a "cache line".

2 Simple Examples

Finding the minimum element in an array Consider a scan over an unsorted array A to find the minimum element in A. This algorithm accesses the elements of A in order: $A[0], A[1], \ldots A[n-1]$, updating the minimum after each access.

Lemma 1. *Finding the minimum element of an array of n elements using the above approach requires $O(1 + n/B)$ cache misses.*

Proof. In total, this algorithm accesses n elements of A in order: $A[0], A[1], \ldots, A[n-1]$.

When accessing element $A[0]$, elements $A[1] \ldots A[B-1]$ will all be brought into cache (as they fit in the same cache line). Therefore, after accessing $A[0]$, there will not be any cache misses until $A[B]$. In general, if there is a cache miss on $A[i]$, the next cache miss will be on $A[i + B]$.

Therefore, there will be a cache miss when accessing $A[0], A[B], A[2B], \ldots, A[\lfloor (n-1)/B \rfloor]$. This is $O(n/B)$ cache misses overall. □

Note that accessing an array of $n < B$ elements requires only one cache miss (after which the whole array is in cache and computation is free).

Binary Search

Lemma 2. *A binary search on an array of size n requires $O(1 + \log(n/B))$ cache misses.*

Proof. Binary search is recursive, so let's solve this with a recurrence.

Each recursive call to binary search requires 1 cache miss (to look up the middle element), after which there is a call to the same problem on size $n/2$.

Base case: a call to binary search on an array of size $n < B$ requires 1 cache misses.

So in sum, we have:

$$\begin{aligned} T(n) &= T(n/2) + O(1) && \text{if } n \geq B \\ T(n) &= O(1) && \text{if } n < B \end{aligned}$$

Solving this equation, we have $T(n) = O(1 + \log n/B)$. (The $+1$ term comes from the fact that we always incur the cost of the base case, even if $n < B$.) □

3 Matrix Multiplication in External Memory

First, let's examine a simple three-loop algorithm for matrix multiplication, as shown in Figure 1.

Lemma 3. *Assume that $n \gg M$ and $n \gg B$ (i.e. n is much larger than the size of cache or the size of a cache line), and assume that all three matrices are stored in row-major order.*[2] *Then multiplying two $n \times n$ matrices using the approach in Figure 1 requires $O(n^3)$ cache misses in the external memory model.*

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

Figure 1: A simple implementation of matrix multiplication.

Proof. Let's break the cost down into three parts: the cost for all accesses in A, in B, and in C.

Cost for all accesses to A: Consider the cost of the inner loop accessing A (the loop over k). Successive accesses to the loop access successive elements in a row of A. Let's assume that, for some i we have a cache miss on $A[i][k]$; since these elements are stored consecutively in memory we won't have another cache miss until we access $A[i][k+B]$. In other words, we have cache misses on $A[i][0]$, $A[i][B]$, $A[i][2B]$, and so on. Therefore, each inner loop requires $O(n/B)$ cache misses.

Each iteration of the middle loop (over j) performs the above n times. Because $n \gg M$, we do not have space to store all of the ith row of A in memory. Therefore, we will again incur n/B cache misses for each iteration of j. In total, over all iterations of the middle loop, we incur $n \cdot n/B$ cache misses.

Finally, each iteration of the outer loop (over i) repeats the above for each row of A. There are n rows, so the total cost to access A is n^3/B.

Cost for all accesses to matrix B: Consider the cost of the inner loop accessing B (the loop over k). Successive accesses to the loop access elements in different rows of B. When accessing $B[k][j]$, we can bring in B elements from row k of B. However, our next access is to $B[k+1][j]$, incurring a cache miss. In total, each new k accesses a new row of cache, causing n cache misses for each iteration of the inner loop.

Each iteration of the middle loop (over j) performs the above n times. Because $n \gg M$, we do not have space to store a cache line from all of the nth rows of B in memory. Therefore, we will again incur n cache misses for each iteration of j. In total, over all iterations of the middle loop, we incur $n \cdot n$ cache misses.

Finally, each iteration of the outer loop (over i) repeats the above n times. This gives a total cost of $O(n^3)$ cache misses.

Cost for all accesses to matrix C: Consider the cost of the inner loop accessing C (the loop over k). All of these accesses access $C[i][j]$. If $C[i][j]$ is not in cache this requires one cache miss; otherwise it requires 0.

Each iteration of the middle loop (over j) iterates over the elements in row i of C. As above, if we have a cache miss on $C[i][j]$, there will not be a cache miss until $C[i][j+B]$. Summing, the middle loop requires a total of n/B cache misses.

Finally, each iteration of the outer loop (over i) repeats the above n times; once for each row of C. This gives a total cost of $O(n^2/B)$ cache misses.

Total: Summing the above costs, we obtain $O(n^3 + n^3/B + n^2/B) = O(n^3)$ cache misses. □

[2] This means that each row of a matrix is stored consecutively in memory.

To improve the cache efficiency of the simple matrix multiplication algorithm in Figure 1, we need to improve the cache efficiency of accesses to matrix B. One way to do this is to change how we store matrix B—perhaps we could store it in column-major order for example. But, a classic observation is that we can easily improve the cache efficiency by swapping the middle and innermost loop, obtaining Figure 2. Note that we are iterating over the exact same elements and doing the same multiplications, so the algorithm remains correct.

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

Figure 2: Swapping the two innermost loops to improve cache efficiency.

The following analysis is extremely similar to the analysis of Lemma 3.

Lemma 4. *Assume that $n \gg M$ and $n \gg B$ (i.e. n is much larger than the size of cache or the size of a cache line), and assume that all three matrices are stored in row-major order.*[3] *Then multiplying two $n \times n$ matrices using the approach in Figure 2 requires $O(n^3/B)$ cache misses in the external memory model.*

Proof. Let's break the cost down into three parts: the cost for all accesses in A, in B, and in C.

Cost for all accesses to A: Consider the cost of the inner loop accessing A (the loop over j). All of these accesses access $A[i][k]$. If $A[i][k]$ is not in cache this requires one cache miss; otherwise it requires 0.

Each iteration of the middle loop (over k) iterates over the elements in row i of A. If we have a cache miss on $A[i][k]$, there will not be a cache miss until $A[i][k+B]$. Summing, the middle loop requires a total of n/B cache misses.

Finally, each iteration of the outer loop (over k) repeats the above n times; once for each row of A. This gives a total cost of $O(n^2/B)$ cache misses.

Cost for all accesses to matrix B: Consider the cost of the inner loop accessing matrix B (the loop over j). Successive accesses to the loop access successive elements in a row of B. Let's assume that, for some i we have a cache miss on $B[k][j]$; since these elements are stored consecutively in memory we won't have another cache miss until we access $B[k][j+B]$. In other words, we have cache misses on $B[k][0]$, $B[k][B]$, $B[k][2B]$, and so on. Therefore, each inner loop requires $O(n/B)$ cache misses.

Each iteration of the middle loop (over k) repeats the above for each row of B. There are n rows, so the total cost for the middle loop is $O(n^2/B)$.

Finally, the outer loop performs the above n times, for $O(n^3/B)$ memory accesses.

Cost for all accesses to matrix C: Consider the cost of the inner loop accessing C (the loop over j). Successive accesses to the loop access successive elements in a row of C. Let's assume that, for some i we have a cache miss on $C[i][j]$; since these elements are stored consecutively in memory we won't have another cache miss until we access $C[i][j+B]$. In other words, we have cache misses on $C[k][0]$, $C[k][B]$, $C[k][2B]$, and so on. Therefore, each inner loop requires $O(n/B)$ cache misses.

Each iteration of the middle loop (over k) repeats the above n times. Since $n \gg M$, there is not enough space to store the entire row in cache, so each cache line may need to be brought into cache again as it is reached. In total, the inner loop requires $O(n^2/B)$ cache misses.

Finally, each iteration of the outer loop (over i) repeats the above n times; once for each row of C. This gives a total cost of $O(n^3/B)$ cache misses.

Total: Summing the above costs, we obtain $O(n^3/B + n^2/B + n^3/B) = O(n^3/B)$ cache misses. □

[3] This means that each row of a matrix is stored consecutively in memory.

3.1 Using the Cache More Effectively

None of the above algorithms use the cache—there is no M in any running time except when the whole problem fits in cache. The reason is that each time something is brought into cache, it's used once and then thrown away. (With some small exceptions—for example, each cache line of C in the proof of Lemma 3 is used in nB multiplications before being thrown away. But the large number of cache misses for A and B dominate the overall performance.)

If we want to use cache properly, we need to bring a set of items into cache such that we can perform many computations on them before writing them back. Note that this is not always possible. For example, when finding the minimum, we only need to check each item once—after we see if it's smaller than the smallest item we've seen so far, there's nothing to do with it except throw it away.

Blocking Blocking is an algorithmic technique to improve cache efficiency. Blocking follows the following strategy:

1. Split the problems into subproblems that fit in cache (so we want subproblems of size $O(M)$)
2. Solve these subproblems in cache one by one
3. Combine the solution to these subproblems to obtain the overall solution

3.2 Using Blocking for Matrix Multiplication

The basic idea of blocked matrix multiplication is to split A, B, and C into blocks of size $M/3$ (that way, one block from each can fit in cache simultaneously). This means that we want $T \times T$-sized blocks, where $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for simplicity.

A classic result states that we can multiply these blocks one at a time, as if we were multiplying $n/T \times n/T$ matrices, and obtain the correct final solution. This leads to the algorithm shown in Figure 3

Perhaps surprisingly, this approach has a somewhat shorter analysis than the simpler algorithms: we can just multiply the number of cache misses for a single call to `BlockMultiply()` by the number of such calls.

Lemma 5. *Assume that $n \gg M$ and $n \gg B$ (i.e. n is much larger than the size of cache or the size of a cache line), and assume that all three matrices are stored in row-major order. Then multiplying two $n \times n$ matrices using the approach in Figure 3 requires $O(\frac{n^3}{B\sqrt{M}})$ cache misses in the external memory model.*

Proof. First, consider the cost of `BlockMultiply()`. We call `BlockMultiply` with three $T \times T$ matrices as arguments; let's call them A', B', and C'. Since A', B', and C' fit in cache simultaneously, we can simply read all three of them in, perform the multiplication in cache, and write out the result to C. All three have size $T^2 = O(M)$, so reading them in row by row requires $O(T+M/B)$ cache misses. (Each cache miss reads B more items in from one of the matrices; the $+T$ term comes from incurring at least one cache miss for each row.)

```
MatrixMultiply(A, B, C, n, T):
  for i = 1 to n/T:
    for k = 1 to n/T:
      for j = 1 to n/T:
       A' = TxT matrix with upper left corner
            A[Ti][Tk]
       B' = TxT matrix with upper left corner
            B[Tk][Tj]
       C' = TxT matrix with upper left corner
            C[Ti][Tj]
       BlockMultiply(A', B', C', T)

BlockMultiply(A, B, C, n):
  for i = 1 to n:
    for k = 1 to n:
      for j = 1 to n:
        C[i][j] += A[i][k] + B[k][j]
```

Figure 3: Blocked matrix multiplication for better cache efficiency

We make $(n/T)^3$ calls to `BlockMultiply()`. Therefore, the total cost is:

$$O\left(\left(\frac{n}{T}\right)^3 \cdot (T + M/B)\right) = O\left(\left(\frac{n^3}{T^2} + \frac{n^3 M}{BT^3}\right)\right) = O\left(\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)\right)$$

Oftentimes, we assume that $M > B^2$ (almost certainly true in practice). In this case, each call to `BlockMultiply` can be simplified to $O(M/B)$ cache misses, leading to a final running time of $O(\frac{n^3}{B\sqrt{M}})$. □

4 Sorting in External Memory

First, let's analyze some important sorting subroutines.

Lemma 6. *Paritioning an array A of length n requires $O(1 + n/B)$ cache misses.*

Proof. Consider a wasteful partitioning algorithm that uses an extra array A' of size n. This algorithm first looks at $A[0]$, and sets two variables `low`$= 0$ and `high`$= n - 1$. Iterate through $A[i]$ for $i = 1$ to $n - 1$. If $A[i]$ is lower than $A[0]$, we put $A[i]$ into slot A[`low`] and increment `low`; otherwise put $A[i]$ into slot A[`high`] and decrement `high`.

Assume we have a cache miss at $A[i]$ for some i. Since we access the items consecutively, we will not have another cache miss until $A[i + B]$. We have a cache miss at $A[0]$, for $O(1 + n/B)$ cache misses overall.

Similarly, we can write B consecutive values of `low` using a single cache miss; we can also write B consecutive values of `high` using a single cache miss.[4] We're only accessing n items of A' in total, so this requires a total of $O(1 + n/B)$ cache misses.

Summing, we obtain $O(1 + n/B)$ cache misses.

Note that most implementations of partition (for example, clever in-place implementations) will usually also require only $O(n/B)$ cache misses. □

Applying Lemma 6 to the classical quicksort analysis gives $O(\frac{n}{B} \log_2 \frac{n}{B})$ cache misses in expectation and with high probability on an array of size $n > B$, and a worst case of $O(n^2/B)$ cache misses. (This analysis isn't included here since it is just the classical probabilistic analysis combined with Lemma 6.)

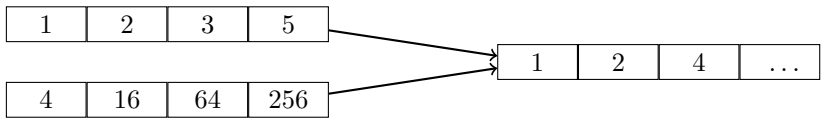


Figure 4: Merging two sorted arrays into a single sorted array

Lemma 7. *Assume that $M > 3B$. Then merging two sorted arrays A, B, each of length n, requires $O(1 + n/B)$ cache misses.*

Proof. Consider a wasteful merge algorithm that creates an array C of length $2n$ and maintains a pointer to each of A and B. At each point, it takes the smaller element being pointed to by either pointer and writes it to C; the corresponding pointer is incremented.

Let's look at the number of cache misses for each array individually. A is scanned over once. Assume that we always keep one cache line from A in our cache. Then when we have a cache miss for $A[i]$ we will not have another cache miss until $A[i + B]$; giving $O(1 + n/B)$ cache misses in total; the same analysis applies for B and C (each of which also need to store a cache line in cache—we have room for this since $M > 3B$. Totaling, we have $O(1 + n/B)$ cache misses. □

With a bound on the cost to merge we can analyze the number of cache misses incurred by merge sort.

[4]Notice that `high` is working backwards. That's OK—you can still bring in a cache line *ending* at the element we're accessing.

Lemma 8. *Merge sorting an array A of size $n > B$ requires $O(\frac{n}{B} \log_2 \frac{n}{B})$ cache misses.*

Proof. An iteration of merge sort on an array A consists of three steps:

1. Break A into two equal parts A_1 and A_2
2. Recursively sort A_1 and A_2
3. Merge the sorted A_1 and A_2 back into A.

Breaking A into two parts can be done in $O(1 + n/B)$ cache misses. (Of course, this depends on the implementation. If A_1 and A_2 are created as new arrays, and each element of A is copied into one or the other, then $O(1+n/B)$ cache misses are required as in Lemma 6. But, if A_1 and A_2 are passed using pointers into portions of A, this can potentially be done with no cache misses at all. This choice does not affect the final running time.)

We'll include the cost of the recursion when we write our recurrence relation. Merging A_1 and A_2 takes $O(1 + n/B)$ cache misses from Lemma 7.

Writing our recurrence relation, we obtain

$$T(n) = 2T(n/2) + O(n/B) \text{ if } n > B$$
$$T(n) = O(1) \text{ otherwise}$$

Drawing the recurrence tree, we can see that the total work is $O(n/B)$ at every level. The number of levels is equal to the number of times we need to divide n by 2 until we obtain an array size smaller than B. Solving $n/2^k = B$, we obtain $k = \log_2 n/B$. Summing the cost of each level over all levels, we obtain

$$\sum_{\ell=0}^{\log_2 n/B} O(n/B) = O(\frac{n}{B} \log_2 \frac{n}{B})$$

□

Using the cache The above algorithms have only used cache lines, without storing large amounts of data in the cache. Can we take better advantage of the cache?

It's not quite clear how to use blocking here. Sure, we could take subarrays of size $O(M)$ into cache and sort them. But how can we combine these subproblems together? Lemma 7 would work, but its cache efficiency is limited.

A practice exercise along these lines:

1. Show that the approach above (sorting subarrays of size $O(M)$, then merging them two at a time) requires $O(\frac{N}{B}(1 + \log_2 \frac{N}{M})$ cache misses.

Instead, we want to use the cache *while merging*. Normal sort compares two items at a time. But with a large cache, we can compare far more items than that with a single cache miss. This brings us to an optimal external-memory sorting algorithm: M/B-way merge sort.

M/B-way merge sort:

- Divide array into M/B equal parts
- Recursively sort all M/B parts
- Merge all M/B arrays in $O(n)$ time (and $O(n/B)$ cache misses)

Figure 5: Methodology for M/B-way merge sort

The heart of the M/B-way merge sort is the method to merge M/B subarrays cache efficiently. Let's go into more detail about how that works. At any point in time, let's keep B slots in cache for each subarray we're merging. (There are M/B subarrays, so this is $\leq M$ total items.) These B slots may not all be full; however, if k of these slots are full, they must hold the k smallest unmerged values from the corresponding subarray.

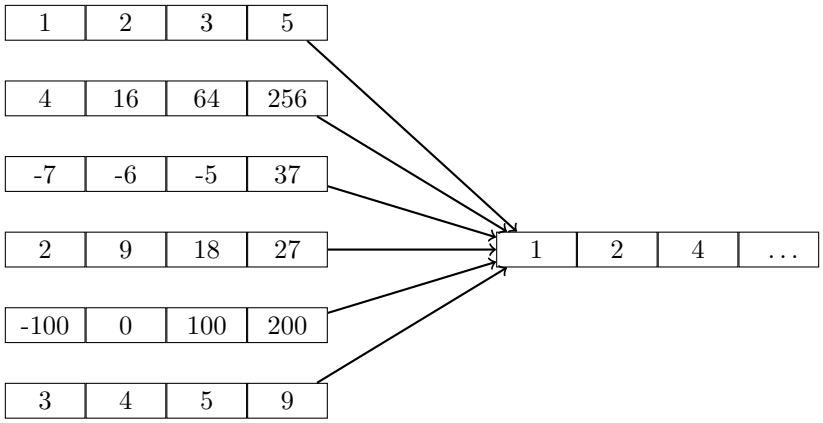


Figure 6: Diagram of M/B-way merge sort

The merge proceeds as follows. Consider B memory slots in cache that we'll use to write the solution out to the final array.[5] Let's fill these slots up by repeating the following procedure B times:

1. Find the smallest item currently in the cache (ignoring any items previously selected for writing), and move it to the next empty write slot
2. Assume that this item comes from subarray i. If there are no more items from i currently in cache, bring the next B smallest items from i into cache

After the B write slots are filled, they must contain the B smallest unmerged item. Write these B items out to the final solution array and repeat.

Lemma 9. *Merging M/B arrays into a single array (of length n) using the above approach requires $O(M/B+n/B)$ cache misses.*

Proof. Each of the subarrays being merged is scanned once, with B elements being brought into cache at a time; therefore, if each subarray has size n', the cost to merge each is $O(1+n'/B)$. Substituting $n' = nB/M$, the merge requires $O(M/B + n/B)$ cache misses to scan all subarrays. The solution array is written to B elements at a time, so it requires $O(1 + n/B)$ cache misses. Summing obtains the lemma. □

As with normal merge sort, the time required to do a single merge is most of our analysis.

Lemma 10. *Performing M/B-way merge sort on an array of size n requires $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ cache misses.*

Proof. Integrating Lemma 9 into our recursion, we obtain

$$T(N) = \frac{M}{B}T\left(\frac{N}{M/B}\right) + O(N/B) \qquad \text{if } n > M$$
$$T(N) = O(1 + N/B) \qquad \text{otherwise}$$

Let's solve this recurrence. Writing out the recursion tree, each level of the tree requires a total of $O(N/B)$ cache misses. The number of levels is the number of times that we need to divide n by M/B until M is reached; plus 1 for the base case itself. Summing over each level ℓ, we obtain:

$$\sum_{\ell=1}^{1+\log_{M/B} n/M} O(n/B) = O\left(\frac{n}{B}(1 + \log_{M/B}\frac{n}{M}\right) = O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right).$$

□

[5]Note that we need an extra B slots in cache for this to work. In this case, we should probably use $(M/B-1)$-way merge sort to make sure we have room for these slots. We'll ignore this off-by-one error to keep things simple.