6.895 Theory of Parallel Systems Lecture 21

Sorting and Permuting on Sequential and Parallel Disks

Lecturer: Michael Bender *Scribe: Pham Duc Minh and Jim Sukha*

Lecture Summary

1. *DAM Model*
   This section introduces the DAM memory model and presents the asymptotic bounds for the number of memory transfers required to sort and permute N records on disk.
2. *I/O Analysis of Mergesort*
   This section presents an analysis of the number of I/O's required for serial and parallel mergesort.
3. *Sorting Lower Bound*
   This section proves the lower bound for the number of memory transfers required for sorting N records.
4. *External Permutation Lower Bound*
   This section proves the lower bound for the number of memory transfers required for permuting N records.
5. *Cache-Oblivious Sorting*
   This section briefly describes a cache-oblivious sort, funnelsort.

1 DAM Model

Consider the problem of sorting N records stored on a disk. To analyze the time required for this problem, we use the DAM model proposed by Aggarwal and Vitter [1]. This model assumes that M records can fit into internal memory, and B records (blocks) can be transfered between internal memory and disk in one memory transfer. The data is stored on P disks, so we can concurrently transfer P blocks in one unit of time. The setup of the DAM model is illustrated in Figure 1.

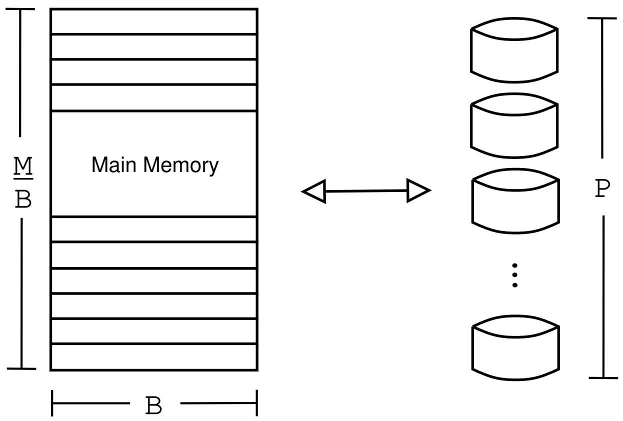

Figure 1: Setup of the DAM model.

Our goal is to minimize the number of memory transfers required to sort N records. Since the time required for an I/O is many orders of magnitude larger than the time required for an internal memory access, in this model, we neglect the cost of manipulating a block already inside internal memory.

The DAM model differs somewhat from the cache-oblivious memory model proposed in [2] because the DAM model requires the user to manage memory explicitly. In the cache-oblivious model, parameters such as the block size are unknown, and the system undertakes the burden of memory management. The cache-oblivious model also only considers the case $P = 1$.

We have asymptotically tight bounds on the number of memory transfers required for the problems of sorting and permuting N records:

Theorem 1 *The average-case and worst-case cost to sort N records is* $\Theta\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right)$ *block transfers.*

Theorem 2 *The average-case and worst-case cost to permute N records is* $\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right\}\right)$ *block transfers.*

In Section 3 and Section 4, we prove the lower bounds in Theorem 1 and Theorem 2, respectively.

Note that these bounds involve expressions with the term $\log(1 + N/B)$ or $\log(1 + M/B)$. Although the 1 is technically required, for clarity, in the sections that follow we omit the 1 in our analysis.

2 Mergesort Analysis

To get a sense of the complexity of the bound stated in Theorem 1, in Section 2.1, we analyze the number of memory transfers required for a normal merge sort algorithm when $P = 1$. Although this simple algorithm is not optimal, in Section 2.2 we show that the optimal number of memory transfers can be attained using a parallel mergesort.

2.1 Normal Mergesort

We first analyze the performance of a normal mergesort using the DAM model. A simple mergesort algorithm, illustrated in Figure 2, works by splitting the N records into half, recursively sorting each half, and merging the two sorted lists. As we see in Theorem 3, the cost of this algorithm is not optimal.

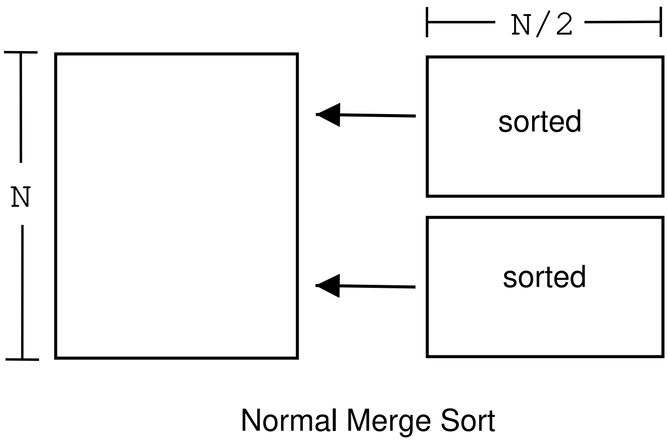

Figure 2: A simple mergesort algorithm.

Theorem 3 *Normal merge sort requires* $O(\frac{N}{B}\lg\frac{N}{B})$ *memory transfers.*

Proof The number of memory transfers satisfies the recurrence

$$T(N) = 2T\left(\frac{N}{2}\right) + O(\frac{N}{B}) \tag{1}$$
$$T(B) = 1. \tag{2}$$

In the recursive step we split the problem into two subproblems of half the size. The merging operation requires $O(N/B)$ memory transfers because we scan both arrays of length $N/2$ once to read in the data, and we require only $O(N/B)$ transfers for the output of the merge. In the base case a problem of size B will certainly fit into internal memory, so we sort the block using only one memory transfer to read the block in.

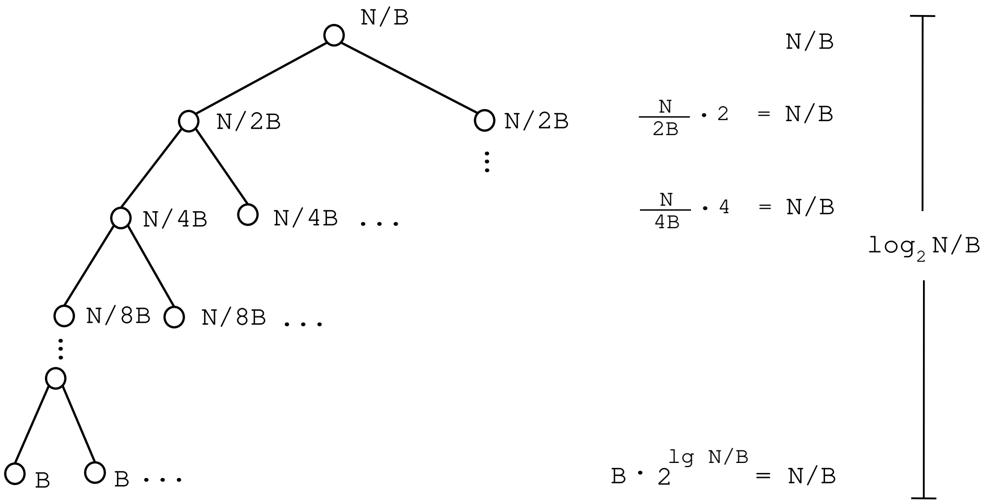

Figure 3: Recursion tree for normal mergesort.

This recurrence can be solved using a recursion tree, as illustrated in Figure 3. At each level, there are twice as many problems of half the size compared to the previous level. Therefore, the cost at each level is N/B. If we recursed all the way to a base case of 1, then we would have $\lg N$ levels. However, since we stop at problems of size B, we only have $\lg N - \lg B$, or $\lg(N/B)$ levels. This analysis gives us the stated bound of $O(N/B \lg(N/B))$. □

Note that this analysis is actually independent of the size of internal memory, M. With a more clever algorithm we might be able to improve the bound slightly, but it would still not match the optimal bound from Theorem 1. The normal mergesort is off by a factor of $\lg(M/B)$ from optimal.

2.2 Parallel Mergesort

We improve the simple mergesort by increasing the number of arrays we merge at once. Consider the algorithm illustrated in Figure 4 that divides the array into M/B problems and recursively sorts arrays of size $\frac{N}{M/B}$. This parallel mergesort for the case $P = 1$ requires an optimal number of memory transfers.

Theorem 4 *Parallel mergesort requires* $\Theta\left(N/B \log_{N/B}(N/B)\right)$ *memory transfers.*

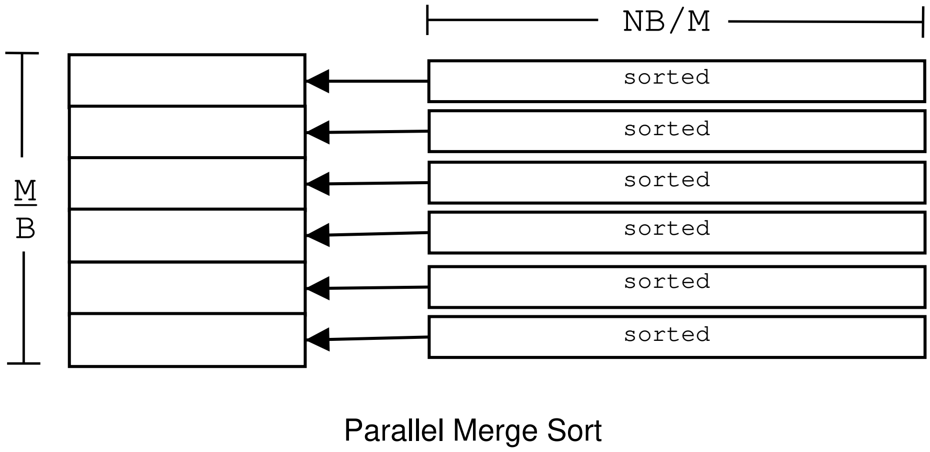

Figure 4: A parallel mergesort algorithm.

Proof The algorithm satisfies the recurrence

$$T(N) = \frac{M}{B}T\left(\frac{N}{M/B}\right) + O\left(\frac{N}{B}\right) \tag{3}$$

$$T(B) = 1. \tag{4}$$

The recursive step splits the problem into M/B problems, each of size $N/(M/B) = NB/M$. As with the normal mergesort, the merge step for each problem requires $O(N/B)$ memory transfers because we only need to scan through each of the sorted sub-arrays once.

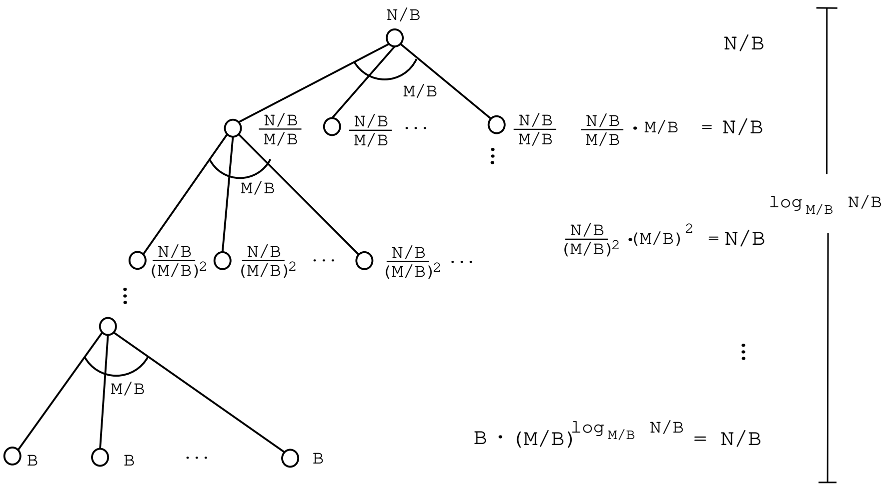

Figure 5: The recursion tree for parallel mergesort.

In the recursion tree shown in Figure 5, we see that each level still has cost $O(N/B)$. However, because we have more subproblems, the number of levels required is now only $\log_{M/B} N - \log_{M/B} B$. Adding up the costs of all the levels, we find that the number of memory transfers required is $\Theta\left(\frac{N}{B}\log_{N/B}\left(\frac{N}{B}\right)\right)$. □

The bound for this mergesort algorithm is equal to the optimal bound shown in Theorem 1. However, this particular parallel mergesort does not immediately work for $P > 1$ if P is not constant. In this case, it becomes important to specify how exactly the merge step is being done. Although we do not present the details here, it is in fact possible to make the parallel mergesort work for non-constant P.

3 Lower Bound for Sorting

In this section we prove the lower bound from Theorem 1 for the number of I/O's required to sort.

Theorem 5 *External sorting requires* $\Omega\left(\frac{N}{B}\log_{M/B} N/B\right)$ *I/O's.*

Proof This bound assumes a comparison-I/O model, where the only operation we are allowed to use in internal memory is to compare values. With this assumption, we prove the theorem using an information theoretic argument. At the beginning of the computation, we have no information about the data, so all $N!$ orderings of the data are consistent with what we know. After each memory transfer, we learn through comparisons, so we reduce the number of possible orderings by eliminating those that are inconsistent with the new information. We have enough information to sort the list when only one ordering is consistent with all our information. To prove the theorem, we show that we need $t = \Omega\left(N/B \log_{M/B} N/B\right)$ memory transfers to narrow down the possibilities to one consistent ordering.

Stating this argument more formally, let S_i denote the set of possible orderings of the N elements that are consistent with the information we have after i memory transfers. A lower bound on $|S_i|$ is the smallest the set of consistent orderings can be after i memory transfers.

At the beginning of the computation, $|S_0| = N!$. We want to find the smallest i such that our lower bound is $|S_i| \geq 1$. This i represents the smallest number of memory transfers we can do while still getting enough information to eliminate all but one ordering.

After each memory transfer, we compute a new lower bound for the size of S_i. There are two types of memory transfers:

- *Case 1*: We know the order of elements in internal memory, but not the order of the block B being input.

  In this case, we learn the correct ordering of the elements within the block. Before the transfer, there were $B!$ possible orderings of the elements, but after the transfer we can sort the block and eliminate all but one of these orderings.

  We also learn where to interleave the elements in the block into internal memory. Before the memory transfer, the B elements in the block could have been interleaved between the other $M - B$ elements in internal memory arbitrarily. Using a stars and bars argument, the number of possible interleavings is the number of ways to place B bars between $M - B$ stars, or $\binom{M-B+B}{B}$. After the input we can eliminate all but one interleaving. This case is illustrated in Figure 6.

  Combining these two pieces of information, we know before the memory transfer, there are

  $$B!\left(\frac{M-B+B}{(M-B)!B!}\right) = B!\binom{M}{B}$$

  possible orderings of the elements in internal memory consistent with our information. After the transfer we reduce this to just one.

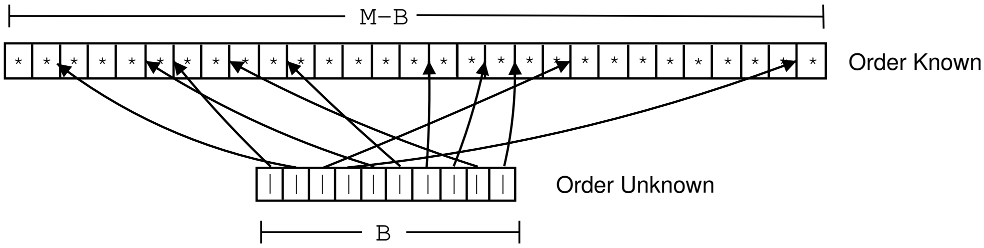

Figure 6: Case 1: the order of elements in B is unknown. After a memory transfer we learn the order of elements in the block and the correct place to interleave them into internal memory.

Recall that

$$|S_i| = \text{ \# of consistent orderings after } i \text{ memory transfers.}$$

From our analysis, we know that our lower bound on the size of this set changes according to the formula

$$\begin{aligned} B!\binom{M}{B}|S_i| &\geq |S_{i-1}| \\ |S_i| &\geq \frac{|S_{i-1}|}{B!\binom{M}{B}}. \end{aligned}$$

If we start with $|S|$ consistent orderings, then after t transfers of this type, the number of remaining orderings is at least

$$|S_t| \geq \frac{|S|}{\left(B!\binom{M}{B}\right)^t}. \tag{5}$$

- Case 2: The order of records in both main memory and the input block are already known. This can happen if the input block was previously output to external memory.

  In this case, we only learn how to interleave the elements in the block into internal memory. This case is illustrated in Figure 7. The analysis is the same as in Case 1, except we lose the $B!$ term because we already knew how to order the block before the transfer.

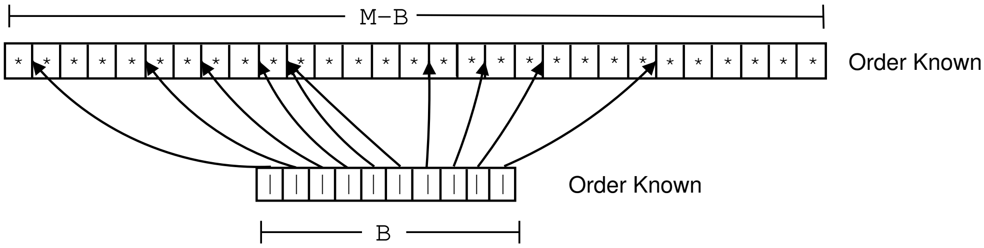

Figure 7: Case 2: The order of elements in B already known. After the memory transfer we only learn how to interleave the elements in the block into internal memory.

Therefore, after the transfer, our lower bound on the size of $|S_i|$ changes according to the formula

$$|S_i| \geq \frac{|S_{i-1}|}{\binom{M}{B}}. \tag{6}$$

We claim that Case 1 can occur at most N/B times, because there are at most N/B blocks of data to read in. After we have read N/B different blocks, every element has been in internal memory at least once. Combining the two cases, we have the following lemma:

Lemma 6 *After t input operations, at least $N!/\left(\binom{M}{B}^t (B!)^{N/B}\right)$ orderings are consistent with the available information.*

Proof $|S_0| = N!$ orderings are possible initially. In the worst case, of our t inputs, N/B of the blocks correspond to Case 1 and $t - N/B$ blocks correspond to Case 2. After reading in the blocks for Case 1, we have

$$|S_{N/B}| \geq \frac{|S_0|}{\left(\binom{N}{B} B!\right)^{N/B}}.$$

After reading in the rest of the blocks, we get

$$|S_t| \geq \frac{|S_{N/B}|}{\binom{N}{B}^{t-N/B}}.$$

Combining the two cases, we get the desired result,

$$|S_t| \geq \frac{N!}{\binom{N}{B}^t (B!)^{N/B}}.$$

□

Recall that our goal is to narrow down the number of possible orderings to 1. To get a lower bound, we need to find the smallest value of t that makes it possible for us to have $|S_t| = 1$. From Lemma 6, we see that for this to be the case, t must satisfy

$$\frac{N!}{\binom{N}{B}^t (B!)^{N/B}} \leq 1. \tag{7}$$

To work with (7), we use the following useful formulae:

- *Stirling's Approximation*: $n! \approx \sqrt{2n\pi}\left(\frac{n}{e}\right)^n$
- $\log(x!) = \Theta(x \log x)$
- $\log \binom{M}{B} = \Theta(B \log \frac{M}{B})$

Rearranging the inequality and taking log of both sides, we get

$$\frac{N!}{\binom{M}{B}^t (B!)^{N/B}} \leq 1$$

$$\binom{M}{B}^t (B!)^{N/B} \geq N!$$

$$t \log \binom{M}{B} + \frac{N}{B} \log (B!) \geq \log (N!).$$

Applying our formula for $\log \binom{M}{B}$, we get

$$tB \log \binom{M}{B} + \frac{N}{B} \log (B!) \geq \Omega (\log(N!)) .$$

Using Stirling's approximation, and simplifying, we have

$$\begin{aligned} tB \log \binom{M}{B} + \frac{N}{B} B \log(B) &\geq \Omega (N \log N) \\ tB \log \binom{M}{B} &\geq \Omega \left(N \log \frac{N}{B} \right) . \end{aligned}$$

Finally, we reduce this expression to the desired bound,

$$\begin{aligned} t &\geq \Omega \left(\frac{N}{B} \frac{\log \frac{N}{B}}{\log \frac{M}{B}} \right) \\ t &= \Omega \left(\frac{N}{B} \log_{M/B} \left(\frac{N}{B} \right) \right) . \end{aligned}$$

This proves the lower bound for t, the number of memory transfers required to sort N elements in external memory. □

In the I/O literature, results are typically expressed in terms of the variables $m = M/B$, and $n = N/B$. We can rewrite our lower bound in this standard notation,

$$t = \Omega (n \log_m n) . \tag{8}$$

4 Lower Bound for External Permutations

In this section, we prove the lower bound stated in Theorem 2. The memory model we use is similar to the one for the sorting problem, and requires the following assumptions:

- External memory is comprised of N blocks of size B for a total memory size of NB. An I/O moves a single block.
- I/O's are simple. A transfer of elements is the only allowed operation. This means the lower bound we prove may not hold if more complicated operations are possible. This model disallows an I/O that might introduce new elements or duplicates existing ones. This assumption is justified because we can ignore any non-simple I/O's that are not needed in finding the permutation.

Under these assumptions, we can view main memory and disk as one big extended array, as illustrated in Figure 8. We can essentially ignore all the other memory blocks because they do not contain any of the elements we are trying to permute.

A ***permutation*** is defined as the order of elements in the extended array, ignoring any empty spaces. This setup implies that there is exactly one permutation at all times during the computation. We prove the lower bound from Theorem 2:

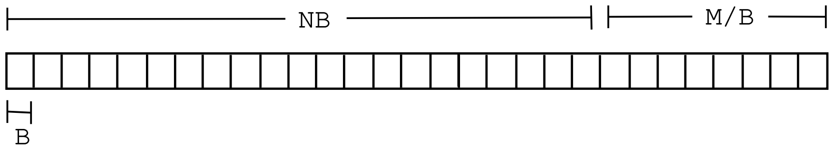

Figure 8: Memory model for external permutation problem.

Theorem 7 *Rearranging N elements according to a given permutation requires $\Omega\Big(\min\Big(N, N/B \log_{M/B} N/B\Big)\Big)$ I/O operations.*

Proof

We can do a permutation either by placing each element directly in its correct destination, or by sorting the list. We analyze the number of memory transfers required for each approach.

- One method for doing a permutation, as illustrated in Figure 9, is to move each element from the input array directly to its correct destination in the output array. The input array is contiguous, so reading the elements in order requires only N/B memory transfers. However, a permutation can send elements to an arbitrary locations in the output array, so in the worst case we require a memory transfer for every element. Therefore, the cost of this method is $\Omega(N)$ I/O operations.

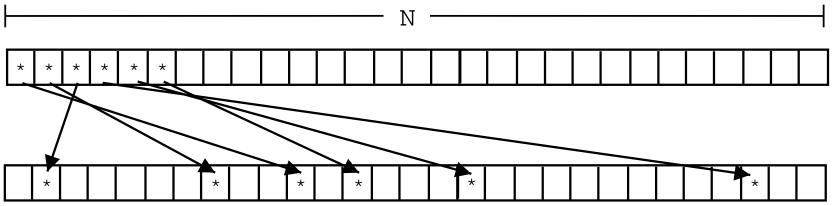

Figure 9: One method for doing a permutation is to scan the input array and move each element directly to the correct destination.

- The other method for doing a permutation is to sort the elements of the array. The order in our sort is the ordering imposed by the desired output permutation. To prove the lower bound on the number of I/O's required by this method, we use an information-theoretic argument similar to the one presented in Section 3.

  Instead of starting with $N!$ possible orders and eliminating those that are inconsistent with what we learn from a memory transfer, we start with one permutation and count how many different permutations we can realize after each memory transfer. We can solve an arbitrary permutation problem when we can realize $N!$ different permutations.

  Let V_i be the number of different permutations we can realize after i memory transfers. An upper bound on V_i represents the maximum number of different permutations we can realize.

  With no transfers, we only have the initial ordering, so $V_0 = 1$. After each memory transfer, we determine how our upper bound on V_i changes. As before, there are two types of memory transfers:

  - *Case 1: New block*: If the block has never been read into memory, then we can generate new permutations by rearranging the order of the elements in the block. Therefore, we can increase the number of realizable permutations by a factor of $B!$.

We can also generate new permutations by interleaving the elements from the block into different places in internal memory. This increases the number of realizable permutations by a factor of $\binom{M}{B}$.

We obtained the same factor of $B!\binom{M}{B}$ in Case 1 of the proof of the sorting lower bound. However, when doing a permutation, we can also choose which of the N blocks from the extended array we transfer. This increases the number of realizable permutations by an extra factor of N.

Recall that

$$V_i = \# \text{ of realizable permutations after } i \text{ memory transfers}$$

Combining all these factors, we have the relationship

$$V_i = N(B!)\binom{M}{B}V_{i-1}. \tag{9}$$

– *Case 2: Already-read block*: Since the block has been read into memory at least once, we can not generate new permutations by reordering the block. This case is similar to Case 2 in the proof of the sorting lower bound. In this case, we do not have a factor of $B!$, so

$$V_i = N\binom{M}{B}V_{i-1}. \tag{10}$$

Once again, the claim is that Case 1 happens at most N/B times. By the same logic as in Lemma 6, after t memory transfers, we know

$$V_t \leq (B!)^{N/B}\left(N\binom{M}{B}\right)^t. \tag{11}$$

We want V_t, the number of different permutations that we can realize after t memory transfers, to be at least $N!$. Therefore, we want the smallest t such that

$$(B!)^{N/B}\left[\binom{M}{B}N\right]^t \geq N!. \tag{12}$$

Taking the log of both sides and using Stirling's approximation, we get

$$\begin{aligned}
\frac{N}{B}\log(B!) + t\left(\log\binom{M}{B} + \log N\right) &\geq \log(N!) \\
N\log B + t\left(B\log\left(\frac{M}{B}\right) + \log N\right) &\geq \Omega(N\log N) \\
t\left(B\log\left(\frac{M}{B}\right) + \log N\right) &\geq \Omega\left(N\log\left(\frac{N}{B}\right)\right).
\end{aligned}$$

This expression can be rewritten as

$$t \geq \Omega\left[\frac{N\log(N/B)}{B\log(M/B) + \log N}\right].$$

For this value of t, there are two cases to consider. If $\log N \leq B\log(M/B)$, then we get the bound

$$t \geq \Omega\left(\frac{N}{B}\log_{M/B}\left(\frac{N}{B}\right)\right). \tag{13}$$

If $\log N > B \log\left(\frac{M}{B}\right)$, then this implies $B \ll \sqrt{N}$. Then, we have

$$\begin{aligned}
\frac{N\log(N/B)}{2\log N} &= \frac{N\log N - N\log B}{2\log N} \\
&= \Omega\left(\frac{1}{2}\left[N - N\frac{\log B}{\log N}\right]\right) \\
&= \Omega\left(\frac{1}{2}\left(N - \frac{N}{2}\right)\right) \\
&= \Omega\left(N\right).
\end{aligned}$$

Taking the minimum of the lower bounds for the two different methods, we prove that the number of I/O's require to do a permutation is

$$\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right\}\right) \tag{14}$$

□

5 Cache-Oblivious Sorting

Although the parallel merge sort presented in Section 2.2 achieves the optimal number of memory transfers, it requires the user to know the value of B. In the case of $P = 1$, it is actually possible to achieve the optimal number of memory transfers from Theorem 1 without knowing the block size B. In this section, we briefly describe funnelsort, a cache-oblivious mergesort presented in [2].

The difficult part of doing an I/O-efficient mergesort is determining how many elements to merge during the recursive step. As we saw in Section 2.1, a merge that only splits the problem in half does not achieve the optimal bound. In general, if we split the problem into too few sub-problems, we do not get the optimal number of memory transfers because we have to recurse too many times before our base case fits completely into cache. On the other hand, if we split the problem into too many sub-problems, then when doing a k-way merge, k may be larger than the number of blocks that can fit in cache. In this case, we do not use the cache efficiently when we merge. To do a cache-oblivious sort, we require a k-way merge, with k chosen so that it is not too large or too small.

In the funnelsort algorithm, illustrated in Figure 10, we take an input array of size N, split the array into $N^{2/3}$ sub-arrays of length $N^{1/3}$, sort the sub-arrays, and then merge the data to obtain a N element sorted list.

This method requires a ***funnel***, a black box algorithm that merges. See Figure 11. The funnel presented in [2] requires $O\left(\frac{k^3}{B}\log_{M/B}\frac{k^3}{B} + k\right)$ memory transfers and $O(k^2)$ space.

The algorithm also makes a tall-cache assumption, that $M = \Omega(B^2)$. This assumption only requires that the number of blocks that can fit into cache, M/B, be larger than the block size B.

References

[1] Alok Aggarwal and Jeffery Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

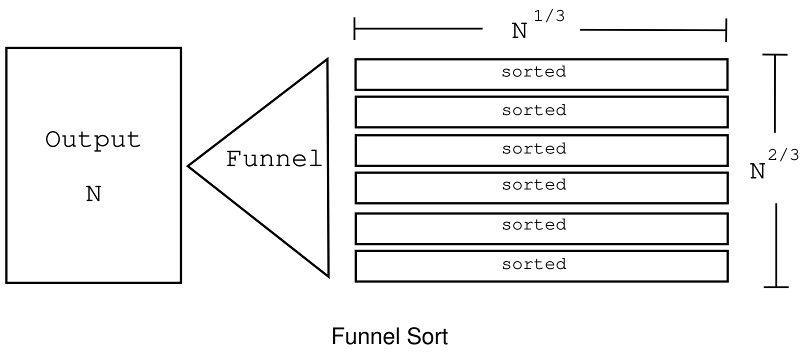

Figure 10: Illustration of funnel sort.

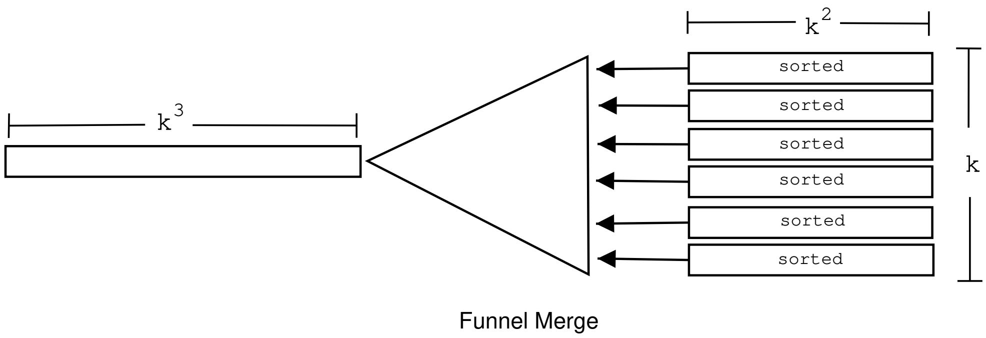

Figure 11: A funnel that takes k sorted input arrays, each of length k^2, and outputs a sorted list of length k^3.

[2] Matteo Frigo, Charles E. Leiseroson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS'99*, October 1999.