k-way merging and k-ary sorts

William A. Greene
Computer Science Department
University of New Orleans
New Orleans, LA 70148
wagcs@uno.edu

Abstract: We present a divide-and-conquer algorithm for merging k sorted lists, namely, recursively merge the first $\lfloor k/2 \rfloor$ lists, do likewise for the last $\lceil k/2 \rceil$ lists, then merge the two results. We get a tight bound for the expense, in comparisons made between list elements, of this merge. We show the algorithm is cheapest among all similar divide-and-conquer approaches to k-way merging. We compute the expense of the k-ary sort, which, in analogy to the binary sort, divides its input list into k sublists. Sometimes the k-ary sort has the same expense as the binary sort. Finally we briefly consider parallelizing these algorithms.

Introduction

Knuth tells us [10, p. 161] that the merge sort -- here we shall call it the binary sort -- was one of the first sorting algorithms suggested for computer implementation. There has been continuing interest in speeding up the merge operation (for instance, Sprugnoli [12], Carlsson [3], Thanh et al. [13], Dudzinski and Dydek [5], Brown and Tarjan [2], Trabb Pardo [14], Hwang and Lin [9]) and also interest in the speed-ups possible through parallelism (Cole [4], Shiloach and Vishkin [11], Hirschberg [8], Gavril [7], Even [6]). In all the references listed, the authors have considered the case of merging two sorted lists. In this paper we shall study the merging of $k \geq 2$ sorted lists. The topic of k-way merging has been considered before, but only lightly and in the context of external sorting ([1], [10]).

We present a simple divide-and-conquer algorithm for k-way merging. The algorithm resembles the merge sort itself: first recursively merge the first $\lfloor k/2 \rfloor$ lists, then do likewise for the last $\lceil k/2 \rceil$ lists, and finally merge the two results. We obtain a good tight bound on the number of comparisons between list elements made by our divide-and-conquer k-way merge algorithm. We show that the algorithm does the fewest comparisons among all similar divide-and-conquer approaches to k-way merging. We compute the cost (in comparisons) of the k-ary sort, which generalizes the binary sort by dividing its input list into (not 2 but) k approximately equal-sized sublists; sometimes the cost of the k-ary sort is identical to that of the binary sort. Finally we briefly consider parallelizing our algorithms.

In this paper we will always sort lists into ascending (versus descending) order. Lists are assumed to be sequentially implemented (versus a linked implementation). The floor $\lfloor x \rfloor$ and ceiling $\lceil x \rceil$ functions have their usual meanings: for a real number x,

$\lfloor x \rfloor$ = greatest integer i such that $i \leq x$,

$\lceil x \rceil$ = least integer i such that $i \geq x$.

"log" will mean base two logarithm $\log_2$; a logarithm to some other base b will be explicitly subscripted $\log_b$. When measuring the run-times of algorithms we will consider worst-case run-times.

The binary sort achieves its good runtime by a

divide-and-conquer strategy, namely, that of halving the list being sorted: the front and back halves of the list are recursively sorted separately, then the two results are merged into the answer list. An implementation is

```
procedure BINARY_SORT(L: in/out List_type);
    local  L1, L2: List_type;
    begin
        if  length(L) > 1  then
            L1 := L(1 .. ⌊n/2⌋);
            L2 := L(⌊n/2⌋ + 1 .. n);
            BINARY_SORT(L1);
            BINARY_SORT(L2);
            MERGE(L1, L2, L);
        end if;
end;
```

The *procedure* MERGE(L1, L2: *in* List_type; L: *out* List_type) that we have in mind for sorting two lists is described as follows. Initialize pointers to the first item in each list L1, L2, and then

```
repeat
    compare the two items pointed at;
    move the smaller into L;
    advance the corresponding pointer to the
            smaller's neighbor;
until one of L1,L2 exhausts;
drain the remainder of the unexhausted
            list into L;
```

The basic operation (BO) we shall be counting to obtain the (worst-case) cost of algorithms will be that of comparing two list elements. If the two input lists to MERGE have lengths m and n respectively, then MERGE does at most $m + n - 1$ BO's (the draining does comparisons, but not of list elements). (MERGE does at least $min\{m, n\}$ BO's.) If $|m - n| \le 1$ then algorithm MERGE does as few BO's as any algorithm for merging two sorted lists [Knuth, Theorem 5.3.2–M].

If $f(n)$ = the (worst-case) number of BO's done by BINARY_SORT in sorting a list of length n, then f satisfies

(1) $f(1) = 0$

(2) $f(n) = n - 1 + f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil)$

When n is a power of 2, the second equation becomes

(2′) $f(n) = n - 1 + 2\,f(n/2)$

which has solution

$$f(n) = n \log_2 n - n + 1 \qquad , n \text{ a power of 2.}$$

The general solution is

$$f(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

for arbitrary integers $n \ge 1$.

[1, Ex. 9.12]. In particular, binary sort's runtime is $O(n \log n)$.

k-way merging

Now let us generalize to an integer $k \ge 2$. Let L be a list of n elements. Divide L into k disjoint contiguous sublists $L_1, L_2, ..., L_k$ of nearly equal length. Some L_i's (namely, n rem k of them, so possibly none) will have length $\lfloor n/k \rfloor + 1$ -- for reasons that will become clear later, let these have the low indices: $L_1, L_2, \ldots$. Other L_i's will have length $\lfloor n/k \rfloor$, and are to have high indices: $..., L_{k-1}, L_k$.

We intend to recursively sort the L_i's and then merge the k results into an answer list. The expense of our k-ary sort is completely determined by the cost of merging k sorted lists. Here are three alternative algorithms for merging k sorted lists. Note below that we do not assume the source lists have approximately equal lengths.

(1) Linear-Search-Merge: Find the smallest of k items (one from each of the k sorted source lists), at a cost of k–1 BO's. Move the smallest into the answer list and replace it by its neighbor (the next largest element) in the source list from which it came. Again there are k items, from among which the smallest is to be selected. (When a list exhausts, the last moved item has no replacement, so next we find the smallest of fewer than k items.)

(2) Heap-Merge: k items (one from each sorted source list) are maintained in a heap (under discipline: root = smallest). Move the smallest item

into the answer list, replace the moved item by its neighbor in the source list from which it came, and then, with cost $2\lfloor \log k \rfloor$ BO's, re-heapify. (When a list exhausts, the last moved item is not replaced, and we re-heapify to a heap with fewer than k items.)

(3) Divide-and-Conquer-Merge: recursively merge the first $\lfloor k/2 \rfloor$ lists, recursively merge the last $\lceil k/2 \rceil$ lists, then MERGE the two results. (If k = 2 then just MERGE; if k = 1 then output = input.)

We shall show that Divide-and-Conquer-Merge performs the fewest BO's among these three alternatives for doing k-way merging.

The problem of k-way merging has been studied before, in the context of external sorting. See [1, Chapter 11, especially pages 354-355]; also see [10, especially section 5.4.1, pages 251-253]. In the cited references the authors in passing generally assume that Heap-Merge is used to merge k lists (we might call them short-ish) stored in central memory. But this is a minor interest to the authors, for they are mostly concerned with those problems peculiar to extenal sorting, namely, minimizing accesses of external memory such as tapes, which amounts to the judicious building and arranging of "runs" (sets of adjacent records that are in sorted order) on k very long tapes.

For external sorting, Heap-Merge is the sensible choice and Divide-and-Conquer-Merge is not. Heap-Merge makes one sequential pass through each of its k source lists; for external sorting this is appropriate. The recursive algorithm Divide-and-Conquer-Merge revisits its input records; for external sorting this has the undesirable effect of increasing the number of accesses of external memory (and unless tapes can be read backwards, also the number of tape rewinds will increase).

Notation: Let n be the sum of the lengths of the k source lists. Also, *D&C-Merge* abbreviates the name Divide-and-Conquer-Merge.

When the k source lists all exhaust at nearly the same time, Linear-Search-Merge performs slightly fewer than (k−1)n BO's. (The exact worst-case number of BO's made by Linear-Search-Merge is (k−1) (n − k/2), and it is a pleasant induction argument on k to show this.)

When the k source lists all exhaust at nearly the same time, Heap-Merge performs approximately $2n\lfloor \log k \rfloor$ BO's.

Now we shall show that, if the k lists are presented in decreasing order of length, then D&C-Merge performs at most

$$n\lceil \log k \rceil - (n/k)2^{\lceil \log k \rceil} + n - k + 1$$

BO's, which is always ≤ $\lceil \log k \rceil$ BO's, so about half as many BO's as Heap-Merge. In fairness, the actual runtimes of D&C-Merge can be expected to approximate and perhaps exceed the runtimes of Heap-Merge. Both have other expenses besides BO's and in particular D&C-Merge has recursion expenses, though these can be reduced by using a stack variable to simulate recursive procedure calls. We shall re-compare Heap-Merge and Divide-and-Conquer-Merge in the section on parallelism.

Now to bound D&C-Merge's expense. First we need a lemma. Below, function “len” is the length function.

Lemma: Let $L_1, L_2, ..., L_k$ be lists, where k is odd, and suppose

$$\text{len}(L_1) \geq \text{len}(L_2) \geq ... \geq \text{len}(L_k).$$

Denote $j = \lfloor k/2 \rfloor$ (which here is (k−1)/2),

$$A = \text{len}(L_1) + \text{len}(L_2) + ... + \text{len}(L_j),$$

$$B = \text{len}(L_{j+1}) + \text{len}(L_{j+2}) + ... + \text{len}(L_k),$$

$$n = A + B.$$

Then

(1) $B - A \leq n/k$,

(2) B–A = n/k if and only if all the lists have the same length,

(3) A/(k–1) + B/(k+1) ≥ n/k, with equality holding if and only if all the lists have the same length.

Proof:

$$\begin{aligned} B-A &= \text{len}(L_k) - [\text{len}(L_1) - \text{len}(L_{k-1})] \\ &\quad - [\text{len}(L_2) - \text{len}(L_{k-2})] - \ldots \\ &\quad - [\text{len}(L_j) - \text{len}(L_{j+1})] \\ &\le \text{len}(L_k) \\ &\quad \text{-- since } \text{len}(L_i) - \text{len}(L_{k-i}) \ge 0, \text{ for } i \le j \\ &\le n/k \\ &\quad \text{--since the shortest list has length} \le \text{average length.} \end{aligned}$$

This gives (1), and implies (2). Part (3) follows from (1) and (2).

Theorem 1: Let $L_1, L_2, \ldots, L_k$ be sorted lists that satisfy

$$\text{len}(L_1) \ge \text{len}(L_2) \ge \ldots \ge \text{len}(L_k).$$

Let n be the sum of their lengths. Then Divide-and-Conquer-Merge performs at most

(1) $$n\lceil \log k \rceil - (n/k)2^{\lceil \log k \rceil} + n - k + 1$$

BO's in merging these lists.

Proof: The proof is by induction on k. When k = 2, formula (1) becomes n–1, which is correct for the (maximum) number of BO's performed by MERGE when it merges two lists whose lengths sum to n. Now assume the desired bound holds whenever h < k and D&C-Merge merges h lists (whose lengths descend). Denoting $j = \lfloor k/2 \rfloor$, we note that list set $L_1, L_2, \ldots, L_j$, and list set $L_{j+1}, L_{j+2}, \ldots, L_k$ are also in descending order of length. Let

A = the sum of the lengths of the first $\lfloor k/2 \rfloor$ lists,

B = the sum of the lengths of the last $\lceil k/2 \rceil$ lists.

There will be two cases, one of which has two subcases.

Case 1: k is even. By induction the number of BO's is at most the sum

$$A\left\lceil \log\frac{k}{2} \right\rceil - \frac{A}{k/2}2^{\lceil \log(k/2) \rceil} + A - \frac{k}{2} + 1$$

$$+ B\left\lceil \log\frac{k}{2} \right\rceil - \frac{B}{k/2}2^{\lceil \log(k/2) \rceil} + B - \frac{k}{2} + 1$$

$$+ n - 1$$

where the first two lines are the costs of recursion and the third line is the cost of MERGE. Next using the relations

$$A + B = n,$$

$$\lceil \log(k/2) \rceil = \lceil \log k - 1 \rceil = \lceil \log k \rceil - 1,$$

our sum easily simplifies to expression (1).

Case 2: k is odd. Then $\lfloor k/2 \rfloor = (k-1)/2$, $\lceil k/2 \rceil = (k+1)/2$, and by induction the number of BO's is at most

$$A\left\lceil \log\frac{k-1}{2} \right\rceil - \frac{A}{\frac{k-1}{2}}2^{\left\lceil \log\frac{k-1}{2} \right\rceil} + A - \frac{k-1}{2} + 1$$

$$+ B\left\lceil \log\frac{k+1}{2} \right\rceil - \frac{B}{\frac{k+1}{2}}2^{\left\lceil \log\frac{k+1}{2} \right\rceil} + B - \frac{k+1}{2} + 1$$

$$+ n - 1$$

which simplifies to expression *E* =

$$A\lceil \log(k-1) \rceil + B\lceil \log(k+1) \rceil$$

$$- \frac{A}{k-1}2^{\lceil \log(k-1) \rceil} - \frac{B}{k+1}2^{\lceil \log(k+1) \rceil}$$

$$+ n - k + 1$$

Subcase 2.1: the odd number k is *not* of the form $1 + 2^p$ (for some positive integer p). Then

$$\lceil \log(k-1) \rceil = \lceil \log k \rceil = \lceil \log(k+1) \rceil$$

so formula *E* simplifies to

$$n\lceil \log k \rceil + n - k + 1$$

$$-[A/(k-1) + B/(k+1)]2^{\lceil \log k \rceil}$$

which is less than or equal to formula (1) of the Theorem's statement if and only if

$$A/(k-1) + B/(k+1) \ge n/k$$

The latter holds by the lemma.

Subcase 2.2: the odd number k *is* of the form $1 + 2^p$. Then

$$\lceil \log(k+1) \rceil = \lceil \log k \rceil = 1 + \lceil \log(k-1) \rceil$$

$$2^{\lceil \log(k-1) \rceil} = k - 1$$

$$2^{\lceil \log k \rceil} = 2^{\lceil \log(k+1) \rceil} = 2(k-1)$$

so *E* becomes

$$A(\lceil \log k \rceil - 1) + B\lceil \log k \rceil - A - \frac{B}{k+1} 2^{\lceil \log k \rceil} + n - k + 1$$

which simplifies to

$$n\lceil \log k \rceil + n - k + 1 - 2A - \frac{B}{k+1} 2(k-1)$$

which is less than or equal to formula (1) if and only if

$$2A + \frac{B}{k+1} 2(k-1) \geq \frac{n}{k} 2^{\lceil \log k \rceil} = \frac{n}{k} 2(k-1)$$

that is, if and only if

$$\frac{A}{k-1} + \frac{B}{k+1} \geq \frac{n}{k}$$

which once again the lemma tells us holds.

Notes:

(1) Examining the above proof and the lemma, it follows that if the k lists all have the same length, then the worst-case number of BO's performed by Divide-and-Conquer-Merge exactly equals

$$n\lceil \log k \rceil - (n/k) 2^{\lceil \log k \rceil} + n - k + 1$$

(This expression is an integer, since n is a multiple of k.) Thus our bound is tight, in the sense that it is achieved, for infinitely many n (namely, all the multiples of k). Of course, the previous sentence is true for all k.

(2) The proof and lemma also show that if the k lists do not all have the same length, then the theorem's bound is strictly greater than the actual worst-case number of BO's that get performed.

When the lists are of rather disparate lengths, the actual worst-case number of BO's performed can be considerably less than the theorem's bound. An extreme example is illustrative. Let k = 3, so that the theorem's bound is (5/3)n – 2. If three lists have respective lengths n–2, 1, 1 then D&C-Merge groups them as indicated by the parenthesization (n–2, (1, 1)) and so will perform at most 0 + 1 + (n–1) = n BO's in merging them, not (5/3) n – 2, so the theorem's bound is about 66% too big, for these three lengths.

The preceding paragraph should not cause discouragement about the theorem's bound. The theorem is to be thought of as quantified over all sets of k lists whose lengths sum to n. There are sets whose lengths are nearly equal (to n/k) and for such sets the theorem's bound is quite near the actual worst-case number of BO's performed. For example, if k = 9 and n = 9005 (which is halfway between two multiples of 9) then for the following list lengths (parenthesized to mirror how recursion groups the lists),

(((1001 1001) (1001 1001))
((1001 1000) (1000 (1000 1000))))

the worst-case number of BO's is actually 29007, whereas the theorem's bound is 29008.11. Since costs as we compute them are integers, this same example shows that floor-ing expression (1) improves the bound but does not in every case calculate exactly the worst-case number of BO's performed by D&C-Merge.

(3) If k is a power of 2 then the theorem's bound simplifies to

n log k – k + 1.

(4) If the k source lists are not initially arranged in descending order of length, then a one-time up-front cost of ≈ k log k will make them so, and the remaining cost of merging them is as stated in the theorem. Thus total cost ≈ (n + k) log k, which ≈ n log k for typical k and n.

Optimality of Halving

Our algorithm Divide-and-Conquer-Merge plays the divide-and-conquer game by halving the number of lists to be merged. Intuition suggests that halving is the best way of dividing up the lists. Indeed, this is so, at least in the sense we now describe.

Call an algorithm for merging k sorted lists a *D&CM-Algorithm* if it takes the form

if k = 1 *then* output = input
elsif k = 2 *then* MERGE
else
    partition the k lists into subsets, say, j of them, 1 < j < k;
    recurse on each of the j subsets;
    recurse on the j results;
end if;

An example of partitioning k = 9 lists into j = 3 subsets is given by ((L1, L2, L3), (L4, L5, L6, L7), (L8, L9)). We do not insist that the number j of subsets is the same on every call. On the non-recursive level, what is happening is that such an algorithm is doing a sequence of MERGE's, the last of which is the MERGE of two lists L^*_1 and L^*_2, where L^*_1 (resp., L^*_2) is obtained from the merging of m (resp., k–m) of the k source lists by some D&CM-Algorithm. The next proposition shows that partitioning into halves is optimal, when merging lists which all have the same length.

Theorem 2: To merge k sorted lists which all have the same length and whose lengths sum to n, a D&CM-Algorithm must do at least

(2) $\quad n\lceil \log k \rceil - (n/k)2^{\lceil \log k \rceil} + n - k + 1$

BO's in the worst-case.

Proof: We induct on k. When k = 2, formula (2) gives MERGE's familiar bound. Now assume the desired result holds whenever 0 < m < k and m lists all of the same length are merged by a D&CM-Algorithm. From the paragraph preceding the statement of this theorem (and noting that the common list length is n/k), what we must show is that the cost of merging m lists, then another k–m lists, followed by a trailing MERGE, that is, cost

$$\frac{mn}{k}\lceil \log m \rceil - \frac{\left(\frac{mn}{k}\right)}{m}2^{\lceil \log m \rceil} + \frac{mn}{k} - m + 1$$

$$+ \frac{(k-m)n}{k}\lceil \log (k-m) \rceil - \frac{\left(\frac{(k-m)n}{k}\right)}{(k-m)}2^{\lceil \log (k-m) \rceil}$$

$$+ \frac{(k-m)n}{k} - (k-m) + 1$$

$$+ n - 1$$

is greater than or equal to formula (2). After simplification, what we must show is that, for any m in the set {1, 2, 3, ..., k–1},

$$m\lceil \log m \rceil - 2^{\lceil \log m \rceil} + (k-m)\lceil \log (k-m) \rceil - 2^{\lceil \log (k-m) \rceil} \geq k\lceil \log k \rceil - 2^{\lceil \log k \rceil} - k$$

By symmetry, it suffices to demonstrate this for any $m \in \{1, 2, ..., \lfloor k/2 \rfloor\}$. We reason as follows.

For real numbers $x \geq 1$, define f(x) = $x\lceil \log x \rceil - 2^{\lceil \log x \rceil}$. Function f consists of linear pieces; for instance,

on interval $(2^{p-1}, 2^p]$, f(x) = $p\,x - 2^p$,

on interval $(2^p, 2^{p+1}]$, f(x) = $(p+1)\,x - 2^{p+1}$.

Moreover, since $p2^p - 2^p$ equals the limit, as x approaches 2^p from the right, of $(p+1)\,x - 2^{p+1}$, we also conclude function f is continuous (in the mathematical sense). Thus the graph of f can be described as: a straight line segment of slope 1, connected to a straight line segment of slope 2, connected to a straight line segment of slope 3, connected to ... and so on. Consequently, the difference between the values of f at two consecutive integers, f(m) – f(m–1), can be calculated as soon as we know which interval $(2^{q-1}, 2^q]$ contains m, for then the difference f(m) – f(m–1) must equal the slope, which is q.

Recall the integer k of this proposition. Let p be

the integer that satisfies $2^p < k \le 2^{p+1}$. For integers $m \in \{1, 2, ..., \lfloor k/2 \rfloor\}$ define $g(m) = f(m) + f(k-m)$ and note that for such m,

$m \le k/2 \le 2^p$ implies: $f(m) - f(m-1) \le p$,

$k-m \ge k/2 > 2^{p-1}$ implies: $f(k-m+1)-f(k-m) \ge p$.

Then $g(m-1) - g(m)$

$= f(m-1) + f(k-m+1) - f(m) - f(k-m)$

$= (f(k-m+1) - f(k-m)) - (f(m) - f(m-1))$

$\ge p - p = 0.$

That is, g is a *decreasing* function of its domain {1, 2, ..., $\lfloor k/2 \rfloor$}. It is straightforward to verify that g's least value $g(\lfloor k/2 \rfloor)$ equals

$$k\lceil \log k \rceil - 2^{\lceil \log k \rceil} - k$$

(again there are the three cases: k even, k odd *and of* form $1 + 2^p$, k odd *but not* of form $1 + 2^p$). We have shown

$f(m) + f(k-m) = g(m) \ge k\lceil \log k \rceil - 2^{\lceil \log k \rceil} - k$

for all $m \in \{1, 2, 3, ..., \lfloor k/2 \rfloor\}$. This was precisely our goal.

Note: Nature is capable of remarkable economies! In the notation of the proposition, $k/2 \in (2^{p-1}, 2^p]$, and m and k–m lie on either side of k/2. If m, k–m both fall into interval $(2^{p-1}, 2^p]$ then it is easily shown that $g(m) = f(m) + f(k-m)$ will equal g's least value $g(\lfloor k/2 \rfloor)$. By the continuity of f, the same can be said even if m is the stranded endpoint 2^{p-1}.

We might express these matters by saying that halving is optimal but other partitions can achieve equally good results. For instance: recall the algorithm D&C-Merge = Divide-and-Conquer-Merge (the halver). Let k = 24 (= 3 times a power of 2, so, halfway between two powers of 2). If 24 lists (of equal length) are partitioned into two subsets, the sizes of the subsets are a pair of numbers that sum to 24. Five such pairs are

(8,16), (9,15), (10,14), (11,13), (12,12).

(D&C-Merge -- the halver -- would use the last pair.) For any one of these five pairs (m, k–m), imagine: invoking D&C-Merge to merge the subset of m lists, invoking D&C-Merge to merge the subset of k–m lists, then MERGE-ing the two results. The (worst-case) number of BO's so performed must exactly equal the number performed when D&C-Merge is called to merge 24 lists. That equality must hold follows from this note's first paragraph and from examining the proposition's proof.

The k-ary sort

Now let us return to the k-ary sort, which divides its unsorted input list into k sublists of nearly equal length and makes k recursive calls, followed by a call of Divide-and-Conquer-Merge. If f(n) = worst-case number of BO's performed by the k-ary sort when sorting a list of length n, then f satisfies

(1) $f(1) = 0$

(2) $f(n) \le n\lceil \log k \rceil - (n/k)2^{\lceil \log k \rceil} + n - k + 1$

$+ (n \text{ rem } k)\, f(\lceil n/k \rceil) + (k - (n \text{ rem } k))\, f(\lfloor n/k \rfloor)$

When n is a power of k, inequality (2) is replaced by equality

(2′) $f(n) =$

$$n\lceil \log k \rceil - (n/k)2^{\lceil \log k \rceil} + n - k + 1 + kf(n/k)$$

which has solution

$f(n) = n\log_k n\lceil \log k \rceil - (n/k)\log_k n\, 2^{\lceil \log k \rceil}$

$+ n\log_k n - n + 1$

as an induction argument (on n = powers of k) will show.

If n is a power of k and k is a power of 2 we get

$f(n) = (n\log_k n)\log_2 k - n + 1$

$= n\log_2 n - n + 1.$

Thus, for example, octary sort (k = 8) performs exactly the same number of BO's as binary sort when sorting lists of length 2^{3m}. On reflection, this is not altogether surprising.

In general, the k-ary sort has runtime $O(n\log n)$.

Parallelism

Heap-Merge does not improve in the presence of parallelism (that is, a multiplicity of processing units operating simultaneously). The recurring expense in Heap-Merge is re-heapifying, and re-heapifying is inherently sequential; it cannot be parallelized.

On the other hand, Divide-and-Conquer-Merge and the k-ary sort can easily be parallelized and thereby sped up, which we now briefly investigate. Using more elaborate algorithms, others have achieved faster runtimes than we shall. The algorithm of Shiloach and Vishkin in section 4.1 of [11] has some outward similarity to our k-ary sort, but their sorter is not recursive and uses a different merging routine. Their runtime is $O((n/k) \log n)$ where k = the number of available processors; our runtime will be $O((n/\log k) \log n)$. Cole's very complicated "cascading" merge [4] achieves a runtime of $O(\log n)$ if there are as many processors as there are list elements to be sorted.

So now let us consider the case that there are 16 lists. Ultimately, Divide-and-Conquer-Merge's behavior is to MERGE these by pairs, MERGE the 8 results by pairs, MERGE *those* 4 results by pairs, etc. Obviously the 8 incarnations of MERGE on the lowest level can run in parallel, and similarly for higher levels. Actually, we can do even better by starting the merging on level m just one tick after starting that on level m+1.

Suppose there are 15 processors, arranged in a full binary tree, in the sense that output from a child processor is input to its parent. The processors we have in mind are quite simple. Each compares two input records from its memory and outputs the smaller into its parent's memory; call that unit of activity a *cycle*. Each of the 8 leaf processors begins with input consisting of two sorted lists. Let n be the sum of the lengths of these 16 lists. On the fourth cycle the root outputs for the first time (outputting, of course, the smallest element among the 16 lists). On each succeeding cycle the root outputs one more element. After n+3 cycles the 16 lists will have been merged. (One can conceive of short-cuts when lists exhaust early, but the worst-case expense is n+3 cycles.) A cycle is hardly different from a BO as defined earlier. The expense n+3 on the parallel machine should be contrasted with the theorem's expense of $\approx n \log_2 16 = 4n$ on a uni-processor machine -- a four-fold speed-up.

Now suppose on our 15-processor machine we have to sort a list L of length n. We do so with a 16-ary sort:

> divide the list into sixteenths;
> make sixteen recursive calls, one for
> each sixteenth;
> merge, using the parallel merge algorithm;

Each recursive call will also perform a 16-way merge, so will occupy all 15 processors, therefore the 16 recursive calls are to be done sequentially. Let f(n) = (worst-case) number of cycles required to sort L. For n's that are powers of 16,

$f(1) = 0,$

$f(n) = n + 3 + 16\, f(n/16)$

which has solution

$f(n) = n \log_{16} n + (n-1)/5$

$= (1/4)\, n \log_2 n + (n-1)/5$

or 4 times faster than binary sort on a uni-processor machine. For a parallel machine with 2^p-1 processors, the measurements are: parallel merge completes after $n+p-1$ cycles; sorting is p times faster than on a uni-processor machine.

If there are as many processors as there are list elements to be sorted, then sorting can become merging where leaf processors start with a pair of singleton lists; then sorting completes after $n+\log n$ cycles. This scenario is overly generous in its use of processors; for instance, after one cycle the leaf-level processors (half of the total) have no more work to do and could be reallocated to elsewhere in the tree.

Summary and Conclusion

Our original interest was in the k-ary sort, which is the generalization of the binary sort to the case of dividing a source list into (not 2 but) k sublists. All the essential expense of the k-ary sort comes from the merging operation. Thus arose our curiosity about ways to do a k-way merging of k sorted lists.

A strategy we named Divide-and-Conquer-Merge was presented, a tight bound was found for its expense, and it was shown less costly than two other strategies for k-way merging (Linear-Search-Merge, Heap-Merge). Our algorithm Divide-and-Conquer-Merge, whose scheme is to recurse on halves of the numbers of source lists being merged, was additionally shown optimal among a class of similar approaches that recurse on subgroups of the source lists. The expense of the k-ary sort was analyzed to be $O(n \log n)$; sometimes its expense exactly equals that of the binary sort. We briefly explored parallel implementations of our merging and sorting approaches, and their costs.

We do not expect to see actual use of the k-ary sort, since simpler approaches such as the binary sort are no costlier. K-way merging may see application. The mathematical techniques used in our cost analyses are, to our knowledge, entirely novel and, in our opinion, intellectually stimulating and esthetically appealing. As with certain other instances we might cite in complexity analysis, the proofs are as intriguing as the statements of the theorems.

Bibliography

1. Aho, Alfred A., Hopcroft, John E., and Ullman, Jeffrey D., *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass., 1983.

2. Brown, Mark R., and Tarjan, Robert E., "A fast merging algorithm", *J. Assoc. Comput. Mach.* 26 (1979), 211-226.

3. Carlsson, Svante, "Splitmerge - a fast stable merging algorithm", *Information Proc. Lett.* 22 (1986), 189-192.

4. Cole, Richard, "Parallel merge sort", *SIAM J. Computing* 17 (1988), 770-785.

5. Dudzinski, Krzysztof, and Dydek, Andrzej, "On a stable minimum storage merging algorithm", *Information Proc. Lett.* 12 (1981), 5-8.

6. Even, Shimon, "Parallelism in tape-sorting", *Communications Assoc. Comput. Mach.* 17 (1974), 202-204.

7. Gavril, Fanica, "Merging with parallel processors", *Communications Assoc. Comput. Mach.* 18 (1975), 588-591.

8. Hirschberg, D. S., "Fast parallel sorting algorithms", *Communications Assoc. Comput. Mach.* 21 (1978), 657-661.

9. Hwang, F. K., and Lin, S., "A simple algorithm for merging two disjoint linearly ordered sets", *SIAM J. Computing* 1 (1972), 31-39.

10. Knuth, Donald E., *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass. 1973.

11. Shiloach, Yossi, and Vishkin, Uzi, "Finding the maximum, merging, and sorting in a parallel computation model", *J. Algorithms* 2 (1981), 88-102.

12. Sprugnoli, Renzo, "The analysis of a simple in-place merging algorithm", *J. Algorithms* 10 (1989), 366-380.

13. Thanh, Mai; Alagar, V. S.; and Bui, T. D., "Optimal expected time algorithms for merging", *J. Algorithms* 7 (1986), 341-357.

14. Trabb Pardo, Luis, "Stable sorting and merging with optimal space and time", *SIAM J. Computing* 6 (1977), 351-372.