International Journal of Foundations of Computer Science
© World Scientific Publishing Company

ON COST-OPTIMAL MERGE OF TWO INTRANSITIVE SORTED SEQUENCES *

JIE WU
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
E-mail: jie@cse.fau.edu

and

STEPHEN OLARIU
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
E-mail: olariu@cs.odu.edu

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The problem of merging two intransitive sorted sequences (that is, to generate a sorted total order without the transitive property) is considered. A cost-optimal parallel merging algorithm is proposed under the EREW PRAM model. This algorithm has a run time of $O(\log^2 n)$ using $O(n/\log^2 n)$ processors. The cost-optimal merge in the strong sense is still an open problem.

Keywords: Hamiltonian path, merging, PRAM, sorting, tournament.

1. Introduction

In this paper we consider the problem of merging two intransitive sorted sequences. Consider a total order $\rightarrow$ on set N, but without the transitive property. That is, if $u_i \rightarrow u_j$ and $u_j \rightarrow u_k$, it is not necessary that $u_i \rightarrow u_k$. The total order requires that for any two elements u_i and u_j, either $u_i \rightarrow u_j$ or $u_j \rightarrow u_i$.

A *intransitive sorted sequence* is a sequence of elements, $u_1 u_2 ... u_n$, in N, such that

$$u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_n.$$

It has been proved that for any subset of N, the elements in the subset can be arranged in a sorted sequence (more than one may exist). The intransitive total

*This work was supported in part by NSF grant CCR 9900646 and grant ANI 0073736.

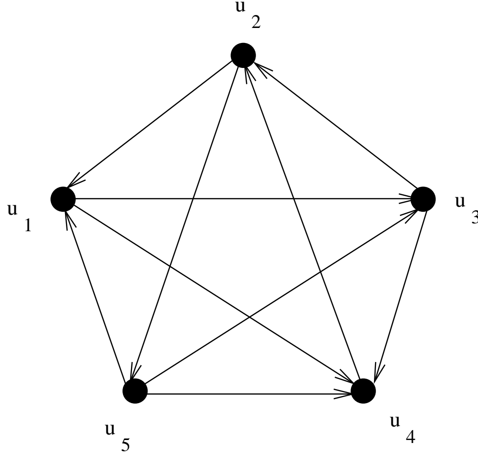

Figure 1: A tournament of five players.

order is also called a *tournament* which is a directed graph with its underlying graph completely connected. Each player in a tournament is represented by a vertex. An edge, $u_i \to u_j$, exists if player u_i beats player u_j. A sorted sequence corresponds to a Hamiltonian path of the graph. Figure 1 shows a tournament of five players. One intransitive total order is $u_3 \to u_4 \to u_2 \to u_5 \to u_1$. When $\to$ is transitive, the intransitive total order arrangement is reduced to a regular sorting problem. Unlike the regular sorting problem, more than one solution exists for the generalized sorting problem. For example, $u_1 \to u_3 \to u_2 \to u_5 \to u_4$ is another intransitive total order for the example of Figure 1.

A parallel algorithm is *cost-optimal* for a given problem if the product of the run time and the number of processors used matches the sequential complexity of the problem, regardless of the run time of the parallel algorithm. A parallel algorithm is *cost-optimal in the strong sense*, or *strongly cost-optimal*, if its run time cannot be improved by any other cost-optimal parallel algorithms.

In this paper, we consider the problem of merging, which deals with merging two given sorted subsequences into one sorted sequence. Sorting and merging are related, for example, the two-way merge strategy can be used to generate a two-way merge sort. However, sorting and merging are different with different lower bounds on computational complexity. For example, under the CREW PRAM model, the lower bound on the regular sorting is $O(\log n)$, whereas the lower bound on the regular merging is $O(\log \log n)$.

Merging two intransitive sorted subsequences poses new challenges. The traditional merging by ranking [3] and bitonic merging [2] cannot be applied. Because both concepts of ranking and bitonic sequences use the transitive property. In this paper, we propose a divide-and-conquer approach called *split-and-merge* that repeatedly splits a pair of merging sequences into two independent pairs of merging subsequences. This split-and-merge algorithm has a run time of $O(\log^2 n)$ using

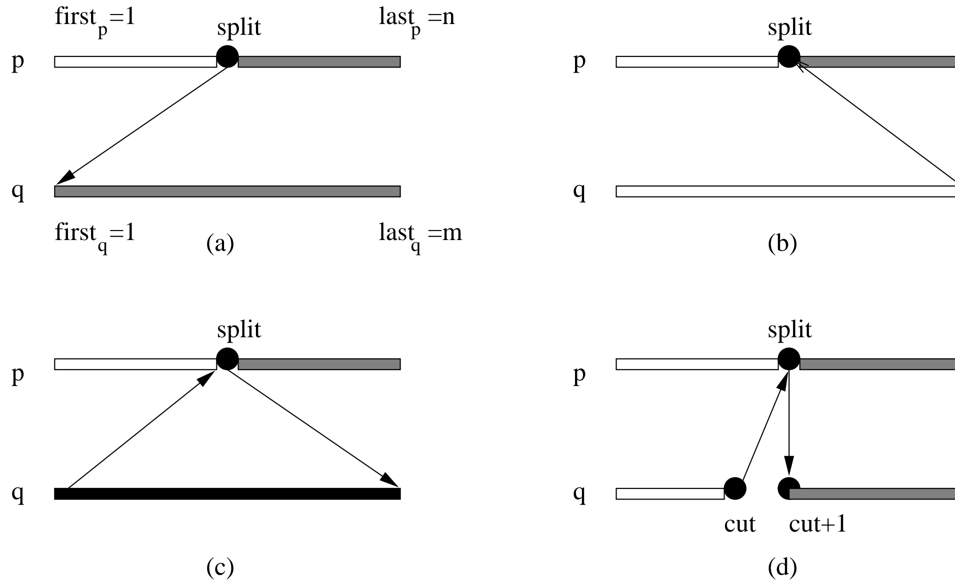

Figure 2: Three possible split-and-merge situations.

$O(n/\log^2 n)$ processors under the EREW PRAM model. Clearly, this solution is cost-optimal.

This paper is organized as follows: Section 2 proposes the cost-optmal merge process. Section 3 discusses related work and some open problems. The paper concludes in Section 4.

2. Cost-Optimal Merge

We use $p[1..n]$ and $q[1..m]$ to represent two given sorted sequences and $p[i]$ represents a specific element. We first introduce a *split-and-merge process* that splits a pair of sequences into two independent pairs of subsequences. Two pairs are merged once two subsequences in each pair have been merged. This process is done recursively by calling the split-and-merge process. Let $p[split]$ be the center of sequence p and it is called the *splitting point*. $first_p$ ($first_q$) and $last_p$ ($last_q$) denote indices for the first and last elements of sequence p (q), respectively.

The general steps are the following:

- Select the center element of p as the splitting point and it is denoted as $p[split]$.
- If $p[split]$ beats the first element of q (i.e., $p[split] \rightarrow q[first_q]$), then subsequence $p[first_p..\ split]$ (the white subsequence in Figure 2 (a)) is merged with an empty subsequence of q, followed by merging $p[split+1..last_p]$ with $q[first_q..last_q]$ (the two gray subsequences in Figure 2 (a)).
- If $p[split]$ is beaten by the last element of q (i.e., $q[last_q] \rightarrow p[split]$), then $p[first_p..split-1]$ is merged with $q[first_q..last_q]$, followed by $p[split..last_p]$

```
par_merge(p[first_p..last_p], q[first_q..last_q]):
1.     case of
2.         (first_p ≰ last_p): return q[first_q..last_q]
3.         (first_q ≰ last_q): return p[first_p..last_p]
4.     end of case
5.     split := ⌊(first_p + last_p)/2⌋
6.     if p[split] → q[first_q] then
7.         p[first_p..split] || par_merge(q[first_q..last_q], p[split + 1..last_p])
8.     else if  q[last_q] → p[split] then
9.             par_merge(q[first_q..last_q], p[first_p..split − 1]) || p[split..last_p]
10.        else cut := cut(p[split], q[first_q..last_q])
11.            par_merge(q[first_q..cut], p[first_p..split])||
12.            par_merge(q[cut + 1..last_q], p[split + 1..last_p])
cut(s, r[first..last]):
1.     cut := ⌊(first + last)/2⌋
2.     if r[cut] → s ∧ s → r[cut + 1] then
3.         return cut
4.     else if r[cut] → s then
5.             cut(s, r[cut..last])
6.         else  cut(s, r[first..cut])
```

(see Figure 2 (b)).

- If both of the above two cases fail (see Figure 2 (c)), we use a binary search on q to find a *cutting point* (denoted as cut as shown in Figure 2 (d)) in q, that is, $q[cut] \rightarrow p[split]$ and $p[split] \rightarrow q[cut+1]$. The final sequence is constructed by merging $p[first_p..split]$ with $q[first_q..cut]$ (two white subsequences in Figure 2 (d)), followed by the result of merging $p[split+1..last_p]$ with $q[cut_q+1..last_q]$ (two gray subsequences in Figure 2 (d)).

- The role of p and q is alternated in the subsequent recursive merging process to ensure the sizes of both p and q are reduced by at least half in every two consecutive rounds.

We use $||$ as a concatenation operation to combine two sequences. The merging process starts by calling **par_merge**$(p[1..n], q[1..m])$. To simplify our discussion, we assume $n = m$. The splitting process is done by locating the splitting point at p and the cutting point at q. The cutting point is located through a binary search process (**cut**).

It is not necessary that merging sequences have to be split into pieces of unit size. **par_merge** can be modified based on the following split-and-merge process, where **par_merge** terminates whenever the size of both sequences is less than or equal to a predefined value, and then, an optimal sequential merging algorithm is applied.

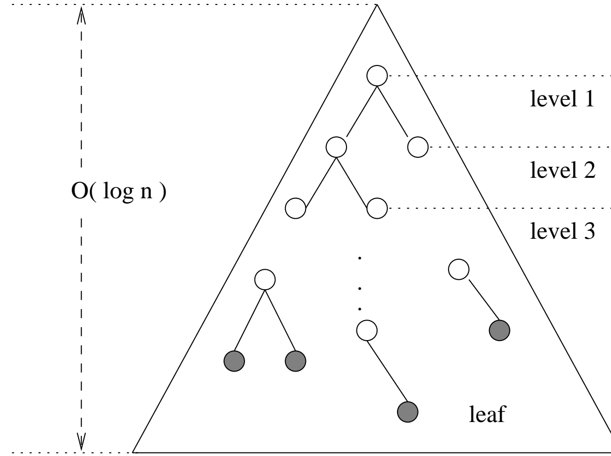

Figure 3: A sample split-and-merge tree.

---

split-and-merge:

- Use **par_merge**$(p[1..n], q[1..n])$; however, the recursive call is terminated whenever the sizes of both p and q are less than or equal to $\log^2 n$.
- When a recursive call terminates, an optimal sequential merge algorithm is applied to combine every pair of subsequences.

To support the above split-and-merge strategy in the **par_merge** algorithm. The following statement is inserted to the case statement: $[(last_p - first_p) < \log^2 n] \wedge [(last_q - first_q) < \log^2 n]$: **seq_merge**$(p[first_p..last_p], q[first_q..last_q])$, where **seq_merge** is a regular optimal sequential merge algorithm.

Figure 3 shows a sample split-and-merge tree, where each node corresponds to a split-and-merge process. Each node (process) may generate up to two nodes (two split-and-merge processes), and therefore, this is a binary tree. The depth of the tree is bounded by $O(\log n)$, since the size of each sequence (p or q) is reduced by half in every two consecutive rounds. An optimal sequential merging algorithm is applied to each leaf in the tree.

Theorem 1: *The run time for the split-and-merge algorithm is* $O(\log^2 n)$ *with* $O(n/\log^2 n)$ *processors under the EREW PRAM model.*

Proof: The split-and-merge process consists of two phases: parallel split and sequential merge. At the parallel split phase, consider a new tree by deleting all leaf nodes (gray nodes in Figure 3) of the split-and-merge tree. Since the number of elements associated with two sequences at each node is bounded by $\Omega(\log^2 n)$ in the new tree, whereas the total number of elements in two original sequences is $O(n)$, the number of leaf nodes in this new tree is bounded $O(n/\log^2 n)$, that

is, the number of parallel splits is bounded by that number. Therefore, there is a sufficient number of processors to handle concurrent splitting activities (within the same level). We only need to calculate the run time of the longest path. Note that each node in a path corresponds to a cutting process (that identifies a cut) using a binary search. The size of each sequence is reduced by at least half in two consecutive rounds. Therefore, the overall cost is bounded by

$$\lceil \log n \rceil + \lceil \log n/2 \rceil + \lceil \log n/2 \rceil + \lceil \log n/4 \rceil + \lceil \log n/4 \rceil + ...$$

which is $O(\log^2 n)$. Sequential merge is used at each leaf of the split-and-merge tree. Throughout the split-and-merge process, no concurrent read or write is needed; consequently, only the EREW PRAM model is needed.

The cost of a sequential merge at each leaf node is equal to the number of elements in the two subsequences to be merged, which is bounded by $2\log^2 n = O(\log^2 n)$. However, the number of leaf nodes could be more than $O(n/\log^2 n)$. In fact, the size of both p and q is reduced by at least half by two consecutive rounds generating up to four new branches in the split-and-merge tree. Consider a complete binary tree with a depth of $2\log(n/\log^2 n)$, the total number of leaf nodes is $2^{2\log(n/\log^2 n)} = (n/\log^2 n)^2$, which is clearly more than $n/\log^2 n$ (the number of processors). We divide leaf nodes into two groups: a leaf node with a size of $\log^2 n$ or more is assigned to *group one* and a node with a size less than $\log^2 n$ is assigned to *group two*. Each leaf node in group one is assigned to a distinct processor. Leaf nodes in group two are assigned in sequence to a processor until its load is no less than $\log^2 n$ (but less than $2\log^2 n$) and, then, a new processor is used for the assignment. Clearly, these sequential merges can be done within $2\log^2 n = O(\log^2 n)$ using no more than $2n/\log^2 n = O(n/\log^2 n)$ processors. Therefore, the overall run time is $O(\log^2 n)$ using $O(n/\log^2 n)$ processors. ■

Because the product of the run time and the number of processors is $O(n)$ which matches the lower bound for sequential computation, the proposed algorithm is cost-optimal.

3. Related Work and Open Problems

Cole [3] shows that the cost-optimal merge of regular sorted sequences in the strong sense under the EREW PRAM model is $O(\log n)$ using $O(n/\log n)$ processors. It is not clear that our solution is cost-optimal in the strong sense for merging two intransitive sorted sequences. Therefore, the cost-optimal solution in the strong sense still remains open.

Several parallel algorithms have been proposed [1, 4, 6] to determine a Hamiltonian path in a tournament. Wu [7] proposes a pipelined solution under the EREW PRAM model using a new data structure called *semi-heap*. Basically, in semi-heap, the notion of max is replaced by $\max_{\rightarrow}$ defined on the intrasitive order $\rightarrow$. Specifically, a semi-heap for a given intransitive total order $\rightarrow$ is a complete binary tree. For every node u in the tree, $u = \max_{\rightarrow}\{u, L(u), R(u)\}$, where $L(u)$ and $R(u)$ are left and right child of u, respectively. $\max_{\rightarrow}$ is defined as $u = \max_{\rightarrow}\{u, L(u), R(u)\}$

if both $L(u) = \max\{u, L(u), R(u)\}$ and $R(u) = \max\{u, L(u), R(u)\}$ are false, where max is the regular maximum function.

Wu and Sheng [8] propose the notion of the *sorted sequence of kings*. A king u in a tournament is a player who beats ($\rightarrow$) any other player v directly or indirectly; that is, either $u \rightarrow v$ or there exists a third player w such that $u \rightarrow w$ and $w \rightarrow v$. A sorted sequence of kings in a tournament of k players is a sequence of players, $u_1 u_2 ... u_n$, such that $u_i \rightarrow u_{i+1}$ and u_i in a king in sub-tournament $\{u_i, u_{i+1}, ..., u_n\}$ for $i = 1, 2, ..., n-1$. Clearly, the sorted sequence of kings adds extra constraints on the intransitive sorted sequence. It has been proved that the sorted sequence of kings exists in any tournament and an $O(n^2)$ solution based in a modified insertion sort is given in [8]. On the other hand, optimal merge of two sorted sequence of kings is still an uncharted territory.

Wu [7] also shows the intransitive sorted sequence as an approximation for ranking players in a tournament. In general, the *tournament ranking problem* [5] is a difficult one without exhibiting "fairness". Suppose $1, 2, ..., n$ is a ranking of players with 1 representing the champion and i representing the ith place winner. Without loss of generality, we assume that player u_i is ranked in the ith place. For any pair of players u_i, u_j with $i < j$, a *happiness* means that u_i beats u_j while an *upset* means that u_j beats u_i. Clearly, a good ranking should have the minimum number of total upsets. A *median order* is defined as a ranking of players with a minimum number of total upsets. However, the problem of finding a median order in a tournament is NP-complete. It is shown in [7] that any median order must be an intransitive sorted sequence.

4. Conclusion

In this paper, we have provided a cost-optimal merge of two intransitive sorted sequences, which is a special total order without the transitive property. The proposed solution is based on the EREW PRAM model with a run time of $O(\log^2 n)$ using $(n/\log^2 n)$ processors. We have also discussed other related problems including sorted sequence of kings and tournament ranking problem. Finally, we have pointed out that the cost-optimal merge in the strong sense is still an open problem.

References

1. A. Bar-Noy and J. Naor. Sorting, minimal feedback sets, and Hamilton paths in tournaments. *SIAM Journal on Discrete Mathematics*. 3, (1), Feb. 1990, 7-20.
2. K. Batcher. Sorting networks and their applications. *Proc. of the AFIPS Spring Joint Computing Conference*. 1968, 307-314.
3. R. Cole. Parallel merge sort. *SIAM Journal on Computing*. **17**, 4, 1988, 770-785.
4. P. Hell and M. Rosenfeld. The complexity of finding generalized paths in tournaments. *Journal of Algorithms*. 1983, 4, 303-309.
5. K. B. Reid and L. W. Beineke. Tournaments. Chapter 7 in: L. W. Beineke and R. Wilson, eds., *Selected Topics in Graph Theory*, Academic Press, New York, 1979.
6. D. Soroker. Fast parallel algorithms for finding Hamilton paths and cycles in a

tournament. *Journal of Algorithms.* 1988, 276-286.

7. J. Wu. On sorting an intransitive total ordered set using semi-heap. *Proc. of IEEE International Parallel and Distributed Processing Symposium.* May 2000, 257-262.
8. J. Wu and L. Sheng. An efficient sorting algorithm for a sequence of kings in a tournament. *Information Processing Letters.* 79, 6, 2001, 297-299.