PARALLEL ALGORITHMS FOR EVALUATING SEQUENCES OF SET-MANIPULATION OPERATIONS
(Preliminary Version)

Mikhail J. Atallah
Dept. of Computer Science, Purdue Univ., West Lafayette, IN 47907

Michael T. Goodrich
Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

S. Rao Kosaraju
Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Given an off-line sequence $S$ of $n$ set-manipulation operations, we investigate the parallel complexity of evaluating $S$ (i.e., finding the response to every operation in $S$ and returning the resulting set). We show that the problem of evaluating $S$ is in NC for various combinations of common set-manipulation operations. Once we establish membership in NC (or, if membership in NC is obvious), we develop techniques for improving the time and/or processor complexity.

1 Introduction

The evaluation of operation sequences is something that is fundamental in the design of algorithms. Given a sequence $S$ of set-manipulation operations the evaluation problem is to find the response to every operation in $S$ and return the set one gets after evaluating $S$. There are a host of problems that are either instances of an evaluation problem or can be solved by a reduction to an evaluation problem. For example, sorting a set $S = \{x_1, x_2, \ldots, x_n\}$ can easily be reduced to the problem of evaluating the sequence $(I(x_1), I(x_2), \ldots, I(x_n), E, E, \ldots, E)$, where $I(x)$ stands for "Insert $x$," $E$ stands for "ExtractMin," and there are $n$ $E$'s. The answers to all the "ExtractMin" operations immediately give us a sorting of the items in $S$.

Given an off-line sequence $S$ of $n$ set-manipulation operations, we investigate the parallel complexity of evaluating $S$. This is a well studied problem in sequential computation, but surprisingly little is known about its parallel complexity. Our motivation, then, comes from a desire to begin a systematic treatment of this important area from a parallel perspective. In addition, because of the foundational aspect of off-line evaluation problems, we are also interested in these problems for their possible applications. We already know of applications to such areas as processor scheduling, computational geometry, and computational graph theory (we discuss some of these below).

This paper contains two types of results:

(i) We show that the problem of evaluating $S$ is in the class NC for various combinations of operations appearing in $S$. That is, it can be evaluated in $O(\log^2 n)$ time using $O(n^d)$ processors, for constants $c$ and $d$.

(ii) Once we establish membership in NC (or, if membership in NC is obvious), we develop techniques for improving the time and/or processor complexity.

The following is a list of the results presented in this paper. We give the results in the following format: First we mention the operations that may appear in $S$, then we give a pair $T(n), P(n)$ representing (respectively) the time and processor complexities of our algorithms, to within constant factors.

1. Insert($x$), Delete($x$), ExtractMin: $\log^2 n$, $n$.
   Showing that this problem is in NC, let alone that it can be solved using a linear number of processors, is perhaps our most surprising result.

2. Insert($x$), ExtractMin($x$): $\log^2 n$, $n^3 / \log n$.
   An ExtractMin($x$) operation returns and simultaneously removes the smallest element larger than or equal to $x$.

3. Insert($z$), ExtractMin($z$), in which the ExtractMin($z$)'s have non-decreasing arguments: $\log n$, $n^3$. The technique we use here is very different from our solution to problem 2. We give an application of this problem to maximum matching in a convex bipartite graph.

4. Insert($x$), ExtractMin: $\log n$, $n$. This $T(n) * P(n)$ product is optimal.
5. Insert(x), Delete(y), and "tree-search" queries: log \( n \) \( n \). The queries include any which do not modify the set and could be performed sequentially in logarithmic time if the set were stored in a binary tree which has \( O(1) \) labels associated with each node \( v \) (which could be computed by applying associative operations to the decedents of \( v \)).

6. Insert(x, A), Delete(x, A), Min(A), Union(A, B), Find(x, A): log \( n \), n. Here \( A \) and \( B \) are set names. When we allow operations such as these, that manipulate sets as well as their elements, then the methods of result 5 no longer hold. In this case we show that the problem can still be solved optimally, however.

We briefly outline the conventions we use in this paper. The computational model we use is the CREW-PRAM model, unless otherwise specified. Recall that this is the shared-memory model of where the processors operate synchronously and can concurrently read any memory cell, but concurrent writes are not allowed. (Some of our results are for the weaker EREW PRAM, when no concurrent memory accesses are allowed.) Another convention we use is that sets are actually multisets (i.e., multiple copies of an element are allowed), so that whenever we say "element \( x \)" we are actually referring to a particular copy of \( x \). It is straightforward to modify our results for the case when having multiple copies of an element would be erroneous.

The details for each result are given in what follows, one per section. We conclude with some final remarks in Section 8.

2 Insert(x), Delete(x), ExtractMin

If \( A \) is a set and \( S \) a sequence of set manipulation operations, then \( AS \) denotes applying the sequence \( S \) to a set whose initial value is \( A \). In addition to the responses to the operations in \( S \), an evaluation of \( AS \) also returns the set "left over" after \( S \) is evaluated. In this notation, we are trying to solve \( \emptyset S \) where the operations in \( S \) come from the set \( \{ I(x), D(x), E \} \).

Let \( S_1 \) (resp., \( S_2 \)) be the sequence consisting of the first (resp., last) \( n/2 \) operations in \( S \). Recursively solve \( \emptyset S_1 \) and \( \emptyset S_2 \) in parallel. The recursive call for \( \emptyset S_1 \) returns (i) the correct responses for the operations in it (i.e., the same as in \( \emptyset S \)), and (ii) the set just after \( \emptyset S_1 \) terminates (let \( L_1 \) denote this set). The recursive call for \( \emptyset S_2 \) returns responses and a final set that may differ from the correct ones, because it applies \( S_2 \) to \( \emptyset \) rather than to \( L_1 \). The main problem that we now face is how to incorporate the effect of \( L_1 \) into the solution returned by the recursive call for \( \emptyset S_2 \), obtaining the solution to \( L_1 S_2 \).

**Notation:** If \( R \) is a subsequence of \( S \), then \( S-R \) denotes the sequence obtained by removing every element in \( R \) from \( S \).

**Convention:** Throughout, our algorithms adopt the following conventions. A \( D(x) \) executed when there are many copies of \( x \) in the set removes the copy that was inserted latest. Similarly, an \( E \) executed when there are many copies of the smallest element in the set removes the copy that was inserted latest. These conventions cause no loss of generality because they do not change any response. However, they do simplify our correctness proofs.

Let us first make some observations about \( \emptyset S_2 \). Let \( L_2 \) be the set resulting from \( \emptyset S_2 \) (i.e., the set after \( \emptyset S_2 \) terminates). Consider an \( I(x) \) for which \( x \) is not removed by any \( E \) in \( \emptyset S_2 \), i.e., it either ends up in \( L_2 \) or gets removed by a \( D(x) \) (in the latter case we say that the \( D(x) \) corresponds to \( I(x) \)). Let \( S' \) be the sequence obtained from \( S_2 \) by removing every such \( I(x) \) and its corresponding \( D(x) \) (if any). In other words the only \( I(x) \) operations in \( S' \) are those whose \( x \) was removed by an \( E \) in \( \emptyset S_2 \), and the only \( D(x) \) operations in \( S' \) are those whose response in \( \emptyset S_2 \) was "not in set". Obviously, the response to any operation in \( S' \) is the same in \( \emptyset S' \) as in \( \emptyset S_2 \). However, the following also holds.

**Lemma 2.1** The responses to the operations in \( S' \) are the same in \( L_1 S' \) as in \( L_1 S_2 \). The set resulting from \( L_1 S_2 \) equals \( L_2 \) plus the set resulting from \( L_1 S' \).

**Proof.** The proof is based on a careful case analysis of the types of operations that can appear in \( S \), which, for space reasons is included only in the full paper.

Lemma 2.1 has reduced the problem of solving \( L_1 S_2 \) to that of solving \( L_1 S' \), so we now focus on obtaining the responses and final set for \( L_1 S' \). The next lemma will further reduce the problem to one in which a crucial suffix property holds, as is later established in Lemma 2.3.

**Lemma 2.2** Let \( \hat{S} \) be obtained from \( S' \) by moving every \( I(x) \) to just before the \( E \) whose response it was in \( \emptyset S_2 \) (such an \( E \) must exist by definition of \( S' \)). Then the responses to the operations in \( S' \) are the same in \( L_1 \hat{S} \) as in \( L_1 S_2 \). The set resulting from \( L_1 S_2 \) equals \( L_2 \) plus the set resulting from \( L_1 \hat{S} \).