Parallel Algorithms for Relational Coarsest Partition Problems

MS-CIS-93-71
GRASP LAB 354

Sanguthevar Rajasekaran
Insup Lee

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

July 1993

Parallel Algorithms for Relational Coarsest Partition Problems *

S. Rajasekaran and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

August 5, 1993

Abstract

Relational Coarsest Partition Problems (RCPPs) play a vital role in verifying concurrent systems. It is known that RCPPs are $\mathcal{P}$-complete and hence it may not be possible to design polylog time parallel algorithms for these problems.

In this paper, we present two efficient parallel algorithms for RCPP, in which its associated label transition system is assumed to have m transitions and n states. The first algorithm runs in $O(n^{1+\epsilon})$ time using $\frac{m}{n^\epsilon}$ CREW PRAM processors, for any fixed $\epsilon < 1$. This algorithm is analogous and optimal with respect to the sequential algorithm of Kanellakis and Smolka. The second algorithm runs in $O(n \log n)$ time using $\frac{m}{n} \log n$ CREW PRAM processors. This algorithm is analogous and nearly optimal with respect to the the sequential algorithm of Paige and Tarjan.

1 Introduction

Relational Coarsest Partition Problems play an important role in verifying concurrent systems in the form of equivalence checking. In their pioneering work, Kanellakis and Smolka [7] present an efficient algorithm for RCPP with multiple relations. Their algorithm has a run time of $O(mn)$, where m is the total number of transitions and n is the number of states

*This research was supported in part by ONR N00014-89-J-1131, DARPA/NSF CCR90-14621 and ARO DAAL 03-89-C-0031.

in the RCPP. Subsequently, Paige and Tarjan [10] show that RCPP (with a single relation) can be solved in $O(m \log n)$ time. Both these algorithms have been used in practice to verify systems with thousands of states. The goal of this paper is to extend the applicability of these algorithms with the use of parallelism.

In a recent work of Zhang and Smolka [11], an attempt has been made to parallelize the classical Kanellakis-Smolka algorithm. However, the main thrust of this work was from practical considerations. In particular, complexity analysis has not been provided and was not the main concern of this paper. On the other hand, it has been shown that RCPP (even when there is only a single function) is $\mathcal{P}$-complete [1]. $\mathcal{P}$-complete problems are presumed to be problems that are hard to efficiently parallelize. It is widely believed that there may not exist polylog time parallel algorithms for any of the $\mathcal{P}$-complete problems that use only a polynomial number of processors.

Since RCPP has been proven to be $\mathcal{P}$-complete, we restrict our attention to designing polynomial time algorithms. In this paper we present two parallel algorithms for RCPP: 1) An algorithm that runs in $O(n^{1+\epsilon})$ time using $\frac{m}{n^\epsilon}$ CREW PRAM processors for any fixed $\epsilon < 1$; the same algorithm runs in time $O(n \log n)$ using $\frac{m}{\log n} \log\log n$ CRCW PRAM processors; and 2) An algorithm that runs in time $O(n \log n)$ using only $\frac{m}{n} \log n$ CREW PRAM processors. The first algorithm is optimal with respect to Kanellakis-Smolka algorithm. We say a parallel algorithm that runs in time T using P processors is optimal with respect to a sequential algorithm with a run time of S, if $PT = O(S)$, i.e., the *work done* by the parallel algorithm is asymptotically the same as that of the sequential algorithm. The two parallel algorithms described in this paper are for single relation RCPP. They can, however, be easily extended for multiple relation RCPP without changing their run-time complexities.

The rest of the paper is organized as follows. In Section 2, we provide some definitions and useful facts about parallel computation. In Sections 2 and 3, we provide our two algorithms, respectively. Finally in Section 4, we provide concluding remarks and list some open problems.

2 Problem Definitions

Definition 1 *A* labeled transition system *(LTS) M is $\langle Q, Q_0, A, T\rangle$, where Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, A is a finite set of alphabet, $T \subseteq Q \times A \times Q$ is a transition relation.*

For a given LTS $M = \langle Q, Q_0, A, T\rangle$, we define functions T_a, T_a^{-1} from Q to 2^Q for every $a \in A$ as follows:

$$\begin{aligned} T_a(p) &= \{q|(p,a,q) \in T\} \\ T_a^{-1}(q) &= \{p|(p,a,q) \in T\} \end{aligned}$$

That is, $T_a(p)$ is the set of next states of p and $T_a^{-1}(q)$ is the set of states which can lead to q via a. We extend the functions T_a and T_a^{-1} from 2^Q to 2^Q. That is, for every $a \in A$ and $S \subseteq Q$,

$$T_a(S) = \bigcup_{p\in S} T_a(p), \qquad T_a^{-1}(S) = \bigcup_{p \in S} T_a^{-1}(p)$$

Given a set S, a partition of S is a set of disjoint sets whose union is equal to S. We say that a partition $\pi' = \{B'_1, \ldots B'_n\}$ is a refinement of a partition $\pi = \{B_1, \ldots B_m\}$ if every B'_i is contained in some B_j.

We can represent an equivalence relation $\pi \subseteq Q \times Q$ as a partition $\{B_i | i \in I\}$ where each block B_i represents an equivalence class in π.

For a state $q \in Q$ and a subset S of Q, let $[q]_\pi$ denote the block in partition π which includes q, and let $[S]_\pi$ denote the set of blocks in partition π which include some state in S, that is, $[S]_\pi = \{[q]_\pi | q \in S\}$.

The notion of bisimulation equivalence as defined by Milner in [9] is used.

Definition 2 *Given a labeled transition system $S = \langle Q, Q_0, A, T\rangle$, a binary relation $\pi \subseteq Q \times Q$ is a* (strong) bisimulation *iff*

$$\begin{aligned} \forall (p_1,p_2) \in \pi.\quad \forall a \in A.(\ & \forall q_1.(q_1 \in T_a(p_1) \Rightarrow \exists q_2.(q_2 \in T_a(p_2) \wedge (q_1,q_2) \in \pi))\wedge \\ & \forall q_2.(q_2 \in T_a(p_2) \Rightarrow \exists q_1.(q_1 \in T_a(p_1) \wedge (q_1,q_2) \in \pi))). \end{aligned}$$

For $p, q \in Q$, p and q are said to be *bisimilar*, denoted by $p \sim q$, if $(p,q) \in \pi$ for some bisimulation $\pi \in Q \times Q$.

Definition 3 *Suppose that a LTS $M_1 = \langle Q_1, Q_{01}, A, T_1\rangle$ and a LTS $M_2 = \langle Q_2, Q_{02}, A, T_2\rangle$. We say two LTS M_1 and M_2 are* bisimilar *if for every $p \in Q_{01}$, there exists $q \in Q_{02}$ such that p and q are bisimilar in a LTS $M = \langle Q_1 \cup Q_2, Q_{01} \cup Q_{02}, A, T_1 \cup T_2\rangle$, and vice versa.*

To show whether or not two states are bisimilar, it suffices to show that there is a bisimulation relation that includes both of them in the same equivalence class.

There are again two important problems in LTS: the bisimulation testing problem and the greatest bisimulation finding problem. The bisimulation testing, for given two LTS's, is to decide whether or not they are bisimilar.

The *greatest bisimulation* of a given labeled transition system is a bisimulation such that any bisimulation relation in the system is a refinement of it. For a given LTS M, finding the greatest bisimulation is the same as finding the minimum LTS that is bisimilar to M.

The state minimization problem, for a given LTS $M = \langle Q, Q_0, A, T \rangle$, is to find a bisimilar LTS $M' = \langle Q', Q'_0, A, T' \rangle$ with the smallest possible number of states.

Suppose π is the greatest bisimulation of a LTS $M = \langle Q, Q_0, A, T \rangle$. Then, the minimal LTS of M is the *reduction* of M according to the greatest bisimulation π, that is, $M/\pi = \langle \pi, [Q_0]_\pi, A, T_\pi \rangle$, where $T_\pi = \{([q]_\pi, a, [q']_\pi) | (q, a, q') \in T\}$.

Both of these problems can be solved by an algorithm for the relational coarsest partitioning problem, which is defined as follows:

Relational Coarsest Partitioning Problem (RCPP)

Input: An LTS $M = \langle Q, Q_0, A, T \rangle$ with a finite state set Q, an initial partition π_0 of a set Q of states and relations $T_1, \cdots, T_k$ on $Q \times Q$.

Output: the coarsest (having the fewest blocks) partition $\pi = \{B_1, \cdots, B_l\}$ of Q such that

1. π is a refinement of π_0, and
2. for every p, q in block B_i, for every block B_j in π, and for every relation T_m,

$$T_m(p) \cap B_j \neq \emptyset \text{ iff } T_m(q) \cap B_j \neq \emptyset$$

That is, either $B_i \subseteq T_m^{-1}(B_j)$ or $B_i \cap T_m^{-1}(B_j) = \emptyset$.

2.1 Parallel Computation Models

A large number of parallel machine models have been proposed. Some of the widely accepted models are: 1) fixed connection machines, 2) shared memory models, 3) the boolean circuit model, and 4) the parallel comparison trees. Of these we'll focus on 1) and 2) only. The *time complexity* of a parallel machine is a function of its input size. Precisely, time complexity is a function $g(n)$ that is the maximum over all inputs of size n of the time elapsed when the first processor begins execution until the time the last processor stops execution.

A fixed connection network is a directed graph $G(V, E)$ whose nodes represent processors and whose edges represent communication links between processors. Usually we assume that the degree of each node is either a constant or a slowly increasing function of the number of nodes in the graph. Fixed connection networks are supposed to be the most practical models. The Connection Machine, Intel Hypercube, ILLIAC IV, Butterfly, etc. are examples of fixed connection machines.

In shared memory models (also known as PRAMs for Parallel Random Access Machines), processors work synchronously communicating with each other with the help of a common block of memory accessible by all. Each processor is a random access machine. Every step of the algorithm is an arithmetic operation, a comparison, or a memory access. Several conventions are possible to resolve read or write conflicts that might arise while accessing the shared memory. EREW (Exclusive Read Exclusive Write) PRAM is the shared memory model where no simultaneous read or write is allowed on any cell of the shared memory. CREW (Concurrent Read Exclusive Write) PRAM is a variation which permits concurrent read but not concurrent write. And finally, CRCW (Concurrent Read Concurrent Write) PRAM model allows both concurrent read and concurrent write. Write conflicts in the above models are taken care of with a priority scheme.

The parallel run time T of any algorithm for solving a given problem can not be less than $\frac{S}{P}$ where P is the number of processors employed and S is the run time of the best known sequential algorithm for solving the same problem. We say a parallel algorithm is *optimal* if it satisfies the equality: $PT = O(S)$. The product PT is referred to as *work done* by the parallel algorithm. We say a parallel algorithm that runs in time T using P processors is optimal with respect to a sequential algorithm with a run time of S, if $PT = O(S)$, i.e., the *work done* by the parallel algorithm is asymptotically the same as that of the sequential algorithm.

The model assumed in this paper is the PRAM. Though a PRAM is supposed to be impractical, it is easy to design algorithms on this model and usually algorithms developed for this model can be easily mapped on to more practical models. Also there is a simulation algorithm that will map any PRAM algorithm into an algorithm for the hypercube network (such as Ncube, Intel Hypercube, Connection Machines) with at the most a logarithmic factor of slow down [8]. Thus, all the time bounds mentioned in this paper will apply to the above machines if multiplied by a logarithmic factor.

2.2 Some Useful Facts

In this section, we state some well-known results which are used to analyze algorithms presented in this paper.

Lemma 1 *[3] If W is the total number of operations performed by all the processors using a parallel algorithm in time T, we can simulate this algorithm using P processors such that the new algorithm runs in time $\lfloor \frac{W}{P} \rfloor + T$.*

As a consequence of the above Lemma we can also get:

Lemma 2 *If a problem can be solved in time T using P processors, we can solve the same problem using P' processors (for any $P' \leq P$) in time $O\left(\frac{PT}{P'}\right)$.*

Given a sequence of numbers $k_1, k_2, \ldots, k_n$, the problem of *prefix sums computation* is to output the numbers $k_1, k_1 + k_2, \ldots, k_1 + k_2 + \ldots + k_n$. The following Lemma is a folklore [5]:

Lemma 3 *Prefix sums of a sequence of n numbers can be computed in $O(\log n)$ time using $\frac{n}{\log n}$ EREW PRAM processors.*

The following Lemma is due to Cole [4]

Lemma 4 *Sorting of n numbers can be done in $O(\log n)$ time using n EREW PRAM processors.*

The following Lemma concerns with the problem of sorting numbers from a small universe:

Lemma 5 *[2] n numbers in the range $[0, n^c]$ can be sorted in $O(\log n)$ time using $\frac{n}{\log n} \log\log n$ CRCW PRAM processors, as long as c is a constant. The same problem can be solved in $O(n^\epsilon)$ time for any fixed $\epsilon < 1$, using $\frac{n}{n^\epsilon}$ CREW PRAM processors.*

3 Algorithm I

In this section, we present a parallel algorithm for RCPP with a single relation. This algorithm runs in time $O(n^{1+\epsilon})$ using $\frac{m}{n^\epsilon}$ CREW PRAM processors, for any fixed $\epsilon < 1$. The same algorithm runs in $O(n \log n)$ time on a CRCW PRAM using $\frac{m}{\log n} \log\log n$ processors. Since our algorithm is analogous to the Kanellakis-Smolka algorithm, we present their algorithm in Figure 1 for the case of a single relation before we describe ours.

Each run of the *for* loop of Kanellakis-Smolka's Algorithm takes $O(m)$ time and this loop can be executed at most n times. Thus, the run time of this algorithm is $O(mn)$.

Figure 2 describes our parallel algorithm, which is based on Kanellakis-Smolka's Algorithm. We first explain the definitions and data structures used in our algorithm.

```
$\pi := \pi_0$
/* Initially $\pi'$ is empty */

while $\pi' \neq \pi$ do
    $\pi' := \pi$
    for every $B$ in $\pi'$ do
        select a state $p$ from $B$
        $B_1 := \emptyset$    /* $B_1 = \{q \in B | T(q) = T(p)\}$ */
        $B_2 := \emptyset$    /* $B_2 = \{q \in B | T(q) \neq T(p)\}$ */
        for every $q$ in $B$ do
            if $[T(q)]_{\pi'} = [T(p)]_{\pi'}$ then
                add $q$ into $B_1$
            else add $q$ into $B_2$
        end for
        if $B_1$ and $B_2$ are not empty then /* $B$ is split */
            $\pi := (\pi - \{B\}) \cup \{B_1, B_2\}$
    end for
end while
```

Figure 1: Kanellakis-Smolka's Algorithm for the Single RCPP

PARTITION	(1,5)	(1,7)	(2,1)	(2,3)	(3,2)	(3,4)	...
TRANSITIONS	(1,2)	(1,4)	(2,4)	(2,5)	(2,7)	(3,2)	...
B	2	3	2	3	1	3	1
TSIZE	2	3	3	3	4	3	1

Table 1: Contents of Data Structures: An Example

Definitions and Data Structures. Let $T(p)$ stand for $\{q \in Q \mid (p,q) \in T\}$, i.e., $T(p)$ is the set of states to which there is a transition from p. Similarly define $T^{-1}(p)$.

The current partition is represented as an array $PARTITION$. It is an array of size n with (block id, state) pairs. For example, a pair (i,q) represents that the state q currently belongs to the i^{th} block. We maintain that the states are stored in the array $PARTITION$ such that states belonging to the same block appear consecutively.

The array $TRANSITIONS$ is used to store the T relation of a LTS. In particular, the array is of size m and each entry contains the (from-state id, to-state id) pair. In the array $TRANSITIONS$, we store the transitions of $T(1)$, followed by the transitions of $T(2)$, and so on. $TSIZE$ is an array of size n such that $TSIZE[q]$ stands for $|T(q)|$ for each q in Q. Note that the arrays $TRANSITIONS$ and $TSIZE$ are never altered during the algorithm.

We also maintain an array B such that for each state p in Q, $B[p]$ is the id of a block in the current partition π, which p belongs to. In addition, for each state $p \in Q$, we let $[p]$ stand for the set, $\{B[q] \mid q \in T(p)\}$. We emphasize here that no repetition of elements is permitted in $[p]$. For any state q in Q, we let $[T(q)]$ stand for the sequence $B[p_1], B[p_2], \ldots, B[p_t]$, where $T(q) = \{p_1, p_2, \ldots, p_t\}$. Notice that $[T(q)]$ can have multiple occurrences of the same element.

As an example to illustrate our data structures, consider the following initial partition, π_0: $\{\{5,7\}, \{1,3\}, \{2,4,6\}\}$. Let the transition relation T be defined as follows: $T(1) = \{2,4\}$; $T(2) = \{4,5,7\}$; $T(3) = \{2,6,7\}$; $T(4) = \{1,5,6\}$; $T(5) = \{1,2,6,7\}$; $T(6) = \{2,4,6\}$; $T(7) = \{1\}$.

Table 1 shows the contents of $PARTITION, TRANSITIONS$, B, and $TSIZE$ at the beginning.

Assume that there is a processor associated with each transition and each state of the LTS. At the beginning, $PARTITION$ has tuples corresponding to the initial partition. The array $TRANSITIONS$ never gets modified in the algorithm. Array B is also initialized appropriately. For any state q, processors associated with $T(q)$ will know the position of state q in the array $PARTITION$.

The algorithm repeats as long as there are possibilities of splitting at least one of the blocks in the current partition and is described in Figure 2. Given that $\pi = \{B_1, B_2, \ldots, B_\ell\}$, $B_i = \{q_{i,1}, q_{i,2}, \ldots, q_{i,n_i}\}$, and $T(q_{i,j}) = \{p_{i,j,1}, p_{i,j,2}, \ldots, p_{i,j,m_{i,j}}\}$, Steps 1-3 are to construct the sequence L:

$$L_{1,1}, L_{1,2}, \ldots, L_{1,n_1}, L_{2,1}, \ldots, L_{2,n_2}, \ldots, L_{\ell,n_\ell}$$

where $L_{i,j}$ is a sequence of triples:

$$(i, j, B[p_{i,j,1}]), (i, j, B[p_{i,j,2}]), \ldots, (i, j, B[p_{i,j,m_{i,j}}]).$$

Steps 4-6 are to eliminate duplicates in L and compress the array L. At the end of Step 6, the array L contains $[p]$ for every state p in each block in the current partition. Furthermore, for each block $B = \{p_1, \ldots, p_k\}$, $[p_1], [p_2], \ldots, [p_k]$ appear consecutively in L.

Step 7 identifies blocks that can be split. Note that even if there is a single j such that $[q_{i,j}] \neq [q_{i,1}]$, we may end up splitting the block B_i and thus the block B_i is marked.

Step 8 picks one of the marked blocks arbitrarily and splits it. If the block B_i is chosen, then B_i is split into B_i and $B_{\ell+1}$, where $B_{\ell+1} = \{p \in B_i | [p] \neq [q_{i,1}]\}$ and B_i is updated to be $B_i - B_{\ell+1}$. After the splitting, we update $PARTITION$ such that states belonging to the same block appear consecutively. Note that we could have split all those blocks that are marked instead of just one such block as done in Step 8; even then, the worst case run-time of the algorithm would be the same.

Analysis. We assume that there are $n + m$ processors, one for each state and one for each transition.

Step 1 takes $O(1)$ time using n processors. Steps 3,5, and 7 also take $O(1)$ but need m processors. In Step 2, prefix computation can be done using $\frac{n}{\log n}$ processors in $O(\log n)$ time (see Lemma 3). In Step 4, we need to sort m numbers in the range $[0, n^3]$, and hence, we could apply Lemma 5 to infer that it can be done in $O(\log m) = O(\log n)$ time using $\frac{m}{\log n} \log\log n$ processors, or in n^ϵ time using $\frac{m}{n^\epsilon}$ processors for any fixed $\epsilon < 1$. Step 6 takes $O(\log m) = O(\log n)$ time using $\frac{m}{\log n}$ processors (see Lemma 3). In Step 8, prefix computation takes $O(\log n)$ time using $\frac{n}{\log n}$ processors and the rest of the computation can be completed in $O(1)$ time using n processors.

Thus, each run of the while loop can be completed in either: 1) $O(\log n)$ time with a total work of $m \log\log n$, or 2) $O(n^\epsilon)$ time with a total work of $O(m)$. Since the while loop can be executed at most n times, we get by applying Lemma 1:

Theorem 1 *RCPP with m transitions and n states can be solved 1) in $O(n \log n)$ time using $\frac{m}{\log n} \log\log n$ CRCW PRAM processors, or 2) in $O(n^{1+\epsilon})$ time and $\frac{m}{n^\epsilon}$ CREW PRAM processors, for any fixed $\epsilon < 1$.*

```
$\pi := \pi_0$; split := true
*while* split *do*
    split := false; let $\pi = \{B_1, B_2, \ldots, B_\ell\}$
    Unmark $B_1, B_2, \ldots, B_\ell$
    **1. for** $i := 1$ **to** $n$ **in parallel do**
        $TEMP[i] := TSIZE[PARTITION[i].state]$
    **2.** Compute the prefix sums of $TEMP[1]$, $TEMP[2]$, $\ldots, TEMP[n]$
        Let the sums be $v_1, v_2, \ldots, v_n$
    **3. for** $i := 1$ **to** $n$ **in parallel do**
        $s_i := PARTITION[i].state$;
        Let $T[s_i]$ be $\{q_1, \ldots, q_k\}$
        **for** $j := 1$ **to** $k$ **in parallel do**
        Let processor in-charge of transition $(s_i, q_j)$ write $(i, j, B[q_j])$ in $L[v_{i-1} + j]$
    **4.** Sort the sequence $L$ in lexicographic order.
    **5. for** $i := 1$ **to** $m$ **in parallel do if** $L[i] = L[i+1]$ **then** $L[i] := 0$
    **6.** Compress the list $L$ using a prefix computation
    **7. for each** block $B_i$ $(1 \leq i \leq \ell)$ **in parallel do**
        **for each** $j$, $2 \leq j \leq n_i$ **in parallel do**
            **if** $[q_{i,j}] \neq [q_{i,1}]$ **then** mark $B_i$
    **8. if** there is at least one marked block **then**
        split := true; $\ell := \ell + 1$
        Pick one of the marked blocks (say $B_i$) arbitrarily
        **for** each $p$ in $B_i$ **do**
            **if** $[p] \neq [q_{i,1}]$ **then**
                $B[p] := \ell + 1$
                Change the corresponding entry in $PARTITION$ to $(p, \ell + 1)$
        /* $B_{\ell+1} := B_i - \{p \in B_i : [p] = [q_{i,1}]\}$ and $B_i := B_i - B_{\ell+1}$ */
        Using a prefix computation, modify $PARTITION$ such that all tuples
        corresponding to the same block are in successive positions.
        When the array $PARTITION$ is modified, positions of some
        states $q$'s might change; inform the processors associated with
        the corresponding $T(q)$'s of this change. $\pi$ has been thus modified.
```

Figure 2: Algorithm I

4 Algorithm II

In this section, we present a parallel version of Paige and Tarjan's algorithm [10]. This algorithm has a run time of $O(n \log n)$ using $\frac{m}{n} \log n$ CREW PRAM processors. Thus, this algorithm is nearly optimal with respect to [10]'s. We first give a brief description of Paige and Tarjan's algorithm, followed by the parallel algorithm.

4.1 The Sequential Algorithm

Paige and Tarjan [10] present an efficient algorithm for the "relational" partitioning problem of an "unlabeled" transition system $S = \langle Q, Q_0, A, T \rangle$, where $T \subseteq Q \times Q$. That is, there is only one kind of relation. Without loss of generality, they assume $|T(p)| \geq 1$ for all $p \in Q$. The reason is that given an initial partition π_0, it can be refined into $\pi_1 \cup \pi_2$, where $\pi_1 = \{B' \neq \emptyset | B' = B \cap T^{-1}(Q)\}$ and $\pi_2 = \{B' \neq \emptyset | B' = B - T^{-1}(Q)\}$. Since the blocks in π_2 will not be split, it suffices to consider only π_1.

For $S \subseteq Q$ and a partition π, we define *split*(S, π) to be a new partition π' such that each block D in π is replaced by $D \cap T^{-1}(S)$ and $D - T^{-1}(S)$. If either of them is empty set, then it is not included in π'. The resulting π' has the following properties: 1) π' is a refinement of π, and 2) π' consists of the largest blocks that are stable with respect to S.

The major idea behind Paige and Tarjan's algorithm is to show that *split*$(S-B,$ *split*$(B, \pi))$ can be computed in $O(|T^{-1}(B')|)$ time, where B' is smaller of B and $S - B$. This idea, called the process-smaller-half strategy, is as follows: Suppose S is a union of some blocks of π such that π is stable with respect to S, and $B \subseteq S$ is in π. For every block $D \in \pi$, if $D \cap T^{-1}(S) = \emptyset$, then D is stable with respect to B and $S - B$; otherwise, D can be split by *split*$(S - B,$ *split*$(B, \pi))$ into three blocks, D_1, D_2, D_3, as follows:

1. $D_1 = D - T^{-1}(S - B)$: the successors of D_1 are only in B since π is stable with respect to S.

2. $D_2 = D \cap T^{-1}(B) \cap T^{-1}(S - B)$: the successors of D_2 are in both B and $S - B$.

3. $D_3 = D - T^{-1}(B)$: the successors of D_3 are only in $S - B$.

For $p \in Q$ and a subset S of Q, let *count*(p, S) be the number of the next states of p in S, that is, *count*$(p, S) = |S \cap T(p)|$. Assuming that we have already computed *count*(p, S) and *count*(p, B) for all $p \in D$, we can decide which of the three blocks D_1, D_2, D_3 that p belongs as follows:

1. p in D_1 if $count(p, B) = count(p, S)$, i.e., there are transitions from p to B but not to $S - B$.

2. p in D_2 if $0 < count(p, B) < count(p, S)$, i.e., there are transitions from p to both B and $S - B$.

3. p in D_3 if $count(p, B) = 0$, i.e, there are transitions from p to $S - B$ but not to B.

The algorithm uses a set X of splitters. An element in X is a tree of height 0 or 1 with the following properties: each leaf is a block in the current partition; the root is the union of its children blocks; and the current partition is stable with respect to the root. There are six major steps in the algorithm:

In Step 0, X is initialized with one tree whose children are blocks in the initial partition π_0. Throughout the algorithm, $count(p, B)$ is maintained for each state p in Q and for each block B that is a root in X such that $p \in T^{-1}(B)$. Step 1 selects an arbitrary block B that is going to be used to split the current partition π. Step 2 computes $count(p, B)$ for $p \in T^{-1}(B)$ since B will become a new tree root in X. Step 3 carries out the three way splitting described above. Step 4 updates $count(p, S)$ for $p \in T^{-1}(B)$ since B has been eliminated from the tree rooted at S in X. For each block D that has been split into D_1, D_2, D_3, Step 5 updates π to include the new blocks D_1, D_2, D_3 and also inserts them into X as potential splitters.

Analysis. For timing analysis, the algorithm uses the following data structures: for each block, the algorithm keeps the size and maintains its member states as a doubly linked list. In addition, each block itself is a member of a doubly linked list. For each state, the algorithm maintains a pointer to a block in which it is a member.

Step 0 takes $O(m)$ time. Step 1 can be completed in constant time. Steps 2-4 take $O(|T^{-1}(B)|)$ time, where $T^{-1}(B) = \sum_{p \in B} T^{-1}(p)$. Step 5 also takes $O(|T^{-1}(B)|)$ time since there are at most $O(|T^{-1}(B)|)$ marked blocks.

Each time state p is in a chosen splitter, it takes $|T^{-1}(p)|$ time to process it (in Steps 2-5). Since each state can be in at most $\log n$ splitters due to the process-smaller-half strategy, the total time incurred due to any state p is at most $|T^{-1}(p)| \log n$. Therefore, the total time is

$$\sum_{p \in Q} |T^{-1}(p)| \log n = (\sum_{p \in Q} |T^{-1}(p)|) \log n = m \log n$$

The running time of the algorithm is $O(m \log n)$ and the space used is $O(n + m)$, where $n = |Q|$ and $m = |T|$.

4.2 The Parallel Algorithm

Figures 4, 5 and 6 describe how to implement each of the above steps in parallel. The basic steps are the same as those of the sequential algorithm. There are, however, some intricate details in the definitions of and operations on the data structures used in our algorithm. X is the collection of splitters. Each entry in X is a tree of height one or zero. A tree of height one is called a *compound splitter*, whereas a tree of height zero is called a *simple splitter*. All the simple splitters as well as leaves of compound splitters are blocks in the current partition. Moreover, the current partition is stable with respect to each root in X (including simple splitters). We do not maintain the current partition as a separate data structure, since the current partition can be readily derived from X.

Data Structures. We employ the following data structures:

- *ITRANSITIONS* is an array of size m. This array is a sequence of records, one record per transition. Each record contains a pair (x, y) which corresponds to a transition from state x to state y, and a number that equals to $count(x, S)$, where S is the root in X that y belongs to. These records are ordered according the second component of (x, y) transition. That is, transitions to state 1 are placed before transitions to state 2, etc.

- *ITSIZE* is an array of size n. This array contains $|T^{-1}(p)|$ for each p in Q.

- B is an array of size n. For each state p in Q, $B[p]$ has a pointer to a node in X that p belongs to.

- *XARRAY* is an array of size $O(n)$. This array of records maintains both compound and simple splitters. Each compound splitter has a structure shown in Figure 3. Children of a compound splitter are represented as a doubly linked list. Each element in this list is a block in the current partition which is represented as a doubly linked list of states. For instance, a tree with three children blocks, say A, B, and C is shown in Figure 3. Blocks A, B, and C themselves are doubly linked lists of states in the corresponding blocks. The root of the tree is not represented as a separate node in *XARRAY*.

  Compound splitters themselves are doubly linked. This linked structure is useful in the following sense: When the current splitting block is removed from its splitter tree, if this compound splitter tree has two children, then it is likely that this splitter becomes simple. If this happens, the splitter tree is linked to the list of simple splitters. In the next phase of the algorithm, we choose the next compound splitter following the link structure of compound splitters. A similar structure is adopted for simple splitters as well (see Figure 3).

The crucial fact about *XARRAY* is that we maintain this linked structure in the form of an array of records. Each record has four pointers and a state id. These four pointers are used to represent a tree structure, as shown in Figure 3.

This enables us to perform deletion of elements from these lists efficiently. In addition, we could retrieve all the elements of any list efficiently in parallel. For instance in Paige and Tarjan's algorithm, one of the basic steps to be performed is the selection of a splitter block B.

One important aspect of *XARRAY* is that blocks always occupy mutually disjoint segments of *XARRAY*. Each segment includes contiguous array elements of *XARRAY*. However, the number of array elements used to represent a block can be larger than the size of the block. That is, there can be some array elements which contain no state records for the block. Such elements are called "cavities."

Algorithm. The functionalities of each step in the parallel algorithm is the same as those of a corresponding step in the sequential algorithm. In each phase of Paige and Tarjan's algorithm, we pick a splitter block B that is a child of a compound splitter and perform three way splitting of blocks, possibly including B. Deleting an element from any block can be done in $O(1)$ time. But then, this might create 'cavities' in the array *XARRAY*. When we have to retrieve any block B, we have to look at all the elements of X that B used to occupy before any cavities were created in B. Thus it seems like some unnecessary work may be done in retrieving B. On the other hand, parallel retrieval of B amounts to just one prefix computation. Whenever a block (say D) is split into (say) D_1, D_2, D_3, we delete elements from D (that belong to D_1 and D_2) and store these new blocks as lists by extending the array (i.e., we store them starting from the first unoccupied position in the array). The remaining list of D will be called D_3.

As a result, now even though D_3 may only have a few states, this list is stored in a space that D used to occupy before. In order to retrieve D_3 at a later stage we will have to search the whole segment of the array that D once occupied. Note that we will have to retrieve any block only if it has been chosen as the splitter block. In our analysis, we will account for all

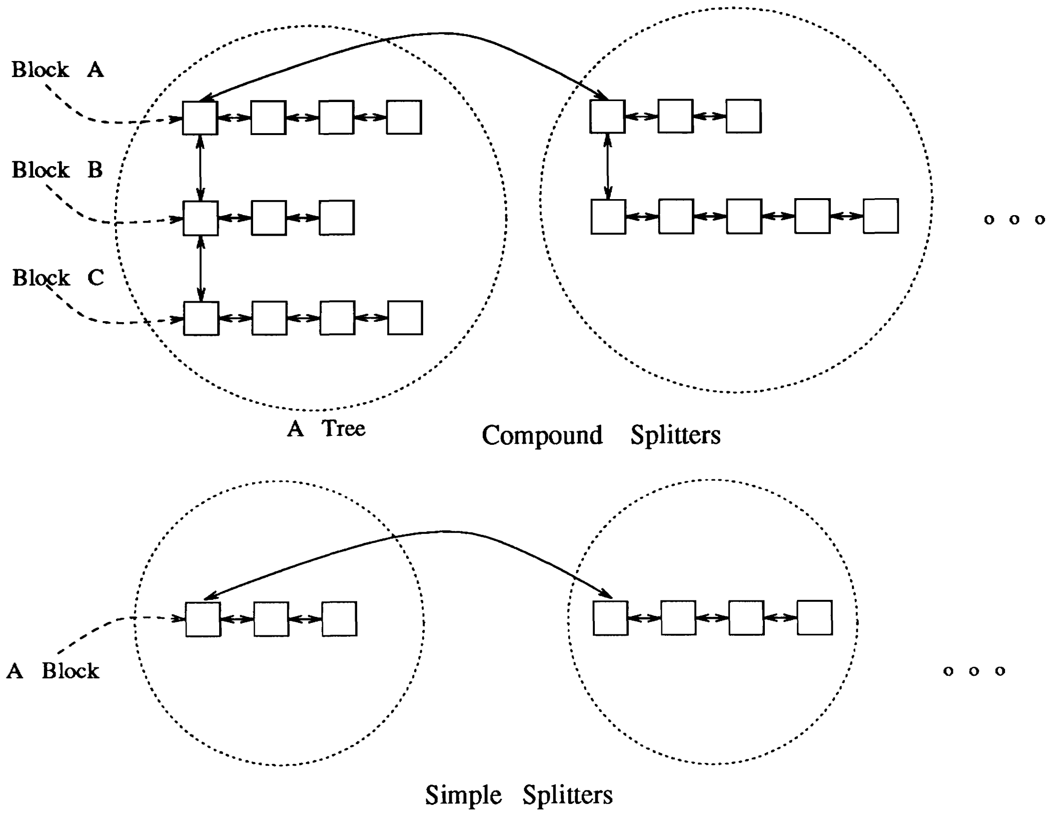

Figure 3: The Data Structures Used for Splitters

the unnecessary retrieval work performed.

Analysis. In Step 0, sorting can be done in time $O(\log m) = O(\log n)$ with a total work of $O(m \log n)$ (see Lemma 4). The rest of the operations here can be completed in $O(\log n)$ time with a total work of $O(n)$. Step 1 can be performed in $O(1)$ time with a single processor. Step 2(a) is nontrivial to analyze and discussed below in detail. In Step 2(b), the dominating operation is sorting, which takes $O(|T^{-1}(B)| \log n)$ work and $O(\log n)$ time. The rest of the tasks such as prefix computations can be completed within the same time and $O(|T^{-1}(B)| + |B|)$ total work. Step 3(a) can be done in $O(1)$ time and $O(|T^{-1}(B)|)$ work. Also in Step 3(b), sorting takes the longest time to complete. The total work done is $O(|T^{-1}(B)| \log n)$ and the time needed is $O(\log n)$. Steps 4 and 5 can be performed in $O(1)$ time and $O(|T^{-1}(B)|)$ work.

Notice that any state of Q can appear as a member of some splitting block at most $\log n$ times. Taking into account Steps 1 through 5 (except 2(a)), whenever a state p appears as a member of a splitting block, we spend a total work of $O(|T^{-1}(p)| \log n)$ for this state.

```
Step 0:
    $\pi := \pi_0$
    Let $S_Q$ be a splitter whose leaves are all the blocks in $\pi_0$.
    /*Note that the union of the leaves of $S_Q = Q$) */
    $X := \{S_Q\}$
/* Initially all blocks in $\pi$ are unmarked */

    For every $(x, y)$ in $ITRANSITIONS$ create $(y, x)$ and sort these tuples.
    Using this sorted tuples
    for every $p$ in $Q$ in parallel do
        count$(p, Q) := |\{q \in Q | q \in T(p)\}|$
    end for
while $X$ not empty do
Step 1:
    Select and remove any $S$ from $X$.
    Let $B_1$ and $B_2$ be the first two blocks in $S$.
    if $|B_1| \leq |B_2|$ then $B := B_1$ else $B := B_2$
    Remove $B$ from $S$ and make it a simple splitter.
    if $S$ includes more than two blocks
        thenPut $S$ back in its previous place.
        else make it a simple splitter.
Step 2(a):
    Let $v$ be the size of the portion of $XARRAY$ used to store $B$.
    Using $v$ processors, perform a prefix operation to retrieve $B$
    in parallel and put the elements of $B$ in $TEMP'$.
    At the same time, compress the portion that $B$ occupies.
```

Figure 4: Algorithm II

```
Step 2(b):
    Let $TEMP''$ be the list of $ITSIZE[q]$ for elements $q$'s in $TEMP'$
    $|B|$ processors perform a prefix sums operation on the sequence $TEMP''$
    Using the above prefix sums and $|T^{-1}(B)|$ processors,
    retrieve $T^{-1}(B)$ and store the elements in $TEMP$.
    Sort the list $TEMP$.
    Using this sorted order and two prefix operations,
    for every $p$ in $TEMP$ in parallel do
        count$(p, B) := |\{q \in B | q \in T(p)\}|$
    end for
Step 3(a):
    for every $p$ in $TEMP$ in parallel do
        Let $D$ be the block including $p$ in $\pi$.
        Mark $D$.
        if count$(p, B) =$ count$(p, S)$ then
            label $p$ as type I /* $p$ belongs to $D_1 = D - T^{-1}(B)$ */
        else /* $0 < count(p, B) < count(p, S)$ */
            label $p$ as type II /* $p$ belongs to $D_2 = D \cap T^{-1}(B) \cap T^{-1}(S - B)$ */
    end for
Step 3(b):
    $|TEMP|$ processors perform a prefix sums operation and
    pick elements of $TEMP$ that are of type I or II.
    Let the new list be $TEMP'$.
    for every $p$ in $TEMP'$ in parallel do
        Create a tuple $(B[p], POS_p)$ and add the tuple to $TUPLES$,
        where $POS_p$ is the position of elements $p$ in $XARRAY$
    Sort $TUPLES$ in lexicographic order /* These elements have to be deleted */
    Append the newly created blocks to $XARRAY$
    Delete these elements from their respective blocks in $XARRAY$;
    The sorted array $TUPLES$ helps in setting the pointers correctly
    in cases where more than one successive nodes will have to be
    deleted in parallel from any list
```

Figure 5: Algorithm II (continued)

```
Step 4:
    for every $p$ in $TEMP$ in parallel do
        count$(p, S)$ := count$(p, S)$ − count$(p, B)$
    end for
Step 5:
    for every $D$ in $\pi$ such that $D$ marked in parallel do
        Unmark $D$
        $P := \{D_i \neq \emptyset | i = 1, 2, 3\}$
        if $|P| = 1$ then /* $D$ is not split */
            retain the original position of $D$
        else
            if $D$ is a child of a compound splitter $S'$ in $X$ then
                Make $D_1$ and $D_2$ (if nonempty) as leaves of $S'$.
            else
                Make $D$ a compound splitter with leaves that are
                all the blocks $D_i$ in $P$.
    end for
end while
```

Figure 6: Algorithm II (continued)

For every splitting block B, we also spend a total work of $O(|B|)$. Therefore, summing over the whole algorithm, the total work done in Steps 0 through 5 (excepting Step 2(a)) is $\sum_{p\in Q} |T^{-1}(p)|\ \log^2 n$, which simplifies to $O(m \log^2 n)$.

We now analyze Step 2(a). Assume that we never compress $XARRAY$. How large can the array grow? Observe that whenever we split any block D, we might create new blocks and hence use new space in $XARRAY$. Clearly, an upper bound for the new space used in any run of the while loop is $|T^{-1}(B)|$, implying that we will never have to extend $XARRAY$ to more than $m \log n$ in length.

How much total time is spent in Step 2(a)? Notice that whenever we have to retrieve a block (say B) as a splitter block, we have to search through the whole space (including cavities) that B is stored in. If each such region is searched no more than once, then the total work done in Step 2(a) is clearly $O(m \log n)$. But the same region may have to be searched again and again. However, whenever we retrieve a block B, we compress it immediately. We, thus, charge an additional work of $O(|B|)$ for compression whenever a splitter block is retrieved (this accounts for the next time that this region may have to be searched).

Therefore, the total work needed for repeated searches of the regions is no more than $O(|B|)$ summed over all the splitter blocks used in the whole algorithm, which is $O(n \log n)$. In sum, the total work needed for processing Step 2(a) in the whole algorithm is $O(m \log n)$.

The total time spent in each run of the while loop is $O(\log n)$, but there could be at most n runs of the while loop. Thus, the total run time is $O(n \log n)$.

An application of Lemma 1 yields the following theorem.

Theorem 2 *Single relation CRPP can be solved in $O(n \log n)$ time using $\frac{m}{n} \log n$ CREW PRAM processors.*

Memory Management. As stated, the above algorithm seems to use $O(m \log n)$ memory to maintain $XARRAY$. But we could easily reduce the memory needed for $XARRAY$ to $O(n)$ as follows: Whenever the memory needed to store $XARRAY$ exceeds $2n$ records, we perform a compression of the array so that at the end of compression, $XARRAY$ will be of size n records. The amount of work done for compression is $O(n)$. Such a compression is done at most $\frac{m}{n} \log n$ times in the algorithm since the array can only grow up to $m \log n$ in size. Therefore, the total work done for compression is $O(m \log n)$.

5 Conclusions

We have presented two parallel algorithms for RCPP. An interesting open problem is to design optimal versions of these algorithms. The bottleneck in these algorithms is the use of sorting. Another important open problem will be to design algorithms with better run times. Since RCPP is known to be $\mathcal{P}$-complete, a reasonable time to aim for will be $O(n^\epsilon)$, for any fixed $\epsilon < 1$.

References

[1] C. Alvarez, J.L. Balcazar, J. Gabarro, and M. Santha, Parallel Complexity in the Design and Analysis of Concurrent Systems, Springer-Verlag LNCS 505, 1991.

[2] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, and S. Saxena, Improved Deterministic Parallel Integer Sorting, Information and Computation 94, 1991, pp. 29-47.

[3] R.P. Brent, The Parallel Evaluation of General Arithmetic Expressions, Journal of the ACM 21(2), 1974, pp. 201-208.

[4] R. Cole, Parallel Merge Sort, SIAM Journal on Computing 17, 1988, pp. 770-785.

[5] J. Já Já, *An Introduction to Parallel Algorithms*, Addison-Wesley Publications, 1992.

[6] P.C. Kanellakis, S.A. Smolka, CCS Expressions, Finite State Processes, and Three Problems of Equivalence, Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing, 1983, pp. 228-240.

[7] P.C. Kanellakis, S.A. Smolka, CCS Expressions, Finite State Processes, and Three Problems of Equivalence, Information and Computation 86, 1990, pp. 43-68.

[8] T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays–Trees–Hypercubes*, Morgan-Kaufmann Publishers, San Mateo, California, 1992.

[9] R. Milner, *Communications and Concurrency*, Prentice-Hall Publishers, 1989.

[10] R. Paige and R.E. Tarjan, Three Partition Refinement Algorithms, SIAM Journal on Computing, 16(6), 1987, pp. 973-989.

[11] S. Zhang and S.A. Smolka, Towards Efficient Parallelization of Equivalence Checking Algorithms, Manuscript, 1993.