A Parallel Algorithm for Relational Coarsest Partition Problems and Its Implementation *

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

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 a parallel algorithm for RCPP, in which its associated label transition system is assumed to have m transitions and n states. This algorithm runs in $O(n^{1+\epsilon})$ time using $\frac{m}{n^\epsilon}$ EREW PRAM processors, for any fixed $\epsilon < 1$. This algorithm is analogous and optimal with respect to the sequential algorithm of Kanellakis and Smolka. The same algorithm runs in time $O(n \log n)$ using $\frac{m}{\log n} \log\log n$ CRCW PRAM processors. We also describe implementation and experimental results on performance of our algorithm.

1 Introduction

Relational Coarsest Partition Problems (RCPPs) play an important role in verifying concurrent systems in the form of equivalence checking. In their pioneering work, Kanellakis and Smolka [6] presented an efficient algorithm for the RCPP with multiple relations. Their algorithm had a run time of $O(mn)$, where m is the number of transitions and n is the number of states in the RCPP. This algorithm has been used in practice to verify systems with thousands of states. Our work is to extend the applicability of this algorithm with the use of parallelism.

In a recent work of Zhang and Smolka [9], 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

*This research was supported in part by NSF CCR93-11622 and DARPA/NSF CCR90-14621.

and was not the main concern of the 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 a parallel algorithm for RCPP: This algorithm runs in $O(n^{1+\epsilon})$ time using $\frac{m}{n^\epsilon}$ EREW (Exclusive-Read Exclusive-Write) PRAM processors for any fixed $\epsilon < 1$. This algorithm is optimal with respect to the 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 same algorithm runs in time $O(n \log n)$ using $\frac{m}{\log n} \log\log n$ CRCW (Concurrent-Read Concurrent-Write) PRAM processors. The parallel algorithm described in this paper is for single-relation RCPP. It can, however, be easily extended for multiple-relation RCPP without changing the asymptotic run-time complexity or processor bound.

The rest of the paper is organized as follows. In Section 2, we state the problem and provide some useful facts about parallel computation. Section 3 gives details of our parallel algorithm and Section 4 describes an example that explains how our algorithm works. Section 5 presents analysis of our algorithm and Section 6 reports our implementation results. Section 7 concludes the paper.

2 Problem Statement

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

Definition 2 *For any state $p \in Q$, let $T(p) = \{q \in Q | (p,q) \in T\}$. Also for any subset B of Q, let $T(B)$ stand for $\cup_{p \in B} T(p)$. Similarly define $T^{-1}(p)$ and $T^{-1}(B)$ for any $p \in Q$ and for any $B \subseteq Q$.*

The Relational Coarsest Partitioning Problem (RCPP) is defined as follows:

Input: An LTS $M = \langle Q, Q_0, T\rangle$ with a finite state set Q, an initial partition π_0 of Q and a relation T 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, and for every block B_j in π,

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

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

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, i.e., 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.

The model assumed in this paper is the PRAM. Though no PRAM machines exist, it is easy to describe algorithms on this model and usually algorithms developed for this model can be easily mapped on to more practical models.

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.*

This problem can also be solved in $O(n^\epsilon)$ time for any fixed $\epsilon < 1$, using $\frac{n}{n^\epsilon}$ EREW PRAM processors.

3 A Parallel Algorithm Based on Kanellakis-Smolka Algorithm

The Kanellakis-Smolka algorithm runs sequentially in $O(nm)$ time, where n is the number of states and m is the number of transitions. The basic idea behind the Kanellakis-Smolka algorithm is to split a block in the current partition if not all states in the block can go to the same set of blocks.

Figure 1 outlines our parallel algorithm which is based on the Kanellakis-Smolka algorithm. The algorithm uses known parallel algorithms for sorting and prefix sums. We describe the data structures used in the algorithm and then the steps of the algorithm.

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. We also define $T^{-1}(p)$ to be $\{q \in Q \mid (q,p) \in T\}$.

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 the array $PARTITION$ such that states belonging to the same block appear consecutively.

```
$\pi$ := $\pi_0$; 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 $(B[s_i], V[s_i], B[q_j])$ 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.
```

Figure 1: Parallel Algorithm Based on Kanellakis-Smolka Algorithm

The array $TRANSITIONS$ is used to store the relation T of the LTS. In particular, the array is of size m and each entry contains the (from-state, to-state) 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 to which p belongs in the current partition π. 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. Also, let $V[s]$ stand for the position of state s within its block. We let $n_i = |B_i|$ for any block B_i in the current partition and denote the jth element of block B_i by $q_{i,j}$.

As an example to illustrate our data structures, consider the following initial partition,

$$\pi_0 = \{\{a,b,c\},\ \{d,e,f\},\ \{g,h,i\}\}.$$

Let the transition relation T be defined as follows: $T(a) = \{d,f\}$, $T(b) = \{d\}$, $T(c) = \{e,f\}$, $T(d) = \{g,i\}$, $T(e) = \{a,b\}$, $T(f) = \{g\}$, $T(g) = \{a\}$, $T(h) = \{b,c,d\}$, $T(i) = \{a,b\}$. Table 1 shows the contents of $PARTITION$, $TRANSITIONS$, B, and $TSIZE$ at the beginning.

$PARTITION$	$(1,a)$	$(1,b)$	$(1,c)$	$(2,d)$	$(2,e)$	$(2,f)$	$\ldots$
$TRANSITIONS$	(a,d)	(a,f)	(b,d)	(c,e)	(c,f)	(d,g)	$\ldots$
B	1	1	1	2	2	2	$\ldots$
$TSIZE$	2	1	2	2	2	1	$\ldots$

Table 1: Contents of Data Structures: An Example

At the beginning, $PARTITION$ has tuples corresponding to the initial partition. The array $TRANSITIONS$ never gets modified in the algorithm. The array B is also initialized appropriately. For any state q, processors associated with $T(q)$ keep track of the position of state q in the array $PARTITION$.

The algorithm repeats as long as there is a possibility of splitting at least one of the blocks in the current partition. Steps 1-3 are to construct a sequence L of triples. Each state contributes a triple corresponding to each one of transitions going out of the state. If s_i is any state such that $T(s_i) = \{q_1, q_2, \ldots, q_k\}$, then the corresponding triples are $(B[s_i], V[s_i], B[q_j])$, for $j = 1, 2, \ldots, k$.

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 in parallel 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.

4 An Illustrative Example

We now illustrate our algorithm with an example. The example considered is the same as above. The initial partition π_0 is given by $\{\{a,b,c\}\ \{d,e,f\}\ \{g,h,i\}\}$. The transition relation is defined as:

$$T(a) = \{d,f\};\ T(b) = \{d\};\ T(c) = \{e,f\};\ T(d) = \{g,i\};\ T(e) = \{a,b\};$$

$$T(f) = \{g\};\ T(g) = \{a\};\ T(h) = \{b,c,d\};\ T(i) = \{a,b\}.$$

The initial contents of various data structures are shown in Table 1. We call each run of the *while* loop as a phase of the algorithm.

Phase I: At the end of Step 3 the list L looks like:

$$(1,1,2),(1,1,2),(1,2,2),(1,3,2),(1,3,2),(2,1,3),(2,1,3),(2,2,1),(2,2,1),$$

$$(2,3,3),(3,1,1),(3,2,1),(3,2,1),(3,2,2),(3,3,1),(3,3,1)$$

In Step 4, L is sorted in lexicographic order. The above L happens to be in sorted order already. Steps 5 and 6 compress L as follows.

$$(1,1,2),(1,2,2),(1,3,2),(2,1,3),(2,2,1),(2,3,3),(3,1,1),(3,2,1),(3,2,2),(3,3,1)$$

In Step 7, the algorithm realizes that: $[q_{2,2}] \neq [q_{2,1}]$; $[q_{3,2}] \neq [q_{3,1}]$. Therefore, the blocks B_2 and B_3 will be marked.

In Step 8, one of the marked blocks is picked arbitrarily. Let B_2 be the picked block. B_2 gets split into two blocks namely $\{d,f\}$ and $\{e\}$. *PARTITION* gets modified to:

$$(1,a)\ (1,b)\ (1,c)\ (2,d)\ (2,f)\ (4,e)\ (3,g)\ (3,h)\ (3,i)$$

Phase II: The list L after Step 3 looks like:

$$(1,1,2),(1,1,2),(1,2,2),(1,3,4),(1,3,2),(2,1,3),(2,1,3),(2,2,3),(4,1,1),$$

$$(4,1,1),(3,1,1),(3,2,1),(3,2,1),(3,2,2),(3,3,1),(3,3,1)$$

After L gets sorted and compressed (in Steps 4 through 6), L becomes:

$$(1,1,2),(1,2,2),(1,3,2),(1,3,4),(2,1,3),(2,2,3),$$
$$(3,1,1),(3,2,1),(3,2,2),(3,3,1),(4,1,1)$$

In Step 7, blocks B_1 and B_3 get marked. In Step 8, one of the marked blocks (say B_1) gets chosen. As a result, $PARTITION$ gets modified as follows:

$$(1,a)\ (1,b)\ (5,c)\ (2,d)\ (2,f)\ (4,e)\ (3,g)\ (3,h)\ (3,i)$$

Phase III: The list L gets formed in Step 3:

$$(1,1,2),(1,1,2),(1,2,2),(5,1,4),(5,1,2),(2,1,3),(2,1,3),(2,2,3),(4,1,1),$$
$$(4,1,1),(3,1,1),(3,2,1),(3,2,5),(3,2,2),(3,3,1),(3,3,1)$$

In Steps 4 through 6, L gets modified as follows:

$$(1,1,2),(1,2,2),(2,1,3),(2,2,3),(3,1,1),(3,2,1),(3,2,2),(3,2,5),(3,3,1),$$
$$(4,1,1),(5,1,2)(5,1,4)$$

In Step 7, B_3 gets marked and hence is chosen in Step 8 for splitting. $PARTITION$ now becomes:

$$(1,a)\ (1,b)\ (5,c)\ (2,d)\ (2,f)\ (4,e)\ (3,g)\ (3,i)\ (6,h)$$

Phase IV: No block gets marked in this phase and hence the algorithm terminates to yield the final partition of: $\{\{a,b\}\ \{c\}\ \{d,f\}\ \{e\}\ \{g,i\}\ \{h\}\}$.

5 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 3,5,7 also take $O(1)$ time but need m processors. In Step 2, prefix computation can be done using $\frac{n}{\log n}$ EREW PRAM processors in $O(\log n)$ time (by Lemma 3). In Step 4, we need to sort m numbers in the range $[0,n^3]$, and hence, we apply Lemma 5 to infer that it can be done in $O(\log m) = O(\log n)$ time using $\frac{m}{\log n}\log\log n$ CRCW PRAM processors, or in n^ϵ time using $\frac{m}{n^\epsilon}$ EREW PRAM processors for any fixed $\epsilon < 1$. Step 6 takes $O(\log m) = O(\log n)$ time using $\frac{m}{\log n}$ EREW PRAM processors (by Lemma 3). In Step 8, prefix computation takes $O(\log n)$ time using $\frac{n}{\log n}$ EREW PRAM 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 $m\log\log n$ on the CRCW PRAM, or 2) $O(n^\epsilon)$ time with a total work of $O(m)$ on the EREW PRAM. Since the while loop can be executed at most n times, we get the following theorem (using Lemmas 1 and 2):

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 $\frac{m}{n^\epsilon}$ EREW PRAM processors, for any fixed $\epsilon < 1$.*

The same algorithm can be modified easily to the case of multiple-relation RCPP to use quadruples instead of triples in the list L. The stated processor and time bounds still hold, where m is the total number of transitions in all the relations.

6 Implementation Details

We have implemented our parallel algorithm on two parallel machines, CM2 and CM5 of the Thinking Machines Corp. We employed the CM2 located in the CIS department of the University of Pennsylvania for program development. CM2 is a SIMD machine and has 4096 processing elements. Input and output are through a front end (which is a sun 3/60 work station). Each processing element is bit serial and can compute any boolean function that maps three bits into two bits. On the other hand, CM5 is a MIMD machine with 512 processing elements. Unlike the CM2 processors, CM5 processors are quite powerful; each processing element is comparable to a work station in computing power. We accessed the CM5 located in the CS department of University of Illinois at Urbana-Champaign through internet.

Both CM2 and CM5 provide a routing network for the processors to communicate. In CM2 the underlying routing network has a topology of a hypercube; whereas in CM5, the routing network takes the topology of a fat tree. There are special hardware to handle operations such as scan, broadcast, etc., in both of these machines.

In CM5, we can choose a subset of the processors to work with at any time. We have exploited this facility to study the scalability of our parallel program. Though CM2 supports virtual processors, it does not support selection of a subset. One could run programs written for CM2 on CM5 without much effort. We have coded our algorithm in C* (a parallel programming language supported by CM2 which is very similar to C). The same program runs on CM5. The main objective of this experiment was to study the behavior of the program when the number of processors used changes.

Input to the program was generated as follows: We fix the number of states (call it N) and the number of initial blocks in the Relational Coarsest Partition Problem. States in each block were chosen randomly (under a uniform distribution). Transitions were also picked randomly. Transition Probability, T_p, is a parameter that the user can choose. Each possible transition is picked with this probability.

Figure 2 shows the results of our experiment with $N = 10,000$ and $T_p = 0.05$. The number of initial blocks was 50. The program was run with various number of processors: 32, 64, 128, 256 and 512. For each processor configuration, the time indicated is the average of 5 independent runs.

Solid lines correspond to total execution time of the program, whereas the dotted lines correspond to the time spent on just sorting (in Step 4). 65 to 70 % of the total execution time is spent on sorting. Since the execution time of our program is always

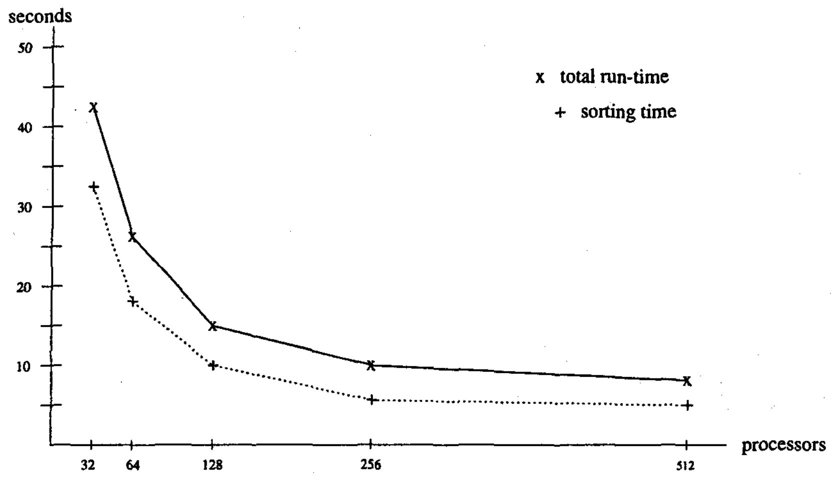

Figure 2: Execution Times on CM5

bounded below by how fast the parallel machine can sort, we are currently exploring ways of substituting sorting with some other operations.

7 Conclusions

We have presented a simple parallel algorithm for RCPP and its implementation. An interesting open problem is to design faster versions of this algorithm. The bottleneck in this algorithm is the use of sorting. Since RCPP is known to be $\mathcal{P}$-complete, a reasonable time to aim for will be $O(n^{\epsilon})$, for any fixed $\epsilon < 1$. In [7], we present an efficient algorithm for RCPP which runs in time $O(n \log n)$ using $\frac{m}{n} \log n$ CREW PRAM processors. This algorithm is based on the sequential algorithm of Paige and Tarjan (whose run time is $O(m \log n)$) [8]. Due to lack of space, we are unable to provide details of this algorithm.

Acknowledgements

We are grateful to Inhye Kang for many stimulating discussions. We are also grateful to Angela Lai and D.R. Mani for their wonderful help in implementing our algorithm.

References

[1] C. Alvarez, J.L. Balcazar, J. Gabarro, and M. Santha. Parallel Complexity in the Design and Analysis of Concurrent Systems. In *PARLE '91. Parallel Architectures and Languages Europe, Vol 1.* 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*, pages 29–47, 1991.

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

[4] R. Cole. Parallel Merge Sort. *SIAM Journal on Computing*, 17:770–785, 1988.

[5] J. Já Já. *Parallel Algorithms: Design and Analysis.* Addison-Wesley Publishers, 1992.

[6] P.C. Kanellakis and S.A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[7] I. Lee and S. Rajasekaran. Parallel Algorithms for Relational Coarsest Partition Problems. Technical Report MS-CIS-93-71, Dept. of CIS, Univ. of Pennsylvania, July 1993.

[8] R. Paige and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[9] S. Zhang and S.A. Smolka. Towards efficient parallelization of equivalence checking algorithms. Unpublished Manuscript, 1993.