Retrieval of scattered information by EREW, CREW and CRCW PRAMs

Faith Fich*

University of Toronto, Toronto, Ontario, Canada M5S 1A4
fich@cs.toronto.edu

Mirosław Kowaluk† and Krzysztof Loryś‡

Universität Würzburg, D-W-8700-Würzburg, Germany,
kowaluk@informatik.uni-wuerzburg.dbp.de,
lorys@informatik.uni-wuerzburg.dbp.de

Mirosław Kutyłowski§

FB Informatik and Heinz-Nixdorf-Institut, Universität–GH Paderborn,
D-W-4790 Paderborn, Germany, mirekk@uni-paderborn.de

Prabhakar Ragde ¶

University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
plragde@plg.uwaterloo.ca

Abstract. The k-compaction problem arises when k out of n cells in an array are non-empty and the contents of these cells must be moved to the first k locations in the array. Parallel algorithms for k-compaction have obvious applications in processor allocation and load balancing; k-compaction is also an important subroutine in many recently developed parallel algorithms. We show that any EREW PRAM that solves the k-compaction problem requires $\Omega(\sqrt{\log n})$ time, even if the number of processors is arbitrarily large and $k = 2$. On the CREW PRAM, we show that every n-processor algorithm for k-compaction problem requires $\Omega(\log\log n)$ time, even if $k = 2$. Finally, we show that $O(\log k)$ time can be achieved on the ROBUST PRAM, a very weak CRCW PRAM model.

*supported by NSERC and the Information Technology Research Centre of Ontario

†supported by Deutsche Forschungsgemeinschaft under grant Wa 549/1; present address: Uniwersytet Warszawski, ul. Banacha 2, PL-02-097 Warszawa, Poland

‡supported by the Alexander von Humboldt-Stiftung; present address: Uniwersytet Wrocławski, ul. Przesmyckiego 20, PL-51-151, Wrocław, Poland

§supported by Deutsche Forschungsgemeinschaft under grant ME 872/1–4; present address: Uniwersytet Wrocławski, ul. Przesmyckiego 20, PL-51-151, Wrocław, Poland

¶supported by NSERC and the Information Technology Research Centre of Ontario

1 Introduction

The *k-compaction problem* is defined as follows: given values in array $A[1..n]$, of which at most k are nonzero, rearrange the values so that the nonzero values occupy the initial fragment of the array. This problem has been considered in many contexts; for instance, this is a natural subproblem to consider in the context of processor or task reallocation. Gil and Rudolph [11] gave a deterministic algorithm to solve this problem in time $O(\log k)$ on a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) using n processors. Rudolph and Steiger [21] gave a probabilistic algorithm running in time $O(\log k)$ (with high probability) using k processors. Hagerup and Nowak [16] gave a probabilistic algorithm running in time $O(\log k/\log\log k)$ using n processors. All of these algorithms use an ARBITRARY CRCW PRAM, with the property that when several processors simultaneously write into the same cell, an arbitrary processor succeeds. (The terminology is from [8]).

A weaker concurrent-read concurrent-write model is the COMMON PRAM, in which simultaneous writes must be of the same value. Ragde [20] gave a k-compaction algorithm and matching lower bound of $\Theta(\log k/\log\log n)$ for the COMMON PRAM using n processors. The immediate utility of this result was in the development of very fast randomized CRCW PRAM algorithms. Intuitively, if many independent instances of problems are worked on simultaneously by a fast algorithm with a low probability of failure, the failures can be gathered up by compaction and dealt with quickly. Papers such as [1, 9, 10, 12, 14, 15, 18] used compaction or approximate compaction as an essential subroutine in algorithms for space allocation, estimation, sorting, PRAM simulation, generation of random permutations, and computational geometry.

In this paper, we look at the problem of k-compaction on weaker models of computation. Hagerup and Nowak [16] considered k-compaction on the concurrent-read exclusive-write (CREW) PRAM. They gave an algorithm running in $O(k)$ time with n^k processors and gave another running in $O(k^2 \log\log n)$ time using n processors. In Section 2, we show that the latter algorithm is optimal for small k by demonstrating an $\Omega(\log\log n)$ lower bound for n processors, even when $k = 2$.

We consider the exclusive-read exclusive-write (EREW) PRAM in Section 3 and show that, for this model, the k-compaction problem requires $\Omega(\sqrt{\log n})$ time, even when $k = 2$. There are very few lower bounds on the EREW PRAM for problems defined on Boolean inputs: Beame, Kik and Kutyłowski [2] show that the problem of broadcasting one value to n processors requires $\Theta(\log n)$ time. Snir's lower bound on searching a sorted table on an EREW PRAM [22] requires a large input domain due to the use of Ramsey theory.

The ROBUST PRAM is a very weak CRCW PRAM [17], in which the result of a simultaneous write might be any value. An algorithm for this model must be correct even if an adversary decides the outcome of every simultaneous write. Every other CRCW PRAM that has been studied can compute the logical OR of n bits in constant time. However, it is not known how to do OR in less than $O(\log n)$ time on the ROBUST PRAM and with $n^{O(1)}$ processors, it requires $\Omega(\sqrt{\log n})$ time [7]. In Section 4 of this paper, we demonstrate an algorithm for k-compaction running in $O(\log k)$ time on the ROBUST PRAM using n processors.

Combined, these results show that an important problem in the context of parallel computation has different complexity depending on whether concurrent reads and concurrent writes are allowed by the model.

2 The bounds for CREW PRAMs

Hagerup and Nowak [16] show that k-compaction of an array of size n can be performed by a CREW PRAM with n^k processors in $O(k)$ time. For $k = 2$, this yields a constant time algorithm with n^2 processors. They also show that, for $n \leq p \leq n^2$, the 2-compaction problem can be solved by a p-processor CREW PRAM in $O(\log\log n - \log\log(p/n))$ time. We will prove a matching lower bound.

Let the k-OR problem be the problem of computing the logical OR of n Boolean variables at most k of which have value one. Note that the k-OR problem can be reduced to k-compaction. Indeed, if we solve the k-compaction problem given a Boolean input that contains at most k ones, then it suffices to inspect the first output cell to determine the value of the OR. Hence any lower bound for computing 2-OR is also a lower bound for the 2-compaction problem. Furthermore, since 2-compaction is a subproblem of k-compaction for $k > 2$, any such lower bound also applies to these problems.

Consider any PRAM and suppose its input variables are $x_1, \ldots, x_n$. For $V \subseteq \{x_1, \ldots, x_n\}$, let I_V denote the inputs in which at most two of the variables in V have value 1 and the rest of the variables have value 0. We say that a processor *depends on at most one variable from the set V at time t* if, for the inputs in I_V, the state of the processor at time t is either fixed or can be expressed as a function of the value of one variable $x_i \in V$. In other words, for each value of x_i, changing the values of the other variables in V does not change the processor's state. Similarly, a memory cell *depends on at most one variable from V at time t* if its contents at time t are either fixed or can be expressed as a function of the value of one variable in V. In particular, if a processor's state or a memory cell's contents can be expressed as a nonconstant function of $x_i \in V$ for all the inputs in I_V, we say that the processor or memory cell depends on x_i.

Lemma 2.1 *Suppose that at the end of its computation, each memory cell of a PRAM depends on at most one variable from a set $V \subseteq \{x_1, \ldots, x_n\}$. If $|V| \geq 2$, then the PRAM does not compute 2-OR.*

Proof. At the end of the computation, the output cell depends on at most one variable in V. Since $|V| \geq 2$, there is a variable $x_i \in V$ that the answer cell does not depend on. Set all other variables to 0. Then the value in the answer cell is fixed, but the OR of the input bits is either 0 or 1, depending on the value of x_i. This means that there is an input in I_V on which the PRAM does not compute OR. □

Theorem 2.2 *A CREW PRAM with $p \geq n$ processors requires more than* $(\log\log n - \log\log(5p/n))/2$ *steps to solve the 2-OR problem.*

Proof. Consider a p-processor CREW PRAM that computes the OR of the Boolean variables $x_1, \ldots, x_n$, at most two of which have value 1. We prove, by induction on t, that there is a set of variables V_t such that $|V_t| \geq n^{2^{2t}}/(5p)^{2^{2t}-1}$ and, at the end of step t, each processor and each memory cell depends on at most one variable from V_t.

This condition is true at the beginning of the computation, since the state of every processor is fixed and the contents of each memory cell depends on at most one variable from $V_0 = \{x_1, \ldots, x_n\}$. Therefore, assume that $t \geq 0$ and that the claim is true at the end of step t.

Consider the situation just before the read phase in step $t+1$. Suppose all variables not in V_t have been set to 0. Let $G_{t+1} = (V_t, E)$ be the undirected graph with $\{x_i, x_j\} \in E$ if there exists a processor P whose state depends on x_i and a cell C whose contents depends on x_j such that P reads from C for some value of x_i.

Note that $|E| \leq 2p$. Turan's theorem [3, page 282] says that any graph $G = (V, E)$ has an independent set of size at least $|V|^2/(|V| + 2|E|)$. In particular, there is an independent set V'_{t+1} in G_{t+1} such that

$$|V'_{t+1}| \geq |V_t|^2/(|V_t| + 4p) \geq |V_t|^2/5p \geq \left(n^{2^{2t}}/(5p)^{2^{2t}-1}\right)^2/5p \geq n^{2^{2t+1}}/(5p)^{2^{2t+1}-1}.$$

Since the read phase does not cause the contents of memory cells to change and $V'_{t+1} \subseteq V_t$, each memory cell depends on at most one variable from V'_{t+1}. Furthermore, each processor depends on at most one variable from V'_{t+1}. To see this, suppose P depends on x_i and P reads from C and C' when $x_i = 0$ and $x_i = 1$, respectively. Also suppose that C and C' depend only on the variables x_j and x_k, respectively. If $x_i \notin V'_{t+1}$, then x_i is set to 0 and P depends only on x_j. Otherwise, $x_j, x_k \notin V'_{t+1}$, so x_j, x_k are set to 0 and P depends only on x_i.

Now consider the write phase in step $t+1$. Suppose all variables not in V'_{t+1} have been set to 0. Let $G'_{t+1} = (V'_{t+1}, E')$ be the undirected graph with $\{x_i, x_j\} \in E'$ if there exists a processor P whose state depends on x_i and a memory cell C whose contents depends on x_j such that P writes to C for exactly one value of x_i.

Again $|E'| \leq 2p$ and there is an independent set V_{t+1} in G'_{t+1} such that

$$\begin{aligned} |V_{t+1}| &\geq |V'_{t+1}|^2/(|V'_{t+1}| + 4p) \geq |V'_{t+1}|^2/5p \\ &\geq \left(n^{2^{2t+1}}/(5p)^{2^{2t+1}-1}\right)^2/5p \geq n^{2^{2(t+1)}}/(5p)^{2^{2(t+1)}-1}. \end{aligned}$$

Since the write phase does not cause processors' states to change and $V_{t+1} \subseteq V'_{t+1}$, each processor depends on at most one variable from V_{t+1}.

Let C be a memory cell and suppose there are two processors P and P' that depend on different variables x_i and x_j in V_{t+1} and such that P writes to C only when $x_i = a$ and P' writes to C only when $x_j = a'$. Setting $x_i = a$ and $x_j = a'$ causes a write conflict; therefore this situation cannot arise. Thus, which processor (if any) writes to C during step $t+1$ depends on the value of at most one variable.

Suppose that the contents of C at the end of step t depend on the variable x_i and which processor writes to C during step $t+1$ depends on the value of another variable x_j. If $x_i \notin V'_{t+1}$, then x_i is set to 0 and C depends only on x_j. Otherwise, $x_j \notin V'_{t+1}$, so x_j is set to 0. Then the contents of C at the end of step $t+1$ is either fixed or depends only on x_i, depending on whether or not C is written to when $x_j = 0$. Thus at the end of step $t+1$, each memory cell depends on at most one variable from V_{t+1}.

It follows by induction that, after the last step, T, of the computation, there is a set $V_T \subseteq \{x_1, \ldots, x_n\}$ such that $|V_T| \geq n^{2^{2T}}/(5p)^{2^{2T}-1}$ and every memory cell depends on at most one variable from V_T. Furthermore, it follows from Lemma 2.1 that $|V_T| \leq 1$. Therefore $T > (\log\log n - \log\log(5p/n))/2$; otherwise, $(5p/n)^{2^{2T}-1} < n$, which implies that $|V_T| > 1$. □

Corollary 2.3 *For $k \geq 2$, a CREW PRAM with $p \geq n$ processors requires more than $(\log\log n - \log\log(5p/n))/2$ steps to solve the k-compaction problem.*

3 The bounds for EREW PRAMs

In this section, we prove that an EREW PRAM with arbitrarily many processors requires $\Omega(\sqrt{\log n})$ steps to solve the k-compaction problem for any $k \geq 2$. We also present a new algorithm for 2-compaction that runs in approximately $0.36 \log n$ steps.

The lower bound uses an approach that is similar to the one used to obtain the lower bound for CREW PRAMs.

Theorem 3.1 *An EREW PRAM requires more than $\sqrt{\log n/2} - 2$ steps to solve the 2-OR problem.*

Proof. Consider an EREW PRAM that solves the 2-OR problem. We prove, by induction on t, that at the end of step t, there is a set of variables $V_t \subseteq \{x_1, \ldots, x_n\}$ such that $|V_t| \geq n/15^t \cdot 4^{t(t-1)}$ and each processor and each memory cell depends on at most one variable from V_t. This condition holds at the beginning of the computation, with $V_0 = \{x_1, \ldots, x_n\}$.

We assume that the property holds for t. In particular, at the end of step t, each processor and memory cell depends on at most one variable from V_t.

Beame, Kik, and Kutyłowski proved that in any EREW PRAM, for any two Boolean inputs that differ on one bit, there are fewer than 4^t processors and 4^t memory cells that can distinguish between them after t steps [2]. Let $z \in I_{V_t}$ denote the input in which all variables are 0 and, for each $x_i \in V_t$, let $z^{\{i\}} \in I_{V_t}$ denote the input in which $x_i = 1$, but all other variables are 0. Then, there are fewer than 4^t processors and 4^t memory cells that distinguish between z and $z^{\{i\}}$ after t steps of the computation. Hence, for each $x_i \in V_t$, there are fewer than 4^t processors and 4^t memory cells that depend on x_i at the end of step t.

Let $G_{t+1} = (V_t, E)$ be the directed graph with $(x_i, x_j) \in E$ if, at the beginning of step $t+1$, there exists a processor P that depends on x_i and a memory cell C that depends on x_j and, for some input in I_{V_t}, P reads from C during step $t+1$.

Two processors that depend on different variables cannot both read from the same cell during step $t+1$. Otherwise, we can set the values of those two variables so that a write conflict arises. Since there are fewer than 4^t cells that depend on each $x_j \in V_t$, the indegree of each node is less than 4^t.

The outdegree of any node $x_i \in V_t$ is less than $2 \cdot 4^t$. This is because there are fewer than 4^t processors that depend on x_i. Each such processor is in one state for those inputs in I_{V_t} in which $x_i = 0$ and in another state for those inputs in I_{V_t} in which $x_i = 1$. Hence each can read from at most 2 different cells during step $t+1$.

Zarankiewicz's theorem [3, page 284] says that a graph $G = (V, E)$ of degree at most k has an independent set of cardinality at least $|V|/(k+1)$. Since G_{t+1} has degree less than $3 \cdot 4^t$, it has an independent set $V'_{t+1} \subseteq V_t$ such that $|V'_{t+1}| \geq |V_t|/(3 \cdot 4^t)$. As in the proof of Theorem 2.2, it is easy to prove that after the read phase of step $t+1$, each processor and each cell depends on at most one variable from V'_{t+1}.

Now consider the write phase of step $t+1$. Let $G'_{t+1} = (V'_{t+1}, E')$ be the directed graph with $(x_i, x_j) \in E'$ if, immediately before the write phase of step $t+1$, there exists a processor P that depends on x_i and a memory cell C that depends on x_j such that P writes into C for exactly one value of x_i.

As in the proof of Theorem 2.2, if two different processors may write into the same cell at step $t+1$ for inputs in $I_{V'_{t+1}}$, then they depend on the same variable in V'_{t+1}. There are less than 4^t cells that depend on each variable in V'_{t+1} after step t. Thus the indegree of each node is less than 4^t.

To show that the outdegree of each node is less than $4 \cdot 4^t$, consider any $x_i \in V'_{t+1}$. There are less than 4^t processors that depend on x_i at the beginning of step $t+1$. A processor may also depend on x_i by reading a cell during step $t+1$ that depends on x_i. For inputs in $I_{V'_{t+1}}$, no processor that depends on $x_j \in V'_{t+1}$ can read a cell that depends on x_i at step $t+1$. Hence a cell that depends on x_i can either be read by a fixed processor or not be read at all. Since there are less than 4^t such cells, there are less than $2 \cdot 4^t$ processors that depend on x_i after the read phase of step $t+1$. Each of these processors have two possible states they can be in. Thus there are less than $2 \cdot 2 \cdot 4^t$ cells that are written to during step $t+1$ by any of these processors for any input in $I_{V'_{t+1}}$. This implies that the outdegree of x_i is less than $4 \cdot 4^t$.

Since G'_{t+1} has degree less than $5 \cdot 4^t$, it has an independent set $V_{t+1} \subseteq V'_{t+1}$ of cardinality at least $|V'_{t+1}|/(5 \cdot 4^t)$. As in the proof of Theorem 2.2, one can see that after the write phase of step $t+1$ each processor and each cell depends on at most one variable from V_{t+1}. Also,

$$|V_{t+1}| \geq \frac{|V'_{t+1}|}{5 \cdot 4^t} \geq \frac{|V_t|}{3 \cdot 4^t \cdot 5 \cdot 4^t} \geq \frac{n/(15^t \cdot 4^{t(t-1)})}{15 \cdot 4^{2t}} = \frac{n}{15^{t+1} \cdot 4^{(t+1) \cdot t}}.$$

Since $|V_t| \geq n/15^t \cdot 4^{t(t-1)}$, it follows that $|V_t| \geq 2$, for $t \leq \sqrt{\log n/2} - 2$. Hence, by Lemma 2.1, the computation requires more than $t \leq \sqrt{\log n/2} - 2$ steps. □

Corollary 3.2 *For $k \geq 2$, a EREW PRAM with $p \geq n$ processors requires more than $\sqrt{\log n/2} - 2$ steps to solve the k-compaction problem.*

Next, we turn attention to upper bounds.

Lemma 3.3 *The k-compaction problem can be solved by an EREW PRAM with n processors in $\log n + O(1)$ steps.*

The idea of this algorithm is to replace each nonzero value by 1 and then compute the prefix sum to determine the number of nonzero values in each initial segment of the input. Using this information, the original nonzero elements can be moved (in order) to the beginning of the array.

We conjecture that, even for $k = 2$, it is not possible to improve k-compaction by more than a constant factor on an EREW PRAM. However, the following algorithm does solve 2-compaction almost a factor of 3 faster.

Algorithm 3.4 *The 2-compaction problem can be solved on an EREW PRAM using n processors in $0.5 \log_b n + 0(1) \approx 0.36 \log n$ steps, where $b = (3+\sqrt{5})/2$.*

Proof. Partition the input into $\sqrt{n}$ groups of $\sqrt{n}$ input variables and associate $\sqrt{n}$ processors with each group. In parallel, the processors $P_{i,1}, \ldots, P_{i,\sqrt{n}}$ associated with each group $C_{i,1}, \ldots, C_{i,\sqrt{n}}$ will determine whether there are 0, 1 or 2 nonzero values in their group, using the subroutine described below. Furthermore, if the group contains a single nonzero value, then all the variables in the group are assigned that value. If the variables in the group are all initially zero, they do not change value.

¿From this point, the solution to the 2-compaction problem can be obtained in at most 2 more steps, as follows. Processors in groups that are entirely zero do nothing. In particular, if the input was initially all zero, the output will be all zero.

If there is a group that originally had 2 nonzero values, 2 processors in the group write these values to the first 2 answer cells. Note that, in this case, all other groups are entirely zero, so no write conflicts will arise.

Now suppose the ith group originally had 1 nonzero value. Then each processor $P_{i,j}$, $j \neq i$, reads $C_{j,i}$, the ith variable in group j. Since there is at most one other group that contains a nonzero value, at most one of these processors can read a nonzero value. If $C_{j,i}$ is nonzero, then processor $P_{i,j}$ writes 0 into $C_{i,1}$ and during the second step, writes the nonzero value originally in the ith group to one of the answer cells. To avoid a conflict with $P_{j,i}$, processor $P_{i,j}$ writes its value to the first answer cell if $i < j$ and to the second answer cell if $i > j$. During the second step, $P_{i,1}$ reads $C_{i,1}$. If it is nonzero, then there was no other group with a nonzero value and $P_{i,1}$ writes this value to the first answer cell.

It remains to describe the subroutine. To be concrete, let $m = \sqrt{n}$ and suppose that cells $C_{i,1}, \ldots, C_{i,m}$ initially contain the values $a_1, \ldots, a_m$. The subroutine is based on the algorithm of Cook, Dwork and Reischuk [5] for computing the OR of m Boolean variables.

We define a function Last that identifies the last occurrence of a nonzero value in a sequence, if one exists. More precisely,

$$\begin{aligned} \text{Last}([x_1, \ldots, x_v]) = 0 \quad & \text{if} \quad x_1 = \ldots = x_v = 0 \text{ and} \\ \text{Last}([x_1, \ldots, x_v]) = (u, x_u) \quad & \text{if} \quad x_u \neq 0 \text{ and } x_{u+1} = \ldots = x_v = 0. \end{aligned}$$

One can easily see that

$$\text{Last}([x_1, \ldots, x_v, x_{v+1}, \ldots, x_w]) = \begin{cases} \text{Last}([x_{v+1}, \ldots, x_w]), & \text{if Last}([x_{v+1}, \ldots, x_w]) \neq 0, \\ \text{Last}([x_1, \ldots, x_v]), & \text{otherwise.} \end{cases}$$

For $t \geq 0$, let $c(t) = F_{2t+1}$ and $p(t) = F_{2t}$, where F_i is the ith Fibonacci number. Then $c(0) = 1$, $p(0) = 0$, $p(t+1) = p(t) + c(t)$, and $c(t+1) = c(t) + p(t+1)$. Inductively, we will ensure that immediately after step t, each cell $C_{i,j}$ stores the value of $\text{Last}([a_j, \ldots, a_{j+c(t)-1}])$, and each processor $P_{i,j}$ knows the value of Last($[a_j, \ldots, a_{j+p(t)-1}]$), Here a_k, $k > m$, denotes the value a_j such that $j = k \bmod m$ and $1 \leq j \leq m$.

When $t = 0$, these conditions are true. During step $t+1$, processor $P_{i,j}$ reads from the cell $C_{i,j'}$ where $j' = j + p(t) \bmod m$. Since $C_{i,j'}$ stores the value of Last($[a_{j'}, \ldots, a_{j'+c(t)-1}]$) and $j' + c(t) - 1 = j + p(t+1) - 1 \bmod m$, $P_{i,j}$ can easily compute Last($[a_j, \ldots, a_{j+p(t+1)-1}]$). During step $t+1$, if Last($[a_j, \ldots, a_{j+p(t+1)-1}]$) is different from 0, processor $P_{i,j}$ writes this value to cell $C_{i,j''}$, where $j'' = j - c(t) \bmod m$. Otherwise, $P_{i,j}$ does not write. Note that $j+p(t+1)-1 = j''+c(t+1)-1 \bmod m$ so that, after step $t+1$, $C_{i,j''}$ contains the value Last($[a_{j''}, \ldots, a_{j''+c(t+1)-1}]$), whether or not $P_{i,j}$ wrote.

If $T \geq \log_b m + 1.34$, $F_{2T+1} \geq m$. (See, for example, [6].) Thus, after T steps, each cell $C_{i,j}$ contains (k, a_k), where a_k is the last element in the sequence $[a_j, a_{j+1}, \ldots, a_m, a_1, \ldots, a_{j-1}]$ that is different from 0, if such an element exists. Otherwise, $C_{i,j}$ contains the value 0 for every $j \in \{1, \ldots, m\}$. Note that if both a_k and $a_{k'}$ are nonzero, where $1 \leq k < k' \leq m$, then $C_{i,k} = (k', a_{k'})$ and $C_{i,k'} = (k, a_k)$. In this case, processors $P_{i,k}$ and $P_{i,k'}$ can write the values a_k and $a_{k'}$ to the first and the second answer cells, respectively. If a_k is the only nonzero element among $a_1, \ldots, a_m$, then $C_{i,j} = (k, a_k)$ for all $j \in \{1, \ldots, m\}$. In particular, $C_{i,k} = (k, a_k)$. Using these observations, it is easy for the processors in the ith group to distinguish between these two cases. □

4 ROBUST PRAM algorithms for the compaction problem

Since the ROBUST PRAM cannot compute OR in less than $\Omega(\log n)$ time using a polynomial number of processors, it is difficult to design algorithms for k-compaction that work in $O(\log k)$ time if k is unknown. The algorithm we present below will have all k items in the first k cells after $O(\log k)$ steps, though the algorithm may not terminate for $O(\log n)$ steps. If k is known, the algorithm will terminate in $O(\log k)$ steps.

We begin by describing a simple n-processor ROBUST PRAM that solves the 2-compaction problem in $O(1)$ steps. Suppose the input is located in cells $C_{i,j}$, $i, j \in \{1, \ldots, \sqrt{n}\}$. Each cell $C_{i,j}$ is read by one processor. If the cell stores a nonzero value, the processor attempts to write that value into cells R_i and S_j.

Note that if the input is entirely 0, then $R_1, R_2, \ldots, R_{\sqrt{n}}$ and $S_1, S_2, \ldots, S_{\sqrt{n}}$ will all contain their initial value, 0. If the input contains exactly one nonzero value then that value will occur exactly once among $R_1, R_2, \ldots, R_{\sqrt{n}}$ and exactly once among $S_1, S_2, \ldots, S_{\sqrt{n}}$. Now suppose that there are two input cells $C_{i,j}$ and $C_{i',j'}$ that contain nonzero values. Then either $i \neq i'$ or $j \neq j'$, so either $R_1, R_2, \ldots, R_{\sqrt{n}}$ or $S_1, S_2, \ldots, S_{\sqrt{n}}$ will contain both these values.

Next, the 2-compaction problems for $R_1, R_2, \ldots, R_{\sqrt{n}}$ and $S_1, S_2, \ldots, S_{\sqrt{n}}$ are each solved by the $O(1)$ time CREW PRAM algorithm that uses $(\sqrt{n})^2 = n$ processors [17]. A constant number of additional steps suffice to combine the solutions to these two problems into the solution of the original problem.

With sufficiently many processors, a ROBUST PRAM can perform k-compaction in $O(\log k)$ time. The idea is to assign one processor to each subset of at most k input variables. In $O(\log k)$ steps, the processors determine which of these sets contain only nonzero values. Call these sets *full*. Every processor whose set is full writes its processor number to a cell associated with its set's size. There is only one full set of maximum size, so a unique processor writes to the corresponding cell. This cell can be identified in $O(\log k)$ steps, since all cells corresponding to larger sets are not written to. Once the full set of maximum size is identified, it is easy to produce the output for the k-compaction problem.

Lemma 4.1 *[13]. The k-compaction problem can be solved in $O(\log k)$ time by a ROBUST PRAM with $\sum_{i=1}^{k}\binom{n}{k}$ processors.*

Using this algorithm as a subroutine, we are able to obtain an algorithm that uses significantly fewer processors.

Theorem 4.2 *The k-compaction problem of size n can be solved in $O(\log k)$ time by an n-processor ROBUST PRAM.*

Proof. If $k \geq n^{1/5}$, the result follows from Lemma 3.3. Therefore, we assume that $k \leq n^{1/5}$.

First, the input cells are partitioned into groups of l cells, where

$$l = \begin{cases} 2k(k-1) & \text{if} \quad k \leq \frac{\log n}{4 \log\log n}, \\ \frac{(k-1)\log n}{3\log\log n - 1} & \text{if} \quad \frac{\log n}{4\log\log n} < k \leq \log n, \text{ and} \\ \frac{(k-1)\log n}{3\log k - 1} & \text{if} \quad \log n \leq k \leq n^{1/5}. \end{cases}$$

We apply the EREW PRAM algorithm from Lemma 3.3 to solve the k-compaction problem within each group in $O(\log l) = O(\log k)$ steps.

Let $y_j = j$ if the jth group contains a cell with a nonzero value and let $y_j = 0$ otherwise. Given a solution of the k-compaction problem for the values $y_1, \ldots, y_{n/l}$, replace each nonzero value j in the solution by the number of nonzero values in the jth group. Then compute the prefix sum of these numbers in $O(\log k)$ steps and use this information to move the nonzero values in each group to their appropriate places in the answer. Thus it suffices to solve this smaller instance of k-compaction.

There are more than $0.6q/\ln q$ prime numbers in the interval $[q, 2q]$ for $q \geq 20.5$ (see [19]). The number l has been chosen so that $0.6n^{(k-1)/l}/\ln\left(n^{(k-1)/l}\right) \geq l$, for n sufficiently large. Therefore there are at least l prime numbers between $n^{(k-1)/l}$ and $2n^{(k-1)/l}$. Let $p_1, \ldots, p_l$ be distinct prime numbers in this range.

Claim *If $a_1, \ldots, a_k \in [1, n]$ are distinct integers, then for each a_j, there is some prime p_i such that*

$$a_r \neq a_j \mod p_i \quad \textit{for all } r \neq j.$$

To prove the claim, consider any number $j \leq k$. Let $Z_{j,r} = \{p_i : i \leq l \text{ and } a_j = a_r \mod p_i\}$. It suffices to show that there exists $p_i \notin \bigcup_{r \neq j} Z_{j,r}$. Since $|a_j - a_r| < n$, there are less than $l/(k-1)$ indices i such that $p_i \mid (a_j - a_r)$; otherwise $|a_j - a_r| \geq (n^{(k-1)/l})^{l/(k-1)} = n$. Hence $|\bigcup_{r \neq j} Z_{j,r}| < (k-1) \cdot l/(k-1) = l$ and there exists $p_i \notin \bigcup_{r \neq j} Z_{j,r}$.

For each prime p_i, our algorithm uses p_i auxiliary cells $S_i(0), \ldots, S_i(p_i - 1)$. In parallel, for each $y_j \neq 0$ and each prime p_i, one processor attempts to write the value j into cell $S_i(j \bmod p_i)$. Of course, write conflicts may occur producing unpredictable results. However, by the claim, for each $y_j \neq 0$, there is some prime p_i such that exactly one processor writes into $S_i(j \mod p_i)$.

To eliminate garbage caused by write conflicts, we check the correctness of the data stored in these cells. Namely, if a cell $S_i(q)$ contains a nonzero value j, but $y_j = 0$, then 0 is written into $S_i(q)$. Since there are at most $l2n^{(k-1)/l} \in O(n)$ of these cells, there are sufficiently many processors to accomplish this in constant time.

If $k > \log n / 4 \log\log n$, then $l2n^{(k-1)/l} \in k^{O(1)}$. Otherwise, we reduce the number of cells to $lk \in k^{O(1)}$ by performing k-compaction on $S_i(0), \ldots, S_i(p_i - 1)$, for each $i \leq l$. Since $k \leq \log n / 4 \log\log n$ implies that $l(2n^{(k-1)/l})^k \in O(n)$, these l different k-compaction problems can be solved simultaneously in $O(\log k)$ steps using the algorithm of Lemma 4.1.

Next, all but one copy of each nonzero value in these $k^{O(1)}$ cells are set to 0. This can be done in $O(\log k)$ steps using Cole's CREW PRAM sorting algorithm [4]. Since the number of cells is also in $O(n)$, there is a sufficient number of processors available.

Finally, the k-compaction algorithm of Lemma 3.3, is applied to these cells to obtain the solution of the k-compaction problem for the values $y_1, \ldots, y_{n/l}$, □

Note that the algorithm presented has the property that during a computation no memory cell has to store a word that is longer than $\max\{\lceil \log n \rceil + 1, w\}$, where w is the maximal length of the words being the initial contents of the input cells.

When the number of cells with nonzero values is small, but an upper bound on this number is not known, it is still possible to produce the answer quickly.

Theorem 4.3 *A ROBUST PRAM with n processors can solve the n-compaction problem so that, if the input contains exactly k cells with nonzero values, then, after $O(\log k)$ steps, these values are in the first k output cells and none of the output cells are subsequently modified.*

Proof. It suffices to solve the compaction problem when, for each i, the ith input variable has value either 0 or i. This is because, when a processor writes the value i to an output cell, it could write the initial value of the ith input cell instead.

The k-compaction algorithm in Theorem 4.2 can also be applied when there are more then k cells with nonzero values. In this case, the algorithm yields some result, but not all of the nonzero values can be contained in the first k output cells. However, even in this case, each output value is correct and cannot occur more than once in the output cells.

Our algorithm works by repeatedly finding nonzero values in the input, appending them to the output, and removing them from the input (by setting them to 0). The k-compaction algorithm in Theorem 4.2 is applied for successively larger values of k. Specifically, the jth stage performs k_j-compaction, where $k_1 = 2$ and $k_{j+1} = k_j^2$, for $j \geq 1$. In total, $1 + \lceil \log\log n \rceil$ stages are performed, and, in particular, the last stage performs n-compaction. During stage $j+1$, if there are more than k_{j+1} nonzero values in the input, not all of them (or, perhaps, none of them) are detected by this procedure, but the values that are output are all correct.

Now suppose the input initially contains k nonzero values, where $k_{j-1} < k \leq k_j$. Since the number of nonzero values in the input only decreases as the algorithm proceeds, there are no more than k nonzero values in the input at the beginning of stage j. Then, during stage j, all remaining nonzero values are identified. During later stages, no nonzero values remain in the input, so the contents of the output cells will not be modified. The number of steps performed during the stages 1 through j is $O(\log(k_1)+\cdots+\log(k_j)) = O(\log(k_j^{1/2^{j-1}})+\cdots+\log(k_j) = O(\log k_j) = O(\log k)$. $\square$

For the inputs that contain at most k cells with nonzero values, the PRAM described in Theorem 4.3 produces the correct output within $O(\log k)$ steps, but does not halt until $\Omega(\log n)$ steps have been performed. The problem is that, without a bound on the number of cells with nonzero values in the input, it is not known whether any additional output cells will be written into. For example, when the input contains many cells with nonzero values, it is possible that, due to write conflicts, all the output cells may remain 0 for a long time. This behavior is not the fault of our algorithm, but rather is inherent in any solution, due to the fact that any n-processor ROBUST PRAM requires $\Omega(\sqrt{\log n})$ steps to compute the OR of n Boolean variables [7].

References

[1] H. Bast and T. Hagerup: *Fast and Reliable Parallel Hashing.* In: Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, 1991, pp. 50–61.

[2] P. Beame, M. Kik and M. Kutyłowski: *Information broadcasting by Exclusive Read PRAMs.* Manuscript.

[3] C. Berge: *Graphs and Hypergraphs.* North–Holland, Amsterdam, 1976.

[4] R. Cole: *Parallel Merge Sort.* SIAM J. Comput. 17, 1988, pp. 770–785.

[5] S. Cook, C. Dwork and R. Reischuk: *Upper and Lower Time Bounds for Parallel Random Access Machines Without Simultaneous Writes.* SIAM J. Comput. 15, 1986, pp. 87–97.

[6] M. Dietzfelbinger, M. Kutyłowski and R. Reischuk: *Exact Time Bounds for Computing Boolean Functions on PRAMs Without Simultaneous Writes.* To appear in J. Computer and System Sciences.

[7] F. E. Fich, R. Impagliazzo, B. Kapron, V. King and M. Kutyłowski: *Limits on the Power of Parallel Random Access Machines with Weak Forms of Write Conflict Resolution.* Manuscript.

[8] F. E. Fich, P. Ragde, A. Wigderson: *Relations Between Concurrent-Write Models of Parallel Computation.* SIAM J. Comput. 17, 1988, pp. 606–627.

[9] J. Gil and Y. Matias: *Fast Hashing on a PRAM – Designing by Expectation.* In: Proc. 2nd Annual ACM Symposium on Discrete Algorithms, 1991, pp. 271–280.

[10] J. Gil, Y. Matias, and U. Vishkin: *Towards a Theory of Nearly Constant Parallel Time Algorithms.* In: Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 698–710.

[11] J. Gil and L. Rudolph: *Counting and Packing in Parallel.* In: Proc. 1986 International Conference on Parallel Processing, pp. 1000–1002.

[12] M. Goodrich: *Using Approximation Algorithms to Design Parallel Algorithms that may Ignore Processor Allocation.* In: Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 711–722.

[13] T. Hagerup. Personal communication

[14] T. Hagerup: *Fast and Optimal Simulations between CRCW PRAMs.* In: Proc. 9th Symposium on Theoretical Aspects of Computer Science, 1992, pp. 45–56.

[15] T. Hagerup: *The Log-Star Revolution.* In: Proc. 9th Symposium on Theoretical Aspects of Computer Science, 1992, pp. 259–280.

[16] T. Hagerup and M. Nowak: *Parallel retrieval of scattered information.* In: Proc. 16th International Colloquium on Automata, Languages, and Programming, 1989, pp. 439–450.

[17] T. Hagerup and T. Radzik: *Every ROBUST CRCW PRAM can Efficiently Simulate a PRIORITY PRAM.* In: Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures, 1990, pp. 125–135.

[18] Y. Matias and U. Vishkin: *On Parallel Hashing and Integer Sorting.* In: Proc. 18th International Colloquium on Automata, Languages, and Programming, 1991, pp. 729–743.

[19] J. B. Rosser and L. Schoenfeld: *Approximate Formulas for Some Functions of Prime Numbers.* Illinois J. Math. 6, 1962, pp. 64–94.

[20] P. Ragde: *The Parallel Simplicity of Compaction and Chaining.* In: Proc. 17th International Colloquium on Automata, Languages, and Programming, 1990, pp. 744–751.

[21] L. Rudolph and W. Steiger: *Subset Selection in Parallel.* In: Proc. 1985 International Conference on Parallel Processing, pp. 11–14.

[22] M. Snir: *On parallel searching.* SIAM J. Comput. 14, 1985, pp. 688–708.