Optimal Separation of EROW and CROW PRAMs

Navin Goyal[1] Michael Saks[2] S. Venkatesh[3]

Abstract

We consider the problem of evaluating a boolean function on PRAMs. We exhibit a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ that can be evaluated in time $O(\log\log n)$ in a deterministic CROW (Concurrent Read Owner Write) PRAM model, but requires time $\Omega(\log n)$ in EROW (Exclusive Read Owner Write) PRAM. Our lower bound also holds in the randomized Monte Carlo EROW model. This boolean function is derived from the well-known pointer chasing problem, and was first considered by Nisan and Bar-Yossef [10]. Our lower bound improves a special case of the previous result of Nisan and Bar-Yossef, who proved a lower bound of $\Omega(\sqrt{\log n})$ for this function in the deterministic EREW model (and hence in the EROW model). Our result is the first to achieve the best possible separation between the CROW and EROW PRAM models for functions on complete domains (boolean or nonboolean), improving the previous results ([7, 6, 10]).

1 Introduction

In this section, we start with a description of the PRAM model and its variants. We then motivate the problem we study in this paper and describe our results and techniques.

1.1 The Model

A PRAM is an abstract model of a parallel computer. It consists of processors $P_1, P_2, \ldots$ and memory cells $M_1, M_2, \ldots$. Processors communicate with each other using memory cells. Computation proceeds in discrete *time steps* (often we will just say *steps*) in which processors act synchronously. Each time step is made up of three phases executed in that order: (1) Read phase, in which processors read from memory cells. (2) Local computation phase, in which processors do local computation using the data they have learned till that time; they use this computation to decide how to act next. (3) Write phase, in which processors write in memory cells. At the end of the computation, answer should appear in some specified cell, say M_1. We place no limits on the capacities of memory cells and on the computational power of the processors. Such a PRAM model is called the ideal PRAM model. Ideal PRAM models are mainly used for proving lower bounds as these imply similar bounds for any realistic parallel machine. The only reason that a problem cannot be solved instantly in an ideal PRAM model is that initially the input is distributed in different memory cells and is not immediately available to any processor. For lower bound purposes, one usually works with nonuniform algorithm, which only makes the bounds stronger. Bounds in the present paper apply to nonuniform algorithms.

This was a description of a generic PRAM. One needs to specify some *concurrency rules* in order to fully define a model. Some of the most important ones are: *Concurrent Read* (CR), which means that two or more processors can read from a memory cell at the same time step; *Exclusive Read* (ER) means that at most one processor can read from a memory cell in one time step. Similarly, CW means that two or more processors can write into the same memory cell. One needs to further specify what gets written if this happens. (There are many ways in which this can be done; see, e.g., Fich [5], but these distinctions are not relevant to this paper.) EW means that at most one processor can write into the same memory cell in one time step. Finally, in the *Owner Write* (OW) model, introduced by Dymond and Ruzzo [4], each processor P *owns* a unique memory cell $M(P)$, and only the owner of a cell can write into it. In the present paper, we will be concerned with the ideal CROW and ideal EROW models.

PRAMs bring into focus the power of parallelism over sequential computation. The computational bottle-

[1]Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA, email: ngoyal@paul.rutgers.edu. Supported in part by NSF grant CCR-9988526.

[2]Department of Mathematics, Rutgers University, Piscataway, NJ 08854, USA, email: saks@math.rutgers.edu. Supported in part by NSF grant CCR-9988526.

[3]Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrüken, Germany, email: srini@mpi-sb.mpg.de. Supported in part by a joint IAS-DIMACS postdoctoral fellowship.

neck in a PRAM algorithm is the communication between processors. PRAMs have proven to be very useful for designing parallel algorithms and proving lower bounds because of their simplicity and because they hide many low level details. Note that on all the variants of the PRAM model discussed above, any problem whose input is distributed in n cells in the beginning can be solved in $\log n$ steps by collecting all the data in one cell in $\log n$ steps, and then processor P_1 reads all the data, solves the problem in one time step and writes the answer in M_1 ($M(P_1)$ in case of OW models). Since PRAMS are very powerful, lower bounds proven in PRAm models are widely applicable. In particular, it has been observed (see, e.g., [4]) that most CREW algorithms can actually be made to run on CROW without major changes with similar efficiency. Also, as remarked in [6], the owner-write models are more powerful than any realistic parallel machine. Thus proving lower bounds in these models provides us useful information.

1.2 The Problem and Previous Work

An important problem in the study of PRAMs is to understand the relative power of different variants discussed above. A very natural question that arises in PRAM complexity is understanding the power of concurrent read. Is there a complete boolean function that can be evaluated quickly on the CREW PRAM but takes a lot of time in the EREW PRAM model? That is, does read concurrency provide true additional power? This question has been looked at by many researchers.

We start with the definition of a complete boolean function. Let $D_1, D_2, \ldots, D_n$ and D be sets, and let $f : D_1 \times D_2 \times \ldots \times D_n \rightarrow D$ be a function. We call domains of this form (that is, the whole domain is the cartesian product of the domains of individual variables) *complete* domain. Domains which are not complete are called *partial*. If $D = \{0, 1\}$ then we call such a function a decision function. Of special interest are complete boolean decision functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$. As usual, we refer to these as boolean functions.

Snir [11] showed that the problem of searching in a sorted list can be solved in time $O(1)$ on CREW model, but requires $\Omega(\sqrt{\log n})$ on EREW model. But, this result holds only for a function on partial domain. It is argued in [7] why it is desirable to obtain separation on complete domains; they give a complete function which can be evaluated in time $O(\log \log n)$ on CREW PRAM but requires time $\Omega(\sqrt{\log n})$ on EREW PRAM. Both of the above two results make use of Ramsey theory which requires the domain size to be huge. Hence the techniques in these papers do not seem to apply to boolean functions. Researchers have studied the relative power of the EROW and CROW models in computing boolean functions and this question is the focus of this paper. Fich and Wigderson [6] gave a boolean function which can be evaluated in time $O(\log \log n)$ on CROW model but requires $\Omega(\sqrt{\log n})$ expected time in the randomized EROW model. Nisan and Bar-Yossef [10] obtained separation between deterministic CREW and EREW with the same parameters for a different function based on the pointer chasing problem. Note that a separation between CREW and EREW for a boolean function implies the same separation between CROW and EROW since it is known (see [9]) that for the problem of evaluating complete boolean functions CREW and CROW have the same power. Hence separating CREW and EREW is a harder problem. It is known by the results of [3] and [11] that any nontrivial function (a function that depends on all its variables) requires time $\Omega(\log \log n)$ in the CREW model and hence in the CROW model. As mentioned above, any boolean function in n variables can be evaluated in time $O(\log n)$ in the EROW model. After the results in [6] and [10], one is left with the question that whether there are boolean function for which CROW model is as good as possible and the EROW model is as bad as possible. That is, can the results in [6] and [10] be improved to obtain optimal separation between the CROW and EROW models?

1.3 Results

In this paper we answer this question in the affirmative by showing that the function of [10] is in fact such a function.

Theorem 1.1 *There is a family $\{f_n\}$ of boolean functions with $f_n : \{0, 1\}^n \rightarrow \{0, 1\}$ such that f_n can be evaluated in time $O(\log \log n)$ on a CROW PRAM but requires time $\Omega(\log n)$ on deterministic EROW PRAMs or on randomized EROW PRAMs which evaluate f correctly with probability $\frac{1}{2} + \epsilon$ for $\epsilon > 0$.*

Our paper is the first to provide the best separation between the CROW and EROW models (deterministic or randomized) for functions on complete domain (boolean or non-boolean). The boolean function we consider was first considered by Nisan and Bar-Yossef [10]. It is derived from the pointer chasing problem (it was called *pointer jumping* in [10], we adopt a different name to avoid confusion with a related algorithmic technique with the same name), which is a basic and well studied problem in parallel computing: We are given a digraph D on n vertices in which each vertex has outdegree exactly one. For $1 \leq i \leq n$, let us denote the vertex that vertex i points to by $p(i)$, that is, $(i, p(i))$ is an edge in D. In the problem PC_k, the goal is to determine the kth vertex reached by following a path of length k starting from the fixed vertex labeled by 1. That is, in PC_1, we need to determine $p(1)$, in PC_2, we need to determine $p(p(1))$ and so on.

We can also think of this graph as represented by an array x of size n in which $x[i]$ contains the value $p(i)$, for $1 \leq i \leq n$. Let us now consider two simple algorithms that solve this problem. The first algorithm uses n processors running in parallel and employs the well-known "pointer doubling" technique:

> At each time step, for all i, do $p(i) \leftarrow p(p(i))$ in parallel.

It is easy to see that this yields a $O(\log k)$ time algorithm for PC_k. The pointer doubling technique has been used in many parallel algorithms involving linked list traversal. Observe that this algorithm crucially uses the fact that processors can read a memory cell concurrently at any time step. For example, many vertices in the graph could point to the same vertex j and hence, many processors working in parallel will try to access $x[j]$ at the same time step. Therefore, this algorithm can be implemented in the CROW model, but not in the EROW model. Indeed, we have the following result which is also of independent interest.

Theorem 1.2 *There exists a fixed constant* $0 < c < 1$ *such that for all* $k \leq c \log n$*, any EROW PRAM algorithm that solves* PC_k *needs at least* $k/2$ *time steps with probability* $\geq 1 - \frac{2}{\sqrt{n}}$*, where the input* $x \in [n]^n$ *is chosen uniformly at random from* $[n]^n$.

By Yao's Lemma [13], Theorem 1.2 also implies the same lower bound for randomized Monte Carlo EROW algorithms. Theorem 1.2 can be interpreted as saying that, for k as in the Theorem above, on most of the inputs, EROW algorithms are no better than the trivial sequential algorithm. Again, by the discussion above, this Theorem is the best possible up to constant c. It is an improvement in the special case of EROW models of a Theorem of Nisan and Bar-Yossef [10] who proved that deterministic EREW algorithms require $\Omega(k)$ steps for $k \leq c\sqrt{\log n}$ in the worst case.

1.4 Techniques

The basic approach for nearly all lower bounds for non-uniform complexity models is to identify a "progress measure" that tracks the progress of the computation towards its goal. One proves (1) an upper bound on the amount of progress per computation step and (2) a lower bound on the amount of progress that must be made in order to complete the computation.

In the EROW PRAM model, this general approach was used for the proofs of the results of [11, 7, 6, 10] mentioned above (though in some of these papers the progress measure is not made explicit). Though these papers are proving lower bounds on different problems, each shows roughly that as the algorithm progresses one can identify a residual sub-problem similar to the original problem which is "clean" in the sense that information gathered by the processors so far is of essentially no value in solving the subproblem. It is shown that after each step of the algorithm, the size of the new subproblem is not too much smaller than the previous subproblem. In order for the algorithm to finish, this smaller problem must shrink to size 0. Thus the implicit progress measure is the size of the residual subproblem.

The limitation of this approach is that the need for a "clean" subproblem may require considerable shrinking of the problem at each step. In [6], the problem being bounded is defined by a binary tree and the subproblem is obtained by restricting to a rooted subtree, while discarding the rest of the tree. In [10], this paring down process is accomplished by eliminating all of the inputs about which a nontrivial amount of information has been learned.

Our approach does not involve reducing to a subproblem; rather we give a more refined progress measure that enables us to show that small amounts of isolated information cannot accumulate into useful information. As is done in [6], we consider the behavior of the algorithm on uniformly random input. Intuitively, our progress measure is this: at time step t, we measure the progress by the probability that some processor has learned a path of length $2t + 1$. We aim to show that this probability can not increase very much in any single step. However, knowing that this probability is small at time t is not a sufficiently strong induction hypothesis to guarantee that it is still true at time $t + 1$.

Instead we consider the structure of the paths of length 3 that have been learned by various processors. We identify certain *bad events* associated with these paths and show (1) if some processor ever learns a path of length $2t + 1$ within t steps then some bad event must have happened and (2) the probability that a bad event has happened by time t behaves like $c^t \log n/n$ and so remains small for $\Theta(\log n)$ steps. The proof of (2) is rather technical and is based on general principles concerning the spread of knowledge in EROW PRAMS. These principles build upon ones used in previous papers, especially [3, 6, 10].

1.5 Organization of the paper

Most of the rest of the paper is devoted to the proof of Theorem 1.2. We start with some preliminaries in the next section and prove some basic facts about the EROW model. In Section 3, we define a set of "bad events" and state a Lemma about these events which is a crucial ingredient of our lower bound proof. In Section 4, we prove Theorem 1.2 assuming this Lemma. In sections 5 and 6, we prove some results about various notions of knowledge. In Section 7, we use these results to prove the Lemma stated for bad events thus completing the proof of Theorem 1.2. We finish with a proof of Theorem 1.1 in Section 8.

2 Preliminaries

In this section, we start with the formal definition of the pointer chasing problem. We then prove some basic facts about the EROW model that will be used in the lower bound proof. $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Logarithms are to the base 2. In the rest of this paper, we will be concerned with the EROW model.

2.1 Pointer chasing

We are interested in proving a lower bound for the following pointer chasing function in the EROW model.

Definition 2.1 *The pointer chasing function $PC_k : [n]^n \to [n]$ is defined recursively as follows: for any input $x \in [n]^n$, $PC_0(x) = 1$ and for $k > 0$, $PC_k(x) = x[i]$, where $i = PC_{k-1}(x)$.*

An input to the pointer chasing function can be either viewed as an *array* of size n with each entry in $[n]$ or a *graph* on n vertices where each vertex has out-degree exactly one. We will use these two representations interchangeably. The digraph associated with x is denoted by $D(x)$.

Consider an EROW algorithm that solves the pointer chasing problem. On an input x of size n, the algorithm uses processors $P_1, P_2, \ldots$, memory cells $M(P_1), M(P_2), \ldots$. We make some standard assumptions: Initially, input x is distributed in memory cells, specifically $M(P_i)$ contains $x[i]$ for $i \in [n]$, and is empty for $i > n$. At the end of the computation, answer PC_k is written into $M(P_1)$.

2.2 Spread of information

In this section, we study the knowledge a processor has about the input from its communication with other processors. It turns out that it has a particularly simple form for the EROW model. Our notion of knowledge is implicit in previous works; we spell it out in detail because it is used crucially in the proof.

We can abstractly think of a processor's algorithm as a decision tree. Each node of the decision tree represents a possible *state* of the processor during the course of the algorithm. In each step, a processor is in some state. The state tells the processor whether to read some cell in the read phase, if yes, which cell to read, and depending on the outcome, the state for the next step. For a processor P at some time t, two inputs x and y are equivalent if in step t, P is in the same state on both the inputs. Thus, for processor P at time t, the input set $[n]^n$ gets partitioned into equivalence classes. Let us call such a class a *view* of P about the input. Denote by $View_x^t(P)$ the equivalence class in which input x lies. From the processor's point of view, $View_x^t(P)$ captures the knowledge it has about the input, and we call it processor's *knowledge* about the input x at time t. We assume wlog that in the write phase of step t, processor P writes into $M(P)$ all the new information that it learnt during the read phase, i.e., $View_x^t(P)$.

Below we give a characterization of $View_x^t(P)$. A *partial input* is an input in which some of the coordinates are unspecified; we assign value $*$ to such coordinates. Thus a partial input is an element of $([n] \cup \{*\})^n$. Associated with each partial input $a \in ([n] \cup \{*\})^n$ is a unique set $C(a) \subseteq [n]^n$ defined by $C(a) = \{x \in [n]^n : a[i] \neq * \Rightarrow x[i] = a[i]\}$; that is, $C(a)$ contains all inputs agreeing with a at all non-$*$ coordinates. We call sets of this form *subcubes* ($[n]^n$ being the full cube). Clearly the correspondence between a and $C(a)$ is bijective. For given $C = C(a)$, the coordinates of a equal to $*$ are called *floating coordinates* of C (and a), and the remaining coordinates are called *fixed coordinates* of C. The size of the set of fixed coordinates of C is called the *codimension* of C. So if C has codimension d, then $|C| = n^{n-d}$, as we can set each of the $n - d$ floating coordinates in n possible ways. We say that P *knows* a coordinate $x[i]$ by time t if P read $M(P_i)$, or read $M(P')$ at some time $t' \leq t$ where P' knew $x[i]$ at time $t' - 1$. That is, a chain can be traced back from P to $M(P_i)$.

Lemma 2.2 *For any processor P, time t and input x, $View_x^t(P)$ is a subcube. Furthermore, coordinates that P knows by that time on input x are the fixed coordinates of $View_x^t(P)$.*

Proof. By induction on time t. It holds at time 0 because then no processor has any knowledge and so the view of every processor is the whole cube $[n]^n$. Suppose that the view of every processor is of the above form at time t. Let us focus on the view of some processor P

at time $t+1$. Depending solely on this knowledge, either P reads some cell in the read phase of step $t+1$ or it does not. Suppose it reads cell of $M(P')$ then its new view $View_x^{t+1}(P)$ is the intersection of the views of P and P' at time t, and so the set of fixed coordinates of $View_x^{t+1}(P)$ is union of the fixed coordinates of $View_x^t(P)$ and $View_x^t(P')$. Thus the set of fixed coordinates of $View_x^{t+1}(P)$ is the same as the set of coordinates that P has learned by reading from memory cells.

If P does not read anything in step $t+1$ then its view remains the same as at time t. So the induction hypothesis holds at time $t+1$ in both cases. ■

In the sequel, we will use the phrase "a processor P knows a coordinate $x[i]$ at time t on input x" to mean that $x[i]$ is a fixed coordinate of $View_x^t(P)$. The following alternative representation of the knowledge of a processor will be useful later. It will be clear from the context which representation is being used.

Definition 2.3 Knowledge *of a processor P at time t on input x, denoted $K_x^t(P)$, is given by $K_x^t(P) = \{(j,k) \in [n] \times [n] : P$ knows that $x[j] = k\}$.*

That is, knowledge $K_x^t(P)$ is a collection of pairs of the form $(j,k) \in [n] \times [n]$, where (j,k) is included if $x[j] = k$ (i.e., it is an edge in $D(x)$) and P knows $x[j]$ at time t. We say that P *knows* an edge (j,k) at time t on input x, if $(j,k) \in K_x^t(P)$.

We will also need to consider which processors know about a particular coordinate.

Definition 2.4 *For $i \in [n]$, denote by $S_x^t(i)$ the set of processors who know $x[i]$ by the end of time step t. That is, $S_x^t(i) = \{P : (i, x[i]) \in K_x^t(P)\}$.*

The following Lemma says that the number of processors that know a particular input value and the also the number of input values that a processor knows at most doubles with each time step. It is proved easily by induction on t.

Lemma 2.5 *[2, 3] For all times $t \geq 0$,*

1. *$|S_x^t(i)| \leq 2^t$ for all $x \in [n]^n$ and all i, $1 \leq i \leq n$.*
2. *$|K_x^t(P)| \leq 2^t$ for all $x \in [n]^n$ and all processors P.*

2.3 Other useful notions

We will need to consider paths in the input graph $D(x)$. These will be useful to measure the progress made by an EROW algorithm.

Definition 2.6 *In the digraph $D(x)$,* the k-walk *(or the walk of length k) starting at vertex v_1 is the sequence of vertices $(v_1, v_2, \ldots, v_{k+1})$ with $x[v_i] = x[v_{i+1}]$ for $i \in [k]$. The* k-path *starting at vertex v_1 is a k-walk all of whose vertices are pairwise distinct.*

We use the notion of a *line* through a point x in our proof. For any fixed i, $1 \leq i \leq n$, a line through x along direction i consists of all the inputs that may not agree with x along the ith coordinate but are identical to x elsewhere.

Definition 2.7 *A line is a set of points in $[n]^n$ defined with respect to $x \in [n]^n$ and a direction $i \in [n]$, denoted by $x^{(i)}$, is given by $x^{(i)} = \{y \in [n]^n : y[k] = x[k]$ if $k \neq i\}$.*

Definitions of $K_x^t(P)$ and $S_x^t(i)$ can be naturally extended to lines: $K_{x^{(i)}}^t(P) = \cup_{y \in x^{(i)}} K_y^t(P)$ and $S_{x^{(i)}}^t(i) = \cup_{y \in x^{(i)}} S_y^t(i)$.

We call i the floating index of line $x^{(i)}$.

Lemma 2.8
1. *For all x and all i with $1 \leq i \leq n$, we have $|S_{x^{(i)}}^t(i)| \leq 2^t$.*
2. *For $I \subseteq [n]$ with $|I| \geq 2^t + 1$ and $x \in [n]^n$, we have $\cap_{i \in I} S_{x^{(i)}}^t(i) = \phi$.*

Proof. (1) follows from the fact that $S_{x^{(i)}}^t(i) = S_x^t(i)$. To prove this, we proceed by induction. At time 0 the assertion holds since both sets are empty. Now assume that the fact holds up to time t. Consider a processor P who first learns $y[i]$ at time $t+1$ on input $y \in x^{(i)}$. By Lemma 2.2, it follows that the knowledge of P at time t is the same for inputs y and x. Hence, in the next round, P will read the same cell on input x as it does on input y, and that cell contains $x[i]$ by induction hypothesis.

For (2), assume it does not hold. Then let, for some P, $P \in \cap_{i \in I} S_{x^{(i)}}^t(i)$. As mentioned above, $\cap_{i \in I} S_{x^{(i)}}^t(i) = \cap_{i \in I} S_x^t(i)$. Therefore, if $P \in \cap_{i \in I} S_{x^{(i)}}^t(i)$, then $|K_x^t(P)| > 2^t$, which is a contradiction. ■

Let us call a processor P *active* on input x at time t if $View_x^t(P) \subsetneq [n]^n$. In other words, we say that a processor is active if it has nontrivial information about the input at a given time. Denote by $A^t(x)$ the set of active processors at the end of step t on input x. Lemma 2.5 easily implies that $|A^t(x)| \leq 2^t n$ for any input $x \in [n]^n$. Until processors become active, they are oblivious to the input, since in the EROW model, no processor can tell another processor any information, so the set of processors who become active in a certain time step is the same for all inputs. Hence $A^t(x)$ is the same set for all x. Let us denote it by A^t. Thus we have the following Lemma.

Lemma 2.9 *For any time t, we have $|A^t| \leq 2^t n$.*

3 Bad events

We analyze the progress of the algorithm when the input x is chosen uniformly at random from $[n]^n$. Henceforth, unless explicitly stated otherwise, all the probabilities are for this probability space; and the events are subsets of $[n]^n$. In order to prove our lower bound we prove a stronger claim that given any EROW algorithm for the pointer chasing problem PC_k (k as in Theorem 1.1), with high probability at any time $t \leq k$ no processor knows a path of length more than $2t$. So we need to upper bound the probability of the event that some processor at time $t \leq k$ knows a path of length $> t$. We reduce this event to a set of simpler local events (local meaning that each involves a single processor and 3 edges of the graph) in the sense that if none of these events happen by time t, no processor knows a long path by that time. Then, we upper bound the probability of each of these events separately. Being local helps in this analysis. We refer to these local events as *bad* events. There are three such events.

Before defining the events we need to describe a labeling of the edges in the digraph $D(x)$ associated with input x. This labeling is used in the definition of the local events. Suppose processor P reads memory cell $M(P')$ (its owner being processor P') at step t. Then the labeling of the edges of $D(x)$ with respect to P, P' and t by labels K, L, B is defined as follows. An edge $(i, x[i])$ is labeled K if P knew it at time $t-1$ (i.e., $(i, x[i]) \in K_x^{t-1}(P)$) but P' did not know it at time $t-1$. K stands for "knows", as P already knew the edge. An edge is labeled L if P did not know it at time $t-1$ but P' did, and since P read $M(P')$ at time t, it learned the edge. L stands for "learns", as P did not know the edge and it learned it from P'. An edge is labeled B if both P and P' knew it at time $t-1$. Clearly, any edge gets at most one of these labels.

Let us define the first event, denoted $[K(L|B)(L|B)]^t$, in detail; the definitions of the other two events will then be evident from their names. The bad event $[K(L|B)(L|B)]^t$ occurs on input x if there exist two processors P and P' and a path (p, q, r, s) in digraph $D(x)$ such that P reads $M(P')$ at time t, and the labeling of the edges (p, q), (q, r), (r, s) with respect to P, P', t is in $[K(L|B)(L|B)]$. $(L|B)$ means L or B. So $[K(L|B)(L|B)]$ stands for the set of four labelings KLL, KLB, KBL, KBB. Similarly $(K|B)$ stands for K or B.

The remaining bad events are $[KLK]^t$, $[L(K|B)(K|B)]^t$.

A crucial Lemma in our lower bound proof states that, for any EROW algorithm, if t is small, then the probability that a bad event occurs on input x in time step t, when the input x is chosen uniformly at random is very small. More precisely,

Lemma 3.1 *For any EROW algorithm on input $[n]^n$, let B^t be the event that at least one of the three bad events happens at time t. Then, there is a constant $d > 1$ such that for time $t \geq 0$,*

$$\Pr_x[B^t] \leq \frac{12d^t \log n}{n}.$$

4 The main result

In this section, we will prove Theorem 1.2 assuming Lemma 3.1, which will be proved in Section 7.

Proof of Theorem 1.2: Fix an EROW algorithm that solves PC_k. Choose an input $x \in [n]^n$ uniformly at random. We are interested in the probability that none of the bad events occurs up to and including time step T, for some T, when x is given as input to the algorithm. From Lemma 3.1, we get that (B^t below is defined in the statement of Lemma 3.1),

$$\sum_{t=1}^{T} \Pr_x[B^t] \leq \sum_{t=1}^{T} \frac{12d^t \log n}{n}.$$

Choose $c = \frac{1}{5 \log d}$ and let $T = c \log n$. Then for n large enough,

$$\sum_{t=1}^{c \log n} \Pr_x[B^t] \leq \sum_{t=1}^{c \log n} \frac{12d^t \log n}{n} \leq \frac{1}{\sqrt{n}}.$$

Thus, on a good fraction of the inputs, none of the bad events occur on or before time step $T = c \log n$.

We have to deal with a technical problem that if the k-walk starting from vertex 1 is not a k-path, then after learning that walk a processor can solve PC'_k for any k'. But this happens with small probability for small k:

$$\begin{aligned} &\Pr_x[\, \log n\text{-walk from vertex 1 is not a } \log n\text{-path}\,] \\ &\qquad \leq \quad 1 - \prod_{i=1}^{\log n} (1 - \frac{i}{n}) \\ &\qquad \leq \quad \frac{(\log n)^2}{n}. \end{aligned}$$

Hence, we have shown that, with probability at least $1 - \frac{1}{\sqrt{n}} - \frac{(\log n)^2}{n} > 1 - \frac{2}{\sqrt{n}}$ (for sufficiently large n), none of the bad events occur in or before time step $T = c \log n$; furthermore, $\log n$-walk starting at 1 is a $\log n$-path. For the rest of this section, fix such an input y. We will now show that the EROW algorithm takes at least $k/2$ steps to solve $PC_k(y)$ for $k \leq c \log n$. To show this, we need the following Lemma.

Lemma 4.1 *Assume that the $2t+1$-walk starting at vertex $j \in [n]$ is a $2t+1$-path. If on an input no bad event happens up to time t, then the maximum length path starting from j, which is known to some processor P at time t, has length $\leq 2t$.*

Proof. By induction on t. Clearly, it holds at the end of the first and second steps: At the end of the first step any processor knows at most one coordinate, and at the end of the second step any processor knows at most two coordinates.

Now assume that the proposition holds at the end of step $t-1$ for $t \geq 3$ and it fails to hold at the end of time step t. I.e., there is a processor P that knows a path of length $2t+1$ starting from vertex j at time t. Let $M(P')$ be the memory cell read by P in time step t. Let us denote the path by $\Pi = (j_0(=j), j_1, j_2, \ldots, j_{2t+1})$. We label the edges of this path using the labels B, K, and L. An edge is labeled by B if the processors P and P' knew this edge at the end of time step $t-1$; it is labeled by K if P knew this edge at the end of time step $t-1$ but P' did not know it by then; and it is labeled by L if P' knew this edge at the end of time step $t-1$ but P did not know it. Clearly all the edges of Π get labeled.

If there is no edge in Π with label K, then it means that Π was known to P' at time $t-1$, which is a violation of the induction hypothesis. So there must be an edge with label K. Similarly, there must be an edge with label L.

Now consider the last edge labeled K in path Π (last meaning last as one traverses the path from j_0 to j_{2t+1}). Since $[K(L|B)(L|B)]^t$ does not occur, this edge must be one of the last two edges of Π. A similar argument gives that the last edge in Π labeled by L, must be one of the last two. Hence the labels of the last two edges of Π form the sequence LK or KL.

Case 1 The last two edges of Π are labeled LK. Consider the first edge e' labeled by K encountered as one traverses Π backwards after starting at the last edge e. At least one such edge exists as otherwise P' knew a path of length $2t-1$ at time $t-1$. Since $[K(L|B)(L|B)]^t$ does not occur, there cannot be a gap of more than one edge between e and e'. So the sequence of edges from e' to e is labeled by KLK. But again, we know that KLK does not occur.

Case 2 The last two edges of Π are labeled KL. Consider the first edge e' labeled L encountered when traversing Π backwards (other than the last edge e). If it is not the third last edge then we get a subsequence from $L(K|B)(K|B)$, which we know cannot occur. Hence there cannot be a gap of more than one edge between e' and the last edge e. So the last three edges are labeled LKL. There must be an edge labeled K before these edges, as otherwise P' knew a path of length $2t-1$ at time t. This K gives a substring of either KLK or $K(L|B)(L|B)$, both of which are forbidden.

This gives a contradiction, and means that for the invariant to be violated at least one of the bad events must occur.

■

Using the Lemma above, we observe that on input y, every processor knows only a path of length at most t starting from vertex 1 at the end of time step t. This, in particular, holds for the processor P_1. By assumption, P_1 writes down the answer in the memory cell $M(P_1)$ in the final step of the algorithm. Hence with high probability, the algorithm will need at least k steps to solve PC_k if $k \leq c \log n$. This proves Theorem 1.2.

5 Knowledge about 2-Paths

In this section we show that the number of 2-paths in the input digraph which are known to some processor at time t behaves as c^t on average. This fact is crucial in the proof that the bad events are rare.

Definition 5.1 $I_x^{t,j}(P;i) = 1$ *if the processor P knows a path of length j starting from vertex i at the end of time step t on input x; $I_x^{t,j}(P;i) = 0$ otherwise. This definition extends naturally to a line l: $I_l^{t,j}(P;i) = 1$ if $I_x^{t,j}(P;i) = 1$ for some input $x \in l$ and is 0 otherwise. That is P knows $x[i]$ and $x[x[i]]$.*

The following lemma is the main result of this section and provides the upper bound mentioned in the beginning of this section.

Lemma 5.2 *For time $t \leq \log n/5$, we have*

$$\frac{1}{n^n} \sum_x \sum_{i,P} I_x^{t,2}(P;i) \leq 2^{4t+3}.$$

Proof. Recall the notion of lines defined in Section 2. The set $[n]^n$ can be partitioned into n^{n-1} lines along direction i for any $i \in [n]$. In particular, define

$$Z_i = \{x \in [n]^n : x[i] = 1\}.$$

Then $\{x^{(i)} : x \in Z_i\}$ is one such partition that we will use in our proof. We have:

$$\frac{1}{n^n} \sum_x \sum_{i,P} I_x^{t,2}(P;i)$$

$$
\begin{aligned}
&= \frac{1}{n^n}\sum_{i,P}\sum_{x} I_x^{t,2}(P;i) \\
&\leq \frac{1}{n^n}\sum_{i,P}\sum_{x\in Z_i} I_{x^{(i)}}^{t,1}(P;i)|K_{x^{(i)}}^t(P)| \\
&= \frac{1}{n^n}\sum_{k}\sum_{i,P}\sum_{x\in Z_i} I_{x^{(i)}}^{t,1}(P;i)I_{x^{(i)}}^{t,1}(P;k) \\
&= \sum_{k} H(k,t)
\end{aligned}
$$

where $H(k,t) = \frac{1}{n^n}\sum_i \sum_{x\in Z_i}\sum_P I_{x^{(i)}}^{t,1}(P;i)I_{x^{(i)}}^{t,1}(P;k)$. The inequality above follows from the fact that if we focus along a line of the form $x^{(i)}$, for some input x and direction i, then the number of paths that processor P knows starting from i can be upper bounded by $|K_{x^{(i)}}^t(P)|$ provided that P knows $x'[i]$ for at least one input $x' \in x^{(i)}$. The equality in the third expression above follows from that fact that the knowledge of P can be expressed as a sum of indicator functions.

We show below that $H(k,t) \leq 2^{4t+3}/n$ for all $k \in [n]$, which will complete the proof. ∎

First we give a combinatorial interpretation for $H(k,t)$.

Definition 5.3 *For any $k, i \in [n]$ and input x, the line $x^{(i)}$ is said to be k-covered by time t if $S_{x^{(i)}}^t(i)\cap S_{x^{(i)}}^t(k)$ is nonempty.*

In words,

$x^{(i)}$ is k-covered with above qualifications if there exist $z_1, z_2 \in x^{(i)}$ such that some processor P knows $z_1[i]$ on input z_1 at time t and P knows $z_2[k]$ on input z_2 at time t. Since $S_{x^{(i)}}^t(i) = S_x^t(i)$ by the proof of Lemma 2.8(1), we can assume that $z_1 = z_2$.

Lemma 5.4 *Fix any $k \in [n]$. If l_k^t is the number of k-covered lines by time t, then*

$$H(k,t) \leq \frac{2^t l_k^t}{n^n}.$$

Proof. From the definition of $H(k,t)$ above, we get that

$$H(k,t) = \frac{1}{n^n}\sum_i \sum_{x\in Z_i} |S_{x^{(i)}}^t(i)\cap S_{x^{(i)}}^t(k)|.$$

From Lemma 2.8(1), $|S_{x^{(i)}}^t(i)| \leq 2^t$. Hence, $|S_{x^{(i)}}^t(i)\cap S_{x^{(i)}}^t(k)|$ is at most 2^t if $x^{(i)}$ is k-covered by time t and is 0 otherwise. Therefore, we get that $H(k,t) \leq (2^t.l_k^t)/n^n$. ∎

Lemma 5.5 *For all $k \in [n]$,*

$$l_k^t \leq 2^{3t+3}n^{n-1}.$$

Proof. To estimate the number of k-covered lines by time t, we start by constructing a bipartite graph between inputs and k-covered lines in which we connect an input to a line if the line passes through it. It is easy to see that each line vertex has degree n. We now show that each point vertex has degree at most 2^{3t+3}. Then it will follow that $nl_k^t \leq 2^{3t+3}n^n$ and hence $l_k^t \leq 2^{3t+3}n^{n-1}$.

Let us consider the set of k-covered lines, $L = \{l_1, l_2, \ldots, l_m\}$ passing through a fixed point vertex x. Let us denote the corresponding floating indices of these lines by $f_1, f_2, \ldots, f_m$. That is, for $1 \leq j \leq m$, $l_j = x^{(f_j)}$. For every line l_j, $1 \leq j \leq m$, fix a processor P_{l_j} and an input $x_j \in l_j = x^{(f_j)}$ such that on input x_j, P_{l_j} knows $x_j[k]$ by the end of time step t. Let $\lambda_j = K_{x_j}^t(P_{l_j})$. Now consider a digraph D with L as the vertex set in which we draw an edge from l_i to l_j if $(f_i, x_j[f_i])$ is in λ_j. Each vertex in the graph has in-degree at most 2^t. Hence the total number of edges is at most $m2^t$.

Proposition 5.6 *In the digraph D, every independent set is of size less than $2^t(2^t+1)$.*

Proof. Suppose not. Without loss of generality, let us assume that $\{l_1, l_2, \ldots, l_s\}$ is an independent set. Consider the set of processors $S = \{P_{l_1}, P_{l_2}, \ldots, P_{l_s}\}$ that caused the labels $\{\lambda_1, \lambda_2, \ldots, \lambda_s\}$ to be assigned to the s lines.

It is possible that P_{l_i} and P_{l_j} are the same for $i \neq j \in [s]$. However, it cannot be the case that more that 2^t of them are equal. Assume, wlog, that $P_{l_1}, P_{l_2}, \ldots, P_{l_{2^t+1}}$ are all P. We construct input y by setting $y[f_1] := x_1[f_1], y[f_2] := x_2[f_2], \ldots, y[f_{2^t+1}] := x_{2^t+1}[f_{2^t+1}]$, and $y[j] := x[j]$ for $j \notin \{f_1, \ldots, f_{2^t+1}\}$. On input y, P knows 2^{t+1} values: $y[f_1], \ldots, y[f_{2^t+1}]$ at time t. This contradicts Lemma 2.5(2).

It follows that there are at least 2^t+1 distinct processors in S. Let us assume, wlog, that $P_{l_1}, P_{l_2}, \ldots, P_{l_{2^t+1}}$ are distinct. We construct input y by setting $y[f_1] := x_1[f_1], y[f_2] := x_2[f_2], \ldots, y[f_{2^t+1}] := x_{2^t+1}[f_{2^t+1}]$, and $y[j] := x[j]$ for $j \notin \{f_1, \ldots, f_{2^t+1}\}$. On input y all the processors $P_{f_1}, P_{f_2}, \ldots, P_{f_{2^t+1}}$ know the value $y[k]$ by the end of time t, a contradiction to Lemma 2.5(1). ∎

Using Turán's Theorem (see, e.g., [1]), we get that there are at least $m^2/2^{2t+3}$ edges in this graph. Therefore,

$$\frac{m^2}{2^{2t+3}} \leq m2^t.$$

This completes the proof of Lemma 5.5. ∎

Lemmas 5.4 and 5.5 give that $H(k,t) \leq 2^{4t+3}/n$. This is what we needed to finish the proof of Lemma 5.2.

6 Learning new information

In this section, we look at the likelihood that a processor can learn "useful" new information by reading a memory cell. In the EROW model it is difficult to go and read the useful information because some other processor might also be attempting to read the same information. Lemmas in this section can be seen as concrete quantitative statements of this type.

6.1 Forward Learning

Suppose on input x at time $t-1$ processor P knows $x[q]$. Now if at time t, another processor P' who knows $x[p] = q$, reads $M(P)$ then it learns some useful information in that it has extended the path it knows starting from p. Since P' is building its path in the forward direction, we say that P' forward-learns. It is difficult to forward-learn in general for more than one processors because they may need to read the same memory cell which is not allowed. The notion of "forward-succeeds" that we actually utilize in the proof is similar to "forward-learns", but more complicated for technical reasons.

Definition 6.1 *For a processor P and an index $q \in [n]$, we say that a processor $P'(\neq P)$ forward-succeeds on an input $x \in [n]^n$ at time step t with respect to P and q if there exists an index $p \neq q$ with the following properties: (1) $x[p] = q$; (2) $(p, q) \in K_x^{t-1}(P')$ but $(p, q) \notin K_x^{t-1}(P)$; $(q, x[q]) \in K_x^{t-1}(P)$, P' reads $M(P)$ at time step t and thereby $(q, x[q]) \in K_x^t(P')$.*

Denote by $F^t(P; q)$ the event that some processor forward-succeeds with respect to processor P and index q at time t.

Lemma 6.2 below is a conditional version of the following: For any P and q, the probability that some processor P' forward-succeeds at time t with respect to P and q is small.

Define $Path^t(P; q, r), q, r \in [n]$, to be the event that processor P knows $x[q] = r$ and $x[r]$ by the end of step t.

Lemma 6.2 *There exists a fixed constant b such that for any processor P, indices $q, r \in [n]$, and $t \leq \log n/5$ we have:*

$$\Pr_x[F^t(P; q) | Path^{t-1}(P; q, r)] \leq \frac{b^t \log n}{n}.$$

The following combinatorial Lemma will be useful in proving Lemma 6.2.

Lemma 6.3 *Let $t \leq \log n$, and let M_t be a set of disjoint subcubes of $[n]^n$ with the following properties: (1) Each subcube in M_t has codimension 2^t; (2) In addition, each subcube has some fixed coordinate set to 1. Then,*

$$|M_t| \leq b^t n^{2^t - 1} \log n,$$

where b is a fixed constant.

Proof. Let m_t denote $b^t n^{2^t-1} \log n$. Think of the subcubes satisfying the properties (1) and (2) in the statement of the Lemma as vertices of a graph $G = (V, E)$, where two subcubes are connected if they have nonempty intersection. The number of vertices $|V|$ in G is $\binom{n}{2^t}(n^{2^t} - (n-1)^{2^t})$. The set M_t in the statement corresponds to an independent set in G. To prove the Lemma, we upper bound the size of maximum independent set in G by upper bounding the clique cover number (the minimum number of cliques in G that cover all the vertices) of G. It is not difficult to see that clique cover number provides an upper bound on the size of maximum independent set. We upper bound $|M_t|$ by showing that the vertices of G can be covered by m_t cliques.

Pick $x \in [n]^n$ at random according the following distribution D:

For each $j \in [n]$,

$$\begin{aligned}
\Pr_D[x[j] = 1] &= \frac{1}{2^t} \\
\Pr_D[x[j] = i] &= (1 - \frac{1}{2^t})(\frac{1}{n-1}) \text{ for } i \in [n] \setminus \{1\}.
\end{aligned}$$

We say that a subcube C agrees with $x \in [n]^n$ if $x \in C$, or in other words, the fixed coordinates of C are fixed in the same way as in x.

Then, for a fixed vertex C in G and a random x picked according to D,

$$\Pr_D[C \text{ agrees with } x] \geq \frac{1}{2^t}\left[\left(1 - \frac{1}{2^t}\right)\frac{1}{n}\right]^{2^t-1}.$$

Choose m_t values for x at random according to D. Then, the expected number of vertices in G that do not agree with any of these m_t choices is at most:

$$\left(1 - \frac{1}{2^t}\left[\left(1 - \frac{1}{2^t}\right)\frac{1}{n}\right]^{2^t-1}\right)^{m_t} |V|.$$

We can choose a fixed constant $b > 1$ such that the expected number of vertices that do not agree with any of the m_t random choices is less than 1. This implies that we can choose m_t values of x such that every vertex of G agrees with one of them. All the vertices that agree with one fixed value of x form a clique. So, the clique cover number of G is at most m_t. ∎

Proof. (of Lemma 6.2) The set of inputs $x \in [n]^n$ on which $Path^{t-1}(P; q, r)$ occurs can be viewed as a collection of disjoint subcubes depending on the knowledge of P at time $t-1$. Each such subcube has codimension at most 2^{t-1}. Fix one such subcube C of codimension $d \le 2^{t-1}$. If $F^t(P; q)$ happens at some input, it can be due to only one processor, because only one processor can read $M(P)$ in step t; and knowledge of a processor who causes $F^t(P; q)$ to happen induces a subcube. Thus the set of inputs on which $F^t(P; q)$ happens is a disjoint union of subcubes $C_1, \ldots, C_k$ of codimension at most 2^t.

Consider the nonempty subcubes $C'_1, \ldots, C'_{k'}$ from $C \cap C_1, \ldots, C \cap C_k$. These subcubes all have codimension $\le d + 2^t$ and each of them has a fixed coordinate, which is floating in C, set to q. This is because in the definition of $F(P; q)$ we required that P did not know (p, q) at time $t-1$. By partitioning into smaller subcubes if necessary, we can assume that each subcube C'_i ($i \in [k']$) has codimension exactly $d + 2^t$. Now we upper bound k' using Lemma 6.3. Subcubes C'_i can be extended to subcubes C''_i ($i \in [k']$) of codimension 2^t by changing the fixed coordinates which are also fixed in C to floating coordinates. C''_i are disjoint, have codimension 2^t, and have at least one fixed coordinate set to q. Hence Lemma 6.3 gives that $k' \le m_t$. Hence, the fraction of inputs in C on which $F^t(P; q)$ happens is $(k' n^{n-d-2^t})/n^{n-d} \le (m_t n^{n-d-2^t})/n^{n-d} \le (b^t n^{2^t-1} \log n)/n^{2^t} = (b^t \log n)/n$. ∎

6.2 Backward Learning

If on input x processor P knows $x[q]$ at time $t-1$ and learns $x[p] = q$ in the next step then we say that it backward-learns. Intuitively, to backward-learn a processor can not do anything but grope in the dark. For the proof we need a similar notion: "backward-succeeds".

Definition 6.4 *Say that a processor P backward-succeeds on an input $x \in [n]^n$ at time step t with respect to a processor $P' \neq P$ and an index q if there exists an index p $\neq$ q with the following properties: (1) $x[p] = q$; (2) $(p, q) \notin K_x^{t-1}(P)$ and $(p, q) \in K_x^{t-1}(P')$; (3) P reads $M(P')$ at step t and thereby $(p, q) \in K_x^t(P)$.*

We denote this event by $B^t(P; q)$.

The probability that a fixed processor P succeeds on input x at time step t with respect to some processor and some index is small. The following Lemma proves a conditional version of this.

Lemma 6.5 *There exists a fixed constant b such that for any processor P, indices $q, r \in [n]$, and time $t \le \log n/5$:*

$$\Pr_x[B^t(P; q) | Path^{t-1}(P; q, r)] \le \frac{b^t \log n}{n}.$$

Proof. This is similar to the proof of Lemma 6.2, the only difference being that now we are working with the event $B^t(P; q)$ instead of $F^t(P; q)$. ∎

7 Bad events are rare

This section is devoted to proving Lemma 3.1 which states that the four bad events defined in Section 3 occur with small probability. We will show the inequality of that Lemma for each event.

Below, upper bounds in the probabilities of the bad events will be seen to be $\le \frac{b^t \log n}{n} 2^{5t+2}$. This implies that in Lemma 3.1, we can take $d = b2^5$.

7.1 The bad event, $[K(L|B)(L|B)]^t$

Let us first recall the meaning of $[K(L|B)(L|B)]^t$ in the language of some of the events that we have defined. For $[K(L|B)(L|B)]^t$ to happen $Path^{(t-1)}(P; q, r)$ should happen for some processor P and path $(q, r, x[r])$ in $D(x)$. Furthermore, $F^t(P; q)$ happens, that is there is a processor P' who at time $t-1$ knows a coordinate $x[p] = q$ (which is not known to P at that time) and reads $M(P)$ in the read phase of step t. That is,

$$[K(L|B)(L|B)]^t = \bigcup_{P; q, r \in [n]} F^t(P, q) Path^{t-1}(P; q, r).$$

This gives,

$$\begin{aligned}
&\Pr[[K(L|B)(L|B)]^t] \\
&\quad \le \sum_{P; q, r \in [n]} (\Pr_x[F^t(P; q) | Path^{t-1}(P; q, r)] \\
&\qquad \times \Pr[Path^{t-1}(P; q, r)]) \\
&\quad \le \frac{b^t \log n}{n} \sum_{P; q, r \in [n]} \Pr[Path^{t-1}(P; q, r)] \\
&\qquad \text{(using Lemma 6.2)} \\
&\quad \le \frac{b^t \log n}{n} 2^{4(t-1)+3} \text{ (using Lemma 5.2 and double counting)} \\
&\quad \le \frac{b^t \log n}{n} 2^{4t}
\end{aligned}$$

This is what we wanted to prove.

7.2 The bad event, $[L(K|B)(K|B)]^t$

By a discussion similar to the one for the event $[K(L|B)(L|B)]^t$ we have,

$$\begin{aligned}
&\Pr[[L(K|B)(K|B)]^t] \\
&\leq \sum_{P;q,r\in[n]} (\Pr_x[B^t(P;q)|Path^{t-1}(P;q,r)] \\
&\quad \times \Pr[Path^{t-1}(P;q,r)]) \\
&\leq \frac{b^t \log n}{n} \sum_{P;q,r\in[n]} \Pr[Path^{t-1}(P;q,r)] \\
&\quad \text{(using Lemma 6.5)} \\
&\leq \frac{b^t \log n}{n} 2^{4(t-1)+3} \text{ (using Lemma 5.2)} \\
&\leq \frac{b^t \log n}{n} \cdot 2^{4t}.
\end{aligned}$$

7.3 The bad event, $[KLK]^t$

Denote by $G^t(P;p,q)$ the event:

$$[(p,q) \in K_x^{t-1}(P)] \wedge [(q,x[q]) \in K_x^t(P)] \wedge [(q,x[q]) \notin K_x^{t-1}(P)].$$

In words, it is the event that the edge (p,q) is known to P at time $t-1$; edge $(q,x[q])$ is not known to P at time $t-1$ and P learns it at time t.

Denote by $H^t(P;q)$ the event $(x[q], x[x[q]]) \in K_x^{t-1}(P)$.

We have

$$[KLK]^t = \bigcup_{P;p,q} G^t(P;p,q) H^t(P;q).$$

Hence,

$$\begin{aligned}
&\Pr[[KLK]^t] \\
&= \sum_{P;p,q} \Pr[G^t(P;p,q)] \Pr[H^t(P;q)|G^t(P;p,q)] \\
&\leq (\max_{P;p,q} \Pr[H^t(P;q)|G^t(P;p,q)]) \\
&\quad \times \sum_{P;p,q} \Pr[G^t(P;p,q)].
\end{aligned}$$

Now, $\sum_{P;p,q} \Pr[G^t(P;p,q)]$ is at most the expected number of tuples $(P,(p,q))$, where P knows $x[p]=q$ and $x[q]$. By Lemma 5.2 we know that this is no more than 2^{4t+3}.

Lemma 7.1

$$\max_{P;p,q} \Pr[H^t(P;q)|G^t(P;p,q)] \leq \frac{2^t}{n}.$$

Proof. Fix t, P, p, q. If on some input $G^t(P;p,q)$ happens then it is because P reads some register $M(P')$ in step t. The set of inputs on which $G^t(P;p,q)$ happens is the union of disjoint subcubes depending on the knowledge of P at time $t-1$ and the knowledge of P' (P' may be different for different inputs) just before the time it first learned $(q,x[q])$. Each such subcube has codimension $\leq 2^t$. Fix any such subcube C.

We claim that $x[q]$ is a floating coordinate in C. This is because P did not know about $x[q]$ at time $t-1$, and the fact that P' learned $x[q]$ at some time before t did not depend on the value of $x[q]$.

Now the event $H^t(P;q)$ happens in C if $x[q]$ points to one of the coordinates which were known to P at time $t-1$. The fraction of inputs on which this happens is $\leq \frac{2^{t-1}}{n}$ because P knows at most 2^{t-1} coordinates at time $t-1$. ∎

Thus, $\Pr[[KLK]^t] \leq \frac{2^{t-1}}{n} 2^{4t+3} \leq \frac{2^{5t}4}{n}$.

8 The separation result

In this section, we prove Theorem 1.1. For any binary string y of length n, let us denote the ith bit of y by y_i. We now define a Boolean function $f : \{0,1\}^N \to \{0,1\}$ with $N = n\log n + \log\log n$ as follows:

Definition 8.1 *For any $y \in \{0,1\}^N$, $f(y) = PC_{c\log n}(x_1, x_2, \ldots, x_n)_i$ where $x_i = y_{(i-1)\log n+1} \cdots y_{i\log n}$ for $1 \leq i \leq n$ and $i = y_{n\log n+1} \cdots y_{n\log n+\log\log n}$ and c is the constant as in Theorem 1.2.*

We claim that f can be evaluated in $O(\log\log N)$ time on an CROW PRAM but requires $\Omega(\log N)$ time on the EROW PRAM. Our proof is exactly the same as the proof of Theorem 2 in [10]. In the CROW PRAM, the processors gather each x_i into a single cell and then use the pointer doubling technique to compute the answer. It can also be seen that if f can be solved in time T in the EROW model, then $PC_{c\log n}$ can be solved in time $T+O(\log\log n)$. This is done by making $\log n$ copies of the input and computing the ith bit of the answer in the ith copy using the algorithm for f. Now Theorem 1.1 follows from Theorem 1.2.

We sketch the proof for the lower bounds for randomized EROW algorithms. If f can be evaluated correctly with probability $\frac{1}{2}+\epsilon$ for a constant $0 < \epsilon < 1/2$ in time T, then by doing sufficiently many parallel repetitions we get an algorithm which evaluates f correctly with probability $1-\delta$ for any constant $\delta > 0$ and runs in time $T + t(\delta)$. Here $t(\delta)$ is the overhead time algorithm incurrs in performing parallel repetitions, and is independent of n. Now, make $\log n$ copies of the input and compute the ith bit of the answer to $PC_{c\log n}$ from the ith copy using the above algorithm. Collect the $\log n$ bits in time $\log\log n$ in M_1. This takes time $T + t(\delta) + O(\log\log n)$, and the computed answer is correct with probability $(1-\delta)^{\log n} \approx 1/n^{\delta}$ for δ small enough. If $T + t(\delta) + O(\log\log n) < c\log n$ we know that success probability is $\leq 2/\sqrt{n}$ and so δ cannot be made arbitrarily small, and hence ϵ cannot be any positive constant if, say, $T < \frac{c}{3}\log n$.

9 Conclusions

We have shown an optimal separation between the CROW and EROW PRAM models. But several aspects of exclusive read remain ununderstood. An important open problem is to achieve optimal separation between the EREW and CREW models. The best result on this is [10]. It is known that EREW is more powerful than EROW if we allow partial functions. What happens for functions on complete domain? Can we find a simple combinatorial characterization of EROW and EREW complexity of a boolean function in the style of [9] where it is done for CROW and CREW complexity of boolean functions in terms of block sensitivity.

References

[1] N. Alon and J. Spencer, The probabilistic method, Wiley-Interscience, New York, 1992.

[2] P. Beame, Lower Bounds in Parallel Machine Computation, TR 198/87, Department of Computer Science, University of Toronto, 1987.

[3] S. Cook, C. Dwork and R. Reischuk, Upper and lower bounds for parallel random access machines without simultaneous writes, *SIAM Journal of Computing*, 15, 1986, 87–98.

[4] P. Dymond and W. Ruzzo, Parallel RAMs with owned global memory and deterministic context-free language recognition, *Journal of the ACM*, 47(1), 2000, 16 - 45.

[5] F. Fich, The Complexity of Computation on the Parallel Random Access Machine, in *Synthesis of Parallel Algorithms*, J. Reif, ed., Morgan-Kaufmann, 1993.

[6] F. Fich and A. Wigderson, Towards understanding concurrent read, *SIAM Journal of Computing*, 19, 1990, 718–727.

[7] E. Gafni, J. Naor and P. Ragde, On separating the EREW and CROW models, *Theoretical Computer Science*, 68, 1989, 343–346.

[8] R. Karp and V. Ramachandran, A survey of parallel algorithms for shared-memory machines, *Handbook of Theoretical Computer Science*, Vol. A, 1988.

[9] N. Nisan, CREW PRAMs and decision trees, *SIAM Journal of Computing*, 20 (1991), no. 6, 999–1007.

[10] N. Nisan and Z. Bar-Yossef, Pointer Jumping Requires Concurrent Read, In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, 1997, 549–558.

[11] H. U. Simon, A tight $\Omega(\log\log n)$ bound on the time for parallel RAMs to compute nonboolean functions, *Proceedings of FCT*, LNCS 158, 1983.

[12] M. Snir, On parallel searching, *SIAM Journal of Computing*, 14, 1985, 688–708.

[13] A. Yao, Probabilistic computations: Towards a unified measure of complexity, *In Proceedings of the Seventeenth IEEE Conference on Foundations of Computer Science*, 1977, pages 222–227.