Pointer Jumping Requires Concurrent Read *

Noam Nisan
Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
noam@cs.huji.ac.il

Ziv Bar-Yossef
Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
zivi@cs.huji.ac.il

December 20, 1999

Abstract

We consider the well known problem of determining the k'th vertex reached by chasing pointers in a directed graph of out-degree 1. The famous "pointer doubling" technique provides an $O(\log k)$ parallel time algorithm on a Concurrent-Read Exclusive-Write (CREW) PRAM. We prove that this problem requires $\Omega(k)$ steps on an Exclusive-Read Exclusive-Write (EREW) PRAM, for every $k \leq \frac{1}{100}\sqrt{\log n}$, where n is the number of vertices.

This yields a boolean function which can be computed in $O(\log \log n)$ time on a CREW PRAM, but requires $\Omega(\sqrt{\log n})$ time on even an "ideal" EREW PRAM. This is the first separation known for boolean functions between the power of EREW and CREW PRAMs. Previously, separations between EREW and CREW PRAMs were only known for functions on "huge" input domains, or for restricted types of EREW PRAMs.

1 Introduction

1.1 Pointer Jumping

Perhaps the most basic technique used in parallel algorithms on graphs or lists is the so-called "pointer jumping" or "pointer doubling" technique. This technique allows traversal of a linked list of length k in $O(\log k)$ time. The basic problem may be defined as follows:

Assume that for every $1 \leq i \leq n$ you are given an index of its "successor", $s[i]$ (a number in the range $\{1, \ldots, n\}$). The problem is to "reach" the k'th index in the list: $1, s[1], s[s[1]], s[s[s[1]]], \ldots$. In general, for a given input array $s[1] \ldots s[n]$, we define $s_0 = 1$, and for every $k > 0$, the k'th element is $s_k = s[s_{k-1}]$.

The basic pointer doubling technique repeatedly executes the following pointer doubling step: In parallel for all i do $s[i] \leftarrow s[s[i]]$. It is not difficult to see that after t such steps, $s[1]$ now holds the value of s_{2^t}. Thus, $O(\log k)$ steps are necessary to reach s_k. This basic technique is very often employed in various list manipulation algorithms, as well as in many tree and graph algorithms which have list sub-structures in them. See [KR88] for a survey and references.

Let us analyze the memory access requirements from such a pointer doubling step. Fix some i, and consider the memory location holding $s[i]$. It is clear that only a single processor writes into this cell. Many processors, though, may need to access its value: for any j such that $s[j] = i$, computing the value of $s[s[j]]$ requires reading the value of $s[i]$. We thus see that concurrent access of processors to a memory cell is needed for reading but not for writing.

In some important cases we can be assured in advance that $s[1] \ldots s[n]$ really hold a simple list, i.e. that for each i there exists (at most) a single predecessor. In these cases even concurrent read access is not needed. Much work

*This work was supported by USA-Israel BSF grant 92-00043 and by Israel Science Foundation grant 69/96-1.

has been done in this special case to fine-tune this algorithm as to improve its efficiency (see [KR88] for references). Our main theorem (stated below) shows that the general problem does indeed *require* concurrent read access, and that otherwise a lower bound of $\Omega(k)$ can be proved.

1.2 EREW vs. CREW PRAMs

PRAMs (Parallel Random Access Machines) provide an elegant model for parallel computation while hiding many "secondary" issues such as the communication mechanism between processors. In the PRAM model processors communicate with each other using a shared memory, where in each time step each processor can read one shared memory cell, perform a computation on its local memory, and write into one shared memory cell. PRAMs are further classified according to whether different processors can concurrently access a memory cell for reading, or writing. We thus get three well studied models of PRAM: Exclusive-Read Exclusive-Write (EREW), Concurrent-Read Exclusive-Write (CREW), and Concurrent-Read Concurrent-Write (CRCW). When attempting to provide lower bounds for PRAMs, the computational power of each processor is usually not limited – the so-called "ideal PRAM". Upper bounds, of course, utilize only "realistic" processing power at each processor. See [KR88] for further details.

A well known result due to Cook, Dwork, and Reischuk [CDR86], shows that write-concurrency provides true additional power: the OR function on n bits requires $\Omega(\log n)$ time on a CREW PRAM, but can be done in $O(1)$ time on a CRCW PRAM. The lower bounds holds for any number of processors or memory cells. It should be noted that any function can be computed on any PRAM in $O(\log n)$ time if the number of processors is not bounded so the separation is tight. A natural question is whether read-concurrency provides true additional power.

Snir [Sni85] proves that given inputs y and a sorted list $x_1 < x_2 < \ldots < x_n$, finding the minimum i such that $x_i > y$ requires $\Omega(\sqrt{\log n})$ time on a EREW PRAM, but can be done in $O(1)$ time on a CREW PRAM. This lower bound uses Ramsey theory, and holds only if the domain of the x_i's is huge. The question of whether concurrent-read helps to compute *boolean* functions remained completely open. It should be noted, though, that such a large gap can not be shown for boolean functions since it is known ([Sim83]) that for a boolean function the gap between CREW and EREW PRAMs can not be more than exponential.

Two partial results have been proved: Gafni, Naor, and Ragde [GNR89] improved the example of Snir to a function over a complete domain (but still a huge one). Again, this sheds no light on the case of boolean functions. Fich and Wigderson [FW90] attempted separating EREW from CREW PRAMs for boolean functions, but their lower bound only held for a weaker variant of EREW PRAM called an EROW (Owner-Write). They left the general question open.

We define the "pointer jumping" function PJ_k as accepting as its input an array $s[1]...s[n]$, where $0 \leq s[i] \leq n$ for each i, and outputting the value of s_k as defined above. As noted above this can be done in $O(\log k)$ time on a CREW PRAM, and we prove:

Theorem 1 Any EREW PRAM algorithm solving PJ_k requires $\Omega(k)$ time, for any $k = k(n) \leq \frac{1}{100}\sqrt{\log n}$.

While PJ_k is not a boolean function, both its domain and its range are small, and thus a separation for a boolean function can be easily deduced (by encoding the input in binary, and adding an "index" input specifying the output bit requested).

Theorem 2 There exists a boolean function which can be computed in $O(\log \log n)$ time on a CREW PRAM, but requires $\Omega(\sqrt{\log n})$ time on a EREW PRAM.

1.3 Intuition for the Lower Bound

The basic limitation of exclusive read is that information cannot be dispersed quickly to many processors. Intuitively the number of processors that "know" the value of an input bit can at most double each time step. This is as opposed to a CREW PRAM in which all processors can know the value of, say, x_1 within a single step. Another "fact" which motivates our technique is that because of exclusive write the number of inputs that each processor "knows" can also at most double each time step. (both these "facts" are essentially true in some precise sense).

These two "facts" combined imply that for almost all pairs of input indices i, j, there is not even a single processor that "knows" simultaneously the i'th input component and the j'th input component.

The basic strategy we will use for the lower bound is a restriction of the input domain. We will build the restriction step-by-step, each step "killing" some of the input indices and fixing their values to constants. This step is done in a way that assures that *every processor or memory cell knows the value of at most a single live input location*. To see why this can be done consider a graph over $1...n$, where an edge connects i and j iff some processor or memory cell simultaneously knows both input values. What we have seen in the previous paragraph is that this graph is not very dense. We thus can use Turan's lemma to find a large independent set in it – these will be the live inputs.

This argument above can be formalized to indeed give a lower bound for the EROW PRAM model. The restriction allows making the stated "facts" indeed formally true within this restriction, and thus the induction can proceed. In each step another pointer in the chain is "sacrificed" in order to allow us to continue the chain into our chosen live set.

The main problem in extending this argument to the EREW model is that the EREW model does provide some mechanism for fast dissemination of information. Assume processor 1 reads input $s[1]$, and then writes a marker, "*", into memory cell number j, where $j = s[1]$. Note that now there are n memory cells which have some information regarding the value of $s[1]$ (each cell j knows whether $s[1] = j$). It is hard to control this knowledge, and thus it is impossible to formalize the "fact" that each input is known by only a few processors. (Note that the previous example cannot be done on a EROW PRAM, and indeed the lower bound for EROW PRAM is not hurt.)

Our solution for this is to note a weaker sense in which it is possible to formalize the "fact" that each input is known by only a few processors. What can be made formal is that for each i, j, the information about whether $s[i] = j$ can only be known by a few processors. With this information we now do not build a graph but rather a hypergraph of triplets i, j, k, where some processor knows both whether $s[i] = j$ and some information about k. The induction step still proceeds by finding a (nearly) independent set within this graph.

Several difficulties creep in at this point starting with the fact that due to the number of hyper-edges, an independent set cannot be found, but rather a "nearly"-independent set is used. This in turn complicates the definition of the restriction. A final complication revolves around triplets where $i = k$ which may influence the computation, but cannot be bounded by the techniques we employ for "normal" triplets. We handle this by finding "normal" triplets which can "take blame" for these special ones.

1.4 Overview of the Paper

The rest of the paper is divided into five major sections. The **first section**, *Basic Definitions*, outlines the framework of our discussion. The EREW model and the properties of algorithms which work according to it are described. Two important notions (processor *state* and memory cell *contents*), which enable us later to quantify the "knowledge" of processors/memory cells, are introduced. A definition of the *pointer jumping* function is presented. The last sub-section introduces the main tool for proving the lower bound, the *restriction*.

The **second section**, *Processor and Memory Cell Dependencies*, discusses the *dependency* of processors and memory cells on components of the input. This notion tries to capture the amount of "knowledge" processors and memory cells have about the input. A sequence of propositions examines the dependency notion and its interaction with the restriction notion. The last part of the section presents special kind of restrictions, the *clean restrictions*. These restrictions allow minimal number of dependencies. The main result of the section (corollary 1) bounds the number of dependencies in a clean restriction.

The objective of the following two sections is to outline an iterative method for proving the existence of a clean restriction at any time t. Each stage of the iteration uses the restriction obtained in the previous stage and throws out the "bad triplets" from it. The **third section**, *Bad Triplets*, discusses these triplets, and presents two lemmas regarding them: lemma 2 proves that it is enough to remove the bad triplets of a restriction in order to make it clean. Lemma 3 bounds the number of bad triplets a restriction may pick up in a single step. The **fourth section**, *The Main Lemma*, gives the main lemma: lemma 4 proves that given a clean restriction at time t, we can produce a clean restriction at time $t + 1$. This implies corollary 2 stating that a clean restriction can be produced for any t which is small enough ($t \leq \frac{1}{100}\sqrt{\log n}$).

Finally, the **fifth section**, *The Lower Bound*, presents the two theorems of the paper.

2 Basic Definitions

2.1 The EREW PRAM Model

In this paper we consider algorithms which work under the EREW version of the ideal PRAM model. The algorithms may use an infinite number of processors or shared memory cells, where the computational power of the processors and the capacity of the memory cells are unlimited. Before the algorithm begins the input is set in the first n shared memory locations. We assume n is known to all the processors. The output is written into the first memory cell by the end of the algorithm. The processors run synchronously, and the execution proceeds in *time steps*. The only difference in the model we use here from the standard model concerns these steps: we separate them into READ steps (in which every processor may read from one shared memory location and then perform a local computation) and WRITE steps (in which every processor may write into one shared memory cell). The odd time steps are READ steps and the even steps are WRITE steps. The exclusive read property doesn't enable two processors to read from the same memory cell at the same time. Similarly, exclusive write prevents two processors from writing into the same memory cell simultaneously.

We use two notions to formalize the knowledge gained by processors and memory cells during the execution of the algorithm. The first is the *processor state* (denoted by $state_p(x,t)$) which describes the precise state of a processor p at a given time t on a given input x (i.e. the contents of its local memory and registers, etc.). This state fully determines the action of the processor at the next time step. The second is the *memory cell contents* (denoted by $content_m(x,t)$) which denotes the data held in a memory cell m at a given time t on a given input x. WLOG, we assume that processors and memory cells don't "forget" anything during the execution. This means that if the states of a processor at time t are different on inputs x and y, then they will stay different at any time $t' > t$. The same for memory cells. Furthermore, if the algorithm terminates its execution by time t we define for each $t' > t$: $state_p(x,t') = state_p(x,t)$ and $content_m(x,t') = content_m(x,t)$.

2.2 The Pointer Jumping Function

The following definition formally describes the function for which we are going to prove the lower bound in the EREW model:

Definition 1 *The* **pointer jumping function** $PJ_k : \{0,1,2,\ldots,n\}^n \rightarrow \{0,1,\ldots,n\}$ *is defined recursively:* $PJ_0(x) = 1$*, and for* $k > 0$*,* $PJ_k(x) = x_i$*, where* $i = PJ_{k-1}(x)$*. We implicitly assume* $x_0 = 0$ *in this definition (i.e. if* $PJ_{k-1}(x) = 0$*, then also* $PJ_k(x) = 0$*.)*

Given an input x of PJ_k, we denote by $x^{i \rightarrow j}$ the input which is identical to x in all its components, except maybe for the i'th component, which equals to j.

2.3 Restrictions

Our main tool in finding inputs of PJ_k, on which a EREW algorithm runs relatively slow is the $restriction$. The restriction is a subset of the PJ_k domain, which has several properties, as specified in the next definition:

Definition 2 *A* **restriction** ρ *is a subset of the* PJ_k *inputs, which is specified by the triplet* $\langle P, A, D\rangle$ *as follows:*
1. $P = \{i_0, \ldots, i_l\} \subseteq \{1,\ldots,n\}$ *is called the* **path** *of* ρ*. The first element in the path,* i_0*, is always 1. The path's elements are arranged in an ascending order, i.e.* $i_0 < i_1 < \ldots < i_l$*.* l *is called the* **length** *of* ρ *and it is denoted by* $len(\rho)$*.*

2. $A \subseteq \{i_l + 1, \ldots, n\}$ is called the **set of live elements** *of ρ.*
3. $D = \{D_i\}_{i \in A}$ are the **forbidden sets** *of the live elements. Each D_i is a subset of $A \cap \{i+1, \ldots, n\}$.*

We identify ρ with the set of all possible inputs that are consistent with its definition. $\rho = S_1 \times \cdots S_n$, where S_i is defined as follows:

$$S_i = \begin{cases} \{i_{s+1}\} & i = i_s, \\ & 0 \le s \le l-1 \\ A \cup \{0\} & i = i_l \\ ((A \cap \{i+1, \ldots, n\}) \setminus D_i) \cup \{0\} & i \in A \\ \{0\} & Otherwise \end{cases}$$

The value $def(\rho) = \max_{i \in A} \frac{|D_i|}{|A|}$ is called the **deficiency** *of ρ.*

Each index $i \in \{1, \ldots, n\} \setminus (A \cup P)$ is called a **dead element** *of ρ. i is called a* **constant element***, if it is either a dead element or an internal element in the path of ρ (i.e. it belongs to the path, but it is not the last element in it).*

An intuition for the restriction notion: let us consider the inputs of a restriction ρ as graphs. All these inputs "begin" with the path P: node no.1 points to node no.i_1, no. i_1 to i_2, ...until node no.i_{l-1} which points to node no.i_l. Node i_l may point to any of the nodes in the live set A, or be a leaf (if it points to 0). Each live node can point to any other live node which is greater than it (i.e. its serial number is greater) and which doesn't belong to its forbidden set of nodes. Or, a node in the live set may be a leaf.

3 Processor and Memory Cell Dependencies

In this section we bind the PJ_k function to a parallel computation in the EREW PRAM model. We define a few notions required for establishing the lower bound. A number of propositions is needed in order to explore the relations between these notions and their implications over the PJ_k computation.

Definition 3 *An algorithm AL is an* **EREW algorithm for computing** PJ_k*, if the following requirements hold for each input x of PJ_k:*
1. AL works according to the described EREW PRAM model.
2. The input of AL is set in n memory cells. The i'th cell contains x_i.
3. In the end of AL's execution, the value $PJ_k(x)$ is written in cell no. 1.

From now on we fix AL as an arbitrary EREW algorithm for computing PJ_k. All the following definitions and propositions relate to AL.

Definition 4 *A* **processor** p **(memory cell** m**) depends on an index** i **at time** t **in a restriction** ρ*, if there exists an input x such that $x, x^{i \to 0} \in \rho$ and $state_p(x,t) \neq state_p(x^{i \to 0}, t)$ ($content_m(x,t) \neq content_m(x^{i \to 0}, t)$).*

Definition 5 *A* **processor** p **(memory cell** m**) depends on** $i \to j$ **at time** t **in a restriction** ρ*, if there exists an input x such that $x^{i \to j}, x^{i \to 0} \in \rho$ and $state_p(x^{i \to j}, t) \neq state_p(x^{i \to 0}, t)$ ($content_m(x^{i \to j}, t) \neq content_m(x^{i \to 0}, t)$).*

We denote the set of processors which depend on an index i at time t in a restriction ρ by $P_{i,\rho}(t)$. Similarly, we define the notations $P_{i \to j, \rho}(t)$ (for dependency on $i \to j$) and $M_{i,\rho}(t)$, $M_{i \to j, \rho}(t)$ (for memory cell dependencies).

Since the four notions of *regular / $i \to j$ processor / memory cell dependencies* are very similar, most of the propositions which regard them have almost identical proofs. Therefore, unless specified otherwise, we will present proofs only for the *regular processor dependency* case.

Proposition 1 **a.** *Each processor which depends on i at time t, depends on it also at any time $t' > t$. The same for dependency on $i \to j$ and for memory cell dependencies.*

b. *Each processor which depends on i at time t, depends on $i \to j$ for some j. And vice versa, each processor which depends on $i \to j$ at time t, depends on i too. The same for memory cells.*

c. *Processors don't gain new dependencies at write steps, and memory cell don't gain new dependencies at read steps.*

d. *If a processor or a memory cell depends on $i \to j$ in a restriction ρ, then i and j are not constant elements of ρ and $i < j$.*

Proof:
a. Let $p \in P_{i,\rho}(t)$. There exists an input $x \in \rho$, such that $state_p(x,t) \neq state_p(x^{i\to 0},t)$. As mentioned before, we assume that once the states of a processor on two different inputs are different at a certain time, they stay different forever. In particular, $state_p(x,t') \neq state_p(x^{i\to 0},t')$. This means that $p \in P_{i,\rho}(t')$. Therefore, $P_{i,\rho}(t) \subseteq P_{i,\rho}(t')$.

b. Let $p \in P_{i,\rho}(t)$, and let $x \in \rho$ be the input, for which $state_p(x,t) \neq state_p(x^{i\to 0},t)$. Take $j = x_i$. Since $x = x^{i\to j}$, then $state_p(x^{i\to j},t) \neq state_p(x^{i\to 0},t)$. This means that $p \in P_{i\to j,\rho}(t)$.

For the opposite direction, let $p \in P_{i\to j,\rho}(t)$ and let x be the input for which $state_p(x^{i\to 0},t) \neq state_p(x^{i\to j},t)$. Consider the input $y = x^{i\to j}$. Clearly, $y = x^{i\to j} \in \rho$, $y^{i\to 0} = x^{i\to 0} \in \rho$ and $state_p(y^{i\to 0},t) = state_p(x^{i\to 0},t) \neq state_p(x^{i\to j},t) = state_p(y,t)$. Therefore, $p \in P_{i,\rho}(t)$.

c. Take a processor $p \notin P_{i,\rho}(t-1)$, where t is a write step. Therefore, $state_p(x,t-1) = state_p(x^{i\to 0},t-1)$ for any input $x \in \rho$. Since t is a write step, p doesn't read from any memory cell at that time on both inputs. Therefore, its state at time t is affected only by t and by its state at time $t-1$. Since both these parameters are the same on the inputs x and $x^{i\to 0}$, then $state_p(x,t) = state_p(x^{i\to 0},t)$. Therefore, $p \notin P_{i,\rho}(t)$.

d. Let $p \in P_{i\to j,\rho}(t)$, and let x be the input for which $state_p(x^{i\to j},t) \neq state_p(x^{i\to 0},t)$. Note, that $x_i^{i\to j} = j$ and $x^{i\to j} \in \rho$. Therefore, $j \in S_i$ (recall definition 2 of the restriction). Moreover, $0 \in S_i$ and $j \neq 0$. Hence, $|S_i| \geq 2$. It follows from definition 2 that $|S_k| = 1$ for any constant element k. Therefore, i is a live element or the last element in the path. In both cases $S_i \subseteq (A \cap \{i+1,\ldots,n\}) \cup \{0\}$. Since $j \neq 0$, then $j \in A \cap \{i+1,\ldots,n\}$. Therefore, j is a live element and $j > i$. •

Proposition 2 *Let ρ be a restriction, and let p be a processor (m be a memory cell) which doesn't depend on any index at time t in ρ. Then, $state_p(x,t) = state_p(y,t)$ ($content_m(x,t) = content_m(y,t)$) for each $x, y \in \rho$.*

Proof: Let x, y be two arbitrary inputs in ρ. We shall use a hybrid argument to show that $state_p(x,t) = state_p(y,t)$.

Arrange the set of non-constant elements of ρ ($A \cup \{i_l\}$) in an ascending order: $j_1, j_2, \ldots, j_k$. Define the following sequence of inputs (denoted by: $x(0), x(1), \ldots, x(k)$):

$$x(s)_i = \begin{cases} x_i & i \text{ } is \text{ } a \text{ } constant \text{ } element \text{ } of \text{ } \rho \\ 0 & i \in \{j_1,\ldots,j_s\} \\ x_i & i \in \{j_{s+1},\ldots,j_k\} \end{cases}$$

First we show that $x(s) \in \rho$ for every $0 \leq s \leq k$. It suffices to prove that $x(s)_i \in S_i$ for each $1 \leq i \leq n$. If i is a constant element of ρ or $i \in \{j_{s+1},\ldots,j_k\}$, then $x(s)_i = x_i \in S_i$, since x is in ρ. Otherwise, i is not a constant element, and in that case $x(s)_i = 0 \in S_i$.

By definition, it holds that $x(s) = x(s-1)^{j_s \to 0}$. p doesn't depend on any index at time t in ρ, and in particular it doesn't depend on j_s. Therefore, $state_p(x(s),t) = state_p(x(s)^{j_s \to 0},t) = state_p(x(s-1),t)$ for each $1 \le s \le k$. A simple induction shows that $state_p(x(0),t) = state_p(x(k),t)$. Note that $x(0) = x$. Therefore, $state_p(x,t) = state_p(x(k),t)$.

We can do the same with the input y, defining the corresponding inputs $y(0),\ldots,y(k)$. We will obtain $state_p(y,t) = state_p(y(k),t)$. However, it is easy to see that $x(k) = y(k)$. Therefore, $state_p(x,t) = state_p(x(k),t) = state_p(y(k),t) = state_p(y,t)$. •

Proposition 3 *Let ρ be a restriction, and let p be a processor (m be a memory cell), which depends on a single index j at time t in ρ. Then $state_p(x,t) = state_p(y,t)$ ($content_m(x,t) = content_m(y,t)$) for every two inputs $x,y \in \rho$, that satisfy $x_j = y_j$.*

Proof: The proof of this proposition resembles the proof of the previous one, hence we will stress the differences only.

First, we should observe (by proposition 1d) that j is either a live element of ρ or the last element in its path. We arrange the set of non-constant elements of ρ excluding j in an ascending order: $j_1,\ldots,j_k$. We define the inputs $x(0),\ldots,x(k)$ as follows:

$$x(s)_i = \begin{cases} x_i & i \text{ is a constant element of } \rho \\ x_i & i = j \\ 0 & i \in \{j_1,\ldots,j_s\} \\ x_i & i \in \{j_{s+1},\ldots,j_k\} \end{cases}$$

We skip the proof that $x(s) \in \rho$ for each s. $state_p(x(s),t) = state_p(x(s-1),t)$ for each $1 \le s \le k$, because $j_s \ne j$ and therefore p doesn't depend on j_s.

We can conclude that $state_p(x,t) = state_p(x(k),t)$ and that $state_p(y,t) = state_p(y(k),t)$. Since $x_j = y_j$, then $x(k) = y(k)$. Therefore, $state_p(x,t) = state_p(y,t)$. •

We would like to look into the relations between processors which access the same memory cell at the same time (on different inputs, of course). For this, we need the following definition:

Definition 6 *A* **processor** p **reads from (writes into) a memory cell** m **in a restriction** ρ **at time** t*, if there exists an input $x \in \rho$, such that p reads from (writes into) m on x at time t.*

Proposition 4 *Let ρ be a restriction, and let p_1, p_2 be two processors that depend on at most one index at time $t-1$ and that read from (write into) the same memory cell m at time t in ρ. Then, necessarily p_1 and p_2 depend on the same index at time $t-1$ in ρ.*

Proof: We prove the proposition for two processors which read from the same memory cell. We will need for that the exclusive read property of the algorithm. The proof of the second case, in which two processors write into the same memory cell is proven identically, using the exclusive write property.

Assume, to the contradiction, that p_1 and p_2 do not depend on the same index at time $t-1$. Our goal is to show that there exists an input on which both p_1 and p_2 read from m. This will be a contradiction to the exclusive read property of the algorithm.

By definition 6 there are two inputs $x,y \in \rho$ such that p_1 reads from m on x and p_2 reads from m on y at time t. If one of the processors doesn't depend on any index at time $t-1$ (WLOG, it is p_1) then by proposition 2 $state_{p_1}(x,t-1) = state_{p_1}(y,t-1)$. Therefore p_1 reads from m at time t on the input y too.

Otherwise, p_1 depends on i only at time $t-1$ and p_2 depends on j only at time $t-1$. Define $z = x^{j \to y_j}$. Clearly, $z \in \rho$, since $z_k \in S_k$ for each $1 \le k \le n$. It holds also that $z_i = x_i$ and $z_j = y_j$ (because $i \ne j$). Therefore, by proposition 3 $state_{p_1}(x,t-1) = state_{p_1}(z,t-1)$ and $state_{p_2}(y,t-1) = state_{p_2}(z,t-1)$. Since the state of p_1

is the same at time $t-1$ on the inputs x and z, it behaves the same on these inputs at time t. This means, that p_1 reads from m on z at time t. Similarly, p_2 reads from m on z at time t. •

Next we would like to examine the cases in which a processor or a memory cell gains a new dependency.

Definition 7 *A* **memory cell** m **causes a processor** p **to depend on** i **in a restriction** ρ **at time** t*, if the following hold:*
1. p doesn't depend on i at time $t-1$*.*
2. There exist inputs $x, x^{i\to 0} \in \rho$*, on which* $content_m(x, t-1) \neq content_m(x^{i\to 0}, t-1)$*, and p reads from* m *on both inputs.*

Definition 8 *A* **processor** p **causes a memory cell** m **to depend on** i **in a restriction** ρ **at time** t*, if the following hold:*
1. m *doesn't depend on i at time* $t-1$*.*
2. There exist inputs $x, x^{i\to 0} \in \rho$*, such that either p writes into* m *on one of the inputs but not on the other, or it writes on both inputs but different values.*

In the same manner we define the corresponding notions regarding dependencies on $i \to j$.

Proposition 5 **a.** *A processor p gains a dependency on i at time t in a restriction* ρ*, iff there exists a memory cell* m *which causes it to depend on i at that time.*

b. *Each processor that causes a memory cell to depend on i at time t in* ρ *depends on i at time* $t-1$*.*

The same for dependency on $i \to j$ *and for memory cell dependencies.*

Proof:
a. Let $p \in P_{i,\rho}(t) \setminus P_{i,\rho}(t-1)$ and let $x \in \rho$ be the input for which $state_p(x,t) \neq state_p(x^{i\to 0}, t)$. Since p doesn't depend on i at time $t-1$, then $state_p(x, t-1) = state_p(x^{i\to 0}, t-1)$. The state of p at time t is a function only of t, of its state at time $t-1$ and of the data it reads at step t. Since the two first parameters are the same on both inputs x and $x^{i\to 0}$, then p necessarily reads from some memory cell m at time t on at least one of the two inputs. Since the state of a processor at time $t-1$ determines its action at time t, then p reads from m on both inputs. If the contents of m is the same on the two inputs, then p would stay in the same state after the read operation. Therefore, $content_m(x, t-1) \neq content_m(x^{i\to 0}, t-1)$. This means that m causes p to depend on i at time t.

The other direction: assume now that $p \notin P_{i,\rho}(t-1)$ and that there exists a memory cell m which causes p to depend on i at time t. By definition, there exists an input x such that $x, x^{i\to 0} \in \rho$, p reads from m on both inputs and $content_m(x, t-1) \neq content_m(x^{i\to 0}, t-1)$. Hence, p reads different values at time t on the two inputs. A part of a processor's state is the input it reads. Therefore, necessarily $state_p(x,t) \neq state_p(x^{i\to 0}, t)$. This means that $p \in P_{i,\rho}(t)$.

The proof for memory cells is a slight different. A similar argument shows that for each memory cell m which gains a dependency on i at time t there exists a processor p which writes into it at time t on at least one of the inputs x or $x^{i\to 0}$. Now, if p writes to m on only one of them, then p is considered to cause m to depend on i at time t. Otherwise, p writes to m on both inputs. If it wrote the same values, then the contents of m would have stayed the same on both inputs. Therefore, p should write different values to m. In this case too p causes m to depend on i at time t.

Assume that $m \notin M_{i,\rho}(t-1)$ and that there exists a processor p which causes m to depend on i at time t. By definition, there exists an input x and a processor p such that $x, x^{i\to 0} \in \rho$ and p either writes into m on one of the inputs but not on the other, or p writes to m different values on the two inputs. Since a memory cell "remembers" all the writings made in it and their time steps, then in any of the above cases the contents of m after time t is different on the two inputs. This means that $m \in M_{i,\rho}(t)$.

b. Let p be a processor which causes a memory cell m to depend on i at time t in ρ. By definition 8, there exists an input x such that $x, x^{i\to 0} \in \rho$ and p behaves differently on these inputs at time t. Since the behavior of a processor at a certain time is determined uniquely by its state in the previous step, then $state_p(x, t-1) \neq state_p(x^{i\to 0}, t-1)$. This means that $p \in P_{i,\rho}(t-1)$.

Let m be a memory cell which causes a processor p to depend on i at time t in ρ. By definition 7, there exists an input x such that $x, x^{i\to 0} \in \rho$ and $content_m(x, t-1) \neq content_m(x^{i\to 0}, t-1)$. This means that $m \in M_{i,\rho}(t-1)$. •

In order to control the knowledge of processors during the algorithm, we would like to produce a restriction which enables the minimal number of dependencies as possible. We call this type of restrictions *clean restrictions*.

Definition 9 *A restriction ρ is called a* **clean restriction at time t**, *if every processor and every memory cell depends on at most one index at time t in ρ.*

We would like to measure the strength of a clean restriction in limiting the number of processor and memory cell dependencies. We will need the next lemma to achieve this.

Lemma 1 *Let ρ be a clean restriction at time t. Then for each pair of indices i, j it holds that: $|P_{i\to j,\rho}(t)| \leq |P_{i\to j,\rho}(t-1)| + |M_{i\to j,\rho}(t-1)|$ and $|M_{i\to j,\rho}(t)| \leq |M_{i\to j,\rho}(t-1)| + 2|P_{i\to j,\rho}(t-1)|$.*

Proof: We begin with the first inequality. If t is a write step, then no processor gains a new dependency which means that $P_{i\to j,\rho}(t) = P_{i\to j,\rho}(t-1)$, and the first inequality follows. Hence, we assume that t is a read step.

The processors that depend on $i \to j$ at time t divide to those which depended on it at time $t-1$ ($P_{i\to j,\rho}(t-1)$) and those which gained the dependency on $i \to j$ at time t. We will show that the number of the last ones is bounded by $|M_{i\to j,\rho}(t-1)|$.

Let us fix a processor p, that gains a dependency on $i \to j$ at time t. p can be related to the memory cell $m \in M_{i\to j,\rho}(t-1)$ that caused its dependency. p reads from m on an input, whose i'th component is j. Since p depends only on i at time t and before (recall that ρ is clean at time t), it behaves the same on all the inputs that share the same i'th component. Therefore, p reads from m on all the inputs that have j in their i'th component. This prevents any other processor from reading from m at time t on such inputs, and therefore from gaining a dependency on $i \to j$ from m. It follows that the number of processors that gain a dependency on $i \to j$ at time t is no more than the number of memory cells that depend on $i \to j$ at time $t-1$.

For proving the second inequality, it is enough to observe that the number of memory cells that gain a dependency on $i \to j$ at a write step is bounded by twice the number of processors that depended on $i \to j$ at the previous step.

Let us fix a memory cell m, which gains a dependency on $i \to j$ at time t. m can be related to a processor p in $P_{i\to j,\rho}(t-1)$, that causes this dependency. We should prove that p could not have caused more than two dependencies. p writes to the memory cells it causes to depend on $i \to j$ on inputs whose i'th component is 0 or j. Since p depends on i only at time $t-1$, it behaves identically on all the inputs that share their i'th component. Therefore, p can write into two memory cells at the most on inputs that have 0 or j in their i'th component. This means, that p can cause two memory cells at the most to depend on $i \to j$. •

We obtain as a corollary the main result of this section:

Corollary 1 *Let ρ be a clean restriction at time t. Then for each pair of indices i, j it holds that: $|P_{i\to j,\rho}(t)| \leq 3^t$ and $|M_{i\to j,\rho}(t)| \leq 3^t$.*

4 Bad Triplets

Our goal is to produce a clean restriction for every given EREW algorithm, which computes the PJ_k function. This clean restriction will enable us to find inputs on which the algorithm runs slowly.

The clean restrictions will be produced in an iterative process. In each iteration we extract a new restriction from the previous one, such that the new restriction is clean for one more step. In order to do this, we need to develop a tool for identifying the "bad" elements of each restriction. Removing these elements would enable us to extend the "cleanness" of the restriction for further time steps. The *bad triplets* notion, defined below, fulfills exactly this purpose.

The intuition behind the definition of bad triplets is as follows: we have a restriction ρ, which is clean at time t, and we would like to keep it clean at time $t+1$ as well. For accomplishing that, we should prevent any processor or memory cell from having more than one dependency at time $t+1$. If there is a processor/memory cell that depends on i at time $t+1$ and gains a dependency on $j \to k$ at time $t+1$, then the triplet (i, j, k) is considered to be a bad triplet. If we either "kill" one of the indices i, j or k (i.e. make them dead elements) or add k to the forbidden set of j, then (i, j, k) will no longer be a bad triplet. A slight complication appears when dealing with triplets of the form (i, j, i) in the two lemmas below. Therefore, we need to define the second type of bad triplets: triplets (i, j, k) that specify a processor / memory cell which depends on $i \to j$ at time t and gains a dependency on k at time $t+1$.

Definition 10 *Let ρ be a clean restriction at time $t-1$. A triplet of indices (i, j, k) $(i \neq j,\ i \neq k,\ j \neq k)$ is called* **a bad triplet of ρ at time t of type 1**, *if there exists a processor (when t is a read step) or a memory cell (when t is a write step) which depends on i at time $t-1$ and gains a dependency on $j \to k$ at time t.*

We say that (i, j, k) is **a bad triplet of type 2** *(at a read step), if there exists a processor which depends on $i \to j$ at time $t-1$, it gains a dependency on k at time t from a memory cell m, and it reads from m on an input $x \in \rho$, whose i'th component is 0 or j.*

(i, j, k) is **a bad triplet of type 2** *(at a write step), if there exists a memory cell which depends on $i \to j$ at time $t-1$ and it gains a dependency on k at time t.*

We say that (i, j, k) is **a bad triplet**, *if it is either a bad triplet of type 1 or bad triplet of type 2.*

Lemma 2 *Let ρ be a clean restriction at time $t-1$. If ρ doesn't have any bad triplets at time t, then ρ is clean at time t.*

Proof: We separate to cases:
Case 1: t is a read step

First, recall that memory cells don't gain new dependencies at read steps. Therefore, since each memory cell depends on at most one index at time $t-1$ in ρ (because ρ is clean at that time), then each memory cell still depends on at most one index at time t.

Suppose that ρ is not clean at time t. Then, there should exist a processor p which depends on two different indices at time t in ρ. p depends on at most one index at time $t-1$, because ρ was clean at that time. We separate to cases:

Case 1.1: p depends on an index i at time $t-1$ in ρ.

Since p depends on i at time $t-1$, it depends on it at time t too. We denote the other index, on which p depends at time t, by j. There exists some index k such that p depends on $j \to k$. p did not depend on j at time $t-1$, therefore it didn't depend on $j \to k$ at that time as well. We conclude that p gains a dependency on $j \to k$ at time t. Let m be the memory cell, which causes this dependency.

Consider the three indices i, j, k. Clearly, $i \neq j$ from the choice of p. $j \neq k$, because p depends on $j \to k$ at time t (which means that $j < k$). We are left with two possibilities:

1. $i \neq k$. Then, the triplet (i, j, k) is a bad triplet of type 1.
2. $i = k$. p depends on i at time $t-1$ in ρ. Therefore, p depends on $i \to r$ at time $t-1$ for some index r. Note that, $r > i > j$ (because p depends on $i \to r$ and on $j \to i$). In particular, $r \neq i$ and $r \neq j$. Let us denote the input on which p reads from m by x. If p reads from m on $x^{i\to 0}$ too, then the triplet (i, r, j) is a bad triplet of type 2. Otherwise, let us denote: $x_i = l$. Since p behaves differently on $x^{i\to l}$ and on $x^{i\to 0}$ at time t, then p depends on $i \to l$ at time $t-1$ in ρ. Again, $l > i > j$, and therefore the triplet (i, l, j) is a bad triplet of type 2.

We found a bad triplet of ρ at time t in each of the possibilities. This contradicts our assumption that ρ does not have such triplets.

Case 1.2: p doesn't depend on any index at time $t-1$ in ρ.

By proposition 2 $state_p(x, t-1) = state_p(y, t-1)$ for each two inputs $x, y \in \rho$. Thus, the behavior of p at time t should be the same for all the inputs in ρ (because the behavior of a processor at time t derives directly from its state at time $t-1$).

p depends on two different indices i and j at time t in ρ. Therefore, by proposition 5a there exist two memory cells m_1 and m_2 which cause p to depend on i and j respectively at time t. This means that p reads from m_1 on some input $x \in \rho$ and from m_2 on some input $y \in \rho$. By proposition 5b m_1 depends on i and m_2 depends on j at time $t-1$. Then, necessarily $m_1 \neq m_2$, because ρ is clean at time $t-1$. We see that p behaves differently on the inputs x and y. This is a contradiction to what we have observed before.

Case 2: t is a write step

The proof here is very similar to what has been done above, thus we will concentrate on the differences only.

First, let us consider case 2.1, in which we have a memory cell m that depends on i at time $t-1$ in ρ, and a processor p which causes m to depend on $j \to i$ at time t. Necessarily, m depends on $i \to r$ for some index r at time $t-1$. It holds that $r > i > j$. Therefore, the triplet (i, r, j) is a bad triplet of type 2.

Let us consider now case 2.2, which deals with a memory cell m that doesn't depend on any index at time $t-1$ in ρ, but depends on two different indices i and j at time t.

m depends on two different indices i and j at time t in ρ. Therefore, by proposition 5a there exist two processors p_1 and p_2 which cause m to depend on i and j respectively at time t. This means that p_1 writes into m on some input $x \in \rho$ and p_2 writes into m on some input $y \in \rho$. By proposition 5b p_1 depends on i and p_2 depends on j at time $t-1$. Then, necessarily $p_1 \neq p_2$, because ρ is clean at time $t-1$.

p_1, p_2 are both processors which depend on only one index at time $t-1$ in ρ and write into the same memory cell at time t in ρ. It follows from proposition 4 that p_1 and p_2 depend on the same index at time $t-1$, i.e. $i = j$. This contradicts our assumption. •

Lemma 3 *Let ρ be a clean restriction at time t. Let E be the set of bad triplets of ρ at time $t+1$. Then, it holds that: $|E| \leq 3^{t+1}(|A|+1)^2$. (Recall that A is the set of live elements in ρ).*

Proof: Let us denote the set $A \cup \{i_l\}$ by A'. Note that for every bad triplet (i, j, k), it holds that $i, j, k \in A'$. This follows from the fact that there are processors or memory cells which depend on i and on $j \to k$ or on $i \to j$ and on k (which means that i, j, k are non-constant elements of the restriction).

Case 1: $t+1$ is a read step.

Let us consider a pair $j, k \in A'$. Each of the bad triplets of type 1, whose two last components are j and k, is caused by a memory cell which depends on $j \to k$ at time t. By corollary 1 there are at most 3^t such memory cells. Each of these memory cells can contribute at most one bad triplet, since all the processors which read from one memory cell at time t depend on the same index (see proposition 4). Therefore, there are at most $3^t|A'|^2$ bad triplets of type 1.

Let us consider now a pair $i, j \in A'$. Each of the bad triplets of type 2, which their first two components are i and j, is caused by a processor which depends on $i \to j$ at time t and gains a new dependency at time $t+1$. By corollary

1 there are at most 3^t such processors. Let us fix a processor p, and show it contributes two bad triplets at the most. According to the definition of bad triplets of type 2, p reads from the memory cell that causes its new dependency on an input whose i'th component is 0 or j. Since p depends only on i at time t, it behaves the same on all the inputs that share the same i'th component. Therefore, p may read from two memory cells at the most (one on the inputs, whose i'th component is 0, and one on the inputs whose i'th component is j). This means that p is involved in at most two bad triplets of type 2, whose first two component are i and j. We conclude that the number of bad triplets of type 2 is not more than $2 \cdot 3^t |A'|^2$.

Now, the total number of bad triplets is bounded by $3 \cdot 3^t |A'|^2 = 3^{t+1}(|A|+1)^2$.

Case 2: t+1 is a write step

Let us fix a pair $j, k \in A'$. Each bad triplet of type 1, whose last two components are j and k, is caused by a processor which depends on $j \to k$ at time t. We fix such a processor p. In order to cause a memory cell to depend on $j \to k$ at time $t+1$, p should write into it on an input whose j'th component is 0 and/or on an input whose j'th component is k. Since p depends only on j at time t, it acts the same on all the inputs that share the same j'th component. Therefore, p can write into two cells at the most (one on the inputs whose j'th component is 0, and one on the inputs whose j'th component is k). This means, that p can cause two memory cells at the most to gain a dependency on $j \to k$. Since there are at most 3^t processors that depend on $j \to k$ at time t, then the number of bad triplets of type 1 is bounded by $2 \cdot 3^t |A'|^2$.

Let us consider now a pair $i, j \in A'$. Each bad triplet of type 2, whose first two components are i and j is caused by a memory cell which depends on $i \to j$ at time t, and gains a new dependency at time $t+1$. Let us fix such a memory cell m. Since all the processors that write into m at time $t+1$ depend on the same index (proposition 4), then m can gain only one new dependency. Therefore, m is involved in at most one bad triplet of type 2 whose first two components are i and j. Again, since there are at most 3^t memory cells that depend on $i \to j$ at time t, then the number of bad triplets of type 2 is bounded by $3^t |A'|^2$.

Therefore, in this case too the number of bad triplets is not more than $3^{t+1}(|A|+1)^2$. •

5 The Main Lemma

Our objective is to prove that for each time step t, which is small enough, there exists a clean restriction ρ_t, which contains an input that AL cannot compute fast. We show that if we are given a restriction ρ_t which is clean at time t, we can produce a new restriction ρ_{t+1} which is clean at time $t+1$. We then infer, by induction on t, the existence of such restrictions.

Let us make a few notations regarding the restriction ρ_t (recall definition 2):

1. $\rho_t = S_1^t \times \cdots S_n^t$
2. The path of ρ_t is P_t.
3. The set of live elements of ρ_t is A_t, and $n_t = |A_t|$.
4. The forbidden set of an index i is denoted by D_i^t.
5. $\delta = \frac{1}{2^{\sqrt{\log n}}}$

Lemma 4 *Let ρ_t be a clean restriction at time t, for which $n_t \geq \frac{4 \cdot 3^{t+4}}{\delta}$ and $def(\rho_t) \leq \frac{1}{108} - 2\delta$. Then, there exists a restriction ρ_{t+1} which is clean at time $t+1$ and satisfies:*

1. $len(\rho_{t+1}) = len(\rho_t) + 1$.
2. $n_{t+1} \geq \frac{\delta n_t}{3^{t+5}}$.
3. $def(\rho_{t+1}) \leq 54 def(\rho_t) + 108\delta$.

Proof: The basic idea is to kill at random a subset of the live elements. This will eliminate most bad triplets, namely, those which contain at least one of the killed elements. Unfortunately, some bad triplets may remain, which we will eliminate by enlarging some of the forbidden sets. Once all bad triplets are eliminated, we are assured that the restriction is indeed clean.

Let B be any subset of A_t. For each $j \in B$ we denote by $BAD_1(j, B)$ the set of k's in B, for which there exists some $i \in B$ such that (i, j, k) is a bad triplet of type 1 at time $t + 1$. Similarly, we denote by $BAD_2(j, B)$ the set of k's in B for which there exists an $i \in B$, such that (j, k, i) is a bad triplet of type 2 at time $t + 1$. Finally, we denote by $BAD(j, B)$ the union of $BAD_1(j, B)$ and $BAD_2(j, B)$.

$BAD(j, B)$ denotes the set of indices we would like to add to the forbidden set of j, in order to prevent bad triplets involving j. Notice the differences in the indices positions between the definition of $BAD_1(j, B)$ and the definition of $BAD_2(j, B)$ (they are a result of the difference between triplets of type 1 and triplets of type 2).

Claim 1 *There exists a subset B of A_t such that:*

1. $|B| \geq \frac{\delta n_t}{3^{t+4}}$
2. $|BAD(j, B)| \leq 54\delta|B|$ *for each* $j \in B$.
3. $\frac{|D_j^t \cap B|}{|B|} \leq 27def(\rho_t)$ *for each* $j \in B$.

Proof: We denote by F the set of bad triplets of ρ_t at time $t + 1$, and by E the set $A_t^3 \cap F$ (E doesn't include bad triplets which involve the last element in the path). We choose at random a subset $T \subseteq A_t$ of size $\frac{\delta n_t}{3^{t+3}}$, and define two random variables: $b_T = |T^3 \cap E|$ and $d_T = \sum_{j \in T} \frac{|D_j^t \cap T|}{|T|}$. By lemma 3 $|F| \leq 3^{t+2}(n_t + 1)^2$. Since $n_t \geq \frac{4 \cdot 3^{t+4}}{\delta} > 4$, then $(n_t + 1)^2 \leq 2n_t^2$. Therefore, $|F| \leq 3^{t+3}n_t^2$ and $Exp(b_T) \leq |F|\frac{|T|^3}{|A_t|^3} \leq \delta|T|^2$. Likewise, $Exp(d_T) = \frac{1}{|T|}\sum_{j \in T} Exp(|D_j^t \cap T|) \leq \frac{1}{|T|}\sum_{j \in T} |D_j^t|\frac{|T|}{|A_t|} \leq \frac{1}{|T|}|T|^2 def(\rho_t) = |T|def(\rho_t)$.

It follows by Markov inequality, that for some choice T' of T, it holds that: $b_{T'} \leq 3Exp(b_T)$ and $d_{T'} \leq 3Exp(d_T)$. Let us define the following sets:

$$T_1' = \{j \in T' \mid |BAD(j, T')| > 18\delta|T'|\}$$

$$T_2' = \{j \in T' \mid \frac{|D_j^t \cap T'|}{|T'|} > 9def(\rho_t)\}$$

$$B = T' \setminus (T_1' \cup T_2')$$

We would like to show that $|B| \geq \frac{|T'|}{3}$. Assume this is not true. Then, either $|T_1'| > \frac{|T'|}{3}$ or $|T_2'| > \frac{|T'|}{3}$. Observe that since each bad triplet (i, j, k) can "contribute" two elements at the most to $\bigcup_{j \in T'} BAD(j, T')$ (to $BAD(j, T')$ if it is of type 1, and to $BAD(i, T')$ if it is of type 2), then $|T'^3 \cap E| \geq \frac{1}{2}\sum_{j \in T'} |BAD(j, T')|$. Therefore, if $|T_1'| > \frac{|T'|}{3}$ then $b_{T'} = |T'^3 \cap E| \geq \frac{1}{2}\sum_{j \in T'} |BAD(j, T')| > \frac{1}{2}|T_1'| \cdot 18\delta|T'| > 3\delta|T'|^2 \geq 3Exp(b_T)$. This contradicts the choice of T'. If $|T_2'| > \frac{|T'|}{3}$, then $d_{T'} = \sum_{j \in T'} \frac{|D_j^t \cap T'|}{|T'|} > |T_2'| \cdot 9def(\rho_t) > 3|T'|def(\rho_t) \geq 3Exp(d_T)$. Again, this is a contradiction to the choice of T'. We conclude that $|B| \geq \frac{|T'|}{3} = \frac{\delta n_t}{3^{t+4}}$.

Each $j \in B$ doesn't belong to T_1'. Therefore, $|BAD(j, T')| \leq 18\delta|T'|$. Clearly, $BAD(j, B) \subseteq BAD(j, T')$. $\Rightarrow$ $|BAD(j, B)| \leq 18\delta|T'| \leq 54\delta|B|$. Each $j \in B$ doesn't belong to T_2' either. Therefore, $\frac{|D_j^t \cap T'|}{|T'|} \leq 9def(\rho_t)$. $D_j^t \cap B \subseteq D_j^t \cap T'$, hence: $\frac{|D_j^t \cap B|}{|B|} \leq \frac{|D_j^t \cap T'|}{\frac{|T'|}{3}} \leq 27def(\rho_t)$. The claim follows from these three properties of B. •

We choose i_{l+1} to be the minimal element in B, and define the following set:

$$C = B \setminus (\{i_{l+1}\} \cup BAD(i_{l+1}, B) \cup (D_{i_{l+1}}^t \cap B))$$

We will show next that C is still large enough. $|C| = |B \setminus (\{i_{l+1}\} \cup BAD(i_{l+1}, B) \cup (D^t_{i_{l+1}} \cap B))| \geq |B| - 1 - 54\delta|B| - 27|B|def(\rho_t) = |B| \cdot (1 - 27(2\delta + def(\rho_t))) - 1$. It is given that $def(\rho_t) \leq \frac{1}{108} - 2\delta$. Therefore, $27(2\delta + def(\rho_t)) \leq \frac{1}{4} \Rightarrow (1 - 27(2\delta + def(\rho_t))) \geq \frac{3}{4}$. On the other hand, it is given that $n_t \geq \frac{4 \cdot 3^{t+4}}{\delta}$. Hence, $|B| \geq \frac{\delta n_t}{3^{t+4}} \geq 4$. From these two facts we obtain: $|C| \geq \frac{3}{4}|B| - \frac{1}{4}|B| = \frac{1}{2}|B|$.

We are now ready to define ρ_{t+1}. ρ_{t+1} has a path of length $l + 1$. The first l elements of the path are $i_0, \ldots, i_l$ (the path of ρ_t). The last element is i_{l+1} (the minimal element in B). The set of live elements in ρ_{t+1} is: $A_{t+1} = C$. For each $i \in A_{t+1}$ we define: $D^{t+1}_i = (A_{t+1} \cap D^t_i) \cup (A_{t+1} \cap BAD(i, B))$. We leave it for the reader to verify that ρ_{t+1} is defined properly and that it is a subset of ρ_t.

Let us show that ρ_{t+1} satisfies all the requirements in the lemma: $len(\rho_{t+1}) = len(\rho_t) + 1$, since we added precisely one element to the path of ρ_t to produce the path of ρ_{t+1}. From claim 1 and from the fact that $|C| \geq \frac{1}{2}|B|$, it follows that $n_{t+1} = |C| \geq \frac{1}{2} \cdot \frac{\delta n_t}{3^{t+4}} \geq \frac{\delta n_t}{3^{t+5}}$ and that $def(\rho_{t+1}) \leq \max_{i \in A_{t+1}} \frac{|A_{t+1} \cap D^t_i| + |A_{t+1} \cap BAD(i,B)|}{n_{t+1}} \leq \max_{i \in A_{t+1}} \frac{|B \cap D^t_i| + |BAD(i,B)|}{\frac{1}{2}|B|} \leq \max_{i \in A_{t+1}} \frac{27|B|def(\rho_t) + 54\delta|B|}{\frac{1}{2}|B|} = 54 def(\rho_t) + 108\delta$.

It is left to observe that ρ_{t+1} is clean at time $t+1$. Since $\rho_{t+1} \subseteq \rho_t$, each processor or memory cell which depends on an index i at time t in ρ_{t+1}, also depends on the same index in ρ_t. Therefore, since ρ_t is clean at time t, then so is ρ_{t+1}.

By lemma 2, it suffices to show that ρ_{t+1} doesn't have any bad triplets at time $t + 1$, in order to prove that it is clean at that time. Let us assume, to the contradiction, that ρ_{t+1} has a bad triplet (i, j, k) at time $t + 1$. As mentioned before, all the components of bad triplets are non-constant elements of the restriction. Therefore, $i, j, k \in A_{t+1} \cup \{i_{l+1}\} \subseteq B$. Let us separate to cases:

Case 1: (i, j, k) is a bad triplet of type 1.

Therefore, $k \in BAD_1(j, B) \subseteq BAD(j, B)$. It means that there exists a processor or a memory cell which depends on $j \to k$. Therefore, $k \in S^{t+1}_j$, and $k > j$. Necessarily $k \neq i_{l+1}$, because i_{l+1} is smaller than all the members of A_{t+1}. On the other hand, if $j = i_{l+1}$ then $k \in A_{t+1} \cap BAD(i_{l+1}, B)$. This contradicts the definition of A_{t+1}. We can conclude that $j, k \in A_{t+1}$. Therefore, $k \in D^{t+1}_j$. It follows that: $k \in S^{t+1}_j \cap D^{t+1}_j$. This is a contradiction to the definition of S^{t+1}_j.

Case 2: (i, j, k) is a bad triplet of type 2.

Hence, $j \in BAD_2(i, B) \subseteq BAD(i, B)$. In a similar way, it follows that $j \in S^{t+1}_i \cap D^{t+1}_i$, in contradiction to the definition of S^{t+1}_i.

Since all the requirements are satisfied, the proof of the lemma is complete. •

Corollary 2 *For each n which is large enough and for each time step $t \leq \frac{1}{100}\sqrt{\log n}$ there exists a restriction ρ_t for PJ_k, which is clean at time t, has at least 4 live elements, its length is t and its deficiency is at most $\frac{1}{2}$.*

Proof: The proof goes by induction on t using lemma 4. We show that there exists a restriction that satisfies the above requirements plus the following two:

1. $n_t \geq \frac{\delta^t(n-1)}{3^{\frac{t(t+9)}{2}}}$
2. $def(\rho_t) \leq \frac{108\delta}{53}(54^t - 1)$

For $t = 0$ we choose as ρ_0 all the inputs in which every node points only to greater nodes. This is exactly the restriction which is defined by the path $P_0 = \{1\}$, live set $A_0 = \{2, \ldots, n\}$ and forbidden sets $D^0_i = \phi$ for each $i \in A_0$. It is easy to see that ρ_0 is a legal restriction and that it satisfies all the requirements. The only non trivial fact is that ρ_0 is clean at time 0. But since every restriction is clean at time 0 (only the memory cells which contain the input components depend on anything, and they all depend on exactly one input index), so is ρ_0.

For the induction step, assume ρ_t satisfies the above requirements for time step t. We should prove that if $t+1 \leq \frac{1}{100}\sqrt{\log n}$, then there exists a restriction ρ_{t+1}, which satisfies the requirements for time step $t+1$. The proof of this step requires the following inequalities, which hold for a sufficiently large n and for $t \leq \frac{1}{100}\sqrt{\log n}$. We leave it for the reader to verify their correctness:

1. $t \leq \sqrt{\log_3 \sqrt{n-1}} - 4$
2. $t \leq \log_{\frac{1}{\delta}} \sqrt{n-1} - 1$
3. $t \leq \log_{54} \frac{53}{108^2 \delta}$

By the induction hypothesis ρ_t is clean at time t. Likewise, $n_t \geq \frac{\delta^t(n-1)}{3^{\frac{t(t+9)}{2}}} \geq \frac{4 \cdot 3^{t+4}}{\delta}$ and $def(\rho_t) \leq \frac{108\delta}{53}(54^t - 1) \leq \frac{1}{108} - 2\delta$. The last two statements follow from the above inequalities.

Therefore, using lemma 4, there exists $\rho_{t+1} \subseteq \rho_t$, which is clean at time $t+1$ and $len(\rho_{t+1}) = len(\rho_t)+1 = t+1$, $n_{t+1} \geq \frac{\delta n_t}{3^{t+5}} \geq \frac{\delta^{t+1}(n-1)}{3^{\frac{(t+1)(t+10)}{2}}}$ and $def(\rho_{t+1}) \leq 54 def(\rho_t) + 108\delta \leq \frac{108\delta}{53}(54^{t+1} - 1)$.

Since $t+1$ satisfies the requirements of the above inequalities, it follows that $n_{t+1} \geq \frac{4 \cdot 3^{t+5}}{\delta} \geq 4$ and $def(\rho_{t+1}) \leq \frac{1}{108} - 2\delta \leq \frac{1}{2}$. This completes the proof of the induction. •

6 The Lower Bound

Theorem 1 *Let AL be any $EREW$ algorithm for computing the PJ_k function, where n is sufficiently large and $k \leq \frac{1}{100}\sqrt{\log n}$. Let us denote by $T(AL)$ the time complexity of AL. Then, $T(AL) \geq k-1$.*

Proof: We shall prove that there exists an input x, on which AL cannot compute $PJ_k(x)$ by time $k-2$. It follows that $T(AL) \geq k-1$.

Let us consider time step $t = k-2$. Since n is large enough, then it follows from corollary 2 that there exists a restriction ρ which is clean at time t, has at least 4 live elements, has a path of length t and its deficiency is less than $\frac{1}{2}$.

Let j be the minimal live element of ρ. Our first objective is to prove that j points to another live element. Recall that the set of live elements to which j points are all the ones which are greater than it but are not included in its forbidden set. Since all the other live elements are greater than j, it suffices to show that the size of its forbidden set is less than the size of the live set minus 1. The deficiency of ρ is at most $\frac{1}{2}$, therefore the size of each forbidden set is bounded by half the size of the live set. This is sufficient for our needs, since the size of the live set is at least 4.

We are now ready to choose an input x from ρ on which AL may not be able to compute PJ_k by time t. Let us denote the live element to which j points by l. x would be the input in which the last element of the path (denoted by i_t) points to j, j points to l and l is a leaf (points to 0). Assume, to the contradiction, that the algorithm terminates its execution on x at time $t' \leq t$ and that it has written $PJ_k(x)$ to memory cell m ($m = 1$) by that time. We separate to cases:

Case 1: m depends on i_t at time t in ρ.

Let us denote the input $x^{j \to 0}$ by y. y belongs to ρ, since j is a live element. If the algorithm doesn't terminate its execution on y by time t, then we have found an input as wanted. Otherwise, by time t the algorithm terminates and writes $PJ_k(y)$ to m.

It holds that $x_{i_t} = y_{i_t}$ (since $i_t \neq j$). It follows from proposition 3 that $content_m(x,t) = content_m(y,t)$. Therefore, $PJ_k(x) = PJ_k(y)$ (because m contains $PJ_k(x)$ at time t on the input x and $PJ_k(y)$ at the same time on the input y). However, $PJ_k(x) = l$, $PJ_k(y) = 0$ and $l \neq 0$.

Case 2: m does not depend on i_t at time t in ρ.

Let y be the input $x^{i_t \to l}$. y belongs to ρ, since the last element in the path (i_t) may point to any live element. If the algorithm doesn't terminate its execution on y by time t, then we are done. Otherwise, by time t the algorithm terminates and writes $PJ_k(y)$ to m.

It holds that $x_i = y_i$ for each $i \neq i_t$. There are two possibilities: either m depends on an index $i \neq i_t$ or that it doesn't depend on any index at time t in ρ. In both cases, by propositions 2 and 3 $\mathit{content}_m(x,t) = \mathit{content}_m(y,t)$. Hence, since m consists of $PJ_k(x)$ on the input x at time t, so it does on the input y. It derives that $PJ_k(x) = PJ_k(y)$, but $PJ_k(x) = l$, $PJ_k(y) = 0$ and $l \neq 0$.

We can sum up, that in either case we reach a contradiction when assuming that the algorithm terminates its execution on all inputs by time t. Therefore, there exists an input, which its computation time is at least $t+1$, that is $k-1$. This means that $T(AL) \geq k-1$, as wanted. •

Theorem 2 *There exists a boolean function which can be computed in $O(\log\log n)$ time on a CREW PRAM, but requires $\Omega(\sqrt{\log n})$ time on a EREW PRAM.*

Proof: First, we should note here that the input of a EREW algorithm computing a boolean function is arranged one bit per memory cell. The output is written to memory cell no. 1, as usual.

We choose the PJ_k function as in theorem 1 (i.e. n is large enough and $k \leq \frac{1}{100}\sqrt{\log n}$). We define a boolean function f, which is based on the PJ_k function. The input of f is the input of PJ_k encoded to binary ($n\lceil \log n \rceil$ bits) plus an index of the output bit of PJ_k we would like to get as a result (another $\lceil \log \lceil \log n \rceil \rceil$ bits). The output of f is the bit in the requested place. Therefore, $f : \{0,1\}^N \to \{0,1\}$, where $N = n\lceil \log n \rceil + \lceil \log \lceil \log n \rceil \rceil$.

A CREW algorithm for computing f:

1. Gather in parallel each $\lceil \log n \rceil$ tuple of the first $n\lceil \log n \rceil$ input bits into one memory cell. After this operation, we have n cells, each one contains a single node of the PJ_k input graph.

2. Compute the PJ_k function using the well known pointer doubling technique.

3. Gather the last $\lceil \log \lceil \log n \rceil \rceil$ bits of the input into one cell. Now this cell contains the binary encoding of the index of the output bit. Extract this bit from the PJ_k output, and write it into cell no. 1.

Stage 1 takes in CREW $O(\log\log n)$ time steps. Stage 2 takes $O(\log k)$ steps, which is $O(\log\log n)$, since $k < \log n$. Stage 3 takes $O(\log\log\log n)$. We sum up, that $CREW(f) = O(\log\log n) = O(\log\log N)$.

Let AL_f be a EREW algorithm for computing f in time complexity T. We will present a EREW algorithm for computing PJ_k using AL_f at time $T + O(\log\log n)$:

1. Make $\lceil \log n \rceil$ copies of the input (i.e. disperse each input index to $\lceil \log n \rceil$ memory cells).

2. Encode each input component to binary, and disperse its bits into $\lceil \log n \rceil$ cells, each one containing one bit. (Do it for all the copies you have made in stage 1 in parallel).

3. Produce for each copy one index in the range $0..\lceil \log n \rceil - 1$. Encode these indices to binary and disperse them one bit per memory cell. At the end of this stage each copy is a legal input for AL_f.

4. Compute in parallel the value of f over all the copies of the input, using AL_f.

5. Gather the outputs obtained in the previous stage into one cell. The result is the binary encoding of the output of PJ_k.

Complexity analysis: Making m copies of data in EREW takes $O(\log m)$ steps in the trivial algorithm (duplicate the data, then duplicate each copy, etc.). Therefore, stages 1 and 2 take $O(\log\log n)$ steps and stage 3 takes $O(\log\log\log n)$ steps. Gathering of m data items into one location takes also $O(\log m)$ steps in the obvious algorithm (gather to pairs, quadruples, etc.). Therefore, stage 5 takes $O(\log\log n)$ steps. Stage 4 takes time T, as we assumed. We sum up that PJ_k can be computed in time $T + O(\log\log n)$.

By theorem 1 PJ_k cannot be computed by a EREW algorithm in less than $k-1$ steps. Therefore, $T + O(\log\log n) \geq k-1 \geq \Omega(\sqrt{\log n})$. Hence, $T = \Omega(\sqrt{\log n}) = \Omega(\sqrt{\log N})$. •

References

[CDR86] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–98, 1986.

[FW90] F. Fich and A. Wigderson. Toward understanding exclusive read. *SIAM J. Comput.*, 19:718–727, 1990.

[GNR89] E. Gafni, J. Naor, and P. Ragde. "On separating the EREW and CROW models". *Theoret. Comput. Sci.*, 68:343–346, 1989.

[KR88] R. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines, 1988. Handbook of Theoret. Comput. Sci., Volume A.

[Sim83] H. U. Simon. A tight $\Omega(\log\log n)$ bound on the time for parallel RAM's to compute nondegenerate boolean functions. FCT'83, Lecture notes in Comp. Sci. 158, 1983.

[Sni85] M. Snir. On parallel searching. *SIAM J. Comput.*, 14:688–708, 1985.