The Red-Blue Pebble Game on Trees and DAGs with Large Input

Niels Gleinig
ETH Zurich
niels.gleinig@inf.ethz.ch

Torsten Hoefler
ETH Zurich
torsten.hoefler@inf.ethz.ch

Abstract

Data movements between different levels of a memory hierarchy (I/Os) are a principal performance bottleneck. This is particularly noticeable in computations that have low complexity but large amounts of input data, often occurring in "big data". Using the red-blue pebble game, we investigate the I/O-complexity of directed acyclic graphs (DAGs) with a large proportion of input vertices. For trees, we show that the number of leaves is a 2-approximation for the optimal number of I/Os. Similar techniques as we use in the proof of the results for trees allow us to find lower and upper bounds of the optimal number of I/Os for general DAGs. The larger the proportion of input vertices, the stronger those bounds become. For families of DAGs with bounded degree and a large proportion of input vertices (meaning that there exists some constant $c > 0$ such that for every DAG G of this family, the proportion p of input vertices satisfies $p > c$) our bounds give constant factor approximations, improving the previous logarithmic approximation factors. For those DAGs, by avoiding certain I/O-inefficiencies, which we will define precisely, a pebbling strategy is guaranteed to satisfy those bounds and asymptotics. We extend the I/O-bounds for trees to a multiprocessor setting with fast individual memories and a slow shared memory.

1 Introduction

Data movement between slow and fast memories (called I/O-transitions, or simply **I/Os**) are widely considered a principal performance bottleneck in computing [25]. Due to the ever increasing gap between the speed at which data can be processed and the speed at which it can be communicated, this phenomenon is particularly noticeable in computations that perform simple operations on large amounts of input data. This computation profile of **low arithmetic intensity + large amounts of input data** is very common and often referred to as *computations with memory bound performance*. For example, when performing inference on trained neural networks of fully connected layers (or other machine learning models), we need to read millions of values (mostly connection weights), but will use most of these values only once. Other problems that belong to this category can be found in Linear Algebra (addition of tensors or matrix-vector multiplication, either sparse or dense), Graph Theory (BFS, DFS, topological sorting), Statistics (computing moving averages, quantiles, covariances, linear regression, ANOVA, ANCOVA, etc.), and Image Processing (applying filters or affine transformations). Also, in general, most "big data" computations tend to belong to this category, because given the large amount of input data, it would not feasible to perform computational work of high complexity. Further, parallel summary statistics are a popular technique in the big data paradigm, naturally leading (on some level of granularity) to data dependencies that form trees.

In this paper, we will show how to perform I/Os efficiently in these computations with large input. As our main contribution, we present a set of rules on how to perform I/Os, together with several theorems that establish I/O-bounds and show: following these rules we are guaranteed to not use more than a constant number of times more I/Os than optimal. Interestingly, these rules are "local": they only require information about the neighbors. Hence they evade expensive precomputing of an I/O-optimal schedule and can be implemented at runtime.

1.1 Related work

1.1.1 Communication-efficient algorithms

Communication-efficient algorithms have been developed for many concrete computational problems and models, including for example matrix multiplication [17, 18], FFT [17, 1], sorting [1], directed shortest paths [23], topological sorting [2], matrix transposition [1], the N-Body problem [14], QR- and LU-factorization [13], prime tables [5], and Cholesky decomposition [4]. In the blocked-I/O-model [1], using *time-forward processing* one can compute functions that have a given computation DAG $G = (V, E)$ with $O(\text{Sort}(|E|))$ I/Os [10]. In the particular case that the DAG is a tree, one can compute an Euler-tour of that tree with $O(\text{Sort}(|E|))$ I/Os and once the tree is laid out according to that Euler-tour, the function can be computed with $O(\text{Scan}(|E|))$ I/Os [21]. There are methods for I/O-efficient scheduling of tasks with tree-dependencies [22]. I/O-efficient algorithms are often closely tailored to the given problem. The red-blue pebble game allows to analyze and optimize the I/Os of *general* computations. For example, it has been used to optimize the I/Os of classical matrix multiplication [17, 18], which can be considered very opposite to the computations of this paper as it allows extensive data reuse.

1.1.2 Pebble games

There is a natural way to associate DAGs (directed acyclic graphs) to computations: vertices represent data and the direct predecessors of a vertex v are the data that are needed to compute v. Vertices with no incoming edges represent the input-data of the computation and those with no outgoing edges represent the output-data. Such a DAG is called a CDAG (computation DAG).

Pebble games are a family of combinatorial games that are played on DAGs [8][17]. By letting the DAG be the CDAG of a given computation, one can use them to analyze resource requirements of that computation: the black pebble game is for example used to model space complexity of general computations, whereas the reversible pebble game is used to model space complexity of reversible computations and the red-blue pebble game is used to model the requirement of I/Os.

Most pebble games are hard to solve (finding a pebbling strategy that is optimal with respect to some metric) or even approximate on general DAGs. It has been shown [16] that even the black pebble game, arguably the most simple pebble game, is PSPACE-complete to solve. Further, it is known that for many pebble games it is even PSPACE-hard to find approximate solutions [9][11][8][3]. It has been shown that the red-blue pebble game is PSPACE-complete and the red-blue pebble game without deletion is NP-complete [20].

There exists a polynomial time algorithm that approximates the solution of the red-blue pebble game (finding the minimal number of I/Os) by a multiplicative factor of $\log^{3/2}(n)$, using a number of red pebbles which is increased by a multiplicative factor of $\log^{3/2}(n)$ [7]. Yet, there is no known algorithm that approximates the solution of the red-blue pebble game by a constant multiplicative factor in polynomial time.

These hardness results suggest to search algorithms for pebble games on restricted classes of DAGs. This is furthermore motivated by the fact that the CDAGs of actual computations are a very restricted subset of all DAGs. They usually have a special structure (for example layered or planar), symmetries (large automorphism groups, i.e., they have many vertices that "look equal"), and other properties (for example regular or bounded in-degree) that can make it easier to find solutions. Hence, [12] raised the question whether for the red-blue pebble game there exist FPT algorithms for restricted classes of DAGs (such as bounded width graphs).

The main contribution of this paper are various theorems that give answers (some approximate, some exact) to the 3 problems that we present in the following section. We present lower and upper bounds that differ from each other by multiplicative constants that depend only on the degrees of a DAG and the proportion of input vertices, but not on the size of the DAG. In

particular, for families of DAGs with bounded degrees and a large proportion of input vertices (meaning that there exists some constant $c > 0$ such that for every DAG G of this family, the proportion p of input vertices satisfies $p > c$), these bounds provide constant approximations.

1.2 The red-blue pebble game

The red-blue pebble game was introduced by Hong and Kung [17] to model the I/Os of a computation that has a given computation DAG $G = (V, E)$ and a fast memory of limited size $S \in \mathbb{N}^+$. Blue pebbles represent data that is stored in slow memory and red pebbles represent data that is stored in fast memory. There are no restrictions on the number of blue pebbles that can reside on G at any given time, but we can never have more than S red pebbles on G, where S represents the size of the fast memory. In the beginning, there is a blue pebble on each of the input vertices $V_{in} \subseteq V$ (those are the vertices with no incoming edges) and the game is completed when we have a blue pebble on each of the output vertices $V_{out} \subseteq V$ (those are the vertices with no outgoing edges). In order to obtain that goal, we are allowed to apply a sequence of the following actions

- R1 (Input): Replace a blue pebble by a red pebble.
- R2 (Output): Replace a red pebble by a blue pebble.
- R3 (Compute): Place a red pebble on a vertex for which all parents are carrying a red pebble (when the vertex has already a blue pebble, we replace it).
- R4 (Delete): Delete a red pebble.

We can think of action R3 as saying "we can compute a certain value, when all values that we need for that computation are in fast memory". Since the size of our fast memory is limited we will have to apply actions R2 and R4 sometimes.

Figure 1 shows a complete red-blue pebble game on a simple DAG. Keeping in mind that the I/Os in this model represent data movements that consume much time and energy, it becomes clear that we want to minimize their number.

Definition 1.1. A **pebbling strategy** (or **computation strategy**) is a sequence of the actions R1, R2, R3, and R4 on the vertices of a DAG, which completes the red-blue pebble game (like for example $[\mathrm{R1}(v_1), \mathrm{R1}(v_2), \mathrm{R3}(v_3), \mathrm{R4}(v_1) \ldots, \mathrm{R2}(v_n)]$). Each application of R1 or R2 counts as one **I/O** and we let $rb(G, S)$ denote the minimal number of I/Os of any pebbling strategy of the DAG G with S red pebbles, setting $rb(G, S) = \infty$ if there is no pebbling strategy of G with S red pebbles. A pebbling strategy which uses this minimal number $rb(G, S)$ of I/Os is called **I/O-optimal**. We say that it is possible to **pebble (or compute) G with k I/Os** if $rb(G, S) \leq k$.

Problem 1 (Red-blue pebble game): given a DAG G and $S \in \mathbb{N}^+$, find the number $rb(G, S)$.

Solving problem 1 for general DAGs is PSPACE-complete [20], but this number can be bounded from above and below using different forms of optimal partitions and coverings of the DAG [17, 15, 6]. However, to the best of our knowledge, there is no known way to find those optimal partitions for general DAGs efficiently.

Problem 2: given a sequence of DAGs G_n, how fast does $rb(G_n, S)$ grow as $n \to \infty$?

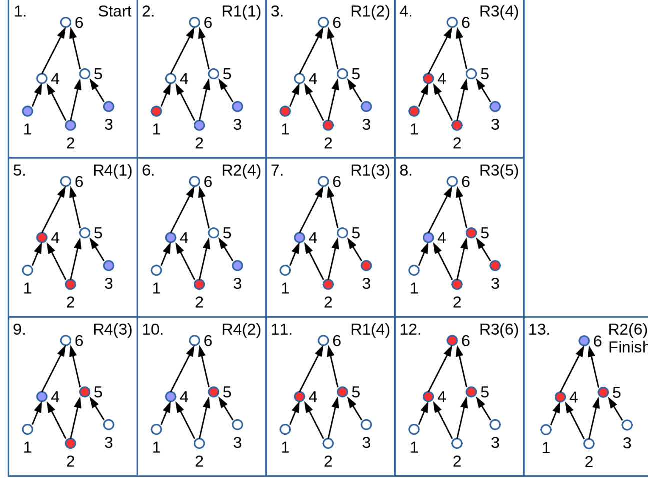


Figure 1: Red-blue pebble game with $S = 3$ red pebbles. $Ri(j)$ denotes an application of action i on vertex j. This pebbling strategy is I/O-optimal and has a total of $rb(G, 3) = 6$ I/Os.

If these DAGs G_n in problem 2 are the CDAGs of some algorithm for input size n, this asymptotic growth is generally known as the **I/O-complexity of the algorithm**. The minimum of the I/O-complexities of all algorithms that solve a given problem is known as the **I/O-complexity of the problem**. Here we are typically satisfied with answers that disregard constant multiplicative factors, i.e., pebbling strategies are called optimal if they use $\Theta(rb(G_n, S))$ I/Os. I/O-complexities have been studied for many algorithms and problems of practical importance [23, 1, 2, 4, 5, 24, 13, 17], often in a more general model that considers additional parameters such as memory block size.

Now if the vertices of the CDAG of a computation represent values produced by constant-time operations and the motivation for studying problems 1 or 2 (or analogous problems for other pebble games) is to improve the performance of that computation, it is important to note the following: the sequential runtime is given, up to constants, by the number of vertices $|V|$ and therefore any algorithm that solves problems 1 or 2 in time above $\Omega(|V|)$ is of little practical value (unless one runs the same CDAG many times, letting the performance gains add up and justify a more expensive algorithm to solve problems 1 or 2). So for example algorithms that are based on spectral methods are likely to be inappropriate, since the time for computing eigenvalues or eigenvectors is at least of the order of $|V|^2$. Likewise, for the practical value of such an algorithm, it is also important that it does not use too many other resources. This imposes great restrictions on us and motivates us to ask if it is possible to know how to run an algorithm in a close to I/O-optimal way without having to do any pre-computations at all on the CDAG $G = (V, E)$. Could it be possible to find a set of simple rules, such that any computation strategy that follows these rules is close to I/O-optimal? More precisely, we are interested in the following question:

Problem 3: given a DAG G, $S \in \mathbb{N}^+$ and a constant $1 \leq \lambda \in \mathbb{R}$, is it possible to find a set of simple rules, such that any computation strategy for G which follows these rules does not use more than $\lambda \cdot rb(G, S)$ I/Os?

Note: our rules do not allow a vertex to have a blue and a red pebble at the same time. We chose this version because it makes the proofs more elegant. However, all of our results (both the bounds and the algorithms) still hold if we allow a vertex to hold both a blue and a red pebble (by using the word "place" instead of "replace" in rules R1 and R2 and removing

the comment in the parentheses in R3). The upper bounds remain true, because by having this additional degree of freedom the pebble strategies can only improve. The lower bounds remain true, because even if we are allowed to have two pebbles on one vertex, we will still have to spend one I/O on each of the input vertices. Trees can be pebbled optimally without ever having to put two pebbles on one vertex.

The paper is organized as follows. In Section 2 we present and prove several lower and upper bounds for the number of I/Os for trees. In Section 3 we extend the bounds to more general classes of DAGs. In Section 4 we extend our bounds to a parallel setting. In section 5 we present our experimental results.

1.3 Notations and definitions

Let $G = (V, E)$ be a DAG. Throughout this paper we say that a vertex w is a child of the vertex v if there is an edge from v to w. A vertex v is a parent of w if and only if w is a child of v. We let $d_{out}(v)$ denote the number of children of v (we only consider simple DAGs), $d_{in}(v)$ the number of parents of v and $d(v) = d_{in}(v) + d_{out}(v)$ the degree of v. We let

$$\begin{aligned} \hat{d}_{out} &= \max_{v \in V} d_{out}(v), \\ \hat{d}_{in} &= \max_{v \in V} d_{in}(v) \\ \hat{d} &= \max_{v \in V} d(v) \end{aligned} \tag{1}$$

denote the maximal out-degree, maximal in-degree, and maximal degree respectively.

An in-tree is a directed tree in which all edges are directed in such a way that there is a unique vertex v (called the root or output vertex) which can be reached from any other vertex by a directed path (like in Figure 2). Throughout this paper, all trees are in-trees. A vertex that has no in-coming edges is called a leaf and the set of all leaves is denoted $L(G)$.

2 Bounds for trees

Notice that the condition of $S > \hat{d}_{in}$ which we require in our results is equivalent to letting S be large enough such that G can be computed.

Theorem 2.1. Let $G = (V, E)$ be a tree with more than one vertex and $S > \hat{d}_{in}$. Then, the set of leaves $L(G)$ satisfies

$$|L(G)| < rb(G, S) \leq 2|L(G)|. \tag{2}$$

Furthermore, when $|L(G)| > 1$ the second inequality is also strict.

Proof of Theorem 2.1. Since the computation depends on all of the leaves, we will have to put at least once a red pebble on each of the leaves. We furthermore will have to spend one I/O to store the output. This gives us the first inequality.

The second inequality we will prove by induction on the number of leaves $|L(G)|$. One can easily check that the inequalities are true for $|L(G)| = 1$ and $|L(G)| = 2$ (with strictness for $|L(G)| = 2$). Now let us assume that it also holds for trees with $n > 2$ leaves and let G be a tree with $n + 1$ leaves.

Without loss of generality the tree has no vertices of in-degree 1, because on trees such vertices can be removed (merging the incoming edge with the outgoing edge) without changing the number of I/Os. We can also assume that the root v has degree at least 2, because otherwise we could remove the root and the incoming edge without changing the number of I/Os. So let w be a parent of v and G^w be the sub-DAG below w (that is, all ancestors of w and w itself) and $\overline{G^w}$ be the DAG that we obtain by deleting from G all vertices and edges that are in G^w

and the edge that goes from w to v (see Figure 2). Then $|L(G^w)| \leq n$ and $|L(\overline{G^w})| \leq n$. So by induction hypothesis these trees can be pebbled with at most $2 \cdot |L(G^w)| - 1$ and $2 \cdot |L(\overline{G^w})| - 1$ respectively. Now consider the following pebbling strategy:

1. Pebble G^w I/O-optimally (getting a blue pebble on w)
2. Pebble $\overline{G^w}$ I/O-optimally and when you have red pebbles on all of the parents of v (and are about to put a red pebble on v), put a red pebble on w first and then put a red pebble on v and finish the game.

This is clearly a complete pebbling strategy and since $|L(G^w)| + |L(\overline{G^w})| = |L(G)|$ the number of I/Os is at most

$$(2 \cdot |L(G^w)| - 1) + \big(2 \cdot |L(\overline{G^w})| - 1\big) + 1 = 2 \cdot |L(G)| - 1. \tag{3}$$

□

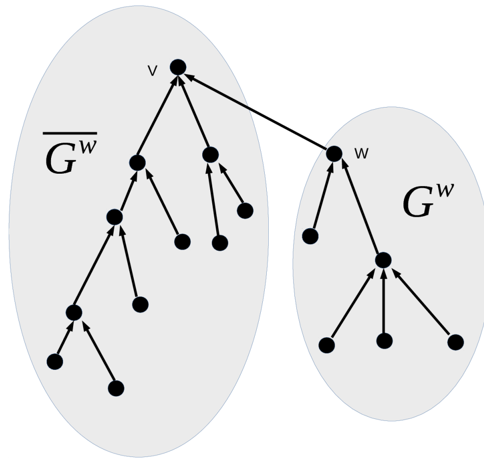


Figure 2: This shows the subtrees G^w and $\overline{G^w}$ from the proof of theorem 2.1.

Note: the multiplicative constants 1 and 2 from the inequalities in the previous theorem are tight:

- To see that the constant 1 is tight, notice that whenever $S > |L(G)|$, we have $rb(G, S) = |L(G)| + 1$. Also, for trees that consist of a line of depth n, with each of the first $n - 1$ vertices on that line having one additional predecessor that is a leaf (like in Figure 3), we have $rb(G, S) = |L(G)| + 1$ independently of S.
- If T_k^2, denotes a full binary tree of k levels, then

$$\lim_{k \to \infty} \frac{rb(T_k^2, 3)}{\mid L(T_k^2) \mid} = 2. \tag{4}$$

To present our next result we need the following definition.

Definition 2.1. We say that a computation has **empty I/Os** if it does at least one of the following things:

1. Put a red pebble on a vertex v which has a blue pebble on it and eventually remove the red pebble (either by deleting it or replacing it with a blue one) or finish the game, without having computed a child of v which had not yet been computed before.

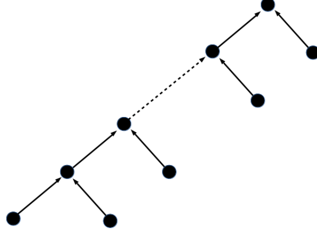


Figure 3: Unbalanced trees require few I/Os. “Ladder trees” like this one, minimize the number of I/Os among all trees with a fixed number of leaves.

2. Put a blue pebble on a non-output vertex v, when v will not be needed again.

Even though intuitively it may seem like avoiding empty I/Os always reduces the total number of I/Os, there are in fact DAGs for which every I/O-optimal pebbling strategy has empty I/Os. The reason for this is that one can sometimes save I/Os by recomputing vertices (and thereby possibly producing empty I/Os) rather than loading them. Figure 4 shows such a DAG.

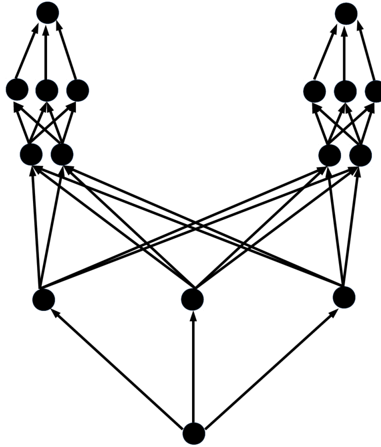


Figure 4: When $S = 5$ this DAG can be pebbled with $rb(G, 5) = 5$ I/Os, but any pebbling strategy without empty I/Os has at least 7 I/Os (and when we allow a vertex to have two pebbles at the same time, the numbers become 4 and 6). The reason is that any pebbling strategy will need to have red pebbles on the three vertices of the second layer at two different moments. For that, the optimal strategy would reload the input and then recompute those three vertices, whereas a strategy without empty I/Os would have to reload the three vertices in order to avoid empty I/Os.

Nevertheless, as the next two theorems show, avoiding empty I/Os does guarantee provably good performance. So the simple rule of avoiding empty I/Os is an answer to problem 3.

Theorem 2.2. Let $G = (V, E)$ be a tree with more than one vertex and without vertices of in-degree 1 and $S > \hat{d}_{in}$. Then, any computation strategy of G which has no empty I/Os has a number of I/Os that is between $|L(G)|$ and $3 \cdot |L(G)|$ and hence optimal up to a multiplicative factor of 3.

Proof of Theorem 2.2. That the number of I/Os is larger than $|L(G)|$ follows again from the fact that we will have to put at least once a red pebble on each of the leaves.

For the other inequality we recall that any tree which has no vertices of in-degree 1 satisfies

$$| I(G) | \leq | L(G) |, \tag{5}$$

where $I(G) = V \setminus L(G)$ denotes the set of inner vertices.

When we put a blue pebble on a vertex which has a red pebble on it, we will eventually put again a red pebble on it, because otherwise we would have a empty I/O of the second type. When we put a red pebble on a vertex v which has a blue pebble on it, we will compute $child(v)$ before doing any other I/Os on v, because otherwise we would have a empty I/O of the first type. Once $child(v)$ has been computed, we will not do any other I/Os on v, because v is no longer needed. It follows that we spend at most one I/O on any leaf and at most two I/Os on any inner vertex. Combining this with Equation (5) we conclude that the total number of I/Os is less than

$$| L(G) | + 2 | I(G) | \leq 3 | L(G) | . \tag{6}$$

□

Note: when a pebbling strategy for trees without vertices of in-degree 1, besides from having no empty I/Os, also

1. "computes subtrees strictly one after another" (that is, whenever it started to perform computations on some subtree t of T it does not do any computations that are not needed for t, until it has finished with the computation of t),
2. whenever a vertex is computed, the pebbles from the parents are immediately deleted,
3. does not perform store operations unless there are S red pebbles on the DAG,

then it is a pebbling strategy like the one from the proof of Theorem 2.1 and hence it is optimal up to a multiplicative factor of 2.

This order ("computing the subtrees strictly one after another") is also known as *postorder*, and related to previous work [19][22]. Using postorders, one can also find exact solutions to the red-blue pebble game on trees. To do this, one needs to determine in which order the subtrees can be traversed most efficiently: depending on the order in which we traverse subtrees, it can be possible to hold the output of one subtree in fast memory while computing the next subtree, sparing 2 I/Os. Finding such an optimal order of the subtrees can be done with dynamic programming approaches. However, we will not discuss the details of that in this paper since our goal is to find general bounds for general DAGs.

3 Bounds for general DAGs

The key property of trees that we made use of in the proof of the previous results is that the set of input vertices makes up a large proportion of the set of all vertices (or at least it can be assumed that it is a tree with that property) and the out-degree is small (bounded by 1). The following result gives good estimates of $rb(G, S)$ for more general DAGs G with those properties.

Theorem 3.1. Let $G = (V, E)$ be a DAG without isolated vertices and $S > \hat{d}_{in}$. Let $p = \frac{|Input(G)|}{|V|}$ be the proportion of input vertices of G. Then,

$$|Input(G)| \leq rb(G, S) \leq |Input(G)| \left(\frac{2\hat{d}_{out}}{p} - 1 \right) \tag{7}$$

and the number of I/Os of any pebbling strategy without empty I/Os is between these bounds.

Proof of Theorem 3.1. Since any input vertex has to be loaded at least once we get $|Input(G)| \leq rb(G,S)$. For the other inequality notice that in any pebbling strategy without empty I/Os, with every [store, load] pair of I/Os on one vertex we compute at least one new child, because otherwise we would get a empty I/O. So we spend at most $2 \cdot \hat{d}_{out}$ I/Os on any non-input vertex. Likewise, one can see that we spend at most $2 \cdot \hat{d}_{out} - 1$ I/Os on any input vertex. Therefore the total number of I/Os is at most

$$\begin{aligned} &|Input(G)| \left(2\hat{d}_{out} - 1\right) + |V \backslash Input(G)| 2\hat{d}_{out} \\ &= 2\hat{d}_{out}|V| - |Input(G)| \\ &= |Input(G)| \left(\frac{2\hat{d}_{out}}{p} - 1\right). \end{aligned} \tag{8}$$

□

Theorem 3.2. Let $G_n = (V_n, E_n)$ be a sequence of regular DAGs without isolated vertices and with constant in-degree $d_{in} < S$ (that is, any vertex of G_n which is not an input vertex has in-degree d_{in}) and constant out-degree d_{out} (that is, any vertex of G_n that is not an output vertex has out-degree d_{out}). If $d_{out} < d_{in}$ then $rb(G_n, S)$ grows asymptotically like $|V_n|$. More precisely,

$$|V_n| \left(1 - \frac{d_{out}}{d_{in}}\right) \leq rb(G_n, S) \leq |V_n| \left(\frac{2 \cdot d_{out}}{1 - \frac{d_{out}}{d_{in}}} - 1\right) \tag{9}$$

and the number of I/Os of any pebbling strategy without empty I/Os is also between these bounds.

Proof of Theorem 3.2. Since the sum of the in-degrees over all vertices equals the sum of the out-degrees, we have

$$(|V_n| - |Input(G_n)|) \cdot d_{in} = (|V_n| - |Output(G_n)|) \cdot d_{out}. \tag{10}$$

From this we obtain

$$\begin{aligned} |Input(G_n)| &= |V_n| - (|V_n| - |Output(G_n)|) \cdot \frac{d_{out}}{d_{in}} \\ &\geq |V_n| \cdot \left(1 - \frac{d_{out}}{d_{in}}\right), \end{aligned} \tag{11}$$

which means that the proportion of input vertices is $p \geq \left(1 - \frac{d_{out}}{d_{in}}\right)$. The result follows now by substituting p by $\left(1 - \frac{d_{out}}{d_{in}}\right)$ in the RHS of the inequalities of Theorem (3.1) and replacing the LHS of those inequalities by the RHS of (11). □

A remarkable aspect of the previous results is that the bounds only require $S > \hat{d}_{in}$, but besides that they do not depend on S. So they identify families of DAGs for which the I/O-complexity does not depend on the size of the fast memory.

Theorem 3.3. Let $G_n = (V_n, E_n)$ be a sequence of DAGs without isolated vertices and $S > \hat{d}^n_{in}$. Let G_n have average in-degree $\bar{d}^n_{in}$ (where the average is taken over all vertices that are not input vertices), average out-degree $\bar{d}^n_{out}$ (where the average is taken over all vertices that are not output vertices) and maximal out-degree $\hat{d}^n_{out}$. Then,

$$|V_n| \left(1 - \frac{\bar{d}^n_{out}}{\bar{d}^n_{in}}\right) \leq rb(G_n, S) \leq |V_n| \left(\frac{2 \cdot \hat{d}^n_{out}}{1 - \frac{\bar{d}^n_{out}}{\bar{d}^n_{in}}} - 1\right) \tag{12}$$

and hence, if

$$\limsup_{n\to\infty} \bar{d}^n_{out}/\bar{d}^n_{in} < 1 \tag{13}$$

and $\hat{d}^n_{out} \leq c, \forall n$ for some constant $c \in \mathbb{N}$, then $rb(G_n, S)$ grows asymptotically like $|V_n|$. Furthermore any pebbling strategy without empty I/Os satisfies these bounds.

Proof of Theorem 3.3. Like in the previous proof the Equation (10) still holds if we replace d_{in} by $\bar{d}^n_{in}$ and d_{out} by $\bar{d}^n_{out}$. The rest of the proof is analogous. □

For algorithms whose CDAGs have vertices that represent values produced by constant-time operations, this previous theorem can be interpreted as saying: "if the out-degrees are smaller than the in-degrees (differing on average by a multiplicative constant which is bounded away from 1) and the maximal out-degree is uniformly bounded, then the I/O-complexity equals the time complexity (which is linear in the input size)."

4 I/O-bounds in a parallel setting

In order to establish I/O-bounds in multiprocessor settings, we introduce a variation of the red-blue pebble game. We model a setting with P processors, each one having a fast memory of size S/P and access to a shared memory of infinite size. Throughout this section we assume S/P is an integer. There are P different types of red pebbles, corresponding to the fast memories of the P processors. We denote the colors of these pebbles $r_1, r_2, \ldots, r_P$. At any time step and for any processor $p \in \{1, 2, \ldots, P\}$ there cannot be more than S/P pebbles of color r_p on the DAG, and we can perform one of the following actions

- R1 (Input): replace a blue pebble by a pebble of color r_p.
- R2 (Output): replace a pebble of color r_p by a blue pebble.
- R3 (Compute): place a pebble of color r_p on a vertex for which all parents are carrying a pebble of color r_p (when the vertex has already a blue pebble, we replace it).
- R4 (Delete): delete a pebble of color r_p.

In this parallel setting we say that a pebbling strategy has empty I/Os if it does at least one the following:

1. Put a pebble of some color $r_p \in \{r_1, r_2, \ldots, r_P\}$ on a vertex v which has a blue pebble on it and eventually remove the red pebble (either by deleting it or replacing it with a blue one) or finish the game, without having computed a child of v which had not yet been computed before by any of the processors.
2. Put a blue pebble on a non-output vertex v, when v will not be needed again.

As in the sequential case, initially, there is a blue pebble on each of the input vertices, and the goal is to get blue pebbles on all output vertices. In contrast to the standard red-blue pebble game, we are now able to perform up to P actions at each time step, one for each processor. Again, we want to minimize the total number of applications of R1 and R2, but in this setting we also want to minimize another quantity: the total amount of time steps. Since it is possible to save I/Os by taking more time steps (for example, by doing all the work with one single of the P processors), we will enforce the pebbling strategy to use a minimal number of time steps. That is, we want to know: what is the minimal number of I/Os used by pebbling strategies that pebble G in a minimal number of time steps.

For a given DAG G, we now let $T_P(G)$ denote the minimal number of time steps used by any strategy that pebbles G with P processors. We define $rb_P(G, S)$ as the minimal number of I/Os used by any pebbling strategy that pebbles G in $T_P(G)$ time steps. The next theorem shows that enforcing a parallel schedule that uses a minimal number $T_P(G)$ of time steps (which is a restriction on the orders in which we can pebble the vertices), does not affect the validity of the I/O-bounds established by Theorem 2.2, and hence, does not increase the number of I/Os by more than a factor 3.

Theorem 4.1. Let G be a tree with $\hat{d}_{in} < S/P$. Then,

$$|L(G)| \leq rb_P(G, S) \leq 3 \cdot |L(G)|, \tag{14}$$

and the number of I/Os of any pebbling strategy without empty I/Os is between these bounds.

To prove this Theorem we need a Lemma that shows how we can remove empty I/Os from time optimal pebbling strategies without losing time-optimality.

Lemma 4.2. Let G be a tree with $\hat{d}_{in} < S/P$ and consider a pebbling strategy that pebbles G in optimal time $T_P(G)$ (possibly having empty I/Os). Then, we can transform this pebbling strategy into another pebbling strategy that uses the same number of time steps (and is hence also time optimal), but does not have any empty I/Os.

Proof. Given a pebbling strategy that pebbles G in $T_P(G)$ steps we can remove all empty I/Os of the second type and obtain another valid strategy that pebbles G in at most the same number of steps (and hence is still time optimal).

It remains to show that we can also remove empty I/Os of the first type. Notice that empty I/Os of the first type occur in two cases: 1) We load a value, but we do not compute the child afterwards, 2) We load a value and we *do* compute the child, but the child has already been computed earlier.

In the first case, we can again remove the I/O without losing the completeness or time-optimality of the pebbling strategy.

Now consider a situation of the second case, where we place a red pebble on some vertex v and we *do* compute its child $child(v)$, but $child(v)$ has already been computed earlier. In this case the steps of loading v and recomputing $child(v)$ (which together take 2 time steps), could be replaced by two other steps: storing the value of $child(v)$ and eventually reading $child(v)$ again (this replacement increases the number of I/Os by one, but this is irrelevant to this proof as it does not increase the number of time steps).

With these changes we can transform any time optimal pebbling strategy *with* empty I/Os into a pebbling strategy *without* empty I/Os. □

Proof of Theorem 4.1. The lower bound is established with the same reasoning as in the sequential model: all leaves have to be read at least once.

Now consider any P processor pebbling strategy that pebbles G in $T_P(G)$ time steps and use Lemma 4.2 to transform it into another pebbling strategy that runs in $T_P(G)$ time steps and does not have empty I/Os. We can associate to this P processor pebbling strategy a sequential strategy without empty I/Os (which we will call the "sequentialized" pebbling strategy):

- In the first time step, perform the action that processor 1 performed in the first time step,
- In the second time step perform the action that processor 2 performed in the first time step,

...

- In the P-th time step, perform the action that processor P performed in the first time step,

- In the $(P+1)$-th time step, perform the action that processor 1 performed in the second time step,

...

Since for any $p \in \{1, 2, \ldots, P\}$ there are not more than S/P pebbles of color r_p on the tree at any moment, the sequentialized pebbling strategy uses not more than $P \cdot (S/P) = S$ red pebbles. Since it also does not perform empty I/Os (now we mean "empty I/Os" in the sense of the sequential model), it follows from Theorem 2.2 that it uses at most $3 \cdot |L(G)|$ I/Os. Since the parallel pebbling strategy uses the same number of I/Os as the sequentialized strategy, we obtain $rb_P(G, S) \leq 3 \cdot |L(G)|$. □

5 Experiments

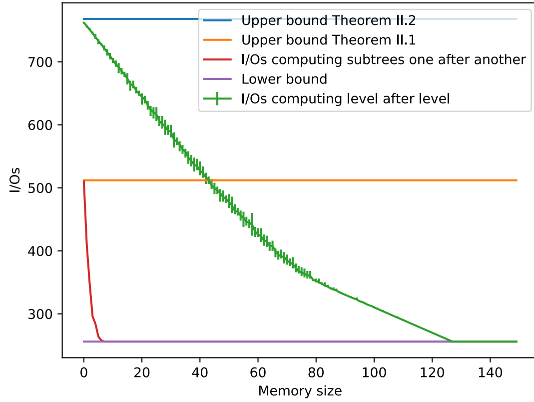
To illustrate the theoretical contributions of our paper, we ran several experiments. The purpose of the experiments is to show the effect that the pebbling techniques used in this paper (avoiding empty I/Os, computing subtrees one after another) have on concrete I/O-counts and to see how close to the bounds we get on different trees and random DAGs.

5.1 Regular tree, varying memory size

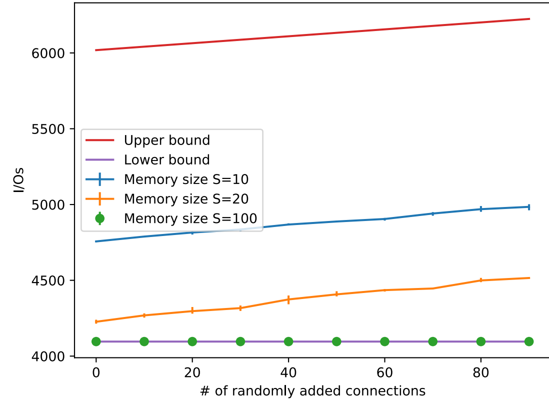
We considered a binary tree of depth 8 and pebbled it in two different orders of the vertices: **A**) Level after level, and **B**) subtrees one after another. In both cases we performed I/Os according to the following rules: 1.) We only put a red pebble on a vertex when we need it for the computation of the next vertex (in the given order), 2.) we only remove a red pebble when there is no more space but we need to place another red pebble for the next computation step, 3.) when we need to remove a red pebble we choose it uniformly at random among the red pebbles that we will not need in the next computation step, and 4.) after each computation step we delete all red pebbles that will not be needed again in any future computations. By following these rules we are guaranteed to not perform empty I/Os (rules 1, 2, and 3 imply that we do not perform an empty I/O of the first kind; rule 4 implies that we do not perform an empty I/O of the second type). Hence, according to Theorem 2.1, using order **A** we should have between $|L(G)| = 256$ and $2 \cdot |L(G)| = 512$ I/Os, and according to Theorem 3.1, with order **B** we should have between $|L(G)| = 256$ and $3 \cdot |L(G)| = 768$ I/Os. In Figure 5a we show how the number of I/Os decreases as we increase the memory size S. For each memory size we pebbled the tree 10 times and show bars that represent the smallest and largest observed number of I/Os (the variation stemming from the random choice when removing red pebbles). We can see that for both computation orders, the respective bounds are satisfied for all memory sizes (as predicted by our theorems). Furthermore, when the memory size is close to zero, the number of I/Os is close to the respective upper bound. When we increase the memory size, the number of I/Os decreases and eventually meets the lower bound. With computation order **A** we already meet the lower bound with memory size $S = 9$, whereas with order **B** this minimum is only attained when $S = 129$.

5.2 Random DAGs with large inputs

To generate more general DAGs with large input, we used the random sampling method described in Algorithm 1 (the constants 4 and 8 in this sampling method are arbitrary constants; we just had to fix free parameters). Notice that for $l = 0$, Algorithm 1 generates a tree, but as we increase l, we obtain DAGs that differ increasingly from a tree. However, the total number of vertices and input vertices remains equal (because we only add edges (i, j) for which j is not an input). Figure 5b shows how the number of I/Os increases as we add more edges to the DAG. The shown numbers are the average over 10 sampled DAGs and the bars correspond to



(a) Number of I/Os for varying memory sizes when pebbling a binary tree of depth 9.



(b) Starting with a regular tree of depth 4 and degree 8, we randomly add directed edges to it.

Algorithm 1 Input: a number $l \in \mathbb{N}$ /Output: a DAG with $8^5 - 1$ vertices and $8^5 - 2 + l$ edges.

Let G be a full 8-ary tree of depth 4 and identify V with the numbers $\{1, 2, \ldots 8^5 - 1\}$, 1 being the root, $2, \ldots, 9$ being the vertices of the first level, and so on;
for i from 1 to l **do**
  Choose uniformly at random vertices $i, j \in G$, such that j is not an input, $(i, j) \notin G$, and $j < i$;
  Add (i, j) to G;
end for

the smallest and largest observed values. The shown bounds are obtained from Theorem 3.1. We can see that the lower bound remains constant (which follows from the fact that the input size of these DAGs is constant), while the upper bound increases linearly with the number of edges that we add (as this impacts the degrees of the vertices). Using a memory size $S = 100$ in the shown range, we use a number of I/Os that matches the lower bound exactly. In the whole range, the lower and upper bound differ by a factor of less than 2.

5.3 General random DAGs with varying numbers of vertices

Finally, we considered a second method for sampling random DAGs, defined in Algorithm 2 (also here the constants are arbitrary). We used this method with different values of l, to generate DAGs with varying numbers of vertices. In Figure 6 we show how the number of I/Os grows as the number of vertices increases. We can clearly see that the I/O-complexity grows linearly with the number of vertices, as predicted by Theorem 3.3.

Algorithm 2 Input: a number $l \in \mathbb{N}$ /Output: a DAG with $3 \cdot l + 1$ vertices.

Let G be a graph with a single vertex
for i from 1 to l **do**
  Choose uniformly at random a vertex $v \in G$ with $d(v) < 9$;
  Add 3 new vertices to G;
  Add 3 edges, pointing from the 3 new vertices to v;
end for
Add 30 additional random edges (as in the previous subsection);

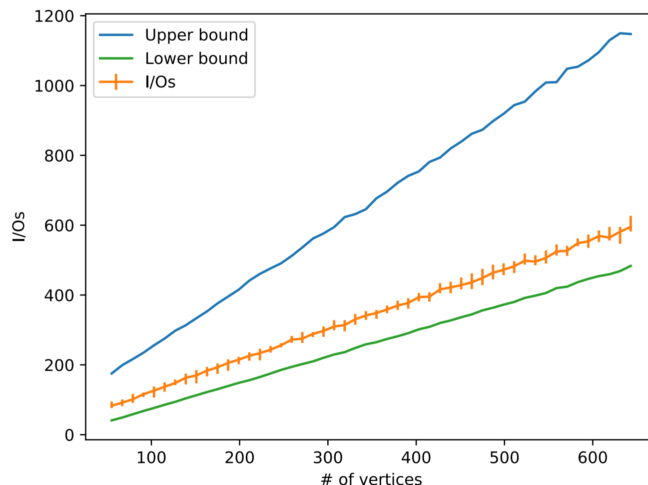


Figure 6: Number of I/Os for varying sizes of randomly generated DAGs using memory size 32.

6 Conclusions

We introduce a set of I/O-rules that can be used as a fundamental tool in the design of I/O-efficient big data computations, both in a sequential and parallel setting. We prove several bounds for the optimal number of I/Os. These bounds are particularly strong when the DAGs have a large proportion of input vertices (meaning that there exists some constant $c > 0$ such that for every DAG G of this family, the proportion p of input vertices satisfies $p > c$). For these DAGs our bounds provide constant factor approximations, which improves the previous logarithmic approximation factors. Any pebbling strategies with exclusively useful (non-empty) I/Os have a number of I/Os within these bounds.

The rule of avoiding empty I/Os is a "local" rule: to decide whether or not it is possible to do any further non-empty I/Os on a given vertex v it suffices to have information about the children of v (have they all been computed or not). Yet we showed that any computation strategy that follows this I/O-rule is guaranteed to be I/O-optimal up to a multiplicative factor that is particularly small when the proportion of input vertices is large and the maximal out-degree is small. For trees, this multiplicative factor is 3 and it can be made 2 by adding some other I/O-rules (computing subtrees strictly one after another).

This raises some questions: could our results be improved if we add more sophisticated and "global" rules (rules that take global properties into account, such as depth, width, or average degree)? Could we get tighter bounds or bounds that are tight on larger classes of DAGs, if we set rules that regulate how we choose at each step the next vertex that we compute? The following rules would be natural candidates:

- "Always continue to compute a vertex that has a maximal number of parents with red pebbles (among all vertices that have not been computed and whose parents have all been computed)."
- "Always continue to compute a vertex for whose computation the smallest number of load-operations needs to be done."

For computations of CDAGs on which any computation strategy without empty I/Os performs well, the problem of scheduling the computation I/O-efficiently may in practice not yet be trivial. This is because avoiding empty I/Os is in fact non-trivial, given that fast and slow memory usually do not communicate single values but whole blocks of them. Here, the problem becomes a problem of external memory data structures: how should we lay out the data in

external memory, such that for every block that we fetch, all data in this block is used at least once?

Acknowledgement

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme grant agreement No 101002047 and from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No.101034126.

References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[2] Lars Arge, Laura Toma, and Norbert Zeh. I/o-efficient topological sorting of planar dags. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 85–93, New York, NY, USA, 2003. ACM.

[3] Per Austrin, Toniann Pitassi, and Yu Wu. Inapproximability of treewidth, one-shot pebbling, and related layout problems. *CoRR*, abs/1109.4910, 2011.

[4] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *CoRR*, abs/0902.2537, 2009.

[5] Michael A. Bender, Rezaul Chowdhury, Alexander Conway, Martín Farach-Colton, Pramod Ganapathi, Rob Johnson, Samuel McCauley, Bertrand Simon, and Shikha Singh. The i/o complexity of computing prime tables. pages 192–206, 2016.

[6] Gianfranco Bilardi, Andrea Pietracaprina, and Paolo D'Alberto. On the space and access complexity of computation dags. In Ulrik Brandes and Dorothea Wagner, editors, *Graph-Theoretic Concepts in Computer Science*, pages 47–58, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[7] Timothy Carpenter, Fabrice Rastello, P. Sadayappan, and Anastasios Sidiropoulos. Brief announcement: Approximating the i/o complexity of one-shot red-blue pebbling. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 161–163, New York, NY, USA, 2016. ACM.

[8] Siu Man Chan. *Pebble Games and Complexity*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2013.

[9] Siu Man Chan, Massimo Lauria, Jakob Nordström, and Marc Vinyals. Hardness of approximation in PSPACE and separation results for pebble games. pages 466–485, 2015.

[10] Yi-Jen Chiang, Michael Goodrich, Edward Grove, Roberto Tamassia, Darren Vengroff, and Jeffrey Vitter. External-memory graph algorithms. 05 1995.

[11] Erik D. Demaine and Quanquan C. Liu. Inapproximability of the standard pebble game and hard to pebble graphs. pages 313–324, 2017.

[12] Erik D. Demaine and Quanquan C. Liu. Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 195–204, New York, NY, USA, 2018. ACM.

[13] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM J. Sci. Comput.*, 34(1):206–239, February 2012.

[14] M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik, and K. Yelick. A communication-optimal n-body algorithm for direct interactions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1075–1084, 2013.

[15] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data movement complexity of computational dags for parallel execution. *CoRR*, abs/1404.4767, 2014.

[16] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. pages 237–248, 1979.

[17] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. pages 326–333, 1981.

[18] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Softw.*, 12(3):249–264, September 1986.

[20] Quanquan Liu. Red-blue and standard pebble games : Complexity and applications in the sequential and parallel models. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Massachusetts, 2018.

[21] Anil Maheshwari and Norbert Zeh. A survey of techniques for designing i/o-efficient algorithms. pages 36–61, 01 2002.

[22] L. Marchal, S. McCauley, B. Simon, and F. Vivien. Minimizing i/os in out-of-core task tree scheduling. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 884–893, 2017.

[23] Ulrich Meyer and Norbert Zeh. I/o-efficient undirected shortest paths. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003*, pages 434–445, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[24] Desh Ranjan, John Savage, and Mohammad Zubair. Upper and lower i/o bounds for pebbling r-pyramids. In Costas S. Iliopoulos and William F. Smyth, editors, *Combinatorial Algorithms*, pages 107–120, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, , and Miquel Pericas. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(10), Oct. 2017.