Data Access Complexity: The Red/Blue Pebble Game Revisited

Technical Report OSU-CISRC-7/13-TR16
Revised: September 15, 2013

Venmugil Elango
The Ohio State University
elango.4@buckeyemail.osu.edu

Fabrice Rastello
Inria;
Université de Lyon;
LIP, UMR
CNRS/ENSL/UCBL1/Inria 5668
Fabrice.Rastello@inria.fr

Louis-Noël Pouchet
University of California at Los Angeles
pouchet@cs.ucla.edu

J. Ramanujam
Louisiana State University
jxr@ece.lsu.edu

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

Abstract

Technology trends will cause data movement to account for the majority of the energy expenditure as well as execution time on emerging/future computers. Therefore the computational complexity of algorithms will no longer be a sufficient metric for comparing algorithms, and a fundamental characterization of the data access complexity of algorithms will be increasingly important.

In this paper we revisit the problem of data access complexity (also called I/O complexity) in a two-level memory hierarchy, first addressed by the seminal work of Hong and Kung [16] using the formalism of the red/blue pebble game. It improves on prior work in several ways: 1) it enables the development of lower bounds on the I/O complexity of composite computations from lower bounds for a set of constituent sub-computations, 2) it develops a complementary graph min-cut based bounding strategy to Hong & Kung's S-partitioning approach, enabling tighter analytical lower bounds for some algorithms, and 3) it develops an automated approach to generate concrete lower bounds on the I/O complexity of arbitrary, possibly irregular computational directed acyclic graphs.

1. Introduction

Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational performance of processors relative to the speed of memory access. Because of the significant mismatch between computational latency and throughput when compared to main memory latency and bandwidth, the use of hierarchical memory systems and the exploitation of significant data reuse in the higher (i.e., faster) levels of the memory hierarchy is critical for high performance. Although hardware techniques for data pre-fetching and overlapping of computation with communication can alleviate the impact of memory access latency on performance, the mismatch between maximum computational rate and peak memory bandwidth is much more fundamental; *the only solution is to limit the volume of data movement to/from memory by enhancing data reuse in registers and higher levels of the cache.*

Thus the characterization of the data locality properties of algorithms is extremely important. Current approaches to computing locality metrics, e.g., stack reuse distance profiles [8, 10, 14, 20], only characterize an algorithm's implementation for a specific execution order corresponding to the given structure of the input program. It is highly desirable to characterize the inherent data locality properties of a program, allowing for all possible dependence-preserving reorderings of the elementary operations of the program.

Several techniques have been developed to maximize data locality and minimize communication for a given program, in particular through loop transformation frameworks [17, 30]. It is of interest to assess how the data access costs of a transformed program compare to an inherent lower bound for the computation. Characterizing a lower bound on the inherent data access complexity of a computation was addressed in the seminal work of Hong & Kung (where they called it I/O complexity; we will use both terms interchangeably) by using the model of the red/blue pebble game on a computational directed acyclic graph (CDAG) [16]. In this paper, we revisit the Hong & Kung approach for developing lower bounds on

the I/O complexity of CDAGs and extend the model and its application in three significant ways:

- **Decomposition of CDAGs**: The original Hong & Kung red/blue pebble game model of I/O complexity is not amenable to deriving I/O lower bounds for a CDAG by decomposing into component sub-CDAGs (this is explained in detail in Sec. 3). We develop a modified CDAG model and pebble game that enables additive combining of I/O lower bounds of decomposed parts of a CDAG. This property is critically important in enabling the analysis of real applications that are composed of a number of component algorithms with different computational structures and inherent I/O complexity.
- **Alternative Lower Bound Approach**: The lower bounds derivable by the Hong & Kung "2S-Partitioning" approach are weak for some algorithms because the model inherently does not account for the internal structure and operations within the components of the graph partition, but only dominators of incoming edges to the components. We develop an alternative lower bounding approach based on convex min-cut partitions of DAGs. We provide details of this approach in Sec. 4, demonstrating tighter analytical lower bounds for some algorithms compared to the Hong & Kung 2S-partitioning.
- **Lower Bounds for Arbitrary CDAGs**: The Hong & Kung model has so far been used for developing I/O lower bounds for only a small number of regular algorithms, with CDAG-structure-specific reasoning being required for each analyzed algorithm. We develop an automated approach for the estimation of I/O lower bounds of arbitrary, possibly irregular CDAGs. We provide details in Sec. 5 and experimental results in 6.

This paper is organized as follows. In Sec. 2 we provide some background on CDAGs and the approach of Hong & Kung for I/O complexity estimation. Sec. 3 identifies the limitations of the Hong & Kung red/blue pebble game model for analyzing composite applications comprised of multiple component algorithms, and presents a modified model of CDAGs and a pebble game to address the problem. Sec. 4 develops an alternative approach to Hong & Kung 2S-partitioning for estimating I/O lower bounds, using convex graph partitioning. Sec. 5 develops the automated analysis approach to characterizing the I/O complexity of arbitrary, irregular CDAGs. Sec. 6 presents experimental results for the automated heuristic approach. Sec. 7 discusses related work. Sec. 8 discusses some open problems raised by this work, and potential uses for the presented analysis approach.

2. Background

2.1 Computational Model

The model of computation we use is a computational directed acyclic graph (CDAG), where computational operations are represented as graph vertices and the flow of values between operations is captured by graph edges. Fig. 1 shows an example of a CDAG corresponding to a simple loop program. Two important characteristics of this abstract form of representing a computation are that (1) there is no specification of any particular order of execution of the operations: although the example program executes the operations in a specific sequential order, the CDAG abstracts the schedule of operations by only specifying partial ordering constraints as edges in the graph; (2) there is no association of memory locations with the source operands or result of any operation (labels in Fig. 1 are used simply to the show the correspondence between the loop code and its CDAG, but are not part of the formal description of the CDAG).

```
for (i = 1; i < 4; ++i)
  S += A[i-1] + A[i];
```

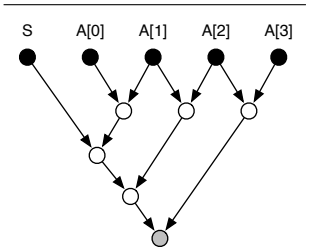


Figure 1: Example of a CDAG. Input vertices are represented in black, output vertices in grey.

We use the notation of Bilardi & Peserico [5] to formally describe CDAGs. We first describe the model of CDAG used by Hong & Kung:

DEFINITION 1 (CDAG-HK). *A computational directed acyclic graph (CDAG) is a 4-tuple $C = (I,V,E,O)$ of finite sets such that: (1) $I \subset V$ is the input set and all its vertices have no incoming edges; (2) $E \subseteq V \times V$ is the set of edges; (3) $G = (V,E)$ is a directed acyclic graph;(4) $V - I$ is called the operation set and all its vertices have one or more incoming edges; (5) $O \subseteq V$ is called the output set.*

2.2 The Red-Blue Pebble Game

Hong & Kung used this computational model in their seminal work [16]. The inherent I/O complexity of a CDAG is the minimal number of I/O operations needed while optimally playing the *red-blue pebble game*. The game uses two kinds of pebbles: a fixed number of red pebbles that represent the small fast local memory (could represent cache, registers, etc.), and an arbitrarily large number of blue pebbles that represent the large slow main memory. Starting with blue pebbles on all inputs nodes in the CDAG, the game involves the generation of a sequence of steps to finally produce blue pebbles on all outputs. A game is defined as follows.

DEFINITION 2 (Red-Blue pebble game [16]). *Given a CDAG $C = (I,V,E,O)$ such that any vertex with no incoming (resp.*

outgoing) edge is an element of I (resp. O), S red pebbles and an arbitrary number of blue pebbles, with a blue pebble on each input vertex. A complete game is any sequence of steps using the following rules that results in a final state with blue pebbles on all output vertices:

R1 (Input) *A red pebble may be placed on any vertex that has a blue pebble (load from slow to fast memory),*

R2 (Output) *A blue pebble may be placed on any vertex that has a red pebble (store from fast to slow memory),*

R3 (Compute) *If all immediate predecessors of a vertex of $V-I$ have red pebbles, a red pebble may be placed on that vertex (execution or "firing" of operation),*

R4 (Delete) *A red pebble may be removed from any vertex (reuse storage).*

The number of I/O operations for any complete game is the total number of moves using rules R1 or R2, that is the total number of data movements between the fast and slow memories. The inherent I/O complexity of a CDAG is the smallest number of such I/O operations that can be achieved, among all valid red-blue pebble games on that CDAG. An *optimal* red-blue pebble game is a game achieving this minimal number of I/O operations.

2.3 *S*-partitioning for lower bounds on I/O complexity

This red-blue pebble game provides an operational definition for the I/O complexity problem. However, it is not practically feasible to generate all possible valid games for large CDAGs. Hong & Kung developed a novel approach for deriving I/O lower bounds for CDAGs by relating the red-blue pebble game to a graph partitioning problem defined as follows.

DEFINITION 3 (Hong & Kung *S*-partitioning of a CDAG [16]). *Let $C=(I,V,E,O)$ be a CDAG. An S-partitioning of C is a collection of h subsets of V such that:*

P1 $\bigcap_{i=1}^{h} V_i = \emptyset$, *and* $\bigcup_{i=1}^{h} V_i = V$
P2 *there is no cyclic dependence between subsets*
P3 $\forall i,\ \exists D \in \mathsf{Dom}(V_i)$ *such that* $|D| \leq S$
P4 $\forall i,\ |\mathsf{Min}(V_i)| \leq S$

where a dominator set of V_i, $D \in \mathsf{Dom}(V_i)$ is a set of vertices such that any path from I to a vertex in V_i contains some vertex in D; the minimum set of V_i, $\mathsf{Min}(V_i)$ is the set of vertices in V_i that have all its children outside of V_i; and $|Set|$ is the cardinality of the set Set; A subset V_i is said to depend on subset V_j if there is an edge in E from a vertex in V_j to a vertex in V_i.

Hong & Kung showed a construction for a $2S$-partition of a CDAG, corresponding to any complete red-blue pebble game on that CDAG using S red pebbles, with a tight relationship between the number of vertex sets h in the $2S$-partition and the number of I/O moves q in the pebble-game:

THEOREM 1 (Pebble game, I/O and $2S$-partition [16]). *Any complete calculation of the red-blue pebble game on a CDAG using at most S red pebbles is associated with a 2S-partition of the CDAG such that*

$$S \times h \geq q \geq S \times (h-1),$$

where q is the number of I/O moves in the game and h is the number of subsets in the 2S-partition.

The tight association from the above theorem between any pebble game and a corresponding $2S$-partition provides the following key lemma that served as the basis for Hong & Kung's approach to deriving lower bounds on the I/O complexity of CDAGs.

LEMMA 1 (Lower bound on I/O [16]). *Let $H(2S)$ be the minimal number of vertex sets for any valid $2S$-partition of a given CDAG. Then the minimal number Q of I/O operations for any valid execution of the CDAG is bounded by*

$$Q \geq S \times (H(2S)-1)$$

This key lemma has been useful in proving I/O lower bounds for several CDAGs [16] by reasoning about the maximal number of vertices that could belong to any vertex-set in a valid $2S$-partition.

3. I/O Lower Bound using CDAG Decomposition

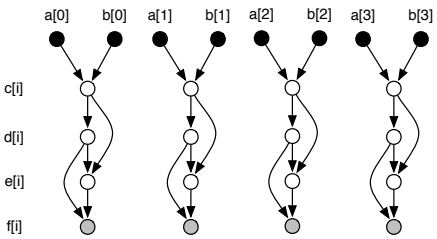
Application codes are typically constructed from a number of sub-computations using the fundamental composition mechanisms of sequencing, iteration and recursion. In contrast to analysis of computational complexity of such composite application codes, I/O complexity analysis poses challenges. With computational complexity, the operation counts of sub-computations can simply be added (either using concrete counts for a specific instance of the code for particular values of problem parameters, or by combining parametric expressions of the complexity of component sub-computations). For example, an application containing an outer loop over t iterations, with a dense matrix-matrix multiplication, a matrix vector product, and a dot-product in the loop body, has a computational complexity of $t \times (2N^3 + 2N^2 + 2N)$ operations. Using the red/blue pebble game model of Hong & Kung, as elaborated in Sec. 3.1, it is not feasible to develop lower bounds on the I/O complexity of a computation by combining I/O lower bounds of constituent sub-computations. We address this limitation by defining a modified model of the pebble game in Sec. 3.2 and formalize decomposition properties for I/O lower bounds in Sec. 3.3. A discussion comparing the original Hong & Kung model with the modified model is provided in Sec. 3.4.

3.1 Overview of the Problem and Solution Approach

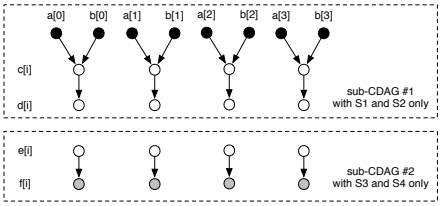
The Hong & Kung red/blue pebble game model places blue pebbles on all CDAG vertices without predecessors, since such vertices are considered to hold inputs to the computation,

```
for(i = 0; i < 4; i++)
  c[i] = a[i] + b[i]; // S1
for(i = 0; i < 4; i++)
  d[i] = c[i] * c[i]; // S2
for(i = 0; i < 4; i++)
  e[i] = c[i] + d[i]; // S3
for(i = 0; i < 4; i++)
  f[i] = d[i] * e[i]; // S4
```

(a) Original code



(b) Full CDAG



(c) CDAG partitioning

Figure 2: Example illustrating limitation of Hong & Kung model regarding composition of lower bounds from sub-components of CDAG

and therefore assumed to start off in slow memory. Similarly, all vertices without successors are considered to be outputs of the computation, and must have blue pebbles at the end of the game. If the vertices of a CDAG corresponding to a composite application are disjointly partitioned into sub-DAGs, the analysis of each sub-DAG under the Hong & Kung red/blue pebble game model will require the initial placement of blue pebbles on all predecessor-free vertices in the sub-DAG, and final placement of blue pebbles on all successor-free vertices in the sub-DAG. The optimal pebble game for each sub-DAG will require at least one load (R1) operation for each input and a store (R2) operation for each output. But in playing the red/blue pebble game on the full composite CDAG, clearly it may be possible to pass values in a red pebble between vertices in different sub-DAGs, so that the total I/O cost for the game on the full CDAG could be less than the sum of the I/O costs for the optimal pebble games for each sub-DAG. In fact, it is even possible for the I/O cost for a valid pebble game on the full DAG to be less than the optimal I/O cost for *each* sub-DAG.

Fig. 2(b) shows the CDAG for the computation in Fig. 2(a). Fig. 2(c) shows the CDAG partitioned into two sub-DAGs, where the first sub-DAG contains vertices of S1 and S2 (and the input vertices corrsponding to a[i] and b[i]), and the second sub-DAG contains vertices of S3 and S4. Considering the full CDAG, with just two red pebbles, it can be computed at an I/O cost of 12, incurring I/O just for the initial loads of inputs a[i] and b[i], and the final stores for outputs f[i]. In contrast, with the partitioned sub-DAGs, the first sub-DAG will incur additional output stores for the successor-free vertices S2[i], and the second sub-DAG will incur input loads for predecessor-free vertices S3[i]. Thus the sum of optimal red/blue pebble game I/O costs for the two sub-DAGs amounts to 20 moves, i.e., it exceeds the optimal I/O cost for the full CDAG.

The above example illustrates a fundamental problem with the Hong & Kung red/blue pebble game model: it is infeasible to combine I/O lower bounds for sub-CDAGs of a CDAG to generate an I/O lower bound for the composite CDAG. It is not even possible to assert that the maximum among the I/O lower bounds of sub-CDAGs of a CDAG is a valid lower bound for the composite CDAG.

The ability to perform complexity analysis by combining analyses of component sub-computations is critical to the analysis of real applications. In order to enable such decomposition of complexity analysis, we make two changes to the Hong & Kung pebble game model, one relaxation, and one restriction:

1. **Flexible input/output vertex labeling:** Unlike the Hong & Kung model, where all vertices without predecessors must be input vertices, and all vertices without successors must be output vertices, the modified model allows flexibility in indicating which vertices are labeled as inputs and outputs. In the modified variant of the pebble game, predecessor-free vertices that are not designated as input vertices do not have an initial blue pebble placed on them. However, such vertices are allowed to fire using rule R3 at any time, since they do not have any predecessor nodes without red pebbles. Vertices without successors that are not labeled as output vertices do not require placement of a blue pebble at the end of the game. However, all compute vertices in the CDAG are required to have fired for any complete game.
2. **Prohibition of multiple evaluations of compute vertices:** In the modified pebble game, recomputation of values are prohibited on the CDAG, i.e., each vertex is only allowed to "fire" once using rule R3. Several other efforts [3–5, 9, 12, 18, 21–27] have also imposed a restriction to disallow recomputation in the pebble game model. While such a model is indeed more restrictive than the original Hong & Kung model, as explained later in this section, this restriction enables the development of techniques to form tighter lower bounds.

We proceed by first defining a modified pebble game where recomputation is disallowed, followed by relaxation to the CDAG model regarding inputs/output vertices.

3.2 Disallowing Recomputation: The Red-Blue-White Pebble Game

The red-blue pebble game model used by Hong & Kung implicitly permits recomputation of vertices in the CDAG, i.e., it is possible to place a red pebble on a vertex by use of rule R3, then remove the red pebble using rule R4, and at a later time place a red pebble at the same vertex again by use of rule R3. Multiple firings of a vertex by use of rule R3 represent recomputation of the same value in a CDAG multiple times and may in some situations be beneficial because it avoids storing and reloading the computed value. An alternative model has also been used by several works [3–5, 9, 12, 18, 21–27] where each value may only be computed once, i.e., recomputation is not permitted. Our adaptation of the standard red-blue pebble game to model computation of CDAGs without recomputation involves the use of an additional kind of pebble, white pebble. A white pebble is initially placed on all input vertices (in addition to blue pebbles) and any vertex as soon as it fires using rule R3. Rule R3 is modified to disallow firing of any vertex that already has a white pebble on it. The Red-Blue-White pebble game is defined as follows.

DEFINITION 4 (Red-Blue-White (RBW) pebble game). *Given a CDAG $C = (I,V,E,O)$ such that any vertex with no incoming (resp. outgoing) edge is an element of I (resp. O), S red pebbles and an arbitrary number of blue and white pebbles, with a blue pebble on each input vertex, a complete game is any sequence of steps using the following rules that results in a final state with blue pebbles on all output vertices:*

R1 (Input) *A red pebble may be placed on any vertex that has a blue pebble.*

R2 (Output) *A blue pebble may be placed on any vertex that has a red pebble.*

R3 (Compute) *If a vertex v does not have a white pebble and all its immediate predecessors have red pebbles on them, a red pebble along with a white pebble may be placed on v.*

R4 (Delete) *A red pebble may be removed from any vertex.*

First, Definition 3 is adapted to this new game so that Theorem 1 and thus Lemma 1 hold for the RBW pebble game.

DEFINITION 5 (S-partitioning of CDAG – RBW pebble game). *Let $C = (I,V,E,O)$ be a CDAG. An S-partitioning of C is a collection of h subsets of $V - I$ such that:*

P1 $\bigcap_{i=1}^{h} V_i = \emptyset$, *and* $\bigcup_{i=1}^{h} V_i = V - I$

P2 *there is no cyclic dependence between subsets*

P3 $\forall i,\ |In(V_i)| \leq S$

P4 $\forall i,\ |Out(V_i)| \leq S$

where the input set of V_i, $In(V_i)$ is the set of vertices of $V - V_i$ that have at least one child in V_i; the output set of V_i, $Out(V_i)$ is the set of vertices of V_i also part of the output set O or that have at least one child outside of V_i.

We now prove Theorem 1 for the Red-Blue-White pebble game.

Proof. Consider a pebble game instance C that corresponds to some scheduling (i.e., execution) of the vertices of the graph $G = (V,E)$ that follows the rules R1–R4 of the Red-Blue-White pebble game (see Definition 6 in Sec. 3.2). We view this pebble game instance as a string that has recorded all the transitions (applications of pebble game rules). Suppose that $\mathcal{P}$ contains exactly q transitions of type $R1$ or $R2$. Let $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_h$ correspond to a partitioning of the transitions of $\mathcal{P}$ into $h = \lceil q/S \rceil$ consecutive sub-sequences such that each $\mathcal{P}_i$ in $\mathcal{P}_1, \ldots, P_{h-1}$ contains exactly S transitions of type $R1$ or $R2$.

Because the CDAG contains no node isolated from the output nodes, and because of the white pebbles, any vertex of $V - I$ is computed exactly once in $\mathcal{P}$. Let V_i be the set of vertices computed (transition $R3$) in the sub-calculation P_i. Property $P1$ is trivially fulfilled.

As transition $R3$ on a vertex v is possible only if its predecessor vertices have a red pebble on them, those predecessors are necessarily executed in some P_j, $j \leq i$ and are thus part of a V_j, $j \leq i$. This proves property $P2$.

To prove $P3$ for a given V_i we consider two sets: V_R is the set of vertices that had a red pebble on them just before the execution of P_i; V_{BR} is the set of vertices on which a red pebble is placed according to rule $R1$ (input) during P_i. We have that $\mathsf{In}(V_i) \subseteq V_R \cup V_{BR}$. Thus $|\mathsf{In}(V_i)| \leq |V_R| + |V_{BR}|$. As there cannot simultaneously be more than S red pebbles, $|V_R| \leq S$; also by construction of P_i, $|V_{BR}| \leq S$. This proves that $|\mathsf{In}(V_i)| \leq 2S$ (property $P3$).

Property $P4$ is proved in a similar way: V'_R is the set of vertices that have a red pebble on them just after the execution of P_i; V'_{RB} is the set of vertices of V_i on which a blue pebble is placed during P_i according to rule $R2$. We have that $\mathsf{Out}(V_i) \subseteq V'_R \cup V'_{RB}$. Thus $|\mathsf{Out}(V_i)| \leq |V'_R| + |V'_{RB}|$. As there cannot simultaneously be more than S red pebbles, $|V'_R| \leq S$; also by construction of P_i, $|V'_{RB}| \leq S$. This proves that $|\mathsf{Out}(V_i)| \leq 2S$ (property $P4$). □

3.3 Decomposition

Fig. 2 illustrated the decomposition problem with the Hong & Kung RB pebble game model. Since any vertex without predecessors was always an input vertex with an initial blue pebble, it was not feasible to decompose a larger CDAG and combine I/O lower bounds from sub-CDAGs of the larger CDAG. We introduce an adaptation of the Hong & Kung CDAG model to overcome the decomposition problem. Here, we allow flexibility in specifying which vertices of a CDAG are tagged as input and output vertices. Thus, vertices without any predecessors are not forced to be input vertices with initial blue pebbles. Similarly, vertices without successors are also not required to be output vertices. The motivation is to develop a model where "interior" computations in a sub-CDAG of a larger CDAG can be analyzed without forcing an initial R1 load transition on predecessor-free vertices of the sub-CDAG or a R2 store transition for successor-free vertices of the sub-CDAG.

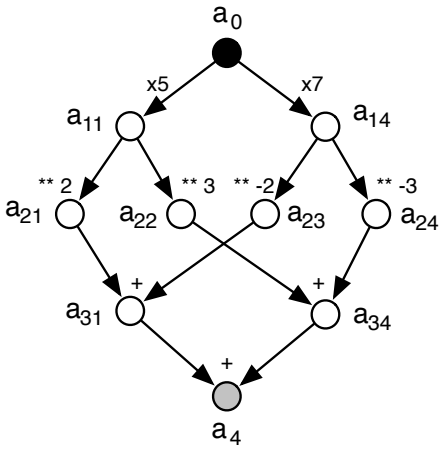


Figure 3: CDAG with large interior vertices being decomposed into two sub-CDAGs

Fig. 3 shows one such CDAG. The computation begins from a single input, expands to a large number of intermediate values and finally reduces to a single output. Application of *S*-partitioning to such CDAGs will lead to a trivial partition with just a single set. Hence, the CDAG is decomposed into two sub-graphs as illustrated in fig. 4 to expose the "interior" vertices to obtain a tighter lower bound as described later in this section.

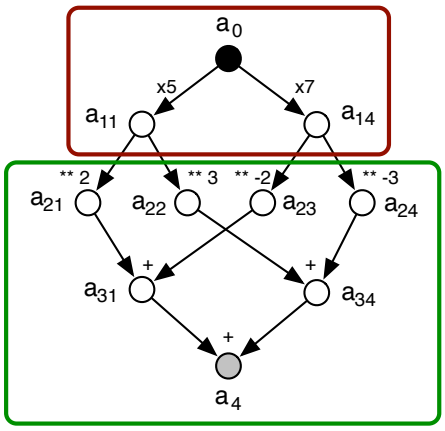


Figure 4: CDAG with large interior vertices being decomposed into two sub-CDAGs

In order to ensure that all computations on such a CDAG proceed even when there are no designated input and output vertices, we modify the rules of the RBW pebble game as follows.

DEFINITION 6 (RBW pebble game: Flexible I/O vertex model). *Given a CDAG $C = (I,V,E,O)$, S red pebbles and an arbitrary number of blue and white pebbles, with a blue pebble on each input vertex, a complete game is any sequence of steps using the following rules that results a final state with white pebbles on all vertices and blue pebbles on all output vertices:*

R1 (Input) *A red pebble may be placed on any vertex that has a blue pebble; a white pebble is also placed along with the red pebble, unless the vertex already has a white pebble on it.*

R2 (Output) *A blue pebble may be placed on any vertex that has a red pebble.*

R3 (Compute) *If a vertex v does not have a white pebble and all its immediate predecessors have red pebbles on them, a red pebble along with a white pebble may be placed on v.*

R4 (Delete) *A red pebble may be removed from any vertex (reuse storage).*

In the modified rules for the RBW game, all vertices are required to have a white pebble at the end of the game, thereby ensuring that the entire CDAG is evaluated. Non-input vertices without predecessors do not have an initial blue pebble on them, but they are allowed to fire using rule R3 at any time – since they have no predecessors, the condition in rule R3 is trivially satisfied. But if all successors of such a node cannot be fired while maintaining a red pebble, "spilling" and reloading using R2 and R1 is forced because the vertex cannot be fired again using R3.

The above modification for flexible input/output in Def. 6 does not change the complexity model of the RBW game from Def. 4; it is in essence the same game, but enabling it to be played on CDAGs with possibly no designated input/output vertices, while still requiring the execution of all computation vertices in the graph. Such a refinement is needed to define proper decomposition rules, as shown below. We note that the modified S-partitioning from Def. 5 and the associated I/O complexity reasoning is also applicable to the flexible I/O RBW pebble game. For (sub-)graphs without input/output sets, the application of S-partitioning will however lead to a trivial partition with all vertices in a single set (e.g., $h = 1$). A careful tagging of vertices as virtual input/output nodes will be required for better I/O complexity estimates, as detailed below and later in Sec. 5.

Definition 6 allows us to partition a CDAG C into sub-CDAGs $C_1, C_2, \ldots, C_p$, to compute lower bounds on the I/O complexity of each sub-CDAG $IO(C_1), IO(C_2), \ldots, IO(C_p)$ independently and simply add them to bound the I/O complexity of C. This is stated in the following decomposition theorem.

THEOREM 2 (Decomposition). *Let $C = (I,V,E,O)$ be a CDAG. Let $V_1, V_2, \ldots, V_p$ be an arbitrary (non necessarily acyclic) partitioning of V ($i \neq j \Rightarrow V_i \cap V_j = \emptyset$ and $\bigcup_{1 \leq i \leq p} V_i = V$) and $C_1, C_2, \ldots, C_p$ be the induced partitioning of C ($I_i = I \cap V_i$, $E_i = E \cap V_i \times V_i$, $O_i = O \cap V_i$). Then*

$$\sum_{1 \leq i \leq p} IO(C_i) \leq IO(C).$$

In particular, if Q_i is a lower bound on the IO of C_i, then $\sum_{1\leq i\leq p} Q_i$ is a lower bound on the I/O of C.

Proof. Consider an optimal valid game $\mathcal{P}$ for C, with cost $Q = IO(C)$. We define the cost of $\mathcal{P}$ restricted to V_i, denoted as $Q_{|V_i}$, as the number of $R1$ or $R2$ transitions in $\mathcal{P}$ that involve a vertex of V_i. Clearly $Q = \sum_{1\leq i\leq p} Q_{|V_i}$. We can build from $\mathcal{P}$, a valid game $\mathcal{P}_{|V_i}$ for C_i, of cost $Q_{|V_i}$. This will prove that $IO(C_i) \leq Q_{|V_i}$, and thus $\sum_{1\leq i\leq p} IO(C_i) \leq \sum_{1\leq i\leq p} Q_{|V_i} = Q = IO(C)$. $\mathcal{P}_{|V_i}$ is built from $\mathcal{P}$ as follows [1]: (1) for any transition in $\mathcal{P}$ that involves a vertex v in V_i, apply this transition in $\mathcal{P}_{|V_i}$; (2) delete all other transitions in $\mathcal{P}$. Conditions for transitions $R1$, $R2$, and $R4$ are trivially satisfied. Whenever a transition $R3$ on a vertex v is performed in $\mathcal{P}$, all the predecessors of v must have a red pebble on them. Since all transitions of $\mathcal{P}$ on the vertices of V_i are maintained in $\mathcal{P}_{|V_i}$, when v is executed in $\mathcal{P}_{|V_i}$, all its predecessor vertices must have red pebbles, enabling transition $R3$. □

I/O complexity with modified Input/Output sets As developed later in this paper, with a divide-and-conquer approach of partitioning a CDAG into disjoint sub-CDAGs, we can use two different techniques to develop a lower bound on the I/O complexity of the sub-CDAGs: one based on the 2S-Partitioning technique of Hong & Kung (Lemma 1), and a new technique using convex min-cuts on graphs (Lemma 2, developed in the next section). These two approaches require different strategies for handling of input/output vertices in order to get good lower bounds. Below, we develop bounding relations between the I/O complexity of a CDAG and a variant of it that has some inputs/outputs modified in specific ways.

COROLLARY 1 (Input/Output Deletion). *Let C and C' be two CDAGs: $C' = (I \cup dI, V \cup dI \cup dO, E', O \cup dO)$, $C = (I, V, E' \cap V \times V, O)$. Then $IO(C')$ can be bounded by a lower bound of $IO(C)$ as follows:*

$$IO(C) + |dI| + |dO| \leq IO(C') \tag{1}$$

Proof. The proof involves a direct application of the decomposition theorem, with $V_1 = dI$, $V_2 = V$, and $V_3 = dO$. Indeed, the I/O complexity of the CDAG made of only dI is exactly $|dI|$ (transition $R1$ is necessary for each of them to place a white pebble on it); the I/O complexity of the CDAG made only of dO is exactly $|dO|$ (transition $R2$ is necessary for each of them to place a blue pebble on them). □

Reciprocally, as discussed below, $IO(C)$ can generally not be bounded tightly by a lower bound of $IO(C')$. However, suppose we start from a CDAG with no input/output vertices, and to each free-predecessor/successor vertex we create an input/output vertex. As the out-degree of any element of dI and the in-degree of any element of dO are exactly one, a bound relating the I/O complexity of the initial graph to the I/O complexity of the modified graph can established:

THEOREM 3 (Input/Output Insertion – RBW pebble game). *Let C and C' be two CDAGs: $C' = (I \cup dI, V \cup dI \cup dO, E', O \cup dO)$, $C = (I, V, E' \cap V \times V, O)$ such that (i) the out-degree of any input vertex in dI is 1 in C' and different input vertices in dI connect to distinct vertices; and (ii) the in-degree of any output vertex of dO is 1 in C'. Then $IO(C)$ with S red pebbles can be bounded by a lower bound of $IO(C')$ with a game with $S+1$ red pebbles as follows:*

$$IO_{S+1}(C') - |dI| - |dO| \leq IO_S(C) \tag{2}$$

where IO_S represents the I/O complexity with S pebbles.

Proof. The inequality is proved by considering an optimal valid game $\mathcal{P}$ for C with S pebbles, of cost $IO_S(C)$. We will build a valid game $\mathcal{P}'$ with $S+1$ pebbles, of cost $IO_S(C) + |dI| + |dO|$. We build $\mathcal{P}'$ from $\mathcal{P}$ as follows: (1) any transition in $\mathcal{P}$ is reported in $\mathcal{P}'$; (2) for each (unique) successor w of an input $v \in dI$, the first (and unique) transition $R3$ involving w is prefixed by $R1(v)$ and postfixed by $R4(v)$; (3) for each (unique) predecessor w of an output $v \in dO$, the first (and unique) transition $R3$ involving w is postfixed by the sequence of $R3(v); R2(v); R4(v)$. □

There are cases where separating input/output vertices leads to very weak lower bounds. This happens when input vertices have high fan-out (Inequality 2 cannot hold) such as for matrix multiplication: in the CDAG for matrix multiplication if we remove all input and output vertices, we get a set of independent chains that can each be computed with no more than 2 red pebbles. To overcome this problem, we develop the following theorem that allows us to compare the I/O of two CDAGs: a CDAG $C' = (I', V, E, O')$ and another $C = (I, V, E, O)$ built from C' by just transforming some vertices without predecessors into input vertices, and some others into output nodes so that $I' \subset I$ and $O' \subset O$. In contrast to the prior development above, instead of adding/removing input/output vertices, here we do not change the vertices of a CDAG but instead only change the labeling (i.e., the tag) of some vertices as inputs/outputs in the CDAG. So the DAG remains the same, but some input/output vertices are relabeled as standard computational vertices and vice-versa.

THEOREM 4 (Input/Output Untagging – RBW pebble game). *Let C and C' be two CDAGs of the same DAG $G = (V, E)$: $C = (I, V, E, O)$, $C' = (I \cup dI, V, E, O \cup dO)$. Then, $IO(C')$ can be bounded by a lower bound on $IO(C)$ as follows (untagging):*

$$IO(C) \leq IO(C') \tag{3}$$

Proof. Consider an optimal valid game say $\mathcal{P}'$ for C', of cost $IO(C')$. We will build a valid game $\mathcal{P}$ for C, of cost no more than $IO(C')$. This will prove that $IO(C) \leq IO(C')$. We build $\mathcal{P}$ from $\mathcal{P}'$ as follows: (1) for any input vertex $v \in dI$, the first

[1] We do the reasoning here for the red-blue-white pebble game. Similar reasoning can also be done for the red-blue pebble game.

transition $R1$ involving v in $\mathcal{P}'$ is replaced in $\mathcal{P}$ by a transition $R3$ followed by a transition $R2$; (2) any other transition in $\mathcal{P}'$ is reported as is in $\mathcal{P}$. □

THEOREM 5 (Input/Output Tagging – RBW pebble game). *Let C and C' be two CDAGs of the same DAG $G = (V,E)$: $C = (I,V,E,O)$, $C' = (I \cup dI, V, E, O \cup dO)$. Then, $IO(C)$ can be bounded by a lower bound on $IO(C')$ as follows (tagging):*

$$IO(C') - |dI| - |dO| \leq IO(C) \tag{4}$$

Proof. Consider an optimal valid game $\mathcal{P}$ for C, of cost $IO(C)$. We will build a valid game $\mathcal{P}'$ for C', of cost no more than $IO(C) + |dI| + |dO|$. This will prove that $IO(C') \leq IO(C) + |dI| + |dO|$. We build $\mathcal{P}'$ from $\mathcal{P}$ as follows: (1) for any input vertex $v \in dI$, the first (and only) transition $R3$ involving v in $\mathcal{P}$ is replaced in $\mathcal{P}'$ by a transition $R1$; (2) for any output vertex $v \in dO$, the last transition $R3$ involving v in $\mathcal{P}$ is complemented by an $R2$ transition; (3) any other transition in $\mathcal{P}$ is reported as is in $\mathcal{P}'$. □

Fig. 5(a) shows the same CDAG as in fig. 3, with the input vertex removed and the output vertex untagged. In fig. 5(b), the free-predecessor vertices are tagged as input vertices.

3.4 RBW versus RB pebble games

All the theorems stated in this Section 3.3 considered a red-blue-white pebble game. Although the proof of Theorem 2 (decomposition) has been written for an RBW pebble game, it is easy to check that it also holds for an RB pebble game. For that reason, Corollary 1 that allows us to derive a bound for a CDAG C' from the bound of a CDAG C obtained by deleting some input/output vertices also holds when recomputation is allowed. Similarly, we could check that its untagging version (Theorem 4) would also hold. As we will see later in this paper, deletion/untagging is useful when the I/O complexity of a CDAG is to be derived from its *inner I/O complexity*. Note that the purpose of Sec. 4 is to develop the so called min-cut approach that evaluates the inner I/O complexity.

Actually, there are many cases where after decomposing a CDAG C into pieces $C_1, \ldots, C_p$, each sub-CDAG has to be enriched with input/output nodes. This happens whenever the sub-CDAG C_i has an I/O complexity that comes from the nature of its boundary (high fan-out input vertices) and that cannot be derived from its inner complexity. To illustrate this point, consider an initial CDAG made up of the outer product of two vectors $A = u \otimes v$ followed by the square of the obtained matrix $B = A^2$. The I/O complexity of this CDAG is quite inhomogeneous: (1) taking the outer product alone, its complexity comes from its output, so it is $\Omega(n^2)$; (2) taking the matrix-matrix multiplication alone, its complexity comes from the extensive reuse of its inputs, and is $\Omega(n^3/\sqrt{S})$. We would like, on this example, to be able to decompose the full CDAG into one made up of the outer product and one made up of the matrix-matrix multiplication. But the I/O complexity of this latter comes from the sharing by many computational vertices of its input vertices. This means that, technically, we need to tag the A vertices as inputs (or insert a predecessor input to each A vertex), compute the I/O complexity of this modified CDAG, and then derive the I/O complexity of the initial CDAG from this. This is possible for the RBW pebble game thanks to Theorem 5 (or Theorem 3). We would be able to conclude that the I/O is the sum of $\Omega(n)$ (outer product with no commit of the outputs), plus $\Omega(n^3/\sqrt{S} - n^2)$ (matrix-matrix multiplication with no inputs using Theorem 5).

This "trick" turns out to be not possible for the RB pebble game: there are no equivalent versions of theorems 3 and 5. Indeed inserting/tagging nodes as input can lead to an I/O complexity arbitrarily larger the actual I/O complexity of the initial graph. Our current example illustrates this point. The RB I/O complexity of a matrix multiplication of A by B where A and B can be computed for free (not tagged as inputs) would be n^2 (the size of the output), an order of magnitude less than its version where A and B are tagged as inputs. The I/O complexity of the combined operations (outer product followed by matrix square) where computations can be interleaved and where A is not forced to be stored in memory, turns out to be something in between: $\Omega(n^3/S)$.

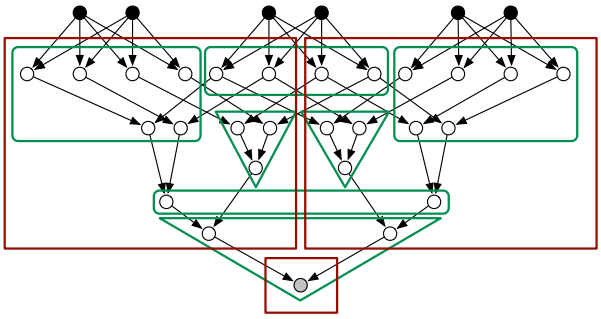
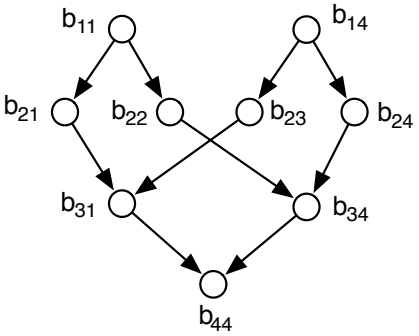
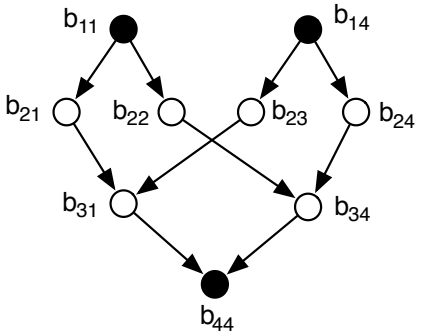


Figure 6: Example of $2S$-partitioning using RB and RBW pebble games, shown in red and green respectively.

There are many cases where actually allowing recomputation into the game does not help the I/O complexity. This point is illustrated by the example in fig. 6. The red and green boxes show the minimal vertex-set $2S$-partitions (for $S = 2$) generated with RB and RBW pebble game rules respectively. Permitting recomputation allows us to create $2S$-partition with as low as two subsets, which provides us a lower-bound of two. But as we can see from the figure, we clearly need to spill much more vertices even with recomputation. In this case, RBW game's $2S$-partition provides us a tighter and more accurate lower bound. This is the case for all examples developed in Hung and Kung's paper [16] that include odd-even transposition sort, FFT, matrix-matrix multiplication, and product graph. For those applications, the lower bound (LB_{RB} – that uses the RB pebble game) asymptotically matches a known upper bound that does not use any recomputation (denoted as



(a) CDAG with input removed



(b) Tagged CDAG

Figure 5: Illustration of Input/Output set modification and vertex Tagging/Untagging

UB_{RBW}). The following inequalities trivially hold:

$$LB_{RB} \leq IO_{RB} \leq IO_{RBW} \leq UB_{RBW}.$$

So, whenever $\Omega(LB_{RB}) = \Omega(UB_{RBW})$ we can conclude that

$$\Omega(IO_{RB}) = \Omega(IO_{RBW}).$$

4. Min-Cut for I/O Complexity Lower Bound

In this section, we develop an alternative lower bounding approach to the Hong & Kung 2S-partitioning. It is motivated from the observation that the 2S-partitioning approach does not account for the internal structure of a CDAG, but essentially only on the boundaries of the partitions. In contrast, we develop an approach that captures internal space requirements using the abstraction of wavefronts.

4.1 The Min-Cut based Approach

We first present needed definitions. Given a graph $G = (V,E)$, a cut is defined as any partition of the set of vertices V into two parts $\mathcal{S}$ and $\mathcal{T} = V - \mathcal{S}$. An $s-t$ cut is defined with respect to two distinguished vertices s and t and is any $(\mathcal{S},\mathcal{T})$ cut satisfying the requirement that $s \in \mathcal{S}$ and $t \in \mathcal{T}$. Each cut defines a set of cut edges (the cut-set), i.e., the set of edges (u,v) where $u \in \mathcal{S}$ and $v \in \mathcal{T}$. The capacity of a cut is defined as the sum of the weights of the cut edges. The minimum cut problem (or min-cut) is one of finding a cut that minimizes the capacity of the cut. We define vertex u as a cut vertex with respect to an $(\mathcal{S},\mathcal{T})$ cut, as a vertex $u \in \mathcal{S}$ that has a cut edge incident on it. A related problem of interest for this paper is the *vertex min-cut* problem which is one of finding a cut that minimizes the number of cut vertices.

Given a DAG $G = (V,E)$ and some vertex $x \in V$, the set $\mathsf{Anc}(x)$ is the set of vertices from which there is a non-empty directed path to x in G ($x \notin \mathsf{Anc}(x)$); the set $\mathsf{Desc}(x)$ is the set of vertices to which there is a non-empty directed path from x in G ($x \notin \mathsf{Desc}(x)$). Using those two notions, we consider a convex cut $(\mathcal{S}_x,\mathcal{T}_x)$ associated to x as follows: $\mathcal{S}_x$ includes $x \cup \mathsf{Anc}(x)$; $\mathcal{T}_x$ includes $\mathsf{Desc}(x)$; in addition, $\mathcal{S}_x$ and $\mathcal{T}_x$ must be constructed such that there is no edge from $\mathcal{T}_x$ to $\mathcal{S}_x$. With this, the sets $\mathcal{S}_x$ and $\mathcal{T}_x$ partition the graph G into two convex partitions. We define the wavefront induced by $(\mathcal{S}_x,\mathcal{T}_x)$ to be the set of vertices in $\mathcal{S}_x$ that have at least one outgoing edge to a vertex in $\mathcal{T}_x$.

Consider a pebble game instance $\mathcal{P}$ that corresponds to some scheduling (i.e., execution) of the vertices of the graph $G = (V,E)$ that follows the rules R1–R4 of the Red-Blue-White pebble game (see Definition 6 in Sec. 3.2). We view this pebble game instance as a string that has recorded all the transitions (applications of pebble game rules). Given $\mathcal{P}$, we define the *wavefront* $W_{\mathcal{P}}(x)$ induced by some vertex $x \in V$ at the point when x has just fired (i.e., a white pebble has just been placed on x) as the union of x and the set of vertices $u \in V$ that have already fired and that have an outgoing edge to a vertex $v \in V$ that have not fired yet. Viewing the instance of the pebble game $\mathcal{P}$ as a string, $W_{\mathcal{P}}(x)$ is the set of vertices x and those white-pebbled vertices to the left of x in the string associated with $\mathcal{P}$ that have an outgoing edge in G to not-white-pebbled vertices that occur to the right of x in $\mathcal{P}$. With respect to a pebble game instance $\mathcal{P}$, the set $W_{\mathcal{P}}(x)$ defines the memory requirements at the time-stamp just after x has fired.

Note that there is a one-to-one correspondence between the wavefront $W_{\mathcal{P}}(x)$ induced by some vertex $x \in V$ and the $(\mathcal{S}_x,\mathcal{T}_x)$ partition of the graph G. For a valid convex partition $(\mathcal{S}_x,\mathcal{T}_x)$ of G, we can construct a pebble game instance $\mathcal{P}$ in which at the time-stamp when x has just fired, the subset of vertices of V that are white pebbled exactly corresponds to $\mathcal{S}_x$; the set of fired (white-pebbled) nodes that have a successor that is not white-pebbled constitute a wavefront $W_{\mathcal{P}}(x)$ associated with x. Similarly, given wavefront $W_{\mathcal{P}}(x)$ associated with x in a pebble game instance $\mathcal{P}$, we can construct a valid $(\mathcal{S}_x,\mathcal{T}_x)$ convex partition by placing all white pebbled vertices in $\mathcal{S}_x$ and all the non-white-pebbled vertices in $\mathcal{T}_x$.

A minimum cardinality wavefront induced by x, denoted $W_G^{\min}(x)$ is a vertex min-cut that results in an $(\mathcal{S}_x,\mathcal{T}_x)$ par-

tition of G defined above. We define $w_G^{\max}$ as the maximum value over the size of all possible minimum cardinality wavefronts associated with vertices, i.e., define $w_G^{\max} = \max_{x \in V} \left(|W_G^{\min}(x)| \right)$.

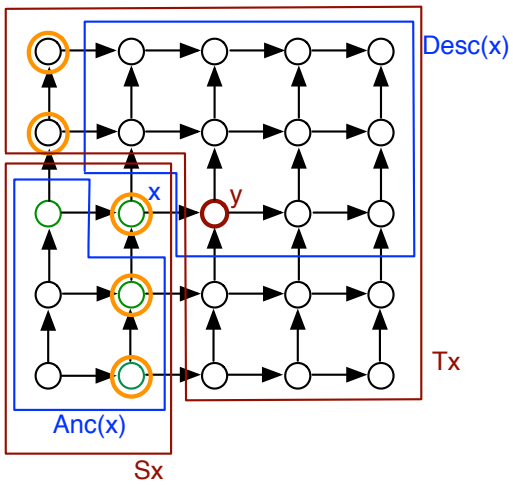


Figure 7: Illustration of mincut based approach

Fig. 7 illustrates some of these ideas on a Diamond DAG. The vertices colored in green represent the wavefront $W_G^{\min}(x)$ induced by the partition $(\mathcal{S}_x, \mathcal{T}_x)$. The maximum value, $w_G^{\max}$, is obtained from the wavefront associated with the vertex y.

LEMMA 2. *Let $C = (\emptyset, V, E, O)$ be a CDAG with no inputs.*
For any $x \in V$, $2\left(|W_G^{min}(x)| - S\right) \leq IO(C)$.
In particular, $2\left(w_G^{max} - S\right) \leq IO(C)$.

Proof. Let x be a vertex in V. Consider a pebble game instance $\mathcal{P}$ of cost $IO(C)$. Let the wavefront induced by the vertex x in $\mathcal{P}$ be $W_{\mathcal{P}}(x)$. Since every vertex in $W_{\mathcal{P}}(x)$ has a successor that is not yet white-pebbled, they must have either a red or a blue pebble on them. Recall that we have S red pebbles. Therefore, at least $|W_{\mathcal{P}}(x)| - S$ white-pebbled vertices have a blue pebble on them. These vertices will have to be red-pebbled at some point in the future and will incur at least $|W_{\mathcal{P}}(x)| - S$ loads after x fires. In addition, as C has no input vertices, those $|W_{\mathcal{P}}(x)| - S$ vertices have been blue pebbled using rule $R2$. Therefore, at least $|W_{\mathcal{P}}(x)| - S$ stores must have happened before x fired in $\mathcal{P}$. Thus the total number of loads and stores $IO(C)$ is at least $2(|W_{\mathcal{P}}(x)| - S)$ which can itself be bounded using the vertex min-cut associated to x:
$2\left(|W_G^{\min}(x)| - S\right) \leq 2\left(|W_{\mathcal{P}}(x)| - S\right) \leq IO(C)$. □
If applied to the whole CDAG, Lemma 2 will usually lead to a very weak bound. To overcome this limitation, the idea is to decompose it into smaller sub CDAGs, and sum up their individual I/Os. The following theorem formalizes this approach:

THEOREM 6 (Min-Cut with divide and conquer). *Let $C = (I, V, E, O)$ be a CDAG. Let $V_1, \ldots, V_p$ be a (non-necessarily acyclic) partitioning of V, and $C_1, \ldots, C_p$ be the induced partitioning of C ($I_i = V_i \cap I$, $E_i = E \cap V_i \times V_i$, $O_i = O \cap V_i$). Let for each i, $C_i' = (\emptyset, V_i', E_i', \emptyset)$ be the sub DAG obtained from C_i by deleting all input and output vertices ($V_i' = V_i - I_i - O_i$, $E_i' = E_i \cap V_i' \times V_i'$, and $G_i' = (V_i', E_i')$). Then the minimum I/O of C can be bounded by:*

$$\sum_{i=1}^{p} 2\left(w_{G_i'}^{max} - S\right) + |I| + |O| \leq IO(C)$$

Proof. Theorem 2 states that $\sum_i IO(C_i) \leq IO(C)$. For each C_i, Corollary 1 states that $IO(C_i') + |I_i| + |O_i| \leq IO(C_i)$. By construction $|I| = |\cup_i I_i| = \sum_i |I_i|$, and $|O| = |\cup_i O_i| = \sum_i |O_i|$. Finally Lemma 2 states that for each i, $2\left(w_{G_i'}^{\max} - S\right) \leq IO(C_i')$. □

4.2 Using Min-Cut Approach for Deriving Analytical Bounds

Up to now we presented two different approaches for studying/computing the I/O complexity of a given application using red-blue-white pebble game model. Note that, as we will discuss further, for all applications we know for which the I/O complexity have been studied, the red-blue-white pebble game model does not allow any better asymptotic complexity than with the red-blue pebble game. This remark concerns in particular the two different applications used in this section to illustrate the min-cut approach: Fast-Fourier-Transform (FFT), and Diamond-DAG. For each of them we propose to perform the I/O complexity study (with exact constant factors) using Lemma 1 (of [16]) or Theorem 6. Following the above remark, when a proof already exists that uses Lemma 1, we will refer to it instead of adapting it to the red-blue-white pebble game (another intuitive remark to motivate this choice is that for each of these applications, the In set of a vertex set turns out to be a minimum size strict dominating set).

4.2.1 Diamond DAG

THEOREM 7 (2S-Partitioning for Diamond DAG). *For a $n \times n$ diamond DAG, the minimal number of vertex sets for any valid 2S-partition $H(2S)$ is 1.*

Proof. The proof is obviously straightforward: a unique partition $V_1 = V - I$ fulfills $P3$ and $P4$ as $|\mathsf{In}(V - I)| = |\mathsf{Out}(V - I)| = 1$. Note that for the red-blue model the same very weak bound would hold: the unique partition $V_1 = V$ fulfills $P3$ and $P4$ as $I \in \mathsf{Dom}(V)$, $|I| = 1$, and $|\mathsf{Min}(V)| = 1$. □

This leads to a lower bound for the I/O complexity of 0 while, as the following theorem shows, the min-cut approach leads to an asymptotic lower bound of $\frac{n^2}{S}$.

THEOREM 8 (Min-cut based I/O bound for Diamond DAG). *For a $n \times n$ diamond DAG, the Minimum I/O cost, Q, satisfies*

$Q \geq \frac{(n-2S)^2}{S}$, *which gives an asymptotic bound of* $\frac{n^2}{S}$ *where* S *is the number of red pebbles.*

Proof. Consider an instance $\mathcal{P}$ of a red-blue-white pebble game for the $n \times n$ diamond-DAG that has the minimum I/O, Q. Consider a vertex x at location (i, j) (i.e., in ith row and jth column). Consider a (rotated) bow-tie-shaped sub-DAG V_x that consists of two dis-joint $m \times m$ squares, whose corners lie at $[(i-m, j), (i-1, j+m-1)]$ and $[(i, j-m), (i+m-1, j-1)]$ (refer fig. 8) . From Lemma 2, we have that Q_{V_x} the minimum I/O restricted to V_x is bounded by $2(w_{V_x}^{\max} - S)$ with $w_{V_x}^{\max}$ the max-min cardinality wavefront restricted to V_x. By considering the time-stamp just before x fires, $w_{V_x}^{\max}$ can itself be bounded by $2 \times m$: there are exactly $2 \times m$ disjoint paths in V_x that must be cut from ancestors $(i-m, j)$, $\ldots$, $(i-2, j)$, $(i-1, j), (i, j-1)$, $(i, j-2)$, $\ldots$, , $(i, j-m)$ to respectively descendants $(i, j+m-1)$, $\ldots$, $(i, j+1)$, $x, (i+1, j)$, $\ldots$, , $(i+m-1, j)$. Hence, $Q_{V_x} \geq 2(2m-S)$.

A rectangle of size $n \times 2m$ can be partitioned into $\lfloor (n-m)/m \rfloor$ bow-ties. A $n \times n$ diamond-DAG can itself be partitioned into $\lfloor n/2m \rfloor$ such rectangles, so on the overall into $\lfloor (n-m)/m \rfloor \times \lfloor n/2m \rfloor$. This gives, a lower bound for the I/O of $2 \times (2m - S) \times \lfloor (n-m)/m \rfloor \times \lfloor n/2m \rfloor$.

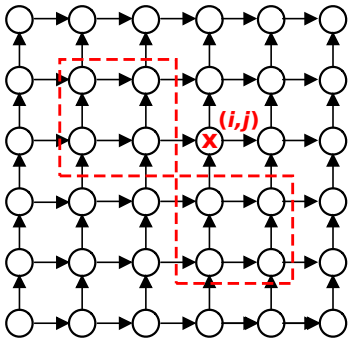


Figure 8: Diamond DAG with bow-tie shaped sub-DAG

If we let $m = S$, we get:

$$\begin{aligned} Q &\geq 2S \times \left\lfloor \frac{n-S}{S} \right\rfloor \times \left\lfloor \frac{n}{2S} \right\rfloor - |I| + |O| \\ &\geq 2S \times \frac{n-S-S+1}{S} \times \frac{n-2S+1}{2S} \\ &\geq \frac{(n-2S)^2}{S} \square \end{aligned}$$

This bound is tight by a factor of 2 as the following theorem states.

THEOREM 9 (I/O Upper-bound for Diamond DAG). *The Diamond DAG can be executed with an asymptotic I/O cost of* $2\frac{n^2}{S}$.

Proof. To prove the upper bound, we consider the following actual valid game made up of bands of width $m = S - 2$:

for $I = 1 : n$ **step** m
  for $j = 1 : n$
    for $i = I : \min(I+m-1, n)$
      if $(i = I$ and $i \neq 1)$: $R1(i-1, j)$
      $R3(i, j)$
      if $(i = I$ and $i \neq 1)$: $R4(i-1, j)$
      if $(j \neq 1) : R4(i, j-1)$
      if $(i = I+m-1$ and $i \neq n)$: $R2(i, j)$

This game uses exactly $m+2$ pebbles; for each of the upper $\lceil n/m \rceil - 1$ band of width m it performs n loads; for each of the lower $\lceil n/m \rceil - 1$ band of width m it performs n stores. This gives an overall cost of $2n\left(\lceil \frac{n}{S-2} \rceil - 1\right)$. $\square$

4.2.2 FFT

THEOREM 10 (2S-Partitioning for FFT). *For an FFT of size* n*, the minimal number of vertex sets for any valid* $2S$*-Partition is* $\left\lceil \frac{n\log(2n)}{4S\log(2S)} \right\rceil$.

Proof. Theorem 4.1 in [16] does not provide any constant factor, but the proof allows to easily derive one: the idea developed by Hung and Kung in their proof is simply to show that "*any vertex set of dominator set of size no more than* S *can have at most* $2S\log(S)$ *vertices*." This proves that h, the minimum number of vertex sets for any valid $2S$-Partition is less than $\left\lceil \frac{n\log(2n)}{4S\log(2S)} \right\rceil$. $\square$

This gives in both cases to an asymptotic lower bound for the I/O complexity of $\frac{n\log(n)}{4\log(S)}$ while, as proved below, the min-cut approach leads for the problem without recomputation to an asymptotic I/O complexity of $\frac{2n\log(n)}{\log(S)}$.

THEOREM 11. *For the* n*-point FFT graph, the minimum I/O cost,* Q*, satisfies* $Q \geq \frac{2n\log(n)}{\log(S)} \times (1 - \varepsilon_{n,S})$*, where* S *is the number of red pebbles, and* $\varepsilon_{n,S}$ *tends to* 0 *for large values of* n*,* S*, and* $\frac{n}{S}$.

Proof. Consider a DAG for an FFT of size m. Consider a pebble game instance $\mathcal{P}$ with minimum I/O and the time-stamp at which the first output vertex (i.e. vertex with no successors) o is fired by this schedule. Let $\mathcal{S}$ be the vertices already fired strictly before o, and $\mathcal{T}$ the others. As o is an output vertex, $\mathcal{S}$ contains all the m input vertices. By construction, $\mathcal{T}$ contains all the m output vertices. Hence, the corresponding wavefront, $|W_{\mathcal{P}}(o)| \geq m$.

Now, a DAG for the n-point FFT (of height $\log(2n)$) can be decomposed into disjoint sub-DAGs corresponding to m-points FFTs (and of height $\log(2m)$). This gives us $\lfloor n/m \rfloor \times \lfloor log(2n)/log(2m) \rfloor$ full sub-DAGs. From Lemma 2, the *IO* complexity of each sub-FFT is at least $2 \times (m - S)$. If we consider $m = S\log(S)$, by combining all the sub-FFTs we get a lower bound for $IO(\text{FFT}_n)$ of

$$\left\lfloor \frac{n}{S\log(S)} \right\rfloor \times \left\lfloor \frac{\log(2n)}{\log(2S\log(S))} \right\rfloor \times 2\left(S\log(S) - S\right) - 2n,$$

which tends to $\frac{2n\log(n)}{\log(S)}$ when n, S, and $\frac{n}{S}$ grow. $\square$

This bound is tight as a well-known tiled version of FFT consists of decomposing it into $\frac{\log(2n)}{\log(S)}$ stages of $\frac{n}{S}$ sub-FFTs of size S. For each sub-FFT the S inputs are loaded, the computation is spill-free, and the S output are stored, leading to an asymptotic I/O cost of $\frac{n\log(n)}{S\log(S)}$.

5. Automated Bounds for Arbitrary CDAGs

All approaches for I/O lower bounds reported in the literature so far have relied on specific properties about the structure of the analyzed computation. In this section, we develop an automated approach to deriving I/O lower bounds for any arbitrary CDAG, regular or irregular, without any restrictions on the structure or reliance on any properties of the CDAG. It uses a combination of two techniques: convex graph min-cut, as introduced in Section 4 for which a graph-based heuristic is detailed hereafter; and Hong & Kung 2S-Partitioning, recapped in Section 2 for which an ILP-based approximation heuristic is detailed below.

5.1 High-Level Meta-Heuristic

Our heuristic IOhierarchical combines both techniques using a recursive decomposition of the CDAG. The high-level principle is to (1) first run both heuristics on the full CDAG. The result of the min-cut based heuristic IOmincut is stored on Q_{mincut}; the result of the 2S-partitioning based heuristic IOmax2S is stored on Q_{2Spart}. (2) The CDAG is then bisected into C_1 and C_2 and IOhierarchical is run independently on each of the two sub CDAGs. The results are stored into Q_1 and Q_2, then summed up into $Q_{rec} = Q_1 + Q_2$. (3) The best of three $max(Q_{mincut}, Q_{2Spart}, Q_{rec})$ is returned.

The precise pseudo-code for IOhierarchical is given in Figure 9. The description of IOmincut is given in Section 5.2. The description of IOmax2S is given in Section 5.3.

Both the min-cut based heuristic and the 2S-Partitioning heuristics are given a timeout proportional to the size of the graph (the implementation details for this are not reported in the pseudo-codes). If no result is found within this timeout, the corresponding procedure returns -1, and the result is not used. Note that there are circumstances under which it is clear that it is not worth bisecting the current CDAG. A simple example is when IOmincut has not been interrupted by the timeout (so the returned result is an accurate value of $\max\left(0, 2(W_G^{\max} - S)\right)$) and it returns 0. In that case IOmax2S might be run on sub CDAGs but not IOmincut. This optimization that effects only the solving time but not the result is not detailed here. The goal of the bisection Bisect is to (almost) equally partition the DAG into two sub DAGs C_1 and C_2 such that an (almost) minimum number of edges between C_1 and C_2 are cut. Ideally this partitioning should be convex (there are edges from C_1 to C_2 but no edges from C_2 to C_1). But Theorem 2 allows to compose the results of non-convex partitioning and we thus can use standard graph bisection library that does not impose any convexity constraint.

```
IOhierarchical(C, V)
Inputs:
   C: CDAG
   V: vertex (sub-)set of C
Outputs:
   Q: lower bound of IO(C) restricted to V
Q ← 0
Q_mincut ← IOmincut (C, V)
Q ← max(Q, Q_mincut)
Q_2Spart ← IOmax2S (C, V)
Q ← max(Q, Q_2Spart)
if |V| > 2S then
      (V_1, V_2) ← Bisect (C, V)
      Q_1 ← IOhierarchical (C, V_1)
      Q_2 ← IOhierarchical (C, V_2)
      Q ← max(Q, Q_1 + Q_2)
end if
return Q
```

Figure 9: Recursive decomposition algorithm for computing I/O bound on an arbitrary CDAG C

5.2 Heuristic for Min-Cut Approximation

The min-cut based heuristic is a direct implementation of Lemma 2. As the requirement for this lemma is for the CDAG to have no input vertices, the first operation of IOmincut is to delete input vertices from V (the application of Corollary 1). It also removes output vertices. The number of such deleted vertices is then added to the final I/O bound result. For this modified graph, say $G' = (V', E')$ with no input/output vertices, the variable W that represents $W_{G'}^{max}$ is computed as the maximum value of ConvexMinVertexCut(G', x) over all $x \in V'$. Vertices of V' are actually enumerated in random order so that in the presence of a time-out any intermediate maximum value gives a correct lower-bound of $W_{G'}^{max}$. The corresponding algorithm is shown in Fig. 10.

As explained in Section 4, the computation of $W_{G'}^{min}(x)$ (computed through the call to ConvexMinVertexCut(x)) corresponds to: (1) a partitioning of V' into two sub-DAGs $\mathcal{S}_x$, $\mathcal{T}_x$ such that; (2) there is no edge from $\mathcal{T}_x$ to $\mathcal{S}_x$; (3) $\mathcal{S}_x$ contains x plus all the ancestors Anc(x) of x; (4) $\mathcal{T}_x$ contains all the descendants Desc(x) of x; (5) The out set of $\mathcal{S}_x$ (Out$(\mathcal{S}_x)$: set of vertices of $\mathcal{S}_x$ that have a successor in $\mathcal{T}$) is of minimum size.

This ConvexMinVertexCut, detailed in Fig. 11, is formulated as a Linear Programming (LP) problem that minimizes the number of cut vertices. Each vertex, $v \in V'$ is associated with two non-negative variables, c_v and t_v. Variable c_v captures the cut vertex count, and the variable t_v determines if vertex v belongs to $\mathcal{S}_x$ or $\mathcal{T}_x$, depending on whether $t_v = 0$ or 1, respectively. The convexity is enforced by the first con-

IOmincut(C,V)
Input:
  $C = (I, \mathcal{V}, E, O)$: CDAG
  V: a (sub)-set of $\mathcal{V}$
Outputs:
  Q: lower bound of $IO(C)$ restricted to V
$dI \leftarrow V \cap I$
$dO \leftarrow V \cap O$
$V' \leftarrow V - dI - dO$
$E' \leftarrow E \cap V' \times V'$
$W \leftarrow 0$
forall $x \in V'$ **do**
  $W_x \leftarrow$ ConvexMinVertexCut $((V', E'), x)$
  $W \leftarrow \max(W, W_x)$
end do
$Q' \leftarrow \max(0, 2 \times (W - S))$
$Q \leftarrow Q' + |dI| + |dO|$
return Q

Figure 10: Compute Minimum I/O cost for a CDAG using Min-Cut

straint that given an edge $(v,w) \in E'$, if v is in $\mathcal{T}_x$, then w also belongs to $\mathcal{T}_x$. The constraint, $c_v \geq t_w - t_v$, accounts for the cut vertex count whenever a vertex $v \in \mathcal{S}_x$ has its successor in $\mathcal{T}_x$. Finally, vertices $v = x \cup v' \in \mathsf{Anc}(x)$ $(resp.\ v \in \mathsf{Desc}(x))$ are restricted to the set $\mathcal{S}_x$ $(resp.\ \mathcal{T}_x)$ by setting $t_v = 0$ $(resp.\ 1)$. Since the constraint matrix of this LP formulation is totally-unimodular, this linear optimization problem always has an integral optimum. The minimization objective $(min \sum_{v \in V'} c_v)$ implicitly ensures that $c_v \leq 1$ and $t_v \leq 1$.

ConvexMinVertexCut(G', x)
Input:
  $G' = (V', E')$: DAG
  x: vertex about which min-cut is computed
Output:
  W_x: $W_{G'}^{min}(x)$
$W_x \leftarrow$ Optimal solution to the following LP:
  $\forall\ v \in V',\ t_v \geq 0,\ c_v \geq 0$
  minimize $\sum_{v \in V'} c_v$
  s.t. $\forall\ (v,w) \in E',\ t_w \geq t_v$
    $\forall\ (v,w) \in E',\ c_v \geq t_w - t_v$
    $t_x = 0$
    $\forall\ v \in \mathsf{Anc}(x),\ t_v = 0$
    $\forall\ v \in \mathsf{Desc}(x),\ t_v = 1$
return W_x

Figure 11: Compute Convex vertex-min-cut

5.3 Heuristic for 2*S*-Partitioning Approximation

We complement our min-cut based heuristic with another method to automatically derive I/O lower bound approximations. This second heuristic is based on an Integer Linear Program that operates on the CDAG, attempting to find an under-approximation of the minimal number of components in the optimal 2S-partition that can be built for the CDAG. The motivation for this second approach comes from the inherent limitation of the min-cut based approach for some CDAG shapes. The min-cut method offers tight estimation of the minimal I/O requirement for graphs where the amount of intermediate values required to complete the computation significantly exceeds the size of the input set (such as for the extreme of diamond-like CDAGs). However, this approach mostly ignores the input set fan-out, a critical factor for some other forms of CDAGs with high data reuse (i.e., matrix-multiply). We develop another heuristic aimed at effectively supporting that kind of (sub-)CDAG, which can be combined with the min-cut approach for more accurate I/O lower bound estimation of arbitrary (sub-)CDAGs.

Approximating the 2S-partitioning problem Optimal 2S-partitioning is in essence the building of a partition of the CDAG into convex sets (circuit-free), such that h, the total number of such sets is minimized. From this optimal partition, one can then derive I/O lower bound given the value of h [16]. A 2S-partition is built by enforcing properties on the set of vertices that are connecting a component V_i in the partition with the rest of the graph: $|\mathsf{In}(V_i)| \leq 2S$ and $|\mathsf{Out}(V_i)| \leq 2S$. There is no further constraint on V_i itself, that is, there is no constraint on how many memory operations are needed to actually execute V_i. This is in striking contrast with the min-cut approach which is geared towards estimating the I/O requirement to execute a sub-CDAG such as V_i, assuming all input data is already in fast memory. Nevertheless for high fan-out CDAGs where a single value is reused for multiple operations, it is also critical to carefully consider how many data movements may be needed based on how many data elements are input to sub-CDAGs. This is achieved using 2S-partitioning of the CDAG, where the focus is to capture input (and output) data movement requirements to execute a sub-CDAG.

To solve this problem, we compute an over-approximation *of the size of the largest component in the optimal 2S-partition* of the CDAG. Once this component is computed, its size (in terms of number of vertices) can be retrieved, and used to under-approximate the I/O requirement. By Lemma 1, $Q \geq S \times (h-1)$. So, taking the optimal 2S-partition of the CDAG, i such that $\forall j \neq i, |V_i| \geq |V_j|$, and $|U|$ an over-approximation of $|V_i|$ we have:

$$\left\lceil \frac{|V|}{|U|} \right\rceil \leq \left\lceil \frac{|V|}{|V_i|} \right\rceil \leq h \quad \Rightarrow \quad \left\lceil \frac{|V|}{|U|} \right\rceil \leq h \tag{5}$$

$$\Rightarrow \quad Q \geq S \times \left(\left\lceil \frac{|V|}{|U|} \right\rceil - 1 \right) \tag{6}$$

In otherwords a valid under-approximation of h is found by computing an over-approximation of $|V_i|$. This is achieved by searching for a single set of vertices $U \subseteq V$ whose size is

maximized, under constraints on the input set size $|\mathsf{In}(U)| \leq 2S$.

Integer Linear Programming formulation To model the problem, we resort to formulating an ILP that maximizes the size of U, using two Boolean variables per vertex in V. One variable p_v captures if the vertex v belongs to U or not, and the other i_v captures if the vertex belongs to $\mathsf{In}(U)$ or not. This maximization ($\max \sum_{v \in V} p_v$) is performed under the constraints given by the existence of an edge (i, j) in the CDAG, where if j is in U then i is either in U or in $\mathsf{In}(U)$. The last constraint, $\mathsf{In}(U) \leq 2S$, limits the size of the set of immediate predecessors to U. The ILP is expressed in the pseudo-code of IOmax2S given in Figure 12.

IOmax2S(C,V)
Inputs:
  C: CDAG
  V: vertex (sub-)set of C
Outputs:
  Q: lower bound of $IO(C)$ restricted to V
$dI \leftarrow \{v \in V \mid \nexists\ (u,v) \in E\} - I$
$U \leftarrow$ Optimal solution to the following ILP:
  $\forall\ v \in V, \quad p_v \in \{0,1\}, \quad i_v \in \{0,1\}$
  maximize $\sum_{v \in V}\ p_v$
  s.t. $\forall\ (v,w) \in E,\ p_v + i_v \geq p_w$
    $\forall\ v \in I \cup dI,\ p_v = 0$
    $\sum_{v \in V} i_v \leq 2S$
$Q \leftarrow \left(\left\lceil \frac{|V|}{|U|} \right\rceil - 1\right) - dI$
return Q

Figure 12: Compute Minimum I/O cost of a CDAG using 2S-Partitioning

Solving this ILP maximizes the number of vertices in U. As we ignore the constraints on the size of $\mathsf{Out}(U)$ (which can be added easily in a similar way than for $\mathsf{In}(U)$) and on the schedulability of the partition (therefore allowing for U to be a non-convex set), this may lead to over-approximating the size of U with respect to V_i, the largest component in the optimal 2S-partition. One may note that necessarily $|U| \geq |V_i|$, as $|U|$ is maximized under looser constraints than the ones imposed on V_i.

One can remark in IOmax2S the tagging of the dI set as input vertices. This is done because, even if Lemma 1 holds for the Flexible IO vertex model given by Definition 6, a 2S-Partitioning on a CDAG with few or even no input/output vertices would lead to a very weak lower bound. For this reason, prior to computing the largest 2S-partition of C, we tag predecessor-free vertices as inputs. Note that in the case we would express the constraint on the size of $\mathsf{Out}(U)$, we would tag successor-free vertices as outputs as well. We can then apply Theorem 5, which leads to substracting $|dI|$ to the IO cost found on this modified CDAG.

5.4 Discussion

The scalability of the automated tool is an important issue and being able to address large problems is currently a challenge. The ConvexMinVertexCut pseudo-code, as reported here, has an average-case complexity of $O(|V|^3)$, leading to an overall complexity for IOmincut of $O(|V|^4)$. The IOmax2S shows an even larger complexity. In other words, the largest sub CDAG that can be treated within time credit of T will have at most $\Theta\left(\sqrt[4]{T}\right)$ vertices. In other words, given a CDAG of size $|V| = n$, a time credit of T (assuming a scalable heuristic is used to perform bisection), the CDAG will have to be decomposed in at least $\sqrt[3]{\frac{T}{n^4}}$ equal size parts. Further bisections, that may improve the quality of the bound, will not impact the overall complexity.

Several techniques could be used to lower the overall complexity. First, as W_G^{max} computed by IOmincut is independent of the pebble count, S, the IOmincut heuristic can be run just once for a given problem size, collecting the W_G^{max} of each (sub-)CDAG in a file and finally post-processing the output at a cheaper computational cost to obtain the lower bounds corresponding to any value of S. Also, the computation of W_G^{max} could be lowered to $O(n^3)$ by taking advantage of an incremental update of $W_G^{min}(x)$ (using a direct graph based implementation). It can also be lowered by taking the maximum value of $W_G^{min}(x)$ on a randomly generated set of vertices x. Further, IOmincut can be parallelized by executing ConvexMinVertexCut for different vertices in parallel.

The solution of the ILP can also be tuned to address its inherent prohibitive complexity. For instance, we can guide the ILP by adding extra constraints on the minimal and maximal value of $|U|$, which can be computed by relaxing the ILP to a Quadratic Program. While running IOmax2S with a timeout factor T, after a solver timeout, we can rerun the problem by setting the lower cutoff tolerance to be the integer solution, say $|P|$, found during the previous run. This allows the solver to discard some of the sub-optimal solution branches, making it run faster during the rerun. In our experience, P found after the first timeout often turned out to be the optimal one. Hence, by constraining the solution to be in $\lfloor 1.1 \times |P| \rfloor \leq \sum p_v \leq \lceil (1.1 + \varepsilon)|P| \rceil$ during the rerun, if the solver finds that the search space is empty, we can safely conclude that the optimal $|U|$ can be bounded by $1.1 \times |P|$.

The last technique, perhaps the most challenging and promising one, is to take advantage of regularities that could be detected [19] on the graph and reason on an abstract representation of it. For example, we could adapt and incorporate the recent automatic approach developed by Christ et al. [11] for characterizing lower bounds for a subclass of affine loop programs. As most regular CDAGs can be decomposed into isomorphic sub-CDAGs, the tool can be run on a single sub-CDAG or on the problem with smaller input size (which corresponds to a sub-CDAG of the same problem of larger size) to compute the lower bounds for the larger problem size by application of the decomposition theorem (detailed in Sec. 3.3).

6. Experimental Results

In this section, we present experimental results using the automated approach described in the previous section.

6.1 Diamond DAG

As shown in Theorem 8, the Minimum I/O cost, Q, for an $n \times n$ Diamond DAG satisfies $Q \geq \frac{(n-2S)^2}{S} \approx \frac{n^2}{S}$

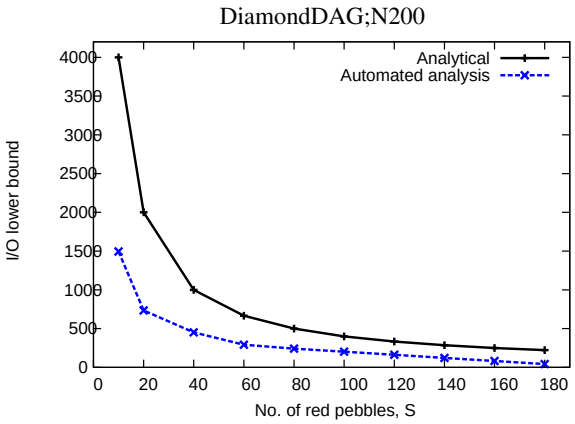


Figure 13: I/O lower bound for Diamond DAG: Analytical versus automated CDAG analysis

Fig. 13 shows the lower bound generated by our automated CDAG analysis for different values of S, for a Diamond DAG of size 200×200. The I/O lower bound from the above analytical expression is also plotted. The bound generated by the automated CDAG analysis is less tight than the lower bound obtained using the analytical reasoning developed in this paper, but shows the same trend as a function of S. The reason why the automated solution does not meet the optimal analytical one comes from the recursive decomposition of the full CDAG that tends to create square shaped sub DAGs instead of the optimal bow-ties used in the proof of Theorem 8.

6.2 Diamond-Chain

Fig. 14 shows a CDAG analyzed by Bilardi et al. [7]. It is a diamond DAG with m^2 nodes composed with a linear chain graph of m^2 nodes, with a dependence edge from each diamond node to a distinct chain node. The dependence edges are in reverse order – the last diamond vertex connects to the first chain vertex, the first diamond vertex connects to the last chain vertex, etc.

Fig. 15 shows the lower bound generated by automated CDAG analysis for different values of S for a diamond-chain of side 200.

It may be seen that the generated lower bound is much higher than that of a diamond DAG of the same size, shown in Fig. 13. This is because the values generated early in the Diamond DAG are live till late in the execution of the chain portion of the CDAG. By analyzing the structure of this CDAG, it can be reasoned that the convex graph min-cut corresponding to the terminal vertex of the diamond in the diamond-chain

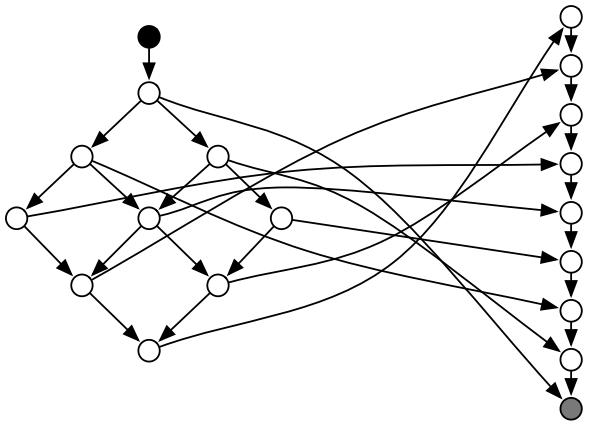


Figure 14: Diamond-Chain CDAG [7]

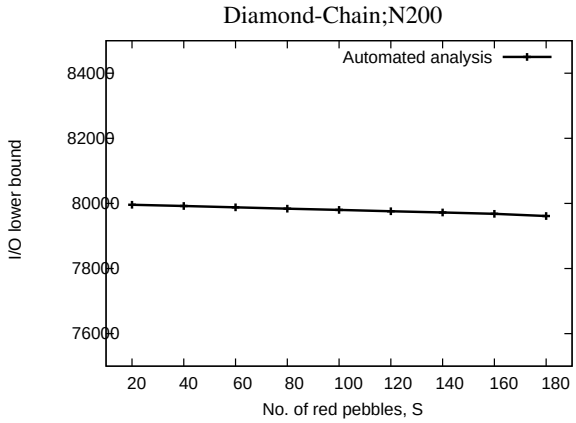


Figure 15: I/O lower bound for Diamond-Chain from automated CDAG analysis

will have a cut size of $m^2 - S$ vertices. It can be seen from the automated analysis as a function of S that the complexity is indeed independent on the value of S and that the complexity of the composed CDAG is evaluated accurately.

6.3 Matrix Multiplication

Fig. 16 shows the lower bounds generated by automated CDAG analysis for a 30×30 Matrix multiplication.

The analytical results are from the work of Irony et al. [18]. In this case, we see that the bounds from CDAG analysis, in this case from the ILP based 2S-partitioning bounds, are actually higher than the analytical bounds. This is because the analytical bounds are asymptotic parametric expressions that do not include lower order terms that can be significant for small values of N and S.

6.4 Case Study: Sorting

Since the automated CDAG analysis can be applied to arbitrary CDAGs, we applied it to CDAGs from different comparison-based sorting algorithms. In this section we present results for two sorting algorithms: quicksort and odd-even

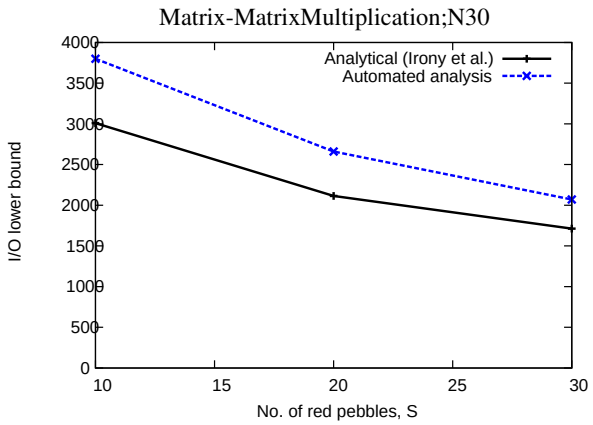


Figure 16: I/O lower bound for Matrix Multiplication: Analytical versus generated by automated CDAG analysis

sort. Quicksort has an average computational complexity of $O(n \log_2 n)$ to sort n elements, while odd-even sort is has a an asymptotically higher complexity of $O(n^2)$, but a lower coefficient for quadratic complexity term. So efficient sorting routines often use a combination of an $O(n \log_2 n)$ algorithm like quicksort along with a routine like odd-even sort, and a threshold value of n (usually set to a value between 16 and 20) for switching between the two algorithms. We were interested in comparing the I/O complexities of the two sorting algorithms. Fig. 17 compares the I/O complexity of quicksort an odd-even sort as a function of S, normalized to the number of comparisons performed. On the same graph, we also see the normalized number of data movements required in executing the sorting algorithms – by playing the red-blue-white pebble game using the code's sequential execution order for firing of CDAG vertices (therefore providing an upper bound on the I/O complexity). It may be seen that while odd-even sort has a higher normalized data access cost than quicksort for the code's sequential execution order, the lower bounds from automated CDAG analysis reveal a significantly lower bound for odd-even sort than quicksort.

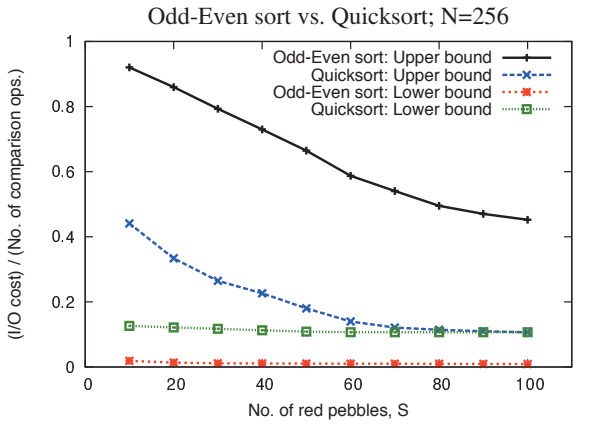


Figure 17: Quick sort versus odd-even sort: Normalized I/O lower bound and cost for sequential execution order

Upon examining the CDAG for odd-even sort more carefully, it became apparent that it was amenable to skewing followed by loop tiling to significantly enhance data locality. In contrast, for the recursive quicksort algorithm, we could not identify any opportunity for loop transformations that would improve data locality. We implemented a register-tiled version of odd-even sort and evaluated performance for different register-tile sizes. Fig. 18 displays relative time for odd-even sort over quicksort as a function of array size (using Intel ICC, for int data type), for the standard version as well as the modified register-tiled version. For the original version, the cross-over was around 16, i.e., odd-even sort was more efficient for sorting fewer than 16 elements, while quick sort was more efficient for sorting more than 16 elements. But with the tiled version of odd-even sort, the cross-over point was much higher - around 110 elements. The experiment was run multiple times with different random inputs and average time was taken. Boundary cases like fully sorted and reverse sorted inputs were also tested and we found similar trends.

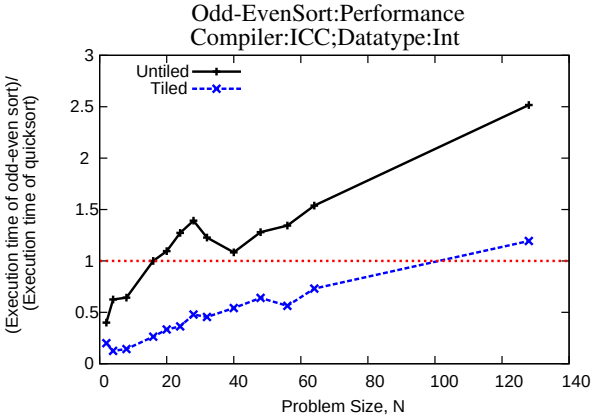


Figure 18: Execution time for original and tiled odd-even sort relative to quick sort

7. Related Work

Hong & Kung provided the first characterization of the I/O complexity problem using the red/blue pebble game and the equivalence to 2S-partitioning of CDAGs [16]. Their 2S-partitioning approach uses dominators of incoming edges to partitions but does not account for the internal structure of partitions. In this paper, we develop an alternate lower bound approach that models the internal structure of CDAGs, and use convex mincut-based partitions enabling tighter analytical bounds for some algorithms. In addition, Hong & Kung's original model does not lend itself easily to development of lower bounds for a CDAG from bounds for component subgraphs.

Several works followed Hong & Kung's work on I/O complexity in deriving lower bounds on data accesses [1–5, 7, 11, 13, 18, 22–26, 28, 29]. Aggarwal et al. provided several lower bounds for sorting algorithms [1]. Savage [24, 25] developed

the notion of S-span to derive Hong-Kung style lower bounds and that model has been used in several works [22, 23, 26]. Irony et al. [18] provided a new proof of the Hong-Kung result on I/O complexity of matrix multiplication and developed lower bounds on communication for sequential and parallel matrix multiplication. More recently, Demmel et al. have developed lower bounds as well as optimal algorithms for several linear algebra computations including QR and LU decomposition and all-pairs shortest paths problem [3, 4, 13, 28]. Bilardi et al. [5, 7] develop the notion of access complexity and relate it to space complexity. Bilardi and Preparata [6] developed the notion of the closed-dichotomy size of a DAG G that is used to provide a lower bound on the data access complexity in those cases where recomputation is not allowed. Our notion of schedule wavefronts is similar to the closed-dichotomy size in their work; but, unlike the work of [6], we use it do develop an effective automated heuristic to compute lower bounds for CDAGs. Extending the scope of the Hong & Kung model to more complex memory hierarchies has also been the subject of research. Savage provided an extension together with results for some classes of computations that were considered by Hong & Kung, providing optimal lower bounds for I/O with memory hierarchies [24]. Valiant proposed a hierarchical computational model [29] that offers the possibility to reason in an arbitrarily complex parameterized memory hierarchy model. While we use a single-level memory model in this paper, the work can be extended in a straight forward manner to model multi-level memory hierarchies.

Unlike Hong & Kung's original model, several models have been proposed that do not allow recomputation of values (also referred to as "no repebbling") [3–5, 9, 12, 18, 21–27]. Savage [24] develops results for FFT using no repebbling. Bilardi and Peserico [5] explore the possibility of coding a given algorithm so that it is efficiently portable across machines with different hierarchical memory systems. without the use of recomputation. Ballard et al. [3, 4] assume no recomputation in deriving lower bounds for linear algebra computations. Ranjan et al. [22] develop better bounds than Hong & Kung for FFT using a specialized technique adapted for FFT-style computations on memory hierarchies. Ranjan et al. [23] derive lower bounds for pebbling r-pyramids under the assumption that there is no recomputation.

The use of Hong & Kung's model has required algorithm-specific reasoning to find S-partitions, even for regular graphs. Savage's [24] S-span model also requires problem-specific insights. We note that Hong & Kung's model and Savage's model are not strictly comparable, as noted by Bilardi et al. [7]. The recent works of Ranjan et al. [21–23] develop a technique inspired by FFT-style computations, r-pyramids and binomial graphs. In addition to these works, other approaches [1, 4, 5, 13, 29] also require problem-specific insights to develop bounds. In contrast, we have developed an approach that can be used to develop I/O lower bounds for an arbitrary CDAG without any problem-specific insights. We note here that very recent work from U.C. Berkeley [11] has developed a very novel approach to developing parametric I/O lower bounds that does not require problem-specific insights. The approach is applicable/effective for a class of nested loop computations but is either inapplicable or produces weak lower bounds for other computations (e.g., stencil computations, FFT, etc.).

8. Open Questions

There are a number of open questions raised by the developments in this paper, that we hope may be addressed in follow-up work.

Allowing vs. prohibiting recomputation The majority of results in this paper (with the exception of Theorem 2) only apply under the RBW pebble game model that prohibits recomputation of vertices in a CDAG. In contrast, Hong & Kung's original RB pebble game model permits recomputation of vertices multiple times and thereby possibly avoid red-to-blue and blue-to-red pebble moves. Since any valid RBW game can be directly translated to a valid RB game, the inherent I/O complexity of any CDAG under the RB model is lower than or equal to that under the RBW model. Therefore any I/O lower bound under the RB model is a valid lower bound under the RBW model (albeit possibly a weak bound) but not vice-versa. In some specific cases, such as the Diamond DAG, it is possible to prove that an I/O lower bound derived under the RBW model is also a valid lower bound under the RB model. As is clear from the developments in Sec.3, more powerful analysis techniques are feasible under the stricter RBW model than under the less restrictive RBW model. Further, for specific CDAGs, the bounds under the RBW model are also valid under the RB model.

This raises a very interesting question: *Are there a class of algorithms for which the inherent I/O complexity under both the RB and RBW models is the same, i.e., are there some algorithms where allowing recomputation cannot possibly help reduce data access costs? Is it possible to develop sufficient conditions on the structure of a CDAG that guarantee equivalence of I/O complexity under both models?*

Parametric Function Fitting An interesting possibility is to seek parametric functions within a class that effectively model the I/O complexity of an algorithm, by performing regression based fit from a number of concrete values generated by running the automated CDAG analysis for different problem sizes and different values of S. We did carry out such an exercise for the diamond DAG, attempting to perform linear regression on log-log plots of the number of pebble moves as a function of S, with N fixed, and vice versa. For a set of experiments varying S ranging from 4 to 256 and N ranging from 50 to 800. The regression fit yielded slopes of -0.9, -1, -0.9, -1, -0.9, -1.1, -1.1, for $log(Q)$ as a function of $log(S)$ for various fixed values of N, and slopes of 2, 2, 1.7, 1.6, 1.7, 1.7, 1.9, respectively for $log(Q)$ as a function of $log(N)$, for the seven different values of S. The manually derived lower bound expression

for the Diamond DAG (Sec. 4.2) had exponents of -1, and 2, respectively, for S, and N. Thus the experiment is suggestive that use of linear regression on log-log plots may be useful in characterizing parametric trends of the I/O lower-bound of an algorithm as a function of problem size and number of red pebbles.

Tightness of Lower Bounds An important question is whether a generated lower bound is tight. The primary means of assessing tightness of lower bounds is by forming upper bounds using concrete algorithm implementations. But is any automated estimation feasible? In complementary work [15], we have developed an automated CDAG analysis approach that generates I/O upper bounds for the RBW pebble game model. By generating both I/O upper bounds and lower bounds, the tightness of the bounds can be gauged from the separation between them.

9. Conclusion

Characterizing the I/O complexity of a program is a cornerstone problem, that is particularly important with current and emerging power-constrained architectures where the data transfer cost will be the dominant energy bottleneck. Previous approaches to modeling the I/O complexity of computations have several limitations that we address through the developments in this paper. First, by suitably modifying the pebble game model used for characterizing IO complexity, we enable analysis of large composite computational DAGs by decomposition into smaller sub-DAGs. Second, by developing an alternate lower bounding technique to Hong & Kung 2S-partitioning, using convex min-cut graph partitioning, we enable tighter parametric lower bounds for some algorithms. Finally, by developing an automated lower bound analysis approach, we enable the characterization of IO complexity of arbitrary, irregular CDAGs.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *19th STOC*, pages 305–314, 1987.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. ACM*, 59(6):32, 2012.

[5] G. Bilardi and E. Peserico. A characterization of temporal locality and its portability across memory hierarchies. *Automata, Languages and Programming*, pages 128–139, 2001.

[6] G. Bilardi and F. P. Preparata. Processor - Time Tradeoffs under Bounded-Speed Message Propagation: Part II, Lower Bounds. *Theory Comput. Syst.*, 32(5):531–559, 1999.

[7] G. Bilardi, A. Pietracaprina, and P. D'Alberto. On the space and access complexity of computation dags. In *Graph-Theoretic Concepts in Computer Science*, volume 1928 of *LNCS*, pages 81–92. 2000.

[8] G. Bilardi, K. Ekanadham, and P. Pattnaik. Efficient stack distance computation for a class of priority replacement policies. *International Journal of Parallel Programming*, pages 1–39, 2012.

[9] G. Bilardi, M. Scquizzato, and F. Silvestri. A lower bound technique for communication on bsp with application to the fft. In *Euro-Par*, pages 676–687, 2012.

[10] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proc. ICS '03*, pages 150–159, 2003.

[11] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays Part 1. EECS Technical Report EECS–2013-61, UC Berkeley, May 2013.

[12] S. A. Cook. An observation on time-storage trade off. *J. Comput. Syst. Sci.*, 9(3):308–316, 1974.

[13] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Scientific Computing*, 34(1), 2012.

[14] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257, 2003.

[15] N. Fauzia, V. Elango, M. Ravishankar, J. Ramanujam, F. Rastello, A. Rountev, L.-N. Pouchet, and P. Sadayappan. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. CSE Technical Report OSU-CISRC-9/13-TR19, The Ohio State University, September 2013.

[16] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th annual ACM sympo. on Theory of computing (STOC'81)*, pages 326–333. ACM, 1981.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[18] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[19] A. Ketterlin and P. Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *MICRO*, pages 437–448, 2012.

[20] R. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, 1970.

[21] D. Ranjan and M. Zubair. Vertex isoperimetric parameter of a computation graph. *Int. J. Found. Comput. Sci.*, 23(4):941–, 2012.

[22] D. Ranjan, J. Savage, and M. Zubair. Strong I/O lower bounds for binomial and FFT computation graphs. In *Computing and Combinatorics*, volume 6842 of *LNCS*, pages 134–145. Springer, 2011.

[23] D. Ranjan, J. E. Savage, and M. Zubair. Upper and lower I/O bounds for pebbling r-pyramids. *J. Discrete Algorithms*, 14: 2–12, 2012.

[24] J. Savage. Extending the Hong-Kung model to memory hierarchies. In *Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281. 1995.

[25] J. E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998.

[26] J. E. Savage and M. Zubair. Cache-optimal algorithms for option pricing. *ACM Trans. Math. Softw.*, 37(1), 2010.

[27] M. Scquizzato and F. Silvestri. Communication lower bounds for distributed-memory computations. *CoRR*, abs/1307.1805, 2013.

[28] E. Solomonik, A. Buluç, and J. Demmel. Minimizing communication in all-pairs shortest paths. In *IPDPS*, 2013.

[29] L. G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77:154–166, Jan. 2011.

[30] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991.