Strong I/O Lower Bounds for Binomial and FFT Computation Graphs

Desh Ranjan[1], John Savage[2], and Mohammad Zubair[1]

[1] Old Dominion University, Norfolk, Virginia 23529
[2] Brown University,Providence, Rhode Island 02912

Abstract. Processors on most of the modern computing devices have several levels of memory hierarchy. To obtain good performance on these processors it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy. In this paper, we propose a new technique, *the boundary flow technique*, for deriving lower bounds on the memory traffic complexity of problems in multi-level memory hierarchy architectures. The boundary flow technique relies on identifying sub-computation structure corresponding to equal computations with a minimum number of *boundary* vertices, which in turn is related to the vertex isoperimetric parameter of a computation graph. We demonstrate that this technique results in stronger lower bounds for memory traffic on memory hierarchy architectures for well-known computation structures: the binomial computation graphs and FFT computation graphs. For binomial computation we improve the lower bound by a factor of three. This reduces the gap between the lower and upper bound from a factor of 4 to a factor of 4/3. For FFT computation, past work has mostly focused on asymptotic lower bounds. We improve the best known previous lower bound for FFT computation by a factor of 8. The lower bound established is almost optimal as there exists a simple FFT algorithm that nearly achieves this bound.

1 Introduction

Modern processors have several levels of memory hierarchy. To obtain good performance on these processors it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy [9, 12]. The memory traffic that is required between different levels of memory depends on the application, memory hierarchy architecture, and the effectiveness of the blocking algorithm. To evaluate the effectiveness of the blocking algorithm for a given application, it is of interest to know what is the lower bound on the memory traffic between different levels of memory. We refer to this lower bound as the memory traffic complexity of the application. A formal definition of memory traffic complexity is given later in the paper.

A number of important straight line computations such as matrix multiplication, FFT, and several financial computations are modeled using DAGs[17]. A DAG captures the data dependency at various stages of the computation and helps in analyzing performance on modern computer architectures, which is determined to a large extent by the memory traffic[9, 19]. Given a DAG, the computation for DAG can be carried out in many different ways, essentially determined by the order in which computations corresponding to various vertices on the DAG are done. The fast memory in the memory hierarchy has limited capacity and for many large computations is not big enough to hold intermediate results during the computation. This forces the architecture to use slower memory for storing intermediate results. Different orderings results in different memory traffic to the slower memory [9]. The key in developing a high performance algorithm for a DAG is to identify the order that results in minimum memory traffic to the slower memory. To evaluate the effectiveness of different orderings and also to gain insight into the structure of the DAG and its relationship to the memory traffic, it is desirable to find minimum possible memory traffic to slower memory for any ordering.

Hong and Kung [10] used the red-blue pebble game to derive memory complexity on a two level memory hierarchy for matrix multiplication and FFT. Savage [16, 17] introduced the S-span of a DAG and generalized the Hong-Kung lower-bound method [10] to multiple levels of memory hierarchy. The S-span intuitively represents the maximum amount of computation that can be done after loading data in a cache at some level without accessing higher level memories. We propose a new technique, *the boundary flow technique*, for deriving lower bounds on the memory traffic complexity of computations that can be represented as DAGs in multi-level memory hierarchy architectures. The boundary flow technique relies on identifying sub-computation structure corresponding to equal computations with a minimum number of *boundary* vertices. The notion of finding minimum number of boundary vertices for a fixed size computation structure is related to determining the vertex isoperimetric parameter of a graph [11, 15]. The VIP is related to separators in graphs. The separators have been used to establish lower bounds, for example[3, 21, 20, 5, 7].

For simplicity, in this paper we will only consider two levels of memory hierarchy. The results for two levels can be extended to multiple levels of memory hierarchy using the multiple-level memory hierarchy model outlined in [16]. (See also [17, Chapter 11].) We demonstrate that this technique results in stronger lower bounds for memory traffic on memory hierarchy architectures for two computations with binomial and FFT as underlying DAGs.

The lower bound bound for memory traffic for both binomial and FFT computations have been addressed in the literature [1, 10, 17]. For binomial computation we improve the lower bound by a factor of three. This reduces the gap between the lower and upper bound significantly. We now have an upper bound that results in memory traffic which is 4/3 times the lower bound established in this paper. For FFT computation, past work has mostly focused on asymptotic lower bounds [1, 10, 17]. We improve bounds for FFT computations by a factor of 8 compared to the FFT bound established in [17]. In fact, this bound is almost optimal as there exists a simple FFT algorithm that achieves this bound. Strengthening the lower bound by a constant factor, besides being of theoretical interest, is important for practical reasons. Deriving these strong bounds gives insight into deriving better algorithms, which are a factor of four to eight times better than the existing algorithms. These factors may look small but are significant in terms of cost saving for applications with real time constraints, such as financial applications.

The rest of the paper is organized as follows. The required definitions and the memory hierarchy model that is used in developing memory complexity is discussed in Section 2. In Section 3 we define the S-span of a graph and cite previous lower bounds on memory complexity derived using it. In Section 4 our new boundary flow technique for deriving improved lower bounds is presented and applied to the r-pyramid and FFT graphs. Finally, in Section 5 we present our summary and conclusions.

2 Background

We first give formal definitions of computation graph, computation structure, and memory traffic complexity. Next we briefly describe the red-blue pebble game for deriving lower-bounds.

2.1 Computation Graphs, Structures, and Memory Traffic Complexity

A *computation graph* is a directed acyclic graph $G = (V, E)$. The vertices of G with in-degree zero are called the *input vertices* and the vertices with out-degree zero are called the *output vertices*. The goal is to compute the values at the output vertices given the values at the input vertices. The value at a vertex can be computed if and only if the value at all its predecessor vertices have been computed and are available. We say that the computation on G is complete if the values at all its output vertices have been computed. A *computation structure* is a parametric description of computation graphs. Formally, a computation structure is a function $\tilde{G} : \mathbb{N}^k \rightarrow \{G \,|\, G \text{ is a computation graph }\}$, where k is the number of parameters used to describe G.

Given a computation graph G, the computation on G can be carried out in many different ways. A *computation scheme* for a computation structure $\tilde{G}$ is an algorithm that completely specifies how to carry out the computation for each $\tilde{G}(t)$ where $t \in \mathbb{N}^k$.

An input in a two level memory hierarchy refers to a read from secondary memory, and an output refers to a write to the secondary memory. The I/O associated with a computation on a graph G is the total number of input and output operations used during the computation. We now define the memory traffic complexity for a single processor with 2-levels of memory hierarchy with $\hat{\sigma} = \langle \sigma_0, \sigma_1 \rangle$ where σ_0 is the primary memory size, and σ_1 is the secondary memory size. Let $\tilde{G} : \mathbb{N}^k \rightarrow \{G \,|\, G \text{ is a computation graph}\}$ be a computation structure. Let $T_1(\hat{\sigma}, \tilde{G})(t)$ be the minimum I/O required by any computation scheme for $\tilde{G}$ on input $\tilde{G}(t)$ where $t \in \mathbb{N}^k$. The function $T_1(\hat{\sigma}, \tilde{G}) : \mathbb{N}^k \rightarrow \mathbb{N}$

as defined above is called the *memory traffic complexity* of $\tilde{G}$. A computation scheme that matches the memory traffic complexity for $\tilde{G}$ is called a *memory traffic optimal scheme* for $\tilde{G}$.

Binomial and FFT Computation Graphs
Binomial option valuation is a popular approach that values an option contract using a discrete time model [13, 6]. The binomial option pricing computation is modelled by a directed acyclic pyramid graph $G_b(n)$ with depth n and $n+1$ leaves as shown in Figure 1. FFT computation graph occurs in many scientific and financial computations[2, 4]. The n-point FFT computation is modelled by a directed acyclic graph $G_f(n)$ with $n(\log n + 1)$ vertices as shown in Figure 2. Note that in $G_f(n)$, there are n input vertices with zero in-degree at level-1, and n output vertices with zero out-degree at level-$(n+1)$.

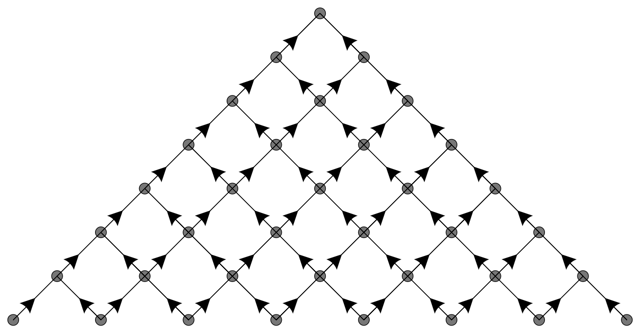

Fig. 1. The binomial graph $G_b(n)$ with depth n and $n+1=8$ leaves.

2.2 The Reb-Blue Pebble Game

The red-blue pebble game models data movement between adjacent levels of a two level memory hierarchy. In the red-blue game, red pebbles identify values held in a fast primary memory whereas blue pebbles identify values held in a secondary memory. An input refers to a read from the secondary memory, and an output refers to a write to a secondary memory. Since the red-blue pebble game is used to study the number of I/O operations necessary for a problem, the number of red pebbles is assumed limited and the number of blue pebbles is assumed unlimited. Before the game starts, blue pebbles reside on all input vertices. The goal is to place a blue pebble on each output vertex, that is, to compute the values associated with these vertices and place them in long-term storage. These assumptions capture the idea that data resides initially in the most remote memory unit and the results must be deposited there.

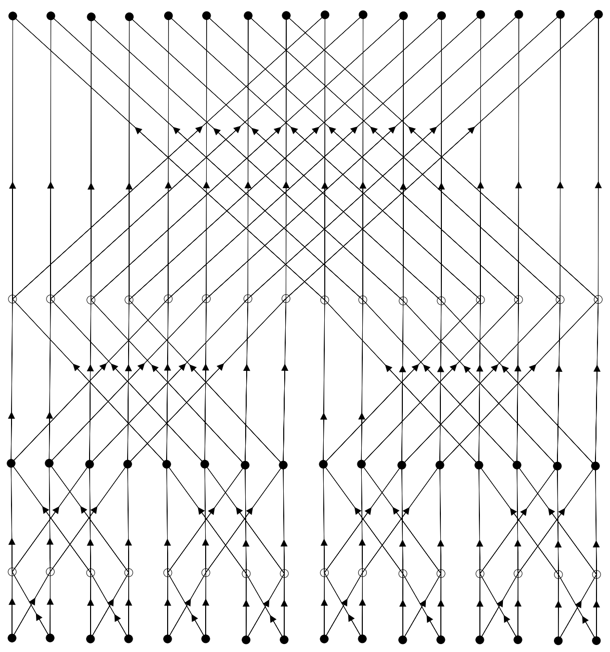

Fig. 2. The FFT graph $G_f(2^s)$ for $s = 2^2$.

<u>*Red-Blue Pebble Game Rules*</u>

- (Initialization) A blue pebble can be placed on an input vertex at any time.
- (Computation Step) A red pebble can be placed on (or moved to) a vertex if all its immediate predecessors carry red pebbles.
- (Pebble Deletion) A pebble can be deleted from any vertex at any time.
- (Goal) A blue pebble must reside on each output vertex at the end of the game.
- (Input from Blue Level) A red pebble can be placed on any vertex carrying a blue pebble.
- (Output to Blue Level) A blue pebble can be placed on any vertex carrying a red pebble.

A pebbling strategy $\mathcal{P}$ is the execution of the rules of the pebble game on the vertices of a computation graph. We assign a step to each placement of a pebble, ignoring steps on which pebbles are removed. The I/O time of $\mathcal{P}$ on the graph G is the number of input and output (I/O) steps used by $\mathcal{P}$.

3 The *S*-Span Approach to Deriving Lower Bounds

The S-span [16, 17] is a measure that intuitively represents the maximum amount of computation that can be done after loading data in a cache at some level without accessing higher level memories (those further away from the CPU).

Definition 1. *The S-span of a DAG G, $\rho(S, G)$, is the maximum number of vertices of G that can be pebbled starting with any initial placement of S red pebbles and using no blue pebbles.*

The S-span is a measure of how many vertices can be pebbled without doing any I/O. S pebbles are placed on the most fortuitous vertices of a graph and the maximum number of vertices that can be pebbled without doing I/O is the value of the S-span. Clearly, the measure is most useful for graphs that have a fairly regular structure. We now state lower bound results for binomial and FFT computation graphs which have been derived based on the S-span approach.

Theorem 1 ([18]). *The memory traffic complexity of $G_b(n)$ on a 2-level memory hierarchy system satisfies*

$$T_1(\hat{\sigma}, G_b)(n) \geq \frac{n(n+1)}{4\sigma_0 + 2}.$$

The following result is from on the S-span for an FFT computation graph.

Theorem 2 ([17]). *The memory traffic complexity of $G_f(n)$ on a 2-level memory hierarchy system satisfies*

$$T_1(\hat{\sigma}, G_f)(n) \geq \frac{n \log n}{4(\log \sigma_0 + 1)}.$$

4 Boundary flow technique for deriving lower bounds

An important class of pebbling strategies is the non re-pebbling strategies. Intuitively, these schemes never repeat a *computation step*. More formally, these pebbling strategies never pebble a node twice using a computation step.

Definition 2. *A non re-pebbling strategy is a pebbling strategy that never pebbles any vertex more than once using a computation step.*

The boundary flow technique currently works for non-repebbling strategies. It works by subdividing the pebbling into sub-pebblings and then deriving an I/O lower bound for each of the sub-pebblings. The overall I/O lower bound for the pebbling is obtained by summing the individual lower bounds. The individual lower bounds for the sub-pebblings are related to the notion of boundary of a subset of vertices in a computation graph and the fact that the number of red pebbles is limited.

Definition 3. *Let $G = (V, E)$ be a directed graph and $S \subset V$. Then*

$$\begin{aligned} out(S) &= \{u \in S \mid v \in \bar{S} \text{ and } (u, v) \in E\} \\ in(S) &= \{u \in \bar{S} \mid v \in S \text{ and } (u, v) \in E\} \\ boundary(S) &= in(S) \cup out(S). \end{aligned}$$

Note that $in(S) \cap out(S) = \emptyset$ and consequently $|boundary(S)| = |in(S)| + |out(S)|$.

The following lemma relates the boundary to the minimum memory traffic required for pebbling a computation graph.

Lemma 1. *Let $G = (V, E)$ be any computation graph and let P be any pebbling of G. Consider subdivision of the pebbling P into consecutive sequential sub-pebblings $P_1, P_2, \ldots, P_h$. Let V_j be the set of vertices that are newly pebbled (i.e. red pebbled using the computation rule) in sub-pebbling P_j. Then the number of I/Os used in pebbling P to pebble G is at least*

$$\sum_{j=1}^{h}(|boundary(V_j)| - 2\sigma_0).$$

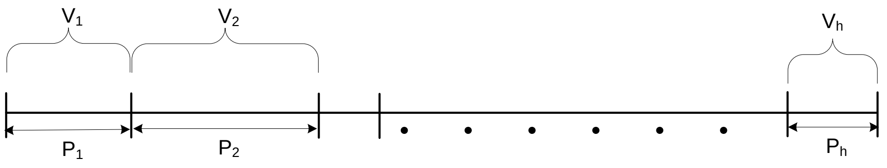

Fig. 3. Subdivision of pebbling P into sub-pebblings P_is. V_i is the set of new vertices pebbled in P_i.

Proof. Consider vertices in $in(V_j)$ at the start of sub-pebbling P_j (Figure 3). All of these vertices are predecessors to some vertices in V_j that will be pebbled during the jth sub-pebbling. Hence, we need to have red or blue pebbles on these $|in(V_j)|$ vertices at the start of P_j. As we only have σ_0 red pebbles, at least $|in(V_j)| - \sigma_0$ of the vertices in $in(V_j)$ have only blue pebbles when P_j starts. Each such vertex leads to at least one input operation during sub pebbling P_j. Similarly, consider vertices in $out(V_j)$. All of these vertices are predecessors to vertices that will be pebbled in some kth sub-pebbling P_k, where $k > j$. Thus, we need to have red or blue pebbles on these $|out(V_j)|$ vertices at the end of the jth sub-pebbling. As we have only σ_0 red pebbles, we need to use at least $|out(V(j)| - \sigma_0$ blue pebbles. Each blue pebble on a vertex of V_j is a result of an output operation during the sub-pebbling P_j. Hence during the sub-pebbling P_j we do at least $|in(V_j)| - \sigma_0 + |out(V_j)| - \sigma_0$ I/Os. This establishes the lemma.

Next we look at how to obtain lower bound for the boundary size, $|boundary(V_j)|$, for a fixed size $|V_j|$. The notion of boundary of a set of vertices of a graph has been extensively studied especially in the context of expander graphs[11]. Isoperimetric parameter of a graph is a way of capturing the notion of the minimum boundary of subgraphs (of the graph) of a fixed size. In context of computation graphs, the boundary of a set of vertices S captures the input and output requirements for completing the pebbling (computation) of the set S. In deriving lower bounds for I/O we focus on a sub pebbling step where we pebble a set of vertices S. We then use the isoperimetric parameter of the graph to derive the lower bound for the boundary size of S, and consequently find the I/O lower bound for the sub pebbling.

Definition 4. *The vertex isoperimetric parameter for a directed graph $G = (V, E)$ is:*

$$\zeta(M, G) = \min_{S \subset V} \{|boundary(S)| \, : \, |S| = M\}$$

Theorem 3. *Let $\tilde{G}$ be a computation structure. Consider any non-repebbling strategy $\mathcal{P}$ for the DAG $\tilde{G}(t) = (V, E)$ in a 2-level memory hierarchy game. Then for any integer $M > 0$ the memory traffic complexity for $\tilde{G}$, $T_1(\hat{\sigma}, \tilde{G})$, satisfies the following lower bound:*

$$T_1(\hat{\sigma}, \tilde{G})(t) \geq \lfloor |V|/M \rfloor (\zeta(M, \tilde{G}(t)) - 2\sigma_0).$$

Proof. Subdivide the pebbling $\mathcal{P}$ into consecutive sequential sub-pebblings $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_h$, where in each sub-pebbling we pebble M new vertices of $\tilde{G}(t)$ except possibly $\mathcal{P}_h$. That defines at least $\lfloor |V|/M \rfloor$ sub-pebblings in which M computation steps occur. From Lemma 1 and definition of vertex isoperimetric parameter, the number of I/O is bounded by $\zeta(M, \tilde{G}(t)) - 2\sigma_0$. Hence, the memory traffic complexity for $\tilde{G}$, $T_1(\hat{\sigma}, \tilde{G})$, satisfies the following lower bound.

$$T_1(\hat{\sigma}, \tilde{G})(t) \geq \lfloor |V|/M \rfloor (\zeta(M, \tilde{G}(t)) - 2\sigma_0).$$

Although this directly provides a lower bound, but this is not the best possible lower bound that can be obtained using the boundary flow technique. To obtain better bounds it is useful to define the vertex isoperimetric parameter for a subset of vertices of a directed graph. This is best illustrated with the help of binomial computation graph. In this graph almost all vertices are internal vertices, and then there are vertices at the "fringes" of the computation graph which constitute a very small fraction (which goes to zero as size of the binomial graph increases) of the total number of vertices see Figure 4. If we use the $\zeta(M, G)$ for the entire graph the set of M vertices that results in minimum boundary size is rooted at the top vertex and is aligned with the two slanted edges (Figure 5). This result is non-optimal because this minimum boundary size is obtained boundary size is obtained only at the fringes and is not representative of minimum boundary size for subsets of size M. On the other hand a typical subset of size M in this graph consists of only of internal vertices. For these subsets of size M the set that gives minimum boundary looks quite different - in fact a hexagonal arrangement of internal vertices gives the minimum boundary size (Figure 5). Hence, to derive strong lower bounds it is necessary to find isoperimetric parameter of the computation graph over a subset of dominating vertices (internal vertices), and then take into account of vertices at the fringes of the computation graph.

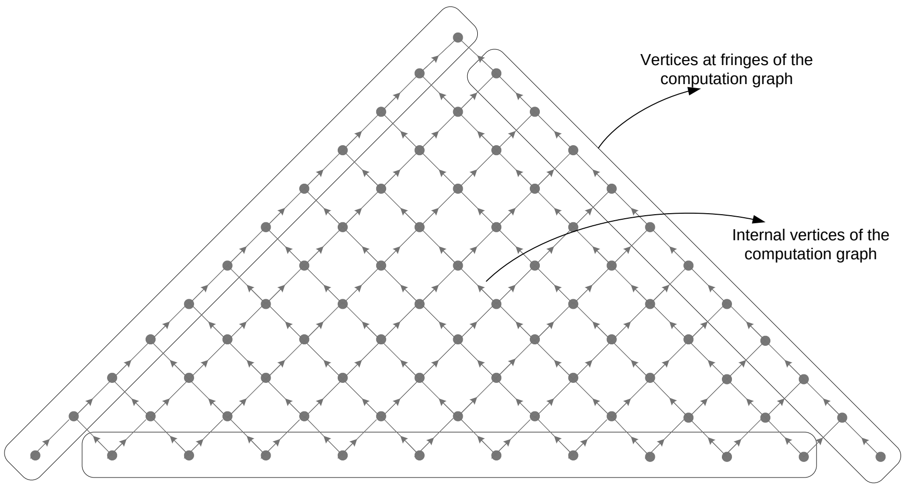
Vertices at fringes of the computation graph
Internal vertices of the computation graph

Fig. 4. Binomial computation graph with fringe and internal vertices.

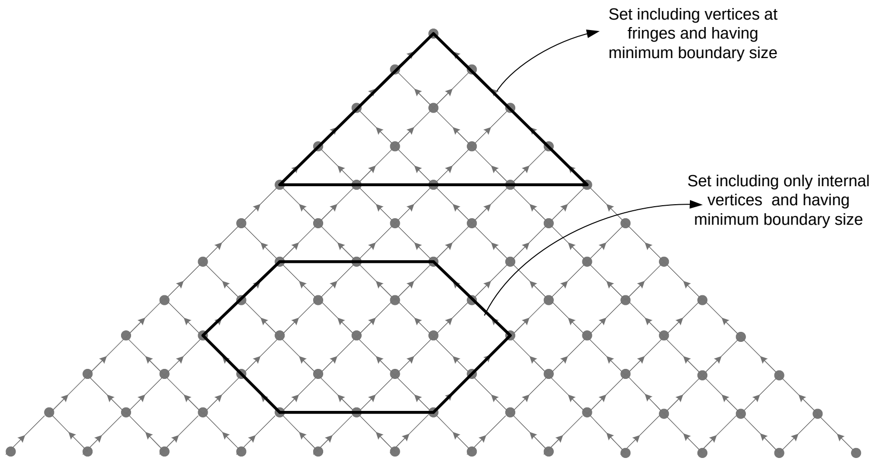
Set including vertices at fringes and having minimum boundary size
Set including only internal vertices and having minimum boundary size

Fig. 5. Sets with minimum boundaries with fixed number of vertices. Hexagonal shape gives minimum boundary when only internal vertices are allowed. Triangular shape shown gives minimum boundary when all vertices all allowed.

Definition 5. *The vertex isoperimetric parameter for a directed graph $G = (V, E)$ over a subset $V_{int} \subset V$ is:*

$$\zeta(M, G, V_{int}) = \min_{S \subset V_{int}} \{|boundary(S)| \, : \, |S| = M\}$$

Definition 6. *Let $G = (V, E)$ be a directed graph. A partition V_{int},V_{ext} of V is called I-partition if there is no edge $(u, v) \in E$ such that $u \in V_{ext}$ and $v \in V_{int}$.*

Definition 7. *Let $G = (V, E)$ be a directed graph let V_{int},V_{ext} be an I-partition of V. Then for $S \subset V_{ext}$ we define*

$$\Gamma(S, G) = \{u \in V_{int} \,|\, v \in S \text{ and } (u, v) \in E\}.$$

Lemma 2. *Consider a directed graph $G = (V, E)$. Let V_{int}, V_{ext} be any I-partition of V. Let $S \subset V$. Let $S_{int} = S \bigcap V_{int}$ and $S_{ext} = S \bigcap V_{ext}$. Then,*

$$|boundary(S) \geq \zeta(|S_{int}|, G, V_{int}) - |\Gamma(S_{ext}, G)|.$$

Proof. Let $w \in in(S_{int})$. Then $w \in V - S_{int}$ and there is a vertex $v \in S_{int}$ such that $(w, v) \in E$. We look at two possible cases for w:

(i) $w \in V - S$. Then $w \in in(S)$ as $(w, v) \in E$ and $v \in S$ since $S_{int} \subset S$.
(ii) $w \in S$. Then $w \in S - S_{int}$, that is $w \in S_{ext} \subset V_{ext}$. Since there are no edges from a vertex in V_{ext} to a vertex in V_{int} this case is not possible.

From this it follows that $|in(S)| \geq |in(S_{int})|$.
Similarly let w be a vertex in $out(S_{int})$. Then $w \in S_{int}$ and there exists $v \in V - S_{int}$ such that $(w, v) \in E$. Once again we consider two possible cases for w:

(i) there exists $v' \in V - S$ such that $(w, v') \in E$. Then $w \in out(S)$.
(ii) there is no $v' \in V - S$ such that $(w, v') \in E$. Hence $v \notin V - S$, that is $v \in S$. Since $v \in V - S_{int}$ this implies that $v \in S - S_{int}$ that is $v \in S_{ext}$.

From this it follows that $|out(S)| \geq |out(S_{int})| - |\Gamma(S_{ext}, G)|$. Hence we can conclude that,

$$\begin{aligned}
|boundary(S)| &= |in(S)| + |out(S)| \\
|boundary(S)| &\geq |boundary(S_{int})| - |\Gamma(S_{ext}, G)| \\
|boundary(S)| &\geq \zeta(|S_{int}|, G, V_{int}) - |\Gamma(S_{ext}, G)|
\end{aligned}$$

The following result follows directly from the above lemma.

Theorem 4. *Let $\tilde{G}$ be a computation structure and let $\mathcal{P}$ be any non-repebbling strategy for the DAG $\tilde{G}(t) = (V, E)$ in a 2-level memory hierarchy game. Let V_{int}, V_{ext} be any I-partition of V. Consider subdivision of the pebbling $\mathcal{P}$ into consecutive sequential sub-pebblings $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_h$, where in each sub-pebbling $\mathcal{P}_i$ we pebble $|S^{(i)}_{int}| = M$ new internal vertices and $|S^{(i)}_{ext}|$ new external vertices of $\tilde{G}(t)$ except possibly $\mathcal{P}_h$, where $S^{(i)}_{int} \subset V_{int}$, and $S^{(i)}_{ext} \subset V_{ext}$. Then for any integer $M > 0$ the memory traffic complexity for $\tilde{G}$, $T_1(\hat{\sigma}, \tilde{G})$, satisfies the following lower bound:*

$$T_1(\hat{\sigma}, \tilde{G})(t) \geq \lfloor |V_{int}|/M \rfloor (\zeta(M, \tilde{G}(t), V_{int}) - 2\sigma_0) - \sum_{i=1}^{\lceil |V_{int}|/M \rceil} |\Gamma(S^{(i)}_{ext}, \tilde{G}(t))|.$$

4.1 Memory Traffic Complexity for Binomial Computation Graph

For the binomial computation graph $G_b(n) = (V, E)$, we define the *I-partition* with V_{int} consisting of internal vertices and V_{ext} consisting of vertices at the fringes (Figure 4). More specifically,

$$V_{int} = \{u \in V \mid \textit{in-degree}(u) = 2 \text{ and } \textit{out-degree}(u) = 2\}$$
$$V_{ext} = V - V_{int}.$$

We first need to find the VIP for the binomial computation graph over a subset of *internal vertices*. Our VIP result for binomial computation graph is similar to a recently solved, long-standing honeycomb conjecture, which states a hexagonal grid represents the best way to divide a surface into regions of equal area with the least total perimeter [8]. Somewhat counter-intuitively, we have shown that the lowest boundary to area ratio in a binomial computation graph (which can be easily mapped to a grid) over a subset of internal vertices is obtained by a hexagonal structure[**?**]. More precisely, we prove the following:

Theorem 5 ([?]). *Consider any pebbling of $G_b(n) = (V, E)$ via a non-repebbling strategy $\mathcal{P}$. Let V_{int}, V_{ext} be the I-partition of $G_b(n)$. Let $S \subset V_{int}$ be any set of internal vertices of $G_b(n)$ that are pebbled in a sub-pebbling of $\mathcal{P}$. Assume that $|S| \geq 3c^2 + 3c + 1$. Then $\zeta(3c^2 + 3c + 1, G_b, V_{int}) \geq 6c + 3$.*

The "shape" of pebbled vertices that realizes the above bound turns out to be a hexagon. The theorem is interesting in its own right and a complete proof of this theorem is quite involved. A manuscript of the proof is available at [**?**]. Applying Theorem 4 and selecting $c = (2/3)\sigma_0$, we have the following result (for keeping the expression simple, we assume that the primary memory size, σ_0, is such that $c = (2/3)\sigma_0$ is an integer):

Theorem 6. *The memory traffic complexity of $G_b(n)$ on a 2-level memory hierarchy system satisfies*

$$T_1(\hat{\sigma}, G_b)(n) \geq \left\lfloor \frac{(n-2)(n-1)/2}{(4/3)\sigma_0^2 + 2\sigma_0 + 1} \right\rfloor (2\sigma_0 + 3) - (2n + 1).$$

Proof. Let $\mathcal{P}$ be any non-repebbling strategy for the DAG $G_b)(n)$ in a 2-level memory hierarchy game. Let V_{int}, V_{ext} be the *I-partition* of V. Consider subdivision of the pebbling $\mathcal{P}$ into consecutive sequential sub-pebblings $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_h$, where in each sub-pebbling $\mathcal{P}_i$ we pebble $|S^{(i)}_{int}| = (4/3)\sigma_0^2 + 2\sigma_0 + 1$ new internal vertices and $|S^{(i)}_{ext}|$ new external vertices of $G_b)(n)$ except possibly $\mathcal{P}_h$, where $S^{(i)}_{int} \subset V_{int}$, and $S^{(i)}_{ext} \subset V_{ext}$. Observe that for the binomial computation graph, we have $|\Gamma(S^{(i)}_{ext}, G_b)(n))| = |S^{(i)}_{ext}|$. As we are pebbling (computation steps) $|S^{(i)}_{ext}|$ new external vertices in each sub-pebbling, the total number of external vertices that are pebbled by $\mathcal{P}$ is given by the vertices on the slanted edges of the binomial computation graph, that is $2n + 1$. Hence, using Theorem 4 we have the following result on the memory traffic complexity of $G_b(n)$ on a 2-level memory hierarchy.

$$T_1(\hat{\sigma}, G_b)(n) \geq \left\lfloor \frac{(n-2)(n-1)/2}{(4/3)\sigma_0^2 + 2\sigma_0 + 1} \right\rfloor (2\sigma_0 + 3) - (2n+1).$$

The above result is roughly three times stronger than the one obtained using S-span approach. In [14] we provide a simple computation scheme for the binomial computation graph that we believe is memory traffic optimal. The memory traffic in this scheme is 4/3 times of the lower bound established by the boundary flow technique. Moreover, the scheme doesn't use any re-pebbling and places a red pebble on a blue pebble exactly once.

4.2 Memory Traffic Complexity for FFT Computation Graph

We first introduce the necessary notations.

Definition 8. *We identify vertices in $G_f(n)$ with its level number and its position on this level. For $1 \leq i \leq \lg n + 1, 1 \leq j \leq n$, $v(i,j)$ denotes is the j^{th} vertex on the i^{th} level.*

For $1 \leq j \leq n$, we refer to the set of vertices $\{v(i,j) \,|\, 1 \leq i \leq \lg n + 1\}$ as $Column(j)$. Note that, if $j \neq j'$ then $Column(j)$ and $Column(j')$ are disjoint.

Definition 9. *We say that a vertex u is an ancestor of a vertex v if there is a directed path from u to v. We include u also in the set of ancestors of u.*

We need the lemma below to complete our proof.

Lemma 3.

$$\forall\, x, y \;\; 2^x + 2^y \geq 2^{1+\frac{(x+y)}{2}}.$$

Proof. Since each of $2^x, 2^y$ and $2^{1+\frac{(x+y)}{2}}$ is non-negative and $(2^x + 2^y)^2 = (2^x - 2^y)^2 + [2^{1+\frac{(x+y)}{2}}]^2$ the inequality follows trivially.

Lemma 4. *Let $U \subset G_f(n)$ with $|U| \geq k \log_2 4k$ for some $k \geq 1/4$. Then $|out(U)| \geq k$.*

Proof. Partition vertices of U by columns. Notice that the topmost vertex in any column is an output vertex for U. Hence, if the set U has vertices in k or more columns of $G_f(n)$, then $|out(U)| \geq k$.

Otherwise, all vertices of U are in fewer than k columns. In this case, we show again that $|out(U)| \geq k$. We establish this by defining a process, $\mathcal{VAL}$, on U (see below) and proving a claim about $\mathcal{VAL}$.

During the process, each vertex $v \in U$ stores a value $n(v)$ at all times. Initially $\mathcal{VAL}$ sets $n(v) = 1$ for all $v \in U$. During the process the values $n(v)$ are updated. However, during the entire process, the sum of these values over all vertices remains the same ($\Sigma_v n(v) = |U|$). At the end of the process only vertices in $out(U)$ have non-zero values. The claim below shows that these values are bounded above by $\log_2 4k$, which gives us the desired lower bound on $|out(U)|$.

```
Process $\mathcal{VAL}(U)$:
for each vertex $v \in U$
  $n(v) \leftarrow 1$
Let $b$ be the bottommost level with a vertex in $U$
Let $h$ be the topmost level with a vertex in $U$
$i \leftarrow b$
while $(i < h)$
  begin
    for each vertex $v(i, j) \in U$ at level $i$ do
      if $v(i, j) \notin out(U)$ then
        let $v(i + 1, j_1)$ and $v(i + 1, j_2)$ be the two “children” of $v(i, j)$
        $n(v(i + 1, j_1)) \leftarrow n(v(i + 1, j_1) + n(v(i, j))/2$
        $n(v(i + 1, j_2)) \leftarrow n(v(i + 1, j_2) + n(v(i, j))/2$
        $n(v(i, j)) \leftarrow 0$
    $i \leftarrow i + 1$
  end
```

Note that during the process, for a vertex v at a level i, the value $n(v)$ can be updated only when either its immediate predecessors (which, if present in U, are on level $i-1$) are considered or when v is considered itself. Also, notice that $n(v)$ for any node $v \in out(U)$ changes in a non-decreasing fashion. Moreover, $n(v)$ for any node $v \notin out(U)$ also changes in a non-decreasing fashion until the time when v is considered at which time it is set to 0 and never changes again. Define $n_{max}(v)$ to be the maximum value that $n(v)$ stores during the entire process. It is easy to see that for a vertex v at level i, $n(v) = n_{max}(v)$ after level $i - 1$ has been considered by $\mathcal{VAL}$ and when $\mathcal{VAL}$ hasn't started considering level i.

Claim 1: At the end of the process $\mathcal{VAL}$ only vertices in $out(U)$ have non-zero values. Moreover, for all vertices v, v has ancestors in at least $2^{n_{max}(v)-2}$ different columns.

Proof (of Claim 1): The process considers each vertex $v \in U$ at most once (vertices at the topmost level h in U are not considered at all). Hence it terminates. At the end of this process, only the vertices in $out(U)$ will have a non-zero values as the value of each vertex $v \notin out(U)$ at a level i gets set to zero when vertices at level i are considered (and it is never reset again). Note that all vertices at level h are in $out(U)$. We will prove the rest of the claim by induction on the level number in which the vertex v is located.

Base Case: For any vertex v at the bottom level b, $n_{max}(v) = 1$. Hence, for each vertex v at the bottom level it is trivially true that v has at least $2^{n_{max}(v)-2}$ ancestors since v is its own ancestor and $2^{n_{max}(v)-2} = \frac{1}{2}$.

Induction: Consider a vertex v at level i. Let u, u' be the two immediate predecessors of v in $G_f(n)$. The value $n_{max}(v)$ depends only on the situation of u and u' with respect to U. Call u *relevant* for v if $u \in U$ and the two edges going out of u are to vertices also in U. Similarly u' is *relevant* for v if $u' \in U$ and the two edges going out of u' are to vertices also in U. Notice that u or u' can affect $n(v)$ during process $\mathcal{VAL}$ only if it is relevant for v. We now consider three cases:

(i) Both u, u' are not relevant for v. In this case $n_{max}(v) = 1$ and v trivially has ancestors in $2^{n_{max}(v)-2}$ different columns.

(ii) u is relevant for v and u' is not. In this case,
$n_{max}(v) = 1 + n_{max}(u)/2$.
Since u is at level $i-1$, by induction, u (and hence v) has ancestors in at least $2^{n_{max}(u)-2}$ different columns. We need to show that v has ancestors in at least $2^{n_{max}(v)-2}$ different columns. If $n_{max}(v) \leq 2$ the statement is trivially true. Otherwise, $n_{max}(v) > 2$ which implies that $n_{max}(u) = 2(n_{max}(v)-1) = n_{max}(v)+n_{max}(v)-2 > n_{max}(v)$. Hence v has ancestors in at least $2^{n_{max}(v)-2}$ different columns.

(iii) Both u, u' are relevant for v. In this case,
Note that $n_{max}(v) = 1 + n_{max}(u)/2 + n_{max}(u')/2$. The ancestors of u and u' are disjoint and in different columns. Thus, by the induction hypothesis v has ancestors in at least $2^{n_{max}(u)-2}$ different columns and u' has ancestors in at least $2^{n_{max}(u')-2}$ different columns. Hence v has ancestors in at least $2^{n_{max}(u)-2} + 2^{n_{max}(u')-2}$ different columns. From Lemma 3, it follows that $2^{n_{max}(u)-2} + 2^{n_{max}(u')-2} \geq \frac{1}{4} \cdot 2^{1+\frac{(n_{max}(u)+n_{max}(u'))}{2}} = 2^{n_{max}(v)-2}$ or v has ancestors in at least $2^{n_{max}(v)-2}$ different columns.

This finishes the proof of the claim.

Now, recall that U has vertices in fewer than k different columns. Hence, for any vertex v, $n_{max}(v)$ must satisfy $2^{n_{max}(v)-2} < k$, or that $n_{max}(v) < \log_2 k+2 = \log_2 4k$.

Since, $\Sigma_v n(v) = |U|$ always and $n(v) = 0$ at the end of the process for vertices $v \notin out(U)$, it now follows that $\Sigma_{v \in out(U)} n(v) = |U|$. From this it follows that $|out(U)| \geq |U|/\log_2 4k \geq k$ which establishes the lemma.

Lemma 5. *Let $U \subset G_f(n) - \{v(1,j) \mid 1 \leq j \leq n\}$ with $|U| \geq k \log_2 4k$ for some $k > 1/4$. Then $|in(U)| \geq k$.*

Proof. The proof follows easily by observing that

(i) the graph $G_f^R(n)$obtained by reversing all edges of $G_f(n)$ is isomorphic to $G_f(n)$

(ii) for any $U \subset (G_f(n) - \{v(1,j) \,| 1 \leq j \leq n\}), out_{G_f^R(n)}(U \cup in_{G_f(n)}(U)) \subseteq in_{G_f(n)}(U)$.

Using Lemma 4 and Lemma 5, we can state the following theorem.

Theorem 7. *The A-boundary for $G_f(n)$ with $A \geq k \log 4k$ satisfies $\zeta(A, G_f(n)) \geq 2k$.*

Comment: Using a different approach, it is actually possible to prove a slightly stronger (and optimal) result. The A-boundary for $G_f(n)$ with $A \geq k \log 2k$ satisfies $\zeta(A, G_f(n)) \geq 2k$. This can be used to slightly strengthen the result in Theorem 6.

Applying Theorem 3 and selecting $k = \sigma_0 \log 2\sigma_0$, we have the following result.

Theorem 8. *The memory traffic complexity of $G_f(n)$ on a 2-level memory hierarchy system satisfies*

$$T_1(\hat{\sigma}, G_f)(n) \geq \left\lfloor \frac{n \log n}{(\sigma_0 \log 2\sigma_0) \log(4\sigma_0 \log 2\sigma_0)} \right\rfloor (2\sigma_0 \log 2\sigma_0 - 2\sigma_0)$$

The above result is roughly eight times stronger than the one obtained using S-span approach, and is nearly optimal as the simple FFT algorithm described in [17] has a memory traffic complexity of $(2n \log n)/(\log \sigma_0)$.

5 Conclusion

We presented a new technique for deriving lower bounds on memory traffic for computations that can be represented by a DAG. We demonstrated the effectiveness of this technique on two important computation structures. We improved the best known lower bound for a binomial computation graph by a factor of three. For an FFT computation graph, we improved the bound by a factor of 8 to obtain a nearly optimal lower bound. There is a gap of a factor of 4/3 between the upper bound and lower bound for binomial computation graph. It would be nice to close this gap one way or the other. The bounds derived in this paper assumes non-repebbling strategies, which form an important class of strategies. It is of interest to see whether these bounds also hold for pebbling strategies that allow re-pebbling.

References

1. Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
2. E. Benhamou. Fast Fourier Transform for discrete Asian options. Computing in Economics and Finance 2001 6, Society for Computational Economics, April 2001.
3. Sergei Bezrukov. Edge isoperimetric problems on graphs. In L. Lovasz, A. Gyarfas, G.O.H. Katona, A. Recski, and L. Szekely, editors, *Graph Theory and Combinatorial Biology*, pages 157–197. Bolyai Soc. Math. Stud. 7, 1999.
4. P. Carr and D. Madan. Option valuation using the Fast Fourier Transform, 1998.
5. Zhixiang Chen, Bin Fu, Yong Tang, and Binhai Zhu. A ptas for a disc covering problem using width-bounded separators. *J. Comb. Optim.*, 11(2):203–217, 2006.
6. John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, September 1979.
7. Bin Fu. Theory and application of width bounded geometric separators. *J. Comput. Syst. Sci.*, 77(2):379–392, 2011.
8. Thomas C. Hales. The honeycomb conjecture. *Discrete & Computational Geometry*, 25(1):1–22, 2001.
9. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, 2007.
10. J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Ann. ACM Symp. on Theory of Computing*, pages 326–333, 1981.

11. Shlomo Hoory, Nathan Linial, Avi Wigderson, and An Overview. Expander graphs and their applications. *Bull. Amer. Math. Soc. (N.S*, 43:439–561, 2006.
12. V. Kumar, A. Sameh, A. Grama, and G. Karypis. Architecture, algorithms and applications for future generation supercomputers. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 346, Washington, DC, USA, 1996. IEEE Computer Society.
13. Y.K. Kwok. *Mathematical Models of Financial Derivatives.* Springer-Verlag, Singapore, 1998.
14. Desh Ranjan, John Savage, and Mohammad Zubair. Upper and lower i/o bounds for pebbling r-pyramids. In *IWOCA 2010: To appear in the proceedings of the 21st International Workshop on Combinatorial Algorithms.* Springer, 2010. `http://www.cs.odu.edu/~zubair/papers/PyramidRSZ.pdf`.
15. Desh Ranjan and Mohammad Zubair. Vertex isoperimetric parameter of a computation graph, 2010. `www.cs.odu.edu/~zubair/papers/hexagonIJFCS.pdf`.
16. John E. Savage. Extending the Hong-Kung model to memory hierarchies. In Ding-Zhu Du and Ming Li, editors, *Computing and Combinatorics*, pages 270–281. Springer-Verlag, Lecture Notes in Computer Science, 1995.
17. John E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison Wesley, Reading, Massachusetts, 1998.
18. John E. Savage and Mohammad Zubair. Evaluating multicore algorithms on the unified memory model. *Scientific Programming*, 17(4):295–308, 2009.
19. John E. Savage and Mohammad Zubair. Cache-optimal algorithms for option pricing. *ACM Trans. Math. Softw.*, 37(1):1–30, 2010.
20. Clark David Thompson. *A complexity theory for VLSI.* PhD thesis, Pittsburgh, PA, USA, 1980.
21. Jonathan Yackel, Robert Meyer, and Ioannis Christou. Minimum-perimeter domain assignment. *Mathematical Programming*, 78:283–303, 1997. 10.1007/BF02614375.