Upper and Lower I/O bounds for pebbling r-pyramids

Desh Ranjan[1], John Savage[2], and Mohammad Zubair[3]

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

Abstract. Modern computers 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 present I/O efficient algorithms to pebble r-pyramids and derive lower bounds on the number of I/O steps to do so. The r-pyramid graph models financial applications which are of practical interest and where minimizing memory traffic can have a significant impact on cost saving.

Key words: Memory hierarchy, I/O, Lower bounds

1 Introduction

Modern computers have several levels of memory hierarchy. To obtain good performance on these computers it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy [1, 2]. The cache blocking technique is used to reduce memory traffic to slower memories in the hierarchy [1]. Cache blocking partitions a given computation such that the data required for a partition fits in a processor cache. For computations, where data is reused many times, this technique reduces memory traffic to slower memories in the hierarchy [1]. The memory traffic reduction that can be obtained using this technique depends on the application, memory hierarchy architecture, and the effectiveness of the blocking algorithm.

In this paper, we present I/O efficient algorithms to compute the values at vertices ("pebble" the vertices) of a computation graph that is an r-pyramid and derive lower bounds on its memory traffic complexity. A formal definition of memory traffic complexity is given later in the paper. 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 [3]. (See also [4, Chapter 11].) This model is an extension of the red-blue model introduced by [5], a game played on directed acyclic graphs with red and blue pebbles.

The paper is motivated by a very practical financial application - that of computing option prices. An option contract is a financial instrument that gives

the right to its holder to buy or sell a financial asset at a specified price referred to as strike price, on or before the expiration date. The current asset price, volatility of the asset, strike price, expiration time, and prevailing risk-free interest rate determine the value of an option. Binomial and trinomial option valuation are two popular approaches that value an option using a discrete time model [6, 7]. The binomial option pricing computation is modelled by the directed acyclic pyramid graph $G_{biop}^{(n)}$ with height n and $n+1$ leaves shown in Figure 1. Here the expiration time is divided into n intervals (defined by $n+1$ endpoints), the root is at the present time, and the leaves are at expiration times. We use $G_{biop}^{(n)}$ to determine the price of an option at the root vertex iteratively, starting from the leaf vertices.

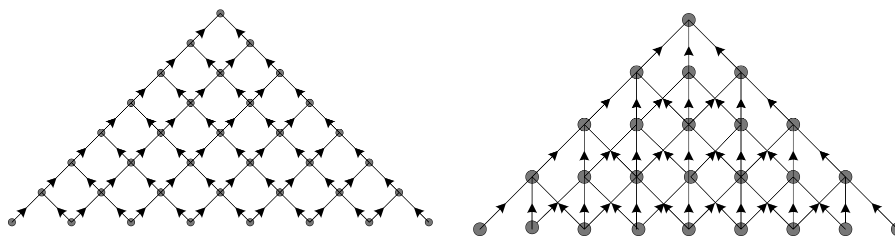

Fig. 1. A 2-pyramid representing binomial computation, and a 3-pyramid representing trinomial computation.

The trinomial model improves over the binomial model in terms of accuracy and reliability [6]. The trinomial option pricing computation is represented using the directed acyclic graph with in-degree 3 denoted $G_{triop}^{(n)}$ of height n on $2n+1$ leaves shown in Figure 1. As in the binomial model, we divide the time to expiration into n intervals and let the root be at the present time and the leaves be at expiration times. As in the binomial model, we use $G_{triop}^{(n)}$ to determine the price of an option at the root vertex iteratively, starting from the leaf vertices. The trinomial model assumes that the price of an asset can go three ways: up, down, and remain unchanged. This is in contrast to the binomial model where the price can only go two ways: up and down.

In [8] the authors derived lower bounds for memory traffic at different levels of memory hierarchy for $G_{biop}^{(n)}$ and $G_{triop}^{(n)}$. The technique used in the paper is based on the concept of a S-span of the DAG [3]. 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 levels (those further away from the CPU) memories.

In this paper we first define a general family of graphs called r-pyramids. $G_{biop}^{(n)}$ and $G_{triop}^{(n)}$ are sub families of this family. We then provide an algorithm to pebble $r-pyramids$ using S pebbles that requires roughly half the I/O needed by previously described algorithms [8]. We also provide a lower bound that is

twice the previous best known lower bound for the same problem [8]. With these improvements, one can prove that the pebbling scheme presented here does no more than twice the I/O required by an optimal pebbling scheme.

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 application.

The rest of the paper is organized as follows. The required definitions and the memory hierarchy model that helps in developing memory complexity is discussed in Section 2. In Section 3 we present an efficient algorithm, in terms of memory I/O, for pebbling r-pyramid. Section 4 gives improved lower bounds for the r-pyramid graph. Finally, in Section 5 we present some open problems.

2 Background

2.1 Computation Graphs, Structures and Memory Traffic Complexity

We define here formally what we mean by a computation graph, a computation structure and memory traffic complexity of a computation structure. 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 idea here is that we wish 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 $\tilde{G} : N^k \rightarrow \{G \,|\, G \text{ is a computation graph}\}$.

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 $t \in N^k$. An input in a 2-level memory hierarchy refers to a read from secondary memory, and an output refers to a write to the secondary memory. 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 $\tilde{G} : N^k \rightarrow \{G \,|\, G \text{ is a computation graph}\}$ 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 $t \in N^k$. The function $T_1(\hat{\sigma}, \tilde{G}) : N^k \rightarrow N$ 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 $\tilde{G}$.

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. Recall, that 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.

Red-Blue Pebble Game Rules

- (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 An Efficient Algorithm for Pebbling $P_r(n)$

3.1 An r-pyramid

A directed graph $G = (V, E)$ is called a layered graph with n levels if V can be written as a disjoint union of n non-empty sets $V_1, V_2, \ldots, V_n$ such that $\forall\ e = (u, v) \in E, \exists\ i$ such that $u \in V_i$ and $v \in V_{i+1}$.

Definition 1. *An r-pyramid of height n, $P_r(n)$, is a graph $(V_r(n), E_r(n))$ with the following properties (see Figure 2):*

1. *$P_r(n) = (V_r(n), E_r(n))$ is a layered graph with height n. Here $V_r(n) = V_1 \cup V_2 \ldots \cup V_{n+1}$, V_i is the set of vertices on level i, and $E_r(n)$ are the edges.*
2. *V_i has $n_r(i) = (r-1)*(i-1)+1$ vertices labeled $v(i,1), v(i,2), \ldots, v(i, n_r(i))$*

3. Vertex $v(i,j)$ has r incoming edges from vertices $v(i+1,j), v(i+1,j+1), \ldots, v(i+1,j+r-1)$.
4. There are no other edges in $P_r(n)$.

With this definition it is easy to see that $G^{(n)}_{biop}$ is a 2-pyramid of height n (or $P_2(n)$) and $G^{(n)}_{triop}$ is a 3-pyramid of height n (or $P_3(n)$). Also, note that an $P_r(n)$ has $|V_r(n)| = (n+1)((r-1)n+2)/2$ vertices. We note the nice recursive structure of r-pyramid. For any vertex v in the r-pyramid, the *subgraph rooted at* v is a smaller r-pyramid itself.

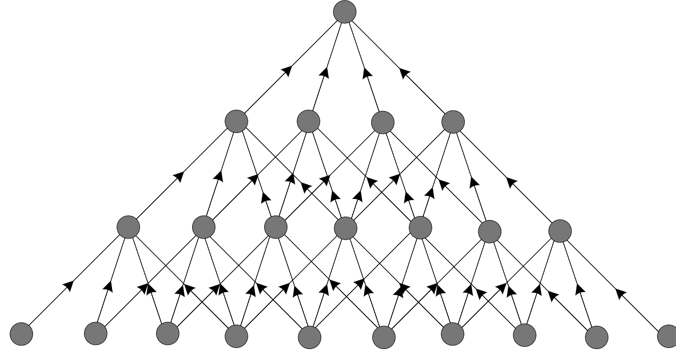

Fig. 2. r-pyramid $P_r(n)$ with $r = 4$ and $n = 3$

3.2 Algorithm

Let, $S = (r-1)m + 1$. We give an algorithm that we can pebble an r-pyramid $P_r(n) = (V_r(n), E_r(n))$ of height n with S red pebbles using no more than $2|V_r(n)|(r-1)/(S-1)$ I/O operations. Note that if $n \leq m$ then $P_r(n)$ can be pebbled without any intermediate I/O. Recall that we are assuming an unlimited supply of blue pebbles.

Let $D^k_{i,j}$ denote the "diagonal" shown in Figure 3 consisting of the k vertices $\{(i,j), (i+1, j+r-1), \ldots, (i+(k-1), j+(k-1)(r-1))\}$ that originate at the vertex (i,j).

The algorithm starts with the pebbling of the r-pyramid, $P^m_{n-m,1}$, of height m rooted at vertex $(n-m, 1)$. This pyramid shares inputs with the inputs to the full pyramid. This is done in a such way that it leaves S red pebbles on S vertices of $P^m_{n-m,1}$ one of which is $(n-m, 1)$. The other vertices are those in $P^m_{n-m,1}$ that are required to compute $D^m_{n-m,2}$. More precisely, this is a collection

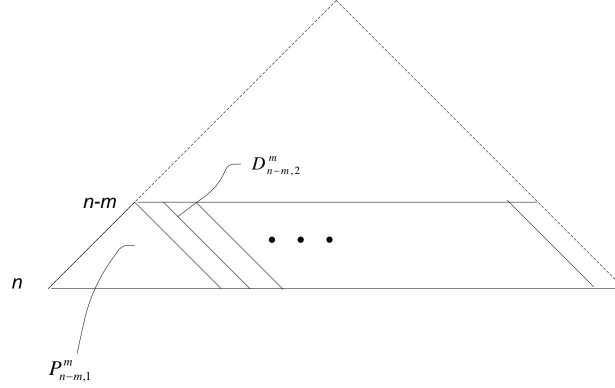

Fig. 3. Processing of r-pyramid at level k

of $(r-1)$ vertices at each of the lower $m-1$ levels. These vertices are

$$\begin{array}{r}(n-m+1,2),(n-m+1,3),\ldots,(n-m+1,(r-1))+1)\\(n-m+2,r),(n-m+2,2),\ldots,(n-m+2,2(r-1)+1)\\\vdots\\(n,(m-1)(r-1)),(n,(m-1)(r-1)+1),\ldots,(n,m(r-1))\end{array}$$

Procedure *PebbleSubPyramid* given in Algorithm 1 explains how this is done.

Procedure *PebbleSubPyramid*(n)
if $n \le m$ **then**
    Pebble the whole subpyramid using $(r-1)*n+1$ red pebbles ;
else
    $t \leftarrow S$;
    for $i = 1$ *to* t **do**
        Place a red pebble at vetex (n,i);
    end
    for $j = 0$ *to* $m-1$ **do**
        $t \leftarrow t-(r-1)$;
        for $k = 1$ *to* t **do**
            move pebble at $(n-j,k)$ to $(n-j-1,k)$;
        end
    end
end

Algorithm 1: An algorithm for pebbling an r-subpyramid of height m at position $(n-m,1)$ using $S=(r-1)m+1$ red pebbles leaving the red pebbles at the vertices needed for future pebbling.

Next we repeatedly pebble the diagonals $D^m_{n-m,i}$ starting with $i = 2$ and progressing incrementally all the way to $D^m_{n-m,(n-m-1)(r-1)+1}$. Observe that pebbling of $D^m_{n-m,2}$ requires the red pebbles on exactly $S-1$ vertices from the pyramid $P^m_{n-m,1}$ that was pebbled earlier (using *PebbleSubpyramid*) and a red pebble on vertex $(n, s+1)$. We place a blue pebble at $(n-m, 1)$ move the red pebble at $(n-m, 1)$ left by *PebbleSubpyramid* to $(n, s+1)$.

It is now easy to verify that all the red pebbles are in exactly the needed locations to compute $D^m_{n-m,2}$. Moreover, we can maintain this property while pebbling consecutive diagonals. That is, after pebbling $D^m_{n-m,2}$ we leave S red pebbles on the vertices that are required for the processing of the next diagonal $D^m_{n-m,3}$ etc. Observe that in general, processing of diagonal $D^m_{n-m,j}$ requires input from vertices on diagonals $D^m_{n-m,j-1}, D^m_{n-m,j-2}, \ldots, D^m_{n-m,j-r+1}$. This way we continue processing diagonals until we process the last diagonal $D_{n-m,(r-1)(n-m-1)+1}$.

Also, observe that while processing these diagonals we only need to preserve vertices at $(n-m, 1), (n-m, 2), \ldots, (n-m, (r-1)(n-m-1)+1)$ for future processing. The basic idea is that with S pebbles we can pebble all vertices at the lower m levels blue pebbling only the vertices at level m. We then repeat this process for the r-pyramid of height $n-m$. The complete algorithm to process $P_r(n)$ is presented in Algorithm 2 and illustrated in Figure 3.

Procedure *PebblePyramid*(n)
PebbleSubPyramid(n);
for $j = 2$ *to* $(r-1)(n-m-1)+1$ **do**
  place a blue pebble on $(n-m, j-1)$;
  move the red pebble on $(n-m, j-1)$ to $(n, j+s-1)$;
  for $i = 0$ *to* $m-1$ **do**
    move the red pebble on $(n-i-1, (j+s-1-(r-1)i)$ to
    $(n-i, (j+s-1-(r-1)i)$;
  end
end
PebblePyramid$(n-m)$;

Algorithm 2: An algorithm to pebble an r-pyramid of height n.

Notice that this pebbling scheme does not "re-pebble" any vertex, that is, a vertex is never pebbled red using the *computation step* rule (Section 2.2) more than once. Additionally, it uses a blue pebbled vertex exactly once for input. It is obviously an optimal scheme in terms of computation. It is natural to ask the question if this is also an I/O optimal scheme. We conjecture that this is indeed the case. To prove this, we need to establish lower bounds on pebbling schemes for pebbling an r-pyramid. We do so in the following section.

4 Lower Bounds for Pebbling an r-Pyramid

Lower bounds for pebbling an r-pyramid can be obtained by using S-span arguments [8].

4.1 A Lower Bound Based on the S-Span of a Graph

In this approach, to derive lower bounds for a given DAG, we first compute its S-span. This 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 2. *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. It has provided good lower bounds on communication traffic for matrix multiplication, the Fast Fourier Transform, the binomial graph and other graphs. This definition applies even if G is not a connected graph.

The following theorem [9] relates the S-span of the graph to its memory traffic complexity.

Theorem 1. *Let $\tilde{G}$ be a computation structure. Consider a pebbling of the DAG $\tilde{G}(t)$ in an 2-level memory hierarchy game. Let $\rho(S, \tilde{G}(t))$ be the S-span of $\tilde{G}(t)$ and $|V_t^*|$ be the number of vertices in $\tilde{G}(t)$ other than the inputs. Assume that $\rho(S, \tilde{G}(t))/S$ is a non-decreasing function of S.*

Then the memory traffic complexity for $\tilde{G}$, $T_1(\hat{\sigma}, \tilde{G})$, satisfies the following lower bound.

$$T_1(\hat{\sigma}, \tilde{G})(t) \geq \frac{\sigma_0 |V_t^*|}{\rho(2\sigma_0, \tilde{G}(t))}$$

Lemma 1. *For a given path π from a leaf vertex x_1 to the output vertex x_{p+1} in $P_r(p)$ consisting of vertices $x_1, x_2, x_3, \ldots, x_{p+1}$ there is a total of $(r-1)p$ distinct paths from leaf vertices to the x_i's for $i > 1$.*

Proof. We use induction on p to prove this result. The lemma holds for the base case $P_r(1)$. Assume the lemma is true for $P_r(p)$ rooted at x_{p+1}. Then for a given path π of length p in $P_r(p)$ consisting of vertices $x_1, x_2, \ldots, x_{p+1}$, we have $(r-1)p$ distinct paths from leaf vertices of $P_r(p)$ to x_i's for $i > 1$. Observe that the leaf vertices corresponding to these paths along with x_1 are the total number of leaf vertices in $P_r(p)$, which is $(r-1)p+1$. We now consider $P_r(p+1)$ rooted at x_{p+2}. $P_r(p+1)$ has $(r-1)(p+1)+1$ leaf vertices. Observe that $P_r(p)$

is a sub-graph of $P_r(p+1)$ and the vertex x_1 has r edges coming from the leaf vertices of $P_r(p+1)$, see Figure 4. Let one of these leaf vertices in $P_r(p+1)$ be x_0. Additionally, for every other leaf vertex of $P_r(p)$, we can identify a distinct leaf vertex in $P_r(p+1)$, which it is connected to, see Figure 4. This demonstrate that for a path in $P_r(p+1)$ consisting of vertices $x_0, x_1, x_2, \ldots, x_{p+2}$, there are a total of $(r-1)(p+1)$ distinct paths from leaf vertices to vertices on this path. This completes the proof.

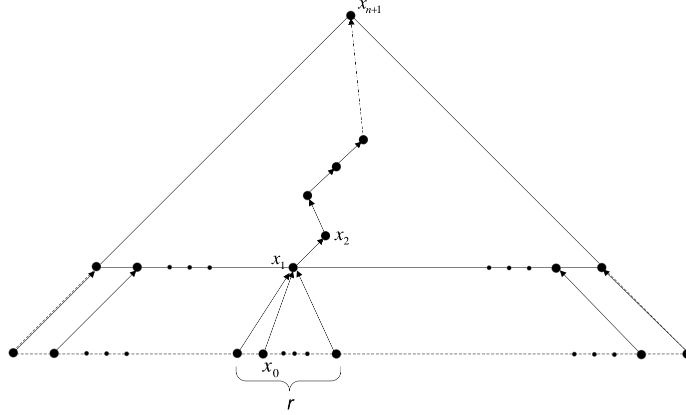

Fig. 4. A r-pyramid with a path π

Lemma 2. *$P_r(p)$ requires a minimum of $S = (r-1)p + 1$ pebbles to place a pebble on the root vertex. The graph can be pebbled completely with S pebbles without repebbling any vertices.*

Proof. The proof uses an argument analogous to the last path argument used in [10]. We say that a path π from a leaf vertex $x_1 \in P_r(p)$ to the root vertex x_{p+1} is **blocked** (at some time instance t) if at least one vertex on the path has a pebble (at time t). Consider the time instance when the root vertex, x_{p+1}, of $P_r(p)$ was pebbled. At this time instance, all paths from all the leaf vertices of $P_r(p)$ to x_{p+1} are blocked. Now let us consider the first time instance t' when all paths from all the leaf vertices to x_{p+1} were blocked. Then at time instance $t'-1$, there must have been an open path from one of the bottom level vertices to x_{p+1}. This implies that all vertices on this path did not have pebbles on them and that at time t' by placing a pebble at the leaf vertex all paths were blocked. Observe that when a pebble is placed on the leaf vertex to block π, the graph already had pebbles on each of the $(r-1)p$ distinct paths leading to each of

the p other vertices on π (Lemma 1). Thus, when the input to π is pebbled, the graph has at least $(r-1)p+1$ pebbles on it.

To show that the graph can be pebbled completely without re-pebbling any vertices, place all $(r-1)p+1$ pebbles on the inputs. Then one can slide the leftmost pebble up one level and then proceed to slide (r-1)(p-1) more pebbles up one level to pebble the leaves of the subgraph $P_r(p-1)$ with $(r-1)(p-1)+1$ leaves. The rest follows by induction. Procedure PebbleSubpyramid provided earlier formally describes this process.

Theorem 2. *The S-span of an r-pyramid is*

$$\tfrac{1}{2}(\lfloor S/(r-1)\rfloor + 1)(2S - (r-1)\lfloor S/(r-1)\rfloor).$$

We present a complete proof for $r = 2$ in Appendix A. The proof for general r is analogous and is omitted because of space limitations.
Applying Theorem 1 we have the following result.

Theorem 3. *Let $\sigma_0 = S$. The memory traffic complexity of P_r on a 2-level memory hierarchy system, $T_1(\hat{\sigma}, P_r)$, satisfies*

$$T_1(\hat{\sigma}, P_r)(n) \geq \frac{Sn((r-1)(n-1)+2)}{(\lfloor 2S/(r-1)\rfloor + 1)(4S - (r-1)\lfloor 2S/(r-1)\rfloor)}.$$

4.2 The Blue Pebble Strategy for Proving Pebbling Lower Bounds

The above results leave a gap of a factor of 4 between the bounds achieved by the scheme provided and the lower bounds obtained. We improve this by strengthening the lower bound. To do so, we develop a new technique for proving lower bounds on I/O in pebbling schemes. We start by making a simple observation.
Observation: Let $\mathcal{P}$ be any I/O optimal scheme for pebbling $P_r(n)$. Suppose $\mathcal{P}$ uses $f(n)$ blue pebbles. Then $In(P_r(n)) + 2f(n)$ is a lower bound on the number of I/Os for any I/O-optimal scheme for pebbling $P_r(n)$ where $In(P_r(n))$ is the number of input vertices in $V_r(n)$.

This is straightforward because in any I/O optimal pebbling scheme if a blue pebble is placed on a vertex then later a red pebble must be placed on this vertex using the rule that a red pebble can be placed on a blue pebble. If this is not the case, placing the blue pebble is redundant and we have a better pebbling scheme that simply does not place the blue pebble.

The Blue Pebble strategy for proving lower bounds in pebbling a graph G simply establishes a lower bound on the number of blue pebbles placed in any I/O optimal pebbling scheme. The overall lower bound for G is obtained through lower bounds for smaller subgraphs (not necessarily disjoint) and combining these lower bounds.

Theorem 4. *Let $G = (V, E)$ be any layered graph. Suppose that we have q subgraphs $H_1, H_2, \ldots H_q$ of $V^* = G - In(G)$ with the following properties:*

(i) In any complete pebbling of G, each H_i must have at least b blue pebbled vertices
(ii) No $v \in V$ belongs to more than l different H_i's.

Then, in any complete pebbling of G at least $q * b/l$ vertices of $\bigcup_i H_i$ are pebbled with blue pebbles.

Proof. Let S_i denote the set of blue pebbled vertices in the subgraph H_i. Then the set of blue vertices in $\bigcup_i H_i$ is $S = \bigcup_i S_i$. By assumption $\forall\, i\, |S_i| \geq b$. Consider the set $A = \{(v,i) \,|\, v \in S_i, 1 \leq i \leq q\}$. Then $|A| \geq q \times b$. For a vertex u denote by A_u the subset of A of pairs where the first component is u, that is, $A_u = \{(u,i) | 1 \leq i \leq q\}$. Then if $u \neq u'$, A_u and $A_{u'}$ are trivially disjoint. Also, by assumption (ii) for each u, $|A_u| \leq l$. Noticing that $A = \bigcup_{u \in S} A_u$, it then follows that $|S| \geq |A|/l = qb/l$.

To make use of the Blue Pebble strategy, one needs to identify an appropriate family of sub-graphs and establish a lower bound on number of blue pebbles on each of these sub-graphs. Naturally, the choice of the subgraphs can be driven by the ability to establish a lower bound on number of blue pebbled vertices in these subgraphs.

4.3 A lower bound for pebbling $P_r(n)$

To obtain a lower bound on number of blue pebbles in a complete pebbling of $P_r(n)$ we first establish the following lemmas:

Lemma 3. *Consider any complete pebbling of $P_r(n)$ with S red pebbles and let $P_r(h)$ be any r-pyramid of height h in $P_r(n)$. Then $P_r(h)$ has at least $(r-1)h + 1 - S$ blue pebbled vertices.*

Proof. Using the argument of Lemma 2, we have at least $(r-1)h+1$ pebbles on the graph $P_r(n)$ when the last path from the leaf vertex of $P_r(n)$ to the root is blocked. Since there are only S red pebbles in total, it follows that at least $(r-1)h+1-S$ of the vertices in $P_r(h)$ have blue pebbles at this time.

We now use the Blue Pebble strategy to establish a lower bound for pebbling $P_r(n)$ with S red pebbles and unlimited number of blue pebbles. We choose for our subgraphs H_i, all r-pyramids of height h in $P_r(n)$. There is one such pyramid with root at each of the vertices at level $n-h$ and above. Hence there are $q = (r-1)(n-h+1)(n-h)/2 + (n-h+1)$ such r-pyramids. From Lemma 1 it follows that in any complete pebbling of $P_r(n)$, each such r-pyramid of height h must have at least $b = (r-1)h + 1 - S$ blue pebbles. Notice that no vertex in $P_r(n)$ is shared by more than $l = |H_i| = (r-1)(h+1)h/2 + (h+1)$ different subgraphs. It then follows that the number of blue pebbles in complete pebbling of $P_r(n)$ is at least $qb/l = q * [(r-1)h - (S-1)]/[(r-1)(h+1)h/2 + (h+1)]$. Choosing, $(r-1)h = 2(S-1)$, this gives us $qb/l = q * (S-1)/S * (h+1) = q * (S-1)/[S * (2(S-1)/(r-1)+1)]$. This is roughly $q(r-1)/2S$ which is roughly $|V|(r-1)/2S$ if $n >> S$. Hence the total number of I/O operations is bounded below by roughly $|V|(r-1)/S$.

5 Remarks and Conclusion

We presented an I/O efficient and computation optimal scheme for pebbling an r-pyramid. We also presented a new technique for proving lower bounds in pebbling and used it to prove improved lower bounds on I/O for pebbling r-pyramids. There is a gap of a factor of (roughly) 2 between the upper bound and lower bound presented for pebbling the r-pyramids. It will be nice to close this gap one way or the other. The pebbling scheme presented here does not use any "re-pebbling". We conjecture, that this is an I/O and (obviously simultaneously) computation optimal scheme for $P_r(n)$. For pebbling schemes that do not use re-pebbling, a better lower bound on the number of I/Os needed to pebble a 2-pyramid of height n has been established by the authors (manuscript available upon request). However, the technique used there does not immediately help to improve lower bounds on the number of I/Os for pebbling r-pyramids for $r > 2$ even when re-pebbling is not allowed. Finally, it is worth noting that for general layered graphs re-pebbling can reduce the number of I/Os. However, our conjecture also implies that this is not the case for r-pyramids.

References

1. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco, CA (2007)
2. Kumar, V., Sameh, A., Grama, A., Karypis, G.: Architecture, algorithms and applications for future generation supercomputers. In: FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, Washington, DC, USA, IEEE Computer Society (1996) 346
3. Savage, J.E.: Extending the Hong-Kung model to memory hierarchies. In Du, D.Z., Li, M., eds.: Computing and Combinatorics, Springer-Verlag, Lecture Notes in Computer Science (1995) 270–281
4. Savage, J.E.: Models of Computation: Exploring the Power of Computing. Addison Wesley, Reading, Massachusetts (1998)
5. Hong, J.W., Kung, H.T.: I/O complexity: The red-blue pebble game. In: Proc. 13th Ann. ACM Symp. on Theory of Computing. (1981) 326–333
6. Kwok, Y.: Mathematical Models of Financial Derivatives. Springer-Verlag, Singapore (1998)
7. Cox, J.C., Ross, S.A., Rubinstein, M.: Option pricing: A simplified approach. Journal of Financial Economics **7**(3) (September 1979) 229–263
8. Savage, J.E., Zubair, M.: Cache-optimal algorithms for option pricing. ACM Trans. Math. Softw. **37**(1) (2010) 1–30
9. Savage, J.E., Zubair, M.: Evaluating multicore algorithms on the unified memory model. Scientific Programming **17**(4) (2009) 295–308
10. Cook, S.A.: An observation on storage-time trade off. J. Comp. Systems Sci **9** (1974) 308–316

Appendix - The S-span of a 2-pyramid

The basic intuition is that the S-span is obtained by placing the S pebbles on contiguous nodes at the same level and then pebbling all possible nodes from this placement. The number of such nodes is $S + (S-1) + \ldots 1$ or $S(S+1)/2$ (this includes S nodes where the pebbles were originally placed). We provide a proof that this intuition is indeed correct.

Lemma 4. *The S-span of a 2-pyramid is at least $S(S+1)/2$.*

Proof. We can place all S pebbles contiguously on a single level and pebble $S(S+1)/2$ nodes by moving the pebbles up by one level from left to right (discarding the rightmost pebble) and then repeating this at the next level. This scheme pebbles $S(S+1)/2$ nodes. Hence the S-span for the 2-pyramid is at least $S(S+1)/2$.

We will next establish that for any placement X of S pebbles on the 2-pyramid, no more than a total of $S(S+1)/2$ nodes can be pebbled. We do so by first defining a function, $pp(X)$, that upper bounds the maximum number of nodes that can be possibly pebbled from a placement X of S pebbles. We then show that $pp(X) \leq S(S+1)/2$ for any placement X with S pebbles. The basic idea behind the definition is that if the maximum number of nodes that can be possibly pebbled at a level i is k_i then the maximum number of new nodes that can be possibly pebbled level $i+1$ is at most $(k_i - 1)$ (except when k_i is zero in which case this is zero).

Definition 3. *Let X be any placement of S pebbles. Let l denote the lowest level on which there is at least one pebble in X. Let h be the highest such level. Let $m = h - l + 1$ and let $s_i \geq 0$ denote the number of pebbles on the ith level starting from level l (i.e. s_1 is the number of pebbles on level l, s_2 on level $l+1$ $\ldots s_m$ on level $l + m - 1 = h$). Then, $pp(X) = \Sigma_{i=1}^{i=m} max_i + (max_m - 1)(max_m)/2$ where max_i is defined recursively as below:*

$max_1 = s_1$
$max_i = s_i + max_{i-1} - 1$ *if* $1 < i \leq m$ *and* $max_{i-1} > 0$
$max_i = s_i$ *if* $1 < i \leq m$ *and* $max_{i-1} = 0$

It is easy to observe that $pp(X)$ is an upper bound on the number of nodes that can be possibly pebbled by *any* pebbling scheme starting with placement X.

Lemma 5. *For any placement X of S pebbles $pp(X) \leq S(S+1)/2$.*

Proof. We first consider the case where all the S pebbles are placed on a single level (say level 1). Then no more than $S-1$ nodes can be possibly pebbled at level 2, consequently, no more than $S-2$ nodes at level 3 and in general no more than $S-i$ at level $i+1$. It then follows that $pp(X) \leq S+(S-1)+\ldots 1 = S(S+1)/2$.

If the maximum value of $pp(X)$ is obtained by placing all the pebbles on one level we have nothing further to prove. Else, let us consider a placement X of pebbles that maximizes $pp(X)$. By our assumption, X places at least one pebble on more than one levels. Among all placements that maximize $pp(X)$, let us consider the one that has the minimum number of levels between the lowest and the highest levels with non-zero pebbles.

As in Definition 3 let m denote the number of levels between the lowest and highest levels (both included) with non-zero pebbles. Let us label the levels as $1, 2 \ldots m$ with 1 being the lowest level. Let s_i denote the number of pebbles on the ith level in the placement S. Note that, while $s_1, s_m > 0$, some of the other s_is can be zero and also that $\Sigma_i s_i = s$. Let us now consider the value $pp(X) = \Sigma_{i=1}^{i=m} max_i + (max_m - 1)max_m/2$. We contend that by choice of X, none of the max_is is zero and hence for all $1 < i \leq m$ $max_i = s_i + max_{i-1} - 1$. If this is not true then consider the lowest j where $max_j = 0$. Then $s_j = 0$ and $s_{j-1} = 1$. Consider a new placement X' of S pebbles which is identical to X except that all the pebbles below level j are moved one level up. Then $pp(X') = pp(X)$ but X' has fewer levels contradicting our assumption. We now show that $pp(X) \leq S(S+1)/2$.

Expanding out the definition of max_i we get,

$$
\begin{aligned}
&max_1 = s_1\\
&max_2 = s_2 + s_1 - 1\\
&max_3 = s_3 + s_2 + s_1 - 2\\
&\vdots\\
&max_m = s_m + s_2 + \ldots s_1 - (m-1) = (s - (m-1))
\end{aligned}
$$

Hence

$$
\begin{aligned}
&pp(X) = \Sigma_{i=1}^{i=m} max_i + (max_m - 1)max_m/2\\
&= ms_1 + (m-1)s_2 + \ldots 1.s_m - m(m-1)/2 + (S-m)(S-(m-1))/2\\
&= m(\Sigma_{i=1}^{i=m} s_i) - \Sigma_{i=2}^{i=m}(i-1)s_i - m(m-1)/2 + (S-m)(S-(m-1))/2\\
&\leq mS - m(m-1)/2 + (S-m)(S-(m-1))/2\\
&= ms - m(m-1)/2 + (S^2 - (2m-1)S + m(m-1))/2 = S(S+1)/2.
\end{aligned}
$$

Theorem 5. *The S-span of a 2-pyramid is $S(S+1)/2$.*