Summary on the (black) pebble game

Red-Blue Pebble Game for I/Os

Hong-Kung Lower Bound Method

Tight Lower Bound for Matrix Product

Extensions and Performance Bounds
Outline

Summary on the (black) pebble game

Red-Blue Pebble Game for I/Os

Hong-Kung Lower Bound Method

Tight Lower Bound for Matrix Product

Extensions and Performance Bounds
Input: Directed Acyclic Graph (≡ computation)

Rules:

▶ A pebble may be removed from a vertex at any time.
▶ A pebble may be placed on a source node at any time.
▶ If all predecessors of an unpebbled vertex \( v \) are pebbled, a pebble may be placed on \( v \).

Objective: put a pebble on each target (not necessary simultaneously) using a minimum number of pebbles

Number of pebbles:

▶ Number of registers in a processor
▶ Size of the (fast) memory (together with a large/slow disk)
Results:
- Hard to find optimal pebbling scheme for general DAGs (NP-hard without recomputation, PSPACE-hard otherwise)
- Recursive formula for trees

Space-Time Tradeoffs:
- Definition of flow and independent function
- \((\alpha, n, m, p)\)-independent function: \([\alpha(S + 1)]T \geq mp/4\)
- Product of two \(N \times N\) matrices:
  \[(S + 1)T \geq N^3/4\]
  (bound reached by the standard algorithm)
Outline

Summary on the (black) pebble game

Red-Blue Pebble Game for I/Os

Hong-Kung Lower Bound Method

Tight Lower Bound for Matrix Product

Extensions and Performance Bounds
(Black) Pebble game: limit the memory footprint

But usually:

- Memory size fixed
- Possible to write temporary data to the slower storage (disk)
- Data movements take time (Input/Output, or I/O)

NB: same study for any two-memory system:

- (fast, bounded) memory and (slow, large) disk
- (fast, bounded) cache and (slow, large) memory
- (fast, bounded) L1 cache and (slow, large) L2 cache
Red-Blue pebble game (Hong and Kung, 1981)

Two types of pebbles:
- **Red pebbles**: limited number $S$ (slots in fast memory)
- **Blue pebbles**: unlimited number, only for storage (disk)

Rules:
1. A red pebble may be placed on a vertex that has a blue pebble.
2. A blue pebble may be placed on a vertex that has a red pebble.
3. If all predecessors of a vertex $v$ have a red pebble, a red pebble may be placed on $v$.
4. A pebble (red or blue) may be removed at any time.
5. No more than $S$ red pebbles may be used at any time.
6. A blue pebble can be placed on an input vertex at any time.

Objective: put a red pebble on each target (not necessary simultaneously) using a minimum rules 1 and 2 (I/O operations)
The pebble game is illustrated in Fig. 10.1 by pebbling the FFT graph $F(3)$ on $n = 2^3$ inputs. Input vertices are on the bottom; edges are directed upward. Four pebbles are shown on the graph when pebbling the leftmost output.

Input variables are held in an auxiliary random-access machine so that it can access them in arbitrary order, a condition not imposed on pebble games. It follows that inputs to a pebble game can be fetched in advance, since the times at which they are needed are data-independent.

Second, lower bounds on the exchange of space for time with branching programs are harder to obtain due to their increased flexibility. Third, straight-line programs are used in many problems, such as integer multiplication, convolution, matrix multiplication, and discrete Fourier transform, and the pebble game gives the relevant lower bounds. For other problems, such as sorting and merging, the branching program model is the model for choices in the problem are typically solved with branching programs. We expand upon this in Section 10.9.1.

**10.1.2 Playing the Pebble Game**

The pebble game is illustrated in Fig. 10.1 by pebbling the FFT graph $F(3)$ with eight inputs and 24 non-input vertices. This graph has the property that the set of paths from input vertices to an output vertex forms a complete balanced binary tree. (See Fig. 10.2.) It follows that we can pebble the FFT graph by pebbling each of the trees. Since two of the eight outputs share the same tree at the next lower level, we can pebble two outputs at the same time.

Binary trees form an important class of graphs. A complete balanced binary tree of depth $4i$ is illustrated in Fig. 10.2. (The depth of a directed tree is the number of edges on the longest path from an input vertex to the output (or root) vertex.) This tree has $16$ input vertices and one output vertex. A complete balanced binary tree of depth $0$, $T(0)$, consists of a single vertex. A complete balanced binary tree of depth $d > 0$, $T(d)$, consists of an output vertex and two copies of $T(d-1)$ whose root vertices each have one edge directed from them to the root vertex of the full tree. Thus in Fig. 10.2 the complete balanced binary tree of depth four $T(4)$ is constructed of two copies of $T(3)$, which themselves are each constructed of two copies of $T(2)$, and so on. It follows by straight-forward induction that a complete balanced binary tree of depth $d$ has $2^d$ inputs and $2^d + 1 - 1$ vertices. (See Problem 10.8.)
Outline

Summary on the (black) pebble game

Red-Blue Pebble Game for I/Os

Hong-Kung Lower Bound Method

Tight Lower Bound for Matrix Product

Extensions and Performance Bounds
Hong-Kung Lower Bound Method

**Objective:** Given a number of red pebbles, give a lower bound on the number of I/Os for any pebbling scheme of a graph.

**Definition (span).**

Given a DAG $G$, its $S$-span $\rho(S, G)$, is the maximum number of vertices of $G$ that can be pebbled with $S$ pebbles in the **black** pebble game without the initialization rule, maximized over all initial placements of the $S$ pebbles on $G$.

Rationale: with large $\rho(S, G)$, you can compute a lot of $G$ with $S$ pebbles (for a given starting point).

Find $\rho(2, G)$
Span of the matrix product

**Definition (span).**

Given a DAG $G$, its $S$-span $\rho(S, G)$, is the maximum number of vertices of $G$ that can be pebbled with $S$ pebbles in the **black** pebble game without the initialization rule, maximized over all initial placements of the $S$ pebbles on $G$.

**Theorem.**

For every DAG $G$ to compute the product of two $N \times N$ matrices in a regular manner (performing the $N^3$ products), the span is bounded by $\rho(S, G) \leq 2S\sqrt{S}$ for $S \leq N^2$.

**Lemma.**

Let $T$ be a binary (in-)tree representing a computation, with $p$ **black** pebbles on some vertices and an unlimited number of available pebbles. At most $p - 1$ vertices can be pebbled in the tree without pebbling new inputs.

(proofs on the board, available in the notes)
**From Span to I/O Lower Bound**

\[ T_{I/O}(S, G) : \text{number of I/O steps (red } \leftrightarrow \text{ blue)} \]

**Theorem (Hong & Kung, 1981).**

For every pebbling scheme \( S \) of a DAG \( G = (V, E) \) in the red-blue pebble-game using at most \( S \) red pebbles, the number of I/O steps satisfies the following lower bound:

\[
\left\lceil \frac{T_{I/O}(S, G)}{S} \right\rceil \rho(2S, G) \geq |V| - |\text{Inputs}(G)|
\]

Recall that for matrix product \( \rho(S, G) \leq 2S \sqrt{S} \), hence:

\[
T_{I/O} \geq \frac{N^3 - N^2}{4 \sqrt{2} S} = \Theta \left( \frac{N^3}{\sqrt{S}} \right)
\]
Outline

Summary on the (black) pebble game

Red-Blue Pebble Game for I/Os

Hong-Kung Lower Bound Method

Tight Lower Bound for Matrix Product

Extensions and Performance Bounds
Tight Lower Bound for Matrix Product

\[ b \leftarrow \sqrt{\frac{M}{3}} \]

\textbf{for} \ i = 0, \rightarrow n/b - 1 \textbf{ do}

\hspace{1cm} \textbf{for} \ j = 0, \rightarrow n/b - 1 \textbf{ do}

\hspace{2cm} \textbf{for} \ k = 0, \rightarrow n/b - 1 \textbf{ do}

\hspace{3cm} \text{Simple-Matrix-Multiply}(n, C_{i,j}^b, A_{i,k}^b, B_{k,j}^b)

- Blocked algorithm: \( 3\sqrt{3}N^3 / \sqrt{M} \)
- Previous bound on I/Os \( \sim N^3 / 4\sqrt{2M} \)
- Many improvements needed to close the gap
- Presented here for \( C \leftarrow C + AB \), square matrices

New operation: Fused Multiply Add

- Perform \( c \leftarrow c + a \times b \) in a single step
- No temporary storage needed (3 inputs, 1 output)
Theorem.

Any algorithm for the matrix product can be transformed into using only FMA without increasing the required memory or the number of I/Os.

Transformation:

- If some $c_{i,j,k}$ is computed while $c_{i,j}$ is not in memory, insert a read before the multiplication.
- Replace the multiplication by a FMA.
- Remove the read that must occur before the addition $c_{i,j} ← c_{i,j} + c_{i,j,k}$, remove the addition.
- Transform occurrences of $c_{i,j,k}$ into $c_{i,j}$.
- If $c_{i,j,k}$ and $c_{i,j}$ were both in memory in some time-interval, remove operations with $c_{i,j,k}$ in this interval.
Step 2: Concentrate on Read Operations

Theorem (Irony, Toledo, Tiskin, 2008).
Using $N_A$ elements of $A$, $N_B$ elements of $B$ and $N_C$ elements of $C$, we can perform at most $\sqrt{N_A N_B N_C}$ distinct FMAs.

\[ |V|^2 \leq |V_1| \cdot |V_2| \cdot |V_3|, \]

Theorem (Discrete Loomis-Whitney Inequality).
Let $V$ be a finite subset of $\mathbb{Z}^D$ and $V_1, V_2, V_3$ denotes the orthogonal projections of $V$ on each coordinate planes, we have
Step 3: Use Phases of $R$ Reads ($\neq M$)

**Theorem.**

During a phase with $R$ reads with memory $M$, the number of FMAs is bounded by

$$F_{M+R} \leq \left(\frac{1}{3}(M + R)\right)^{3/2}$$

Number $F_{M+R}$ of FMAs constrained by:

$$\begin{cases} 
F_{M+R} \leq \sqrt{N_A N_B N_C} \\
0 \leq N_A, N_B, N_C \\
N_A + N_B + N_C \leq M + R
\end{cases}$$

Using Lagrange multipliers, maximal value obtained when $N_A = N_B = N_C$
Step 4: Choose $R$ and add write operations

in one phase, nb of computations: \( F_{M+R} \leq \left( \frac{1}{3}(M + R) \right)^{3/2} \)

Total volume of reads:
\[
V_{\text{read}} \geq \left\lfloor \frac{N^3}{F_{M+R}} \right\rfloor \times R \geq \left( \frac{N^3}{F_{M+R}} - 1 \right) \times R
\]

Valid for all values of $R$, maximized when $R = 2M$:
\[
V_{\text{read}} \geq 2N^3 / \sqrt{M} - 2M
\]

Each element of $C$ written at least once: \( V_{\text{write}} \geq N^2 \)

**Theorem.**

The total volume of I/Os is bounded by:
\[
V_{I/O} \geq \frac{2N^3}{\sqrt{M}} + N^2 - 2M
\]
Asymptotically Optimal Algorithm

- Partition $C$ into blocks of size $(\sqrt{M} - 1) \times (\sqrt{M} - 1)$
- Load block-columns of $A$ of size $(\sqrt{M} - 1) \times 1$
- Load block-rows of $B$ of size $1 \times (\sqrt{M} - 1)$
- Compute all rank-1 updates

Volume of I/O:
- elements of $C$ read and written only once: $2N^2$
- Number of iterations: $N \times \left( \frac{N}{(\sqrt{M} - 1)} \right)^2$
- Read of $A$ and $B$ per iteration: $2(\sqrt{M} - 1)$
- Total:
  \[
  \frac{2N^3}{\sqrt{M} - 1} + N^2 \approx \frac{2N^3}{\sqrt{M}} + N^2
  \]
Outline

Summary on the (black) pebble game

Red-Blue Pebble Game for I/Os

Hong-Kung Lower Bound Method

Tight Lower Bound for Matrix Product

Extensions and Performance Bounds
Extension to the Memory Hierarchy Pebble Game

Generalization for a memory/cache hierarchy of \( L \) levels:

- **Level 1**: fastest/most limited memory
- **Level \( L \)**: slow/unlimited memory
- \( p_i \) available pebbles at level \( i < L \):
- Computation steps only with level-1 pebbles
- Initialization only with level-L pebbles
- Input from level \( i \): if level-\( i \) pebble, put level-(\( i - 1 \)) pebble
- Output to level \( i \): if level-(\( i - 1 \)) pebble, put level-\( i \) pebble

Cumulated number of pebbles up to level \( i \): \( s_i = \sum_{i=1}^{I} p_i \).

Number of inputs from/outputs to level \( i \):

\[
T_i = \begin{cases} 
\Theta\left(\frac{N^3}{\sqrt{s_{i-1}}}\right) & \text{if } s_{i-1} < 3N^2 \\
\Theta(N^2) & \text{otherwise}
\end{cases}
\]
Recent Developments of Pebble Games

Restrict to pebbling \textbf{without recomputation}:

- Add white pebbles with red pebbles when computing
- White pebbles stay on vertices
- No computation possible if white pebble already present
- All nodes must be white-pebbled at the end

This restriction increases the number of red pebbles and I/Os by at most a $\log^{3/2} n$ factor

Towards \textbf{automatic derivation} of lower bounds:

- Extend bounds for composite graphs
- Use special min-cuts instead of span

Parallel Red-Blue-White Pebble Game (cf. memory hierarchies)

Still an inspiring model!
Why so much fuss about matrix product?

BLAS: Basic Linear Algebra Subprograms

- Introduced in the 80s as a standard for LA computations
- Written first in FORTRAN
- Library provided by the vendor to ease use of new machines
- Organized by levels:
  - Level 1: vector/vector operations ($x \cdot y$)
  - Level 2: vector/matrix ($Ax$)
  - Level 3: matrix/matrix ($AB^T$, blocked algorithms)
- Implementations:
  - Vendors (MKL from Intel, CuBLAS from NVidia, etc.)
  - Automatic Tuning: ATLAS
  - GotoBLAS
- Matrix product: still a large share of LA computations
Partition $n$ with blocksize $n_c$.

Partition $k$ with blocksize $k_c$.

Partition $m$ with blocksize $m_c$.

Matrix partition is reused in L3 cache.

Matrix partition is reused in L2 cache.

Matrix partition is reused in L1 cache.

Matrix partition is reused in registers.

Fig. 2. Diagram of Goto's Algorithm implemented in BLIS.
Computation ceilings:
- Theoretical peak,
- Matrix-Matrix product (DGEMM)
- LINPACK (Top 500 ranking)

Bandwidth ceilings:
- Cache bandwidth
- Memory bandwidth
- NUMA (Non Uniform Memory Access)