CR10: Data Aware Algorithms Autumn 2019

Part 1. Pebble-game models

Lecturer:Loris Marchal

1 Introduction: on the importance of data movement

Fast memory, that is the immediate storage space available for the computation, has always been a scarce resource in computers. However, the amount of available memory has largely increased in the previous decade, so that one could imagine that this limitation is about to vanish. However, when looking at the evolution of memory bandwidth and of processor speed, the future is not so bright: it has been acknowledged that the processing speed of a micro-processor (measured in floating point operations per seconds) has increased annually by 59% on average from 1988 to 2004, whereas the memory bandwidth (measured in byte per second) has increased annually only by 26%, and the memory latency (measured in seconds) only by 5% on the same period [3]. More recent figures (see Figure 1) suggest a similar behavior. This means that the *time to process the data* is reduced at a much higher pace to the *time to move the data* from the memory to the processor. This problem is known as the “memory wall” in micro-processor design, and has been alleviated by the introduction of cache memories: the large but slow memory is assisted with a small but fast *cache*.

Time per flop: 59%

Data movement	Bandwidth	Latency
Network	26%	15%
DRAM	23%	5%

Table 1: Annual improvements (2004 figures based on data on the period 1988-2002)

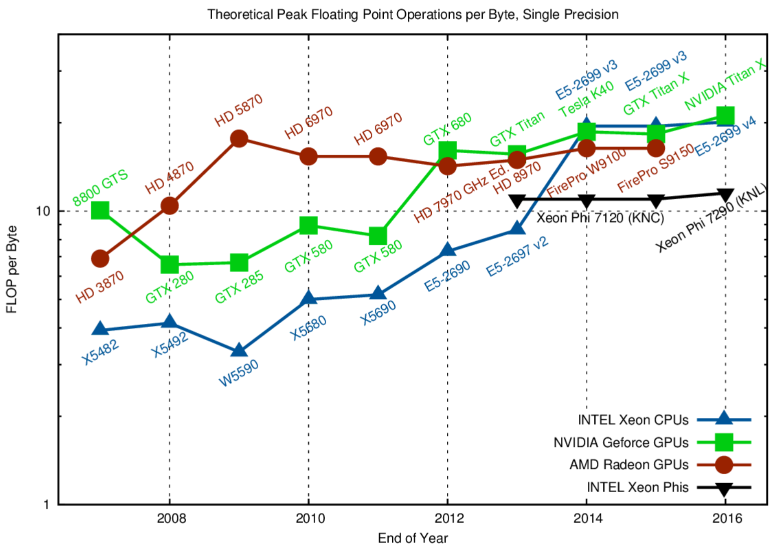

Figure 1: Flop per byte moved ratio for a variety of processing units (from `http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`)

Thus, to overcome the decrease of the memory bandwidth/computing speed ratio, chips manufacturer have introduced a hierarchy of caches. This makes the performance of an application very sensitive to data locality: only the applications that exhibit a large amount of temporal locality (data reuse within a short time interval) or spatial locality (successive use of data from close storage locations) may take advantage of the speed of the processor. Furthermore, the cost of data movement is expected to become dominant also in terms of energy [11, 7]. Thus, avoiding data movement and favoring data reuse are crucial both to obtain good performance and to limit energy consumption. Numerous works have considered this problem. In this document, we concentrate on some theoretical studies among them, namely how to design algorithms and schedules that have good and guaranteed performance in a memory-limited computing environment. In particular, we will not cover the numerous studies in hardware design or compilation techniques that target the same problem, because they are mainly orthogonal to the present approach.

2 Algorithm design and memory usage

Let us consider here a very simple problem, the matrix product, to show how the design of the algorithm can influence the memory usage and the amount of data movement[1]. We consider two square $n \times n$ matrices A and B, and we compute their product $C = AB$.

Algorithm 1: SIMPLE-MATRIX-MULTIPLY(n, C, A, B)

```
for $i = 0 \to n-1$ do
    for $j = 0 \to n-1$ do
        $C_{i,j} = 0$
        for $k = 0 \to n-1$ do
            $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```

We consider that this algorithm is executed on a simple computer, consisting of a processing unit with a fast memory of size M. In addition to this limited memory, a large but slow storage space is available. In the following, we assume that this space is the disk and has unlimited available storage space.[2] Our objective is to minimize the data movement between memory and disk, also known as the volume of I/O (input/output), that is the number of A, B and C elements that are loaded from the disk to the memory, or written back from the memory to the disk. We assume that the memory is limited, and cannot store more than half a matrix, i.e., $M < n^2/2$.

In Algorithm 1, all B elements are accessed during one iteration of the outer loop. Thus, as the memory cannot hold more than one half of B, at least $n^2/2$ elements must be read. For the n iterations, this leads to $n^3/2$ read operations. This is huge as it is the same order of magnitude of the number of computations (n^3).

Fortunately, it is possible to improve the I/O behavior of the matrix product by changing the algorithm. We set $b = \sqrt{M/3}$ and assume that n is a multiple of b. We consider the blocked version of the matrix product, with block size b, as detailed in Algorithm 2. In this algorithm $C^b_{i,j}$ denotes the block of size b at position (i, j) (all elements $C_{k,l}$ such that $ib \leq k \leq (i+1)b - 1$ and $jb \leq l \leq (j+1)b - 1$).

Each iteration of the inner loop of the blocked algorithm must access 3 blocks of size b^2. Thanks to the choice of b, this fits in the memory, and thus, each of these $3b^2$ elements are read and written at most once. This leads to at most $2M$ data movements. Since there are $(n/b)^3$ iteration of the inner loop, the volume of I/O of the blocked algorithm is $O((n/b)^3 \times 2M) = O(n^3/\sqrt{M})$.

Changing the algorithm has allowed us to reduce the amount of data movements. The question is now: can we do even better? Using a pebble game model introduced below, we will be able to answer this question.

[1]A large part of this section is adapted from [13].

[2]Note that this study detailed for the case "main memory vs. disk" may well apply to other pairs of storage such as "fast small cache vs. large slower memory".

Algorithm 2: BLOCKED-MATRIX-MULTIPLY(n, C, A, B)

```
$b \leftarrow \sqrt{M/3}$
for $i = 0, \rightarrow n/b - 1$ do
    for $j = 0, \rightarrow n/b - 1$ do
        for $k = 0, \rightarrow n/b - 1$ do
            Simple-Matrix-Multiply$(n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j})$
```

3 (Black) pebble game for memory minimization

3.1 Definition

We present here the first theoretical model that was proposed to study the space complexity of programs. This model, based on a pebble game, was originally used to study register allocation. Registers are the first level of storage, the fastest one, but also a scarce resource. When allocating registers to instructions, it is thus crucial to use them with caution and not to waste them. The objective is thus to find the minimum amount of registers that is necessary for the correct execution of the program. Minimizing the number of registers is similar to minimizing the amount of memory. For the sake of consistency, we present the pebble game as a problem of memory size rather than register number minimization. This does not change the proposed model nor the results, but allows us to present all results of this document with the same model and formalism.

The pebble-game was introduced by Sethi [9] to study the space complexity of "**straight-line**" programs, that is, programs whose control flow does not depend on the input data. A straight-line program is modeled as a directed acyclic graph (DAG): a vertex represents an instruction, and an arc between two vertices $i \rightarrow j$ means that the results of the vertex i is used for the computation of j.

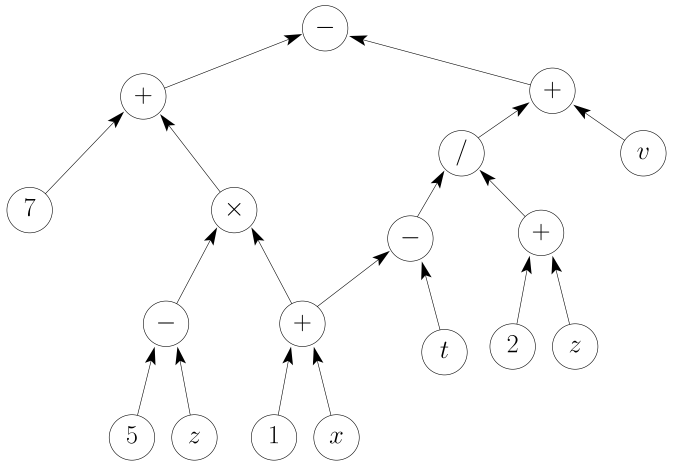

Figure 2: Graph corresponding to the computation of the expression $7 + (5 - z) \times (1 + x) - (1 + x - t)/(2 + z) + v$

When processing a vertex, all its inputs (as well as the result) must be loaded in memory, and the goal is to execute the program using the smallest amount of memory. Memory slots are modeled as pebbles, and executing the program is equivalent to playing a game on the graph with the following rules:

- (PG1) A pebble may be removed from a vertex at any time.
- (PG2) A pebble may be placed on a source node at any time.

- (PG3) If all predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.

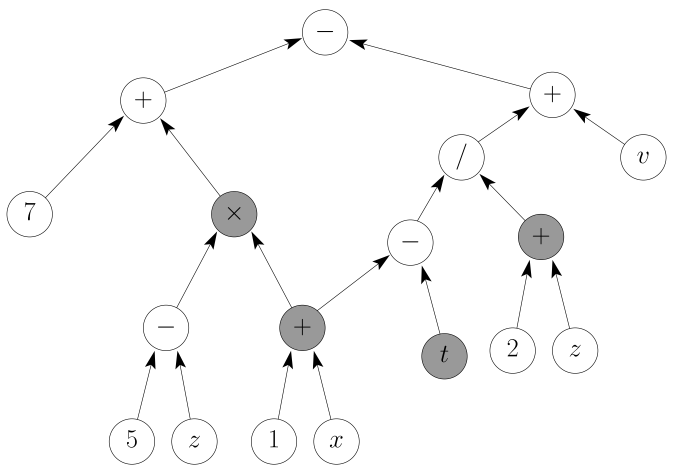

Figure 3: Playing the (black) pebble game on the graph of Figure 2. Dark nodes are the ones currently pebbled, meaning that four values are now in memory: $(5 - z) \times (1 + x)$, $1 + x$, t and $2 + z$.

The goal of the game is to put a pebble on each output vertex at some point of the computation, and to minimize the total number of pebbles needed to reach this goal. In this game, pebbling a node corresponds to loading an input in memory (rule PG2) or computing a particular vertex (rule PG3). From a winning strategy of this game, it is thus straightforward to build a solution to the original memory allocation problem.

3.2 Variants and hardness

Note that the game does not ensure that each vertex will be pebbled only once. Actually, in some specific graphs, it may be beneficial to pebble several times a vertex, that is, to compute several times the same values, to save a pebble needed to store its value. A variation of the game, named the *Progressive Pebble Game*, forbids any recomputation, and thus models the objective of minimizing the amount of memory without any increase in the computational complexity. In this latter model, the problem of determining whether a directed acyclic graph can be processed with a given number of pebbles has been shown NP-hard by Sethi [9]. The more general problem allowing recomputation is even more difficult, as it has later been proved PSPACE-complete by Gilbert, Lengaeur and Tarjan [2]. Another variant of the game slightly changes rule PG3 and allows to *shift* a pebble to an unpebbled vertex if all its predecessors are pebbled. Van Emde Boas [14] shows that it can at most decrease the number of pebbles required to pebble the graph by one, but in the worst case the saving is obtained at the price of squaring the number of moves needed in the game.

Another variant, called the black-white pebble game has been proposed to model non deterministic execution [5, 6], where putting a white pebble on a node corresponds to guessing its value; a guessed vertex has to be actually computed later to check the value of the guess.

3.3 Special case: tree-shaped graphs

A simpler class of programs consists in trees rather than general graphs: for example, arithmetic expressions are usually described by in-trees (rooted directed trees with all edges oriented towards the root), unlike the one of Figure 2 which uses a common sub-expression twice.

3.3.1 Complete binary trees

We first consider **Complete binary trees** and establish the following result for the variant when shifting pebbles is allowed:

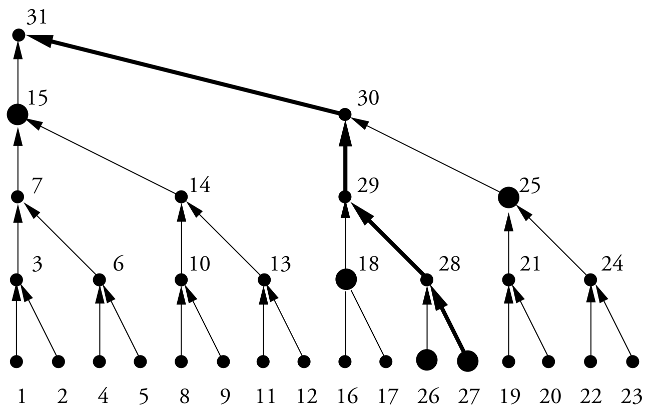

Figure 4: Complete binary tree of depth $k = 4$ and pebbles when the last leaf (27) is pebbled. This figure as well as the followings used to illustrate the pebble games are borrowed from [8].

Theorem 1. *Any pebbling strategy (with or without recomputation) for the complete balanced binary tree $T(k)$ of depth k uses at least $k+1$ pebbles and $2^{k+1}-1$ steps. There is a pebbling strategy that reaches both bounds.*

Proof. For the first lower bound, for a given strategy, consider the last time when it pebbles a leaf, called v. We assume that no vertex on the path P from v to the root is already pebbled (otherwise pebbling v at this time is unnecessary and can be avoided). Since v is the last pebbled leaf, for any other leaf v', there should be a pebble on the path from v' to some node on the path P from v to the root. Minimizing this number is obtained by pebbling all the descendants not in P of nodes of P, which needs k pebbles. Together with the pebble on v, it makes $k+1$ pebbles.

The second lower bound simply derives from the fact that each vertex of the tree should be pebble once.

Finally, the following postorder pebbling strategy reaches both bounds:

1. Pebble the left subtree, leave a pebble on its root
2. Pebble the right subtree, leave a pebble on its root
3. Shift the pebble of one child of the root to the root, remove the other pebble

A tree of depth 0 (single node) can be pebbled with one pebble. Hence, a tree of depth k can be pebbled with $k+1$ pebbles using this strategy. Furthermore, it only pebbles once each vertex. □

Exercise: what changes if shifting pebbles is not allowed?

3.3.2 General trees

We now get back to the variant without shifting pebble. Even in this case, the problem gets much simpler than for general graph: Sethi and Ullman [10] designed an optimal scheme which relies on the following theorem.[3]

[3]When pebble shifting is allowed, the minimum number of pebbles needed to pebble a tree is the Strahler number (as outlined in [1]), which is a measure of a tree's branching complexity that appears in natural science, such as in the analysis of streams in a hydrographical bassin [12] or in biological trees such as animal respiratory and circulatory systems https://en.wikipedia.org/wiki/Strahler_number.

Theorem 2. *An optimal solution for the problem of pebbling an in-tree with the minimum number of pebbles using rules PG1 – PG3 is obtained by a depth-first traversal which orders subtrees by non-increasing values of $P(i)$, where the peak $P(v)$ of the subtree rooted at v is recursively defined by:*

$$P(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf} \\ \max_{i=1\dots k} P(c_i) + i - 1 & \begin{array}{l}\text{where } c_1, \dots c_k \text{ are the children of } v \\ \text{such that } P(c_1) \geq P(c_2) \geq \dots \geq P(c_k)\end{array} \end{cases}$$

The first step to prove this result is to show that depth-first traversals are dominant, i.e., that there exists an optimal pebbling scheme which follows a depth-first traversal. Pebbling a tree using a depth-first traversal adds a constraint on the way a tree is pebbled: consider a tree T and any vertex v whose children are $c_1, \dots c_k$, and assume w.l.o.g that the first pebble that is put in the subtree rooted at v is in the subtree rooted at its leftmost child c_1. Then, in a depth-first traversal, the following pebbles must be put in the subtree rooted at c_1, until c_1 itself holds a pebble (and all pebbles underneath may be removed). Then we are allowed to start pebbling other subtrees rooted at $c_2, \dots, c_k$. These traversals are also called *postorder*, as the root of a subtree is pebbled right after its last child.

To prove that depth-first traversals are dominant, we notice that whenever some pebbles have been put on a subtree rooted at some vertex v, it is always beneficial to completely pebble this subtree (until its root v, which uses a single vertex) before starting pebbling other subtrees.

The second step to prove Theorem 2 is to justify the order in which the subtrees of a given vertex must be processed to give a minimal number of pebbles. This result follows from the observation that after having pebbles $i-1$ subtrees, $i-1$ pebbles should be kept on their respective roots while the i^{th} subtree is being pebbled.

3.4 Space-Time Tradeoffs

3.4.1 Example with the FFT graph

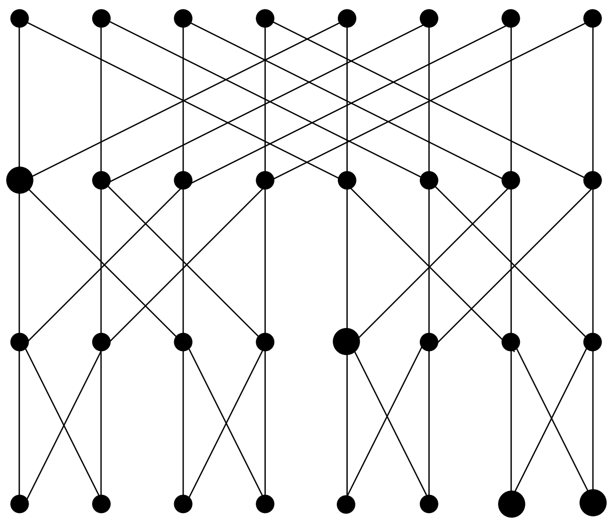

Figure 5: FFT graph with 8 input/output vertices (depth $k = 3$)

The FFT graph (see Figure 5 contains many complete binary trees, hence it inherits some of the previous results: we need at most $k+1$ pebbles to pebble the FFT graph of depth k. A pebbling scheme reaching this number of pebbles is obtained by pebbling each output one after the other. A simple optimization is possible: when two vertices of the third level are pebbled, two outputs (on the fourth level) can be pebbled at once. However, this results in a large number of re-computations: all level-3 and 4 vertices are pebbled once, each level-2 vertex is pebbled twice, and each input (level-1) vertices are pebbled four times. Using a larger number of pebbles allows to reduce the number of steps. For example, with $2n = 2^{k+1}$ pebbles, each level can be pebble one after the other, which allows to reach the minimum number of steps.

3.4.2 Grigoriev's Lower-Bound Method

Definition 1 ($w(u, v)$-flow)**.** *Let f be a function from $\mathcal{A}^n$ to $\mathcal{A}^m$. f has a $w(u, v)$-flow if for all partition (X_0, X_1) of the n inputs and all partition (Y_0, Y_1) of the m outputs, such that $|X_1| = u$ and $|Y_1| = v$, there exists a subfunction h of f obtained by assigning all variables in X_0 to fixed values and discarding output values in Y_0, such that h has at least $|A|^{w(u,v)}$ distinct points in the image of its domain f.*

This definition is illustrated on Figure 6. The idea is to characterize a significant information flow from X_1 to Y_1, even if inputs not in X_1 are fixed and outputs not in Y_1 are discarded.

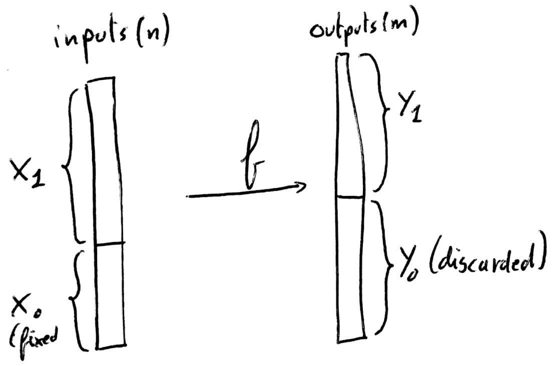

Figure 6: Definition of a flow

A more general property is defined below.

Definition 2 ((α, n, m, p)-independant function)**.** *A function $f : \mathcal{A}^n \mapsto \mathcal{A}^m$ is an (α, n, m, p)-independant function for $\alpha \geq 1$ and $p \leq m$ if it has a $w(u, v)$-flow with $w(u, v) \geq v/\alpha - 1$ for all u, v such that $n - u + v \leq p$.*

The matrix multiplication will be used as a running example in this section. We consider any standard algorithm for multiplying two matrices of size $N \times N$, that is, we exclude Strassen algorithm as well as its variants and concentrate on algorithms performing N^3 multiplications.

Lemma 1. *The matrix multiplications of two matrices of size $N \times N$ is $(1, 2N^2, N^2, N)$-independant.*

Proof. Note that $n = N^2$ and $p = N$ so that $n - u + v \leq p$ with $u = |X_1|$ and $v = |Y_1|$ and is indeed $|X_0| + |Y_1| \leq N$. We consider here the worst case (equality).

We consider the matrix product $C = AB$, a set of inputs X_0 (elements of A and B) and a set of outputs Y_1 (elements of C) such that $|X_0|+|Y_1| = N$. The outputs of Y_1 lie in at most $|Y_1|$ columns of C (denoted by C_1), while the inputs of X_0 lies in at most $|X_0|$ columns of A (some elements may even be in B). Thus, at least $N - |X_0|$ columns of A contains only elements in X_1 (elements that are allowed to vary), we call these columns A_1 (note that $|A_1| \geq N - |X_0| \geq |Y_1| \geq |C_1|$). We fix the entries in B in $\{0, 1\}$ so that AB is a permutation of the columns of A: some of these elements are in X_0 and should be fixed anyway, the others are in X_1 and are also allowed to be fixed in the subfunction h. We design the permutation matrix B so that (some of) the columns of A_1 are sent to columns of C_1. Hence, when varying the inputs of X_1 over R, the outputs of Y_1 can take up to $R^{|Y_1|}$ values, which corresponds to a $|Y_1|$-flow, larger than the $(|Y_1| - 1)$-flow requested by the theorem. □

The following theorem establishes a lower bound on the number of steps based on the existence of the flow. The proof uses the fact that during an interval where b outputs are pebbled, the information coming from the variables corresponding to the initial position of the S pebbles at the beginning of the interval is limited. Hence, if f has a large flow, a lot of inputs must be read during such an interval, which gives a lower bound on the total number of steps.

Theorem 3. *Let $f : \mathcal{A}^n \mapsto \mathcal{A}^m$ be a function with a $w(u, v)$-flow. Any pebbling of any DAG computing f that takes T steps and S space respects:*

$$T \geq \lfloor m/b \rfloor (n - d)$$

for any $b \leq m$ and d the largest integer such that $w(d, b) \leq S$.

Proof. We consider such a function f and a pebbling scheme that computes it with S pebbles and T steps. Let $b \leq m$ an integer.

We divide the pebbling schemes into phases, such that exactly b outputs are pebbled in each phase (except possibly the last phase in which less outputs may be pebbled). We consider such a regular phase. Let Y_1 be the set of outputs pebbled in this phase ($|Y_1| = b$), x_0 the number of inputs pebbled inside the phase (resp. outside the phase). Since f has a $w(u, v)$-flow, there exists an assignment of the x_0 inputs such that the outputs in Y_1 have at least $|\mathcal{A}|^{w(n-x_0,b)}$ different values. If $w(n-x_0, b) > S$, there are more possible outputs in Y_1 that the possible values computed at the beginning of the phase, represented by the (at most) S pebbles located in the graph in the beginning of the phase. As the other x_0 inputs are supposed fixed in the flow, this contradicts the assumption on f. Hence x_1 cannot be larger than d (by definition of d). Thus, the number of inputs pebbled during the phase satisfies $x_0 \geq n - d$.

Since there are $\lfloor m/b \rfloor$ regular phases, the number of times the inputs are pebbled is at least $\lfloor m/b \rfloor (n - d)$ and so is the total number of steps in the pebbling scheme. $\square$

For independant function, choosing b appropriately gives the following corollary:

Corollary 1. *Let $f : \mathcal{A}^n \mapsto \mathcal{A}^m$ be (α, n, m, p)-independant. For every pebbling of every DAG computing f using S pebbles and T steps, we have*

$$\lceil \alpha(S + 1) \rceil T \geq mp/4$$

Proof. By definition f has a $w(u, v)$-flow satisfying $w(u, v) > v/\alpha - 1$ for all u, v with $n - u + v \leq p$. We choose $b = v = \lceil \alpha(S+1) \rceil$, and thus have $v/\alpha - 1 \geq S$. We define d as in the previous theorem (larger integer with $w(d, b) \leq S$). By contradiction, we assume that $(n - d) + b \leq p$. By definition of the flow, we would have $w(d, b) > S$, which contradicts the definition of d. Hence, we have $(n - d) + b > p$, that is $n - d > p - \alpha(S + 1)$.

We notice that $\lfloor m/x \rfloor \geq (m - x + 1)/x$. The bound of the previous theorem now translates to: translates to:

$$T \geq \frac{(m - \lceil \alpha(S + 1) \rceil + 1)}{\lceil \alpha(S + 1) \rceil} (p - \lceil \alpha(S + 1) \rceil)$$

We consider two cases:

- Case 1: $\lceil \alpha(S+1) \rceil \leq p/2$. Then $p - \lceil \alpha(S+1) \rceil \leq p/2$ and $m - \lceil \alpha(S+1) \rceil \leq m/2$ as $m \geq p$, which leads to the desired bound.
- Case 2: $\lceil \alpha(S + 1) \rceil > p/2$, then $\lceil \alpha(S + 1) \rceil T \geq mp/2$ since $T > m$ (all outputs need to be pebbled).

$\square$

We now get back to the multiplication of two $N \times N$ matrices. We proved earlier that it was $(1, 2N^2, N^2, N)$-independant. This leads to the following bound:

Theorem 4. *Every pebbling strategy for any straight-line program computing the multiplication of two $N \times N$ matrices uses a space S and time T respecting the following inequality:*

$$(S + 1)T \geq N^3/4$$

Note that the standard algorithm that computes each $C_{i,j}$ values one after the other by computing the inner (dot) product of two vectors reaches this bound (at least asymptotically): each dot product can be perform with 4 pebbles in time $4N$, and there are N^2 of them to compute, hence $ST = 16N^3$ for this algorithm.

4 Red-Blue pebble game for data transfer minimization

In some cases, the amount of fast storage (i.e., memory) is too limited for the complete execution of a program. In that case, communication are needed to move data from/to a second level of storage (i.e., disk), which is usually larger but slower. Because of the limited bandwidth of the secondary storage, the amount of data transfers, sometimes called Input/Output (or simply I/O) volume, is a crucial parameter for performance, as seen in the introduction. Once again, we present this problem for the pair (main memory,disk), but this may well apply to any pair of storages in the usually deep memory hierarchy going from the registers and fastest caches to the slowest storage systems.

4.1 Definition and examples

While the first pebble game allows to model algorithms under a limited memory, Hong and Kung have proposed another pebble game to tackle the I/O volume minimization in their seminal article [4]. This game uses two types of pebbles (of different colors) and thus is also called the red/blue pebble game to distinguish with the original (black) pebble game. The goal is to distinguish between the main memory storage (represented by red pebbles), which is fast but limited, and the disk storage (represented by blue pebbles), which is unlimited but slow. As in the previous model, a computation is represented by a directed acyclic graph. The red and blue pebbles can be placed on vertices according to the following rules:

- (RB1) A red pebble may be placed on any vertex that has a blue pebble.
- (RB2) A blue pebble may be placed on any vertex that has a red pebble.
- (RB3) If all predecessors of a vertex v have a red pebble, a red pebble may be placed on v.
- (RB4) A pebble (red or blue) may be removed at any time.
- (RB5) A blue pebble can be placed on an input vertex at any time.
- (RB6) No more than S red pebbles may be used at any time.

The goal of the game is to put a red pebble on each output vertex at some point of the computation, and to use the minimum number of RB1/RB2 rules to reach this goal. Red vertices represents values that currently lies in the main memory, after a computation, while blue pebbles represents values that are on disk. A value on disk may be read from disk (rule RB1) and similarly a value in memory can be stored on disk (rule RB2). Finally, we may compute a value if all its inputs are already in memory (rule RB3). The volume of I/O is the total number of moves using rules RB1 or RB2, that is the total number of data movements between memory and disk.

Consider now the FFT graph presented above. With $S = 3$ red pebbles, the graph can be computed by putting each intermediate vertex in slow memory, so the number of I/Os is $(k-1)2^{k+1}$ (inputs only read, outputs only written, intermediate vertices both written and read).

4.2 The Hong-Kung Lower-Bound method

We present here the method proposed by Hong and Kung in their seminal paper [4] to derive lower-bounds on the volume of I/O needed for some computations with limited storage, that is, limited number of red pebbles. We present a slightly simpler version of their result as revisited by Savage [8]. We first define the S-span of a DAG.

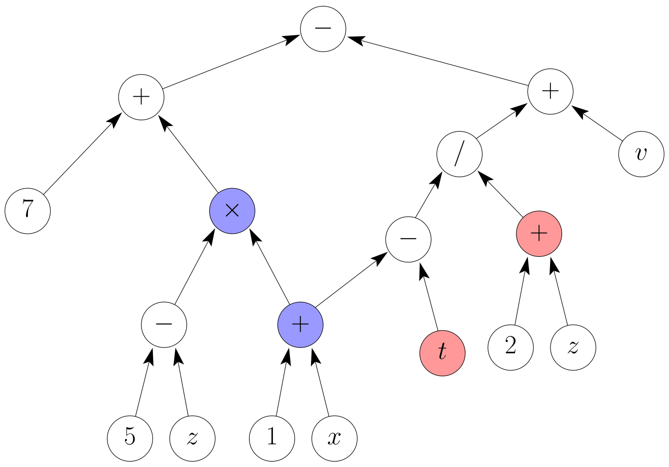

Figure 7: Playing the red/blue pebble game on the graph of Figure 2. The two red nodes represent values that are currently in memory, whereas the two blue nodes represent values that have been computed but then been evicted to disk. Before pebbling node $(1 + x) - t$, a red pebble has to be put again on node $1 + x$, corresponding to reading this value from the disk.

Definition 3. *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 where the initialization rule (PG2) is disallowed, maximized over all initial placements of the S pebbles.*

Then we express the lower bound on the number of I/O steps $T_{I/O}(S, G)$, defined as the number of steps using rules RB1 or RB2 in a pebbling scheme S of the graph G.

Theorem 5. *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:*

$$\lceil T_{I/O}(S, G)/S \rceil \rho(2S, G) \geq |V| - |Inputs(G)|$$

Proof. We first divide the pebbling scheme S into phases where each phase has exactly S I/O operations (except the last one which may have less). There are $h = \lceil T_{I/O}/S \rceil$ such phases.

We now establish a bound on the maximal number of vertices computed during each phase. To do this, we transform the pebbling of a phase to move all read operations to the beginning of the phase. Let V_{read} be the set of vertices that are pebbled with rule RB1 (a blue pebble is transform in a red pebble) during the phase. Note that $|V_{read}| \leq S$ by definition of the phases. We consider a supplementary set of $|V_{read}|$ red pebbles, that are placed in the beginning of the phase on the vertices of V_{read}, and remove the read steps from the pebbling scheme of the phase. Note that the new pebbling scheme of the phase is still valid. At the start of the new phase, there are at most $2S$ red pebbles on the graph. Clearly, the number of vertices pebbled with a red pebbled in not larger that if $2S$ vertices were available and allowed to move freely. Thus, the number of compute steps is at most $\rho(2S, G)$.

In total, the number of compute steps of the whole pebbling scheme is at most $h\rho(2S, G)$, and it sould be at least $|V| - |Inputs(G)|$ to cover the whole graph, which gives the results. □

We now apply this result to our running example: the product of two $N \times N$ matrices. In order to do this, we first estimate the span of any DAG computing this product, corresponding to any standard algorithm.

We start with a small lemma.

Lemma 2. *Let T be a binary (in-)tree representing a computation, with p pebbles on some vertices. At most $p - 1$ vertices can be pebbled in the tree.*

Proof. We first identify in the tree the s subtrees with all leaves pebbled. All the non-leaf vertices of these subtrees (and only these vertices) can be pebbled. For such a subtree with l leaves, this represent $l-1$ nodes. Hence, exactly $p-s$ vertices can be pebbled in the whole tree, which is at most $p-1$ in the case of a single subtree. □

Theorem 6. *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 $\rho(S,G)$ is such that $\rho(S,G) \leq 2S\sqrt{S}$ for $S \leq N^2$.*

Proof. We consider any initial placement of the S pebbles, and try to bound the number of vertices that can be pebbled from this placement. Since we consider a regular matrix product $C = AB$, all $C_{i,j}$ elements are obtained as inner product (or dot product) of the inputs: all products $A_{i,k}B_{k,j}$ are performed, and then summed through addition trees (which is a binary tree). Note inputs are shared for several products, while addition trees are not shared.

We consider that among the vertices already pebbled initially, r are nodes of addition tree (either product or addition) and $S-r$ are inputs (no output are pebbles to maximize the span). Let p be the number of products that can be pebbled from the $S-r$ inputs (we will bound it below). After all products are performed (it can be done in a first step without loss of generality), we end up in $p+r$ nodes in several addition trees. Thanks to the previous lemma, at most $p+r-t$ new vertices can be pebbled when t trees are used, which is at most $p+r-1$. Thus, the span is bounded by $2p+r-1$ (p products and $p+r-1$ additions).

We now concentrate on bounding p. We consider matrices A' and B' whose (i,j) entry is 1 when the corresponding input in A or B is initially pebbled, and 0 otherwise. Then we consider the product $C' = A'B'$. $C_{i,j}$ is equal to the number of products that can be performed from the initial position of the pebbles that will contribute to the $C_{i,j}$ element. Thus, $p = \sum_{i,j} C'_{i,j}$.

Let a (resp. b) be the number of 1 in A' (resp. B'), such that $a+b = S-r$ and let α be any integer smaller than N. There are at most a/α rows of A' with at least α ones ("dense" rows). The maximum number of non-null products that can be performed using these rows is ab/α, since each (full) row can contribute to at most b products. We now consider the other rows of matrix A' (or A). Given the limited number of pebbles, at most S final elements of C can be produced. For the "sparse" part of the matrix A, this corresponds to at most αS products. In total, this gives $p = ab/\alpha + \alpha S$, which gives $p = 2\sqrt{abS}$ for $\alpha = \sqrt{ab/S}$. The maximum number of products is obtained with $a = b = (S-r)/2$ and is equal to $p = (S-r)\sqrt{S}$. The span is the bounded by $2(S-r)\sqrt{S} + r - 1 \leq S\sqrt{S}$. □

Hence, the I/O time for the matrix product is bounded by:

$$T_{I/O} = \Theta\left(\frac{N^3}{\sqrt{S}}\right)$$

Note that these results may be extended to deeper memory hierarchy, using more pebble colors (see [8]).

Material used for this course and further documentation

Refer to chapter 10 ("Space-Time Tradeoffs") of the book "Models of Computation" by J. Savage [8] for more information on the black pebble game, and to chapter 11 ("Memory-Hierarchy Tradeoffs") for more information on the red-blue pebble game.

References

[1] Philippe Flajolet, Jean-Claude Raoult, and Jean Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.

[2] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3):513–524, 1980.

[3] Susan L Graham, Marc Snir, Cynthia A Patterson, et al. *Getting up to speed: The future of supercomputing.* National Academies Press, 2005.

[4] J.-W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *STOC'81: Proceedings of the 13th ACM symposium on Theory of Computing*, pages 326–333. ACM Press, 1981.

[5] Thomas Lengauer. Black-white pebbles and graph separation. *Acta Informatica*, 16(4):465–475, 1981.

[6] Friedhelm Meyer auf der Heide. A comparison of two variations of a pebble game on graphs. *Theoretical Computer Science*, 13(3):315–322, 1981.

[7] Lynette I Millett and Samuel H Fuller. *The Future of Computing Performance:: Game Over or Next Level?* National Academies Press, 2011.

[8] John E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley, 1998.

[9] Ravi Sethi. Complete register allocation problems. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 182–195, New York, NY, USA, 1973. ACM Press.

[10] Ravi Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.

[11] John Shalf, Sudip S. Dosanjh, and John Morrison. Exascale computing technology challenges. In *9th International conference on High Performance Computing for Computational Science - VECPAR 2010*, pages 1–25, 2010.

[12] Arthur N Strahler. Hypsometric (area-altitude) analysis of erosional topography. *Geological Society of America Bulletin*, 63(11):1117–1142, 1952.

[13] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[14] Peter van Emde Boas and Jan van Leeuwen. Move rules and trade-offs in the pebble game. In *Theoretical Computer Science 4th GI Conference*, pages 101–112. Springer, 1979.