A MODEL FOR HIERARCHICAL MEMORY

Alok Aggarwal
Bowen Alpern
Ashok K. Chandra
Marc Snir

IBM T. J. Watson Research Center
P. O. Box 218, Yorktown Heights, New York, 10598.

ABSTRACT

In this paper we introduce the Hierarchical Memory Model (HMM) of computation. It is intended to model computers with multiple levels in the memory hierarchy. Access to memory location x is assumed to take time $\lceil \log x \rceil$. Tight lower and upper bounds are given in this model for the time complexity of searching, sorting, matrix multiplication and FFT. Efficient algorithms in this model utilize locality of reference by bringing data into fast memory and using them several times before returning them to slower memory. It is shown that the circuit simulation problem has inherently poor locality of reference. The results are extended to HMM's where memory access time is given by an arbitrary (nondecreasing) function. Tight upper and lower bounds are obtained for HMM's with polynomial memory access time; the algorithms for searching, FFT and matrix multiplication are shown to be optimal for arbitrary memory access time. On-line memory management algorithms for the HMM model are also considered. An algorithm that uses LRU policy at the successive "levels" of the memory hierarchy is shown to be optimal.

1. The Hierarchical Memory Model

Modern computer systems typically have a great deal of complexity associated with their memory hierarchy. There is often a small amount of fast memory (e. g. registers) augmented with increasingly larger amounts of slower memory. There may be a level of cache (and sometimes two levels as in the Hitachi 680 and Fujitsu 380 machines), main memory, extended store (e.g. in the IBM 3090 machine), disk drives, and mass storage. In order to solve problems efficiently in such an environment, it is important to stage the flow of data through the memory hierarchy so that once the data are brought into faster memory they are used as much as possible before being returned to the slower memory. This is sometimes called the *(temporal) locality of reference* property of the algorithm. It has been much studied [De68, Ch76, HDR80, Sm82] and utilized by operating systems [De70, Sp78, Ba80, Sm82] and algorithm designers.

In this paper we introduce the hierarchical memory model of computation (henceforth called, HMM) and use it for a theoretical investigation of the inherent complexity of solving problems in a RAM with a memory hierarchy. One of the consequences is some understanding of inherent limits of reference locality in problems. There has been some theoretical work directed at hierarchical memories [e. g. MGS70, Si83, HK81]. It is different than ours in that the first two references aim to optimize hierarchies given some locality of reference (or hit ratio) behavior, and the third considers a single level of cache.

An HMM is a random access machine where access to memory location x requires $\lceil \log x \rceil$ *time* instead of the typical constant access time: see [MC80] for a justification of the $\log x$ cost function. (later we consider other cost functions). It is a particularly simple model of computation that mimics the behavior of a memory hierarchy consisting of increasingly larger amounts of slower memory (see also [MC80]). We view this model only as a beginning in the theoretical exploration of memory hierarchies (it does not capture, for example, the added complication of collecting the data into blocks). This model is successful in as much as it shows nontrivial facts about certain problems (e.g. sorting and searching) being inherently hard and that efficient algorithms for certain problems (e. g. matrix multiplication, sorting) tend to mimic the behaviour of pragmatically good algorithms in utilizing the locality of reference [IBM86, Ag86].

Definition of the Hierarchical Memory Model (HMM): There are an unlimited number of registers $R_1, R_2, R_3, \ldots$ each of which can store an integer (or an element in some domain specified by the problem). The operations are similar to those of a RAM [AHU74], except that accessing register R_i takes time $\lceil \log i \rceil$ and executing any operation takes unit time. For convenience, we will allow n-ary operations: the time to perform $R_i \leftarrow f(R_{i_1}, R_{i_2} \ldots)$ takes time $1 + \lceil \log i \rceil + \lceil \log i_1 \rceil + \lceil \log i_2 \rceil + \ldots$. (In passing, we observe the similarity with the log l delay model for signal propagation in VLSI, since we can imagine that a memory level with 2^i locations would be physically 2^{ci} distant from the computing element and require $\Theta(i)$ time to access in the log l delay model. Of course, there are essential differences since the log l delay model contains many processing elements, hence the techniques developed in [A85] do not seem applicable to the HMM).

The HMM can be viewed as having a hierarchy of memory levels:

1 location taking time 0 to access
1 location taking time 1 to access (level 0)
2 locations taking time 2 to access (level 1)
. . . . .
2^i locations taking time $i + 1$ to access (level i).

It can be noted that algorithms in the HMM are fairly robust. For instance, the memory available for an algorithm could be doubled by moving all data down one level in the hierarchy. The time for each access would increase by one, and hence the total running time would increase by at most a multiplicative constant. Likewise, a bounded number of $O(1)$ access time registers could be added to an algorithm without increasing the running time appreciably for the other instructions.

It is clear that for polynomially bounded $T(n)$, problems which can be solved in time $T(n)$ (resp. require time $T(n)$) on a Random Access Machine can be solved in $O(T(n) \log n)$ (resp. require time $T(n)$) on the HMM. Some algorithms can be implemented such that when data are brought into fast memory they are used many times before being returned to slower memory. In such cases it may be possible to solve the problem in time $O(T(n))$ on the HMM. We say such problems are *local*. An example is matrix multiplication using only the semiring operations (+ , x). Here the best possible algorithm on a RAM takes time $\Theta(n^3)$, and likewise for the HMM. For other problems, however, the time $T(n) \log n$ cannot be improved. We say such problems are *nonlocal*. Trivial examples are linear time RAM algorithms where the entire input has to be examined (eg. finding the sum or maximum of a given set of numbers, depth first search in a graph, planarity, etc.). It is shown that searching and circuit simulation are also nonlocal. There are yet other problems where the ratio of the running times of the best HMM algorithm to the best RAM algorithm is between these two extremes, and we call these problems *semilocal*. It is shown that this ratio for FFT and sorting is $\Theta(\log \log n)$.

We can also consider the HMM generalized to other cost functions: the memory access time is measured by an arbitrary (monotone nondecreasing) function f. When f is the function defined by $f(x) = 1$ if $x > m$, $f(x) = 0$ otherwise, then one measures *I/O complexity*, in a machine with memory of size m : accesses to any of these m memory locations is free, any other access has cost one. I/O complexity has been previously studied, and lower bounds for sorting, matrix multiplication and FFT are known in this model [HK81, AV86]. It turns out that there lower bounds can be used to derive lower bounds for arbitrary cost functions: the cost of an algorithm with arbitrary cost function f is a weighted sum of its I/O complexity, for varying memory size m. Also, an algorithm that has optimal I/O for any memory size is an optimal HMM algorithm for any cost function. These ideas are used to derive lower bounds on time complexity of searching, sorting, matrix multiplication and FFT on HMM's with logarithmic and polynomial memory access time; the algorithms for searching, matrix multiplication and FFT are shown to be optimal for any cost function.

The management of memory in a computer with a two level memory hierarchy is usually done transparently from the user; the operating system uses adaptive demand paging algorithms so as to take advantage of the temporal locality of memory accesses in the user code. Recent results of Sleator and Tarjan [ST85] provide a theoretical justification for such policy: the memory access cost when an LRU page replacement algorithm is used is worse by at most a constant factor compared with the cost that would obtain for an optimal, off-line memory management algorithm, that has advance knowledge of the sequence of memory accesses to be executed. We prove in this paper a similar result for an HMM with arbitrary memory access time: the memory is divided into successive levels, where memory access time doubles at each level; LRU policy is used to manage data movement accross each level. The results allow us to relate directly the complexity of an algorithm in the HMM model to the temporal pattern of its memory accesses.

Section 2 of the paper gives HMM algorithms for some problems: matrix multiplication, FFT, and sorting. Section 3 provides techniques for determining lower bounds, with applications to searching, FFT, sorting, and circuit simulation. In section 4 we consider generalizations of the HMM to arbitrary cost functions for memory access, and to the function n^α in particular. Section 5 concerns memory management in the HMM. Section 6 presents conclusions and open problems.

2. Upper Bounds

2.1. Matrix Multiplication using Semiring Operations

Two matrices can be multiplied as quickly on an hierarchical memory machine as on a RAM (within a multiplicative constant).

Theorem 2.1: Two $n \times n$ matrices can be multiplied in time $O(n^3)$ using only (+, ×) on an HMM.

Proof Outline: We use a divide and conquer algorithm: to multiply two $n \times n$ matrices, the algorithm recursively multiplies 8 pairs of $n/2 \times n/2$ matrices by first moving them into faster memory. Then the resulting 4 pairs of $n/2 \times n/2$ matrices are added. For such an algorithm, it is easy to show that the execution time, $T(n)$, obeys the recurrence relation:

$$T(n) \leq 8T(n/2) + O(n^2 \log n).$$

Since $T(1) = \mathit{constant}$, it can be verified that $T(n) = O(n^3)$.

■

Of course, $\Omega(n^3)$ is a lower bound for matrix multiplication (using semi-ring operations) on a RAM. Thus, matrix multiplication (using semi-ring operations) is a local problem.

2.2. Fast Fourier Transforms

Computational problems can often be represented as directed acyclic graphs whose nodes correspond to the computed values and whose arcs correspond to the dependency relations. Here, the input nodes (i. e., the nodes with in-degree zero) are provided to the algorithm with given values and a node can only be computed if all its predecessor nodes have been computed. The time taken to compute a dag is the time taken by the algorithm to compute all its output nodes (i. e., the nodes with out-degree zero). The FFT (Fast Fourier Transform) graph is one such directed acyclic graph that is quite useful, and several problems can be solved by using an algorithm for computing the FFT graph.

On a RAM, the n-point FFT graph can be computed in $\Theta(n \log n)$ time. In this section, we present an upper bound on the worst-case time for computing the n-point FFT graph in the HMM. Theorem 2.2 provides an $O(n \log n \log \log n)$ time algorithm that computes such a graph. In the next section, we will show a matching lower bound. Thus, FFT is a semilocal problem.

For $n = 2^k$, the n-point FFT graph can be algorithmically represented as follows:

Inputs: $x_{0,0}, x_{0,1} \cdots, x_{0,n-1}$.

Outputs: $x_{k,0}, \cdots, x_{k,n-1}$.

Computation: $x_{j,m} = f(x_{j-1,m}, x_{j-1,p})$,

where f is function on two variables that can be computed in constant time and the binary representations of m and p are identical except in the $(j-1)$-th position.

Theorem 2.2: An n-point FFT graph can be computed in $O(n \log n \log \log n)$ time in the HMM.

Proof Outline: The n-point FFT graph ($n = 2^{2k}$) can be represented as 2^k independent 2^k-point FFT graphs that compute $x_{k,0}, \ldots, x_{k,n-1}$ followed by another 2^k independent 2^k -point FFT graphs. Each of these $2n^{1/2}$ FFT graphs can be solved by first bringing the corresponding inputs into the faster memory (in the first $O(n^{1/2})$ locations), solving the smaller FFT graphs and then returning the results to the slower memory. For $n = 2^{2k+1}$, the n-point FFT graph can be represented as two sets of 2^{k+1} independent 2^k-point FFT graphs, followed by n applications of the function f. A simple analysis shows that the time, $T(n)$, for executing such an algorithm obeys the following recurrence relation:

$$T(n) \leq 2n^{1/2}T(n^{1/2}) + O(n \log n).$$

Furthermore, since $T(1) = \mathit{constant}$, it can be verified that $T(n) = O(n \log n \log \log n)$.

■

It is interesting to contrast the locality of reference for matrix multiplication with that for FFT. In the former, when an element is moved into fast memory, say into the log k level of memory (which has k locations), it is used in about $\sqrt{k}$ computations before it is returned to slower memory. For FFT on the other hand, it is used in only log k computations (actually k elements are collectively used in $k \log k$ computations). The lower bounds of the next section show that this is about the best that can be hoped for.

2.3. Sorting

The HMM can use locality of reference to sort n elements in better than the obvious time bound of $O(n \log^2 n)$. Consider the following algorithm:

> Partition the given n inputs into $2^{\sqrt{\log n}}$ subsets of size $n/2^{\sqrt{\log n}}$ each,
>
> sort these subsets recursively,
>
> and merge the resulting sorted subsets.

This algorithm can be made to run in $O(n \log^{3/2} n)$ time in the HMM. We show that this can be improved to $O(n \log n \log \log n)$ time by generalizing the classical median-sort algorithm in a non-trivial manner.

Theorem 2.3: A set, S, of n elements can be sorted in the HMM in $O(n \log n \log \log n)$ time.

Proof Sketch: Assume that the elements of S reside in the $(\log n + 2)$-th level of the hierarchical memory and are all distinct. Consider the algorithm *Approx-Median-Sort* (S) that has the following six steps:

Step 1: Partition S into $\sqrt{n}$ subsets, $S_1, \ldots, S_{\sqrt{n}}$, each of size $\sqrt{n}$. Sort each subset by calling *Approx-Median-Sort* recursively, after bringing the elements of the subset to the $(0.5 \log n + 2)$-th level of the hierarchical memory.

Step 2 : Form a set A containing the $(i \log n)$-th element of each S_j, $1 \le i \le \sqrt{n} / \log n$, $1 \le j \le \sqrt{n}$. (Note that A has $n/ \log n$ elements. Furthermore, a simple analysis shows that the l-th smallest element in A is greater than at least $l \log n - 1$ elements in S and also this element is greater than no more than $l \log n + \sqrt{n} \log n - 1$ elements in S.) Sort A by simply adapting any procedure that takes $O(m \log m)$ time for sorting m elements in the RAM model.

Step 3: Form a set, $B = \{b_1, b_2, \ldots, b_r\}$ of $r = \sqrt{n} / \log n$ *approximate-partitioning elements*, such that for $1 \le k \le r$, b_k equals the $(k\sqrt{n})$ -th smallest element of A. (Note that because of the remark made in step (2), there are at least $k\sqrt{n} \log n - 1$ but no more than $(k+1)\sqrt{n} \log n - 1$ elements of S that are less than b_k.)

Step 4: For $1 \le j \le \sqrt{n}$, merge B with S_j, forming sets $S_{j,0}, \ldots, S_{j,r}$, where $b_k < x < b_{k+1}$, for each element $x \in S_{j,k}$.

Step 5 : For $0 \le k \le r$, form $C_k = \bigcup_j S_{j,k} \cup \{b_k\}$, ignoring the $\{b_k\}$ term when $k = 0$. (By the remarks in step 3, $1 \le |C_k| \le 2\sqrt{n} \log n$.)

Step 6 : For $1 \le k \le r$, sort C_k by bring it into the $(\log |C_k| + 2)$-th level and calling *Approx-Median-Sort*(C_k) recursively. The sorted list for S consists of the sorted lists for $C_0, \ldots, C_r$ in sequence.

***end*.**

The correctness of the *Approx-Median-Sort* is easy to establish and hence omitted. To analyze the running time, let $T(n)$ denote the time taken by *Approx-Median-Sort* to sort n elements. Step 1 can be executed in $\sqrt{n}\, T(\sqrt{n}) + O(n \log n)$ time. Step 2 takes time $O(n \log n)$ since $|A| = n/ \log n$ and sorting it on a RAM would take time $O(n)$, and hence $O(n \log n)$ on the HMM. Similarly, steps 3 , 4 , 5 also take time $O(n \log n)$. The running time obeys the following recurrence relation:

$$T(n) \le \sqrt{n}\, T(\sqrt{n}) + \sum_{k=0}^{\sqrt{n}/\log n} T(c_k) + O(n \log n)$$

where $1 \le c_k \le 2\sqrt{n} \log n$ and $\Sigma c_k = n$. Since $T(4) = constant$, it can then be verified that $T(n) = O(n \log n \log \log n)$.

■

3. Lower Bounds

In what follows we will ignore processing time, charging only for memory accesses. This merely strengthens lower bounds, and affects upper bounds at most by a constant factor, provided that access to each register has positive cost. In order to prove lower bounds it is convenient to introduce new *cost functions* , that charge differently for memory accesses. A cost function f can be any nonnegative, monotone nondecreasing function defined on the set of positive integers. It is convenient to let $f(0) = 0$. We denote by T_f the running time of a computation where an access to register i requires time $f(i)$. In particular, we shall use *threshold* cost functions. The threshold function U_m is defined by

$$U_m(i) = \begin{cases} 0 & \text{if } i \le m \\ 1 & \text{if } i > m \end{cases} .$$

We write T_m for T_{U_m}.

T_m is the running time of a computation, where an access to one of the first m memory locations is free, and access to any other memory location has cost one. This is the *I/O complexity* of the computation, when executed on a machine with a memory of size m: a computation is charged according to the number of accesses executed to locations outside the main memory. The I/O complexity of computations has been studied by several authors, and lower bounds are known for most of the problems considered in this paper. We have

Theorem 3.1:

(i) The I/O complexity of sorting or computing an FFT graph is

$$T_m(n) = \Omega(\frac{n \log n}{\log m} - m).$$

(ii) The I/O complexity for the multiplication of two $n \times n$ matrices using (+, x) is

$$T_m(n) = \Omega(\frac{n^3}{\sqrt{m}} - m).$$

(iii) The I/O complexity for searching is

$$T_m(n) = \Omega(\log n - \log m).$$

Proof:

(i)&(ii) Lower bounds for sorting, FFT computations and matrix multiplications are given in [HK81] and [AV86]. Note that their results differ from ours by a term of $-m$: the model used by these authors assume that all inputs are initially in secondary memory, whereas we assume that m of the inputs can initially be in the main memory. In particular, if the problem fits in main memory, then its I/O complexity is zero.

(iii) We represent the searching algorithm by a binary decision tree. Each node is associated with a comparison that involves one access to memory. Each access to a key that is not initially in main memory has a cost of one. The decision tree has $n + 1$ leaves, and at most m of the internal nodes have cost zero, whereas the remaining nodes have cost one. It is easy to see that there must be a path in the tree that contains at least $\log n - \log m$ nodes with cost one. The lower bound follows.

■

Any cost function can be represented as a weighted sum of threshold functions, with nonnegative weights. Let Δf be the first order difference of the function f:

$$\Delta f(x) = f(x+1) - f(x).$$

Since $f(0) = 0$ it follows that

$$f(x) = \sum_{m<x} \Delta f(m) = \sum_{m} \Delta f(m) U_m(x).$$

It follows that for any computation the cost T_f of that computation with f cost function, and the I/O cost T_m are related by the equality

$$T_f = \sum_{m} \Delta f(m) T_m. \qquad [1]$$

As f is monotone nondecreasing the coefficients $\Delta f(m)$ are nonnegative. Thus, lower bounds on I/O complexity can be used to derive lower bounds for arbitrary cost functions.

Theorem 3.2: Let T denote cost in the HMM model, with cost function $f(x) = \lceil \log x \rceil$. Then

(i) The cost of searching is

$$T(n) = \Omega(\log^2 n).$$

(ii) The cost of computing an FFT graph or sorting is

$$T(n) = \Omega(n \log n \log\log n).$$

Proof: If $f(x) = \lceil \log x \rceil$ then

$$\Delta f(x) = \begin{cases} 1 & \text{if} \quad x = 2^k \\ 0 & \textbf{otherwise} \end{cases}$$

Thus, by identity [1], the complexity of a computation with log cost is given by

$$T = \sum_{k} T_{2^k}.$$

We use now the lower bounds of Theorem 3.1. to derive lower bounds for log cost. For searching we have

$$T(n) = \Omega\left(\sum_{k} \max(0, \log n - \log 2^k) \right)$$

$$= \Omega\left(\sum_{k=0}^{\log n} (\log n - k) \right)$$

$$= \Omega\left(\log^2 n \right).$$

For sorting and FFT we have

$$T(n) = \Omega\left(\sum_{k} \max(0, \frac{n \log n}{\log 2^k} - 2^k) \right)$$

$$= \Omega\left(\sum_{k=1}^{\log n} (\frac{n \log n}{k} - 2^k) \right)$$

$$= \Omega(n \log n \log\log n).$$

■

Definition: An *n-circuit* (*circuit* for short) is a directed acyclic graph with bounded in-degree, n inputs (vertices with zero in-degree), and n outputs (vertices with zero out-degree). Let $V(C)$ be the number of vertices in a circuit C. We think of each vertex with in-degree k as representing a k -ary function of its predecessors computable in $O(1)$ time, and the circuit as computing n outputs given n inputs. For an n-circuit C, let C^p be the n-circuit obtained from p copies $C_1, C_2, \ldots, C_p$ of C by identifying the outputs of C_i with the inputs of C_{i+1}, $1 \leq i < p$. The *circuit simulation problem* is the following: given a circuit C, values for its n inputs, and $p > 0$, compute the outputs of C^p (the value of each node is to be computed from the values of its immediate predecessors).

The circuit simulation problem corresponds to simulating a VLSI circuit for p cycles, where the inputs and outputs are thought of as the flip flops. It is widely used for designing machines, VLSI testing, timing analysis, etc, and it can be solved in time $V(C)p$ on a RAM. We show that it is a nonlocal problem.

Theorem 3.3: For every n, p there are n-circuits with $O(n)$ vertices such that the I/O complexity for circuit simulation is $T_m(n, p) = \Omega(np)$, for $m \leq cn$, for a fixed constant $c > 0$. The circuits can be restricted to have in-degree two.

Proof: Let C' to be an expanding bipartite graph with in-degree bounded by k representing an n-circuit, such that for any subset S of outputs, $|S| \leq an$, the number of adjacent inputs is at least $(1+b)|S|$, for some $a, b > 0$. C is obtained from C' by replacing outputs of C' with in-degree j by a tree of $j-1$ vertices each with in-degree 2. We will call the additional $j-2$ vertices "internal" and the output vertices of C "external", and apply these designations to the corresponding vertices in C^p. Let S be any set of vertices in C^p containing r external vertices, $r \leq an$, and let T be the set of immediate predecessors of vertices in S. Then it can be shown that $|T - S| \geq \frac{b}{1+b} r$ (by observing that if x is the number of external vertices in S all of whose corresponding internal vertices are also in S

then $|T - S| \geq \max(r - x,(r - x) + (1 + b)x - r))$. Let $m \leq \frac{ab}{2(1 + b)}n$. Then the I/O complexity for computing the values of *an* external vertices of C^p is at least $\frac{ab}{2(1 + b)}n$ since it requires $\frac{ab}{1 + b}n$ values of which at most half could reside in the m locations. Hence

$$T_m(n,p) \geq \frac{ab}{2(1 + b)}n \times \lfloor np/an \rfloor \geq \frac{b}{2(1 + b)}n(p - a) = \Omega(np).$$

■

Corollary 3.4: Circuit simulation of n vertex graphs for p cycles requires time $\Omega(np \log n)$ on the HMM with cost function $f(x) = \lceil \log x \rceil$.

Since p can be chosen to be a function of n (eg. $\log n$, n, n^2, etc.), circuit simulation provides an example of a nonlinear RAM problem (its RAM time is $O(np)$) that is nonlocal.

4. Other Memory Access Cost Functions

The lower bounds given in the previous section can be readily extended to other cost functions, using identity [1]: A cost function defines a "distribution" over possible I/O boundaries, and lower bounds on complexity with respect to an arbitrary cost function is obtained by "integrating" I/O complexity with respect to that distribution. Consider, for example, polynomial cost functions:

$$f(x) = x^{\alpha},$$

for some positive exponent α. We have

Theorem 4.1. Let T denote cost in the HMM model, with cost function $f(x) = x^{\alpha}$, where $\alpha > 0$. Then

(i) The cost of matrix multiplication is

$$T(n) = \blacksquare \begin{matrix} n^{2\alpha+2} & \text{if} & \alpha > 1/2 \\ n^3 \log n & \text{if} & \alpha = 1/2 \\ n^3 & \text{if} & \alpha < 1/2 \end{matrix}$$

(ii) The cost of searching is

$$T(n) = \Omega(n^{\alpha})$$

(iii) The cost of sorting or computing an FFT is

$$T(n) = \Omega(n^{\alpha+1}).$$

Proof: We use tne lower bounds of Theorem 3.1, together with identity [1]. It is convenient to use the identity

$$\sum_{i=1}^{n} \Delta f(i) g(i) = -\sum_{i=1}^{n-1} \Delta g(i) f(i + 1) + f(n + 1)g(n) - f(1)g(1).$$

For matrix multiplication we have $g(x) = n^3/\sqrt{x} - x$, so that $-\Delta(x) = \Omega(n^3/x^{3/2})$. It follows that

$$T(n) = \Omega(n^3 \sum_{i=1}^{n^2} i^{\alpha-3/2}).$$

But

$$\sum_{i=1}^{n^2} i^{\alpha-3/2} = \begin{cases} \Theta(n^{2\alpha-1}) & \text{if } \alpha > 1/2 \\ \Theta(\log n) & \text{if } \alpha = 1/2 \\ \Theta(1) & \text{if } \alpha < 1/2 \end{cases}$$

which implies (i).

For searching we have $g(x) = \log n - \log x$ and $-\Delta g(x) = \Omega(1/x)$. It follows that

$$T(n) = \Omega(\sum_{i=1}^{n} i^{\alpha-1}) = \Omega(n^{\alpha}).$$

The lower bound for sorting and FFT follows from the fact that each input must be accessed at least once, which costs

$$\Omega(\sum_{i=1}^{n} i^{\alpha}) = \Omega(n^{\alpha+1})$$

■

The representation of arbitrary cost functions as weighted sums of threshold functions can be used to build algorithms that are optimal for any cost function: an algorithm that has optimal cost T_m for any threshold m will also have optimal cost T_f for any cost function f. Such algorithms are said to be *uniformly optimal*. We proceed to show that algorithms presented in section 2 are uniformly optimal (with some constraints on the cost function f).

The binary search algorithm has I/O complexity $\log n - \log m + O(1)$, provided that the keys are stored in a correct order: We store them according to their position in the binary decision tree that describes the algorithm; the tree is stored in a breadth-first order. Thus, for each k, the $2^k - 1$ keys that may be accessed in the first k steps of the search are stored in the lowest $2^k - 1$ memory locations. It follows that binary search is uniformly optimal.

We do not have such sharp upper bounds for the remaining problems. In order to obtain bounds which are correct within a constant factor we assume that the cost function f is polynomially bounded: There exists a constant c such that for any x

$$f(2x) \leq cf(x) \qquad [2]$$

When f fulfills that condition the amount of memory available for an algorithm can be doubled while increasing running time at most by a constant running factor. Moreover,

$$\sum_{i=1}^{n} f(i) = \Theta(nf(n))$$

and

$$\sum_{i=1}^{n} i\Delta f(i) = nf(n+1) - f(1) - \sum_{i=1}^{n-1} f(i+1) = O(nf(n)).$$

Thus, any algorithm that uses space n has complexity at least $\Omega(nf(n))$; an $O(m)$ gap between lower bound and upper bound for I/O complexity T_m contributes at most an $O(nf(n))$ gap between lower bound and upper bound for T_f cost, which is at most a constant factor gap.

The I/O complexity of the matrix multiplication algorithm presented in section 2 fulfills the recursion

$$T_m(n) \leq 8T(n/2) + O(n);$$
$$T_m(\sqrt{m}) = O(m).$$

Solving that recursion, we obtain that

$$T_m(n) = O(\frac{n^3}{\sqrt{m}}).$$

This matches (up to an m term) the lower bound for I/O. It follows that this algorithm is uniformly optimal for any cost function that fulfills inequality [2].

The I/O complexity of the FFT algorithm presented in section 2 fulfills the recursion

$$T_m(n) \leq 2n^{1/2}T(n^{1/2});$$
$$T_m(m) \leq m.$$

Solving this recursion we obtain that

$$T_m(n) = O(\frac{n \log n}{\log m}),$$

which matches the lower bound for I/O. It follows that this algorithm is optimal, up to a constant factor, for any cost function that fulfills inequality [2].

The circuit simulation problem has poor locality of reference for any cost function. For an n vertex input graph and a p cycle simulation, whereas a RAM takes time $\Theta(np)$, an HMM with cost function f requires time $\Omega(npf(n))$ from Theorem 3.3 (for any f satisfying [2]). There is, of course, a matching upper bound.

5. Memory Management

We have assumed so far in our analysis that the user has explicit control of the physical location of data. Real machines often do not allow such control. The user programs in terms of a virtual memory space. The mapping of virtual addresses into physical memory is determined by the operating system (that allocates page frames in main memory) or by the hardware (that allocates lines in cache). This situation can be modelled as follows: A program computation consists of a sequence $s_1, ..., s_r$ of *memory accesses*; s_j is the virtual address of the j-th accessed location, e.g. the index of the j-th accessed variable. A *memory management algorithm* allocates one or more physical locations to each variable. The mapping of variables onto physical memory may change over time: After each memory access of the computation the memory management algorithm may move variables in physical memory. The memory management algorithm is *on-line* if the location of variables before memory access s_{i+1} is executed depends only on the sequence $s_1, ..., s_i$ of memory accesses; it is *off-line* otherwise. When memory is managed by the operating system or the hardware then memory management is on-line: The system has knowledge of previous memory accesses, but no knowledge of future memory accesses.

Memory management algorithms have been extensively studied for two level memory hierarchies, i.e. for models were memory access cost is measured by a threshold function. One of the most frequently used algorithm is *Least Recently Used* (LRU): when a variable is accessed, it is moved to main memory, where it replaces the least recently accessed variable there. This algorithm is on-line. Sleator and Tarjan [ST85] have shown that this on-line memory management policy is optimal in a very strong sense: for any sequence of memory accesses the performance of LRU is essentially as good as the performance of the best off-line algorithm for this sequence, provided that the LRU algorithm commands the use of a larger main memory. Let $T_f^A(s)$ be the cost of the sequence of memory accesses s when memory management policy A is used, and memory access cost is measured by the cost function f. Let *m-LRU* be the LRU algorithm applied on a memory of size m. We have

Theorem 5.1. For any sequence s of memory accesses, any memory management algorithm A, and any $N > n$

$$T_N^{N\text{-}LRU}(s) \leq \frac{N+1}{N-n+1} T_n^A(s) + n.$$

Proof: A proof of essentially the same result is given in [ST85]. Their model is slightly different as they assume that a variable accessed must be brought into main memory. However, their proof carries almost verbatim to our model.

■

The LRU algorithm can be extended to a memory hierarchy with several levels; LRU is used at the boundary between each pair of successive levels: When a variable is accessed, it is promoted to the fastest memory; the least recently used variable in that memory is demoted to the next level in memory; the least recently used variable at that level is demoted; and so on. We shall show that this extension of LRU is optimal for the HMM model with arbitrary memory access costs. The definition of the successive "memory levels" depends on the cost function f; they are defined so that memory access cost increases by a constant factor at successive levels.

We assume that f fulfills the constraint

$$f(2x) \leq cf(x).$$

We define inductively a sequence of indices i_j: $i_0 = 0$ and $i_1 = 1$. Assume i_{j-1} defined; then i_j is the largest integer such that $f(i_j) \leq 2f(i_{j-1})$. Let $B_j = \{i_{j-1}+1, ..., i_j\}$ We define the *Block-LRU* (BLRU) policy as follows: Initially one copy of each variable is stored in memory. If a variable $x \in B_j$ is accessed then x is moved to B_1 where it replaces the least recently used variable there. This variable is moved to B_2, where it replaces the least recently used variable there; and so on, until the least recently used variable from B_{j-1} is moved to B_j.

Theorem 5.2. For any sequence s of memory accesses involving n variables and any memory management algorithm A

$$T_f^{BLRU}(s) = O(T_f^A(s) + nf(n)).$$

Proof: It is easy to see that

$$f(x) = \Theta(\sum_j f(i_j) U_{i_j}(x)).$$

For each initial segment of the memory of the form $\{1, ..., i_j\} = \bigcup_{r=1}^{j} B_{i_r}$ the BLRU policy on that segment coincides with the LRU policy for this segment: Whenever an access to a variable outside this segment occurs both BLRU and i_j-LRU remove from this segment the least recently used variable. It follows that

$$T_{i_j}^{BLRU} = T_{i_j}^{i_j-LRU}.$$

By our assumption on f there is a constant $c > 1$ such that $i_j \geq ci_{j-1}$. It follows, by the previous theorem, that

$$T_{i_j}^{i_j-LRU} \leq c/(c-1)T_{i_{j-1}}^A + i_{j-1}.$$

Thus,

$$\begin{aligned} T_f^{BLRU} &= O\left(\sum_{i_j \leq n} f(i_j) T_{i_j}^{BLRU} \right) \\ &= O\left(\sum_{i_j \leq n} f(i_j)(T_{i_{j-1}}^A + i_{j-1}) \right) \\ &= O(T_f^A + nf(n)). \end{aligned}$$

■

Note that the memory management problem we consider here is essentially the same as the problem of maintaining a self-organizing linear list, were accesses to the i-th item in the list cost $f(i)$ (with no insertions or deletions). Sleator and Tarjan [ST85] have shown that *Move-to-Front* is an optimal policy, whenever f is a convex function. The assignment of variables to blocks achieved by move-to-front is identical to that achieved by BLRU. Using this fact, the previous proof can be adapted to show that move-to-front is an optimal policy for any cost function f that fulfills the condition $f(2x) \leq cf(x)$ and, in particular, when f is concave. Thus move-to-front is essentially optimal for any cost function.

The previous result does not imply that any algorithm is automatically transformed into an efficient algorithm for an HMM machine when BLRU memory management is used; it implies that the user need not worry about the spatial distribution of variables in memory. The cost of the memory accesses will be determined by the temporal distribution of these accesses. The BLRU policy will reduce the time for memory accesses whenever accesses to a variable are clustered in time by moving this variable into faster memory.

The performance of the BLRU policy on a sequence s of memory accesses is easy to measure. By the previous theorem, this is, up to a constant factor, the best performance of a memory management algorithm on this sequence. Let $j_i = \max \{j \mid j < i, s_j = s_i\}$; $j = 0$ if the first access to variable s_i occurs at cycle i. We define $gap(i)$ to be the number of distinct variables accessed at cycles $j_i+1, ..., i-1$. If LRU policy is applied on a memory of size m then, with the exception of the first m variables accessed, an access s_i has cost zero if $gap(i) \leq m$, cost one otherwise. Thus,

$$\sum_i U_m(gap(i)) \leq T_m^{m-LRU}(s) \leq \sum_i U_m(gap(i)) + m.$$

Using the same argument as in the last theorem we obtain that

$$c_1 \sum_i f(gap(i)) \leq T_f^{BLRU}(s) \leq c_2 \left(\sum_i f(gap(i)) + nf(n) \right)$$

for some positive constants c_1, c_2, where n is the number of variables used. Thus, the cost of a sequence of memory accesses on an HMM machine is obtained by summing the "gaps" between successive accesses to the same variable, where each gap is weighted by the cost function used.

6. Conclusions

This paper introduces the hierarchical memory model (HMM) of computation which is like a RAM except that access to location x takes time $\lceil \log x \rceil$. This implies that the standard polynomial time algorithms can run on the HMM with at most an $O(\log n)$ factor more in the running time. This cannot be improved for some problems such as finding the maximum of an unsorted list, searching for an element in a sorted list, or circuit simulation. For other problems, however, one can find algorithms to reduce this factor by carefully moving the data elements into faster memory, using these elements as much as possible, and then moving these back into slower memory, perhaps temporarily, while other data are processed in the faster memory.

Algorithms are shown in this paper for matrix multiplication using only semiring operations ($O(n^3)$), computing the FFT graph ($O(n \log n \log\log n)$) and sorting ($O(n \log n \log\log n)$). A general technique is developed for showing lower bounds, and is applied to the above problems to give matching lower bounds. The $\Omega(n \log n \log\log n)$ for FFT can be shown to apply to other similar computation networks, such as permutation networks, transitive networks, etc.

The HMM model can be extended to arbitrary cost functions $f(x)$, subject to a few technical restrictions (monotone nondecreasing, polynomially bounded). Perhaps the most relevant is the function $f(x) = x^\alpha, \alpha > 0$. Here, just reading the input of length n takes time $\Theta(n^{1+\alpha})$, and this amount of time suffices for sorting and FFT. Interestingly, matrix multiplication of $n \times n$ matrices (using only $+, \times$) has the following complexity: $\Theta(n^3)$ for $\alpha < 1/2$, $\Theta(n^3 \log n)$ for $\alpha = 1/2$, and $\Theta(n^{2+2\alpha})$ for $\alpha > 1/2$. Furthermore, it is shown that some of the algorithms (FFT, matrix multiplication, searching) are uniformly optimal, i.e. they are optimal within multiplicative constants for any polynomially bounded cost function.

Finally, we examine memory management strategies for the HMM with arbitrary cost functions. It is shown that the memory hierarchy can be divided up into levels such that executing an LRU scheme for each level is within a multiplicative constant of optimal, when optimality even allows knowledge of future memory accesses.

There are several directions for further studies. Many problems can be analyzed for their locality of reference, and running times on the HMM. It would be interesting to find other semilocal problems, particularly where the ratio of the best HMM algorithm and best RAM algorithm is different from $\Theta(\log\log n)$ (with logarithmic memory access time). It would be nice to have a general characterization of local and nonlocal computation networks in terms of their graph-theoretic properties. Finally, there are also possible extensions to this model, particularly those having to do with blocking data (corresponding to "lines of cache" or "pages of storage"). A clean and robust model for this in a general memory hierarchy could be very significant.

Acknowledgements: Prabhakar Raghvan proposed an initial version of the hierarchical memory model; also discussions with Jonathan Buss and Larry Carter have been very helpful.

References:

[A85] A. Aggarwal, "Tradeoffs for VLSI Models with Subpolynomial Delay," 17th Symp. on Theory of Comp., 1985, pp. 59-68.

[Ag86] R. Agarwal, *Personal Communication,* 1986.

[AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, 1974.

[AV86] A. Aggarwal and J. S. Vitter, "The I/O Complexity of Sorting and Related Problems", Proc. 14th. ICALP, Karlsruhe, July 1987 (to appear).

[Ba80] J. L. Baer, "Computer Systems Architecture," Computer Science Press, Potomac, MD, 1980.

[Ch76] C. K. Chow, "Determination of Cache's Capacity and its Matching Storage Hierarchy," IEEE Trans. on Computers, TC-25, No. 2, Feb. 1976, pp. 157-164.

[De68] P. J. Denning, "The Working Set Model for Program Behavior," Comm. of ACM, Vol. 11, No. 5, 1968, pp. 86-96.

[De70] P. J. Denning, "Virtual Memory," ACM Computing Surveys, Vol. 2, Sept. 1970, pp. 153-189.

[HDR80] W. J. Harding, M. H. MacDougall and W. J. Raymond, "Empirical Estimation of Cache Miss Ratios as a Function of Cache Size," Tech. Report PN 820-420-700S, Amdahl Corp., Sept. 26, 1980.

[HK81] J. W. Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," Proc. of the 13th Ann. Symp. on Theory of Computing, Oct. 1981, pp. 326-333.

[IBM86] IBM Engineering and Scientific Subroutine Library, Guide and Reference, Program No. 5668-863, SC23-0184-0, Feb. 1986.

[MC80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980, pg. 316.

[MGS70] R. L. Mattson, J. Gacsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," IBM Systems Journal, Vol. 9, No. 2, 1970, pp. 78-117.

[Si83] G. M. Silberman, "Delayed-Staging Hierarchy Optimization," IEEE Trans. on Computers, TC-32, No. 11, Nov. 1983, pp. 1029-1037.

[Sm82] A. J. Smith, "Cache Memories," ACM Computing Surveys, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

[Sp78] F. J. Sparacio, "Data Processing System with Second Level Cache," IBM Tech. Disclosure Bulletin, Vol. 21, No. 6, 1978, pp. 2468-2469.

[ST85] D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," Com. ACM, Vol. 28, No.2, Feb. 1985, pp. 202-208.