Linear-Time Algorithms in Memory Hierarchies

Kenneth W. Regan[a*]

[a]Department of Computer Science, University at Buffalo
226 Bell Hall, Buffalo NY 14260-2000, USA

This paper studies linear-time algorithms on a hierarchical memory model called *Block Move* (BM), which extends the *Block Transfer* (BT) model of Aggarwal, Chandra, and Snir, and which is more stringent than a pipelining model studied recently by Luccio and Pagli. Upper and lower bounds are shown for various data-processing primitives, and some interesting open problems are given.

Keyword Codes: F.1.1; E.2; F.2.2
Keywords: Models of Computation; Data Storage Representations; Nonnumerical Algorithms and Problems.

1. Introduction

Recent years have seen marked dissatisfaction with the computational realism of the classic machine models, such as Turing machines or the standard integer RAM (see [13, 12]). Many algorithms that theoretically run in linear time on the RAM scale non-linearly when it comes time to implement them. Cook [5,6] proposed replacing the usual unit-cost RAM measure by the *log-cost criterion*, by which an operation that reads an integer i stored at address a is charged $\log i + \log a$ time units. Aggarwal, Alpern, Chandra, and Snir [1] went further by introducing a parameter $\mu : \mathbf{N} \to \mathbf{N}$ called a *memory access cost function*, such that the above operation costs $1 + \mu(a)$ time units. Note that this still treats register contents i as having unit size. Besides $\mu(a) = \log a$, they studied the parameters $\mu_d(a) = a^{1/d}$, which model the asymptotic *latency* of memory laid out on a d-dimensional grid.[2] To allow for pipelining, Aggarwal, Chandra, and Snir [2] defined the *Block Transfer* (BT) model with a special instruction

$$\textit{copy } [a - m \ldots a] \textit{ into } [b - m \ldots b],$$

which is charged $m + \max\{\, \mu(a), \mu(b)\,\}$ time units. In the BT, every function that depends on all bits of the input has time lower bounds of $\Omega(n \log^* n)$ under $\mu_{\log}$, $\Omega(n \operatorname{loglog} n)$ under μ_d (for all $d > 1$), and $\Omega(n \log n)$ under μ_1 [2]. Hence virtually no algorithms can be called linear-time on the BT. Luccio and Pagli [8] argued that the benefits of pipelining need not be limited to individual block-transfers. Their LPM model is a RAM with $m = n^{O(1)}$ registers available on inputs of size n, and $O(\log m)$ delay for accessing any register, irrespective of its address.

None of these models, however, takes the complexity of dealing with real data at the bit-level into account. We give such a model, and investigate fundamental list-processing

*Part of this work was supported by NSF Grant CCR-9011248
[2]The cited authors write f in place of μ and α in place of $1/d$.

operations at the bit level, under the memory access cost functions μ_d. Then we give a Kolmogorov-complexity technique that may impact on what Aggarwal and Vitter [3] call the "challenging open problem" of extending the lower-bound results of theirs and other papers to models that "allow arbitrary bit-manipulations and dissections of records."

2. The Block Move (BM) Model

The BM makes three important changes to the BT. First, units of data are characters rather than integers, and are held in numbered cells on a single tape. Second, any finite transduction S, not just *copy*, can be applied to the data stream in a *block move*

$S\ [a_1 \ldots b_1]$ *into* $[a_2 \ldots b_2]$.

Formally S is a deterministic generalized sequential machine (DGSM), as defined in [7]. If z is the string formed by the characters in cells $a_1 \ldots b_1$, then $S(z)$ is written into the block $a_2 \ldots b_2$ beginning at a_2. The *validity condition* for the move is that the intervals $[a_1 \ldots b_1]$ and $[a_2 \ldots b_2]$ be disjoint, and the *strict boundary condition* is that the output neither overflows nor underflows $[a_2 \ldots b_2]$, i.e., $|S(z)| = |b_2 - a_2| + 1$. Third, the BM provides *shuffle* and *reversal* operations, the latter by allowing $b_1 < a_1$ and/or $b_2 < a_2$ in block moves. Perfect bit-shuffle is implemented by allowing the *blank* B to be a writable character in block moves, with the proviso that every B appearing in the output stream $S(z)$ leaves the previous content of its target cell unchanged. Under a given cost function μ, the charge for a block move is $m + \mu(c)$, where $m = |z| = |b_1 - a_1| + 1$, and $c = \max\{ a_1, b_1, a_2, b_2 \}$.

Several particular machine forms of the BM are given in [10,11]. As shown in [11], the BM is very *robust*: Under any μ_d function ($d \geq 1$ and rational), these forms all simulate each other up to *linear* time, not just polynomial time. Even machines allowed to violate the validity and strict-boundary conditions are simulated with constant-factor overhead by machines that observe them. Thus the complexity classes $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$ of functions computed in time $O(t(n))$ under μ_d are the same for all these forms. Hence we may describe BM algorithms without reference to machine-specific details. Other main points of distinction for the BM are that in contrast to the BT, many interesting functions are computable in linear time, even under the highest cost function $\mu_1(a) = a$. The BM under μ_d is more realistically constrained than the LPM under log-cost; it compromises between fixing the size of linear blocks as in [3] and having unrestricted pipelining as in [8].

3. Fundamental List Operations

Non-negative integers are encoded over $\Sigma = \{ 0, 1 \}$ in standard dyadic notation, with the least significant bit first ("leftmost"). *Lists* whose *elements* are nonempty 0-1 strings are encoded using boundary markers #. In order not to obscure the main ideas, we abstract away the issue of space taken by these markers by extending Σ to an alphabet Γ that includes the characters $0^{\#}$ and $1^{\#}$. Γ also includes a special "padding character" @ and its compound with #, plus a primed "alias" c' for each of the foregoing characters c.

Definition 3.1. A list is *normal* if its elements all have the same length m. A list is *balanced* there exists $j \geq 0$ such that for all i, $1 \leq i \leq n$, $2^{j-1} < |x_i| \leq 2^j$. To *normalize* a list means to pad each of its elements out to the same length m, while to *rectify* the list means to pad each element out to length the next power of 2.

To illustrate timing considerations on the BM, consider the operation $member(w, \vec{x})$. Comparing w separately with each member of $\vec{x}$ would incur access charges for each element addressed, and the sum total of these under μ_d could approach $n^{1+1/d}$. Copying chunks of $\vec{x}$ into low-numbered cells ("cache memory") is better, but would still not run in linear μ_d-time if $\vec{x}$ has many small elements. If, however, the list $\vec{x}$ is *normal* and $|w| = m$, a standard "recursive doubling" idea can be applied: First generate $w\#w\#$, $w\#w\#w\#w\#$, etc. in successive passes until the w-list is at least as long as $\vec{x}$. Then shuffle the two lists character-wise, and do all the comparisons in a single sweep by a DGSM. This runs in linear time even under μ_1 (with the drawback is using linear auxiliary memory). But what to do is $\vec{x}$ is not normal? Normalizing an unbalanced list can nearly square the total bit-length n, but rectifying a list, or normalizing a balanced list, at most doubles the length, and we have:

Proposition 3.1 *A list can be rectified, and a balanced list normalized, in linear μ_1-time and $O(logn)$ block moves.*

Proof. Suppose first that $\vec{x}$ is balanced. Let $m' := 2^j$ from the definition of "balanced," and let $n' := rm'$, which will be the length of the output list. In a single pass over $\vec{x}$, a BM M can produce the list $\vec{x}^e$ whose ith element is x_i^e if $|x_i|$ is even, and $x_i^e@$ if $|x_i|$ is odd. M places $\vec{x}^e$ into cells $n'/2...n'-1$ of a special "holding track," with underflows permitted. A second pass computes the list $\vec{x}^o$ of odd bits of elements. M writes $\vec{x}^o$ onto the main track and recurses on it, writing the "even pieces" to the holding track. The invariants on the downward recursion are that at each level $\vec{x}^o$ is balanced, and for each i, $1 \leq i \leq r$, the ith element of the even piece has the same length as the ith element of the odd piece. The downward recursion stops when each element of $\vec{x}^o$ has length 1.

The upward recursion maintains the invariant that the number k of marked characters in each element y_i makes $|y_i| + k$ a power of 2. At each upward step, M shuffles the leftmost unused piece $\vec{x}^e$ with $\vec{y}$. Corresponding elements have the same number of bits owing to the downward invariant. A single right-to-left pass by a DGSM with two states s, t then makes the following transformations on pairs of bits (first bit from $\vec{x}^e$):

$$\begin{array}{lll} s(c,d) & \mapsto & (c,d)s \quad s(@,d) \mapsto (d')s \quad s(c,d') \mapsto (c',d')s \quad s(@,d') \mapsto (d')t \\ t(c,d) & \mapsto & (c',d')s \quad t(c,d') \mapsto (c',d')t. \end{array}$$

The DGSM is always in state s when it reaches the end of one pair of list elements and encounters a pair $(c^{\#}, d^{\#})$ or $(@, d^{\#})$ or $(c^{\#}, d'^{\#})$ or $(@, d'^{\#})$ that marks the boundary of the next element. These are translated analogously as above. At the end of the upward recursion the original list is reconstituted with the correct number of trailing symbols in each element primed.

Finally, one more pass inserts the padding by translating primed characters c' to $c@$. The recursion for an unbalanced list is similar, except that extra markers are used to indicate when a short element's length has been cut to 1. $\square$

The padding characters are inserted into the middle, so that e.g. 10100 becomes 101@0@0@. Since all elements of equal length are padded the same way, and since all x_i

with $\lfloor \log_2 |x_i| \rfloor \neq \lfloor \log_2 |w| \rfloor$ can be marked for erasure during the recursion, this is good enough for

Corollary 3.2 *All occurrences of a given string w in a given list $\vec{x}$ can be found and marked in linear μ_1-time and $O(\log n)$ passes.* □

However, to pad elements in front or in back, a further trick seems needed. The *shuffle* of two r-element lists $\vec{x}$ and $\vec{y}$ equals $x_1\#y_1\#x_2\#y_2\#\ldots\#x_r\#y_r\#$.

Proposition 3.3 *Within the same asymptotic time bounds, the padding in Proposition 3.1 can be made leading or trailing in each list element.*

Proof. Lemma 6.1(b) of [11] shows how two normal lists can be shuffled in constant-many passes. (The idea is to make spare copies of both lists, overwrite the even elements of one copy and odd elements of the other by @ symbols, triple each @ symbol in one pass, and then overlay the four lists.) Then shuffle the output of Proposition 3.1 with a copy of itself, and in each successive pair of items, mark the padding in one and the original of the other for erasure. This marking can be done in one pass by a DGSM that keeps track of parity, and the added work is linear. □

Binary addition and comparison can be done on the fly by DGSMs after shuffling arguments bitwise. The methods in all these results combine to implement the standard algorithm for parallel prefix sums, even making room for element growth due to carries.

Theorem 3.4 *Prefix-sum, prefix-maximum, and other "census" ([8]) operations on lists, can be computed in linear μ_1-time and $O(\log n)$ passes.*

Moreover, there is a straightforward extension to *segmented* prefix-sum and other "scan" operations. By results of Blelloch [4] on expressing many other operations in terms of prefix-sum and prefix-max, this is enough to prove that the BM efficiently simulates his integer-based *scan* model, except that each scan operation takes $O(\log n)$ block moves.

Consider normal lists $\vec{x}$ and $\vec{y}$ of size $n = rm$ whose elements are sorted. Define the lists to have *small* elements if $m = O(\log n)$, and *large* elements if for some $\epsilon > 0$, $m = \Omega(n^\epsilon)$.

Theorem 3.5 *Under any cost function μ_d with $d > 1$, the problem of merging two lists with large elements can be solved in linear μ_d-time.*

Proof. Consider first the case $m = r = n^{1/2}$. Then the obvious merge by piecemeal comparisons runs in linear time under μ_2, basically because $\sum_{i=1}^{n^{1/2}} (in^{1/2})^{1/2} = O(n)$.

If now $m = n^{1/4}$ and $r = n^{3/4}$, the simple method takes μ_2-time $n^{5/4}$. However, "two levels" of this method makes the time once-again linear: Mark the lists at every interval of $n_0 = n^{1/2}$ bits. Then each chunk has $n_0^{1/2}$ elements, each of size $n_0^{1/2}$. The simple merge of the first chunks takes $O(n_0)$ time under μ_2. The first half of the merge of these two chunks is a correct initial segment of the final merged list, and is copied to a separate portion of memory, thus making room in the "cache" for a new chunk. The next chunk of n_0 bits (from whichever list was exhausted first in the previous step) is copied into the cache in one block move, and the process repeated. The previous analysis now applies to the time under μ_2 to bring down the chunks, and the overall time is linear.

With $m = n^{1/8}$, extending the above recursion to three levels gives linear time under μ_2. Working under higher cost functions μ_d with $1 < d < 2$ has a similar effect to scaling down the element size. That is enough for the proof. $\square$

Corollary 3.6 *Sorting lists with large elements on the BM takes the same time under μ_d with $d > 1$ as the best sequential algorithms do under unit cost.*

This leaves two interesting problems: (1) Can lists with large elements be merged in linear μ_1-time? (2) Can lists with small elements be merged in linear μ_d-time, for any $d \geq 1$? The results on sorting in [2] suggest a negative answer to (2) for all d, but disallow bit operations on data in block transfers. The next section tries to extend their work.

4. Nonlinear Lower Bounds

We consider the problem of changing a given binary string x into a string y of the same length n. Given a cost function μ and a fixed set $\mathcal{S}$ of DGSMs available to a BM program, define $E_\mu(x, y)$ to be the least t such that some sequence of block moves (with DGSMs in $\mathcal{S}$) changes x to y in total μ-time t. We always suppose $\mathcal{S}$ contains *copy* and the single-cell operations S_0, S_1 which write 0 or 1. Also define $e_\mu(n) := \max\{\, E_\mu(x, y) : |x| = |y| = n\,\}$, and write $e_d(n)$ as short for $e_{\mu_d}(n)$ ($d \geq 1$). In particular with $x = 0^n$, $E_\mu(0^n, y)$ is a notion of description complexity for y, and we can fix $x = 0^n$ in defining $e_\mu(n)$.

Theorem 4.1 *For any fixed set $\mathcal{S}$ of block-move operations, $e_1(n) = \Theta(n \log n)$, and for all $d > 1$, $e_d(n) = \Theta(n \operatorname{loglog} n)$.*

Proof Sketch. The upper bounds follow as in [2], and need only S_0, S_1, and *copy*. The lower bounds intuitively hold because an operation with max address a can be specified in $O(\log a)$ bits, but is charged $\mu(a) \gg \log a$ time. (Since there is no disparity for $\mu = \mu_{\log}$, no matching lower bound of $\Omega(n \log^* n)$, analogous to that in [2], is given for $\mu_{\log}$.)

Details of the lower bound for $d = 1$: Given y, let P be a straight-line program such that $P(0^n)$ outputs y, and let $ng(n)$ be the μ_1-time for this. Note that P itself is a description of y. We will modify P into a short(er) description P''. For each i, $1 \leq i \leq k$, call the tape interval $[2^{i-1} \ldots 2^i - 1]$ "region i." Cell 0 is also part of region 1, while cells 2^k onward also count as being in region k. Say that a block move is "charged in region i" if its max address a is in region i. At only a constant-factor cost, we may round charges in region i up to 2^i, and add "dummy moves" to create P' such that every move by P' charged in some region i is followed by a move charged in region $i-1$, i, or $i+1$. Now for each i, $1 \leq i \leq k$, define $N(i)$ to be the number of steps in P' charged in region i. There must exist some i such that $2^i N(i) \leq ng(n)/k$. Choose the *greatest* such i.

Then $N(i) \leq ng(n)/2^i \log n$, and also for each $j > i$, $N(j) > 2^{k-j} g(n)/k$. The moves charged in regions $j > i$ consume μ_1-time at least $(k-i)2^k g(n)/k$. Since $n = 2^k$, the total μ_1-time available for all other moves is at most $ng(n)(1 - (k-i)/k) = ng(n)i/\log n$. By the "adjacency" condition imposed on P', all the moves charged in regions i and above fall into at most $N(i)$ *high segments* of the program P'. For each high segment, let P'' give: (1) the contents of cells $[0 \ldots 2^{i-1}]$ prior to the first move of the segment, and (2) the instructions executed by P in that segment. Finally, after the last high segment, append the first 2^{i-1} bits of y. This finishes P''. Elementary calculation then bounds the length of a straightforward encoding of P'' by $\frac{g(n)}{\log n}[n + C_1 i 2^{k-i} + C_2 i^2 2^{k-i}] = \frac{g(n)}{\log n}\Theta(n)$, where C_1 and C_2 depend on $\|\mathcal{S}\|$. Since there exist (many) strings $y \in \Sigma^n$ such that the conditional Kolmogorov complexity $K(y|0^n)$ is $\geq n$, it follows that $g(n)$ must be $\Omega(\log n)$. $\square$

Corollary 4.2 (compare [2,3]) *There exist permutations of small-element lists that cannot be realized in linear μ_d-time, for any d.*

Proof. Let $N = n \log n$ be the bit-length of the list. Since there are $n! = 2^{\Theta(n \log n)}$ permutations, some have Kolmogorov complexity $\Theta(N)$. The above proof, with input 0^n replaced by the list $1\#2\#\ldots\#n$, shows that every straight-line BM program computing such a permutation requires time $\Theta(N \log N)$ under μ_1, and $\Theta(N \operatorname{loglog} N)$ under μ_d. $\square$

We suspect, eyeing the time-space tradeoff arguments of Mansour, Nisan, and Tiwari [9], that this should lead to non-linear lower bounds on μ_d-time for natural functions such as sorting, string convolutions, FFTs, and universal hashing. This may impact on their conjecture that these functions require non-linear time on a Turing machine. We also ask for lower bounds on testing element distinctness or on taking the intersection of two lists.

REFERENCES

1. A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. 19th STOC*, pages 305–314, 1987.
2. A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th FOCS*, pages 204–216, 1987.
3. A. Aggarwal and J. Vitter. The input-output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.
4. G. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
5. S. Cook. Linear time simulation of deterministic two-way pushdown automata. In *Proceedings, IFIP '71*, pages 75–80. North–Holland, 1971.
6. S. Cook and R. Reckhow. Time bounded random access machines. *J. Comp. Sys. Sci.*, 7:354–375, 1973.
7. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison–Wesley, Reading, MA, 1979.
8. F. Luccio and L. Pagli. A model of sequential computation with pipelined access to memory. *Math. Sys. Thy.*, 26:343–356, 1993.
9. Y. Mansour, N. Nisan, and P. Tiwari. The computational complexity of universal hashing. *Theor. Comp. Sci.*, 107:121–133, 1993.
10. K. Regan. Machine models and linear time complexity. *SIGACT News*, 24:5–15, October 1993. Guest column, L. Hemachandra ed., "Compelxity Theory Column".
11. K. Regan. Linear time and memory efficient computation, 1994. Revision of UB-CS-TR 92-28, accepted to *SIAM J. Comput.*
12. P. van Emde Boas. Machine models and simulations. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1–66. Elsevier and MIT Press, 1990.
13. K. Wagner and G. Wechsung. *Computational Complexity*. D. Reidel, 1986.