Relative cost random access machines

Martti Penttonen*

November 26, 1998

Abstract

The purpose of a model of computation is to provide the algorithm designer with a device for running algorithms. It should be conceptually clear to let him or her concentrate at the algorithmic ideas for solving the problem. At the same time it should be concrete enough to give a realistic estimate on the use resources when the algorithm is executed on a real computer. In this paper we analyze some weaknesses of existing models of computation, namely sequential access machine and random access machine, and propose a new cost model, called relative cost random access machine, which solves some contradictions between these models. The new model actually only generalizes the way of counting the complexity, and includes sequential access machines and random access machines as special cases. At the same time, it is flexible enough to characterize the cost of memory access in current computers.

1 Introduction

In the first half of twentieth century, before the existence of computers, the concept of decidability became actual in mathematics. There was a need to define exactly, what is computation, and that gave rise to several models of computation, the best known of which is perhaps the Turing machine [6]. The pricipal elements in these models were the program that was characterized by a finite set of instructions, and the memory could grow without limit. The essential instructions were branching in the program depending on data in memory that was available at that moment, and moving in memory, i.e. making new memory locations immediately available and losing the immediate availability of others. In Turing machine, the memory is organized as a linear sequence, and in one step of computation, one can only move to the memory location next to the current location. Therefore, we call Turing's model a *sequential access machine*, SAM for short, in contrast with other memory access models to be defined later. The sequential access models makes, of course, an access to a random memory location slow, because the time needed to access it is proportional to the difference of the addressess of the current and the required memory locations.

*E-mail penttonen@cs.uku.fi

When computers became available and algorithm theory developed, there grew a need to model the computation more exactly so that the complexity of the algorithm should translate to the real running time of the program in the computer. The new model was called a *random access machine* (RAM), because all of the memory locations were more or less immediately accessible [5]. In the extreme case, in the *unit cost model*, the memory access is not charged at all, but all instructions take unit time only. The abuse of this feature may lead to unpleasant results, such as solving NP-complete problems in polynomial time [2]. A more realistic approach is to charge memory accesses by the sizes of the address and contents of the memory location, usually logarithmically. This prevents artificial coding of large amounts of data in a few memory locations, to be manipulated at unit cost. Polynomial time transformations between SAM and RAM computations are one of the cornerstones of the NP-completeness theory [1].

In the next section, we show some examples, where SAM (and respectively RAM) models well the computation, and where it doesn't. In many occasions, neither of the two models captures a right abstraction of the computation by a modern computer. We suggest a minor but significant extension for the definition of memory access. Our *relative cost* random access machine works otherwise like the traditional random access machine, but instructions are charged depending on the recent history of computation so that memory operations on a new memory area are costlier than those using the neighborhood of recently used memory locations.

2 SAM and RAM

A *sequential access machine* M consists of a finite *alphabet* A, m *registers* $r_1, \ldots, r_m$, k *memory tapes*, *pointers* $p_1, \ldots, p_k$ into the memory tapes, and a program P. A program is a sequence of labelled or unlabelled *instructions*, where each instruction is one of the following:

read r_i	read a symbol from input device and store it into register r_i
write r_i	write the symbol in register r_i to output device
$r_i := *p_j$	load the pointed symbol from tape j to register r_i
$*p_j := r_i$	store the symbol in register r_i to the pointed cell of tape j
if $r_i \diamond r_j$ **goto** l	if registers r_i and r_j fulfill the relation $\diamond$ ($=$ or $\neq$) then continue at instruction labelled with l otherwise continue at the next instruction
right i	move pointer on tape i one cell to the right
left i	move pointer on tape i one cell to the left

A *random access machine* M consists of an *input/output alphabet* A, m *registers* $r_1, \ldots, r_m$ containing (arbitrarily large) integers, a memory whose locations are indexed with addresses $1, 2, 3, \ldots$, a *pointer* p ($p \geq 1$) into the memory, and a program P. A program is a sequence of labelled or unlabelled *instructions*, where each instruction is one of the following:

read r_i	read a symbol from input device and store its integer code into register r_i
write r_i	write the symbol corresponding to the contents of register r_i to the output device
$r_i := *p$	load the contents of address p in register r_i
$*p := r_i$	store the contents of register r_i to the address p
if $r_i \diamond r_j$ **goto** l	if registers r_i and r_j fulfill the relation $\diamond$ ($=, \neq, <, >, \leq, \geq$) then continue at instruction labelled with l otherwise continue at the next instruction
$r_i := r_j \circ r_k$	apply operation $\circ$ ($+$ or $-$) to the contents of the registers r_j and r_k and store the result in r_i
$r_i := p$	store pointer (address) in the register r_i
$p := r_i$	load pointer p from the register r_i

The readability of the algorithms written for these machines can be improved by adding control structures such as **while do** and **for to** loops that can be translated to the language defined above.

The *cost* of computation of a sequential access machine is the number of steps taken before the program halts. The *unit cost* of computation of a random access machine is defined likewise as the number of steps. Under the *logarithmic cost* model of random access machine, the cost of the expression $*p := r$ is $l(v(r)) + l(v(p))$, and the cost of the expression $r := *p$ is $l(v(p)) + l(v(v(p)))$, where $v(x)$ refers to the numerical value that the register or pointer x contains, and $l(0) = l(1) = 1$, $l(y) = \lceil \log y \rceil$ for $y > 1$.

With the following examples we point out some strengths and weaknesses of these two models of computation.

Example 2.1 Palindromes by SAM. Test if an input string equals its reversal.

read r_1
while $r_1 \neq \emptyset$ **do**
  $*p_1 := r_1$; **right** 1
  $*p_2 := r_1$; **right** 2
  read r_1
left 1; $r_1 := *p_1$
while $r_1 \neq \emptyset$ **do**
  left 1; $r_1 := *p_1$
right 1; $r_1 := *p_1$; **left** 2; $r_2 := *p_2$
while $r_1 = r_2 \neq \emptyset$ **do**
  right 1; $r_1 := *p_1$; **left** 2; $r_2 := *p_2$
if $r_1 = r_2 = \emptyset$ **goto** yes

On input of length n the cost of recognition is about $2n$. The SAM algorithm can easily be modified to a RAM algorithm.

Example 2.2 Palindromes by RAM.

$r_1 := p := 1$; **read** r_3; $r_4 := r_3$
while $r_4 \neq \emptyset$ **do**
  $*p := r_4$; **inc** p
  read r_4
dec p; $r_2 = p$
while $r_3 = r_4 \neq \emptyset$ **do**
  inc r_1; $p := r_1; r_3 := *p$
  dec r_2; $p := r_2; r_4 := *p$
if $r_3 = r_4 = \emptyset$ **goto** yes

If unit cost is used, palindromes are recognized in linear time. However, if logarithmic cost is used, the time is $\Theta(n \log n)$, which is worse than the time required by SAM. Actually Schönhage [4] has proved a nonlinear lower bound for random access machines.

The following, somewhat artificial example shows, that by defining the problem context differently, one can get drastically different results.

Example 2.3 *Touching problem* is an abstraction of a database problem. Given n keys. For m times, choose a key at random and touch it.

It the problem is given on an input tape, a SAM can read constant length keys in time n. As keys are touched randomly, each touching may imply reading through all the data area, i.e. $\Theta(n)$ time, and thus all touches require $\Theta(mn)$ time.

By using unit cost RAM reading the input can also be done in time $O(n)$. In this case, each touch takes only $O(1)$ time, and total time is $\Theta(n + m)$.

By logarithmic cost, reading takes time $\Theta(n \log n)$, and each touching is done in $\Theta(\log n)$ time, giving the total of $\Theta((m + n) \log n)$.

Note that if reading time of an input is not counted, touching an individual key takes the time $\Theta(n)$, $\Theta(1)$, and $\Theta(\log n)$ in these models, respectively.

3 Relative Cost RAM

As the computation and cost models of the previous section give different results, one should ask, which is right and which is wrong. It should be remembered that the original role of Turing machine was to model algorithmic computability, it was not intended to be a tool of performance analysis. Random access model was to model computers, as they were invented in late fourties. As decades have passed, it is natural to ask, how exactly these models model performance of computations in current computers.

Referring to the previous section, SAM does well in the analysis of the palindrome problem. Unit cost RAM also also gave right complexity, but logarithmic cost RAM probably pays too much, as this kind of sequential access should be easy in all memory architectures. In the touching example, SAM is too expensive, unit cost RAM is too cheap, while logarithmic cost RAM probably matches best the reality. However, if the problem size grows larger than the main memory, the results lose validity.

The SAM favours *locality* in the memory access, assuming that the cost of access depends linearly on the difference of addresses. As the speed of processors has grown continuously, speed of light is already a non-negligible factor in memory access times. Hence, locality and sequential access should not be completely forgotten, and in future even less so. Common techniques to add locality have been to increase the number of *registers* (as in RISC processors), to use *caches* between the processor and the main memory, and *paging* between the main memory and the external memory. Even if these features of hardware make estimating the complexity of the algorithm more difficult, they should be used when appropriate, and hence included in the abstract model.

The RAM assumes a fast access to a uniform main memory. Unit cost RAM is unrealistic if there is no word size limit, because it would allow coding large subproblems in a single integer. Using explicit word sizes, on the other hand, would make algorithm writing and complexity analysis too complicated. Logarithmic cost RAM eliminates the need to use word size, but it ignores the locality. Strangely, low addresses are cheaper than high addresses, which would suggest concentrating the computation in low addresses. This is questionable in practice and probably just an anomaly of the model.

When considering the complexity of the memory access, at least the following factors are significant:

- *Overhead.* Each memory operation needs some kind of activation of the access mechanism. We may assume that it is constant, but probably this constant is higher than the time required by the internal operations of the processor. This cost appears even in the unit cost RAM.

- *Logical complexity.* A part in memory access is to recognize the target of the operation. If the hardware uses some kind of binary switching for opening the gates to the target location, we may assume that the cost is logarithmic to the address of the memory location. Logarithmic cost RAM emphasizes this complexity.

- *Volume.* Moving bigger data should be more expensive than moving smaller data. A natural measure is the number of bits, or the logarithmic cost of the contents. This complexity factor is included in the logarithmic cost RAM.

- *Distance.* Moving to a longer distance requires more time than moving to a short distance. In 3-dimensional world the physical distance is at least proportional to the cubic root of the number of memory locations. A simple estimate is the difference of addresses. Turing machine models the distance complexity.

One more feature of existing computers, *cache*, can be taken to the model. The idea of cache is based on locality. If some memory location is needed, it is likely that in near future more data in its neighborhood will be needed. Therefore, the whole neighborhood is transferred from the distant place (main or external memory) to a near place, cache.

Now the neighborhood will be easily available for some time. Therefore, the computation is history dependent.

We suggest the following *relative cost* function

$$Cost = \min\{c_1(a, a_{-1}), \ldots, c_k(a, a_{-k})\},$$

where

1. k is a constant referring to the number of memory pages the computer keeps in memory, and $a_{-1}, \ldots, a_{-k}$ are addresses accessed in k last memory operations
2. each $c_i(a, a_{-i})$ is of the form

   $$d_0 + d_1 \log|a - a_{-i}| + d_2\lfloor\frac{|a - a_{-i}|}{b}\rfloor$$

   where
3. d_0 is the access overhead required by the processor and memory chips
4. $d_1 \log|a - a_{-i}|$ describes the switching complexity of choosing the required address, and
5. $d_2\lfloor\frac{|a-a_{-i}|}{b}\rfloor$ estimates the complexity of fetching a block of size b, derived from the speed of light and the properties of the bus and the network of the computer

The formula with all components is unmanageable, and therefore only the most significant terms should be used. In case $k = 1$ we have the following subcases:

- If $d_0 = 1$, $d_1 = d_2 = 0$, we have unit cost RAM.
- If $d_1 = 1$, $d_0 = d_2 = 0$, we have logarithmic cost RAM.
- If $d_2 = b = 1$, $d_0 = d_1 = 0$, we have SAM.

By the definition of the cost function, it is obvious that the use of memory is an important factor in the efficiency of a program. It the program is memory intensive, the latency of memory access is wasted time, and there may be no method to avoid this inefficiency. However, parallelism may help. If there is a parallel algorithm for the problem, *slackness* [7] may help. If the cost of a memory access is c and at least c subcomputations can be started independently, useless waiting can be avoided. Actually, the same idea is applied is smaller case in the processor level parallelism, such as pipelining or VLIW (very long instruction word) architectures [3].

4 Conclusions

Choosing a model of computing is difficult play of balancing between simplicity and exactness. If model is not simple, it cannot be used in algorithm design. If it is not exact, it is useless because it does not help to forecast the performance of the program.

We propose a cost model, or a framework of cost models, that generalizes the well-known sequential access and random access models, and allows for more exact modeling of real computations. The essential feature of the new model is the relativization of the access. At simplest, the model is not more complex than SAM or RAM, but with some more detail it makes possible to analyse more exactly algorithms to be run in a computer with multiple registers, or paging.

Acknowledgement

This article was motivated by long discussions with Jyrki Katajainen. Many of these ideas were also discussed in our parallelism research group. I wish to thank Juha Hakkarainen for a proposal about including the paging in the model.

References

[1] Cook, S.: "The complexity of theorem proving procedures". *Proc. 3rd Annual ACM Symposium on Theory of Computing*, 151-158.

[2] Y. Feldman and E. Shapiro, Spatial machines: Towards a more realistic approach to parallel computation, Technical Report CS88-05, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, 1988. See also *Communications of the ACM*, Volume 35, Number 10, pages 61–73, 1992.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Francisco, 1994.

[4] A. Schönhage, A nonlinear lower bound for random-access machines under logarithmic cost, *Journal of the ACM* 35: 748–754 (1988).

[5] J. C. Shepherdson and H. E. Sturgis, Computability of recursive functions, *Journal of the ACM*, Volume 10, pages 217–255, 1963.

[6] A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Series 2, Volume 42, pages 230–265, 1937. A Correction, *ibidem*, Volume 43, pages 544–546, 1937. Also in [**?**, pages 115–154].

[7] L. G. Valiant, General purpose parallel architectures, J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Volume A: *Algorithms and Complexity*, Elsevier, Amsterdam/New York/Oxford/Tokyo, pages 943–971, 1990.