FAST SIMULATION OF TURING MACHINES BY RANDOM ACCESS MACHINES

J.Katajainen, J. van Leeuwen, and M.Penttonen

RUU-CS-85-7
March 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

FAST SIMULATION OF TURING MACHINES BY RANDOM ACCESS MACHINES

J.Katajainen, J. van Leeuwen, and M.Penttonen

Technical Report RUU-CS-85-7
March 1985

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands

FAST SIMULATION OF TURING MACHINES BY RANDOM ACCESS MACHINES

Jyrki Katajainen*, Jan van Leeuwen**, and Martti Penttonen*

Abstract. We prove that a T(n) time-bounded, S(n) space-bounded Turing machine can be simulated in O(T(n) loglog S(n)) time by a random access machine (with no multiplication or division instructions) under the logarithmic cost criterion.

Keywords and Phrases: Turing machine, random access machine, simulation, computational complexity.

* Authors' address:
Department of Computer Science
University of Turku
20500 Turku, FINLAND

** Author's address:
Department of Computer Science
University of Utrecht
P.O. Box 80012
3508 TA Utrecht, THE NETHERLANDS

1. Introduction

In the theory of computation one uses both the Turing Machine (TM) and the Random Access Machine (RAM) as standard models of effective computing (see e.g. [AHU]). Whereas the models are vastly different in detail, it is well-known that the machines are "equivalent" in computational strength. More precisely, one can show that the machines are polynomially related in the sense of computational complexity theory (see [CR] or [AHU, Section 1.7]): a TM can simulate a RAM in $O(T(n)^2)$ time and a RAM can simulate a TM in $O(T(n) \log T(n))$ time, where $T(n)$ is the time complexity of the simulated machine and RAMs are assumed to use the so-called logarithmic cost criterion. In the result, RAMs are assumed without explicit "single" multiplication or division instructions in their instruction set. Slot and van Emde Boas [SE] have shown that TMs and RAMs can simulate one another within only a constant factor of extra space.

Several studies have attemted to refine or lower the simulation costs between the two models, especially for the case of simulating RAMs by TMs (see e.g. Wiedermann [W] for some recent results). In this paper we consider the efficient simulation of TMs by RAMs. Let $T(n)$ denote the time complexity, $S(n)$ the space complexity, and $U(n)$ an upperbound on the length of the longest output on inputs of length n. The following results are known.

Theorem A (Folklore, see e.g. [AHU, Section 1.7]) A TM can be simulated in $O(T(n) \log S(n))$ time by a RAM (with no multiplication or division instructions) under the logarithmic cost criterion.

Theorem B (Paul [P, Section 3.3]) A TM can be simulated in $O(n \log n + U(n) \log T(n) + T(n))$ time by a RAM (with no multiplication or division instructions) under the logarithmic cost criterion.

Theorem B shows that TMs can be simulated by RAMs with no essential time-loss provided $T(n) \geq n \log n$ and $U(n) \leq T(n)/\log T(n)$. (Note that [P, Section 3.3] assumes RAMs with shift instructions.) In this paper we improve Theorem A to

Theorem C. A TM can be simulated in $O(T(n) \operatorname{loglog} S(n))$ time by a RAM (with no multiplication or division instructions) under the logarithmic cost criterion.

In fact we can also improve Theorem B to time bound $O((n + U(n)) \operatorname{loglog} S(n) + T(n))$, again without multiplication and division. We will prove this result in a future version of this paper. None of the results

assume that T(n), S(n), or U(n) are constructible.

As an example of the use of Theorem C we mention the following corollary.

Corollary D. Any linear time TM can be simulated in O(n loglog n) logarithmic time by a RAM (with no multiplication or division instructions).

It follows that e.g. the reversal of a string of n inputs can be output by a RAM in O(n loglog n) units of logarithmic time. We can apply the above corollary also to the string-matching problem, where the task is to find all occurrences of a given pattern of length m from the text of length n, $m \leq n$. The string-matching can be done in O(n) time on a TM as shown by Fischer and Paterson [FP] (see also [GS]), and therefore in O(n loglog n) units of logarithmic time on a RAM.

The paper is organized as follows. In Section 2 we recapitulate some basic definitions. In Section 3 we develop the proof of Theorem C through a number of stages.

2. Machine models

We define TMs and RAMs such that they appear as instances of the same abstract model, following the guidelines of [S]. The machines have very similar input, output, and control structures but differ in the structure and the use of the memory. The definition of TMs and RAMs is included to fix the particular instruction sets.

2.1. Turing machines

We describe the "parts" of a Turing machine without much formal notation. We assume that the input, output, and work-tape alphabet is {0,1} and refer to the individual symbols as bits. A (multitape) TM consists of the following parts (compare [AHU, Section 1.6]):

(i) a one-way read-only input tape, containing a bit string followed by an endmarker #.

(ii) a one-way write-only output tape, where a bit string will be written.

(iii) k two-way read-write work-tapes ("memory"), containing bits in successive memory cells. The tapes are two-way infinite. On each tape there is a separate read-write head that can be activated for reading, writing, or moving one tape-cell to the left or to the right.

(iv) a TM program, which is a finite sequence of labelled or unlabelled instructions from a fixed instruction set (see below). No two instructions should carry the same label.

We shall identify "similar" parts for a RAM in Section 2.2. The instruction set of a TM contains eight instruction types:

(1) **input** $\lambda_0, \lambda_1, \lambda_\#$: causes a "next" input symbol to be read, and the input head moves one cell to the right (except on #). Depending on whether is 0,1, or # control is transferred to the instruction with label λ_0, λ_1, $\lambda_\#$.

(2) **output** β : causes a bit β to be output, and the output head moves one cell to the right.

(3) **jump** λ : transfers control to the instruction with label λ.

(4) **halt** : halts the program.

(5) **head** i : activates the read-write head on the i'th work-tape ($1 \leq i \leq k$). Only one read-write head will be active at a time.

(6) **write** β : causes a bit β to be written in the tape-cell designated by the active read-write head.

(7) **branch** λ_0, λ_1 : causes the bit β to be read from the tape-cell designated by the active head. Depending on whether β is 0 or 1 control is transfered to the instruction with label λ_0 or λ_1.

(8) **move** δ (with $\delta \in \{L, R\}$) : moves the active read-write head one cell to the left of to the right depending on whether δ is L or R.

We assume that initially all work-tapes contain 0 in every cell, and that head 1 is active. The computation starts from the first instruction and thereafter the instructions of a program are executed in their successive order unless a jump instruction orders otherwise.

The time complexity T(n) of a TM is the largest number of instructions executed in halting computations on inputs of length n. The space complexity S(n) is the largest number of cells occupied on any work-tape in halting computations on inputs of length n. The output complexity U(n) is the length of the longest output produced in halting computations on inputs of lenght n.

Because a TM with a two-way infinite tape can be simulated by a TM with a one-way infinite tape in real time (see e.g. [HU, Section 7.5]), we shall

assume that the work-tapes of a TM are one-way infinite, say infinite to the right. Initially all read-write heads are positioned on the leftmost cell of their work-tape. By the standard construction used in the above simulation [HU, Section 7.5], we can further assume that a read-write head is never moved off the left end of the work-tape. (Thus the computation is stopped by a **halt** instruction, not by the fall of a read-write head.) Although in the construction the tape alphabet is enlarged, it is straightforward to return into the binary alphabet (see also [HU, Section 7.8]).

2.2. Random access machines

In describing the "parts" of a random access machine, we only emphasize the parts that are different from a TM in their implementation. Parts (i) and (ii) are very similar for a RAM but instead of (iii) one has the following set-up (compare [AHU, Section 1.2]):

(iii') a special register called the accumulator (AC) and a countable sequence of ordinary registers ("memory") indexed by the nonnegative integers (used as addresses). Each register can hold an arbitrary nonnegative integer in binary notation. Only data stored in the AC can be operated upon.

The contents of register j is denoted by ⟨j⟩. Clearly part (iv) is a RAM program, but the instruction set of a RAM differs from the instruction set of a TM. The instruction set of a RAM contains twelve instruction types:

(1')-(4') : similar to the instructions (1)-(4) of a TM.

(5') **jzero** λ : transfers control to the instruction with label λ if ⟨AC⟩=0, and continues to next instruction otherwise.

(6') **load** =j : load the integer j into the AC.

(7') **load** j : load ⟨j⟩ into the AC.

(8') **load** *j : load ⟨⟨j⟩⟩ into the AC ("indirect addressing").

(9') **store** j : stores ⟨AC⟩ into register j.

(10') **store** *j : stores ⟨AC⟩ into register ⟨j⟩.

(11') **add** j : adds ⟨j⟩ to the current value in the AC.

(12') **sub** j : subtracts ⟨j⟩ from the current value in the AC.

We assume that all registers, including the AC, initially contain 0. Memory need not be used contiguously.

We do not simply count the number of instructions executed in a RAM program but use the so-called logarithmic cost criterion: the "time" charged for an instruction is equal to the sum of the sizes (in bits) of the integers (addresses and data) involved in its execution. Note that the size of a positive integer m is $\lceil \log(m+1) \rceil \sim \log m$, and the size of zero is 1. The time complexity T(n) of a RAM is the largest amount of time, measured according to the logarithmic cost criterion, used in halting computations on inputs of length n. See Slot and van Emde Boas [SE] for notions of space complexity for RAMs.

It will be convenient to use various extensions to the basic RAM instruction set, provided that the execution time is adequately measured by the logarithmic cost criterion. (This is the case for e.g. comparison instructions [S, pp. 495-496], but not for multiplication or division.) Also in some algorithms it is convenient to have a RAM with k separate memories (or arrays as called by Cook and Reckhow [CR]), $k>1$, each consisting of a countable sequence of registers indexed $0, 1, 2, \ldots$. We call this a "multimemory" RAM.

Lemma 2.2.1. Every T(n) time-bounded multimemory RAM can be simulated in $O(T(n))$ time by an ordinary RAM.

Proof. The technique was essentially given in [CR]. The idea is simply to interleave the RAM memories into one, using addresses $i+kj-1$ for register j of the i'th memory ($1 \leq i \leq k$, $j \geq 0$). Translating addresses costs an overhead of a factor in the order of k, which is a constant.

#

Instead of writing RAM programs with the original instruction set, we shall freely use Pascal-like control structures and notations for greater readibility.

3. The simulation of a TM by a RAM

Consider a T(n) time-bounded, S(n) space-bounded TM. The simple idea underlying Theorem A is to represent the cells of the work-tapes in consecutive registers of a RAM, with additional registers containing current read-write head positions. Every step of the TM is easily simulated in $O(\log S(n))$ time on a RAM, assuming the logarithmic cost criterion. In the simulation underlying

same registers. The neighbouring blocks are taken along in order to guarantee that in all cases b simulation steps can be taken staying in the unpacked zone. This unpacked 3b bit zone is kept in a memory, called the window.

The simulation of the single TM instructions is quite obvious, it is done like in the proof of Theorem A. However, we need a mechanism by which it is possible to simulate one instruction of the TM at a time. If we first number the instructions of the TM program, it is easy to construct a program P(λ, head), which simulates the λ'th instruction in the window, and which also updates the head position and the label λ ready for the further processing. Now we can represent the simulation in the form of a RAM program as follows:

```
(*)   procedure simulate
      {Suppose that the block size b is given.}
      λ:= 1
      activeblock:= 1   {the left neighbour of the first block is kept empty}
      head:= the first address of the middle block of the window
      loop   {until a halt instruction in the simulation}
         loadwindow(activeblock, b)
         for b times do P(λ, head)
         storewindow(activeblock, b)
         if head moved to neighbour then update head and activeblock addresses
```

The procedure loadwindow fetches the contents of the active block and its neighbours into low-indexed registers, and unpacks the b-bit integers. The procedure storewindow packs the window blocks, and stores them by overwriting their older copies.

We will now attack the problem of packing and unpacking the blocks efficiently. As our RAM model does not include division or shift instructions, we have to invent another method for finding the bit representation of a number and vice versa. We will see that unpacking and packing can be done efficiently with precomputed tables.

As a first attempt one could decode numbers to bit-strings by building a table that gives the decoding directly. For example the table could contain the b bits of a number n ($< 2^b$) in the registers $nb, nb+1, \ldots, nb+b-1$. A disadvantage of this method is that, while bits are obtained directly, the access of them may cost $O(\log n)$. For this reason loading a window takes $O(b^2)$ time. By a similar analysis as what follows, one can see that this would give $O(T\sqrt{\log S})$ simulation algorithm. However, we can do unpacking and packing in $O(b \log b)$ time.

The efficient decoding of a number to its bit representation and vise versa is based on a divide-and-conquer strategy with precomputed shift tables. We will first build the necessary tables and then give the unpacking and packing algorithms.

We assume that each table is stored in its own memory. We will need tables lshift, rshift, origin, and power. By lshift(i), rshift(i),... we denote the contents of the register i reserved for lshift, rshift,... .

The tables lshift and rshift in Figure 1 contain as subtables shift tables for 1-bit numbers, 2-bit numbers, 4-bit numbers etc. The divide-and-conquer strategy implies that b-bit numbers are shifted b/2 bits to the right or b bits to the left. The entries of the tables are numbers rather than bit strings. Thus for example the number $75 = 01001011_2$ has the right shift $4 = 0100_2$, and $4 = 0100_2$ has the left shift $64 = 01000000_2$. The origin table expresses where subtables begin: The shift tables for 2^i-bit numbers begin at origin(i).

origin:

register	0	1	2	3	4	...	i
contents	0	2	6	22	278		$2^{2^0}+2^{2^1}+\ldots+2^{2^{i-1}}$

rshift:

register	0	1	2	3	4	5	6	7	8	...	21	...	origin(i)+j
block size			2 bits				4 bits						2^i bits
block value			0	1	2	3	0	1	2		15		j
contents	0	0	0	0	1	1	0	0	0		3		$j \text{ div } 2^{2^{i-1}}$

lshift:

register	0	1	2	3	4	5	6	7	8	...	21	...	origin(i)+j
block size	1 bit		2 bits				4 bits						2^i bits
block value	0	1	0	1	2	3	0	1	2		15		j
contents	0	2	0	4	8	12	0	16	32		240		$j \cdot 2^{2^i}$

Figure 1. The origin, rshift, and lshift tables.

We have to first analyze how much the building of the tables costs.

Lemma 3.1.1. The tables origin, rshift, and lshift up to block size b ($= 2^k$) can be built in logarithmic time $O(b\,2^b)$.

Proof. Assuming that the values origin(i-2) and origin(i-1) are already computed, the following program will compute the i'th origin value, $i \geq 2$.

```
procedure build origin(i)
t:= origin(i-1) - origin(i-2)     {t:= 2^(2^(i-2))}
origin(i):= origin(i-1)
for t times do origin(i):= origin(i) + t
```

Clearly, the time complexity of this program is $O(2^{2^{i-2}} \cdot 2^{i-1})$.

When constructing the i'th rshift and lshift subtables we can use the origin values for i-1, i, and i+1.

```
procedure build rshift(i)
j:= origin(i); x:= 0; t:= origin(i) - origin(i-1)     {t:= 2^(2^(i-1))}
for t times do
   for t times do rshift(j):= x; j:= j+1
   x:= x+1

procedure build lshift(i)
j:= origin(i); x:= 0; t:= origin(i+1) - origin(i)     {t:= 2^(2^i)}
for t times do lshift(j):= x; j:= j+1; x:= x+t
```

The execution of the both procedures requires $O(2^{2^i} \cdot 2^i)$ time. Thus the tables up to k can be constructed in time $O(\sum_{i=1}^{k} 2^{2^i} \cdot 2^i)$, which is $O(2^k \cdot 2^{2^k})$, or in terms of b, $O(b\, 2^b)$.

∎

We also need powers of 2 for unpacking numbers to bit strings, and for packing bit strings to numbers. It is useful to precompute also them in the table power.

Lemma 3.1.2. The table power(i) = 2^i up to k'th power can be build in $O(k^2)$ logarithmic time.

Proof. A new power can be computed by doubling the previous one by addition. This method gives the time bound $O(k^2)$.

∎

Now we are ready to present the unpacking and packing algorithms.

Lemma 3.1.3. Assuming that the tables lshift, rshift, origin, and power up to the block size b are available, it is possible to compute the b-bit representation of an integer $n < 2^b$, and the numeric value of a b-bit string, both in $O(b \log b)$ time.

Proof. The procedure unpack(n, j, a) unpacks a number $n < 2^{2^j}$ to its 2^j-bit representation beginning at the a'th register of the window. The procedure is as follows:

```
procedure unpack(n, j, a)
if j=0 then window(a) := n
else n1 := rshift(origin(j)+n); n2 := n - lshift(origin(j-1)+n1)
     unpack(n1, j-1, a); unpack(n2, j-1, a+power(j-1))
```

For clarity, we have written the algorithm in recursive form. The recursion can be eliminated by using one memory as a stack where the second recursive call is stored while the first is executed. While unpacking a number $n < 2^j$ there are never more than log j calls in the stack. In order to balance access cost it is economical to initiate the stack at the address log b and let it grow downwards. If we denote by t(x) the logarithmic time of unpacking an x-bit number, by analyzing the program we get

$$t(1) = k_1 \log b$$

$$t(x) = 2\,t(x/2) + k_2 x + k_3 \log b$$

which gives $t(b) = O(b \log b)$.

The procedure pack(a, j, n) computes the numeric value of the bit string window(a), window(a+1), ..., window(a+2^j-1). Also it is written recursively:

```
procedure pack(a, j, n)
if j=0 then n := window(a)
else pack(a, j-1, n1); pack(a+power(j-1), j-1, n2)
     n := lshift(origin(j-1)+n1) + n2
```

The recursion is controlled as in unpacking. Also the time analysis is analogous.

∎

We can now obtain a preliminary version of Theorem C:

Theorem 3.1.4. Assuming that n and S(n) are known, a T(n) time-bounded, S(n) space-bounded TM can be simulated in O(T(n) loglog S(n)) logarithmic time by a RAM.

Proof. The total time of the simulation (*) is bounded by

$$T_{RAM} = b\,2^b + T/b\,(\log S + b \log b + b \log b + b \log b + \log S)$$

where $b\,2^b$ is needed for the construction of the tables, T/b is the number of

growth of the series $\{S_i\}$ implies that all reblockings can be done in time $O(S)$.

For the final analysis of the simulation, assume that T_i steps of the TM are simulated using block size b_i. Hence the total time of the simulation is bounded by

$$\sum_{i=0}^{\log\log S} T_i/b_i(\log S_i + b_i \log b_i) = \sum_{i=0}^{\log\log S} T_i \log b_i \leq \log b \sum_{i=0}^{\log\log S} T_i = T \log b = T \log\log S.$$

Hence, the time bound $O(T \log\log S)$ holds also for the dynamic simulation. We have proved

Theorem 3.2.1. A $T(n)$ time-bounded and $S(n)$ space-bounded TM can be simulated in $O(T(n) \log\log S(n))$ time on a RAM without multiplication and division instructions.

References

[AHU] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, Massachusetts, 1974).

[CR] S.A. Cook, and R.A. Reckhow, Time bounded random access machines, Journal of Computer and System Sciences 7 (1973) 354-375.

[D] H.M. Deitel, An Introduction to Operating Systems (Addison-Wesley, Reading, Massachusetts, 1984).

[FP] M.J. Fischer, and M.S. Paterson, String-matching and other products, in: R.M. Karp, ed., Complexity of Computation, SIAM-AMS Proceedings 7 (American Mathematical Society, Providence, Rhode Island, 1974) 113-125.

[GS] G. Galil, and J. Seiferas, Time-space-optimal string matching, in: Proceedings of the 13'th Annual ACM Symposium on Theory of Computing (ACM, New York, 1981) 106-113.

[HU] J.E. Hopcroft, and J.D. Ullman: Introduction to Automata Theory, Languages, and Computation (Addison-Wesley, Reading, Massachusetts, 1979).

[O] M.H. Overmars, The Design of Dynamic Data Structures, Lecture Notes in Computer Science 156 (Springer-Verlag, Berlin 1983).

[P] W.J. Paul, Komplexitätstheorie (Teubner Verlag, Stuttgart, 1978).

[PK] M. Penttonen, and J. Katajainen, Notes on the complexity of sorting in abstract machines, BIT to appear.

[S] A. Schönhage, Storage modification machines, SIAM Journal on Computing 9 (1980) 490-508.

[SE] C. Slot, and P. van Emde Boas, On tape versus core: an application of space efficient hash functions to the invariance of space, in: Proceedings of the 16'th Annual ACM Symposium on Theory of Computing (ACM, New York, 1984) 391-400.

[W] J. Wiedermann, Deterministic and nondeterministic simulation of the RAM by the Turing machine, in: R.E.A. Mason, ed., Proceedings of the IFIP Congress 83 (North-Holland, Amsterdam, 1983) 163-168.