Information Processing Letters 24 (1987) 363–367
North-Holland

6 April 1987

AN IMPROVED SIMULATION OF SPACE AND REVERSAL BOUNDED DETERMINISTIC TURING MACHINES BY WIDTH AND DEPTH BOUNDED UNIFORM CIRCUITS

Ian PARBERRY

Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, U.S.A.

Communicated by David Gries
Received 12 May 1986
Revised 4 August 1986

We present an improved simulation of space and reversal bounded Turing machines by width and depth bounded uniform circuits. (All resource bounds hold simultaneously.) An S(n) space, R(n) reversal bounded deterministic k-tape Turing machine can be simulated by a uniform circuit of $O(R(n)\log^2 S(n))$ depth and $O(S(n)^k)$ width. Our proof is cleaner, and has slightly better resource bounds than the original proof due to Pippenger (1979). The improvement in resource bounds comes primarily from the use of a shared-memory machine instead of an oblivious Turing machine, and the concept of a 'special situation'.

Keywords: Deterministic k-tape Turing machine, shared-memory machine, uniform circuit, reversal, NC, extended parallel computation thesis, reduction to sorting, simultaneous resource bounds

1. Introduction

Pippenger [16] has demonstrated that NC (the class of languages recognized by polynomial size, polylog depth uniform circuits) is precisely the class of languages recognized by polynomial space, polylog reversal deterministic Turing machines. We assume the reader to be familiar with the terminology of that paper, and for conciseness will restrict ourselves mainly to those definitions which differ for technical reasons. Pippenger's simulation of Turing machines by circuits makes frequent use of a reduction to sorting. We obtain a more straightforward proof, with a slight improvement in resource bounds, using a single reduction.

A preliminary version of the results in this paper have appeared in [14], and in the author's Ph.D. thesis [13]. The reader requiring a more detailed account is directed to the latter reference. The remainder of this paper is divided into two sections, the first containing some definitions, and the second containing the result.

2. Some machine models

In this section we sketch two popular parallel machine models. Let $\mathbb{Z}$ denote the set of integers and $\mathbb{N}$ denote the set of natural numbers. Let $D, P, S, T, W, Z : \mathbb{N} \to \mathbb{N}$.

A *shared-memory machine* consists of an infinite number of processors attached to a globally accessible shared memory. Each processor possesses an infinite number of general purpose registers, and a unique read-only processor identity register PID which is preset to i in the *ith* processor, $i \in \mathbb{N}$. A *program* for this machine consists of a finite list of instructions; each instruction is of the form either:

(i) read a value from the shared memory,
(ii) write a value to the shared memory,
(iii) perform an internal computation,
(iv) conditional transfer of control or halt.

The allowable internal computations usually consist of direct and indirect register transfers, and arithmetic operations. We will allow the arith-

metic operations of addition, subtraction and multiplication and division by powers of two.

More formally, a shared-memory machine is specified by a program $\mathscr{P}$ and a processor bound P(n). A computation proceeds roughly as follows. An input of size n (where the 'size' measure depends on the problem in question) is broken up into n unit-size pieces, and the *ith* piece is encoded as an integer and stored in shared memory location i, $0 \leqslant i < n$. All other memory locations and general purpose registers are set to zero. Processors $0, 1, \ldots, P(n) - 1$ are activated simultaneously; they synchronously execute the program $\mathscr{P}$. When all processors have halted, the output is to be found in some specified place in the shared memory.

The *processor* bound P(n) is a measure of the number of processors used as a function of input size. The *space* S(n) is the maximum number of nonzero entries in the shared memory and registers at any time during the computation. (Note that this includes the input and the processor identity registers, so $S(n) \geqslant n + P(n) - 1$.) The machine is said to have *word-size* W(n) if every value placed into a register or shared memory location during a computation on an input of size n has absolute value less than $2^{W(n)}$. The *time* bound T(n) is the number of instructions executed before all processors have halted, again as a function of input size.

Variants of this model have appeared, for example, in [9,10,11,12,18,19,20,21,22]. We assume some reasonable protocol for dealing with memory access conflicts, as in those references. The general consensus of opinion is that whilst the shared-memory model is a powerful theoretical tool, it is not feasibly buildable using any foreseeable technology. Note that our space bound is slightly unusual. It is more normal to measure the 'number of registers or memory locations used' regardless of the amount of time during which they are active (see, for example, [1] for the case of a single RAM).

A *uniform circuit* is an infinite family $C = (C_0, C_1, \ldots)$ of combinational circuits, one for each input size (see, for example, [5,6,16,17]). Without loss of generality we assume that the circuits are built using gates which realize functions drawn from the class B_2 of two-input Boolean functions. An input of size n is presented, in some suitably encoded form, to the inputs of C_n. The output of C_n is then taken as the output of C. C is said to have *depth* D(n) if the length of the longest path from an input to an output in C_n is at most D(n), for $n \geqslant 0$. It has *width* W(n) if C_n has width (as defined in [16]) W(n) and *size* Z(n) if C_n has Z(n) gates. We assume $D(n) = \Omega(\log W(n))$.

The function $f: \mathbb{N}^2 \times \{\text{left, right}\} \to \mathbb{N}$, where for $n \geqslant 0$ the j-input of gate $i \geqslant n$ is connected to the output of gate f(i, n, j) (we assume that gates $0, 1, \ldots, n - 1$ are distinguished gates representing the inputs), is called the *interconnection function* of C. The function $g: \mathbb{N}^2 \to B_2$, where for $n \geqslant 0$ gate $i \geqslant n$ of C_n is a g(i, n)-gate, is called the *gate function* of C. We insist that the interconnection and gate functions be computable by a deterministic Turing machine in O(log Z(n)) space.

2.1. Lemma. *Every shared-memory machine which uses* S(n) *space,* T(n) *time and* W(n) *word-size can be simulated by a uniform circuit of* O(T(n) log S(n)) *depth and* O(S(n)W(n)) *width.*

Proof. The proof uses the common method of 'reduction to sorting' (see, for example, [13,14,15]), and the sorting network of Ajtai, Komlós and Szemerédi [2,3]. □

3. The simulation theorem

The extended parallel computation thesis of Dymond [7] (see also [8]) states that parallel time and hardware (on any 'reasonable' model) are simultaneously polynomially equivalent to reversals and space on a deterministic Turing machine. The justification for this is based on the seminal paper by Pippenger [16] which relates depth and size of uniform circuits to Turing machine reversals and time. Dymond prefers to use Turing machine space instead of time, and circuit width as a measure of hardware (rather than size) since it is a measure of the amount of hardware which comes into play at any given instant in time. The aim of this section is to provide an improved simulation of space and reversal bounded Turing

machines by width and depth bounded uniform circuits.

We follow the general structure of the proof appearing in [16]. Pippenger simulates a Turing machine on an oblivious Turing machine, and then simulates this on a uniform circuit. We will simulate a Turing machine on a shared-memory machine. We can then build a uniform circuit by application of Lemma 2.1.

3.1. Theorem. *An* S(n) *space*, R(n) *reversal bounded* k-*tape deterministic Turing machine can be simulated on a shared-memory machine with* $O(S(n)^k/\log S(n))$ *processors and space*, $O(R(n) \log S(n))$ *time, and* $O(\log S(n))$ *word-size.*

Proof. Let M be a k-tape deterministic Turing machine which runs in S(n) space and R(n) reversals. Following [16] define a *phase* to be all the steps of M from one reversal to the next (the first move is counted as a reversal for this purpose), and a *situation* to be the control state and head positions of M. It may be assumed that all transition rules of M which write a new value onto a tape cell also move the head away from that cell. This implies that symbols written during one phase cannot be read until the next. Let $d(n) = 2^{\lceil \log \log S(n) \rceil}$ and call a situation *special* if it has at least one head on the $(i \cdot d(n))$th cell of its tape, for some $i \in \mathbb{N}$. Note that there are at most $O(S(n)^k/\log S(n))$ special situations, and that at most $O(\log S(n))$ steps of M can occur between special situations.

The simulation proceeds roughly as follows. The tape contents at the start of the current phase, the head directions and the initial situation for the current phase are stored in the shared memory. This is easy to do at the start of the initial phase; the algorithm will maintain this information from phase to phase. We reserve one processor (and two shared-memory locations) for each special situation. The aim is to have these processors confer, via the shared memory, and decide which special situations are involved in the current phase. The processors corresponding to these special situations then simultaneously update the tape cell contents in shared memory; the final situation (which is detected by an attempted reversal) determines the head directions and the initial situation for the next phase. This proceeds for a total of R(n) phases.

The simulation of a phase is achieved as follows. Processor i handles the *ith* special situation. Firstly, each processor i computes in parallel the special situation which follows from special situation i, by doing a step-by-step read-only simulation of M on the tape contents in shared memory (by 'read-only simulation' we mean that the tape-contents are not updated). This value is stored into array element s[i] in shared memory. If an illegal situation occurs during this process, or a reversal is detected (determined by examining the head directions for the current phase, which are stored in shared memory), then s[i] is set to i. All processors i execute the following code synchronously in parallel. Before activation, shared array element active[i] is set to true iff situation i is the first situation in the current phase. Upon termination, active[i] will be true iff special situation i occurs in the current phase. Each processor can determine whether its special situation is the first special situation to occur in the current phase by using a step-wise read-only simulation of M starting at the initial situation of the phase.

```
for b := 1 to ⌈log S(n)⌉ do
  if active[i] then active[s[i]] := true
  s[i] := s[s[i]]
```

Suppose that s_0 is the initial situation of the current phase, and that the subsequent situations in the current phase are $s_1, \ldots, s_{S(n)-1}$. It can easily be proved by induction on m that after the m*th* interation of the **for**-loop, active[s_j] is true for $0 \leqslant j < 2^m$, active[i] is false for all other situations i, and

$$s[s_j] = \begin{cases} s_{j+2^m} & \text{for } 0 \leqslant j < S(n) - 2^m, \\ s_j & \text{for } S(n) - 2^m \leqslant j < S(n). \end{cases}$$

Those processors i with active[i] = true can then update the tape contents; the last special situation is readily available (in all entries of s), from which the final situation of the current phase can be determined. The running time is dominated by $O(\log S(n))$ for each phase. This comes from:

(1) decoding of PIDs (each of O(log S(n)) bits) into special situations,

(2) determining the first special situation from the initial situation and the final situation from the last special situation by simulating at most O(log S(n)) steps of M,

(3) computing the special-situation transition function by simulating O(log S(n)) steps of M,

(4) computing the active array in O(log S(n)) steps,

(5) updating the tape contents by simulating O(log S(n)) steps of M.

Repeating this for R(n) phases gives us the required result. □

3.2. Corollary. *An* S(n) *space,* R(n) *reversal bounded deterministic* k*-tape Turing machine can be simulated by a uniform circuit of* $O(R(n)\log^2 S(n))$ *depth and* $O(S(n)^k)$ *width.*

Proof. The proof follows from Theorem 3.1 and Lemma 2.1. □

3.3. Corollary. *A* T(n) *time,* R(n) *reversal bounded deterministic* k*-tape Turing machine can be simulated by a uniform circuit of* $O(R(n)\log^2 T(n))$ *depth and* $O(R(n)T(n)^k \log^2 T(n))$ *size.*

This is a small improvement over the original results of Pippenger [16], whose size and depth bounds are inferior by a factor of $O(\log^2 T(n))$. Our improvements can be summarized as follows:

(i) A factor of O(log T(n)) is easily removed from both the size and the depth by using the sorting network of Ajtai, Komlós and Szemerédi in place of the sorting network (due to Batcher [4]) used by Pippenger.

(ii) A factor of O(log T(n)) is removed from the depth by using shared-memory machines instead of oblivious Turing machines.

(iii) A factor of O(log T(n)) is removed from the size by using 'special situations'.

Acknowledgment

The author wishes to thank Mike Paterson for a suggestion which led to the concept of a 'special situation'.

References

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

[2] M. Ajtai, J. Komlós and E. Szemerédi, An $O(n \log n)$ sorting network, Proc. 15th Ann. ACM Symp. on Theory of Computing, Boston, MA (1983) 1–9.

[3] M. Ajtai, J. Komlós and E. Szemerédi, Sorting in $c \log n$ parallel steps, Combinatorica 3 (1983) 1–48.

[4] K.E. Batcher, Sorting networks and their applications, Proc. AFIPS Spring Joint Computer Conf., Vol. 32 (1968) 307–314.

[5] A. Borodin, On relating time and space to size and depth, SIAM J. Comput. 6 (4) (1977) 733–744.

[6] S.A. Cook, Deterministic CFL's are accepted simultaneously in polynomial time and log squared space, Proc. 11th Ann. ACM Symp. on Theory of Computing (1979) 338–345.

[7] P.W. Dymond, Simultaneous resource bonds and parallel computations, Ph.D. Thesis, Tech. Rept. TR145/80, Dept. of Computer Science, Univ. of Toronto, 1980.

[8] P.W. Dymond and S.A. Cook, Hardware complexity and parallel computation, Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science (1980) 360–372.

[9] S. Fortune and J. Wyllie, Parallelism in random access machines, Proc. 10th Ann. ACM Symp. on Theory of Computing (1978) 114–118.

[10] Z. Galil and W.J. Paul, An efficient general purpose parallel computer, J. ACM 30 (2) (1983) 360–387.

[11] L.M. Goldschlager, A universal interconnection pattern for parallel computers, J. ACM 29 (4) (1982) 1073–1086.

[12] D. Nassimi and S. Sahni, Parallel permutation and sorting algorithms and a new generalized connection network, J. ACM 29 (3) (1982) 642–667.

[13] I. Parberry, A complexity theory of parallel computation, Ph.D. Thesis, Dept. of Computer Science, Univ. of Warwick, 1984.

[14] I. Parberry, Some practical simulations of impractical parallel computers, in: P. Bertollazzi and F. Lucio, eds., VLSI: Algorithms and Architectures, Proc. International Workshop on Parallel Computing and VLSI (North-Holland, Amsterdam, 1985) 27–37.

[15] I. Parberry, Some practial simulations of impractical parallel computers, Parallel Comput. 4 (1) (1987) 93–101.

[16] N. Pippenger, On simultaneous resource bounds, Proc. 20th Ann. IEEE Symp. on Foundations of Computer Science (1979) 307–311.

[17] W.L. Ruzzo, On uniform circuit complexity, J. Comput. System Sci. 22 (3) (1981) 365–383.

[18] J.T. Schwartz, Ultracomputers, ACM TOPLAS 2 (4) (1980) 484–521.

[19] Y. Shiloach and U. Vishkin, Finding the maximum, sorting and merging in a parallel computation model, J. Algorithms 2 (1981) 88–102.

[20] E. Upfal, A probabilistic relation between desirable and feasible models for parallel computation, Proc. 16th Ann. ACM Symp. on Theory of Computing, Washington, D.C. (1984) 258–265.

[21] U. Vishkin, Implementation of simultaneous memory address accesses in models that forbid it, J. Algorithms 4 (1) (1983) 45–50.

[22] U. Vishkin, A parallel-design space distributed implementation space (PDDI) general purpose computer, Theoret. Comput. Sci. 32 (1984) 157–172.