Parallel Speedup of Sequential Machines: a Defense of the Parallel Computation Thesis.

Ian Parberry

Department of Computer Science,
333 Whitmore Laboratory,
The Pennsylvania State University,
University Park, Pa. 16802.

ABSTRACT

It is reasonable to expect parallel machines to be faster than sequential ones. But exactly how much faster do we expect them to be? Various authors have observed that an exponential speedup is possible if sufficiently many processors are available. One such author has claimed (erroneously) that this is a counterexample to the parallel computation thesis. We show that even more startling speedups are possible. In fact, if enough processors are used, any recursive function can be computed in constant time. Although such machines clearly do not obey the parallel computation thesis, we argue that they still provide evidence in favour of it. In contrast, we show that an arbitrary polynomial speedup of sequential machines is possible on a model which satisfies the parallel computation thesis. If, as widely conjectured, P$\not\subseteq$POLYLOGSPACE, then there can be no exponential speedup on such a model.

1. Introduction.

Intuitively, a parallel computer consists of a collection of interconnected sequential processors. One would expect that, with all this extra computing power available, parallel machines would be much faster than sequential ones. The obvious question to ask is "how much faster?". This apparently simple question is fundamental to the theory of parallel computation. By careful investigation we will highlight a number of basic issues in the design of parallel machine models, in particular, the viability of the parallel computation thesis.

In a recent paper, Blum [2] attacks the parallel computation thesis by considering parallel speedups of sequential machines. Essentially, the paper proposes a parallel machine model which does not obey the parallel computation thesis. The author therefore concludes that the parallel computation thesis is false. The purpose of this paper is to demonstrate that the attack by Blum is shallow, and to provide a deeper understanding of the parallel computation thesis and its

ramifications.

The parallel computation thesis is analogous to the so-called *Church-Turing thesis*. The latter states that all "reasonable" machine models are equal in computing power (and that they compute what one might call the "intuitively computable" functions). Although it is sometimes useful to consider "unreasonable" machine models (for example, augmenting a Turing machine with an oracle for the halting problem), such models are not considered to violate the Church-Turing thesis. Similarly, the *sequential computation thesis* [10] states that time on all "reasonable" sequential machine models is related by a polynomial. The classic example is that of log-cost RAMs and deterministic Turing machines [1]. The fact that the sequential computation thesis does not hold for, say, Alternating Turing machines [4] does not mean that the sequential computation thesis is false, merely that Alternating Turing machines are too powerful to be called "reasonable" sequential machines.

Analogously, the parallel computation thesis [9] states that time on all "reasonable" parallel machine models is related by a polynomial. Note that the parallel computation thesis does not attempt to say that time on *all* parallel machine models is polynomially related; as with the Church-Turing and sequential computation theses, it talks only about "reasonable" models. Thus Blum's model is a counterexample to the parallel computation thesis only if it is "reasonable". Many would not call a parallel machine which runs for T(n) steps and has $2^{2^{O(T(n))}}$ processors "reasonable". A more acceptable number of processors would be $2^{O(T(n))}$, which is achievable in a *lazy activation* model [15] (initially only one processor is active, and in each time-step every active processor can activate an inactive one). At worst, we should perhaps bound the number of processors to be $2^{T(n)^{O(1)}}$. In this paper we give more evidence that machines with a large number of processors are not "reasonable". A (uniform) shared-memory machine with sufficiently many processors can compute *any recursive function* in constant time. Thus we argue that a parallel computer with a large number of processors is surely too powerful to be called a "reasonable" model; on this basis, Blum's counterexample can be dismissed.

The main body of this paper is divided into four short sections. In Section 2 we define our parallel machine model, a variant of the *shared-memory machine* (in which a collection of sequential processors communicates via a common memory). In Section 3 we take a closer look at the parallel computation thesis. As well as equating time on all "reasonable" parallel models, it also attempts to characterize parallel time in terms of a sequential resource, that of *space*. As we shall see, this means that existing work on sequential space requirements can be translated into powerful results on parallel time. In particular, if as conjectured, P$\not\subseteq$POLYLOGSPACE, then there are natural problems in P which have no exponential speedup in parallel.

In Section 4 we find that shared-memory machines with an arbitrary number of processors can compute any recursive function in a constant amount of time. Thus such machines are too powerful to be called "reasonable" parallel models. The fact that the parallel computation thesis rejects such machines is, in our opinion, further evidence that it is a good characterization of a "reasonable" parallel machine. Finally, in Section 5 we examine shared memory machines for which the parallel computation thesis *does* hold. These are obtained by restricting the local instruction-sets of the parallel processors, and placing an upper bound on the size of all registers and memory locations. We find that an arbitrary polynomial speedup of sequential machines is possible, which is an interesting result, since (as we observed in the previous paragraph) no such exponential speedup is likely.

A preliminary version of the results in this paper have appeared in the author's Ph.D. thesis [15]. The reader interested in pursuing some of the ideas presented here is directed to that reference.

2. A Parallel Machine Model.

A *shared memory machine* consists of an infinite number of processors attached to a common 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 i^{th} processor, i$\in$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 a specified place in the shared memory.

(ii) Write a value to a specified place in the shared memory.

(iii) Perform an internal computation.

(iv) Conditional transfer of control, halt.

The allowable internal computations usually consist of direct and indirect register transfers, logical and arithmetic operations.

More formally, each machine is specified by a program 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 i^{th} piece is stored in shared memory location i, $1 \leq i \leq n$. We also allow our machines to have knowledge of a constant number of non-uniform values, which depend only on n. For example, the value of P(n) is known to all processors. All other memory locations and general purpose registers are set to zero. Processors 0,1,...,P(n)-1 are activated simultaneously; they synchronously execute the program P When all processors have halted, the output is to be found in some specified place in the shared memory. In particular, a shared-memory machine acts as an acceptor if, at the end of a computation, shared memory location 0 contains either 1 or 0, indicating acceptance or rejection of the input string respectively. Simultaneous reads of a single memory location are allowed, and in the case of simultaneous write attempts, we assume some reasonable protocol whereby some processor is deemed to *succeed* and is allowed to write its value.

The *processor* bound P(n) is a measure of the number of processors used as a function of input size. The machine is said to have *word-size* W(n) if the maximum value in any register or shared memory location during any computation on an input of size n has absolute value less than $2^{W(n)}$. Note that this includes the inputs, outputs and processor identity registers, so in particular, $W(n) = \Omega(\log P(n))$. 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 in many papers, the earliest of which include

[6, 9, 20, 21]. Opponents of the shared-memory model point out that a multi-access memory is not practical given current technological trends. Despite this opposition, the shared-memory machine has proved to be a powerful vehicle for the discovery and dissemination of theoretical results. It can (under certain conditions) be simulated on more practical models without too great a loss in resources (see [16] for a survey). A number of our design-decisions, for example, the choice of a synchronous model with each processor having the same program, and all processors started simultaneously at the start of the computation, are defended in [15]. We believe that the unit-cost measure of time is a valid one for parallel processors (the so-called *unit-cost hypothesis* of [15]).

3. The Parallel Computation Thesis.

The parallel computation thesis as proposed by Goldschlager [9, 10], states the following: *Time on any "reasonable" model of parallel computation is polynomially equivalent to sequential space.* This has two implications. Firstly, a machine which is too weak to simulate an S(n) space-bounded Turing machine in time $S(n)^{O(1)}$ is *not powerful enough* to be called a parallel machine. Secondly, a machine which is so strong that a T(n) time-bounded computation cannot be simulated in space $T(n)^{O(1)}$ by a Turing machine is *too powerful* to be called parallel. We are interested in the latter aspect of the parallel computation thesis, since the unrestricted shared-memory model proposed in Section 2 is really too powerful to be called parallel. For example, we left the type of internal instruction unspecified. If we allow the internal instructions to be too complex, then our machines (as with purely sequential machines) can compute ordinary functions in an unnaturally small amount of time. Furthermore, as we shall see later, allowing arbitrarily large word-size also confers this power. (It is the large word-size used by Blum [2] and Savitch [19] that allows an exponential speedup of sequential machines.) Thus we see that in order to have our shared-memory machines conform to our intuitions about how a parallel machine should behave, it is necessary to place restrictions on:

1. The internal instruction set, and

2. The word-size.

But what exactly should these restrictions be? The parallel computation thesis suggests an answer. It defines a "reasonable" parallel machine as being one whose time bound is polynomially equivalent to sequential space. Evidence for this thesis is provided by a large number of "reasonable" models, for example Alternating Turing machines [4], uniform circuits [3] and vector machines [18], as well as Goldschlager's SIMDAG and conglomerate. The following theorem tells us what restrictions need to be placed on the instruction-set and word-size of shared-memory machines in order to have them satisfy the parallel computation thesis.

Theorem 1. A T(n) time-bounded shared memory machine M with word-size W(n) can be simulated by a deterministic Turing machine using space T(n).W(n)+S(n), where S(n) is the space required for the Turing machine to simulate a single instruction of a processor of M.

Proof. Similar to Theorem 2.2 of [9]. □

Thus it is sufficient to choose W(n) and S(n) to be $T(n)^{O(1)}$. Goldschlager [9] ensures that his SIMDAG obeys the parallel computation thesis by:

1. Choosing a limited instruction set so that S(n) is negligible.
2. Keeping the word-size small by:
   (a) Insisting that the initial word-size is small. This is achieved by using input-symbols with small word-size, and limiting the size of PIDs by considering only machines for which $T(n) = \Omega(\log P(n))$
   (b) Choosing the instruction-set so that word-size cannot increase too rapidly with time.

The parallel computation thesis is attractive in the sense that it provides us with a powerful theoretical tool. Suppose that we are interested in those problems from P which have an exponential speedup in parallel, that is, those members of P which can be solved in time $\log^{O(1)}n$ by a "reasonable" parallel machine. If a "reasonable" machine is one which obeys the parallel computation thesis, then these are precisely the members of P which can be solved in polylog space by a Turing machine.

Let POLYLOGSPACE denote the class of languages which can be accepted in space $\log^{O(1)}n$ by a Turing machine. It is widely conjectured that P$\not\subseteq$POLYLOGSPACE (although it is not known for sure whether either class contains the other). Evidence is provided for this conjecture by the existence of *log space complete* problems (see, for example, [7, 8, 11, 12, 13, 14]); that is, problems which are members of P, yet if any one of them is a member of POLYLOGSPACE then P$\subseteq$POLYLOGSPACE. Thus log-space complete problems probably do not have an exponential speedup on any "reasonable" parallel machine, where the parallel computation thesis is used as a criterion for "reasonableness".

4. Speedups by Machines with Unbounded Word-size.

In Section 3 we claimed that machines with unbounded word-size were too powerful to be called parallel. In fact, they are so powerful that they can compute any non-recursive function in constant time. This is certainly not the kind of behaviour that we expect from any "reasonable" model.

We investigate the simulation of k-tape Turing machines by shared-memory machines. We use the definitions of deterministic and nondeterministic Turing machines found in [1]. The instruction-set to be used by the shared-memory machines is fairly basic; we allow addition, subtraction, direct and indirect register transfers, as well as logical shifts to the right and extraction of least-significant bits. By the latter two operations we mean the following, where r_i, r_j and r_k are registers:

$$r_i \leftarrow \left\lfloor r_j / 2^{r_k} \right\rfloor$$

$$r_i \leftarrow r_j \ mod \ 2^{r_k}$$

Theorem 2. A T(n) time-bounded nondeterministic Turing machine can be simulated in constant time by a shared-memory machine with word-size $O(T(n)^2)$.

Proof. (Outline). Let M be a T(n) time-bounded, k-tape nondeterministic Turing machine. A *configuration* of M consists of kT(n) tape symbols corresponding to the tape contents, k integers corresponding to the head positions on the k tapes, and a finite string representing the current

state of the finite-state control. We assume that, initially, the input to M is placed in words 1 through n of the shared memory, one symbol per word. Without loss of generality, assume that both n and T(n) are powers of 2.

Firstly, we place an encoding of the initial configuration of M into location 0 of the shared memory, as follows. The processors are divided into $2^{O(n)}$ *teams*, one for each possible input string, each of n processors. Processor $n(j-1)+i-1$, $1 \leq i \leq n$, $1 \leq j \leq 2^{O(n)}$, is the i^{th} member of the j^{th} team. The lowest numbered processor of each team is a distinguished processor called the *manager* of that team. Any processors which are not members of a team remain idle. The i^{th} member of the j^{th} team does the following. It decodes j into a possible input string, and verifies that the i^{th} symbol of this string is the same as the i^{th} input symbol (which is stored in shared memory location i). All team members do this simultaneously. Those processors which detect a discrepancy are said to *fail*. Failed processors notify their respective managers as follows. We reserve a shared memory location for each manager. Each manager stores a zero into its location. Next, any team member which has failed attempts to write a one into the memory location corresponding to its manager. The manager whose location has retained a zero value knows that its decoded string is the same as the input string. It uses this string to construct the initial configuration of M, encoded as an integer of word-size O(T(n)), which it then writes into shared memory location 0.

A *computation* of M consists of a string of T(n) configurations. Now the processors are broken up into $2^{O(T(n)^2)}$ teams, one for each possible computation, each of T(n) processors. Again, the lowest numbered processor of each team is a distinguished processor called the *manager* of that team. The j^{th} team is made up of processors for which $\lfloor PID/T(n) \rfloor + 1 = j$. The i^{th} member of each team has $(PID \ mod \ T(n)) + 1 = i$. The value j is interpreted as the encoding of a computation. Note that the computation is the same for all members of a team, and that each team is assigned a different computation. The i^{th} member of each team, $1 \leq i \leq T(n)$ verifies that the i^{th} configuration of its computation follows from the $(i-1)^{st}$ one; where the 0^{th} configuration is interpreted as meaning the initial configuration of M which has been stored in shared memory location

0. Those team members which detect a discrepancy notify their managers as in the previous paragraph. Those managers which have not been notified know that their computations are valid according to the rules of M. They set the contents of shared memory location 0 to zero, and then extract the final state of their respective computations. Those which detect an accept state attempt to write a one into shared memory location 0.

Thus memory location 0 contains a one iff M accepts the given input. The total time taken is a constant, and the maximum word-size is dominated by the size of the largest PID, which in this case is $O(T(n)^2)$. □

From this we can conclude that every recursive function can be computed in constant time with sufficiently many processors. Further, every language in NP can be recognized in constant time with $2^{n^{O(1)}}$ processors. Theorem 2 has more recently been improved by Parberry and Schnitger [17], who improve the word-size to $O(T(n))$ for the simulation of constant-alternation alternating Turing machines, and hence to $O(T(n)/\log^* n)$ for the simulation of deterministic Turing machines with $T(n) = \Omega(n.\log^* n)$. Theorem 2 is sufficient for our purposes, however.

5. Speedups by "Reasonable" Machines.

So far we have seen that an "unreasonable" parallel machine can be much faster than a sequential one. In Section 2 we proposed the use of the parallel computation thesis as a definition of a "reasonable" machine. In this section we address the following question: how much faster are "reasonable" parallel machines than sequential ones? Dymond and Tompa [5] have shown that speedup by a square-root is possible in the case where word-size is linear in parallel running-time. We will show that an arbitrary polynomial speedup is possible if word-size is allowed to be polynomial in time. First, we present a general speedup result.

Theorem 3. Suppose a shared-memory machine with word-size W(n) can simulate a B(n) time bounded deterministic Turing machine in time B′ (n). Then a shared memory machine with word-size O(W(n)+B(n)+log T(n)) can simulate a T(n) time bounded deterministic Turing machine in time O(T(n)/B(n) + B′ (n)).

Proof. (Outline). Let M be a T(n) time-bounded k-tape deterministic Turing machine. We will store the current configuration of M in the first kT(n)+k+1 words of the shared memory (corresponding to the T(n) tape cells on each tape, k head positions and the control state). The simulation will consist of $\lceil T(n)/B(n) \rceil$ phases, each corresponding to B(n) steps of M. The initial configuration is easy to compute from the input, and the simulation will endeavour to maintain it from phase to phase.

A *situation* consists of that portion of the tape which may be altered during the current phase, that is, the k.(2B(n)–1) tape-cells that are within distance B(n) from a head at the start of the phase. During each phase the simulation will be conducted using these situations - at the end of each phase the final situation will be used to update the stored configuration. To be more precise, a situation consists of k(2B(n)–1) tape symbols, k head-pointers (each of O(log B(n)) bits), and the current state of the finite-state control.

Before the first phase, some pre-computation is carried out. We divide the processors into $2^{O(B(n))}$ teams, one for each possible situation. The i^{th} team, $1 \leq i \leq 2^{O(B(n))}$ simulates B(n) steps of M from the i^{th} situation, in time B′ (n) and word-size W(n). From this information a look-up table is constructed in the shared memory. The i^{th} entry of that table is the situation which follows in B(n) steps from the i^{th} situation of M.

Each phase is broken up into three parts.

(1) Determine the initial situation from the initial configuration of the phase. The processors are broken up into $2^{O(B(n))}$ teams (one for each possible situation), each of 2B(n)–1 processors. The lowest numbered processor of each team is a distinguished processor called the *manager* of that team. Processors which are not members of a team remain idle.

The i^{th} member of the j^{th} team $(i,j \geq 0)$ has $i = \text{PID} \bmod (2B(n)-1) + 1$ and $j = \lfloor \text{PID}/(2B(n)-1) \rfloor + 1$. Each processor first computes i and j. The value j is interpreted as the encoding of a situation (note that j is the same for all members of any particular team). The i^{th} processor of each team, $1 \leq i \leq (2B(n)-1)$ decodes the head positions and the i^{th} symbol of each tape from this situation. Every processor of every team then compares its symbols to the corresponding symbols of the stored configuration. If they disagree, the processor is said to *fail*. Failed team members report to their respective managers via the shared memory using the technique from the proof of Theorem 2.

The team leader whose situation has the correct state, head-pointers equal to B(n), and whose shared memory location remains at zero knows that its value of j is an encoding of the initial situation of the phase. It writes this value to the shared memory for safekeeping.

(2) Determine the final situation of the current phase. Processor 0 can obtain this information from the i^{th} entry in the look-up table, where i is the initial situation of the current phase computed in (1) above.

(3) Determine the final configuration of the phase from the final situation. Processor i, $1 \leq i \leq k$ updates the stored head positions, whilst processor $k+j+cT(n)$, $1 \leq c \leq k$, $1 \leq j \leq T(n)$ determines whether the j^{th} tape cell of the c^{th} tape is within B(n) cells of the head, and if so, updates its value. Processor 0 updates the current state of the finite-state control.

The machine is then in a position to begin the next phase. T(n) steps of M are simulated by repeating this for $\lceil T(n)/B(n) \rceil$ phases. The pre-computation takes time $O(B'(n))$, and each of $\lceil T(n)/B(n) \rceil$ phases takes a constant amount of time, provided the instruction-set of Section 4 is used. The maximum word-size during the computation is the larger of $O(W(n)+B(n))$ (for the pre-computation), $O(B(n))$ (for the processor identity registers during each phase) and $O(\log T(n))$ (to access the stored configuration). □

Corollary 4. Let B:N→N. A T(n) time-bounded deterministic Turing machine can be simulated in time O(T(n)/B(n)) by a shared-memory machine with word-size $O(B(n)^2 + \log T(n))$.

Proof. The proof of Theorem 2 can be modified to work for (deterministic) Turing machines which actually compute values (instead of just being acceptors). The desired result then follows by application of Theorem 3. □

Suppose we require that the parallel computation thesis holds. In this case, since the instruction-set is sufficiently simple, it is enough to bound the word-size to be a polynomial in the parallel running time. In particular, we can choose B(n) to be $T(n)^{1-\epsilon}$ for any real number $0 < \epsilon < 1$. Thus a T(n) time bounded deterministic Turing machine can be simulated in time $T(n)^{\epsilon}$ by a "reasonable" shared memory machine, for $0 < \epsilon < 1$. Thus an arbitrary polynomial speedup is possible. This is an extremely strong result, since, as we observed in Section 3, there are natural problems in P which probably have no exponential speedup.

6. Conclusion.

We have seen that the attack by Blum on the parallel computation thesis is not justifiable. It is possible to construct models for which parallel time is more powerful than sequential space, but all such models encountered so far seem to be "unreasonable". For example, allowing machines to have a large number of processors confers the ability to compute any recursive function in constant time.

The parallel computation thesis is an attempt to formalize this intuitive concept of a "reasonable" parallel machine model. In order to make a shared memory machine "reasonable" according to the parallel computation thesis, it is sufficient to place restrictions on the instruction-set and word-size. We have seen that shared memory machines which lack either of these restrictions can compute natural functions in an unnaturally small amount of time. We claim that this provides further evidence that the parallel computation thesis is a good characterization of "reasonableness".

Other points in favour of the parallel computation thesis are the existence of a large number

of "reasonable" parallel machine models, and the fact that it provides us with a powerful theoretical tool. For example, it can be used to provide guidelines as to which functions are inherently sequential, and which have a large speedup in parallel. We can provide an arbitrary polynomial speedup of sequential machines on a parallel machine which is "reasonable" according to the parallel computation thesis; whereas the theory of log-space completeness tells us that there are natural problems in P which probably have no exponential speedup on such a machine.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

2. N. Blum, "A note on the 'parallel computation thesis'," *Inf. Proc. Lett.*, vol. 17, pp. 203-205, 1983.

3. A. Borodin, "On relating time and space to size and depth," *SIAM J. Comp.*, vol. 6, no. 4, pp. 733-744, Dec. 1977.

4. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation," *J. ACM*, vol. 28, no. 1, pp. 114-133, Jan. 1981.

5. P. W. Dymond and M. Tompa, "Speedups of deterministic machines by synchronous parallel machines," *J. Comput. Sys. Sci.*, vol. 30, pp. 149-161, 1985.

6. S. Fortune and J. Wyllie, "Parallelism in random access machines," *Proc. 10th Ann. ACM Symp. on Theory of Computing*, pp. 114-118, 1978.

7. L. M. Goldschlager, "The monotone and planar circuit value problems are log space complete for P," *SIGACT News*, vol. 9, no. 2, pp. 25-29, 1977.

8. L. M. Goldschlager, "ϵ-productions in context-free grammars," *Acta Inf.*, vol. 16, no. 3, pp. 303-308, 1981.

9. L. M. Goldschlager, "A universal interconnection pattern for parallel computers," *J. ACM*, vol. 29, no. 4, pp. 1073-1086, Oct. 1982.

10. L. M. Goldschlager and A. M. Lister, *Computer science: a modern introduction,* Prentice-Hall, 1983.

11. L. M. Goldschlager and I. Parberry, "On the construction of parallel computers from various bases of Boolean functions," *Theor. Comput. Sci.*, vol. 41, no. 2,3, 1986.

12. L. M. Goldschlager, R. A. Shaw, and J. Staples, "The maximum flow problem is log space complete for P," *Theor. Comput. Sci.*, vol. 21, no. 1, pp. 105-111, 1982.

13. N. D. Jones and W. T. Laaser, "Complete problems for deterministic polynomial time," *Theor. Comput. Sci.*, vol. 3, pp. 105-117, 1977.

14. R. E. Ladner, "The circuit value problem is log space complete for P," *SIGACT News*, vol. 7, no. 1, pp. 18-20, 1975.

15. I. Parberry, "A complexity theory of parallel computation," *Ph. D. Thesis*, Dept. of Computer Science, Univ. of Warwick, May 1984.

16. I. Parberry, "Some practical simulations of impractical parallel computers," in *VLSI: Algorithms and Architectures*, ed. P. Bertollazzi and F. Lucio, Proc. International Workshop on Parallel Computing and VLSI, pp. 27-37, North-Holland, 1985.

17. I. Parberry and G. Schnitger, "Parallel computation with threshold functions (Preliminary Version)," in *Proc. Structure in Complexity Theory Conference*, Springer-Verlag Lecture Notes in Computer Science, vol. 223, pp. 272-290, Berkeley, California, June 1986.

18. V. Pratt and L. J. Stockmeyer, "A characterization of the power of vector machines," *J. Comput. Sys. Sci.*, vol. 12, pp. 198-221, 1976.

19. W. J. Savitch, "Parallel random access machines with powerful instruction sets," *Math. Syst. Theory*, vol. 15, pp. 191-210, 1982.

20. J. T. Schwartz, "Ultracomputers," *ACM TOPLAS*, vol. 2, no. 4, pp. 484-521, Oct. 1980.

21. Y. Shiloach and U. Vishkin, "Finding the maximum, sorting and merging in a parallel computation model," *J. Algorithms*, vol. 2, pp. 88-102, 1981.