stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 69/76 NOVEMBER

W.J. SAVITCH & M.J. STIMSON

TIME BOUNDED RANDOM ACCESS MACHINES WITH PARALLEL PROCESSING

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A20

ACM-Computing Reviews-categories: 5.25, 5.26, 5.23

Time bounded random access machines with parallel processing *)

by

W.J. Savitch & M.J. Stimson **)

ABSTRACT

The RAM model of COOK and RECKHOW [2] is extended to allow parallel recursive calls and the elementary theory of such machines is developed. The uniform cost criteria is used. The results include proofs of (1) the equivalence of nondeterministic and deterministic polynomial time for such parallel machines and (2) the equivalence of polynomial time on such parallel machines and polynomial space on ordinary nonparallel RAMs.

KEY WORDS & PHRASES: *parallelism, nondeterminism, random access machine, time, storage*

---

*) This report will be submitted for publication elsewhere.
This research was supported, in part, by NSF Grant MCS-74-02338A01.

**) M.J. STIMSON, Department of Nuclear medicine (115) Veterans Administration Hospital 3350 La Jolla Village Drive,
San Diego, California 92061

I. INTRODUCTION

A machine model called a parallel random access machine (PRAM) is introduced. The model is obtained by extending the RAM model of Cook and Reckhow [2] to allow parallel recursive calls. These PRAMs are then used to develop a theory for the time complexity of parallel algorithms. In this paper we consider only the uniform cost criteria [2]; that is, in computing running times, we charge one time unit for each memory transfer. Our results include proofs of (1) the equivalence of nondeterministic and deterministic polymomial time on PRAMs and (2) the equivalence of polynomial time bounded PRAMs and polynomial storage bounded ordinary non-parallel RAMs.

II. THE PRAM MODEL

A *k-offspring parallel random access machine* (or k-PRAM) consists of a finite program and a set of processors. As with an ordinary RAM, each k-PRAM processor has a pointer into the program telling it which instruction is to be executed next (all processors use the same program). Each processor has a memory which consists of a sequence of registers, $R_0, R_1, \ldots, R_i, \ldots$, each of which is capable of holding an integer of arbitary size. Each processor also has available to it k other processors (its offspring) which it can call. Each processor may have up to k of its offspring computing in parallel with it. These k processors may, in turn, call up to k offspring each, and so forth. When an offspring is done with its computation, it returns its response to its parent by way of a special bank of registers called *channels*. That offspring is then available for another recursive call. Channel ℓ, $1 \leq \ell \leq k$ is used by a processor to receive parameters passed to it by its ℓ^{th} offspring. Channel ℓ is a bank of read-only channel registers, $C_0^\ell, C_1^\ell, \ldots, C_i^\ell, \ldots$, each of which is changed only as a result of offspring ℓ completing a computation. Therefore, each channel register is a read-only register to the parent, but the registers in channel ℓ are modified whenever offspring ℓ returns from a recursive call.

The *program* for a PRAM is a sequence of (optionally) labeled instructions. If the labels in a PRAM program are unique, the program is said to be *deterministic*; otherwise, it is said to be *nondeterministic*. The instructions for a PRAM program are drawn from the instruction set given in Table 1. In Table 1, ℓ is an integer such that $1 \leq \ell \leq k$; a and b are operands of the form

(i) i, indicating the integer value i, or
(ii) R_i, indicating the contents of register R_i, or
(iii) C_i^{ℓ}, indicating the contents of channel register C_i^{ℓ}.

The relation COMP in Table 1 may be any of the binary relation symbols $<$, $\leq$, $=$, $\geq$, $>$, $\neq$, where these symbols have their usual interpretation over the integers. In what follows, we will consider each instruction (except the call and return instructions) to have a cost of one time unit.

Instructions 1 through 7 are exactly the same as their counterparts for ordinary RAMs (see, for example, Cook and Reckhow [2]). Instruction 8 is the same as instruction 3, except that in 8 it is a channel register that is accessed. Instruction 8 changes the value of R_i to the value of channel register $v(R_j)$ of the ℓ^{th} channel bank, $1 \leq \ell \leq k$. (v(a) denotes the value of operand a. See, for example, Aho, Hopcroft and Ullman [1].) Instruction 9 allows a processor (the parent) to initiate its ℓ^{th} offspring, $1 \leq \ell \leq k$. If that offspring is currently active, the parent is blocked at that instruction until the offspring returns. The operands a and b specify the upper and lower register addresses of the parameter list. When the call instruction is executed, the contents of the parent's registers $R_{v(a)}$ through $R_{v(b)}$ are copied into registers R_0 through $R_{v(b)-v(a)}$ of the offspring. The cost of the call instruction is v(b)-v(a) + 1; that is, the cost of copying the parent's registers over to the offspring's registers. If the parent attempts to access channel register C_i^{ℓ} or recall offspring ℓ while it is still active, then the parent is blocked at that point in the computation until offspring ℓ completes its computation and returns. Instruction 10 allows an offspring to return an answer to its parent. The operation of this instruction is similar to that of the call instruction, except that

Instruction Number	Instruction Format	Function
1	$R_i \leftarrow b$	direct assignment
2	$R[R_i] \leftarrow b$	indirect assignment
3	$R_i \leftarrow R[R_j]$	indirect assignment
4	$R_i \leftarrow a + b$	addition
5	$R_i \leftarrow a - b$	subtraction
6	IF a *COMP* b THEN $LABEL_1$ ELSE $LABEL_2$	conditional branch
7	GOTO LABEL	unconditional branch
8	$R_i \leftarrow C^{\ell}[R_j]$	indirect assignment
9	CALL ℓ(a,b)	call to offspring ℓ
10	RETURN (a,b)	return to parent

Table 1. Basic PRAM Instructions

information is passed back up to the parent. When offspring ℓ executes a RETURN (a,b) instruction, the contents of offspring ℓ's registers $R_{v(a)}$ through $R_{v(b)}$ are loaded into the parent's channel registers C_0^{ℓ} through $C_{v(b)-v(a)}^{\ell}$. All other channel registers, C_i^{ℓ}, $i > v(b)-v(a)$, will contain zero at the completion of the return instruction. As with the call instruction, the return instruction costs $v(b)-v(a) + 1$ time units.[1] (In instructions 9 and 10, we assume $a < b$. If $a \geq b$, then the program terminates.)

The computation of a PRAM proceeds as follows. Initially, there is a single processor active (level 1). The input is in register R_0 of this processor and all other registers are set to zero. The PRAM computes in much the same way as an ordinary RAM until a call instruction is executed. When this occurs, an offspring processor is activated. Information is copied from the parent's parameter registers to the registers of the offspring as outlined in the description of the call instruction above. All of the offspring's other registers are set to zero. The offspring then starts computing at level m+1 (where the parent is computing at level m). Both processors remain active and compute in parallel (and, in fact, each processor may make additional calls) until one executes a return instruction. When this occurs, the return parameters are loaded into the channel registers corresponding to the returning offspring and this processor and all of its descendants are removed from the computation.

A *subcomputation* of a PRAM program P on parameter list $(a_0,a_1,\ldots,a_u)$ is a computation performed by a processor (and its descendants), executing program P, from the instant it is initialized with $R_i \leftarrow a_i$, $0 \leq i \leq u$, $R_i \leftarrow 0$, $i > u$, until it executes a return instruction. A subcomputation of P performed by the initial processor (that is, level 1) on some input x ($a_0=x$, $u=0$) is called a *computation of P on x*. An integer x is *accepted* (respectively *rejected*) if there is a computation of P on x such that the computation terminates with 1 (respectively 0) returned at level 1. A set A is *accepted* by a PRAM program P if P accepts exactly the integers in A.

---

[1] Actually, we could argue that a cost of one time unit is more reasonable for this instruction since channel register C_i^{ℓ} may be considered to be register R_i^{ℓ} of offspring ℓ. However, it turns out that the particular criterion we select is unimportant to our study here.

Similarly, a set A is *recognized* by a PRAM program P if P accepts every integer in A, rejects every integer not in A, and (if P is nondeterministic) there is no integer x such that P both accepts x and rejects x.

Nondeterministic PRAMs have a peculiar property which deserves some discussion. If P is a nondeterministic PRAM which recognizes some language A, then P is consistent in the sense that for each input x, P either accepts or rejects w but not both. P may, however, exhibit inconsistency at lower levels. P may make a recursive call and, since P is nondeterministic, there can be one computation of the offspring which returns one value and some other perfectly valid computation that returns a different value. We will assume our PRAM programs have the property that, for any particular parameter list, an offspring can return at most one list of values to its parent. PRAMs with this property are called *consistent*. Deterministic PRAMs are always consistent.

We will measure time and storage as a function of the length of the input. By the length of an integer x, we mean the length of the binary numeral for $|x|$ with no leading zeros.[2] ($|x|$ denotes the absolute value of x; length(x) denotes the length of x.) A k-PRAM P is said to *accept* the set A of integers *in time bound* T(n) provided that P accepts A and that, for every x in A, there is an accepting computation of P on x that takes at most T(n) steps. P is said to *recognize* A *in time bound* T(n) provided that P recognizes A and, for every integer x, there is a computation of P on x that is either an accepting or rejecting computation and that takes at most T(n) steps. In this paper we will talk about both accepting and recognizing sets. Whenever we say that a machine X can simulate a machine Y, we will mean that machine X accepts the same set as machine Y and furthermore, if machine Y actually recognized a set, then machine X also recognizes that set. We now describe a standard niceness condition that we will assume our time bounds satisfy.

We say that a function f(n) is *T(n) time countable* provided there is a deterministic ordinary RAM which, given some input x of length n, will

---

[2] length(0) = 1

construct the value f(n) in one of its registers within T(n) time units. A function is *O(T(n)) time countable* if it is cT(n) time countable for some c. The concept of *S(n) storage countable functions* is defined in an analogous way. In what follows we will assume that every time bound function T(n) is O(T(n)) time countable. We also assume that all time bound functions T(n) are such that $T(n) \geq n$, for all n.

III. PROGRAMMING LANGUAGES

Cook and Reckhow [2] describe a high-level programming language for RAMs which they call RAM-ALGOL. The language is similar to ALGOL 60 but is more restricted: the only data types are integer and Boolean; the only arithmetic operations are + and -; arrays are one-dimensional and infinite; procedures and switches are not allowed to be recursive. In an exactly similar way we can construct a programming language for PRAMs called PRAM-ALGOL. PRAM-ALGOL is obtained from RAM-ALGOL by adding instructions for parallel recursion. Specifically, PRAM-ALGOL has instructions like the CALL and RETURN instructions of the PRAM model and also has special variables C_i^{ℓ} that are treated like the channel variables of the PRAM model. In PRAM-ALGOL, the CALL and RETURN instructions list the variables whose values are passed as parameters. So, for example, CALL ℓ(X,A[7]) passes as parameters the values of variable X and array element A[7], and RETURN (X,A[7]) when executed by the ℓ^{th} offspring recursive copy of the program causes variables C_0^{ℓ} and C_1^{ℓ} of its parent program to be set to the values of X and A[7] respectively. After a RETURN instruction the copy of the program that executed the instruction and its descendants disappear as in the PRAM model.

In this paper we will usually state our algorithms informally in English. It should be clear that the algorithms can be converted to PRAM-ALGOL and ultimately to PRAM programs. In particular, we will use mnemonics for variables and arrays. As with RAM-ALGOL, the cost of a PRAM-ALGOL program is computed by charging one time unit for each memory access. Also

as with RAM-ALGOL, the run time of a PRAM-ALGOL program is proportionate to the run time of its equivalent PRAM program. (See [2] for a discussion of the details.) Similarly, it should be clear that the run time of our informally stated algorithms are also proportionate to their equivalent PRAM programs.

IV. ELIMINATING NONDETERMINISM

In this section we show that, for k-PRAMs, the class of languages accepted in deterministic polynomial time is the same as the class of languages accepted in nondeterministic polynomial time.

THEOREM 4.1: Every nondeterministic, T(n) time bounded k-PRAM can be simulated by a deterministic k-PRAM in time $O(T^2(n))$, provided $k \geq 2$.

Proof. Let P be a T(n) time bounded nondeterministic k-PRAM. Assume, without loss of generality, that no label occurs more than twice in the program for P. We will describe a deterministic k-PRAM, P_D, which will recognize the set accepted by P. P_D operates by simulating P. Since P is nondeterministic, it may have more than one subcomputation for a given parameter list $(a_0,a_1,\ldots,a_u)$. P_D will be constructed so that, given a parameter list $(a_0,a_1,\ldots,a_u)$ to simulate P on, P_D will simulate all possible subcomputations of P on this parameter list. In order to do this P_D uses a "clock" running at the simulated time of P. That is, a register called CLOCK, will be initialized to T(n) and will be decremented every time a move of P is simulated. So when CLOCK is less than or equal to zero, T(n) moves of P have been simulated and the simulation can end. In order to keep CLOCK current, the value of CLOCK will be passed as a parameter from parent to offspring and from offspring to parent.

Note that if an offspring of P initially starts computing (say at level m) with a clock value of t, then there are at most 2^t possible sequences of moves that can be made by that offspring at level m until it returns. We will construct P_D so that the offspring which is called to simulate a particular offspring of P will generate 2^t offspring (by a sequence of t pairs

of recursive calls) while constructing 2^t vectors of t bits. Each vector is given to one of the 2^t offspring and the i^{th} element of the vector, V[i], tells the offspring which of the two occurrences of a label it should branch to at the i^{th} nondeterministic move of its computation (at the simulated level of P). That is, V[i] = 0 tells the offspring to branch to the first occurrence of the label; V[i] = 1 tells the offspring to branch to the second occurrence of the label. In this way, each of the 2^t offspring may simulate a different computation of the offspring of P.

The simulated parameter list, the vector for determining nondeterministic choices and the current value of the parent's CLOCK are passed to each of these 2^t offspring. This value of CLOCK becomes the value of CLOCK for each offspring. Each of the 2^t offspring then simulate P at the appropriate level, consulting its vector to determine which nondeterministic choices to make and decrementing CLOCK by an amount equal to the elapsed simulated run time of P. (Note that the vector only determines nondeterministic moves for one processor at one level. When it needs to simulate an additional recursive call to a lower level, another 2^t offspring are generated with t set to the current value of CLOCK.) When an offspring returns it passes its simulated returned parameters and its current value of CLOCK to its parent. If a processor ever gets a value of CLOCK which is zero or less, then it immediately returns to its parent and reports that it has run out of time by returning a value of CLOCK equal to zero.

The ancestors of the 2^t offspring wait for both of their descendants to return, whereupon they return the parameter list and CLOCK value returned by the descendant which was successful (that is, the one whose clock did not run out). Notice that if more than one of the 2^t offspring are successful, they will return the same parameter list, since P was originally consistent. In this case, the highest value of CLOCK is returned as the value of CLOCK. If all offspring are unsuccessful, the processor simply

returns a zero clock value. In this way, P_D can simulate all possible sub-computations that could occur as a result of a recursive call by P.

In order to insure that the value of CLOCK is correct, the returned value of an offspring's clock is used to correct the value of its parent's clock. Whenever a processor wishes to simulate P accessing channel ℓ, or recalling offspring ℓ, it first checks the value of CLOCK returned by its ℓ^{th} offspring. If that value is less than the parent's value of CLOCK, then the parent resets its value of CLOCK to this lower value. In this way, the parent's clock is adjusted to reflect the time that P would spend waiting for an offspring to return.

Since a clock value of zero or less causes a processor to return, all offspring eventually return. If the value of CLOCK at level one is ever zero or less, then P could not accept the input within time T(n). So if the processor at level one ever gets a value of CLOCK which is zero or less, then it rejects the input.

In order to complete the description of P_D, we must tell how the vectors V[i] are passed to the 2^t offspring. Passing the vector as a single array would yield a simulation time of $O(T^3(n))$. By passing the vector as a single number V and viewing the number as a bit string, we can get the simulation time down to $O(T^2(n))$. The numbers V are passed down as follows. The first two offspring are passed values V=1 and V=0. Each intermediate offspring, at $L < t$ levels below, passes its two offspring the values 2^L+V and V respectively. It also passes each offspring an indication of their level L+1 and the value t. The 2^t offspring at level L=t then unpack the bits in V and store them in a vector V[i]. The simulation then proceeds as outlined above. The unpacking can be done in time O(t). The unpacking algorithm is given in Appendix 1.

We shall now examine the time requirements for P_D. All moves of P, except for calls and returns, require c_1 steps to simulate, for some constant c_1. In the case of calls, P_D must construct the 2^t vectors (where t is the

clock value at the time of the call). If the cost of the original call instruction was s time units, then generating 2^t offspring and their vectors will require t+1 recursive calls, each with a cost bounded above by $s+c_2$ for some constant c_2. The 2^t offspring must then unpack the vector V and place it in an array V[i]. This costs an additional O(t) time units. Therefore the cost of generating 2^t offspring and their vectors will be O(st) time units. In the case of a simulated return instruction, the return parameter list must be passed up through t+1 levels of recursion (where t is the clock value at the time of the call). If the cost of the original return instruction was s time units, then the parameter list can be returned up to the simulating processor in O(st) time units. It may also require O(t) time units to wait for all of the siblings to return, so the total cost of a return instruction is still O(st) time units. So, taking T(n) as an upper bound on t, we see that all steps of P can be simulated in time

$$\max\{O(s),O(st)\} = O(st),$$

where s is the cost of the simulated move in P. Therefore, since one P time unit can be simulated in $O(T(n))P_D$ time units and P_D needs to simulate T(n) time units, it follows that P_D runs in time $O(T^2(n))$. □

Notice that the algorithm in the proof of Theorem 4.1 gives us a deterministic recognizer, even if the original machine is only an acceptor. Therefore, if we are interested in polynomial time, we may assume that our k-PRAMs are recognizers.

The next two results show that deterministic 2-PRAMs are strictly more powerful than nondeterministic ordinary RAMs, provided the 2-PRAMs are allowed a small additional run time.

THEOREM 4.2: If h is any positive integer and T(n) is any (O(T(n)) time countable) function (such that $T(n) \geq n$), then there is a set A such that A is recognized by a O(T(n)) time bounded deterministic 2-PRAM but A is not

accepted by any T(n) time bounded nondeterministic ordinary RAM with a program in which each label occurs at most h times.

Proof. We will first give the proof for the case h=2.

In a straightforward way, code nondeterministic ordinary RAM programs in which each label occurs at most twice as binary strings with first digit one. Do this in such a way that labels can be considered to be positive integers and that each statement is labeled. Then every such program can be viewed as a positive integer (in binary). Let A be the set of all integers x such that x is the code for such a RAM program p(x) and such that p(x) does not accept x in time T(n) where n is the length of x. By a standard diagonal argument it follows that A is not accepted by any T(n) time bounded nondeterministic ordinary RAM with a program in which each label occurs at most twice. To complete the proof, we will describe a O(T(n)) time bounded 2-PRAM, P, that recognizes the set A.

P operates as follows. Given input x, P first checks to see if x codes a RAM program of the appropriate type. If not, then P rejects x. If x does code such a RAM program, then P constructs T(n) and, as in the proof of Theorem 4.1, P generates $2^{T(n)}$ offspring. Each offspring is passed x, T(n) and one of the $2^{T(n)}$ possible vectors consisting of T(n) bits. As in the proof of Theorem 4.1, this vector is passed as a single binary numeral. Each of these offspring puts T(n) in a variable called CLOCK and, as in the proof of Theorem 4.1 (Appendix 1), it unpacks its vector of T(n) bits and stores it in an array V[i]. The offspring then saves a copy of x and viewing x as a binary string, it unpacks the string x. As the bit string x is unpacked, the program p(x), coded by x, is stored in nine arrays: I[i], A1[i], A2[i], A3[i], A4[i], R1[i], R2[i], R3[i], and R4[i]. The program is stored in such a way that it can be efficiently simulated. To simplify the description of how the program is stored, assume that the only comparison allowed is $a \leq b$. It is trivial to extend the techniques to allow all the arithmetic comparisons. The instruction labeled by the first occurrence of label k is stored in array elements indexed by 2k. The instruction (if any) labeled by the second occurrence of label k is stored in array elements indexed by

2k+1. For i = 2k or 2k + 1, whichever is appropriate, I[i] holds the instruction number as given in Table 1. Each instruction can have up to four constants occurring in it. Each such constant is either a literal, a register address or a label. These constants are stored in array elements A1[i], A2[i], A3[i] and A4[i]. If the constant in Aj[i] is the address of a register, then Rj[i] is set equal to one otherwise Rj[i] is set equal to zero. For example, if the first instruction with label 7 is

IF $5 \leq R_3$ THEN 12
ELSE 9

then I[14] = 6, A1[14] = 5, A2[14] = 3, A3[14] = 12, A4[14] = 9, R1[14] = 0, R2[14] = 1, R3[14] = 0 and R4[14] = 0.

After doing all of the above, the offspring simulates program p(x) on input x. After each simulated move, CLOCK is decremented. If the input is accepted or the value of CLOCK gets to zero, then the simulation stops. In this way the offspring can determine whether or not the RAM program p(x) accepts the input x in T(n) steps using the nondeterministic choices indicated by the vector of length T(n) which it received. The offspring returns a value saying whether or not its simulation accepted the input x. The intermediate processors return a value saying whether or not any of their descendants performed a simulation that accepted x. The level one processor rejects x if at least one of the $2^{T(n)}$ possible computations is an accepting computation; otherwise, the input x is accepted. Clearly, P recognizes the set A. It remains to estimate the run time of the 2-PRAM P.

It takes O(n) time units to unpack the input x and check to see if it is a suitable RAM program and O(T(n)) time units to construct T(n). Generating the $2^{T(n)}$ offspring takes time O(T(n)). Each offspring takes time O(T(n)) to unpack its vector and O(n) time unit to unpack and store the program p(x). Because of the manner in which the program p(x) is stored, these offspring can use indirect addressing to simulate T(n) steps of p(x) in time O(T(n)). Finally, it takes another O(T(n)) steps to wait for all offspring to complete their simulation and to pass their results up to the level one processor. So the total time is O(T(n)). This completes the proof for the case h=2.

The result for arbitrary h follows from the result for the case h=2. To see this let h be a fixed but arbitrary positive integer. It is easy to show that there is a constant c, depending only on h, and such that: if a set A is accepted by a T(n) time bounded nondeterministic RAM in which each label occurs at most h times, then A is accepted by a cT(n) time bounded nondeterministic RAM in which each label occurs at most twice. Now let A be a set satisfying the theorem with h replaced by 2 and T(n) replaced by cT(n). It follows that A satisfies the theorem for this arbitrary h and the time bound T(n). □

THEOREM 4.3: There are sets that can be recognized by $O(T_1(n))$ time bounded deterministic 2-PRAMs but cannot be accepted by any $O(T_2(n))$ time bounded nondeterministic ordinary RAM, provided $\sup_{n\to\infty} T_2(n)/T_1(n) = 0$.

Proof. The proof is almost identical to the proof of Theorem 4.2. So we will simply sketch the changes that need to be made in the proof of Theorem 4.2 in order to obtain a proof of this result. Let A be the set of all integers x such that x is the code for some RAM program p(x) in which each label occurs at most twice and such that p(x) does not accept x in time $T_1(n)$ where n is the length of x. As in the proof of Theorem 4.2, A is recognized by a $O(T_1(n))$ time bounded deterministic 2-PRAM. So it remains to show that A is not accepted by any $O(T_2(n))$ time bounded nondeterministic ordinary RAM. Suppose A is accepted by a $cT_2(n)$ time bounded nondeterministic ordinary RAM, M, where c is a constant. Without loss of generality, we may assume that each label in the program for M occurs at most twice and that every such program has infinitely many codings. Let x be a coding of M which is large enough to insure that $cT_2(n) \leq T_1(n)$, where n is the length of x. By standard arguments it follows that M accepts x if and only if M does not accept x. Thus we get a contradiction and conclude that no such RAM, M, can exist. □

V. LIMITING PARALLELISM

In this section we examine the effect of limiting parallelism by limiting the number of offspring which each processor may call and by limiting the total number of available processors. We also examine the problem of eliminating parallelism altogether. Our first result states that a k-PRAM can be simulated by a 2-PRAM in time that is polynomially related to the k-PRAM running time. The general idea of the proof is fairly simple. If for example $k = 2^t$, then a 2-PRAM can simulate k offspring by a series of t levels of recursive calls producing $2^t = k$ descendents. The details of the proof are, however, surprisingly complicated; so, rather than include the details in the main text, we have put the proof in Appendix 2.

THEOREM 5.1: A nondeterministic T(n) time bounded k-PRAM can be simulated by a nondeterministic 2-PRAM in time $O(T^6(n))$.

Combining Theorems 4.1 and 5.1 yields:

THEOREM 5.2: A nondeterministic T(n) time bounded k-PRAM can be simulated by a deterministic 2-PRAM in time $O(T^{12}(n))$.

It is worth noting that we have not been able to prove a better bound for Theorem 5.2 in the case where the k-PRAM is already deterministic.

COROLLARY 5.3: Every nondeterministic, polynomial time bounded k-PRAM can be simulated by a deterministic, polynomial time bounded 2-PRAM.

In the proofs of many of our results on limiting parallelism, it will be useful to have yet another machine model, the recursive RAM. A *recursive* RAM is like a 1-PRAM except that when a recursive RAM calls its offspring, the parent is blocked until the offspring returns. A recursive RAM has the same instruction set as a 1-PRAM, but no parallelism. The following result is fairly well known, and its proof quite intuitive; therefore, the proof will not be given here.

LEMMA 5.4: A T(n) time bounded recursive RAM can be simulated by an ordinary RAM in time $O(T(n))$. Furthermore, if the original algorithm were deterministic, then the simulating algorithm will also be deterministic.

Theorem 4.1 says that for k-PRAMs, deterministic and nondeterministic polynomial times are equivalent. If we insist that the number of available processors is also polynomially bounded, then it appears likely that the nondeterministic model is more powerful. Our next two theorems say that k-PRAMs which work in polynomial time and use only a polynomial number of processors are equivalent to ordinary polynomial time bounded RAMs. In order to state and prove this result, we need a definition.

We say that a k-PRAM program P is *simultaneously* T(n) *time bounded and* U(n) *processor bounded* provided the following holds for all inputs x of length n. If x is accepted (respectively rejected) by P, then there is some computation of P on x such that

(i) this computation is an accepting (respectively rejecting) computation,

(ii) this computation takes T(n) steps, and

(iii) no more than U(n) processors are ever computing in parallel during the computation.

(A blocked processor is considered to be computing.) Notice that, if we view a k-PRAM as an acceptor, then the bounds T(n) and U(n) need only be satisfied for computations which accept the input. If we view a k-PRAM as a recognizer, then every input must yield at least one suitably resource-bounded computation.

THEOREM 5.5: A nondeterministic k-PRAM program P, which is simultaneously T(n) time bounded and U(n) processor bounded, can be simulated by an ordinary nondeterministic RAM in time O(U(n)T(n)).

Proof: By Lemma 5.4 it will suffice to produce a recursive RAM program R which simulates P in time O(U(n)T(n)). We construct R so that it performs a simple step-by-step simulation of P. However, when P would make a parallel call and continue computing in parallel with its offspring, R will make the call but will "wait" for the return of the offspring before it continues the simulation. The above algorithm works except for one rather pathological case. If left alone, a processor in a P computation might never return and

might, in fact, generate an unbounded number of descendants. Such "run-away" processors are "shut off" in P by having one of their ancestors execute a return instruction. In R, their ancestors are waiting and so do not shut them off. To overcome this problem R will, for each simulated processor, nondeterministically guess at when to shut the processor off. With this modification, R clearly works and works in time $O(U(n)T(n))$. □

THEOREM 5.6: A deterministic k-PRAM program P which is simultaneously T(n) time bounded and U(n) processor bounded can be simulated by an ordinary deterministic RAM M in time $O(U(n)T^2(n))$.

Proof. M keeps a complete description of P at an instant of time in M's storage. Then M updates this description one step at a time. Since M does not know how many processors will be active at any one time, it cannot simply interleave storage to allow an infinite bank of registers for each processor. So M uses a linked list structure to assign an infinite bank of registers to each active processor. Since there are always at most U(n) active processors, M will need to update the description of individual processors at most $O(U(n)T(n))$ times. Since the simulated memory of a simulated P processor is stored as a linked list, it may take up to, but no more than, $cT(n)$ steps to update the configuration of a single processor; here c is a fixed constant. So the total run time of M is $O(U(n)T^2(n))$. The details of the algorithm for M are quite involved but use only standard techniques. □

COROLLARY 5.7: Every simultaneously polynomial time bounded, polynomial processor bounded k-PRAM can be simulated by an ordinary RAM in polynomial time. Furthermore, if the original algorithm were deterministic, then the simulating algorithm will also be deterministic.

Observing that every T(n) time bounded 1-PRAM is also T(n) processor bounded, we get the following.

COROLLARY 5.8: A polynomial time bounded 1-PRAM can be simulated by a polynomial time bounded ordinary RAM. Furthermore, if the original algorithm were deterministic, then the simulating algorithm will also be deterministic.

The following two results are immediate consequences of Corollaries 5.8 and 5.3.

COROLLARY 5.9: If every polynomial time bounded, deterministic 2-PRAM can be simulated by a deterministic 1-PRAM in polynomial time, then P=NP (where P and NP are to be understood in their usual non-parallel sense).

COROLLARY 5.10: If every nondeterministic, polynomial time bounded RAM can be simulated by a deterministic, polynomial time bounded 1-PRAM, then P=NP. (Again, we mean P and NP in the usual non-parallel sense.)

VI. TIME-STORAGE TRADEOFF

We wish to examine the complexity of storage bounded computations for k-PRAMs and ordinary RAMs. The conventional way of defining a storage bounded computation on a RAM involves charging for all registers used from register R_0 through register R_a, where a is the largest register address used in the computation. More precisely, we say that a computation of a RAM program R on input x is S(n) *storage bounded* if, at any point in this computation

$$\sum_{i=0}^{a} \text{length}(R_i) \leq S(n)$$

where a is the maximum register address used in the computation and n = length(x).

In what follows, it will be convenient to use a slightly different notion of storage bounded computations, wherein we charge only for those registers which are actually "used" in the computation. By "used" we shall mean that the registers contain nonzero values. More precisely, we say that the computation of a RAM program R on input x is S(n) *register bounded* if, at any point in its computation on x,

$$\sum_{i} [\text{length}(R_i) + \text{length}(i)] \leq S(n)$$

where the sum is over all i such that the contents of R_i is not zero and where n = length(x). It is easy to show that these two storage measures are equivalent up to growth rate. The proof is therefore omitted.

LEMMA 6.1: An O(S(n)) storage bounded RAM can be simulated by an O(S(n)) register bounded RAM and conversely. Furthermore, if one RAM is taken to be deterministic, then the simulating RAM will also be deterministic.

We also need a storage measure for k-PRAMs. Consider some computation of a k-PRAM program P on input x, where n = length(x). We say that a subcomputation of some processor of P in this computation is S(n) *register bounded* if at any point in the subcomputation the registers of this processor satisfy the following inequality.

$$\sum_i [\text{length}(R_i) + \text{length}(i)] + \sum_{\ell=1}^{k} \sum_j [\text{length}(C_j^{\ell}) + \text{length}(j)] \leq S(n)$$

where the first sum is over all i such that the contents of R_i are not zero and the last sum is over all j such that the contents of C_j^{ℓ} are not zero. We say that a computation of the k-PRAM program P on input x is *locally* S(n) *register bounded* provided every subcomputation of every processor in the computation of P on x is S(n) register bounded.

We will talk about computations that have several simultaneous resource bounds. The definition is given in terms of recognition programs. The definition for accepting programs is analogous. A k-PRAM program P is said to recognize the set A with *simultaneous register bound* S(n), *depth bound* D(n) and *time bound* T(n) provided that R recognizes A and that, for each input w such that n = length(w) there is at least one computation of P on input w such that

(i) this computation is either an accepting or rejecting computation,
(ii) this computation takes at most T(n) time units,
(iii) this computation is locally S(n) register bounded, and
(iv) if m is the lowest level of recursion, then $m \leq D(n)$.

In order to obtain a storage efficient simulation of a k-PRAM by an ordinary RAM, we will proceed in two steps. First, give an algorithm to simulate a k-PRAM on a recursive (non-parallel) RAM. Second, give an algorithm to simulate the recursive RAM on an ordinary RAM. We will now give these two simulation algorithms; however, we will present them in the reverse order.

LEMMA 6.2: A simultaneous S(n) register bounded, D(n) depth bounded recursive RAM program R can be simulated by a O(S(n)D(n)) register bounded ordinary RAM. Furthermore, if the original algorithm were deterministic, then the simulating algorithm will also be deterministic.

Proof. The usual method for implementing recursion by constructing and using a stack yields a O(S(n)D(n)) storage bounded RAM program. By Lemma 6.1, we know this O(S(n)D(n)) storage bounded program can be simulated by a O(S(n)D(n)) register bounded program. □

LEMMA 6.3: Let S(n), T(n) and D(n) be any functions such that S(n), T(n) and D(n) are O(S(n)) storage countable. A simultaneous S(n) register bounded, D(n) depth bounded, T(n) time bounded k-PRAM program P can be simulated by a O(S(n)D(n)) register bounded ordinary RAM. Furthermore, if the original algorithm were deterministic, then the simulating algorithm is also deterministic.

Proof. The proof uses the techniques developed in the proofs of Theorems 4.1 and 5.5. So we will merely sketch the construction. By Lemma 6.2, it will suffice to describe a recursive RAM program, R, which simulates P, which is simultaneously O(S(n)) register bounded and D(n) depth bounded and which is deterministic if P is deterministic. We now describe such a recursive RAM program R. Basically, R does a step-by-step simulation of P except that, when a processor of R makes a recursive call, it must wait for its offspring to return before proceeding with its computation. If this simulation is implemented in the simplest way, three problems can arise. First of all, the offspring called by P may never return, but P may continue to compute and ignore this subcomputation (by returning to its parent) and go on to recognize the input. Since R has no parallelism, it would wait for this offspring before proceeding, and hence its computation would never terminate and it would not recognize the input. The second problem is that an offspring whose computation is to be ignored (as outlined above) may still recurse deeper than D(n) in T(n) time units. The third problem is that there may be some processor in P's computation which, if left to compute, would exceed the

register bound S(n), but which disappears before reaching this bound, because its parent executes a return instruction. Such a processor would cause R to exceed the register bound O(S(n)), if the simulation were implemented in the most naive way. To avoid all these problems, R has three routines which terminate processor computations before any of these problems arise. These routines insure that R simulates at most T(n) steps of P, and that R is simultaneously O(S(n)) register bounded and D(n) depth bounded.

To insure that R simulates at most T(n) steps of P, the following clocking routine is used. A variable CLOCK is initialized to T(n). Every time a move of P is simulated, the variable CLOCK is decremented by an amount equal to the elapsed simulated run time of P. Whenever an offspring is called, its value of CLOCK is set equal to its parent's value of CLOCK. When an offspring returns, it passes its value of CLOCK to its parent; before a parent simulates accessing an offspring's channel or recalling the offspring, it resets its value of CLOCK to the minimum of its value of CLOCK and the value of CLOCK returned by that offspring. Whenever a processor gets a value of CLOCK ≤ 0, it ends its computation and returns. If the level-one processor gets a value of CLOCK ≤ 0, then the computation is aborted.

To guarantee that R is D(n) depth bounded, it uses a variable called LEVEL. This variable is initialized to D(n). Every time a recursive call is made, the value LEVEL-1 is passed to the offspring and the offspring sets its variable LEVEL to this value. Whenever a processor has a value of LEVEL equal to one and wishes to simulate a recursive call, it returns to its parent and reports that its subcomputation has exceeded the depth bound. If, in the simulation process, a processor wishes to access an offspring's channel, it first checks to see if the offspring has exceeded the depth bound. If the offspring has exceeded the depth bound, then the parent processor immediately returns and reports that its subcomputation has exceeded the depth bound. Should the level-one processor determine that its subcomputation has exceeded the depth bound, then the computation is aborted. Notice that the above two routines guarantee that every recursive call of R will eventually return.

To insure that R is $O(S(n))$ register bounded, each processor of R keeps a tally of how much simulated P storage (in the sense of register bound) it has used. If a simulated instruction would cause this tally to exceed $S(n)$, then the processor aborts the simulation and executes a return instruction. The news of storage overflow is passed up to parent processors in the same way that the news of exceeding depth $D(n)$ is passed up to parent processors.

R clearly has the desired properties. □

The next lemma is from Hartmanis [4]. In [4] it is stated for RAMs but the same proof technique yields the same result for k-PRAMs.

LEMMA 6.4: If a k-PRAM program P is given some input x where $|x| > 0$, it must be the case that after t time units, no register contains a value of y such that $|y| > c|x|2^t$, where c is a constant depending only on P.

THEOREM 6.5: A $T(n)$ time bounded k-PRAM can be simulated by a $O(T^3(n))$ register bounded ordinary RAM. Furthermore, if the original algorithm were deterministic, then the simulating algorithm is deterministic.

Proof. First observe that a $T(n)$ time bounded k-PRAM program will recurse no deeper than $T(n)$. Furthermore, by Lemma 6.4 we know that no register of P will ever contain a value larger than $O(|x|2^{T(n)})$. So for any register R_i used in the computation length$(R_i) = O(T(n))$. Therefore, we know that P is simultaneous $T(n)$ time bounded, $T(n)$ depth bounded and $O(T^2(n))$ register bounded. By Lemma 6.4 and the two conventions that $T(n)$ is $O(T(n))$ time countable and that $T(n) \geq n$, we know that $T(n)$ is $O(T^2(n))$ storage countable. Therefore, by Lemma 6.3 we can conclude that P can be simulated by a $O(T^3(n))$ register bounded ordinary RAM. □

THEOREM 6.6: Let $S(n)$ be any $O(S^3(n))$ time countable function such that $S^3(n) \geq n$. An $S(n)$ register bounded nondeterministic RAM program R can be simulated by a nondeterministic $O(S^3(n))$ time bounded 2-PRAM program P.

Proof. We will prove the theorem by exhibiting a 2-PRAM algorithm P which can simulate the computation of R in time $O(S^3(n))$. Before we consider the

actual algorithm, P, we will make some simplifying assumptions about the operation of the RAM (without losing any generality). First, we will assume that the "halt" instruction operates like a "trap" instruction. That is, once the RAM executes a halt instruction, it continues to reexecute that in-instruction for an infinite period of time. Second, we will assume that if $R_0 = 1$ when R halts, then it has accepted the input; if $R_0 \neq 1$ when R halts, then it has not accepted the input.

We also need to examine the time complexity of algorithm R. Since R is S(n) register bounded, we know there is an N such that for all $n \geq N$: if R accepts an input x of length n then R can accept x within $2^{cS(n)}$ time units, where c is a constant. Since we are assuming that the RAM will continue to cycle on the halt instruction, we know, if x is accepted, then there exists a computation of R on input x such that after exactly $2^{cS(n)}$ time units, P is executing a halt instruction. The algorithm P, on input x, will first compare x and 2^{N+1}. This can be done in one step. If $x < 2^{N+1}$, P will recognize x by table lookup, using a binary search method. This can be done in time O(n). When $x \geq 2^{N+1}$, the algorithm P is more complicated. We will give an intuitive notion of the operation of P in this case. The detailed algorithm is given in Appendix 3.

P will not perform a step-by-step simulation of R. Instead, P will guess at R's memory configuration after it has completed half its computation (hereafter referred to as the "midpoint" configuration). P will pass one of its two offspring the start configuration and the midpoint configuration and it will pass the other offspring the midpoint configuration and a guess of the final configuration. Each of these offspring guess at a configuration halfway between the two they received from their parent and repeat the process of calling two offspring. This procedure is repeated recursively until level cS(n) is reached, whereupon each of the $2^{cS(n)}$ offspring which were generated by this process simulate a single move of R and return to their parent if that single move transformed the first configuration (received from the parent) into the second configuration. After a pair of processors have completed their simulation and returned, their parent checks

to see if the concatenation of the two subcomputations is valid. If so, it returns to its parent as well; otherwise, the computation is aborted. This process is repeated recursively until level 1 is reached; whereupon the input is accepted provided the concatenation of all the guessed instructions is a valid accepting computation of R.

The time bound $O(S^3(n))$ is shown as follows. There are recursive calls to a depth of $cS(n)$. Each call will be preceded by a nondeterministic guess at a storage configuration. This configuration can contain up to $S(n)$ registers and each register can contain a number as large as $2^{S(n)}$. A number as large as $2^{S(n)}$ can be generated by a nondeterministic program in $O(S(n))$ steps. [See Lemma A2.3, Appendix 2.] So the entire configuration of up to $S(n)$ registers can be nondeterministically generated in $O(S^2(n))$ steps. Now there are recursive calls to depth $cS(n)$ and the cost of each recursive call is $O(S^2(n))$. So the total time to generate the $2^{cS(n)}$ offspring is $O(S^3(n))$. The time to compute $S(n)$, simulate the moves of the RAM and pass the results back up to the tree is also clearly bounded by $O(S^3(n))$. So the total run time is $O(S^3(n))$ as desired. □

The main result in this section says that there is a time-storage tradeoff for k-PRAMs. To facilitate the exposition, we will give a notation which expresses a polynomial relationship between two functions. We will say that the function $T_1(n)$ is $P(T_2(n))$ if there exists a constant c such that $T_1(n) \leq T_2^c(n)$ for all but some finite set of non-negative values for n.

THEOREM 6.7: A set is accepted by a $P(T(n))$ time bounded k-PRAM if and only if it is accepted by a $P(T(n))$ storage bounded RAM. Furthermore, we may take either of the machines to be deterministic, even if the other machine is nondeterministic.

Proof. Immediate from Theorem 1 in [7] and Theorems 4.1, 6.5, and 6.6. □

COROLLARY 6.8: The following statements are equivalent, provided $k \geq 2$:

(1) A is accepted by a nondeterministic polynomial time bounded k-PRAM.
(2) A is accepted by a deterministic polynomial time bounded k-PRAM.
(3) A is accepted by a nondeterministic polynomial storaged bounded RAM.
(4) A is accepted by a deterministic polynomial storage bounded RAM.

APPENDIX 1

Program to unpack a string V of T bits and store the bits in an array V[I]

```
I ← 0 ;
B[I] ← 2^0 ;
WHILE  I ≤ T  DO
   BEGIN
      I ← I+1 ;
      B[I] ← B[I-1] + B[I-1] ;
   END ;

Comment:  I=T = the length of V
including the appropriate number of
leading zeros, B[J] = 2^J for
J = 0,1,2,...I=T ;

WHILE  I ≥ 0  DO
   BEGIN
      WHILE  V < B[I]  DO
         BEGIN
            V[I] ← 0 ;
            I ← I-1 ;
         END ;
      V[I] ← 1 ;
      V ← V - B[I] ;
      I ← I-1 ;
   END
```

Appendix 2
Proof of Theorem 5.1

The proofs of many of the lemmas are either omitted or reduced to brief sketches. Complete details of the proofs can be found in [9]. In order to simplify the construction, we will use an intermediate model called a restricted k-PRAM. Informally, a restricted k-PRAM is a k-PRAM with the following additional property. If it calls any one offspring, then it calls all of its offspring and waits for all offspring to return before it either accesses a channel or recalls an offspring. Furthermore, each offspring is blocked from starting its computation until all of its siblings have been called. The formal definition is a bit involved, but is just a formalization of this informal idea. A *restricted k-PRAM* is a k-PRAM such that each processor has associated with it a vector of k bits and such that all computations are restricted to behave as follows: When a processor is called, its vector is set to all zeroes. For $\ell = 1,2,\ldots,k$, if the ℓ^{th} bit is not zero, and the processor tries to call its ℓ^{th} offspring, then it is blocked until the ℓ^{th} bit becomes zero. When the ℓ^{th} bit becomes zero, the ℓ^{th} offspring is called, the ℓ^{th} bit is set to one, the parent processor continues with its computation but that offspring is blocked from starting its computation until the entire vector is all ones. Whenever the vector is all ones, all k offspring begin their computations. The vector gets reset to all zeroes as soon as all the offspring return. If the vector is not all zeroes and the processor tries to access any channel, then it is blocked until the vector becomes all zeroes. Note that if a processor attempts to access a channel or recall an offspring before the processor's vector is either all zeros or all ones, then the processor will become permanently blocked because it will be unable to call the remaining offspring. In this case, we say the processor is *vector-blocked*.

Running times for restricted k-PRAMs are computed exactly the same as running times for ordinary k-PRAMS. In particular, no charge is made for manipulating the associated vectors.

Technically speaking, a restricted k-PRAM is not a special type of k-PRAM, but is a different type of device. However, it is not difficult to show that if a restricted k-PRAM runs in time T(n), then we can find an equivalent ordinary k-PRAM that runs in time O(T(n)); furthermore, if the restricted k-PRAM is deterministic, then the ordinary k-PRAM will also be deterministic.

LEMMA A2.1: If A is accepted by a restricted k-PRAM program P then, for any $h \geq k$, we can find a restricted h-PRAM program P' such that

(1) If P is deterministic, then P' is also deterministic and P' recognizes A.

(2) If P is nondeterministic then P' accepts A and, after at most cT(n) moves of any computation, P' is at level one in a halt state. Here c is a constant depending only on P.

(3) P' runs in time O(T(n)).

(4) No processor of P is ever vector-blocked.

(5) When implemented on an ordinary nonrestricted h-PRAM, P' still has properties (1), (2) and (3).

Proof. P' does a step-by-step simulation of P except that, when P would either become vector-blocked or enter an infinite loop, P' will instead do the appropriate thing. By the "appropriate thing" we mean that: if P is deterministic, then P' rejects the input and if P is nondeterministic, then P' returns to level one and halts in a nonaccepting state. In order to perform this simulation, P' must overcome three problems: (1) it must be able to detect when P would enter an infinite loop, (2) it must be able to detect when P would become vector-blocked, and (3) if the simulation is implemented in the most naive way, then after k recursive calls, P' would always become vector-blocked, since the remaining h-k offspring are never called. In order to overcome the first problem, P' uses a clocking routine like that used in the proof of Theorem 4.1. To overcome the second problem, P' reserves k registers to explicitly implement the k bit vector associated with a processor. To overcome the last problem, P' makes "dummy calls" to the extra h-k processors. If a processor receives a dummy call, then it immediately returns. □

LEMMA A2.2: A T(n) time bounded restricted k-PRAM, P, can be simulated by a restricted 2-PRAM, P', in time $O(T^2(n))$. Furthermore, if the original algorithm were deterministic, then the simulating algorithm will also be deterministic.

Proof. By Lemma A2.1, we may assume that $k = 2^j$ for some $j \geq 2$ and that there is a constant, c, such that, after cT(n) moves, P is guaranteed to be at level one. The 2-PRAM P' does a step-by-step simulation of the k-PRAM P but calls and returns are implemented in a special way. When P' wishes to simulate a call, it does not call one of its offspring but instead saves the parameter list that would have been passed by P to its offspring. After k simulated calls, P' will have saved k such parameter lists. At this point, P' generates $k = 2^j$ descendants by means of j levels of recursive calls. Each of these descendants receives one of these k parameter lists. The k descendants of the P' processor then simulate the k offspring of the simulated P processor. When one of these k descendants wishes to simulate a return, it returns the simulated parameter list that the P' processor would return. The P' processors at these j intermediate levels wait for both of their offspring to return and then return the parameter lists returned by these offspring. In this way the k simulated returned parameter lists eventually get returned to the processor that simulated the k calls. Since every computation of P eventually returns to level one, we know that all calls will be returned. So P' is guaranteed to complete its simulation of P. It remains to estimate the run time of P'.

Each processor of P has k offspring processors which it may call again and again. Although each of these k processors may be called many times, we may consider them to be just k individual processor units. So we can consider P to consist of a rooted k-branch tree of processors with the level one processor at the root node. In a similar way P' can be considered to be a rooted binary tree of processors. Some of these P' processors simulate processors of P and some serve as intermediate processors which do nothing except pass parameter lists up and down. Call the former type of P' processors *simulation processors* and call the latter type *liaison processors*. Consider a computation of P on an input of length n and consider

the corresponding P' computation. Because of the time consumed by the liaison processors and because of the bookkeeping involved, some simulation processors will complete their simulation at a somewhat faster rate than other simulation processors. So, if two processors of P execute instructions at the same time, it does not follow that the corresponding simulation processors will be simulating these instructions at the same time. To avoid any analysis problems due to this lack of synchronization, we will assume that the processors of P' are slowed down, by some outside agent, so that steps that are concurrent in the P computation are simulated concurrently in the P' computation. (Of course, this outside agent only slows processors down by the minimum amount of time needed to insure synchronization.) We will denote this slowed-down version of P' by P". Surely P" runs no faster than P'. So it will suffice to show that P" runs in time $O(T^2(n))$. We will show that, if a processor of P takes s time units to execute an instruction, then the corresponding P" processor can simulate the instruction in time $O(sT(n))$, including any time needed to wait for parameters to be passed by the liaison processors. Suppose that the instruction is a call instruction which takes s time units in P, including any time units during which it is blocked because an offspring has not yet returned. In P" this instruction is simulated by saving the parameter list and, if it is the k^{th} parameter list to be saved, by a call that passes all k-saved parameter lists. If all that P" need do is to save the parameter list, then the simulation can be done in time O(s). In the case where k parameter lists are passed, the P" processor must pass up to T(n) values down through j levels of liaison processors. This takes time O(T(n)). So, in any case, the simulation can be done in time O(T(n)). Next, consider return instructions. We will charge the time spent passing parameter lists up through liaison processors to the simulated parent's channel access instructions rather than to the simulated offspring's return instructions. So if a return costs s time units in P, then it will cost only O(s) time units to simulate it in P". A channel reference that takes s time units in P, including any time during which the processor is blocked, can be simulated in time O(s) plus the time needed for parameters to be passed up

through the liaison processors. Since there are at most T(n) values passed up by each of the k offspring, it will take at most O(T(n)) time units to pass the parameters up. Thus the cost of simulating a channel access is O(s+T(n)), and so certainly is O(sT(n)), where s is the cost of the instruction in P. All the remaining instructions can be simulated in constant time. Thus any instruction that takes time s in P can be simulated in time O(sT(n)) by P". So P", and hence P', run in time $O(T^2(n))$. □

LEMMA A2.3: There is a (nondeterministic) RAM program such that the following holds for all x and t: If $(0 < x < 2^t$ and $t \geq 1)$, then there exists a computation of that program which requires no more than 6t+5 time units to compute x.

Proof: We give such a program, P:

```
 1.          I←0
 2.          A←0
 3.          B←1
 4.  LOOP:   IF I ≥ t GOTO DONE
 5.          GOTO ADDIN
 6.  ADDIN:  A←A+B
 7.  ADDIN:  I←I+1
 8.          B←B+B
 9.          GOTO LOOP
10.  DONE:   HALT AND OUTPUT A
```

□

LEMMA A2.4: A nondeterministic T(n) time bounded k-PRAM, P, can be simulated by a nondeterministic restricted k-PRAM, P', in time $O(T^3(n))$.

Proof: P' operates as follows. Each processor of P' will perform a step-by-step simulation of the corresponding P processor except that calls and returns are handled in a special way. The level one processor of P' will first guess at the number of simulated calls it will make to each of its offspring, what parameters will be passed to the offspring, what values will be returned by these recursive calls and which of these returned parameter lists it will ever access. It then calls all k of its offspring, telling them how many subcomputations they should simulate and what the passed

parameter lists are. It then proceeds with its step-by-step simulation but, instead of making recursive calls and accessing channels, it uses the guessed-at parameter lists. If it wishes to simulate a call and the guessed-at parameter list for the call is incorrect, then the computation is aborted. If the guessed-at parameter list is correct, then no call is made but the simulation proceeds with the next step. If it wishes to simulate accessing a channel, it reads the value from the appropriate guess of the returned parameter list. Meanwhile, the k offspring simulate the corresponding offspring of P on the guessed-at parameter lists but, instead of returning parameter lists, they save the parameter lists they would have returned. When an offspring has completed all the simulations required of it, it then passes all the simulated returned parameter lists to its parent. The parent completes its simulation up to a simulated return instruction. The parent then compares its guesses at parameter lists with the parameter lists actually returned by its offspring. If they do not match, then it aborts its computation. If they do match, then it proceeds with its computation and returns the computed value. Lower level processors proceed in the same way except for two points. First, they do not return any parameter lists until they have completed all the simulations required of them. At that point, they return all their simulated returned parameter lists. Second, they do not bother to do a simulation if the parent guessed that it would not access the values returned from that call. Clearly, P' accepts (respectively rejects) exactly the same inputs that P accepts (respectively rejects). It remains to estimate the run time of P'.

We will show that it takes $O(T^2(n))$ time units between the time that a P' processor starts to simulate one time unit of P and the time it starts to simulate the next time unit of P. Since there are T(n) time units to simulate, it will then follow that P' runs in time $O(T^3(n))$. Consider a P' processor that is simulating a P processor from the time it is called until the time it returns. Charge the time for guessing at its offsprings' parameter lists and for passing these parameter lists to the first simulated time unit. Charge the time for returning and checking these parameter lists to the last simulated time unit. If the input is of length n, then there need be a total of at most T(n) numbers in any such list of parameter lists. By Lemma

6.4, each number need be of size at most $0(2^{cT(n)})$, for some constant c. By Lemma A2.3, each number can be nondeterministically generated in time $0(T(n))$. So the total time to guess and pass such a list of parameter lists is at most $0(T^2(n))$. So the first time unit can be simulated in $0(T^2(n))$. The returning and checking can certainly be done in time $0(T^2(n))$. So the last time unit can be simulated in time $0(T^2(n))$. All other time units can be simulated in a constant number of time units. So every P time unit can be simulated in $0(T^2(n))$ time units as promised. □

Theorem 5.1 now follows directly from Lemmas A2.2 and A2.4.

APPENDIX 3

2-PRAM program for the proof of Theorem 6.6

Before giving the detailed algorithm for P, we will first describe some of the data structures used in the simulation. We will need two special registers: "LEVEL" (containing the level of recursion) and "MAX" (containing the value of cS(n)), which will be the maximum level in the computation. The memory configuration of R will be represented by a list of pairs of registers. The first element in the pair will be the register address and the second element will be that register's contents. Since we do not store registers whose content is zero, a zero value will indicate the end of the list. We will refer to this entire list as an R-*configuration*. We now describe the operation of algorithm P on some input x.

1. *Initialization:* At level 1 (detected by the fact that LEVEL = 0) P will first compare x and 2^{N+1}. Then P will do one of the following:
   (a) If $x < 2^{N+1}$, P will recognize x by table lookup using a binary search.
   (b) If $x \geq 2^{N+1}$, P will construct cS(n) in register MAX and nondeterministically guess at the final R-configuration. It then continues with step 2.

2. *Entry:* At all levels, upon entry, P must increment LEVEL by 1 and compare it to MAX. If LEVEL = MAX, then P must simulate an instruction of R (go to step 4). If LEVEL < MAX, then P must divide its task between its two offspring (go to step 3).

3. *Computation Reapportionment:* P will now divide its subcomputation between its two offspring so that the first will simulate the first half and the second will simulate the second half. First P nondeterministically guesses at a new R-configuration, which should be the configuration of R after it has reached the point in its computation

corresponding to the midpoint of this processor's subcomputation. Then P calls its first offspring, passing it the first R-configuration it received from its parent and the guessed midpoint R-configuration. Then P calls its second offspring, passing it the guessed midpoint R-configuration and the second R-configuration it received from its parent. It now waits for its offspring (at step 5).

4. *Simulation:* If LEVEL = MAX, P must simulate an instruction of R. First, P guesses at a location pointer into the program R (we will denote this pointer by p), then it executes this instruction, updating the first R-configuration it received from its parent. If this instruction transforms the first R-configuration it received into the second R-configuration it received and changes the location pointer to q, then P returns the pair (p,q) to its parent; otherwise, the computation is aborted.

5. *Verification:* When both offspring return, P must verify that a valid instruction sequence has been followed by its descendants. Assume that the instruction pointer pair (p_1,q_1) was returned by the first offspring and the pair (p_2,q_2) was returned by the second. If $q_1 \neq p_2$, then the subcomputation of the second offspring did not start with the next instruction dictated by the simulated computation of the first offspring, so the computation will be aborted. Otherwise, the computations match and so P returns the instruction sequence pair (p_1,q_2) to its parent (if LEVEL $\neq$ 1; when LEVEL = 1, see step 6).

6. *Termination:* If the computations match (in step 5) and LEVEL = 1 and $p_1 = 1$ (that is, the first instruction to be executed is the first instruction in the program) and $q_2 = i$, where i is the location of a halt instruction, then P accepts the input if and only if $R_0 = 1$ in the final R-configuration (recall that we assumed R halts with $R_0 = 1$ if and only if it accepts the input).

REFERENCES

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

[2] Cook, S.A., and Reckhow, R.A. "Time bounded random access machines," JCSS 7 (1973), 354-375.

[3] Glushkov, V.M., Ignatyev, M.D., Myasnikov, V.A., and Torgashev, V.A. "Recursive machines and computing technology," Proc. of the IFIP Congress on Information Processing, August 1974, 66-70.

[4] Hartmanis, J. "Computational complexity of random access stored program machines," Math. Systems Theory 5 (1971), 232-245.

[5] Hartmanis, J., and Simon, J. "On the power of multiplication in random access machines," Proc. of the 15th Annual IEEE Symposium on Switching and Automata Theory, New Orleans, October 1974, 13-23.

[6] Hopcroft, J.E., and Ullman, J.D. Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Massachusetts, 1969.

[7] Savitch, W.J. "Relationships between nondeterministic and deterministic tape complexities," JCSS 4, (1970), 177-192.

[8] Savitch, W.J., and Stimson, M.J. "The complexity of time bounded recursive computations," Proc. of the 1976 Conference on Information Sciences and Systems, The Johns Hopkins University, Baltimore, Maryland, April 1976.

[9] Stimson, M.J., The Complexity of Parallel Algorithms, Ph.D. dissertation, Information and Computer Science, Univ. of California, San Diego, 1976.