PARALLELISM IN RANDOM ACCESS MACHINES

Steven Fortune* and James Wyllie*
Cornell University
Ithaca, N.Y. 14853

Abstract.

A model of computation based on random access machines operating in parallel and sharing a common memory is presented. The computational power of this model is related to that of traditional models. In particular, deterministic parallel RAM's can accept in polynomial time exactly the sets accepted by polynomial tape bounded Turing machines; nondeterministic RAM's can accept in polynomial time exactly the sets accepted by nondeterministic exponential time bounded Turing machines. Similar results hold for other classes. The effect of limiting the size of the common memory is also considered.

1. Introduction

The speed of serial computers has increased enormously over the past few decades. Unfortunately, due to ultimate physical limitations, these increases cannot continue forever. An alternative strategy for increasing speed is to perform as much of the desired computation as possible in parallel. Considerable current research in semantics and programming languages is directed at the interaction of parallel processes. However, there seems to have been less study made of the computational power of parallel machines. We present a model for parallel computation and relate its power to that of more familiar models. We believe this model captures the spirit of the parallel computers that are likely to be built in coming years without being constrained by the engineering considerations that have led to existing machines such as ILLIAC IV [1]. We note that the model, or something very similar to it, has been used implicitly by authors who have developed parallel algorithms for a variety of problems [4], [5], [7].

Our main results are the following. We show that deterministic parallel processors can accept in polynomial time exactly the class of sets accepted by polynomial tape-bounded Turing machines; thus parallelism can give more than a polynomial increase in computing speed if and only if PTIME≠PSPACE. Nondeterministic parallel processors can accept in polynomial time exactly the sets accepted by nondeterministic exponential time-bounded Turing machines; hence parallelism does in fact give an exponential speedup in this case. We have similar results for other time classes.

2. The Model

A parallel random access machine (P-RAM) consists of an unbounded set of processors P_0, P_1, ..., an unbounded global memory, a set of input registers, and a finite program. Each processor has an accumulator, an unbounded local memory, a program counter, and a flag indicating whether the processor is running or not. All memory locations and accumulators are capable of holding arbitrary nonnegative integers. The program consists of a sequence of possibly labelled instructions chosen from the list below. A program is nondeterministic if some label occurs more than once, deterministic otherwise.

Instruction		Function
LOAD	operand	Perform the indicated operation using the accumulator of the processor executing the instruction.
STORE	operand	
ADD	operand	
SUB	operand	
JUMP	label	Change program counter to label.
JZERO	label	
READ	operand	See text.
FORK	label	See text.
HALT		See text.

Each operand may be a literal, an address, or an indirect address. Each processor may access either global memory or its local memory, but not the local memory of any other processor. Indirect addressing may be through one memory to access another.

Initially the input to the P-RAM is placed in the input registers, one bit per register, all memory is cleared, the length of the input is placed in the accumulator of P_0, and P_0 is started. At each step in the computation each

*Research supported in part by the Office of Naval Research under grant number N00014-76-C-0018.

running processor simultaneously executes the instruction given by its program counter in one unit of time, then advances its counter by one unless the instruction causes a jump.

A READ instruction uses the value of the operand to specify one of the input registers; the contents of the selected register is placed in the accumulator. A FORK label instruction executed by processor P_i selects the first inactive processor P_j, clears P_j's local memory, copies P_i's accumulator into P_j's accumulator, and starts P_j running at label. A HALT instruction causes a processor to stop running.

Simultaneous reads of a location in global memory are allowed, but if two processors try to write into the same memory location simultaneously, the P-RAM immediately halts and rejects. Several processors may read a location while one processor writes into it; all reads are performed before the value of the location is changed.

Execution continues until P_0 executes a HALT instruction (or two processors attempt to write into the same location simultaneously). The input is accepted only if there is some computation in which P_0 halts with a one in its accumulator; the time required to accept the input is the minimum over all such computations of the number of instructions executed by P_0. A language L is in the class deterministic (nondeterministic) T(n)-time-P-RAM if there is a deterministic (nondeterministic) P-RAM M such that for all words x of length n, x is in L if and only if x is accepted by M and requires time at most T(n).

Several details of the model deserve further comment. We choose to use special registers to hold the input in order to be able to discuss sublinear running times. If, say, the input were encoded in binary and placed in the accumulator of P_0, then it would take at least linear time (in the length of the input) to unpack the input. Also, we could restrict the local memory of each processor to only a constant number of locations by mapping the local memory of each processor into global memory. However, we will be interested in what happens when the size of global memory is restricted, hence we allow each processor unbounded local memory.

3. Main Theorems

In the following, resource bounds unqualified by "P-RAM" refer to Turing machine computations.

Theorem 1 (deterministic P-RAM's) For $T(n) \geq \log n$

$$\bigcup_{k=1}^{\infty} T(n)^k\text{-time-P-RAM} = \bigcup_{k=1}^{\infty} T(n)^k\text{-space}$$

In particular, $\bigcup_k \log^k n$-time-P-RAM $= \bigcup_k \log^k n$-space

and polynomial-time-P-RAM = PSPACE.

Theorem 2 (nondeterministic P-RAM's) For $T(n) \geq \log n$

$$\bigcup_{c>0} \text{nondet-}cT(n)\text{-time-P-RAM} = \bigcup_{c>0} \text{nondet-}2^{cT(n)}\text{-time}$$

In particular, nondeterministic $c \cdot \log n$-time-P-RAM = NP and nondeterministic-polynomial-time-P-RAM = nondeterministic exponential time.

These theorems follow from four simulations given in the lemmas below. Before giving the proofs, we mention a few programming details.

One processor can initiate two other processors in constant time using the FORK instruction. By iterating this, a "tree" of m processors can be initiated in time O(log m). If processor i initiates processor j to perform a subtask, it can pass via its accumulator the address in global memory of a block containing parameters. Among these parameters can be the address of a location in global memory where processor j is store its result. Processor i can then repeatedly test the location to determine when processor j terminates.

Memory interference between processors in global memory can be circumvented by interleaving the locations assigned to each processor. Suppose a processor has free storage consisting of every k locations starting at address m. If the processor then initiates two new processors, it can assign one every 2k locations starting at m and the other every 2k locations starting at m+k.

Lemma 1a Let L be accepted by a deterministic T(n) space-bounded TM M, for $T(n) \geq \log n$. Then L is accepted by a deterministic cT(n) time bounded P-RAM P, for some constant c.

Proof We will first present a simulation of M by P which assumes that the value of T(n) is available, then show how to remove this assumption. Given T(n), P will construct a directed graph where each node represents a configuration of M. The number of configurations of M is bounded by $2^{dT(n)}$ for some d depending on M, hence so is the number of nodes. Leaving each node will be a single edge to the node of its successor configuration. Accepting configurations of M are their own successors. Thus there is a path from the initial configuration node to an accepting configuration node if and only if M accepts its input within T(n) space. To build the graph, P first initiates $2^{dT(n)}$ processors in O(T(n)) steps, each holding a different integer, representing each possible configuration of M. Each processor then, in O(T(n)) time, unpacks its configuration integer into local memory, computes the successor configuration, and packs the result into a single integer. The graph is then stored in global memory, the address of a configuration node being the integer representing the configuration. To find the endpoint of the path from the initial configuration, the following is executed iteratively. Each processor finds the successor of the successor of its configuration node, then stores that as its successor in global memory. After i iterations, each node has its descendant at distance 2^i stored as its successor. Since the terminal configuration is at distance at most $2^{dT(n)}$ from the initial configuration, only dT(n) iterations are needed. As

each iteration takes constant time, and the rest of the simulation O(T(n)), the total time is cT(n), for some c.

To avoid the constructibility assumption about T(n), modify the above simulation as follows: P_0 will start processors one at a time which will execute the procedure above assuming T(n) = 1,2, ... When any of these simulations succeed, P_0 will be notified and will accept. The order of the running time is unchanged, since it requires only O(T(n)) time to start the simulation which uses the correct guess for T(n). The individual simulations must also be modified so that they each use disjoint locations in global storage to store their configuration graphs. This can be accomplished by increasing each address by the assumed value of $2^{T(n)}$. Details are left to the reader.

Lemma 1b Let L be accepted by a deterministic T(n) time-bounded P-RAM. Then L is accepted by $T(n)^2$ space-bounded TM.

Proof We will first construct a nondeterministic $T(n)^2$ space-bounded TM accepting L. We will then show how to make the TM deterministic without increasing the space bound.

In order to determine whether the P-RAM accepts its input, the TM needs to know the contents of P_0's accumulator when it halts and to verify that no two writes occur simultaneously into the same global memory location. The simulation is based on a recursive procedure ACC which checks the contents of a processor's accumulator at a particular time. By applying the procedure to P_0, we can determine if the P-RAM accepts. ACC is similar to the procedure FIND in [5].

ACC will check that at time t, processor P_j executed the i^{th} instruction of its program, leaving c in its accumulator. In order to check this, ACC needs to know

i) the instruction executed by P_j at time t-1 and the ensuing contents of its accumulator, and

ii) the contents of the memory location(s) referenced by instruction i.

ACC can nondeterministically guess (i) and recursively verify it. To determine (ii), for each memory location m referenced, ACC guesses that m was last written by some processor P_k at time t' < t. ACC can recursively verify that P_k did a STORE of the proper contents into m at time t'. ACC must also check that no other processor writes into m at any time between t' and t. It can do this by guessing the instructions executed by each processor at each such time, recursively verifying them, and verifying that none of the instructions changes m. Checking that two writes do not occur into the same memory location simultaneously can be done in a similar fashion. For each time step and each pair of processors, ACC nondeterministically guesses the instructions executed, recursively verifies them, and checks that the two instructions were not writes into the same location.

The correctness of the simulation follows from the determinism of the P-RAM. In general, each instruction executed by the P-RAM will be guessed and verified many times by ACC. However, since the P-RAM is deterministic, at any step for each running processor there is exactly one instruction which can be executed; thus all verified guesses of that instruction must be identical.

To analyze the space requirements, note that there can be at most $2^{T(n)}$ processors running after T(n) steps, so writing down a processor number takes T(n) space. Since addition and subtraction are the only arithmetic operators, numbers can increase in length by at most one at each step. Thus, writing down the contents of an accumulator takes at most T(n) + log n = O(T(n)) space. Writing down a time step takes log T(n) space and the program counter requires only constant space. Hence the arguments to a recursive call of ACC can be written down in T(n) space. Cycling through time steps and processor numbers to verify that a memory location was not overwritten also takes only T(n) space, hence the total space requirement at each level is T(n). As the total depth of recursion is T(n), the total space required is $T(n)^2$.

Note that the simulation can be performed directly by a deterministic Turing machine. At each step in the simulation outlined above where a nondeterministic guess was made, the Turing machine can deterministically cycle through all possible outcomes, until the correct one is found. This requires no more space.

Lemma 2a Let L be accepted by a nondeterministic T(n) time-bounded TM. Then L can be accepted by a nondeterministic c•log T(n) time-bounded P-RAM, for some constant c.

Proof For an input of size n accepted by the TM, the length of an accepting sequence of configurations is at most $T(n)^2$. In d•log T(n) steps, for some d, enough processors can be activated so that each can guess one symbol of the computation. The first n processors check that the initial configuration corresponds to the initial configuration of the TM on the input; each of the remaining processors verifies that its symbol follows from the corresponding symbol and the ones on either side of it in the preceding configuration. The processors for the last configuration must also check that it is an accepting configuration. In another d'•log T(n) steps, for some d', the information that the computation is correct can be bubbled up to P_0, and P_0 can accept.

We note that T(n) need not be constructible, as the length of the accepting computation can be guessed nondeterministically.

Lemma 2b Let L be accepted by a nondeterministic T(n) time-bounded P-RAM, $T(n) \geq \log n$. Then L is accepted by a nondeterministic $2^{cT(n)}$ time-bounded TM, for some constant c.

Proof The brute force simulation of the P-RAM suffices. There can be at most $2^{T(n)}$ processors, each of which can access at most T(n) memory locations. The contents and address of each memory location can be of size at most $T(n) + \log n = O(T(n))$, since numbers at most double at each step and P_0's accumulator originally contains n. Hence the total tape required is $2^{T(n)} \cdot T(n)^2$. The TM can simulate one step of one processor in one scan of its tape; $2^{T(n)}$ processors with T(n) steps each takes time $2^{2T(n)} \cdot T(n)^3 < 2^{cT(n)}$ for some c.

4. Memory Limited Computations

Theorems 1 and 2 seem to depend on the ability to communicate exponential amounts of information through global storage. Thus it seems natural to ask what happens when global storage is restricted to a polynomial in the running time. We can characterize the power of nondeterministic P-RAM's with restricted storage and can partially characterize the power of similarly contrained, but deterministic, P-RAM's. The theorems below are stated for the familiar classes NP and PSPACE, but as with the earlier theorems, similar results hold for higher and lower complexity classes. For the remainder of this section, all P-RAM's are assumed to possess only a polynomial number of global storage locations.

Theorem 3 The class of sets accepted by nondeterministic polynomial time-bounded, polynomial global storage-bounded P-RAM's is identically PSPACE.

Proof The key to the P-RAM's simulation of a PSPACE bounded Turing machine M is a recursive subroutine TEST(i,j,t) which verifies that M's configuration j follows from configuration i within d^t steps, for some constant d depending on M. TEST works by guessing the configuration k midway between i and j, then forking to execute TEST(i,k,t-1) and TEST(k,j,t-1) in parallel. The middle configuration k can be guessed using a processor's local storage, and parameters can be packed into the accumulator prior to the fork, so no global storage is required to perform all of the parallel subroutine calls. There is not sufficient global storage for each call of TEST to bubble its result back to its father as in lemma 2a, so an alternate strategy must be employed. When a processor determines that TEST(i,j,t) is false, it causes the P-RAM to reject by creating two sons, each of which does a store into global memory location 0. If TEST(i,j,t) is found to be true, the processor merely halts. The root processor P_0 computes D(n), an upper bound on the depth of recursion and calls TEST(initial configuration, final configuration, D(n)). It then waits long enough for D(n) levels of recursive calls to TEST to complete and accepts. If P_0 accepts, then no processor found a mistake or attempted a recursive call deeper than D(n), so M must have accepted its input. D(n) may be taken to be a polynomial since M is PSPACE bounded. The processing at each call to TEST, exclusive of recursive calls, is proportional to the length of a configuration, a polynomial, so the P-RAM runs for time at most some polynomial in the length of the input.

To simulate a global memory limited, NP-time-P-RAM within NPSPACE (hence PSPACE), first observe that although exponentially many processors may be active at once, only polynomially many of them may write into global memory on any step of the computation. Thus, a nondeterministic Turing machine may guess and write down within polynomial space the entire contents of global memory after each step of the computation, along with documentation of which processors modified which storage locations at each step. The Turing machine can then traverse the tree of activated processors implied by FORK instructions, and verify that all instructions executed are consistent with the guessed global memory contents and conversely. If this implied tree is traversed so that only the local memories of the processors directly on the path back up to P_0 are stored at any time, the simulation can be carried out within polynomial space.

Theorem 4 The class of sets accepted by deterministic polynomial time-bounded, polynomial global storage-bounded P-RAM's contains the class co-NP.

Proof It is sufficient to show how to accept the complement of some NP-complete set deterministically on a P-RAM. We show how to accept the set of Boolean formulas which are not satisfiable. In $c_1 n$ time, 2^n processors can be activated, each holding a different integer in the range 0 to 2^n-1. In an additional $c_2 n$ time each processor can unpack its integer into its local memory and determine if the assignment of truth values represented by the bits of its integer cause the input formula to be satisfied. Processors finding a satisfying assignment force the P-RAM to reject as in Theorem 3. If no processor has forced rejection after $(c_1+c_2)n$ time, P_0 accepts the input.

Notice that we cannot accept the set of satisfiable Boolean formulas by trying all truth assignments in parallel and setting a flag in global memory if any satisfying assignment is found, because there may be more than one such satisfying assignment, resulting in several processors trying to set the global flag simultaneously. If an NP-complete set could be found such that each member of the set had exactly one "certificate", then memory limited deterministic polynomial time P-RAM's could accept at least NP ∪ co-NP.

5. Discussion

We can compare our results to others that have been obtained for nontraditional machines modeling some aspects of parallelism. Several authors have constructed models for which deterministic and nondeterministic polynomial time are equivalent and are equal to PSPACE on a Turing machine. These include Hartmanis and Simon [5] and Pratt and Stockmeyer [9] using RAM's augmented by instructions that can manipulate exponentially

large numbers at unit cost; Kozen [8] and Chandra and Stockmeyer [2], using alternating Turing machines (nondeterminism is subsumed by alternation, hence adds no power); and Savitch and Stimson [11], using parallel RAM's without global memory. While we do not know that nondeterminism is more powerful than determinism on our model, settling the question would decide PSPACE = nondeterministic exponential time. Savitch [12] has also proven a result similar to our Theorem 2 for nondeterministic parallel RAM's without global memory augmented by list processing instructions that can manipulate exponential amounts of information in unit time. The power of the deterministic version of his machines is still open.

It is appropriate to note at this point that although we have used the uniform cost criterion [3] throughout this paper, our results still hold (at most squaring the simulation cost) if the logarithmic cost criterion is used. This is in contrast to the results mentioned above for RAM's which depend on unit time manipulation of large numbers. Similarly, we could charge memory accesses at a cost proportional to the logarithm of the size of the global memory, with only a polynomial increase in computing time.

Finally, we observe that other open problems in computational complexity can be rephrased in terms of parallel RAM's. For example, we have shown that any problem in NP can be sped up exponentially using nondeterministic parallelism. A proof that such exponential speedups are not obtainable by applying deterministic parallelism to problems in P would settle the question

$$\bigcup_k \mathrm{LOG}^k\mathrm{SPACE} = \mathrm{P}$$

in the negative.

References

[1] Barnes, G.H., et.al. "The ILLIAC IV Computer", IEEE Trans. Computers. C-17 (Aug. 1968), pp. 746-757.

[2] Chandra, A.K. and L.J. Stockmeyer. "Alteration", Proc. of the 17th Annual IEEE Symposium on Foundations of Computer Science, Houston, Texas, Oct. 1976, pp. 98-108.

[3] Cook, S.A. and R.A. Reckhow. "Time Bounded Random Access Machines", JCSS 7 (1973), pp. 354-375.

[4] Csanky, L. "Fast Parallel Matrix Inversion Algorithms",Proc. of the 16th Annual Symposium on Foundations of Computer Science, Berkeley, California, Oct. 1975, pp. 11-12.

[5] Hartmanis, J. and J. Simon. "On the Power of Multiplication in Random Access Machines", Proc. of the 15th Annual IEEE Symposium on Switching and Automata Theory, New Orleans, Oct. 1974, pp. 13-23.

[6] Hirschberg, D.S. "Parallel Algorithms for the Transitive Closure and the Connected Component Problems", Proc. of the 8th Annual ACM Symposium on Theory of Computing, Hershey, Penn., May 1976, pp. 55-57.

[7] Kogge, P.M. "Parallel Solution of Recurrence Problems", IBM J. Res. Develop. 18 (March 1974), pp. 138-148.

[8] Kozen, D. "On Parallelism in Turing Machines", Proc. of the 17th Annual IEEE Symposium on Foundations of Computer Science, Houston, Texas, Oct. 1976, pp. 89-97.

[9] Pratt, V.R. and L.J. Stockmeyer. "A Characterization of the Power of Vector Machines", JCSS 12 (1976), pp. 198-221.

[10] Savitch, W.J. "Relationships between Nondeterministic and Deterministic Tape Complexities", JCSS 4 (1970), pp. 177-192.

[11] Savitch, W.J. and M.J. Stimson. "Time Bounded Random Access Machines with Parallel Processing", Sept. 1976, (revised Aug. 1977)., technical report, Dept. APIS, University of California, San Diego, 78-CS-011.

[12] Savitch, W.J. "Parallel and Nondeterministic Time Complexity Classes", November 1977, technical report, Dept. APIS, University of California, San Diego, 78-CS-012.