Abstract And-Parallel Machines*

Nachum Dershowitz and Naomi Lindenstrauss

Department of Computer Science, The Hebrew University, Jerusalem 91904, Israel

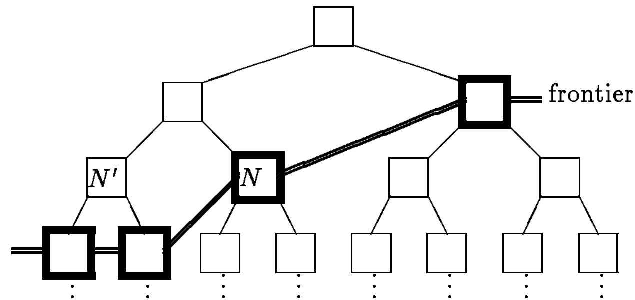
The deterministic Turing machine, though abstract, can still be seen as a model of a realistic computer. The same cannot be said for the nondeterministic Turing machine as a model of parallel computing. We introduce several abstract machines with fine-grained parallelism—the *and-parallel Turing machine*, the stronger *parallel rewriting machine*, and extensions of both with an *interrupt capability.* These machines are very powerful: the parallel rewriting machine can compute the permanent in polynomial time and the and-parallel machine with interrupt can simulate nondeterministic and alternating Turing machines of polynomial time complexity in polynomial time. All the same, they may be viewed as realistic models if time and space are suitably restricted.

And-Parallel Turing Machines. One way of viewing the nondeterministic Turing machine is to say that at each stage, when confronted with k choices, it splits into k replicas of itself, each of which makes one of the choices. The input string is accepted if one of these machines enters an accepting state. This kind of parallelism can be termed "or-parallelism". However, the nondeterministic Turing machine does not model real parallel machines in which different processors communicate with each other. Also, when faced with an input of size n that must be read, it will need at least n units of time. But by then the machine may have arrived at an exponential number of possibilities, an unrealistic scenario.

Our *and-parallel machine* works with an unbounded binary tree. (We take a binary tree for simplicity—any bounded arity will do.) Instead of the square scanned by the "head" (representing the memory cell dealt with at the moment by a sequential computer), the tree has a "frontier" of nodes, representing the memory cells dealt with at the moment by the processors comprising the parallel computer. (By "frontier" we mean a subset of the binary tree such that none of its nodes is an ancestor of another, while every node in the binary tree is either an offspring or an ancestor of a node in the frontier.) In the beginning the frontier consists of the root node. The input is a finite binary tree. Each node in the frontier is in one of a finite number of states, and can be thought of as being taken care of by a different processor. Given a binary tree with frontier, the transition function δ acts on its frontier nodes and transforms it into another binary tree with frontier. It may change the content of the nodes of the frontier in the following way: if N is a frontier node in state q with symbol t, δ can change the symbol at N. According to q and t, N may remain a frontier node, although its state may change. It can also be replaced in the frontier by its two children, who get states determined by δ. If q and t are suitable, and N's sibling N' also has suitable symbol and state, then both N

* The full version of this extended abstract can be found on the World Wide Web at `http://www.cs.huji.ac.il/~naomil`.

and N' are removed from the frontier and their parent enters the frontier with a state determined by δ according to the states and symbols of its children. This step models communication between different processors. If N is ready to be replaced by its parent and N' is not (see the following figure), it waits for N'. This models suspension and synchronization. If N has a blank symbol, then it may be overwritten with a nonblank, and have two children appended. At each stage, δ performs all the actions it can perform on the nodes of the frontier according to the above rules. We use deterministic rules, so the machine is deterministic. If there is a frontier node in a non-accepting state on which δ cannot act, the machine stops with failure. If a node reaches an accepting state it will remain in it. If all the nodes of the frontier are in an accepting state the machine accepts the input.



Throughout this paper, we make the following assumptions:

Assumption 1: *An unlimited number of processors is available.* Of course there is a real-life limit to the number of processors, but we can assume that this is transparent to the user (just as physical limitations on the size of memory are ignored by the deterministic Turing machine model). If the depth of the tree is $O(\log n)$, where n is the input size, then the number of processors will be polynomial in n (hence problems that can be solved on such a machine in polylogarithmic time and space of logarithmic depth are in NC).

Assumption 2: *The input should be given in a form suitable for parallel processing, that is, lists have to be replaced by balanced trees.* If the depth of the input is not logarithmic in its size, just looking at it will take too much time.

Assumption 3: *The memory on which the processors work is organized as a tree and not as a directed acyclic graph (dag).* Clearly this simplifies matters. Though it may seem that a dag is more economical, because it avoids having the same computation performed several times, on the average not that much is saved. Under a variety of probabilistic models, a tree of size n has a maximally compacted form as a dag (where all common subtrees are represented only once) of expected size (asymptotically) $cn/\sqrt{\log n}$, where c is a constant depending on the type of trees compacted and the statistical model. Though this represents a drastic savings in storage, what is important in our case is the logarithm of the size, for which the savings is only marginal ($\log\log n$).

Assumption 4: *On each branch of the memory tree there is at most one node*

on which a processor works at any given time. Again this simplifies matters, since it will never happen that two processors will attempt to work on the same node, and is enough for good performance.

Parallel Rewriting Machines. We would like to have a parallel machine that is easy to program yet stronger than the and-parallel machine, for instance one that can sort in polylogarithmic time. Assume that the input to the machine is given as a tree and that its behavior is determined by "flat" conditional rewrite rules of the form $C \quad | \quad L \quad \longrightarrow \quad R$ where L is a tree "pattern" containing variables representing subtrees, R is a replacement pattern for L, and C is some simple property of the tree that can be checked locally at the location in the tree where L appears. The computational paradigm associated with rewriting has the full power of Turing machines, while lending itself naturally to parallel execution, since different subtrees can be rewritten in parallel. The advantage of this formalism is that machine operations are very straightforward (almost as simple as for a Turing machine), and the machine gets the problem in a form that preserves its "meaning". A tree to which no rewrite rule can be applied is said to be in *normal* form, and computation essentially amounts to bringing a tree to its normal form. Of course there is no guarantee that the rewriting process will terminate, and even if it does, that the normal form will be unique. Different strategies for rewriting may be used (under certain well-defined conditions, the order does not affect the final result) and in a parallel environment several steps of rewriting may be performed simultaneously.

The *parallel rewriting machine* implements a top-down strategy by first trying to rewrite a tree by a rule that applies at its root, and only if this is not possible it tries to rewrite the children of the root, and so on. After the children of a node have been rewritten to normal form, rewriting continues at the parent node. The machine has an unbounded tree-tape available. In the beginning this tree consists of the input. At each stage there is a frontier of nodes at which processing is taking place. These nodes can be in any of five states: *up, down, going-up, going-down, stopped.* In the beginning the frontier consists of the root node in the state *going-down.* As long as there is a rule which can be applied at the root this rule is applied, possibly causing some local change in the form of the tree. If no rule can be applied the node goes into state *down.* A node in state *down* is replaced in the frontier by its children, each in state *going-down.* Now the machine tries to apply rules at the nodes corresponding to the children. Thus the processing continues until it arrives at nodes that cannot be rewritten and have no children. Such nodes go into state *up.* If all the children of a node are in state *up*, they are replaced in the frontier by their parent node in state *going-up.* If a rule can be applied to the node in state *going-up* it is applied and the node goes into state *going-down.* Otherwise the node goes into state *up.* If the root node is in state *up* it goes into state *stopped.* In this case the computation terminates and the tree holds the normal form of the original input.

The essential advantage of this machine over the previous one is its local pointer capability: it can graft new nodes within the tree and can replace the children of a node by a subset of them in any order and possibly with repeti-

tions. (The last property bears similarity with top-down tree transducers, which traverse a tree from top to bottom, rewriting it.)

When the rewrite rules are linear (that is, no variable appears more than once in either L or R) each transition takes constant time, since applying a rewrite rule consists of checking a simple condition and performing a local change in the tree. Otherwise:

Theorem. *In the case of nonlinear rules the time is bounded by the number of machine transitions multiplied by a constant times the depth of the tree.*

When there are rewrite rules in which the same variable appears more than once in L, we must replace them by rules where this does not happen and instead check for equality. With our rewriting machine, it is not difficult to see that checking equality of subtrees can be done in time which is of the order of magnitude of the depth of the trees.

When the same variable appears more than once on the right hand side this means that the tree it represents has to be copied. One can use the rewriting machine to copy trees in time that is of the order of magnitude of their depth. Without loss of generality assume that the tree is binary with inner nodes of the form $t(Symbol, Left, Right)$ and its leaves are constant symbols. The normal form of $makecopy(T)$ is $copied(T', T'')$ where T' and T'' are copies of T. The rules are

$$makecopy(T) \longrightarrow top(double(T))$$
$$double(t(S, Left, Right)) \longrightarrow dt(S, S, double(Left), double(Right))$$
$$leaf(A) \mid double(A) \longrightarrow build(A, A)$$
$$dt(S', S'', build(X, X'), build(Y, Y')) \longrightarrow build(t(S', X, Y), t(S'', X', Y'))$$
$$top(build(T', T'')) \longrightarrow copied(T', T'')$$

(Notice that while S and A appear twice in the second and third rules they stand for function and constant symbols, so the rules can be thought of as schemas for linear rules.) Basically there is one movement down where things are doubled, and then one movement up where things are appropriately shuffled. The time needed to copy a term is proportional to its depth.

The class $\#P$ comprises those functions that can be computed by *counting Turing machines* of polynomial time complexity. Computing the permanent is an $\#P$-complete problem, yet the parallel rewriting machine can compute the permanent in polynomial time. Of course using our machine in this case is not realistic. However, if the depth of the space used is logarithmic, only a polynomial number of processors will be required. All the following examples are of this kind:

1. **Parallel Bits Algorithm:** adding two numbers in time logarithmic in their length.
2. **Parallel Merging and Sorting:** in time polylogarithmic in length of list.
3. **Evaluating Algebraic Expressions:** in average time $O(\sqrt{n})$ for size n.
4. **Scalar Product:** in time logarithmic in length of vectors.
5. **Matrix Multiplication:** matrices of size $n \times n$ in time $O(\log^2 n)$.
6. **Reachability:** for a graph of n vertices in time $O(\log^3 n)$.

In each, the machine, given a straightforward recursive description of the problem in the form of rewrite rules, works out an efficient implementation of the rewrite algorithm.

Parallel Machines with Interrupt. Once a node in the tree representing the memory of either the and-parallel machine or the rewriting machine has launched its children, it must wait for their answers—it must wait for both its children to become frontier nodes in a suitable state before it can become a frontier node again. There may, however, be cases when an answer from one child is enough (for example, when evaluating a Boolean expression of the form $E \vee F$, we know that if E evaluates to *true* so does the whole expression, irrespective of the value of F). It even may happen that the answer of one child is sufficient while the other child embarks on an infinite computation. We therefore consider more powerful machines, in which the frontier may move to the parent even if only one child is in a suitable state. This means that instead of having all processors work according to a uniform program, there may be cases in which one informed processor can cause other processors to terminate. Suppose that we are evaluating $E \vee F$ where E is small and F is large. In that case the more powerful machine may be faster, because if it discovers that E evaluates to *true* it can abandon evaluation of F. It is clear that an and-parallel machine evaluating a Boolean expression will take time proportional to the depth of the expression, because the frontier moves once from the root to the leaves of the input and once back to the root. If we consider all different Boolean expressions of the same length n to have the same probability, we can show that the and-parallel machine can evaluate them in $\Theta(\sqrt{n})$ time, and, for the interrupt machine, in asymptotically constant time.

Theorem. *The nondeterministic (and alternating) Turing machine can be simulated by an interrupt parallel machine. If a problem can be solved in time polynomial in the size of the input with the nondeterministic (alternating) Turing machine, the same is true for the interrupt parallel machine.*

Theorem. *Nondeterministic versions of our interrupt machines are not stronger than the deterministic ones.*

The machines described here are very powerful. However, if we restrict ourselves to polylogarithmic time and space of logarithmic depth, as in all our examples, they can be seen as models of realistic machines. It is also interesting to observe that the and-parallel machine gained much power by adding a pointer capability and an interrupt possibility—two things that can be easily implemented. Because of the tree structure of the memory and the way the computation proceeds it can never happen that two processors try to write to the same node (a problem that arises in the PRAM). Using rewrite rules makes the machine very easy to program—rules simply cannot be applied as long as their arguments have not reached the right form.