Processor Networks and Alternating Machines

Jonathan F. Buss
University of Waterloo*

Abstract

Internally monotone networks of processors are defined and discussed. Internally monotone networks are shown to be equivalent to alternating machines, in which the memory-access structure of the alternating machine corresponds to the topology of the network. Hypercubes, butterflies, and k-dimensional meshes occur as special cases. Algorithms written for internally monotone networks are well-suited for implementation on shared-memory machines.

1 Introduction

The study of parallel computation has resulted in a wide variety of models of parallel computers. Although the "realistic" models are all similar in their computational power, the differences are large enough to prevent true portability of programs among machines. It is shown here that two models, those of alternating machines and internally monotone networks of processors, are precisely equivalent in computational power. As a result, programs written for alternating machines can be efficiently executed on physical networks or on shared-memory machines.

Alternating machines use no direct communication. Instead, values that are required in several parts of the computation are computed separately every time they are needed. Nevertheless, the redundancy can be efficiently eliminated using the methods presented below, resulting in an implementation whose complexity is directly under the control of the programmer. The elimination of redundancy and the resultant conversion to a communication-based algorithm can be done automatically by a compiler. Thus alternation is a suitable paradigm for the design and implementation of efficient parallel algorithms.

One significant advantage of computations derived from alternating machines by the method presented here is that the synchronization requirements are very weak. Hardware implementations have no need for a global clock, and software emulations need not keep strict accounting of the time at which operations take place. For example, on a shared-memory machine, an emulation of an internally monotone network has about half of the memory requirements of an emulation of a synchronous network with the same topology.

Implementation of an algorithm under the proposed paradigm involves three steps.

1. Write a program for an alternating machine.
2. Compile the program onto an internally monotone network of processors.
3. Emulate the processor network.

It would be possible to directly implement the network of step 2. However, the number of processors in the network depends on the size of the input. The emulating machine for step 3 can be fixed, and only its execution time and memory requirements depend on the particular input.

The mapping from the alternating machine to the network is reversible—any computation on an internally monotone network can be expressed by an alternating program of the same complexity. The emulation of the network may involve slow-down in order to use a feasible number of processors, and thus

*This research was funded in part by a grant from the Natural Sciences and Engineering Research Council (NSERC) of Canada. Author's address: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. Electronic mail: jfbuss@uwaterloo.ca or jfbuss@waterloo.edu.

is not reversible in general. For example, the emulating machine may be a uniprocessor machine, and arbitrary serial computations cannot be sped up arbitrarily without loss of efficiency.

The remainder of this paper is organized as follows. Section 2 defines internally monotone networks and examines their basic properties. Section 3 defines general alternating machines. Section 4 explains the correspondence between the two models. Section 5 discusses the implementation of alternating programs on physical networks and on shared-memory machines.

2 Processor networks

A network comprises a set of processors linked by communication paths. A processor p with d neighbours is specified by a set S of possible states, together with a transition function $F^p\colon S^{d+1} \times \Sigma \mapsto S$ that determines the next state of p according to the current states of p and its neighbours and the external input presented to p. We will restrict consideration to the case when the state of a processor is representable by a vector of k bits; i.e., $S = \{0,1\}^k$. In this case, F^p can be decomposed into k bit-transition functions $F_i^p\colon S^{d+1} \times A \mapsto \{0,1\}$.

The input to the network is provided according to an input schedule In, that assigns to each processor a symbol of the input string. Normally, In is a simple function, specifying values of particular memory locations in a straightforward way. The output from the network is simply the final states of each processor.

The network is called *internally monotone* if each function F_i^p is monotone on states; i.e., if $F_i^p(\vec{s}, \mathrm{In}(p)) = 1$ implies $F_i^p(\vec{t}, \mathrm{In}(p)) = 1$ for all $\vec{t} \geq \vec{s}$. Because F_i^p is not restricted to be monotone on the input, non-monotone functions can be computed by an internally monotone network.

A familiar example of an internally monotone network is a combinational circuit comprising only and-gates and or-gates. Each gate is a processor, and F is either $\wedge$ or $\vee$. In this case, the restriction to internal monotonicity does not significantly affect the complexity of circuits comprising two-input gates (assuming negated inputs are available). One corollary of the results of the present paper is that arbitrary internally monotone networks, having perhaps large processors and cycles in their communication graph, are also nearly as powerful as their unrestricted counterparts.

A fundamental property of internally monotone networks is that no synchronization is required. Physical realizations of binary systems may be considered to have three states, named 0, 1, and #, where 0 and 1 denote the corresponding binary bits and # denotes a transitional, intermediate, oscillating or otherwise indeterminate state. The value # is for descriptive purposes only, and is not expected to be recognizable by a processor attempting to read the value. (A recognizable # is best interpreted as a 1.) For a state vector $\vec{s}$, denote by $\min(\vec{s})$ the result of substituting 0 for every occurrence of # in $\vec{s}$, and denote by $\max(\vec{s})$ the result of substituting 1 for every occurrence of # in $\vec{s}$. If $F_i^p(\max(\vec{s})) = 0$ or $F_i^p(\min(\vec{s})) = 1$, then the proper next state of bit i is determined, although the actual value may be #. Otherwise, the actual value may be 0, 1, or #.

The following conditions are sufficient for the proper execution of an internally monotone network.

1. The initial state of bit i of processor p is 0 if $F_i^p(0, 0, \ldots, 0, \mathrm{In}(p)) = 0$ and is # otherwise.
2. If $\vec{s}$ is the input vector to processor p and $F_i^p(\max(\vec{s}), \mathrm{In}(p)) = 0$, then state bit i of processor p has value 0 or #.
3. If $\vec{s}$ is the input vector to processor p and $F_i^p(\min(\vec{s}), \mathrm{In}(p)) = 1$, then state bit i of processor p has value 1 or #.
4. If an output value is determined by the corresponding inputs, and the actual value is equal to the determined value, then the actual value does not change.
5. If an output value remains determined, the actual value remains # for only a finite period of time.
6. There is a finite time period ϵ such that no change of value occurs within time ϵ of another change to the same value.

THEOREM 1. *If the above conditions hold, an internally monotone network will achieve and maintain its least fixed point in finite time.*

PROOF. Condition 6 implies that changes of state occur only at discrete times t_i, $i \geq 0$, with $t_0 = 0$. Denote bit i of processor p by (i, p).

LEMMA 1.1. *If a value becomes # or 1 at any time t_i, then the proper final value is 1.*

The proof of the lemma is by induction on i. The base case $i = 0$ follows immediately from condition 1.

Suppose the lemma holds up to time t_i, and bit i of processor p changes from 0 to # or 1 at time t_i. By condition 4, we must have $F_i^p(\max(\vec{s}), \mathrm{In}(p)) = 1$, where $\vec{s}$ is the inputs to p before time t_i. By induction, the proper final value of $\vec{s}$ is $\max(\vec{s})$. Hence the lemma holds up to time t_{i+1}.

Define a bit to be *indefinite* if it has value # for infinitely many time periods, or if it has value # at some time, but eventually remains 0. A bit *becomes indefinite* at time t_j if it is indefinite and first becomes # or 1 at t_j.

LEMMA 1.2. *No bit becomes indefinite.*

Suppose that some nonempty set of bits becomes indefinite. Let t_m be the first time at which some bit becomes indefinite and select some bit (i,p) that becomes indefinite at time t_m. We consider two cases.

Case I: $m = 0$. By condition 1 and the monotonicity of F_i^p, bit (i,p) is always determined. By conditions 5 and 3 it becomes 1 at some time, and condition 4 implies that the value 1 is permanent. Hence case I cannot occur.

Case II: $m > 0$. By condition 4, the value of (i,p) is not determined to be 0 just before t_m; i.e, $F_i^p(\max(\vec{s}), \mathrm{In}(p)) = 1$ before t_m. Because m is minimum, the eventual inputs $\vec{t}$ to (i,p) satisfy $\vec{t} \geq \max(\vec{s})$. The monotonicity of F_i^p then implies that (i,p) is determined to be 1 for an infinite time. Conditions 5 and 4 then imply that the actual value becomes 1 and stays 1. Hence case II cannot occur, and the lemma is proved.

Now consider the eventual values of the network. All are 0 or 1, by the second lemma. Conditions 2 and 3 imply that the values are a fixed point. The first lemma implies that the fixed point is least, and the theorem is proved. ∎

The computation time of an internally monotone network is defined in terms of the time for each bit to become set. Bit i of the state of processor p is *set by time 1* if $F_i^p(0, 0, \ldots, 0, \mathrm{In}(p)) = 1$, and bit i is *set by time* $r > 1$ if there are $b_1, b_2, \ldots, b_{dk}$ such that $F_i^p(\vec{b}, \mathrm{In}(p)) = 1$ and all b_j that are 1 are set by time $r-1$ in the appropriate processor. The network runs in time T if all bits that ever become set during the computation on input $x \in \Sigma^*$ are set by time $T(|x|)$.

If the time unit is an upper bound on the time for a state transition to occur, then $T(|x|)$ is an upper bound on the time for the network to reach its final state. If the time unit is the average time for a state transition, then $T(|x|)$ is an upper bound on the average time for the network to settle. If the network is synchronous, then $T(|x|)$ is the actual completion time.

3 Alternating machines

Alternation was defined by Chandra, Kozen and Stockmeyer [1] as a general model for parallel computation. Because of its close relationship to logic programming, alternation has been suggested as a good paradigm to write and reason about parallel algorithms [5, 3]. It is closely related to many other models of parallel computation [2]. The fundamental feature of an alternating machine is that existential and universal quantification are allowed as primitive operations. These operations may be considered for arbitrary machines.

A general alternating machine comprises an internal state machine (finite or infinite) and an external memory. Each state specifies an operation on the external memory. Each state transition is optionally labelled with a test operation on the external memory, and is active at a given time only if the result of the test is true. Each state s has a type $\tau(s)$ from the set $\{\wedge, \vee\} \cup \{\mathrm{Read}(a) \mid a \in \Sigma\}$. States of type $\mathrm{Read}(a)$ must have no successors.

A set of values for the external memory is termed an *external configuration.* An internal state and an external configuration together form a *configuration* of the machine. Two external configurations e_1 and e_2 are *adjacent* if there is a transition from one to the other; i.e., if there are internal states s_1 and s_2 such that either $(s_1, e_1) \in \delta(s_2, e_2)$ or $(s_2, e_2) \in \delta(s_1, e_1)$, where δ is the transition function on configurations. The *neighbourhood* of an external configuration e is the set of external configurations adjacent to e, except for e itself. The *external connectivity* of a machine M is the maximum size of any neighbourhood of an external configuration of M.

The input string is not given explicitly to the machine; it can be accessed only in states of type Read. Each external configuration is associated with one symbol of the input string by a function In (which may be undefined for some configurations). A configuration γ is defined to accept if and only if

- γ has type $\mathrm{Read}(a)$, and the kth symbol of the input is a, where $k = \mathrm{In}(\gamma)$, or

- γ has type $\wedge$ and all active successors of γ accept, or

- γ has type $\vee$ and some active successor of γ accepts.

This behaviour is formalized via the configuration graph of the machine.

The *configuration graph* Γ of an alternating machine M is a directed graph whose nodes are the configurations of M. The pair (γ_1, γ_2) is an edge of Γ if and only if $\gamma_2 \in \delta(\gamma_1)$. Nodes of Γ are labelled as follows:

- If $\tau(\gamma) = \mathrm{Read}(a)$, then γ is labelled Accept if $a = \mathrm{In}(\gamma)$, and Reject otherwise.

- If $\tau(\gamma) = \wedge$, then γ is labelled Accept if all successors of γ are labelled Accept, and is labelled Reject if some successor of γ is labelled Reject.

- If $\tau(\gamma) = \vee$, then γ is labelled Accept if some successor of γ is labelled Accept, and is labelled Reject if all successors of γ are labelled Reject.

If Γ has cycles or infinite branches, then some nodes may not be assigned a label by the above conditions; these nodes remain unlabelled. *M accepts from configuration* γ if and only if γ has label Accept in Γ. *M rejects from configuration* γ if and only if γ has label Reject in Γ.

It is possible, but not necessary, to add functions other than $\wedge$ and $\vee$ to the definition of an alternating machine. In particular, negation (interchanging Accept and Reject) may be simulated using only $\wedge$ and $\vee$ by introducing "positive" and "negative" versions of each state [1]. Any other function can be obtained using $\wedge$, $\vee$ and $\neg$.

An alternating machine M computes a function g if M can recognize each output symbol; i.e., M accepts from configuration $\mathrm{Out}(k, a)$ with x as the input string if and only if the kth symbol of $g(x)$ is a, where Out is a predetermined mapping from output symbols to configurations of M. Note that the computation order is reversed from the usual order—the computation starts with an output symbol and proceeds toward the inputs, which are accessible only at leaves of the computation tree. The simulation of M by a processor network will reverse the computation of M, resulting in the natural order of starting from inputs and ending with outputs.

The time and memory requirements of an alternating machine are taken to be the maximum over the usage by all $g(k, a)$.

The distinction between internal and external parts of the machine is, in many cases, purely arbitrary. For example, one special case is machines with no "external" memory, in which the entire machine is considered to be an internal infinite-state machine. At the other extreme, a small internal state machine (program) may control operations on a large external memory. A single machine may be described in several ways, and the various ways will give rise to different simulations.

One view of an alternating machine is that a multi-successor state results in the creation of multiple copies of the machine, one copy for each successor. Each copy continues a separate branch of the computation independently, and the forking machine suspends itself. When a branch of the computation accepts or rejects, the terminal machine reports this to its parent. When all children of a suspended machine terminate, the parent then can accept or reject and passes this information to its parent. If all branches terminate, the original machine eventually accepts or rejects.

A machine that runs as described above is very wasteful of processors, because the computation is highly redundant. If the machine runs in space s, then as many as $2^{2^{O(s)}}$ processors may be used, although only $2^{O(s)}$ are different. An efficient implementation must eliminate the redundancy.

4 The simulations

If the size of an alternating machine is taken to be the number of reachable external configurations, alternating machines are equivalent to internally monotone networks.

THEOREM 2. *Suppose that M computes f in time t, has m reachable external configurations, and has external connectivity d. Then f can be computed by an internally monotone network of m processors of degree d in time t.*

PROOF. Let M have with internal state set Q and external configuration set R. Let Γ be the reachable portion of the configuration graph of M. The simulating network N has one processor for each state in R, and r_1 and r_2 are adjacent in N if and only if there is a transition from r_1 to r_2 or from r_2 to r_1. The configuration graph Γ maps naturally onto M, with each edge of Γ being mapped to a link of N.

Without loss of generality, we may assume that $Q = \{1, 2, 3, \ldots, q\}$. The states of processors of N are vectors of length q. We design the transition functions F so that bit i of processor r will become set if and only if the configuration (i, r) is accepting in Γ. The transitions implied by F will mimic the parts of the definition of acceptance for a node in a configuration graph.

For $r \in R$ and $1 \leq e \leq d$, let $r(e)$ be the eth neighbour of r. Let Succ_i^r be the set of pairs (j, e) such that some transition from (i, r) leads to $\big(j, r(e)\big)$. Then

$$
\begin{aligned}
&F_i^r(x, x_1, \ldots, x_d, a) = 1 \iff \\
&\qquad \begin{cases}
\tau(i) = \mathrm{Read}(a), \text{ or} \\
\tau(i) = \wedge \text{ and } \forall (j, e) \in \mathrm{Succ}_i^r,\ x_{r(e)}(j) = 1, \\
\qquad \text{or} \\
\tau(i) = \vee \text{ and } \exists (j, e) \in \mathrm{Succ}_i^r,\ x_{r(e)}(j) = 1.
\end{cases}
\end{aligned}
$$

The functions F_i^r are internally monotone, and a straightforward induction shows that bit i of processor r is set by time t if and only if there is an accepting subtree for (i, r) in Γ of height at most t. Hence the network N has the desired properties. ∎

The simulation can be reversed in a similar fashion. We start with a weaker result.

THEOREM 3. *Suppose that a function g can be computed in time t by an internally monotone network of m processors of degree d. Then g can be computed in time $2t$ by an alternating machine with m reachable external configurations and external connectivity at most d.*

PROOF. The following recursive procedure accepts if and only if state bit i of processor p becomes set.

```
Procedure Set(p, i)
Begin
    ForAny(a, x, x_1, x_2, ..., x_d)
    AllOf
    { Read(a) }
    { If F_i^p(x, x_1, ..., x_d, a) = 1 then accept
                                     else reject
    }
    { ForAll( j : 1 ≤ j ≤ k, j ≠ i )
      Set(p, j)
    }
    { ForAll( q : 1 ≤ q ≤ d )
      ForAll( j : 1 ≤ j ≤ k and x_q(j) = 1 )
      Set(p(q), j)
    }
End
```

The construct ForAny is an existential assignment, whose successors are all states obtainable by giving a value to the listed variables, and therefore accepts if some set of possible values lead to acceptance. The construct ForAll is a universal assignment, which accepts if all possible values lead to acceptance. AllOf is a general and-branch; it accepts if all of the following predicates accept.

The input and output functions of the alternating machine, In_A and Out_A, are defined from the corresponding functions In_N and Out_N of the network. If $I(p)$ is the configuration of A that implements the Read for processor p, then define $\mathrm{In}_A\big(I(p)\big) = \mathrm{In}_N(p)$. If $O(p)$ is the configuration of A that implements the call to $\mathrm{Set}(p,i)$, then $\mathrm{Out}_A\big(\mathrm{Out}_N(p,i),1\big) = O(p)$ and $\mathrm{Out}_A\big(\mathrm{Out}_N(p,i),0\big) = \neg O(p)$. (The negation must be pushed to the leaves of the computation as defined earlier.)

The complexity of the algorithm depends on the separation into internal states and external memory. To obtain the desired simulation, we use the value of p as the external configuration, and store the values of i, a, x and $x_1, \ldots, x_d$ in the internal state. The number of external configurations is then m, the number of processors. The external connectivity is d, because only processors adjacent to p are examined in a call $\mathrm{Set}(p,i)$.

Each call to Set requires two steps prior to the recursion; one step suffices for the existential branching and one for the universal branching. Hence a straightforward induction shows that bit i of processor p is set by time r if and only if $\mathrm{Set}(p,i)$ accepts via a tree of height $2r$. The result follows. ∎

The simulation of Theorem 3 produces a slowdown of a factor of two. More significantly, the number of states of the alternating machine is exponential in the number of state bits per processor of the network. Both losses arise because alternating machines are limited to $\wedge$ and $\vee$, rather than arbitrary monotone functions. A simple modification to the simulations eliminates the losses.

THEOREM 4. *The previous simulations can be implemented as inverses of one another.*

PROOF SKETCH. Consider a machine A derived from a network N using Theorem 3. The predicate $\mathrm{Set}(p,i)$ is a monotonic function of the values $\mathrm{Set}\big(p(q),j\big)$, for $1 \le q \le d$ and $1 \le j \le k$, and of $\mathrm{Read}(a)$ for $a \in \Sigma$. This function is the transition function F_i^p of the original network. Furthermore, the configurations implementing Set are exactly the existential configurations of A. Hence if the construction of Theorem 2 is applied to only the existential configurations of A, the original network N is recovered. ∎

One open question is whether general synchronous networks are more powerful than their internally monotone counterparts. The difference cannot be arbitrarily large. The following theorem generalizes a result of Ruzzo [4] on alternating Turing machines.

THEOREM 5. *A synchronous network of p processors using time t can be simulated by an alternating machine with pt reachable external configurations in time t.*

PROOF SKETCH. In a synchronous network (not necessarily monotone), the state of each processor is a function $S_p(t)$ of time. The transition functions $F^p\colon S^{d+1} \times \Sigma \mapsto S$ determine the values by the condition

$$S_p(t+1) = F^p\big(S_p(t), S_{p(1)}(t), \ldots, S_{p(d)}(t), a\big).$$

for all p and t. The corresponding modification to the simulating predicate Set adds a time parameter.

Procedure Set(p, i, t)
Begin
If $t = 0$ then accept iff $i = 0$
Else
    ForAny$(a, x, x_1, x_2, \ldots, x_d)$
    AllOf
    { Read(a) }
    { If $F_i^p(x, x_1, \ldots, x_d, a) = 1$ then accept
                                else reject
    }
    { ForAll$(\, j : 1 \leq j \leq k,\ j \neq i\,)$
      Set$(p, j, t-1)$
    }
    { ForAll$(\, q : 1 \leq q \leq d\,)$
      ForAll$\big(\, j : 1 \leq j \leq k$ and $x_q(j) = 1\,\big)$
      Set$\big(p(q), j, t-1\big)$
    }
End

Putting the value of t into the external memory gives the desired simulation. ∎

Certain special cases of the above simulations are of particular interest.

- Hypercube networks, cube-connected lines and butterfly networks correspond to alternating one-tape Turing machines with binary tape alphabet. Hypercubes arise if the position of the tape head is considered to be part of the internal state. Cube-connected lines arise if the Turing machine only changes a symbol on the tape when the head is stationary. Butterflies arise if the Turing machine only changes a symbol when the head is moving. The dimension of the network is equal to the space used by the Turing machine.
- k-dimensional meshes correspond to alternating k-counter machines. The width in each dimension is the maximum value of the corresponding counter.
- The networks corresponding to alternating random-access machines are a kind of "cube-connected complete graphs." Such a network can be obtained from a hypercube by replacing each edge by a complete graph on r nodes and connecting all pairs of nodes corresponding to each original edge.

Other networks correspond to non-standard machines. One view of the alternating machine corresponding to a given network is as an automaton performing a walk on the nodes of the network. At each step, the automaton may change state, read the input symbol at the current node, replicate existentially or universally, or traverse an edge of the network.

5 Implementation of alternating programs

Any network topology is representable by a corresponding alternating machine. Thus the designer of parallel programs may eschew communication in favor of quantification. The fundamental measure of complexity for an alternating machine is the memory size. Memory usage is easier to measure and control than communication across a network; therefore, the programmer can more readily determine, and design for, program efficiency.

The restriction to internally monotone networks, necessary for a precise correspondence with alternation, has some significant advantages to offset its limitations. In practice, computational networks have fixed size, independent of the particular problem and input. Thus a program whose natural instantiation is on a varying network must be mapped in a standard way onto the physical machine. Internally monotone networks can be emulated without regard to a time variable.

The following simple simulation method suffices for internally monotone networks. Assign to each simulating processor one processor of the original network. One unit of work involves reading the input values for one state bit, determining the next state, and creating new units of work for each changed bit. The simulation is complete when all units of work are completed. The complexity of the simulation will depend on the method of queuing up units of work for processing.

The proposed method is substantially more efficient for simulation of internally monotone networks than it is for simulation of synchronous networks. It is faster, because bits that do not change at a given time need not be examined, and because processors need not wait to synchronize with their neighbours. It also uses about half of the memory space, because there is no need to distinguish "old" and "new" values for each simulated state. A system is currently under development for the implementation of alternating programs on a ten-processor Sequent Symmetry.

References

[1] A. K. Chandra, D. C. Kozen and L. J. Stockmeyer, "Alternation," *J. Assoc. Comput. Mach.* 28 (1981) 114–133.

[2] S. A. Cook, "Towards a Complexity Theory of Synchronous Parallel Computation," in *Logic and Algorithmic,* L'Enseignement Mathématique 30, 1982, pp. 75–100.

[3] D. Harel, "And/Or Programs: a New Approach to Structured Programming," *ACM Trans. Program. Lang. Syst.* 2 (1980) 1–17.

[4] W. L. Ruzzo, "Simple Universal Parallel Computers Based on Hypercube Interconnections," Technical Report 86-09-11, Computer Science Dept., U. of Washington, 1986.

[5] E. Y. Shapiro, "Alternation and the Computational Complexity of Logic Programs," *J. Logic Programming* 1 (1984) 19–33.