On Parallel Evaluation of Game Trees

Richard M. Karp & Yanjun Zhang[1]

TR-89-025

May, 1989

Abstract

We present parallel algorithms for evaluating game trees. These algorithms parallelize the "left-to-right" sequential algorithm for evaluating AND/OR trees and the α-β pruning procedure for evaluating MIN/MAX trees. We show that, on every instance of a uniform tree, these parallel algorithms achieve a linear speed-up over their corresponding sequential algorithms, if the number of processors used is close to the height of the input tree. These are the first non-trivial deterministic speed-up bounds known for the "left-to-right" algorithm and the α-β pruning procedure.

1 Both authors are from the Computer Science Division of the University of California, Berkeley. This work was sponsored by NSF Grant DCR-8411954 and by the International Computer Science Institute, Berkeley, California.

On Parallel Evaluation of Game Trees[*]

Richard M. Karp Yanjun Zhang

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

We present parallel algorithms for evaluating game trees. These algorithms parallelize the "left-to-right" sequential algorithm for evaluating AND/OR trees and the α-β pruning procedure for evaluating MIN/MAX trees. We show that, on every instance of a uniform tree, these parallel algorithms achieve a linear speed-up over their corresponding sequential algorithms, if the number of processors used is close to the height of the input tree. These are the first non-trivial deterministic speed-up bounds known for the "left-to-right" algorithm and the α-β pruning procedure.

1 Introduction

A *game tree* is a finite rooted ordered tree in which each leaf has a real value, the root is a MAX-node, the internal nodes at odd distance from the root are MIN-nodes and the internal nodes at even distance from the root are MAX-nodes. A *Boolean* game tree is a game tree in which the value on each leaf is 1 or 0, and is called an AND/OR tree. The *value* of a MAX-node (MIN-node) is recursively defined as the maximum (minimum) of the values of its children. The value of node v is denoted by val(v). The *value of a game tree* is the value of its root. The *evaluation problem* is to determine the value of a game tree from the given values on the leaves.

Game trees traditionally occur in the game-playing applications of AI such as chess, and game tree evaluation is a central problem in AI. The evaluation problem for AND/OR trees is closely related to the problem of efficiently executing theorem-proving algorithms for the propositional calculus based on backward-chaining deduction.

[*]This work was supported by NSF Grant DCR-8411954 and by the International Computer Science Institute, Berkeley, California.

The best known heuristic in practice for evaluating game trees is the α-β pruning procedure [5]. In the case of AND/OR trees, a similar but simpler "left-to-right" algorithm can be used instead. Both algorithms are effective sequential search methods. Within certain models of computation it has been shown [7, 8, 9, 10] that these two algorithms are optimal among all the sequential game-tree evaluation algorithms for evaluating uniform MIN/MAX trees and AND/OR trees, respectively.

There is a great deal of interest in the prospect of speeding up game-playing programs and theorem-proving programs through parallel computations, and consequently there has been a considerable research effort on parallel game-tree search over the past decade. Early studies focused on identifying different methods of conducting the α-β pruning procedure in parallel [2, 4]. More recent work has focused on experimental studies of parallel game-tree search methods [6, 11, 3]. Apart from a recent paper [1], which is discussed in Section 6, there has been little theoretical study in this area.

This paper is a theoretical study of parallel game-tree evaluation algorithms. We shall base our study on the *leaf-evaluation model* in which the unit of computational work is the evaluation of a leaf, all other computation being considered free. The basic step of an algorithm in this model is to evaluate a set of leaves of the input tree simultaneously; and the algorithm decides its next step from the values observed at previous steps. The running time of an algorithm is the number of basic steps it requires to determine the value of the root. The number of processors used in an algorithm is the maximum number of leaves evaluated at one step during the execution of the algorithm. An algorithm is *parallel* if it uses more than one processor; otherwise, it is *sequential*. Our primary interest in a parallel algorithm is its speed-up factor over the sequential ones as a function of the number of processors used.

The leaf-evaluation model, however, fails to reflect the reality that the game tree occurring in an application is usually generated by the algorithm that evaluates it. To capture the process of generating the input tree, we also consider the *node-expansion model* in which the algorithm is given only the root of the input tree, and it generates the other nodes of the tree by using an operation called *node expansion*. When this operation is applied to a node v it either evaluates v if v is a leaf or else produces the children of v. The unit of computational work in this model is a node expansion, all other computation being considered free. The basic step of an algorithm in this model is to expand a set of nodes simultaneously in one step. The running time of an algorithm is the number of steps at which it performs node expansions. The number of processors used by an algorithm is the maximum number of nodes expanded at one step of the algorithm.

In this paper we present simple parallel algorithms that parallelize the "left-to-right" algorithm for evaluating AND/OR trees and the α-β pruning procedure for evaluating MIN/MAX trees. Our main result is that, on *every* instance of a uniform tree, these parallel algorithms achieve a linear speed-up over their corresponding sequential algorithms, if the number of processors used is close to the height of the input tree.

Throughout this paper, $B(d,n)$ denotes the set of uniform d-ary AND/OR trees of height n and $M(d,n)$ the set of uniform d-ary MIN/MAX trees of height n.

2 AND/OR Trees

In this section we study the problem of evaluating AND/OR trees in parallel. For convenience, we present an AND/OR tree as a NOR-tree by replacing each AND-node or OR-node by a NOR-node. The value of a NOR-node is 0 if any of its children is 1; otherwise, it is 1. An AND/OR tree is equivalent to its NOR-tree representation up to complementation of the value of the root and possibly the values on the leaves.

The *total work* of an algorithm is the number of leaves evaluated. For a NOR-tree T, a *proof tree* of T is a smallest tree contained in T that verifies the value of T. Any evaluation of T must be able to exhibit a proof tree of T in which each leaf has been evaluated. For $T \in B(d,n)$, a proof tree of T has degree 1 and d on alternate levels. The number of leaves in a proof tree of T is at least $d^{\lfloor n/2 \rfloor}$. So we have the following elementary fact, which gives an inherent lower bound on the total work of any algorithm which evaluates any instance of $B(d,n)$.

Fact 1 *For any $T \in B(d,n)$, the total work of any algorithm to evaluate T is at least $d^{\lfloor n/2 \rfloor}$.*

Let T be any rooted tree with root r. Let v be a node in T. An *ancestor* of v is a node on the path from r to v. So v is an ancestor of itself. The value of v is *determined* if val(v) can be computed from the values of the leaves that have been evaluated. We say v is *dead* if the value of some ancestor of v is determined; otherwise, v is *live*. A simple sequential algorithm for evaluating NOR-trees is the "left-to-right" algorithm, called *Sequential SOLVE*, which evaluates the leaves from left to right while skipping over dead leaves.

Sequential SOLVE

At each step, evaluate the leftmost live leaf.

The following program S-SOLVE describes Sequential SOLVE recursively. Let v be the root of the subtree to be evaluated.

```
S-SOLVE ($v$: node): boolean;
if $v$ is a leaf then
  evaluate $v$;
  return(val($v$));
else
  let $u_1, u_2, \ldots, u_d$ be the children of $v$;
```

```
for $i = 1$ to $d$ do
  $b \leftarrow$ S-SOLVE($u_i$);
  if $b = 1$ then
    return (0);
return (1);
```

Our goal is to parallelize Sequential SOLVE. The following algorithm, called *Team SOLVE*, parallelizes Sequential SOLVE in the most direct way and achieves a square-root speed-up.

Team SOLVE with p processors

At each step, evaluate the leftmost p live leaves.

Proposition 1 *Let d be fixed and p be such that $0 \leq p \leq d^n$. Then, on any instance of $B(d,n)$, Team SOLVE with p processors has a speed-up of $\Omega(\sqrt{p})$ over Sequential SOLVE.*

Proof: It suffices to prove the result for $p = d^k$ where $0 \leq k \leq n$. We think of a subtree of height k as a *super-leaf* and the d^k processors used in Team SOLVE as a *team* which evaluates one super-leaf at each step. Sequential SOLVE and Team SOLVE evaluate the same set of super-leaves. By Fact 1, Sequential SOLVE takes at least $d^{\lfloor k/2 \rfloor}$ steps to evaluate each super-leaf. The proposition follows. □

On the other hand, for any n, p and fixed d, it is easy to construct a tree in $B(d,n)$ on which Team SOLVE with p processors achieves a speed-up of at most $O(\sqrt{p})$ over Sequential SOLVE.

We present a new parallel algorithm, called *Parallel SOLVE*, that achieves a linear speed-up over Sequential SOLVE on uniform trees, using a moderate number of processors. A central notion to the design of Parallel SOLVE algorithms is the "pruning number" of a live leaf. Node u is a *sibling* of v if u and v have the same parent; u is a *left-sibling* (*right-sibling*) of v if u and v have the same parent x, and u precedes (follows) v in the ordering of the children of x. The *pruning number* of a live leaf v is the total number of live left-siblings of the ancestors of v. The significance of the pruning number is that a live leaf with small pruning number is "likely" to be evaluated by Sequential SOLVE. In particular, a live leaf with pruning number 0 is the leftmost live leaf, which is the one evaluated by Sequential SOLVE.

The strategy of Parallel SOLVE is to evaluate live leaves with small pruning numbers. Parallel SOLVE has a parameter *width* to control its parallelism.

Parallel SOLVE of width w

At each step, evaluate all live leaves with pruning number at most w.

In particular, Parallel SOLVE of width 0 is identical with Sequential SOLVE.

When viewed top-down, Parallel SOLVE can be seen as a set of "left-to-right" sequential algorithms running in parallel, coordinated in a cascading structure. To illustrate the top-down view, the following program P-SOLVE describes Parallel SOLVE of width 1 on a binary NOR-tree. Let v be the root of the subtree to be evaluated and w the leftmost live leaf in the subtree. P-SOLVE has v and w as its parameters. Let $T(v)$ be the subtree rooted at v.

```
P-SOLVE($v$, $w$: node): boolean;
if $v$ is a leaf then
  evaluate $v$;
  return (val($v$));
else
  $u_1 \leftarrow$ left child of $v$;
  $u_2 \leftarrow$ right child of $v$;
  if $w$ is a leaf in $T(u_2)$ then
    $r \leftarrow$P-SOLVE($u_2$, $w$);
    return $(1 - r)$;
  else
    in parallel do
      $l \leftarrow$ P-SOLVE($u_1$, $w$);     /*parallel on left subtree*/
      $r \leftarrow$ S-SOLVE($u_2$);     /*sequential on right subtree*/
    if P-SOLVE($u_1$, $w$) returns first then
      if $l = 1$ then
        abort S-SOLVE($u_2$);
        return (0);
      else
        $u \leftarrow$ leftmost live leaf in $T(u_2)$;
        abort S-SOLVE($u_2$);
        $r \leftarrow$ P-SOLVE($u_2$, $w$);
        return $(1 - r)$;     /*finish evaluating right subtree*/
    if S-SOLVE($u_2$) returns first then
      if $r = 1$ then
        abort P-SOLVE($u_1$, $w$);
        return (0);
      else
        wait until P-SOLVE($u_1$, $w$) returns;
        return $(1 - l)$;
    if P-SOLVE($u_1$, $w$) and S-SOLVE($u_2$)
      return simultaneously then
      return (nor($l$,$r$)).     /* "nor" is the NOR-function*/
```

We analyze the effectiveness of Parallel SOLVE of width 1. We show that Parallel

SOLVE of width 1 has a linear speed-up over Sequential SOLVE on every instance of a uniform NOR-tree.

Theorem 1 [Main Theorem] *For a NOR-tree T, let $S(T)$ be the number of leaves evaluated by Sequential SOLVE to evaluate T and $P(T)$ the number of steps that Parallel SOLVE of width 1 takes to evaluate T. Then, for any $d \geq 2$, there is an $n_0 > 0$, depending on d, such that for any $T \in B(d,n)$ with $n > n_0$,*

$$\frac{S(T)}{P(T)} \geq c\,(n+1),$$

where $c > 0$ is an absolute constant and $n+1$ is the number of processors used by Parallel SOLVE of width 1 on T.

Corollary 1 *For $T \in B(d,n)$, let $W(T)$ denote the total work of Parallel SOLVE of width 1 on T. Then there is an n_0, depending on d, such that for $n > n_0$,*

$$W(T) \leq c'\,S(T),$$

where $c' > 0$ is an absolute constant.

3 Proof of Main Theorem

For a NOR-tree T, let $L(T)$ be the set of leaves that are evaluated during the execution of Sequential SOLVE on T. Thus $S(T) = |L(T)|$. Let H_T denote the NOR-tree obtained from T by deleting the nodes that are not ancestors of leaves in $L(T)$. We call H_T the *skeleton* of T. Note that for a node v in H_T, v has the same set of left siblings in T and in H_T.

The running time of Sequential SOLVE on T is the same as on H_T. A fundamental relation between T and its skeleton H_T is that the running time of Parallel SOLVE on H_T is at least as large as the running time of Parallel SOLVE on T. This is because the evaluations occurring in some subtrees of T that are not present in H_T may accelerate the evaluation of T.

Proposition 2 *Let $P_w(T)$ denote the number of steps Parallel SOLVE of width w takes to evaluate a NOR-tree T. Then, for any width w and any NOR-tree T, $P_w(T)) \leq P_w(H_T)$.*

Proof: We run Parallel SOLVE of width w on both T and H_T side by side. This process has the following invariant property which implies the proposition. Note that any node of H_T is also a node of T.

Property $\mathcal{A}$
At any time, if $v \in H_T$ is dead in H_T, then v is dead in T.

We prove Property $\mathcal{A}$ by induction. Property $\mathcal{A}$ trivially holds initially. Assume inductively that Property $\mathcal{A}$ holds up to step t. We show that Property $\mathcal{A}$ holds after step t. Consider $v \in H_T$ such that (i) v is live in both H_T and T before step t and (ii) v is dead in H_T after step t. We want to show that v will also be dead in T after step t. Without loss of generality, we may assume that the value of v in H_T is determined at step t. We use induction on the height of v.

Basis: v is a leaf. Since the value of v is determined in H_T at step t, v must be evaluated in H_T at step t. Thus, at step t, the pruning number of v in H_T is at most w. Suppose that a left-sibling u of some ancestor of v is dead in H_T at step t. By the assumption that Property $\mathcal{A}$ holds up to step t, u must also be dead in T at step t. Hence, at step t, the pruning number of v in T is no larger than that of v in H_T. Hence, v will also be evaluated in T at step t and, therefore, will be dead in T after step t.

Inductive Step: Assume inductively that Property $\mathcal{A}$ holds for any node of height at most $h-1$ after step t. Let v be of height h. Let D be the set of children of v in H_T whose values are determined in H_T at step t. Since the value of v is determined in H_T at step t, we must have (a) $D \neq \emptyset$ and (b) the value of v can be determined by the values of its children in D. Since each $u \in D$ is dead in H_T after step t and is of height $h-1$, by our inductive assumption, u must be dead in T after step t. Thus, the value of some ancestor v' of u must be determined in T after step t. If v' is also an ancestor of v, then v is dead in T after step t; otherwise, we have $v' = u$. Suppose that the latter case holds for each $u \in D$. Then, for each $u \in D$, the value of u is determined in T after step t. By (b), the value of v in T must also be determined, and therefore v must be dead after step t.

The induction step is complete. □

By Proposition 2, Theorem 1 will be proved if we can show that Parallel SOLVE of width 1 has a linear speed-up over Sequential SOLVE on the skeleton of any tree in $B(d,n)$. The advantage of focusing on H_T instead of T is that the total work of Parallel SOLVE of width 1 on H_T is at most the total work of Sequential SOLVE on H_T. Therefore, the effective speed-up follows if we can show that when Parallel SOLVE of width 1 executes on H_T, it evaluates a large number of leaves for a large portion of its running time. The rest of section is devoted to showing this.

The *parallel degree* of a step is the number of leaves evaluated at that step. A step of small parallel degree is considered as "bad". We want to bound the number of bad steps of Parallel SOLVE of width 1. The following proposition gives such bounds. Let $t_k(T)$ denote the number of steps of parallel degree k during the execution of Parallel SOLVE of width 1 on a NOR-tree T.

Proposition 3 *For any $T \in B(d,n)$,*

$$t_{k+1}(H_T) \leq \binom{n}{k}(d-1)^k,$$

where $k = 0, 1, \dots, n$.

Proof: Let w_t denote the leftmost live leaf of H_T at step t. For each step t of Parallel SOLVE of width 1 on H_T, the *base path at step* t, denoted by P_t, is the root-leaf path in H_T ending at w_t. Because w_t changes at each step, the base paths at different steps are distinct.

Consider base path $P_t = v_1, v_2, \dots, v_n$ at step t. The *code* of P_t, denoted by $C(t)$, is a vector $(c_1, c_2, \dots, c_n) \in \{0, 1, \dots, d-1\}^n$, where c_i is the number of live right-siblings of v_i prior to step t. We show that the codes of different base paths are distinct. Let $i_0 = \min\{i \mid \text{value of } v_i \text{ is known after step } t\}$. Since leaf $v_n = w_t$ is evaluated at step t, $i_0 \le n$. Let $C(t+1) = (c'_1, c'_2, \dots, c'_n)$. Then

$$c'_{i_0} < c_{i_0}. \tag{1}$$

For $i = 1, 2, \dots, i_0 - 1$, if the value of a right-sibling of v_i is not determined before step t but is determined after step t, then $c'_i < c_i$; otherwise, $c'_i = c_i$. So

$$c'_1 \le c_1, c'_2 \le c_2, \dots, c'_{i_0-1} \le c_{i_0-1}. \tag{2}$$

By (1) and (2), $C(t+1)$ precedes $C(t)$ in the lexicographic order of $\{0, 1, \dots, d-1\}^n$, which implies the distinctness of the codes.

The code $(c_1, c_2, \dots, c_n)$ of base path $P_t = v_1, v_2, \dots, v_n$ "encodes" the parallel degree of step t. Let $R = \{i \mid 1 \le i \le n, \text{ and } c_i > 0\}$. For $i \in R$, let T_i be the subtree whose root is the leftmost right-sibling of v_i that is live at step t. Then the leftmost live leaf in T_i is evaluated at step t. Also, w_t is evaluated at step t. Hence, the parallel degree of step t is $|R| + 1$. Let σ_k be the total number of vectors in $\{0, 1, \dots, d-1\}^n$ with exactly k non-zero components. Clearly, $\sigma_k = \binom{n}{k}(d-1)^k$. From the distinctness of the base paths and the distinctness of the codes of the base paths, we can conclude that $t_{k+1}(H_T) \le \sigma_k$. Hence, $t_{k+1}(H_T) \le \binom{n}{k}(d-1)^k$. $\Box$

By Proposition 3, the number of steps with small parallel degrees is limited. So are their contributions to the total work. The inherent lower bound on the total work would imply that much of the total work is contributed by the steps of large parallel degrees. This is shown by the following two lemmas.

Lemma 1 *For $d \ge 2$, let*

$$k_1 = \max\{k : \binom{n}{k} d^k \le d^{\lfloor n/2 \rfloor}\}. \tag{3}$$

Then there are absolute constants $b > 0$ and $\alpha > 0$ such that for for $n \ge b$,

$$k_1 > \alpha n.$$

Proof: Let $b > 0$ be any constant satisfying $(2be)^2 < 2^b$. Then, for $n \geq b$, we can derive

$$(be)^{\lceil n/b \rceil} \leq 2^{\lceil n/2 \rceil - \lceil n/b \rceil}. \tag{4}$$

Let $m = \lceil n/b \rceil$. So $b \geq n/m$. Then

$$\binom{n}{m} \leq \left(\frac{ne}{m}\right)^m \leq (be)^m \leq d^{\lfloor n/2 \rfloor - m}, \tag{5}$$

where the last inequality is by (4) and the condition $d \geq 2$.

By (3) and (5), we have $k_1 \geq m \geq \alpha n$ with $\alpha = b^{-1}$. □

Lemma 2 *For $d \geq 2$, let*

$$k_2 = \max\{k : \sum_{i=0}^{k} (i+1)\binom{n}{i}(d-1)^i \leq d^{\lfloor n/2 \rfloor}\}. \tag{6}$$

There is an n_0, depending on d, such that for $n > n_0$,

$$k_2 > \alpha n,$$

where α is the absolute constant in Lemma 1.

Proof: Without loss of generality, we assume $k_2 \leq \frac{n}{2}$. Let k be less than k_2. Then, for all $k' \leq k$, $\binom{n}{k'} \leq \binom{n}{k}$. Thus,

$$\sum_{i=0}^{k} (i+1)\binom{n}{i}(d-1)^i < (k+1)^2\binom{n}{k}(d-1)^k. \tag{7}$$

Let $x_0 = x_0(d) = \inf\{x | (x+1)^2(d-1)^x \leq d^x\} = \inf\{x | \frac{1}{x}\log(x+1) \leq \frac{1}{2}\log(\frac{d}{d-1})\}$. Since $f(x) = \frac{1}{x}\log(x+1)$ is decreasing in $x > 0$,

$$(k+1)^2(d-1)^k \leq d^k, \quad \text{if } k \geq x_0. \tag{8}$$

Let $n_0 = \max\{\alpha^{-1}x_0, b\}$, where α and b are the constants in Lemma 1. Then, by Lemma 1,

$$k_1 \geq \alpha n \geq x_0, \quad \text{if } n \geq n_0. \tag{9}$$

By (7), (8) and (3), we have

$$\sum_{i=0}^{k_1} (i+1)\binom{n}{i}(d-1)^i \leq \binom{n}{k_1} d^{k_1} \leq d^{\lfloor n/2 \rfloor}. \tag{10}$$

Hence, by (6), (10) and (9),

$$k_2 \geq k_1 \geq \alpha n.$$

□

Proposition 4 *For any $d \geq 2$, there is an n_0, depending on d, such that for any $T \in B(d,n)$ with $n > n_0$,*

$$P(H_T) \leq \frac{S(T)}{c(n+1)},$$

where $c > 0$ is an absolute constant.

Proof: The proof is a combination of Proposition 3 and Lemma 2. We have

$$P(H_T) = \sum_{i=1}^{n+1} t_i(H_T). \tag{11}$$

We maximize (11), subject to two constraints:

a) $t_{i+1}(H_T) \leq \binom{n}{i}(d-1)^i$;

b) $\sum_{i=1}^{n+1} t_i(H_T)\, i \ \leq S(T)$.

Constraint (a) is by Proposition 3 and constraint (b) is by the fact that the total work of Parallel SOLVE on H_T cannot exceed the total number of leaves of H_T, which is $S(T)$.

It is clear that, subject to (a) and (b), $P(H_T)$ is maximized when

i) $t_{i+1}(H_T) = \binom{n}{i}(d-1)^i$, for $i = 0, 1, \ldots, k_0$,

ii) $t_{k_0+2}(H_T) = \lfloor x \rfloor$ and

iii) $t_i(H_T) = 0$ for $i > k_0 + 2$,

where

$$k_0 = \max\{k : \sum_{i=0}^{k}(i+1)\binom{n}{i}(d-1)^i \leq S(T)\} \tag{12}$$

and x satisfies

$$\sum_{i=0}^{k_0}(i+1)\binom{n}{i}(d-1)^i + (k_0+2)x = S(T). \tag{13}$$

Hence, by (11) and (i)-(iii),

$$P(H_T) \leq \sum_{i=0}^{k_0}\binom{n}{i}(d-1)^i + \lceil x \rceil. \tag{14}$$

By Fact 1, $S(T) \geq d^{\lfloor n/2 \rfloor}$. Then, by Lemma 2 and (12), there is an n_0, depending on d, such that for $n > n_0$,

$$k_0 > \beta(n+1), \tag{15}$$

where $\beta > 0$ is an absolute constant. Let

$$a = \sum_{i=\lfloor k_0/2 \rfloor}^{k_0} \binom{n}{i} (d-1)^i + \lceil x \rceil. \tag{16}$$

Thus, by (14) and (16),

$$P(H_T) \leq 2a. \tag{17}$$

By (13),

$$S(T) > \sum_{i=\lfloor k_0/2 \rfloor}^{k_0} (i+1) \binom{n}{k} (d-1)^i + (k_0+2)\lceil x \rceil \geq \frac{k_0}{2} a. \tag{18}$$

Hence, by (17), (18) and (15), for $n \geq n_0$,

$$P(H_T) \leq 2a \leq \frac{4}{k_0} S(T) \leq \frac{S(T)}{c(n+1)},$$

where $c = 4/\beta$. □

Theorem 1 follows immediately from Propositions 2 and 4. □

The proof of Theorem 1 given in this section reveals that the absolute uniformity of the input tree is not required. The conditions that make the proof work are (i) the lower bound on the sequential time is large, exponential in the height of the input tree, and (ii) the upper bound on the number of possible steps of small parallel degrees is relatively small. These conditions holds for trees that are "close" to be uniform. The following corollary is just one example.

Corollary 2 *Let $0 < \alpha \leq 1$ and $0 < \beta \leq 1$. Let T be a NOR-tree such that the number of children of any non-leaf node in T is between αd and d and each root-leaf path in T has a length between βn and n. Then there is an n_0, depending on d, α and β, such that for $n > n_0$, the conclusion of Theorem 1 holds for T.*

4 MIN/MAX Trees

In this section we turn to the problem of evaluating MIN/MAX trees. We describe a parallel algorithm, called *Parallel α-β*, which parallelizes the α-β pruning procedure for evaluating MIN/MAX trees.

We shall describe a general method for evaluating MIN/MAX trees, which includes the sequential α-β pruning procedure and Parallel α-β as special cases. This method is a *pruning* process which evaluates the input tree while pruning away certain nodes whose values cannot affect the value of the root.

Let T be the input MIN/MAX tree with root r. At a general step of the pruning process, we have a *pruned* tree, denoted by $\tilde{T}$, which is a tree obtained from T by deleting some subtrees of T. Let $\mathrm{val}_{\tilde{T}}(v)$ denote the value of node v in $\tilde{T}$. This pruning process maintains the invariant property that $\mathrm{val}_{\tilde{T}}(r) = \mathrm{val}_T(r)$. Initially, $\tilde{T} = T$ and no leaves are evaluated. At a general step, certain leaves of $\tilde{T}$ are evaluated. A node $v \in \tilde{T}$ is *finished* if every leaf in the subtree rooted at v in $\tilde{T}$ is evaluated; otherwise, v is *unfinished*. For each finished node v in $\tilde{T}$, the pruning process is able to compute $\mathrm{val}_{\tilde{T}}(v)$, the value of v in $\tilde{T}$.

A general step of the pruning process consists of a *leaf-evaluation step* in which one or more leaves are evaluated and a sequence of *pruning steps* in which certain nodes are pruned away and *propagation steps* in which the values of the newly finished nodes are computed. The value of the pruned tree is returned as the value of the input tree when the root is finished.

The pruning steps are governed by the *pruning rule*. The pruning rule is based on two bounds, the α-bound and the β-bound. The *α-bound* of v in $\tilde{T}$, denoted by $\alpha_{\tilde{T}}(v)$, and the *β-bound* of v in $\tilde{T}$, denoted by $\beta_{\tilde{T}}(v)$, are defined as follows:

$$\alpha_{\tilde{T}}(v) = \max\{-\infty, \max\{\mathrm{val}_{\tilde{T}}(u) \mid u \text{ is a finished sibling of a MIN-ancestor of } v\}\}.$$
$$\beta_{\tilde{T}}(v) = \min\{+\infty, \min\{\mathrm{val}_{\tilde{T}}(u) \mid u \text{ is a finished sibling of a MAX-ancestor of } v\}\}.$$

Notice that the α-bound never decreases and the β-bound never increases.

The pruning rule applies only to the unfinished nodes.

Pruning Rules
Delete an unfinished v from $\tilde{T}$ if $\alpha_{\tilde{T}}(v) \geq \beta_{\tilde{T}}(v)$.

The pruning rule allows a node to be deleted when it cannot influence the value of the root. This ensures that the root of the pruned tree remains unchanged. More precisely, we have the following theorem.

Theorem 2 *At any time, we have $val_{\tilde{T}}(r) = val_T(r)$. Hence, when root r is finished, the pruning process returns $val_T(r)$.*

Proof: (sketch) Suppose that v is deleted from $\tilde{T}$ by the pruning rule. Then $\alpha_{\tilde{T}}(v) \geq \beta_{\tilde{T}}(v)$. Let $FC(z)$ denote the set of finished children of z. Let u be a MAX-ancestor of v and w a MIN-ancestor of v such that

$$f_{\tilde{T}}(u) = \max\{\mathrm{val}_{\tilde{T}}(u') | u' \in FC(u)\} = \alpha_{\tilde{T}}(v),$$

$$g_{\tilde{T}}(w) = \min\{\mathrm{val}_{\tilde{T}}(w') | w' \in FC(w)\} = \beta_{\tilde{T}}(v).$$

Without loss of generality, we assume that u is an ancestor of w. Let $\tilde{T}'$ be the pruned tree after v is deleted. Notice that v is in the subtree of an unfinished child of w. Since the values of unfinished children of w can only decrease $g_{\tilde{T}}(w)$, we must have $\mathrm{val}_{\tilde{T}'}(w) \leq g_{\tilde{T}}(w)$. Thus, $f_{\tilde{T}}(u) \geq \mathrm{val}_{\tilde{T}'}(w)$. Since u is a MAX-ancestor of w, $\mathrm{val}_{\tilde{T}'}(w)$ cannot influence the value of u, i.e., $\mathrm{val}_{\tilde{T}'}(u) = \mathrm{val}_{\tilde{T}}(u)$. Consequently,

$\mathrm{val}_{\widetilde{T}'}(r) = \mathrm{val}_{\widetilde{T}}(r)$. The same conclusion holds when there is more than one node being deleted at the same step. The value of the root is maintained at each pruning step and the theorem follows. □

The sequential α-β pruning procedure, in the leaf-evaluation model, is as follows.

Sequential α-β
At each step, evaluate the leftmost unfinished leaf of the current pruned tree.

The *pruning number* of an unfinished leaf v is the total number of unfinished left-siblings of ancestors of v. The following parallel algorithm, *Parallel α-β*, parallelizes Sequential α-β. Like Parallel SOLVE, Parallel α-β has a width parameter to control its parallelism.

Parallel α-β of width w
At each step, evaluate all unfinished leaves of the current pruned tree whose pruning numbers are at most w.

In particular, Parallel α-β of width 0 is identical with Sequential α-β.

When viewed from top down, Sequential α-β can be seen as a depth-first search that traverses the input MIN/MAX tree from left to right while maintaining the α-bound and the β-bound for the currently visited node v. It may backtrack from v upon discovering that the children of v meet the condition of the pruning rule. Parallel α-β can be seen as a set of Sequential α-β algorithms running in parallel, each having its own α-bound and β-bound, coordinated in a cascading structure.

The correctness of Sequential α-β and Parallel α-β are guaranteed by Theorem 2. The following theorem is the counterpart of Theorem 1 for MIN/MAX trees.

Theorem 3 *For a MIN/MAX tree T, let $\widetilde{S}(T)$ be the number of leaves evaluated by Sequential α-β to evaluate T and $\widetilde{P}(T)$ the number of steps that Parallel α-β of width 1 takes to evaluate T. Then, for any $d \geq 2$, there is an n_0, depending on d, such that for any $T \in M(d, n)$ with $n > n_0$,*

$$\frac{\widetilde{S}(T)}{\widetilde{P}(T)} \geq c(n+1),$$

where $c > 0$ is an absolute constant and $n+1$ is the number of processors used by Parallel α-β of width 1 on T.

The proof of Theorem 3 follows the same line as that of Theorem 1. The only task is to extend Proposition 2 to MIN/MAX trees. We shall state this extension without proof in the next proposition. For a MIN/MAX tree T, let $\widetilde{L}(T)$ be the set of leaves evaluated during the execution of Sequential α-β on T. So $\widetilde{S}(T) = |\widetilde{L}(T)|$. Let $\widetilde{H}_T$ denote the MIN/MAX tree obtained from T by deleting the nodes that are not ancestors of leaves in $\widetilde{L}(T)$.

Proposition 5 *Let $\tilde{P}_w(T)$ denote the running time of Parallel α-β of width w on a MIN/MAX tree T. Then, for any width w and any MIN/MAX tree T, $\tilde{P}_w(T)) \leq \tilde{P}_w(\widetilde{H}_T)$.*

The intuitive reason that Proposition 5 holds is that the evaluations of the subtrees of T that are not present in $\widetilde{H}_T$ may provide additional information to "sharpen" the α-bounds and β-bounds for other nodes, thus speeding up the process of evaluating T as a whole.

The following fact is well-known (see [5]). It plays the same role for MIN/MAX trees as Fact 1 for AND/OR trees.

Fact 2 *For any MIN/MAX tree $T \in M(d,n)$, $d^{\lfloor n/2 \rfloor} + d^{\lceil n/2 \rceil} - 1$ is an inherent lower bound on the total work of any algorithm that evaluates T.*

Proof: Let r be the root of T. Let a and b be any two numbers such that $a <$ val$(r) < b$. An algorithm that evaluates T would be able to verify both statements "$a <$ val(r)" and "val$(r) < b$" by viewing. T as a Boolean tree. Since r is a MAX-node, a proof tree for verifying "$a <$ val(r)" has $d^{\lfloor n/2 \rfloor}$ leaves and a proof tree for verifying "val$(r) < b$" has $d^{\lceil n/2 \rceil}$ leaves and, moreover, these two proof trees have exactly one leaf in common. The lemma follows. □

The conclusion of Proposition 3 remains valid for MIN/MAX trees. Proposition 5, Fact 2 and Proposition 3 are the ingredients needed for the proof of Theorem 3.

5 Node-Expansion Model

We have developed parallel algorithms for evaluating game trees in the leaf-evaluation model. In this section we consider the counterparts of these algorithms in the node-expansion model introduced in Section 1.

The algorithms described previously in the leaf-evaluation model have their counterparts in the node-expansion model. As an illustration, we describe the node-expansion versions of both Sequential SOLVE and Parallel SOLVE, called N-Sequential SOLVE and N-Parallel SOLVE, respectively, and show that the counterpart of Theorem 1 holds in the node-expansion model. Consider a node-expansion algorithm on input tree T. Let T^* denote the tree consisting of the nodes of T that have been generated so far by the algorithm in consideration. Initially, T^* consists of only the root of T. A node $v \in T^*$ is *dead* if the value of any ancestor of v is determined in T^*; otherwise, v is *live*. A *frontier node* is a live node that is not expanded. The *pruning number* of a frontier node v is the total number of live left-siblings of ancestors of v.

N-Sequential SOLVE
At each step, expand the leftmost frontier node.

N-Parallel SOLVE of width w

At each step, expand all the frontier nodes with pruning number at most w.

In particular, N-Parallel SOLVE of width 0 is identical to N-Sequential SOLVE.

In Section 7, we shall discuss the implementation of N-Parallel SOLVE of width 1. As a preparation, we present the algorithm as a program. For convenience, we assume the input tree to be binary. The following program S-SOLVE* describes N-Sequential SOLVE. Let v be the root of the subtree to be evaluated.

```
S-SOLVE*($v$: node): boolean;
expand $v$;
if $v$ is a leaf then
  return(val($v$));
else
  $u_1 \leftarrow$ left-child of $v$;
  $u_2 \leftarrow$ right-child of $v$;
  $l \leftarrow$ S-SOLVE*($u_1$);
  if $l = 1$ then
    return (0);
  else
    $r \leftarrow$ S-SOLVE*($u_2$);
    return (1-r);
```

The following program P-SOLVE*(v, g) describes N-Parallel SOLVE of width 1 on a binary NOR-tree. P-SOLVE*(v, g) is similar to program P-SOLVE in Section 2. P-SOLVE*(v, g) has two parameters, v and g, where v is the root of the subtree to be evaluated and g is the base path in the subtree i.e., the path from v to the leftmost frontier node in the subtree. We assume that g carries with it the right-siblings of the nodes on g. Initially, g consists of only v.

```
P-SOLVE*($v$, $g$: node): boolean;
if $v$ is the only node on $g$ then
  expand $v$;
  if $v$ is a leaf then
    return (val($v$));
  else
    $u_1 \leftarrow$ left child of $v$;
    $u_2 \leftarrow$ right child of $v$;
    $g \leftarrow \{u_1\}$;
    $g$ records $u_2$ as the right-sibling of $u_1$;
else /* $v$ has a child on $g$ */
  $g \leftarrow g \setminus \{v\}$;
  $u \leftarrow$ the child of $v$ on $g$;
```

```
    if $u$ is the right child of $v$ then
      $r \leftarrow$ P-SOLVE*$(u, g)$;
      return$(1 - r)$;
    else /* $u$ is the left child of $v$ */
      $u_1 \leftarrow u$;
      $u_2 \leftarrow$ right child of $v$;
  in parallel do
    $l \leftarrow$ P-SOLVE*$(u_1, g)$;
    $r \leftarrow$ S-SOLVE*$(u_2)$;
  if P-SOLVE*$(u_1, g)$ returns first then
    if $l = 1$ then
      abort S-SOLVE*$(u_2)$;
      return (0);
    else
      $g \leftarrow$ base path in subtree rooted at $u_2$;
      abort S-SOLVE*$(u_2)$;
      $r \leftarrow$ P-SOLVE*$(u_2, g)$;
      return $(1 - r)$;
  if S-SOLVE*$(u_2)$ returns first then
    if $r = 1$ then
      abort P-SOLVE*$(u_1, g)$;
      return (0);
    else
      wait until P-SOLVE*$(u_1, g)$ returns;
      return $(1 - l)$;
  if P-SOLVE*$(u_1, g)$ and S-SOLVE*$(u_2)$
    return simultaneously then
    return (nor$(l,r)$).
```

The following theorem is the counterpart of Theorem 1 in the node-expansion model.

Theorem 4 *For a NOR-tree T, let $S^*(T)$ be the number of nodes expanded by N-Sequential SOLVE to evaluate T and $P^*(T)$ the number of steps that N-Parallel SOLVE of width 1 takes to evaluate T. Then, for any $d \geq 2$, there is n_0, depending on d, such that for any $T \in M(d,n)$ with $n > n_0$,*

$$\frac{S^*(T)}{P^*(T)} \geq c(n+1),$$

where $c > 0$ is an absolute constant and $n+1$ is the number of processors used by N-Parallel SOLVE of width 1 on T.

The proof of Theorem 4 goes as that of Theorem 1. The only part in the proof that needs to be changed is Proposition 3. The *parallel degree* of a step in the node-expansion model is the number of nodes expanded at that step. Let $t^*_k(T)$ denote the number of steps of parallel degree k during the execution of N-Parallel SOLVE of width 1 on T. The skeleton H_T defined in Section 3 consists of precisely those nodes of T that are expanded by N-Sequential SOLVE on T.

Proposition 6 *For any $T \in B(d,n)$,*

$$t^*_{k+1}(H_T) \leq (n-k)\binom{n}{k}(d-1)^k,$$

where $k = 0, 1, \ldots, n$.

Proof: At each step of N-Parallel SOLVE of width 1 on H_T, the *base path* is the path from r to the leftmost frontier node. By the same argument in Proposition 3, the number of base paths of length m with parallel degree $k+1$ is at most $\binom{m}{k}(d-1)^k$, where $m \geq k$. Hence, $t^*_{k+1}(H_T)$ is at most

$$\sum_{m=k}^{n} \binom{m}{k}(d-1)^k \leq (n-k)\binom{n}{k}(d-1)^k.$$

□

The bound in Proposition 6 is larger than the one in Proposition 3 by a factor of $O(n)$. It is easy to check that the asymptotics of the subsequent analysis is only affected up to a constant factor. Therefore, Theorem 4 holds.

Sequential α-β and Parallel α-β can also be converted into their node-expansion versions, so is Theorem 3. Given the space limitation, we shall not present this changes here.

6 Probabilistic Approaches

It is easy to construct instances of uniform AND/OR trees such that Sequential SOLVE would have to evaluate all the leaves. In fact, any *deterministic* algorithm that evaluates uniform AND/OR trees would have to evaluate all the leaves in the worst case. One can also construct such worst-case instances for the α-β pruning procedure. To avoid this worst-case behavior, researchers have taken up probabilistic approaches.

One approach is to make some probabilistic assumptions on input instances and study the expected number of leaves evaluated in a random input. In the i.i.d. model, the value on each leaf in an AND/OR tree is determined by an independent coin flip with a fixed bias p, $0 \leq p \leq 1$, and in a MIN/MAX tree the values of leaves are drawn independently from some common distribution. Under this model, it has been shown that Sequential SOLVE and Sequential α-β are asymptotically optimal for evaluating a random uniform AND/OR tree and a random uniform MIN/MAX tree, respectively [8, 10]. This suggests that these sequential procedures are good candidates for parallelization. A recent paper by Althöfer [1] gives a probabilistic analysis of a certain algorithm for evaluating uniform binary AND/OR trees in the i.i.d. model where the bias $p = \frac{\sqrt{5}-1}{2}$. He states that, when the number of processors is moderate, the expected speed-up over Sequential SOLVE is proportional to the number of processors. In contrast, our analysis is conducted deterministically. Consequently, our theorems hold in the i.i.d. model automatically.

Another approach that avoids imposing any assumptions on the input is through randomization. A *randomized* algorithm is allowed to flip coins, and the actions of the algorithm may depend on the outcomes of these flips. For our purpose, we restrict our discussion of randomized algorithms to the node-expansion model. The complexity of a sequential randomized algorithm is the expected number of nodes expanded on a worst input. There is a natural way to randomize N-Sequential SOLVE: expand the root; repeatedly choose an unexpanded child of the root at random and evaluate the child recursively until the value of the root can be determined. We call this algorithm *R-Sequential SOLVE.* As shown in [9], R-Sequential SOLVE is optimal among randomized sequential algorithms for evaluating uniform AND/OR trees.

Conceptually, R-Sequential SOLVE is like Sequential SOLVE acting on a randomly permuted input tree, i.e., a tree obtained from the input tree by randomly permuting the children of each node. We can extend the randomization to N-Parallel SOLVE to obtain randomized algorithm *R-Parallel SOLVE.* Conceptually, R-Parallel SOLVE is equivalent to the execution of N-Parallel SOLVE on a random permuted input tree. In practice, of course, the entire randomly permuted tree is not explicitly constructed; instead, randomizations are performed only to the extent necessary to determine the steps of the algorithm.

Theorem 5 *For a NOR-tree T, let $P_R^*(T)$ and $S_R^*(T)$ denote the random variables that are the number of steps that R-Parallel SOLVE of width 1 and R-Sequential SOLVE take to evaluate T, respectively. Let $E(P_R^*(T))$ and $E(S_R^*(T))$ denote the expectations of $P_R^*(T)$ and $S_R^*(T)$, respectively. Then, for any $d \geq 2$, there is an n_0, depending on d, such that for any $T \in B(d,n)$ with $n > n_0$,*

$$\frac{E(S_R^*(T))}{E(P_R^*(T))} \geq c(n+1).$$

where $c > 0$ is an absolute constant.

Proof: Follows from Theorem 4 by averaging. □

We can randomize N-Sequential α-β by letting the algorithm traverse the input tree in a random depth-first search. We call the resulting randomized algorithm *R-Sequential α-β*. Similarly, we can extend the randomization to N-Parallel α-β to obtain randomized algorithm *R-Parallel α-β*.

Theorem 6 *R-Parallel α-β of width 1 achieves a linear expected speed-up over R-Sequential α-β for uniform MIN/MAX trees.*

Remark: We do not know whether R-Sequential α-β is optimal among randomized sequential algorithms for evaluating uniform MIN/MAX trees, although a randomized version of a variant of N-Sequential α-β called SCOUT [7] was proved to possess this optimality [9].

7 Implementation

Although the node-expansion model is more realistic than the leaf-evaluation model, it omits important details as to how the invocations of procedures S-SOLVE* and P-SOLVE* are assigned to processors, how the results of such invocations are passed from one processor to another, and how pruning occurs. In the present section we describe an implementation of N-Parallel SOLVE of widht 1 on a message-passing multiprocessor system in which any processor can send a message in unit time to any other. Our implementation maintains the linear speed-up of N-Parallel SOLVE of width 1 over N-Sequential SOLVE on uniform NOR-trees. For convenience in exposition we restrict ourselves to the case that the input NOR-tree is binary in which each internal node has exactly two children.

In our implementation, the procedure S-SOLVE* will not be implemented recursively. Instead, the processor responsible for executing S-SOLVE*(v) simply executes a depth-first search of the subtree rooted at v, skipping over subtrees whose leaves are all dead. A pushdown stack is used to control the search. At each step the stack contains a description of the path g from v to the node currently being expanded. Along with each node in the path are stored the names of its two children, and an indication of whether its successor in the path is its left child or its right child.

Our implementation of procedure P-SOLVE* will differ from the description given in Section 5. In that description P-SOLVE* has two parameters: a node v and a path g from v to the leftmost frontier node of the search. In our implementation, only the parameter v will be passed; the processor executing the procedure will always have enough information available to determine g for itself. Additional procedures P-SOLVE** and P-SOLVE*** will be required. These procedures are variants of P-SOLVE* and, like P-SOLVE*, require a single parameter v, giving the root of the subtree to be searched. Procedure P-SOLVE**(v) is called instead of P-SOLVE* when it is known, at the time of invocation, that node v has already been expanded but the value of its left child has not been determined; procedure P-SOLVE***(v) is called when it is known that v has already been expanded and that the value of v's left child is 0. The circumstances under which these variants of P-SOLVE* come into play will be described later in this section.

Let $d(v)$, the *level* of node v, be defined as the distance of node v from the root. Our processor allocation method is extremely simple. Each level of the NOR-tree has a processor assigned to it. The processor assigned to level d is responsible for precisely those invocations of S-SOLVE*, P-SOLVE* , P-SOLVE** and P-SOLVE*** in which the root node v is at level d.

Because of pruning, the execution of a procedure may have to be aborted before its completion. For example, suppose that nodes w and x are siblings, and that both P-SOLVE*(w) and S-SOLVE*(x) are being executed. If one of these procedures returns a 1 then the execution of the other procedure should be aborted. If P-SOLVE*(w) returns a 0 then the execution of S-SOLVE*(x) should be aborted and, instead, an execution of P-SOLVE*(x) should be initiated, with a base path g equal to the path that was on the stack at the time S-SOLVE*(x) was aborted. The desired behavior can be achieved without explicit messages directing procedures to abort, provided that the following "pre-emption" rule is obeyed: processor d works only on the most recent invocation of S-SOLVE* whose root node is at level d and on the most recent invocation of P-SOLVE*, P-SOLVE** or P-SOLVE*** whose root node is at level d; moreover, it works on S-SOLVE(v) only if it has not been directed to execute P-SOLVE*(v); all other invocations automatically become terminated and the space allocated to these invocations is released. The only point at which one processor directs another to halt some invocation occurs at the end of the computation, when the value of the root is determined. At that point, a "halt" message is broadcast by processor 0 to all other processors.

We now describe the implementation in greater detail. A processor may send or receive messages of six types: S-SOLVE*(v), P-SOLVE*(v), P-SOLVE**(v), P-SOLVE***(v), val(v) = 1 and val(v) = 0. Processor d may receive a message of one of the first four types only if $d(v) = d$. A message of one of the last two types is always directed from processor $d(v)$ to processor $d(v) - 1$.

When processor $d(v)$ receives the message "S-SOLVE*(v)" it begins a nonrecursive execution of the left-to-right sequential NOR-tree evaluation algorithm on the subtree rooted at v. The execution continues until one of the following events occurs: i) the execution terminates and the value of v is reported to processor $d(v) - 1$; ii) processor $d(v)$ receives a message of the form "S-SOLVE*(w)," where $d(w) = d(v)$ and $w \neq v$. In this case processor $d(v)$ terminates the execution of S-SOLVE*(v), as val(v) is no longer relevant; iii) processor $d(v)$ receives a message of the form "P-SOLVE*(v)." In this case it terminates the execution of S-SOLVE*(v) and begins executing P-SOLVE*(v), as described below.

When processor $d(v)$ receives the message "P-SOLVE*(v)" its behavior depends on whether an execution of S-SOLVE*(v) is in progress. This gives two cases. The first case is that no execution of S-SOLVE*(v) is in progress. In this case, v is not expanded and processor $d(v)$ does the following: it expands v; if v is a leaf then it evaluates v, sends the value to processor $d(v) - 1$ and halts; otherwise, it obtains a left child w of v and a right child x of v. It then sends the messages "P-SOLVE*(w)" and "S-SOLVE*(x)" to processor $d(v) + 1$, and waits for messages giving the values of w and x. If it learns that one of these values is 1 then it sends the message "val(v) = 0" to processor $d(v)-1$ and halts; if the first message it receives is "val(w) $= 0$" then it sends the message "P-SOLVE*(x)" to processor $d(v)+1$. As soon as it has received the two messages "val(w) $= 0$" and "val(x) $= 0$" it sends the message "val(v) $= 1$" to processor $d(v)-1$ and halts.

The second case is more complicated. In this case, processor $d(v)$ receives the message "P-SOLVE*(v)" when it has already received the message "S-SOLVE*(v)." This is the case in which it must switch from executing the sequential left-to-right evaluation algorithm on the subtree rooted at v to coordinating the execution of the width-1 algorithm on that subtree. Processor $d(v)$ continues the execution of S-SOLVE*(v) until it is ready to expand a node. At that point, its pushdown stack contains a path g from node v down to the node being expanded. Processor $d(v)$ traverses that path, starting at v, and sends messages corresponding to the nodes it encounters, as follows. Let u be a node in the path g (the case $u = v$ is not excluded), let w be the left child of u, and let x be the right child of u. If w is on the path g then processor $d(v)$ sends the message "P-SOLVE**(u)" to processor d(u) and the message "S-SOLVE*(x)" to processor $d(u)+1$. If x is on the path g (in this case it is known that the value of w, the left child of u, is 0) then processor $d(v)$ sends the message "P-SOLVE***(u)" to processor $d(u)$. If u is the terminal node of the path g then it sends the message "P-SOLVE*(u)" to processor $d(u)$. When the traversal of the path is complete and all the required messages have been sent, the execution of P-SOLVE*(v) terminates.

When processor $d(v)$ receives message "P-SOLVE**(v)", it behaves as in case one of "P-SOLVE*(v)," except that v is already expanded and so there is no need to expand v and send the messages "P-SOLVE*(w)" and "S-SOLVE*(x)". It simply waits for the messages giving the values of w and x and then takes its subsequent actions.

The message "P-SOLVE***(v)" is similar to "P-SOLVE**(v)" except that, in addition, it is known that the value of the left child of v is 0. Thus, the task of processor $d(v)$ is to wait until it receives a message of the form "val(x) $= b$," where x is the right child of v. Upon receipt of this message it sends the message "val(v) $= 1-b$" to processor $d(v)-1$.

This completes our description of the implementation of N-Parallel SOLVE of width 1 for binary NOR-trees.

From the description above, one can deduce that the "pre-emption" rule achieves the correct pruning behavior. This ensures the correctness of the implementation. The major time delays introduced in our implementation occur with the actions of processor $d(v)$ in case two of "P-SOLVE*(v)". In this case, it has to traverse the path g maintained on its stack and send messages as it traverses. This traversal is considered as instantaneous in the node-expansion model. We show that the delays caused by these traversals can be incorporated into the path-counting in the proof of Proposition 6 of Section 5. Consequently, the conclusion of Theorem 4 holds for N-Parallel SOLVE, implemented as described. With each time step τ we associate a "base path" as follows. Let $X(\tau) = \{y|\text{ P-SOLVE*}(y)$ is being executed at time $\tau\}$. Let v be the rightmost node in $X(\tau)$. Let u be the node most recently visited by processor $d(v)$ during the execution of P-SOLVE*(v) (There are two cases; either $u = v$ or u is the last node reached in the top-down traversal of the path g held on processor $d(v)$'s stack). The *base path* is the path from the root of the NOR-tree to u. This base path has the following properties: i) the number of processors that are evaluating the right-siblings of the nodes on this base path is equal to the number of 1's in the "code" of the base path; ii) it has not been counted as a base path before; iii) it is a path counted in Proposition 6. By these properties, the base paths associated with the steps of the traversal will not increase the bounds stated in Proposition 6. One can conclude that our implementation does not compromise the linear speed-up of N-Parallel SOLVE over N-Sequential SOLVE.

Finally, we remark that our implementation can be adapted to the restriction of having only a fixed number p of processors available. This can be done by dividing levels of the input trees into "zones" of p consecutive levels, and let processor d be responsible for level d in each zone and switch between the levels in different zones by multiplexing.

8 Conclusion

We presented parallel algorithms that parallelize the sequential "left-to-right" algorithm for evaluating AND/OR trees and the sequential α-β pruning procedure for evaluating MIN/MAX trees. We showed that these parallel algorithms achieve a linear speed-up over their corresponding sequential algorithms on uniform trees, if the number of processors used is close to the height of the input tree.

All of our parallel algorithms presented are based on the same strategy. This strategy is a general and effective paradigm for parallelizing a class of sequential tree search algorithms. It is suitable for efficient implementations on various parallel computer architectures. We hope that the parallel algorithms presented here will suggest some efficient parallel programs for evaluating the game trees occurring in practice.

The major weakness of our results is that the effective speed-up is proved only in the case of a low degree of parallelism. Our algorithms allow an increasing degree of parallelism by increasing their width parameter. For example, when the width parameter is 2 or 3, the number of processors used on a uniform tree of height n is $O(n^2)$ or $O(n^3)$, respectively. We believe that the speed-up on uniform trees should remain linear in the number of processors for any fixed width. We are not able to prove this. The counting argument that works for width 1 is no longer applicable to higher widths. It appears that new proof techniques are needed for the analysis of higher widths.

Our results are asymptotic in the height of the input tree. The larger the branching factor is, the larger the height is needed to be. This should be contrasted with the "wide-and-shallow" game trees encountered in chess programs, which has relatively large branching factor and limited depth. The provable constant c in Theorem 1 is rather poor. Some simulations we did indicates that a better constant is achievable. It would be highly desirable to establish this theoretically.

References

[1] Althöfer, I., A parallel game tree search algorithm with a linear speedup, submitted to *J. Algorithms* in December, 1988.

[2] Baudet, G., The design and analysis of algorithms for asynchronous multiprocessors, Computer Science Tech. Rept. CMU-CS-78-116, Carnegie-Mellon University, Pittsburgh, PA, 1978.

[3] Ferguson, C., and Korf, R., Distributed tree search and its application to alpha-beta pruning, *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988 (AAAI-88).

[4] Finkel, R.A., Fishburn, J.P., Parallelism in alpha-beta search, *Artificial Intelligence* **19** (1982), 89-106.

[5] Knuth, D.E., and Moore, R.N., An analysis of alpha-beta pruning, *Artificial Intelligence* **6** (1975), 293-326.

[6] Marsland, T.A., Popowich, F., Parallel Game-Tree Search, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No.4, (1985), 442-452.

[7] Pearl, J., Heuristics, Addison-Wesley, 1984.

[8] Pearl, J., The solution for the branching factor of the alpha-beta pruning algorithm and its optimality, *CACM* **25**(8) 1982, 559-564.

[9] Saks, M., and Wigderson, A., Probabilistic Boolean Decision Trees and the Complexity of Evaluating, *27th Annual Symposium on Foundations of Computer Science*, 29-38, 1986.

[10] Tarsi, M., Optimal Search on Some Game Trees, *JACM* **30** (1983), 389-396.

[11] Vornberger, O., Parallel alpha-beta versus parallel SSS*, *Proceedings of the IFIP Conference on Distributed Processing,* Amsterdam, 1987.