

Parallel Algorithms for Combinatorial Search Problems

By

YanJun Zhang

M. S. (Wuhan University, China) 1981

M.A. (University of Massachusetts at Amherst) 1984

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: *Richard M. Karp* *11/20/89*
.....
 Chair *Ronald L. Rivest* *11/22/89* Date
.....
 David Alderson *11/27/89*
.....

Parallel Algorithms for Combinatorial Search Problems

by

Yanjun Zhang

Abstract

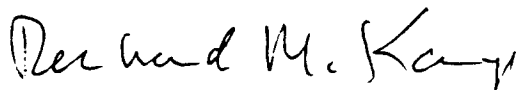
This thesis is a theoretical study of parallel algorithms for combinatorial search problems. In this thesis we present parallel algorithms for backtrack search, branch-and-bound computation and game-tree search.

Our model of parallel computation is a network of processors communicating via messages. Our primary interest in a parallel algorithm is its speed-up over the sequential ones. Our goal is to design parallel algorithms that achieve a speed-up proportional to the number of processors used.

We first study backtrack search that enumerates all solutions to a combinatorial problem. We propose a simple randomized method for parallelizing sequential backtrack search algorithms for solving enumeration problems. We show that, uniformly on all instances, this method is likely to achieve a nearly best possible speed-up.

We then study the branch-and-bound method for solving combinatorial optimization problems. We present a randomized method called Local Best-First Search for parallelizing sequential branch-and-bound algorithms. We show that, uniformly on all instances, the execution time of this method is unlikely to exceed a certain inherent lower bound by more than a constant factor.

In the rest of this thesis we study the problem of evaluation of game trees in parallel. We present a class of parallel algorithms that parallelize the “left-to-right” algorithm for evaluating AND/OR trees and the α - β pruning algorithm for evaluating MIN/MAX trees. We prove that the algorithm achieves a linear speed-up over the left-to-right algorithm on uniform AND/OR trees when the number of processors used is close to the height of the input tree. We conjecture that the same conclusion holds for the speed-up of the algorithm over the α - β pruning algorithm.



Professor Richard M. Karp

Committee Chairman

Acknowledgements

I would like to express my deep gratitude to my thesis advisor, Richard Karp, who introduced me to the research presented in this thesis. We have worked together on this research over the last three years, and this thesis is an outcome of our joint work. It has been a precious experience to work with him. Especially, I thank him for his guidance in doing research from which I shall benefit for the rest of my life. I also thank him for being an elucidating and rigorous teacher. Finally, I thank him for his kindness, patience and financial support that helped me complete my graduate studies.

I thank Raimund Seidel for his encouragement and for serving on my committee, Manuel Blum for his interests in listening to my work, and David Aldous for serving as my third committee member and for helpful discussions on probability.

I thank my fellow theory students I have had at Berkeley: Sampath Kannan, Lisa Hellerstein, Sally Floyd, Valerie King, Joel Friedman, Danny Soroaker, Arvind Raghunathan, Phil Gibbons, Steven Rudich, Mark Gross, David Wolfe, Russell Impagliazzo, Prabhakar Ragde, Naomi Nishimura, Marshall Bern, Alice Wong, Jon Frankle, Howard Karloff, Miklos Santha, and others, for their friendship and support. I also thank my non-theory friends Yongdong Wang and Chien Chen, and Joe Yang for his cooperation on the implementation presented in Section 7.7.2..

I thank Mike Sipser and Gary Miller for their initial help that led me to theoretical computer science.

I thank my family in China for their encouragement and patience during the years of my graduate school abroad.

Finally, I thank my wife, Jun Ren, for her love and support throughout my Berkeley years.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Tree Searches	2
1.3	Model of Parallel Computation	3
1.4	Summary	4
1.5	Terminology	5
2	Backtrack Search	6
2.1	Introduction	6
2.2	Two Examples	7
2.3	Abstraction of Backtrack Search	9
2.4	Previous Work on Parallel Backtrack Search	11
3	Parallel Backtrack Search	13
3.1	A Generic Algorithm	13
3.2	Properties of the Generic Algorithm	17
3.3	Parallel Backtrack Search Algorithms	20
3.3.1	A Deterministic Algorithm	21
3.3.2	Randomized Algorithm 1: Busy-Initiated	22
3.3.3	Randomized Algorithm 2: Idle-Initiated	26
3.3.4	PRAM Implementation	27
3.4	Open Problems and Further Research	29
4	Branch-and-Bound Procedure	30
4.1	Introduction	30
4.2	An Example	32
4.3	Generic Branch-and-Bound Algorithms	32

4.4	Previous Work on Parallel Branch-and-Bound	34
5	A Randomized Parallel Branch-and-Bound Procedure	36
5.1	Local Best-First Search	36
5.2	Proof of Theorem 6	40
5.2.1	A Queueing Process	41
5.2.2	A Continuous-Time Model	45
5.2.3	Shooting Gallery Game	49
5.2.4	A Simple Process	50
5.2.5	The Distribution of M_k	53
5.3	PRAM Implementation	57
5.4	Remaining Issues	60
5.4.1	Some Omitted Details	60
5.4.2	Consequence of Equal Costs	61
5.5	Open Problems and Further Research	61
6	Game Tree Search	63
6.1	Introduction	63
6.2	Models of Computation	64
6.3	Sequential Algorithms	65
6.3.1	Sequential SOLVE	65
6.3.2	α - β Pruning	68
6.4	Previous Work on Parallel Game-Tree Search	71
7	Parallel Game-Tree Evaluation Algorithms	74
7.1	Parallel SOLVE	74
7.2	Proof of Main Theorem	76
7.3	Parallel α - β	83
7.4	Node-Expansion Model	88
7.5	Fixed Number of Processors	91
7.6	Randomization	93
7.7	Implementations	94
7.7.1	Level-Allocation Method	95
7.7.2	Dynamic-Allocation Method	99
7.8	Open Problems and Further Research	102

Chapter 1

Introduction

1.1 Motivation

Combinatorial search methods such as the branch-and-bound procedure are used in practice for solving combinatorial search problems, and they have been extensively studied in operations research, computer science and artificial intelligence.

To solve a combinatorial search problem, one often needs to search through a large set of possible arrangements to find a desired solution. In doing so, one encounters the phenomenon of “combinatorial explosion” — the time or space used to solve the problem grows exponentially with the size of the problem, which severely limits the range of problems that can be solved. Worst of all, the combinatorial explosion is believed to be unavoidable for many problems in the worst case.

In the last few years we have witnessed a rapid development of parallel computing technology. A parallel computer can speed up a computation significantly by solving different computational tasks simultaneously. Combinatorial search problems are well suited for parallelism — the set of possible arrangements can be searched through in parallel so that the desired solution may be found substantially faster. Parallelism may well be our most powerful means to reduce the effect of the combinatorial explosion in solving combinatorial search problems.

How can we perform a combinatorial search in parallel? This question represents a research area with both theoretical and practical significance. Considerable research has been conducted on the parallel processing of combinatorial searches. But theoretical studies are so far scarce.

This thesis is a theoretical study of parallel algorithms for combinatorial search

problems. In this thesis we present parallel algorithms for backtrack search, branch-and-bound computation and game-tree search. We prove that these algorithms achieve a good speed-up over their sequential counterparts. We believe that these algorithms can lead to useful programs for solving combinatorial search problems in practice.

1.2 Tree Searches

The combinatorial search methods that we will study in this thesis share a common view: they can be thought of as a tree search. We state this view abstractly here.

We are given an algorithm for solving some combinatorial search problem. The algorithm has a certain procedure, which we shall call GEN, to generate subproblems. When GEN is applied to a subproblem A , it either solves A directly or derives from A a set of subproblems A_1, A_2, \dots, A_d such that the solution of A can be found from the solutions of A_1, A_2, \dots, A_d .

Given a problem instance as the input to the algorithm, we associate with the problem instance a rooted tree as follows. The root of the tree corresponds to the given problem instance. An internal node corresponds to a subproblem of the given problem instance. A leaf presents a subproblem that can be solved directly by the procedure GEN. The children of node v in the tree correspond to the set of subproblems that can be derived by the procedure GEN from the subproblem represented by v .

The execution of the algorithm corresponds to a search in the tree associated with the input problem instance. The search starts with the root of the tree and generates certain nodes of the tree as the search progresses. The nodes of the tree are generated by using the so-called “node expansion” operation, which corresponds to a call of procedure GEN. When this operation is applied to v it either determines that v is leaf or produces the children of v . A node can be expanded only if it is the root of the tree or if it is a child of some node previously expanded. The search, starting with the root, successively applies node expansion to the unexpanded nodes of the tree until a set of leaves are identified as a solution to the problem.

In addition to the procedure GEN, the algorithm needs a rule to determine the order in which subproblems are explored. In terms of the associated tree, the algorithm uses this rule to decide which node or which set of nodes to be expanded

next; the decisions are based on the information gathered from the nodes that have been generated so far. Different rules may result in drastically different search behaviors. The portion of the tree that will be generated as well as the outcome of the search may depend on the choice of the rule.

An algorithm using p processors can expand up to p nodes simultaneously at one step. The *number of processors used* by a search algorithm is the maximum number of nodes expanded at some step. An algorithm is *parallel* if the number of processors used by the algorithm is greater than one; otherwise, it is *sequential*.

In the subsequent chapters, we shall present a number of parallel algorithms for solving combinatorial search problems and compare the performance of these parallel algorithms with that of their sequential counterparts.

1.3 Model of Parallel Computation

Our model of parallel computation is a multiprocessor system in which there is no global memory and processors communicate via messages. In a multiprocessor system, each processor has its local memory and is capable of performing computations on the data gathered in its local memory; the processors are connected by a network over which messages are sent. There is no central control in the network. Two processors are neighbors if they are connected directly by the network. A network is *fully-connected* if any two processors of the network are neighbors. Two neighboring processors can send one message to each other in unit time. We assume that the input/output is handled by a single designated processor.

We assume that the processors are synchronized. A *step* of computation consists of two phases: the *computation phase* in which each processor performs some computation locally and the *communication phase* in which each processor can send a message to each of its neighboring processors. The *execution time* of an algorithm is the number of steps it takes. A fundamental measure of a parallel algorithm is its “speed-up” over the sequential algorithms. The *speed-up* of a parallel algorithm A over a sequential algorithm B is the ratio of the execution time of A to the execution time of B with respect to the same input instance. We express the speed-up as a function of the number of processors used by the parallel algorithm.

The assumption that processors are synchronized is for the convenience of describing and analyzing our algorithms. For the majority of our algorithms, the

correctness of the algorithm holds if the the processors are not synchronized. As a result, most of our algorithms can be implemented directly on a distributed system.

1.4 Summary

In this section we summarize the content of this thesis. The thesis consists of three parts, each of two chapters, on backtrack search, branch-and-bound method and game tree search, respectively.

In the next two chapters we study backtrack search. In Chapter 2 we introduce backtrack search and make an essential distinction between the backtrack search that seeks only one solution and the backtrack search that seeks all solutions. The latter will be called a backtrack enumeration. We shall only consider the parallelization of backtrack enumeration. In Chapter 3 we present three simple algorithms for parallelizing sequential backtrack search. The first algorithm is deterministic but requires global control of the computation. The other two are randomized implementations of the deterministic algorithm and they require no global knowledge of the computation. We prove that, in case of a fully-connected network, both randomized algorithms are likely to yield a speed-up within a small factor from optimal in general and within a constant factor from optimal in some important special cases.

In the subsequent two chapters we study the branch-and-bound method. In Chapter 4 we introduce the branch-and-bound method and identify an inherent lower bound on the execution time of any branch-and-bound algorithm. In Chapter 5 we present a randomized parallel branch-and-bound procedure. The procedure is a parallel implementation of the “best-first” search strategy with no global data structures or complex communication protocols. We prove that, in the case of a fully-connected network, the execution time of this procedure is likely to be within a constant factor of the inherent lower bound.

In the last two chapters we study game tree search. In Chapter 6 we introduce the standard “left-to-right” sequential algorithm for evaluating AND/OR trees and the well-known α - β pruning algorithm for evaluating MIN/MAX trees. In Chapter 7 we present a class of parallel game-tree evaluation algorithms that parallelize the left-to-right algorithm and the α - β pruning algorithm. We show that, on any instance of a uniform AND/OR tree, the algorithm achieves a linear speed-up over

the left-to-right algorithm when the number of processors used is close to the height of the input tree. We conjecture that the same linear speed-up also hold for the α - β pruning algorithm. In the last part of the chapter we discuss the implementation of some of the algorithms we have presented.

1.5 Terminology

We fix some terminology that is basic to our presentation. We assume that we are given a rooted tree with root r .

An *ancestor* of a node v is a node on the path from r to v . So a node is an ancestor of itself. Node u is a *descendant* of v if v is an ancestor of u . A *leaf* is a node with no children. A *root-leaf path* is a path from r to some leaf. Two nodes v and u are *siblings* if v and u have the same parent.

A rooted tree is *ordered* if the children of each internal node are ordered. In an ordered tree, v is a *left-sibling* (*right-sibling*) of u if v and u have the same parent x , and v precedes (follows) u in the ordering of the children of x . Let Σ be a set of nodes such that no two nodes of Σ are on the same root-leaf path. For any such set Σ , a left-to-right ordering upon the nodes of Σ is defined as follows. For $v, u \in \Sigma$ and $v \neq u$, v is *on the left of* u if v has an ancestor v' and u has an ancestor u' such that v' is a left-sibling of u' ; otherwise, v is *on the right of* u . In particular, the *leftmost* (*rightmost*) node in Σ is the node that is on the left (right) of every other node in Σ . A subtree T is *on the left of* another subtree T' if the root of T is on the left of the root of T' .

Chapter 2

Backtrack Search

2.1 Introduction

Backtrack search is an enumerative method for solving combinatorial search problems. Problems that yield to backtrack search have an essential property: it is possible to determine that some initial choices cannot lead to a solution. This property allows the search procedure to terminate an unproductive search and then “backtrack” to a point where a new search can be started.

The idea of backtrack search is best understood in the context of finding an exit in a maze: starting at the entry, keep extending the path from the entry until an exit is reached; when facing a dead end, retreat one step and try to extend the path in another direction. More formally, backtrack search works by continually trying to extend a partial solution to a problem; when it is found that the current partial solution can not possibly be extended to a complete solution, the algorithm then backtracks to its previous partial solution and attempts to extend that partial solution again in a way that has not been attempted previously. This process is repeated until a solution is found or it is found that there is no solution.

Backtrack search is most useful for enumerating all the solutions to a given combinatorial problem. We call a backtrack search that solves an enumeration problem a *backtrack enumeration*. A backtrack enumeration can be much more efficient than a brute-force method that generates all possible configurations in some fashion and tests whether each of these configurations constitutes a solution. A discussion of backtrack enumeration can be found in [RND77].

In essence, a backtrack search implicitly enumerates all the configurations associ-

ated with the given problem. The running time of a backtrack search, though small in many cases, may be exponential in the size of the problem. Studies on the complexity of backtrack search can be found in [CSW85] and [Ga76]. The latter shows that a large class of search strategies, including backtrack search, are exponential in the worst case.

2.2 Two Examples

Example 1: The *Eight Queens Problem* is to place eight queens on a 8×8 chess board such that no queen can attack another, that is, no two queens are on the same row or on the same column or on the same diagonal of the board. A non-attacking arrangement of eight queens is possible.

The Eight Queens Problem can be solved by backtrack search. A (complete) solution to the Eight Queens Problem is a non-attacking arrangement of eight queens. A partial solution is the arrangement of some queens such that no two of them are attacking each other. We call a row *occupied* if it contains a queen; otherwise, it is *unoccupied*. To find a non-attacking arrangement, we proceed as follows. We will place one queen on each row in the increasing order of rows. We shall maintain the property that the arrangement of the queens that are currently on the board is a partial solution, which is called the current partial solution. Given the current partial solution, we try to extend it by placing a queen at the first position of the first unoccupied row such that (i) the newly placed queen does not attack another queen already on the board and (ii) this extension was not attempted previously. If such a position exists, we place a queen at that position; otherwise, we know that the current partial solution can not be extended to a solution. Notice that we may still be able to add a queen in some other row such that the newly added queen will not attack another queen. But the point is that we can no longer place all the rest of queens without attacks. So we backtrack — remove the queen that was last placed on the board. We then try to extend the current partial solution again in some way that has not been attempted. This process is repeated until all queens are placed on the board.

The above algorithm can be easily modified to solve the enumerative version of the Eight Queens Problem in which one is asked to find all non-attacking arrangements of eight queens. Instead of stopping at the first complete solution it finds,

the algorithm backtracks from each complete solution it encounters after it records the solution. The algorithm stops when it finishes extending the partial solution that consists of one queen on the last position of the first row.

Example 2: The *Enumerative Satisfiability Problem* (ESAT) is to enumerate all satisfiable assignments, if any, of a given boolean formula in conjunctive normal form. We give a backtrack algorithm that solves ESAT.

Let f be the input formula and let x_1, x_2, \dots, x_n be all variables of f . Each variable of f has a value which is either 0 or 1 or \star . Initially, each variable has the value \star . A variable is *fixed* if it has a value of 0 or 1. The *present state* consists of the current values of the variables. We say f is *falsified* by the present state if all the variables of some clause of f are currently fixed and that clause is evaluated to 0 given the current values of the variables. Whether a formula in conjunctive normal form is falsified by the present state can be checked in linear time in the size of the formula.

We fix the variables of f in the increasing order. We maintain the property that the set of fixed variables consists of x_1, x_2, \dots, x_k for some $k \leq n$ and f is not falsified by the present state. We first consider the case $k < n$. In this case, some variables are not fixed. We then attempt to fix the first unfixed variable x_{k+1} . There are three cases. *Case 1:* We have not attempted to fix x_{k+1} previously under the present state. In this case, we fix x_{k+1} to 0 if f is not falsified by fixing x_{k+1} to 0; otherwise, we fix x_{k+1} to 1 if f is not falsified by fixing x_{k+1} to 1. If both way of fixing x_{k+1} falsify f , we know that the current values of $\{x_1, x_2, \dots, x_k\}$ cannot possibly lead to a satisfying assignment of f . So we backtrack — “unfix” x_k by setting the value of x_k to \star . *Case 2:* x_{k+1} was previously fixed to 0 but not to 1 under the present state. In this case, we fix x_{k+1} to 1 if f is not falsified by fixing x_{k+1} to 0; otherwise, we backtrack by setting the value of x_k to \star . *Case 3:* x_{k+1} was previously fixed to both 0 and 1 under the present state. In this case, we backtrack immediately by setting the value of x_k to \star . In case that $k = n$, all the variables are fixed, and the values of these fixed variables forms a satisfiable assignment of f . The algorithm records this assignment and backtracks by setting $k = n - 1$. The process halts when it attempts to backtrack from x_1 , and outputs all the recorded assignments if f is satisfiable; otherwise, outputs “unsatisfiable”.

Figure 2.2 contains a procedure called ESAT describing the given backtrack algorithm. ESAT takes f as its input and enumerates all the satisfying assignments

of f . Subroutine CHECK checks whether f is falsified by the present state, and returns 0 if only f is falsified by the present state. The inputs to subroutine CHECK are f and the values of the fixed variables in the present state. A call of CHECK runs in a linear time in the size of f .

2.3 Abstraction of Backtrack Search

A backtrack search can be viewed as a search through a tree of partial solutions. Given a problem instance, we associate with it a rooted tree as follows: the root represents the empty solution; an internal node represents a partial solution; a child of an internal node represents one possible minimal extension to the partial solution represented by the internal node; a leaf of this tree represents either a solution or a *blocked* partial solution, i.e., a partial solution that can not be extended to a solution. A minimal extension in the Eight Queens Problem is to add a queen on the first row that contains no queen, and a minimal extension in ESAT is to fix the first unfixed variable. The execution of the backtrack search corresponds to a depth-first search in this tree using the “node expansion” operation. When this operation is applied to a node v , it either determines that v is a leaf or generates the children of v , which are all possible minimal extensions of the partial solution represented by v . The search terminates when it finds one or more leaves representing the desired solution. The extensions to a particular partial solution depends solely on the partial solution itself. In terms of the associated tree, the node expansions in one subtree have no effect on the node expansions in another subtree. This property is essential for executing backtrack search in parallel. We assume that the traversal of the tree is *lexicographic*, i.e., the children of internal nodes are expanded in the left-to-right order.¹

A backtrack search that seeks only one solution may look at only a small portion of the tree if a leaf representing a solution appears to the far left among all the leaves of the tree. It would be difficult or perhaps impossible to parallelize such a search with a guaranteed good speed-up. The situation is quite different when we consider backtrack enumerations. A backtrack enumeration must generate the *entire* tree associated with the input problem instance. The reason is that the

¹The effect of a different traversal ordering can be achieved by arranging the order of the children of the expanded nodes.

```

ESAT( $f$ );
{
  let  $x_1, x_2, \dots, x_n$  be all the variables of  $f$ ;
  for  $i = 1$  to  $n$  do  $x_i \leftarrow \star$ ;
   $i \leftarrow 1$ ;  FOUND  $\leftarrow 0$ ;

  while ( $i > 0$ ) {
    if ( $x_i = \star$ ) {
       $x_i \leftarrow 0$ ;
      if (CHECK( $f, x_1, x_2, \dots, x_i$ )  $\neq 0$ )  $i \leftarrow i + 1$ ;
      else {
         $x_i \leftarrow 1$ ;
        if (CHECK( $f, x_1, x_2, \dots, x_i$ )  $\neq 0$ )  $i \leftarrow i + 1$ ;
        else { /* backtrack */
           $x_i \leftarrow \star$ ;   $i \leftarrow i - 1$ ; }
      }
    }
    else if ( $x_i = 0$ ) {
       $x_i \leftarrow 1$ ;
      if (CHECK( $f, x_1, x_2, \dots, x_i$ )  $\neq 0$ )  $i \leftarrow i + 1$ ;
      else { /* backtrack */
         $x_i \leftarrow \star$ ;   $i \leftarrow i - 1$ ; }
    }
    else if ( $x_i = 1$ ) { /* backtrack */
       $x_i \leftarrow \star$ ;   $i \leftarrow i - 1$ ; }
    /* end of if */

    if ( $i = n + 1$ ) { /* a solution found */
      record the values of  $x_1, x_2, \dots, x_n$  as a solution;
      FOUND  $\leftarrow 1$ ;   $i = n$ ; /* backtrack */ }
  } /* end of while */

  if (FOUND = 0) return("unsatisfiable");
  else output all the recorded solutions;
}

```

Figure 2.2

algorithm cannot tell whether a partial solution can be extended to a solution unless the partial solution corresponds to a leaf in the tree. Hence, every node in the tree must be expanded. To parallelize a backtrack enumeration, we can let each processor generate a different subtree. As every node of the tree must be expanded, there can be no wasteful node expansions. Therefore, a good speed-up can be expected if the processors are kept busy performing node expansions. In other words, the problem of achieving a good speed-up is really a problem of achieving a good “load-balancing”. We shall show that a good load-balancing can be achieved. This is the fundamental reason that backtrack enumeration admits efficient parallelization.

2.4 Previous Work on Parallel Backtrack Search

There is little published literature on parallel backtrack search. The most relevant paper is [FM87] which describes a software package called DIB, “distributed implementation of backtracking”, for writing distributed or parallel programs involving backtracking, which includes backtrack search as well as branch-and-bound computation and game-tree search. This paper is closely related to what we will present for parallel backtrack search. We shall focus our discussion on this paper in the context of backtrack search.

Program DIB is given as its input the root of the tree representing the problem to be solved. It solves the subproblem represented by a node v of the input tree by solving all the subproblems represented by the children of v . The input tree is searched concurrently in which each processor works on a different set of subtrees. Each processor performs the depth-first search on its subtrees in the lexicographic ordering. When a processor finishes traversing all its subtrees, it seeks new work from other processors by sending requests for work to other processors. There are two components in the work-sharing scheme of DIB. One component is the “donation rule” which determines how one processor gives away some of its work to another processor; the other is the “request rule” which determines how a processor with no work seeks new work.

The donation rule of DIB, with some details left unspecified, is as follows. When a processor receives a request for work, it does the following: If it has subtrees other than its working subtree, the subtree it currently traverses, it donates some of the

earliest ones in the lexicographic ordering to the requesting processor. If not, it goes down along the depth-first-search path in its working subtree to find the first node that has unvisited siblings. If this node is not at the end of the path, the subtrees rooted at some of the leftmost siblings of that node are donated to the requesting processor; otherwise, the request is not granted. The number of subtrees donated in each donation is not specified. The donation rule for the case that a processor receives more than one request at the same time is not discussed.

As to the request rule, two different methods are considered. One method is to organize the processors into a ring. In this method, an idle processor seeks work by going around the ring. More precisely, each processor has a “helper”. The helper for processor i is initially processor $(i + 1) \bmod p$ where p is the total number of processors in the ring. An idle processor requests work from its helper; if its helper has no work to share, the request is forwarded to the successor in the ring; the request is forwarded along the ring until the request is granted and the helper of the requesting processor is reset to be the successor of the processor that granted the request. In practice, the requests for work tend to be distributed fairly evenly in the ring. The main drawback of this method is that it is not fault-tolerant. Another drawback is that the forwarded messages tend to flood the network near the end of the computation, when many processors are idle.

The second method proposed in [FM87] for sending requests uses randomization. This method is to let each idle processor send requests to k other randomly selected processors for some constant $k \geq 1$, where the choice of k depends on the number of processors used and on the application. Requests that cannot be granted are ignored, not forwarded. This method tends to keep the number of messages small, even near the end of the computation. With a proper choice of k , it is rare that an idle processor remains idle through many cycles of sending requests, except close to the end of the computation. The paper did not specify whether a requesting processor would accept all the new work in response to its k requests when $k > 1$. Interestingly, the special case with the choice of $k = 1$ coincides with the so-called “idle-initiated” method we present in the next chapter.

The experimental results provided by the paper show that DIB performs excellently on backtrack enumeration problems such as finding all non-attacking arrangements of eight queens, but less impressively on game-tree search.

Chapter 3

Parallel Backtrack Search

In this chapter we study parallel backtrack search. Our main result is a simple randomized method for parallelizing sequential backtrack enumeration algorithms. We present two implementations of this method, the Busy-Initiated Backtrack Search and the Idle-Initiated Backtrack Search. The latter improves upon the former in terms of efficiency. We show that, in the case of a fully-connected network where each processor is directly connected to all other processors, both algorithms are likely to achieve a nearly optimal speed-up on any instance.

3.1 A Generic Algorithm

We give a generic description of the parallel backtrack search algorithms we wish to study. Specific algorithms are given in the next section.

We are given a sequential backtrack search algorithm we wish to parallelize. Let H be the rooted tree of partial solutions associated with the given algorithm on some given input instance. Let r be the root of H . Let n be the number of nodes in H , and let h be the number of nodes in a longest root-leaf path in H . Then the execution time of any algorithm is at least h , since the nodes along a path must be expanded one at a time, and the execution time of any p -processor algorithm is at least n/p , since all n nodes in H must be expanded and the algorithm can expand at most p nodes at a single step. Thus $\max\{n/p, h\}$ is an inherent lower bound on the execution time of any p -processor algorithm on the instance H . Our goal is to design p -processor algorithms whose execution time comes close to this lower bound.

Let p be the number of processors and P_i be the i -th processor. A *frontier node* is a node that has been generated but not expanded. The frontier nodes are distributed among the processors, with each frontier node belonging to exactly one processor. The *local frontier* of P_i , denoted by F_i , is the set of frontier nodes of H possessed by P_i . A processor is *busy* if its local frontier is non-empty; otherwise, it is *idle*. A processor is *overloaded* if it is busy and has more than one frontier node. The *level* of a node v , denoted by $l(v)$, is the number of nodes on the path from r to v . A *top-node* of P_i is a frontier node of minimum level in F_i . Let T_i denote the set of top-nodes of P_i . Let $\Gamma(v)$ denote the set of children of an internal node v .

Figure 3.1 contains a generic description of the kind of algorithm we consider. A step of the algorithm is an execution of the while loop. Each step consists of a *Node Expansion Step* in which each busy processor expands its leftmost frontier node, a *Decision Step* in which some overloaded processors are paired up, one-to-one, with some idle processors and a *Donation Step* in which each of the paired overloaded processors transfers some of its top-nodes to its paired idle processor. A *donation* is a set of frontier nodes transferred from one processor, called the *donating* processor, to another, called the *receiving* processor. When the context is clear we also use the word “donation” to refer to the transfer of a donation. We call the set R in the Decision Step the *pairing set*. Our generic algorithm leaves the pairing set unspecified. Different ways of specifying the pairing set lead to different algorithms. We shall present in the next section some algorithms that use very different approaches in specifying the pairing set.

At the Node Expansion Steps, a busy processor keeps expanding its leftmost frontier node while receiving no new frontier nodes from other processors. The computation of the processor corresponds to the lexicographic depth-first traversal of some subtree. A busy processor would complete a traversal of a subtree, with a possibility of donating some portions of the subtree to other processors, before it starts to traverse another subtree.

Each processor can manage its frontier nodes conveniently by using a stack in its local memory. When a processor is idle, its stack is empty. By Proposition 1 of the next section, the set of nodes in a donation is a set of (consecutive) sibling nodes. After receiving a donation which consists of a set of siblings, a processor initializes its stack by pushing the received nodes onto the stack in the reverse order of their sibling ordering. When its stack is non-empty, the processor removes the node at

Generic Parallel Backtrack Search

```

/* Initialization */
 $F_1 = \{r\};$ 
for  $i = 2, 3, \dots, p$ ,  $F_i \leftarrow \emptyset;$ 

while some  $F_i \neq \emptyset$  do
    /* Node Expansion Step */
    for  $i = 1, 2, \dots, p$  in parallel do
        if  $F_i \neq \emptyset$  then
            let  $v_i$  be the leftmost node in  $F_i;$ 
            expand  $v_i;$ 
             $F_i \leftarrow F_i \setminus \{v_i\};$ 
            if  $v_i$  is not a leaf then  $F_i \leftarrow F_i \cup \Gamma(v_i);$ 

    /* Decision Step */
     $N \leftarrow \{(i, j) : |F_i| > 1, |F_j| = 0, 1 \leq i \leq p, 1 \leq j \leq p\};$ 
     $N_i \leftarrow \{j : (i, j) \in N\}$  for  $1 \leq i \leq p;$ 
     $N^j \leftarrow \{i : (i, j) \in N\}$  for  $1 \leq j \leq p;$ 
    determine a subset  $R \subseteq N$  such that
     $|R \cap N_i| \leq 1$  for  $1 \leq i \leq p$  and  $|R \cap N^j| \leq 1$  for  $1 \leq j \leq p;$ 

    /* Donation Step */
    for  $i = 1, 2, \dots, p$  in parallel do
        let  $T_i$  be the set of top-nodes in  $F_i;$ 
        let  $D_i \subseteq T_i$  be the set of the rightmost  $|D_i| = \lceil |T_i|/2 \rceil$  nodes in  $T_i;$ 
        if  $(i, j) \in R$  for some  $j$  then /*  $i$  donates  $D_i$  to  $j$  */
             $F_i \leftarrow F_i \setminus D_i;$ 
            send message " $i$  donates  $D_i$  to you" to  $j;$ 

    for  $j = 1, 2, \dots, p$  in parallel do
        if  $j$  receives message " $i$  donates  $D_i$  to you" then  $F_j \leftarrow D_i.$ 

```

Figure 3.1

the top of the stack, expands it, and pushes its children, if any, onto the stack in the reverse order of the children. This ensures that the leftmost frontier node is always at the top of the stack and the rightmost one at the bottom. Again, by Proposition 1, the top-nodes of the processor, either received or generated, are the rightmost nodes among the frontier nodes of the processor, hence, at the bottom of the stack. We assume that we can remove a node at the bottom of the stack. To donate, a processor removes about half of the top-nodes at the bottom of the stack, which are the rightmost half of the top-nodes.

The pairing set R in the Decision Step imposes the following conditions for donation:

- (a) only idle processors may receive donations;
- (b) only overloaded processors may donate;
- (c) a donating processor may donate to only one processor at a time;
- (d) a receiving processor may receive donations from only one processor at a time;
- (e) a donating processor donates about half of its top-nodes.

The motivation behind conditions (a)–(e) is as follows. As every node expansion is helpful, there is no reason for a busy processor not to continue expanding its frontier nodes. Thus a busy processor has no need for receiving a donation. This gives (a). When a busy processor has only one frontier node, the processor can expand its only frontier node immediately, which is better than giving the node to another processor and leaving itself idle. This gives (b). There is no strong reason for (c). Actually, it would be advantageous to allow a processor to donate to more than one idle processor at the same time so that all these idle processors can start to work. We adopt (c) for simplicity. Condition (d) is, however, necessary for our analysis. In particular, Proposition 1 would not hold if we did not have (d). There is asymmetry between (c) and (d). As an idle processor can start to work after receiving a donation from one processor, relaxing (d) seems to give no obvious advantage to the computation. The reason for (e) is less straightforward. As we will see, it turns to be a good strategy for a processor to donate its top-nodes. Furthermore, a donating processor should donate about half of its top-nodes, because it shares evenly its top-nodes with the only processor it donates to.

The size of the message required for a donation is important in practice. If the donated nodes are unrelated to each other, the size of the message may have to be proportional to the number of donated nodes. However, by Proposition 1, the top-nodes of a processor are actually consecutive siblings. Hence, the collection of the donated nodes in one donation permits a succinct description. One method to ensure short messages for donation, though somewhat wasteful in terms of node expansion, is to specify the parent of the donated top-nodes and the interval of the donated top-nodes among their siblings; the receiving processor can create the donated nodes by applying node expansion to the node specified in the message and extracting the children of the expanded node in the specified interval. We call this the *parent-specified* method.

The donation scheme governed by conditions (a)–(e) is in principle similar to the donation scheme of DIB [FM87] discussed in the last section of the previous chapter. However, the given description of DIB leaves some important details unspecified, such as the number of nodes donated at one time and, in particular, whether condition (d) is enforced. Hence, one cannot deduce with certainty that a processor, according to DIB, donates only its top-nodes, though it seems to be the case. In comparison, one may regard the conditions (a)–(e) as one specification of the donation scheme considered in [FM87].

3.2 Properties of the Generic Algorithm

In this section we prove some general properties of the generic parallel backtrack algorithm described in the preceding section.

The following proposition gives a fundamental property of our generic parallel backtrack algorithm. This property is a consequence of conditions (a) and (d) and the lexicographic depth-search traversal.

Proposition 1 *At any time, (i) the top-nodes of each processor are consecutive siblings and (ii) the top-nodes of each processor are the rightmost frontier nodes of that processor.*

Proof: We prove the proposition by induction. Initially, the root of H is the only frontier node, and the proposition holds trivially. We assume inductively that the proposition holds at step t and show that it holds at step $t+1$. Consider the actions

of P_i in step t , separately at the Node Expansion Step and at the Donation Step of step t .

At the Node Expansion Step, consider three cases. *Case 1*: P_i is idle. In this case, the inductive hypothesis implies that the proposition holds after the Node Expansion Step. *Case 2*: P_i is busy but not overloaded. In this case, $F_i = T_i = \{v\}$ for some v , and v is expanded at this step. Afterwards, $F_i = T_i$ is either empty, if v is a leaf, or consists of all the children of v . In both cases, the proposition holds after v is expanded. *Case 3*: P_i is overloaded. In this case, the leftmost node u of F_i is expanded at this step. As $|F_i| > 1$, each node in F_i that is a top-node of P_i after u is expanded was a top-node of P_i before u is expanded. Then, no matter u is a top-node or not, by the inductive hypothesis on (i), the top-nodes of P_i after u is expanded are consecutive siblings. The children of u , if any, are to the left of v . Together with the inductive hypothesis on (ii), the top-nodes of P_i are the rightmost node in F_i after u is expanded. So the proposition holds after u is expanded. In all cases, the proposition holds after the Node Expansion Step.

At the following Donation Step, P_i can either donate or receive or do nothing. *Case 1*: P_i donates. There are two subcases. *Subcase 1.1*: $|T_i| > 1$. In this case, P_i keeps left half of its top-nodes after the donation. The inductive hypothesis implies that the proposition holds after the donation. *Subcase 1.2*: $T_i = \{v\}$. In this case, P_i donates v . The frontier nodes of P_i after the donation are the nodes that are generated in the lexicographic depth-first traversal of a subtree. Among these nodes, those with minimal level must be consecutive siblings and appear at the end of the lexicographic ordering. Hence, the proposition holds after v is donated. *Case 2*: P_i receives a donation. By condition (a), the local frontier of P_i must be empty prior to this step. By condition (d), P_i receives a donation from only one processor, say P_k , and the received frontier nodes are a set of top-nodes of P_k . By the inductive hypothesis, the top-nodes of P_k are a set of consecutive siblings. So the frontier nodes of P_i are consecutive siblings after the donation. So the proposition holds for P_i after P_i receives the donation. *Case 3*: P_i does nothing. In this case, the inductive hypothesis implies that the proposition holds for P_i after the Donation Step. In all cases, the proposition holds after the Donation Step.

The induction is complete. \square

Corollary 1 *Suppose that H is a binary tree in which each internal node has exactly two children. Then a processor only donates one node at a time, its rightmost*

frontier node. \square

A node is allowed to be donated more than once. The number of times a node can be donated is limited by the number of siblings it has.

Proposition 2 *A node can be donated at most $\lceil \log d \rceil$ times where d is the number of children of its parent. In particular, for a binary tree H , any node of H can be donated at most once.*

Proof: By Proposition 1, the top-nodes are siblings. By condition (e), the set of siblings together with any given node v is halved in each donation. Hence, v can be involved in at most $\lceil \log d \rceil$ donations before either v is expanded or v is the only node in a donation. In the latter case, v will be expanded immediately after v is donated. \square

A notion that plays an important role in our analysis is that of the “base node” of a busy processor. The *base node* of a busy P_i , denoted by b_i , is defined as v if v is the only frontier node of P_i ; otherwise, it is defined as the parent of the top-nodes of P_i . In the latter case, the base node is well-defined by Proposition 1. Different processors may have the same base node; the base node of one processor may be an ancestor of the base node of another processor. The *level* of a busy processor is the level of its base node.

Proposition 3 *Suppose that $|F_i| = 1$ just after a Node Expansion Step. Then the level of P_i is greater than the level of P_i before that Node Expansion Step.*

Proof: Assume that $F_i = \{v\}$ just after a Node Expansion Step. Then $v \neq r$ and, by definition, $b_i = v$ after that Node Expansion Step. Let $F'_i \neq \emptyset$ and b'_i be the local frontier and the base node of P_i , respectively, just before that Node Expansion Step. We show that b'_i is the parent of v . Then $l(b'_i) < l(v) = l(b_i)$ as desired. There are two cases: *Case 1:* $|F'_i| = 1$. In this case, we must have $F'_i = \{u\}$ where u is the parent of v . So $b'_i = u$ by definition. *Case 2:* $|F'_i| > 1$. Then $F'_i = \{v, w\}$ where w is a leaf and was expanded at that Node Expansion Step. So v was the rightmost frontier node in F'_i , and by Proposition 1, a top-node in F'_i . Hence, by definition, b'_i is the parent of v . \square

Proposition 4 *Suppose that $|F_i| > 1$. Let u be the parent of the top-nodes of P_i . Then the level of P_i increases before P_i donates $\lceil \log d \rceil$ times where d is the number of children of u .*

Proof: As $|F_i| > 1$, u is the current base node. Each time P_i donates, the number of top-nodes of P_i , the children of u , is reduced by at least half. So P_i can donate at most $\lceil \log d \rceil - 1$ times before the number of the top-nodes of P_i is reduced to one. Let v be the only remaining top-node of P_i . Consider two cases. *Case 1:* P_i expands v . By Proposition 1(ii), at the time v is expanded, say, at time t , v will be the only frontier node of P_i . Hence, at time t , $b_i = v$ and $l(b_i) > l(u)$. *Case 2:* P_i donates v . Then P_i donates v in the next donation. Afterwards, the level of each frontier node of P_i is larger than $l(v)$ and consequently, $l(b_i) \geq l(v) > l(u)$. \square

We end this section with a proposition that is fundamental to our analysis. A *unit* of work of our algorithm is taken as one of the following three operations: “expand” by which a processor expands a frontier node, “donate” by which a processor donates a node and “receive” by which a processor receives a donation. The *total work* of an algorithm is the total number of work units performed by the algorithm. The *degree* of a tree is the maximum number of children of any internal node of the tree.

Proposition 5 *For any instance of H of degree d , the total work of our algorithm on H is at most $3n\lceil \log d \rceil$, even using the parent-specified method to encode the messages for donation.*

Proof: When the parent-specified method is used in encoding the messages for donation, an expanded node is expanded again when some of its children are donated. By Proposition 2, a node can be donated at most $\lceil \log d \rceil$ times. So any node can be expanded at most $\lceil \log d \rceil$ times. Thus, the total number of “expand” units is at most $n\lceil \log d \rceil$. By Proposition 2, a node can be donated at most $\lceil \log d \rceil$ times, and thus received at most $\lceil \log d \rceil$ times. So the total number of “donate” and “receive” units is at most $2n\lceil \log d \rceil$. Hence, the total work is at most $3n\lceil \log d \rceil$. \square

3.3 Parallel Backtrack Search Algorithms

In this section we present three algorithms. The first is deterministic but requires global control of the computation. The other two are randomized variants of the first algorithm that do not require any global knowledge of the computation.

3.3.1 A Deterministic Algorithm

Our first algorithm is called *Full-Donation Backtrack Search* (FDBS). The strategy of this algorithm is to let as many overloaded processors donate as possible.

Rule for Full-Donation

Choose the pairing set as large as possible.

The above rule does not fully specify the pairing set when the number of overloaded processors is not equal to the number of idle processors. In a fully-connected network, one idle processor is as good as any other idle processor in terms of sharing the work of a overloaded processor. Thus, when there are more idle processors than overloaded ones, an arbitrary maximal pairing set will do. On the other hand, the overloaded processors can distinguish among themselves by various attributes such as level number, size of local frontier, length of local depth-first-traversal path. When there are more overloaded processors than idle processors, one may take these attributes into account for giving preference of donation to certain overloaded processors. But the Rule for Full-Donation does not explore the potential computational advantages in distinguishing overloaded processors by the attributes mentioned.

The following theorem states that the execution time of the Full-Donation Backtrack Search comes close to the inherent lower bound $\max\{n/p, h\}$.

Theorem 1 *Suppose that H is a tree of degree d . Then the execution time of Full-Donation Backtrack Search is at most $\lceil \log d \rceil (3n/p + h)$.*

Proof: A step is *perfect* if every processor does at least one unit of work at that step, i.e., it expands a node at the Node Expansion Step, or donates or receives at the Donation Step; otherwise, the step is *imperfect*. By Proposition 5, the total units of work is at most $3n \lceil \log d \rceil$. Hence, there are at most $3n \lceil \log d \rceil / p$ perfect steps. We show that there are at most $h \lceil \log d \rceil$ imperfect steps.

Let the *search-level* be the minimum level of all busy processors. We show that the search-level increases after at most $\lceil \log d \rceil$ imperfect steps. Since the search-level can increase up to at most h , which is the height of H , there are at most $h \lceil \log d \rceil$ imperfect steps after all nodes of H are expanded. Consider an imperfect step. There must be a processor which was idle at the Node Expansion Step. Then this processor must have not received any donations at the preceding Donation Step.

By the maximality of R , every overloaded processor must have donated at that Donation Step. By Proposition 4, there can be at most $\lceil \log d \rceil$ such imperfect steps before the level of each overloaded processor increases. For a processor that has only one frontier node at a Donation Step, the level of the processor, by Proposition 3, must have increased after the preceding Node Expansion Step.

Therefore, the level of every busy processor increases before there are $\lceil \log d \rceil$ imperfect steps. Consequently, the search-level increases before there are $\lceil \log d \rceil$ imperfect steps. The theorem follows. \square

Remark: The proof of Theorem 1 suggests that, when there are more overloaded processors than idle processors, it might be advantageous in practice to give preference to the overloaded processors of smaller levels, because such a preference tends to increase the search-level more rapidly.

Corollary 2 *If H is a tree of constant degree, then the execution time of Full-Donation Backtrack Search is at most $O(n/p + h)$.*

By Theorem 1, FDBS is within a factor of $O(\lceil \log d \rceil)$ from the inherent lower bound $\max\{n/p, h\}$ on any instance H of degree d . The important special case is that H is of a bounded degree, as in the Example 2 of Section 2.2. In such cases, FDBS is optimal within a constant factor.

Though exhibiting optimality, Full-Donation Backtrack Search requires global control to determine the pairing of overloaded processors with idle processors at the Decision Steps. It turns out that the global control needed in FDBS can be replaced effectively by randomization. The idea of randomization is to achieve a pairing of some overloaded processors and some idle processors through random requests. We present two implementations of this randomization, depending whether it is the overloaded processors that initiate the requests or the idle ones that initiate the requests.

3.3.2 Randomized Algorithm 1: Busy-Initiated

Our first randomized algorithm is called *Busy-Initiated Backtrack Search* (BIBS) in which the overloaded processors initiate the requests for donation. The strategy of BIBS is simple: each overloaded processor initiates a request for donation to a random processor; a busy processor rejects all received requests; an idle processor accepts an arbitrary request from the received requests, if any, and rejects the others;

Decision Step of BIBS

```
/* Requesting Message Step */
for  $i = 1, 2, \dots, p$  in parallel do
  if  $i$  is overloaded then
     $dest(i) \leftarrow$  a random element of  $\{1, 2, \dots, p\}$ ;
    send message " $i$  has work to share" to  $dest(i)$ ;
/* Accepting Message Step */
for  $j = 1, 2, \dots, p$  in parallel do
  if  $j$  is idle then
    let  $D_j = \{i \mid i \text{ has received a message } "i \text{ has work to share}"\}$ ;
    if  $D_j \neq \emptyset$  then
      choose an arbitrary  $k$  from  $D_j$ ;
      send message " $j$  can share your work" to  $k$ .
      /*  $k$  donates to  $j$  in the next Donation Step */
```

Figure 3.3.2

a overloaded processor whose request is accepted by an idle processor donates to that idle processor. Figure 3.3.2 contains code describing the Decision Step of BIBS in which the pairing set is implicitly determined. The corresponding modification in the Donation Step in Figure 3.1 is to replace the line "if $(i, j) \in R$ for some j then" with "if i receives message ' j can share your work' then".

Remarks: Following the previous remark, it may be advantageous in practice to implement Busy-Initiated Backtrack Search with the provision that an idle processor that receives more than one request accepts the request from the requesting processor of the smallest level. The implementation of such a provision requires that the request message contain the level number of the requesting processor and that the receiving processor have a facility to select the request from the minimum-level processor.

The following is our main theorem. It states that, uniformly on all instances, the execution time of BIBS is likely to be within a small factor of the inherent lower bound $\max\{n/p, h\}$, provided that n is sufficiently larger than p . In the case that the degree of H is bounded, the execution time of BIBS is likely to be optimal up to a constant factor.

Theorem 2 *Let the random variable $T(H)$ be the execution time of Busy-Initiated Backtrack Search on H . Let d be the degree of H . Then for any instance H and for any $p \geq 2$,*

$$\Pr[T(H) > 7\lceil \log d \rceil (\frac{n}{p} + h)] < ne^{-\frac{1}{16}n \log d / p^2}.$$

Theorem 2 is an immediate consequence of the following theorem, combined with the fact that H has n nodes.

Theorem 3 *Let the random variable $T(H, w)$ denote the number of steps of Busy-Initiated Backtrack Search on instance H , up to the point when node w is expanded. Let d be the degree of H . Then for every instance H , for every w in H and for any $p \geq 2$,*

$$\Pr[T(H, w) > 7\lceil \log d \rceil (\frac{n}{p} + h)] < e^{-\frac{1}{16}n \log d / p^2}.$$

Proof: Let L_w be the path from r to w . Before w is expanded, there is a unique frontier node s on path L_w . We call the processor possessing s the *special processor*, denoted by S . Let b denote the base node of S . Initially, $b = s = r$. Let x be the level of b when w is expanded. We analyze how many steps occur while the level of b remains less than or equal to x . Before w is expanded, $F_S \neq \emptyset$. In a Decision Step, if $|F_S| = 1$, S does nothing; otherwise, S issues a request to a random processor. By Proposition 3, there can be at most $x \leq h$ steps at which $|F_S| = 1$. We call a step at which $|F_S| > 1$ a *trial* step. We will show that, with the probability indicated in the statement of the theorem, the number of trial steps is no more than $7n\lceil \log d \rceil / p + h\lceil \log d \rceil$. Then the theorem follows.

At a trial step, S attempts to donate by sending a request to a random processor P_k . We call a trial step *successful* if S donates at that step. A trial step is successful if (i) P_k is idle and (ii) P_k accepts the request of S given that P_k is idle. The probability of (i) is δ/p where δ is the number of idle processors. Given (i), (ii) would hold if no other overloaded processors sent requests to P_k . For $p \geq 2$, the probability that S is the only overloaded processor that sends request to P_k is $(1 - 1/p)^{\lambda-1}$ where λ is the number of overloaded processors. Therefore, independent of previous steps,

$$\Pr[\text{a trial step is successful}] \geq \frac{\delta}{p} (1 - \frac{1}{p})^{\lambda-1}. \quad (3.1)$$

We call a trial step *good* if more than $\lceil p/2 \rceil$ processors do at least one unit of work at that step; otherwise, it is *bad*. By Proposition 5, there can be at most $6n\lceil \log d \rceil / p$ good trial steps. To bound the number of bad steps, we consider the

probability that a bad trial step is successful. In a bad step, $\delta \geq p/2 \geq \lambda$. So by (3.1), independent of previous steps,

$$\Pr[\text{a bad trial step is successful}] \geq \frac{1}{2}(1 - \frac{1}{p})^{p/2} \geq \frac{1}{4}, \quad (3.2)$$

where the last inequality is by the facts that $(1 - \frac{1}{x})^x$ increases as x increases for $x > 1$ and that $p \geq 2$.

By Proposition 4, the special processor can donate at most $x \lceil \log d \rceil$ times. So there can be at most $x \lceil \log d \rceil$ successful steps. By (3.2), the probability that a bad trial step is successful is at least $1/4$ independently. Therefore, it is unlikely that there will be too many bad trial steps. More precisely, let $B(t, N, \rho)$ denote the probability that there are less than t successes in N independent Bernolli trials with each trial having probability ρ to be a success. By the fact that there can be at most $x \lceil \log d \rceil \leq h \lceil \log d \rceil$ successful steps and by (3.2),

$$\begin{aligned} & \Pr[\text{more than } \lceil \log d \rceil (n/p + h) \text{ bad trial steps}] \\ & \leq B(h \lceil \log d \rceil, \lceil \log d \rceil (n/p + h), 1/4). \end{aligned} \quad (3.3)$$

By the Chernoff bound [AV79], we have that for $t = (1 - \gamma)N$ where $1 > \gamma > 0$,

$$B(t, N, \rho) = B((1 - \gamma)N, N, \rho) \leq e^{-\frac{1}{2}\gamma^2 \rho N}. \quad (3.4)$$

Set $t = h \lceil \log d \rceil$, $N = \lceil \log d \rceil (n/p + h)$ and $t = (1 - \gamma)N$. Then $\gamma = (N - t)/N$. Hence,

$$\gamma^2 N = \frac{(\frac{n \lceil \log d \rceil}{p})^2}{\lceil \log d \rceil (\frac{n}{p} + h)} = \frac{\lceil \log d \rceil (\frac{n}{p})^2}{(\frac{n}{p} + h)}. \quad (3.5)$$

To bound $\gamma^2 N$ in (3.5) from below, we consider two cases. *Case 1:* $n/p \geq h$. In this case, by (3.5),

$$\gamma^2 N \geq \frac{\lceil \log d \rceil (\frac{n}{p})^2}{\frac{n}{p} + \frac{n}{p}} = \frac{n \lceil \log d \rceil}{2p}.$$

Case 2: $n/p < h$. In this case, by (3.5),

$$\gamma^2 N \geq \frac{\lceil \log d \rceil (\frac{n}{p})^2}{h + h} \geq \frac{n \lceil \log d \rceil}{2p^2},$$

as $n \geq h$. By taking the smaller lower bound in the two cases, we have

$$\gamma^2 N \geq \frac{n \lceil \log d \rceil}{2p^2}. \quad (3.6)$$

By (3.3), (3.4), (3.6) and the fact that there are at most $6n \lceil \log d \rceil / p$ good steps, hence, at most $6n \lceil \log d \rceil / p$ good trial steps,

$$\begin{aligned} & \Pr[\text{there are more than } 7n \lceil \log d \rceil / p + h \lceil \log d \rceil \text{ trial steps}] \\ & \leq \Pr[\text{there are more than } \lceil \log d \rceil (n/p + h) \text{ bad trial steps}] \\ & \leq e^{-\frac{1}{16} n \log d / p^2}. \quad \square \end{aligned}$$

3.3.3 Randomized Algorithm 2: Idle-Initiated

Our second randomized algorithm is called *Idle-Initiated Backtrack Search* (IIBS) in which the idle processors initiate the requests for donation. IIBS is a reversal of BIBS: each idle processor initiates a request for work to a random processor; a processor that is not overloaded rejects any received requests; a overloaded processor accepts an arbitrary request from the received requests, if any, and rejects the others; the idle processor whose request is accepted by some overloaded processor receives a donation from that overloaded processor.

The major advantage of IIBS over BIBS is efficiency. In IIBS, busy processors need not interrupt their computation to seek idle processors to share their work whenever they have more than one frontier node; instead, the idle processors, having nothing to do, can spend all their time looking for new work.

Figure 3.3.3 contains the code describing the Decision Step of BIBS. The corresponding modification in the Donation Step in Figure 3.1 is to replace the line “if $(i, j) \in R$ for some j then” with “if i has selected j in the Accepting Message Step then”.

The proof of the following theorem is similar to that of Theorem 2. It states that, uniformly on all instances, the execution time of Idle-Initiated Backtrack Search is likely to be within a small factor of the inherent lower bound $\max\{n/p, h\}$, provided that n is sufficiently larger than p . In case that H is of bounded degree, the execution time of IIBS is likely to be optimal up to a constant factor.

Theorem 4 *Let the random variable $T(H)$ be the execution time of IIBS on H . Let d be the degree of H . Then for any instance H and for any $p \geq 2$,*

$$\Pr[T(H) > 7 \lceil \log d \rceil \left(\frac{n}{p} + h \right)] < ne^{-\frac{1}{16} n \log d / p^2}.$$

Proof: The proof of Theorem 4 is similar to that of Theorem 2, except for a difference in the lower bound on the probability that a bad trial step is successful. In

Decision Step of IIBS

```
/* Requesting Message Step */
for  $j = 1, 2, \dots, p$  in parallel do
  if  $j$  is idle then
     $dest(j) \leftarrow$  a random element of  $\{1, 2, \dots, p\}$ ;
    send message " $j$  wants new work" to  $dest(j)$ ;
/* Accepting Message Step */
for  $i = 1, 2, \dots, p$  in parallel do
  if  $i$  is overloaded then
    let  $A_i = \{j \mid i \text{ has received a message } "j \text{ wants new work}"\}$ ;
    if  $A_i \neq \emptyset$  then
      select any  $k \in A_i$ ;
      send message " $i$  can give you new work" to  $k$ .
      /*  $i$  donates to  $k$  in the next Donation Step */
```

Figure 3.3.3

IIBS, a trial step of IIBS is a step in which the special processor is overloaded; a trial step is successful if any idle processor requests work from the special processor. Let k be the number of idle processors. In a bad step, $k \geq p/2$. Hence, the probability that a bad trial step is successful is the probability that some idle processor randomly "hits" the special processor, which is $1 - (1 - 1/p)^k > 1 - (1 - 1/p)^{p/2} > 1 - e^{-1/2} > 2/5$ where the second last inequality is by $1 + x < e^x$ for $x \neq 0$. Consequently, the corresponding constant in the exponent of the probability bound is $1/10$. \square

3.3.4 PRAM Implementation

Both BIBS and IIBS can be directly implemented on various PRAM models. Consider the ARBITRARY model, a weak Concurrent-Write PRAM model. In an ARBITRARY PRAM, when two or more processors write to the same cell simultaneously, an arbitrary processor succeeds in writing its value into the cell. To implement BIBS in the ARBITRARY PRAM, each processor has a designated cell in the shared memory. To send a request, an overloaded processor writes its processor-id into the designated cell of a random processor; an overloaded proces-

sor that succeeds in writing its value into the designated cell of an idle processor donates to that idle processor. One can implement IIBS in a similar way.

Other weak PRAM models can also be used. One such a model is the COLLISION model in which, when two or more processors write to the same cell simultaneously, a special value, the “collision symbol”, appears in that cell. To implement our algorithms using a COLLISION PRAM, a request may be accepted if no other requests were sent to the same processor. The proof of Theorem 3 actually works for the implementation of BIBS in the COLLISION model. The same proof works for the implementation of IIBS in the COLLISION model.

In the PRAM model equipped with a minimum-value-selection mechanism, the minimum value appears in a cell when there are concurrent writes into that cell. With such a PRAM model, BIBS can be implemented with more flexibility. The minimum-value-selection mechanism enables an idle processor to accept the donation from the overloaded processor of minimum level, or with the largest number of frontier nodes, or with a longest DFS path.

The shared memory of a PRAM allow more complex donation schemes than we have used. A good example is the donation scheme described in [Ma86] which is primarily designed for the shared-memory machines. This scheme is identical to ours except that a processor donates a constant portion, about a half, of all its frontier nodes, not just its top-nodes. Because each donation involves a large number of nodes, this method tends to minimize the number of donations. The shared memory is used to store all the frontier nodes. A shared-memory data structure is proposed to implement donations efficiently. The local frontier nodes of each processor are organized into a balanced binary tree. A donation entails splitting the binary tree at the root into two subtrees and giving away one subtree. The transfer of a subtree can be accomplished by giving the receiving processor a “pointer” to the segment of the shared memory where the subtree of the donated nodes resides instead of moving around the subtree itself. In this way, donations are managed efficiently. In a message-passing system, all the donated nodes in one donation must be fully specified by messages. The set of nodes in one donation may not necessarily be closely related to each other in a way to permit a succinct description. Hence, long messages may be needed in encoding donations in a message-passing system.

3.4 Open Problems and Further Research

We discuss some open problems and further research related to the parallel backtrack algorithms presented in this chapter.

1. Our results are proved under the strong assumption that the network is fully-connected. The algorithms can be implemented on a sparsely interconnected network such as a hypercube or butterfly network by routing the messages, but some of their advantage is lost in that case, since the message-routing delay will grow with the diameter of the network. One approach to avoiding that message-routing delay would be to modify the algorithms so that, whenever a processor performs a random selection, it randomly selects one of its neighboring processors instead of a random processor from the entire network. The effectiveness of these modified algorithms will depend critically on the interconnection structure. The main open problem is to analyze these modified algorithms in the hypercube, the butterfly or other interesting sparsely interconnected networks.
2. When the degree d of H is not bounded, the proven upper bounds on the execution time of our algorithms are off by a factor of $O(\lceil \log d \rceil)$ from optimal. It would be desirable to get rid of the $O(\lceil \log d \rceil)$ factor, perhaps, by using more complex schemes for donation. One possibility is to allow a processor to donate to two or more idle processors at one time.
3. As we have remarked, some priority rules may be adopted in implementing Full-Donation Backtrack Search and Busy-Initiated Backtrack Search. It would be interesting to conduct simulations to compare different priority rules for donation on some real applications.

Chapter 4

Branch-and-Bound Procedure

4.1 Introduction

Branch-and-bound algorithms are the most frequently used method in practice for the solution of combinatorial optimization problems [LW66,PS82,Ba85]. In this section we give a general discussion of this method.

The fundamental ingredients of a branch-and-bound method are a *branching procedure* and a *bounding procedure*. The branching procedure takes a given combinatorial optimization problem A and either solves it directly or derives from it a set of subproblems, A_1, A_2, \dots, A_d , such that an optimal solution to problem A can be found by solving each of A_1, A_2, \dots, A_d and then, among these d solutions, taking the one of least cost. The bounding procedure computes a lower bound on the cost of an optimal solution to a given problem or subproblem A ; such lower bounds can be used to guide the order in which subproblems are solved, or to determine that certain subproblems need not be considered at all. A branch-and-bound computation can be viewed as a search through a tree of subproblems, in which the original problem occurs at the root, and the children of a given subproblem are those subproblems obtained from it by branching. A leaf of the tree corresponds to a subproblem that can be solved directly by the branching procedure. The object of the search is to find a leaf of minimum cost. One can view the primitive step as the expansion of a given node of the tree to produce its children and their cost bounds.

The lower bounds computed by the bounding procedure possess the monotonicity property: if subproblem A_i is derivable from subproblem A by the branching

procedure, the lower bound on A_i is at least as large as the lower bound on A . This property of the lower bounds allows the algorithm to rule out certain subproblems from further consideration and to determine when it has found the solution of the least cost. Suppose that the algorithm have found a solution of cost c . By the monotonicity of the bower bounds, any subproblem A with a lower bound greater than or equal to c cannot give a solution of cost less than c , thus can be discarded from further consideration. To terminate, the algorithm must ensure that all the remaining subproblems have lower bounds as least as large as the cost of some found solution, and if so, return one solution of least cost among the found solutions.

In addition to the branching and bounding procedures, a *selection rule* is needed in deciding the order by which subproblems are branched on. In view of a tree search, such a rule determines the order in which the nodes of the tree are explored and the portion of the tree that will be generated. The “best-first” rule is to branch on the subproblem of least lower bound. An algorithm using the best-first rule may return the first solution it finds, which must be a solution of minimum cost. The best-first rule tends to minimize the number of subproblems that are branched. The set of subproblems present at any given time can be kept in a priority queue so that the subproblem of least lower bound can be easily found. The size of this queue may be large at times. On the other hand, the “depth-first” rule is to branch on a latest generated subproblem. This rule tends to minimize the memory required in executing the algorithm, but may explore subproblems that would not render the least-cost solutions. One may combine these rules into some mixed rule so that a trade-off between time and space can be achieved. A recent study on the time-space trade-off of the sequential branch-and-bound computation can be found in [KSW86].

We are interested in executing branch-and-bound method in parallel. The fundamental problem in a parallel branch-and-bound computation is to allocate the subproblems to the processors so that they can all be performing useful work. Unlike the parallel backtrack search algorithms discussed in the previous chapter, a parallel branch-and-bound algorithm may not achieve an effective speed-up by merely achieving a good “load-balancing”. A successful solution must ensure that processors will not spend much time explore useless subproblems. Furthermore, the overhead for interprocessor communication must not be excessive.

4.2 An Example

An instance of the *Traveling Salesman Problem* (TSP) is a complete directed graph on n nodes in which each directed edge (i, j) is associated with a real number $c_{i,j}$ called the cost of edge (i, j) . A tour is a traversal of the graph in which each node of the graph appears exactly once. The cost of a tour is the sum of the costs of the edges in the tour. A solution to the TSP is a tour of least cost.

We shall describe a branch-and-bound procedure for solving the TSP. The bounding procedure is a procedure that solves the Assignment Problem (AP). The problem instance of AP is the same as that of TSP. The objective of AP is to find a permutation σ on the set $\{1, 2, \dots, n\}$ such that $c(\sigma) = \sum c_{i,\sigma(i)}$ is minimized. AP can be solved by the Hungarian method in at most $O(n^3)$ steps [PS82]. A tour induces a permutation on $\{1, 2, \dots, n\}$ by mapping i to j if (i, j) is an edge of the tour. Hence the cost of a solution to the associated AP is no larger than the cost of a minimum-cost tour. To bound a problem or subproblem A of TSP, we solve the AP associated with A . If the solution σ to the AP has only one cycle, then return σ as the solution to A ; otherwise, A is not solved and $c(\sigma)$ is returned as a lower bound on the solution to A .

The branching procedure is based on the following fact: if a cycle contains less than n edges, then not all the edges on the cycle can be in the same tour. Suppose that the solution σ to the AP associated with a problem or subproblem A has more than one cycle. Let D be a smallest cycle in σ and e_1, e_2, \dots, e_d be the edges of D . To branch on A , we create d subproblems A_1, A_2, \dots, A_d by “breaking” cycle D as follows: for $1 \leq i \leq d$, A_i is derived from A by deleting edge e_i (by setting the cost of e_i to $+\infty$). Since not all edges of cycle D can be in the same tour, any tour of A must be also a tour in *some* A_i , $1 \leq i \leq d$. So a solution to A can be found by solving subproblems A_i , $1 \leq i \leq d$, and among the d solutions, taking the one of least cost. The lower bounds obey the monotonicity property: deleting an edge can only increase the cost of a minimum-cost permutation.

4.3 Generic Branch-and-Bound Algorithms

In this section we give an abstract generic description of the branch-and-bound algorithms we consider.

Let H be a rooted tree in which each node has a finite number of children. Let

V be the set of nodes of H . Associated with each node $v \in V$ is a cost $c(v)$ such that

- (i) if $v \neq w$ then $c(v) \neq c(w)$;
- (ii) if w is a child of v then $c(v) < c(w)$.

Tree H is intended to model the tree of subproblems associated with a branch-and-bound algorithm on a given problem instance. The nodes of H correspond to the subproblems that can be generated by the branching procedure on the given problem instance. The children of node v correspond to the subproblem that can be derived by the branching on the subproblem represented by v . For $v \in V$ that is not a leaf, $c(v)$ is the lower bound computed by the bounding procedure of the algorithm on the subproblem represented by v ; for a leaf v , $c(v)$ is the cost of a solution to the subproblem represented by v . The monotonicity of the lower bounds is reflected by (ii). Moreover, the lower bounds on different subproblems are required by (i) to be distinct. The requirement of (i) can be too restrictive in application. For instance, a branch-and-bound computation for determining the chromatic number of a graph may produce many subproblems of the same lower bound. The purpose of (i) is for technical convenience. Later in the paper we will discuss the consequences if we allow equal lower bounds.

We consider algorithms whose objective is to generate the leaf of least cost in H , using the *node expansion* operation. When this operation is applied to a node v it either determines that v is a leaf or produces the children of v and makes their values known to the algorithm. A node can be expanded only if it is the root of H or if it is a child of some node previously expanded. This model is similar to one introduced for a different purpose in [KSW86].

Figure 4.3 is a generic description of the kind of algorithm we consider. For any set $S \subseteq V$ let $\Gamma(S)$ denote the children of the nodes in S . The *frontier*, denoted by variable F , is the set of nodes that have been generated but not expanded, and the variable B denotes the minimum cost of any expanded leaf. We think of the nodes in S in Figure 4.3 as being expanded simultaneously; thus the execution time of the algorithm is defined to be the number of executions of the body of the while loop.

There is an inherent lower bound on the execution time of the algorithms described. For a given problem instance (H, c) let v^* be the leaf of minimum cost in H . Let \tilde{H} be the subtree determined by the nodes in H of cost less than or equal to

Generic Node Expansion Algorithm

```
 $F \leftarrow \{r\};$   
 $B \leftarrow \infty;$   
while  $F \neq \emptyset$  do  
    select a set of nodes  $S \subseteq F$ ;  
    expand the nodes in  $S$ ;  
     $F \leftarrow \{F \setminus S\} \cup \Gamma(S)$ ;  
     $B \leftarrow \min(\{B\} \cup \{c(v) : v \in S \text{ and } v \text{ is a leaf}\});$   
     $F \leftarrow \{v \in F : c(v) \leq B\}.$ 
```

Figure 4.3

v^* . Every node expansion algorithm that determines the minimum-cost leaf must expand every node of \widetilde{H} . This can be seen as follows. Suppose that the algorithm did not expand $u \in \widetilde{H}$. Let H' be the tree that differs from H in having a leaf w in the subtree rooted at u such that $c(u) < c(w) < c(v^*)$. On input H' , the algorithm would still identify v^* as the least-cost leaf of H' . But this is wrong, since the cost of w is less than the cost of v^* in H' . Hence, a correct algorithm must expand every node of \widetilde{H} . Let n be the number of nodes in \widetilde{H} , and let h be the number of nodes in a longest root-leaf path in \widetilde{H} . Then the execution time of any algorithm is at least h , since the nodes along a path must be expanded one at a time, and the execution time of any p -processor algorithm is at least n/p , since n nodes must be expanded and at most p nodes can be expanded at a single step. Thus $\max\{n/p, h\}$ is an inherent lower bound on the execution time of any p -processor algorithm on the instance (H, c) . Our goal is to design the p -processor algorithms whose execution time would come close to the lower bound $\max\{n/p, h\}$.

4.4 Previous Work on Parallel Branch-and-Bound

In this section we survey some previous work on parallel branch-and-bound. Most were reported in the last five years.

Lai and Sahni [LS84] studied the anomaly that a parallel algorithm may achieve a super-linear speed-up over its sequential counterpart. This type of anomaly occurs when a high-quality solution found late by the sequential algorithm is found

early in the parallel algorithm. Li and Wah [Li85] suggest an execution order of a parallel algorithm that tend to maximize the number of super-linear speed-up anomaly. Though having some theoretical interests, the study of anomaly does not offer direct insights to the effective parallelization of branch-and-bound computation.

A number of parallel implementations of branch-and-bound computation have been proposed. In the days the computer memory was still expensive, Imai *et al* [IYF79] proposed an implementation of parallel depth-first search in which k processors explore the k most recently generated subproblems where the set of subproblems generated are kept in a shared memory for global access and updates. Wah and Ma [WM84] proposed an architecture for implementing global-best-first search in which k processors explore the k best active subproblems. Further discussion on the global-best-first search can be found in [WLY85]. They found that the selection of the globally best subproblems represents significant system overhead. Kumar and Kanai [KK84] suggested the scheme in which each processor performs the same branch-and-bound search but with a different pre-set lower bound to started with. The processor started with a high pre-set lower bound may finish fast, and if it ever finds any solution, that solution must be an optimal solution. In 1985, a simulation tool for performance evaluation of parallel branch-and-bound algorithms was developed at the University of Colorado at Boulder in an attempt to understand the significance of various parameters involved in a parallel branch-and-bound computation through experimental means. A report of this research can be found in [BRT87]. More recently, Kumar *et al* [KRR88] considered several parallel formulations of the best-first branch-and-bound algorithm based on different data structures.

There are some recent efforts on achieving effective speed-up. Grandine [Gra87] implemented the branch-and-bound algorithm given as the example in Section 4.2 on a 18-processor shared-memory Sequent, using the global best-first strategy, and reported speed-up about 6 on large input instances. Vornberger [Vo87a] implemented a sequential branch-and-bound algorithm for solving the Vertex-Cover Problem on a network of 8 computers and reported a nearly perfect speed-up. In their paper of DIB [FM87], Finkel and Manber also reported a good speed-up achieved by DIB in parallelizing a branch-and-bound algorithm for solving the TSP.

Chapter 5

A Randomized Parallel Branch-and-Bound Procedure

In this chapter we present and analyse a universal randomized method called Local Best-First Search for parallelizing sequential branch-and-bound algorithms. The method is a parallel implementation of the “best-first” search strategy, and it uses no global data structures or complex communication protocols. The computation alternates between node expansion steps, in which each processor expands the cheapest node in its possession, and the node distribution steps, in which the children of the nodes that were expanded in the previous node expansion step are sent to random processors. Our main result is that, with high probability, the execution time of Local Best-First Search is unlikely to exceed the inherent lower bound.

5.1 Local Best-First Search

Among algorithms that expand at most p nodes per step, the following *best-first rule* for selecting the set S at each step is a direct extension of sequential “best-first” search.

Best-First Rule

if $|F| \leq p$ then $S = F$
else S consists of the p nodes in F of least cost

The algorithm that implements this rule will be called *Global Best-First Search*. One can easily show that the execution time of Global Best-First Search comes close

to the inherent lower bound.

Proposition 6 *The execution time of Global Best-First Search is at most $n/p + h$.*

Proof: Let w be any node in \widetilde{H} and $P(w)$ be the path from the root of H to w . Let v be the node on $P(w)$ that is currently in the frontier F . Consider the next node expansion step. If v does not get expanded next, then by the best-first rule, all p nodes expanded in this step are in \widetilde{H} . There can be at most n/p such steps. If v gets expanded next, the child of v on $P(w)$ will be in F next. There can be at most h such steps. So w will be expanded within $n/p + h$ steps. Since w is an arbitrary node in \widetilde{H} , all the nodes in \widetilde{H} will be expanded within $n/p + h$ steps. \square

However, in order to implement Global Best-First Search, it seems necessary to keep the set F in a global priority queue so that, at each step, the p nodes of least cost in F can be selected, assigned in one-to-one fashion to the p processors, and distributed to their assigned processors. The implementation of these selection and distribution operations using messages is costly; to avoid this overhead, we propose a more local algorithm, called *Local Best-First Search*, which uses no shared data structures. Instead, the unexpanded nodes are distributed among the processors, with each unexpanded node belonging to exactly one processor. The computation alternates between *node expansion steps*, in which each processor expands the cheapest node in its possession, and *node distribution steps*, in which the children of the nodes just expanded are sent to random processors. More precisely, processor i maintains a set of nodes F_i , its *local frontier*, and a cost bound B_i , which is certified to be the cost of some leaf. F_i is the set of nodes of cost less than or equal to B_i which have been received from other processors but not yet expanded. At each step every processor i does one of two things:

- (i) if F_i is not empty then it expands the node of minimum cost in F_i and sends its children to processors chosen at random;
- (ii) if F_i is empty then it sends the message "there is a leaf of cost B_i " to a processor chosen at random.

The processors then update the sets F_i and bounds B_i on the basis of the messages they have received. The computation continues until all sets F_i are empty: at that point the minimum cost of a leaf is given by $\min(\{B_i, i = 1, 2, \dots, p\})$. The execution

Local Best-First Search

```
/* Initialization */
 $F_1 = \{r\};$ 
for  $i = 2, 3, \dots, p$ ,  $F_i \leftarrow \emptyset;$ 
for  $i = 1, 2, \dots, p$ ,  $B_i \leftarrow \infty;$ 

while some set  $F_i \neq \emptyset$  do
    /* Node Expansion Step */
    for  $i = 1, 2, \dots, p$  in parallel do
        if  $F_i \neq \emptyset$  then
            let  $v_i$  be the node of least cost in  $F_i;$ 
            expand  $v_i;$ 
             $F_i \leftarrow F_i \setminus \{v_i\};$ 
            if  $v_i$  is a leaf then  $B_i \leftarrow c(v_i)$ 
            else
                for each child  $w$  of  $v_i$  do
                     $dest(w) \leftarrow$  a random element of  $\{1, 2, \dots, p\};$ 
                    send  $w$  to  $dest(w)$ 
            else
                send "there is a leaf of cost  $B_i$ " to some designated destinations;

    /* Message Arrival Step */
    for  $i = 1, 2, \dots, p$  in parallel do
         $F_i \leftarrow F_i \cup \{w : dest(w) = i\};$ 
    for  $i = 1, 2, \dots, p$  in parallel do
         $B_i \leftarrow \min(B_i \cup \{x : i \text{ has received a}$ 
            message "there is a leaf of cost  $x$ "  $\});$ 
    for  $i = 1, 2, \dots, p$  in parallel do
         $F_i \leftarrow \{v \in F_i : c(v) \leq B_i\}.$ 
```

Figure 5.1

time of the algorithm is the number of node expansion steps it performs. The code of Local Best-First Search is contained in Figure 5.1. The code of Figure 5.1 does not specify the designated destinations for a processor to send its message when its local frontier is empty; and it also omits the details of how messages are used to notify all processors of the minimum cost and turn the computation off. We will deal with this in section 5.4.

On each fixed problem instance (H, c) the execution time of the randomized algorithm Local Best-First Search is a random variable. We will prove that there exists a universal constant d such that, for every instance (H, c) the following holds with high probability: the ratio between the execution time of the Local Best-First Search and the minimum possible execution time of any p -processor algorithm is less than d . Thus Local Best-First Search is a universal method of executing branch-and-bound algorithms efficiently in parallel without shared data structures.

The following is the main theorem of the paper. It states that, uniformly on all instances, the probability that the execution time of our method is within a constant factor of the inherent lower bound $\max\{n/p, h\}$ converges rapidly to 1 as n tends to infinity.

Theorem 5 (Main Theorem) *There exist positive constants α, β, γ and d such that the following holds for every instance (H, c) : Let n be the number of nodes in \widetilde{H} and let h be the maximum number of nodes in a root-leaf path of \widetilde{H} . Let the random variable $T(H, c)$ denote the execution time of Local Best-First Search on the instance (H, c) . Suppose that $n \geq p^3$, then*

$$\Pr[T(H, c) > d(\frac{n}{p} + h)] < \gamma p n^2 \exp(-\frac{\beta}{p} n^\alpha).$$

Theorem 5 is an immediate consequence of the following theorem, combined with the fact that \widetilde{H} has at most n nodes.

Theorem 6 *For every instance (H, c) and for every node w in \widetilde{H} , let the random variable $T_w(H, c)$ denote the number of steps of Local Best-First Search on instance (H, c) , up to the point when node w is expanded. Suppose that $n \geq p^3$, then*

$$\Pr[T_w(H, c) > d(\frac{n}{p} + h)] < \gamma p n \exp(-\frac{\beta}{p} n^\alpha),$$

where α, β, γ and d are the constants stated in Theorem 5.

5.2 Proof of Theorem 6

In this section we give a proof of Theorem 6. The proof given here is the complete version of the proof given in [KZ88]. Though lengthy and somewhat complicated, the proof contains several ingredients that appear to be of considerably mathematical interest in their own right. Recently, A. Ranade [Ra89] had found a clever argument that greatly simplifies the proof given here. However, for the purpose of the thesis, only the original proof is given here.

In analyzing the time required by Local Best-First Search to expand the nodes in \widetilde{H} we can disregard all nodes of H that do not lie in \widetilde{H} , since no processor will ever choose to expand such a node when it has a node of \widetilde{H} available. We divide the nodes of \widetilde{H} into two types: *special nodes*, which lie on the path from the root to w , and *regular nodes*, which do not lie on that path. At each step, there is exactly one unexpanded special node s present, and we concentrate on the processor that owns s . We can distinguish among three possible actions by that processor:

- (i) The processor expands s .
- (ii) The processor expands a node that was generated after s was generated; such a step is called a *post-delay*.
- (iii) The processor expands a node that was generated earlier than s was generated, or at the same time. Such a step is called a *pre-delay*.

Action (i) can occur at most h times, since there are at most h special nodes. A node can cause a post-delay only if, at the time it is generated, it is sent to the unique processor containing a special node. Since the destinations of newly generated nodes are drawn independently at random from $\{1, 2, \dots, p\}$, the probability distribution of the number of nodes capable of causing a post-delay is stochastically no greater than the number of successes in a Bernoulli process with n trials, where the probability of success at each trial is $1/p$. By the Chernoff bound [AV79], the chance that the number of post-delays is greater than $2n/p$ is at most $e^{-n/3p}$.

Thus the crux of the proof of Theorem 6 lies in bounding the number of pre-delays. To approach this bound we view the computation as a queueing process.

5.2.1 A Queueing Process

To describe the execution of Local Best-First Search as a queueing process, we regard each processor as a server and each node in \widetilde{H} as a customer. Customers corresponding to special nodes are called *special customers*, and those corresponding to regular nodes are called *regular customers*. Associated with each customer is a number called his *cost*. Initially, queue 1 contains one special customer, and queues 2, 3, ..., p are empty. The queueing process alternates between *service steps* and *arrival steps*. At the beginning of each service step there is exactly one special customer in the system, and the queue containing that customer is called the *special queue*. At each service step, the customer of least cost in each nonempty queue is served. At each arrival step, a sequence of customers arrives at the queues. If a special customer was served at the preceding service step then exactly one of the arriving customers is special; otherwise, all the arriving customers are regular. The total number of customers arriving during the entire process is n , and at most h of these are special. The *cost* of the process is the number of service steps in which a customer is served who is in the special queue and arrived there before the current special customer did.

The relationship between the queueing process and the execution of the algorithm is apparent. The service steps correspond to node expansion steps in the algorithm, and the arrival steps correspond to message arrival steps in the algorithm. The cost corresponds to the number of pre-delays.

Define a *destination sequence* as an infinite sequence a_1, a_2, \dots of elements from $\{1, 2, \dots, p\}$. The intended meaning is that a_k is the destination of the k th node of \widetilde{H} to be generated (to obtain a well-defined ordering we use the convention that, among the nodes generated simultaneously at a single node expansion step, the special node, if any, is placed last, and the regular nodes are arranged in order of increasing cost). It should be clear that the following data completely determines a run of the queueing process: an n -node tree \widetilde{H} in which no root-leaf path is of length greater than h , a function c assigning a cost $c(u)$ to each node u in \widetilde{H} , a node w in \widetilde{H} (w is the end point of the path of special nodes) and a destination sequence. Since the algorithm chooses the destinations of the generated nodes independently at random, it is appropriate to assume that the components of the destination sequence are drawn independently from the uniform distribution over $\{1, 2, \dots, p\}$. We may think of \widetilde{H} , c and w as being chosen by an *adversary* whose goal is to

maximize the probability that the cost of the queueing process exceeds $d(n/p + h)$ for some suitable constant d .

We now modify the rules of the queueing process in favor of the adversary. At each step, the adversary chooses one of five types of *events*: the arrival of a special customer, the arrival of a regular customer, and three types of service events, depending on who in the special queue gets served: the special customer, a regular customer who arrived before the special customer did, or a regular customer who arrived after the special customer did. The two types of arrival events are denoted S and R (for special and regular), and the three types of service events are denoted s , pre and $post$ (the service of the special customer, a pre-delay or a post-delay). When an S -event or an R -event occurs, an element is drawn from the uniform distribution over $\{1, 2, \dots, p\}$ to determine the queue at which the arrival will take place. When a service event occurs one customer from each nonempty queue is served, with the type of the event determining whether the customer served from the special queue is the special customer, a customer who arrived before the special customer, or a customer who arrived after the special customer.

In selecting each event the adversary knows the random choices made at all earlier events, and is thus able to calculate which customers reside in each queue. The adversary is constrained by the following rules:

- (a) The first event is an S -event.
- (b) S -events and s -events must alternate;
- (c) the number of R -events and S -events is n , and at most h of these are S events;
- (d) a pre -event can occur only if the special queue contains a regular customer who arrived before the last S -event;
- (e) a $post$ -event can occur only if the special queue contains a regular customer who arrived after the last S -event.

In the modified queueing process the adversary preserves the ability to simulate an instance given by the triple (\widetilde{H}, c, w) ; i.e., he has the freedom to specify the events as they would occur for that instance, given the destinations of the successive arrivals, so that the number of pre -events is the number of pre-delays in the instance (\widetilde{H}, c, w) .

The adversary *succeeds* if the number of pre-events exceeds $d(n/p + h)$. Thus we can prove Theorem 6 by showing that the adversary's chance of success in the modified queueing process is exponentially small.

We will show that the adversary has an optimal strategy in which the following rules are respected:

Rule 1 *Always serve the regular customers in the special queue before the special customer;*

Rule 2 *Schedule no arrivals when the special customer is present.*

Schedules respecting these two rules are completely described by the sequence of arrival events. Service always occurs just after an S -event, and continues until all customers in the special queue get served, with the special customer being served last, and no arrivals would occur during the period of these services. Post-delays never occur, and the cost of the process is simply the number of regular customers who are in the special queue when they are served. Equivalently, the cost is the sum of the lengths of the queues at which the special customers arrive.

Recall that a destination sequence $A = a_1, a_2, \dots$ is a sequence of elements of $\{1, 2, \dots, p\}$; a_i gives the number of the queue at which the i th arrival occurs. A strategy S for the adversary is a rule that determines the sequence of events as a function of the arrival sequence. The rule must respect causality: i.e., the sequence of events that occur before the k th arrival must be determined by a_1, a_2, \dots, a_{k-1} . Any strategy S determines which of the arrivals are regular and which are special. This information can be encoded as an *arrival pattern* $B = b_1, b_2, \dots$, where $b_i = 1$ if the i th customer to arrive is special, and $b_i = 0$ if the i th customer to arrive is regular. Given the destination sequence A and the arrival pattern B , the behavior of a strategy respecting Rules 1 and 2 is completely determined. The proof that the strategy respecting Rules 1 and 2 is optimal is a consequence of the following proposition.

Proposition 7 *For each n and each choice of the destination sequence A and the arrival pattern B of n arrivals, the unique sequence of events consistent with A , B and Rules 1 and 2 yields at least as large a number of pre-events as any sequence of events consistent with A and B .*

Proof: Fix a choice of destination sequence A and arrival pattern B of n arrivals. Let $L = L(A, B)$ be the unique sequence of events consistent with A and B and Rules 1 and 2. Let \tilde{L} be any sequence of events consistent with A and B . Let Q and \tilde{Q} be the set of queues associated with L and \tilde{L} , respectively.

Let k be the number of special customers among the n arrivals and let s_i denote the i th special customer. Let d_i and \tilde{d}_i be the number of the pre-delays to s_i that occur in L and \tilde{L} , respectively. We want to show that

$$\sum_{i=1}^k d_i \geq \sum_{i=1}^k \tilde{d}_i. \quad (5.1)$$

The *state* of a set of p queues is denoted by $\langle q(1), q(2), \dots, q(p) \rangle$ where $q(j)$ is the length of the j th queue. Let $Q_i = \langle q_i(1), \dots, q_i(p) \rangle$ and $\tilde{Q}_i = \langle \tilde{q}_i(1), \dots, \tilde{q}_i(p) \rangle$ denote the states of Q and \tilde{Q} , respectively, upon the arrival of s_i . Set

$$\alpha_i = \max_{1 \leq j \leq p} \{ \tilde{q}_i(j) - q_i(j) \}.$$

Notice that $\alpha_1 = 0$. We will show that

$$\tilde{d}_k \leq d_k + \alpha_k \quad (5.2)$$

and for $i = 1, 2, \dots, k-1$,

$$\alpha_{i+1} \leq \alpha_i - \tilde{d}_i + d_i. \quad (5.3)$$

With (5.2) and (5.3) one can easily show (5.1) by starting with (5.2) and then repeatedly using (5.3).

We first show that for $i = 1, 2, \dots, k$,

$$\tilde{d}_i \leq d_i + \alpha_i, \quad (5.4)$$

which includes (5.2) as a special case.

Let us fix i and assume that s_i arrives at the 1st queue. Then, by Rule 1, $d_i = q_i(1)$. Since there can be at most $\tilde{q}_i(1)$ pre-delays to s_i in \tilde{Q} , $\tilde{d}_i \leq \tilde{q}_i(1)$. But $\tilde{q}_i(1) - q_i(1) \leq \alpha_i$, so $\tilde{d}_i \leq \tilde{q}_i(1) \leq q_i(1) + \alpha_i = d_i + \alpha_i$, which proves (5.4).

To prove (5.3), let $a_i(j)$ be the number of arrivals to the j th queue between s_i and s_{i+1} exclusively. $a_i(j)$ is determined by A and B . Then, by Rules 1 and 2,

$$q_{i+1}(j) = \max\{0, q_i(j) - d_i\} + a_i(j)$$

and

$$\tilde{q}_{i+1}(j) = \max\{0, \tilde{q}_i(j) + a_i(j) - \tilde{d}_i\},$$

for $j = 1, 2, \dots, p$. Since $\max\{a, b\} - c = \max\{a - c, b - c\}$,

$$\begin{aligned}\tilde{q}_{i+1}(j) - q_{i+1}(j) &= \max\{0 - q_{i+1}(j), \tilde{q}_{i+1}(j) + a_i(j) - \tilde{d}_i - q_{i+1}(j)\} \\ &\leq \max\{0, \tilde{q}_i(j) - \tilde{d}_i - q_i(j) + d_i\} \\ &\leq \max\{0, \alpha_i - \tilde{d}_i + d_i\} \\ &\leq \alpha_i - \tilde{d}_i + d_i,\end{aligned}$$

where the last inequality is by (5.4). Hence

$$\alpha_{i+1} = \max_{1 \leq j \leq p} \{\tilde{q}_{i+1}(j) - q_{i+1}(j)\} \leq \alpha_i - \tilde{d}_i + d_i,$$

which proves (5.3) and completes the proof. \square

5.2.2 A Continuous-Time Model

The imposition of Rules 1 and 2 simplifies the queueing process and enables us to give the following clean description of it. A sequence of n customers arrives at a system of p queues. When each customer arrives he is assigned to a random queue. An adversary who observes the arrivals decides, after each arrival, whether to trigger a *service phase*. The total number of service phases is at most h . When a service phase is triggered a random queue is chosen. If the queue contains m customers then the adversary receives a payoff of m and $m + 1$ service events occur; at each service event one customer in each nonempty queue is served and deleted from his queue. No arrivals occur during a service phase. The adversary's goal is to maximize the probability that his total payoff exceeds $d(n/p + h)$, where d is independent of p , n and h . We wish to prove that his probability of achieving this goal goes exponentially to zero as n tends to infinity.

To facilitate the analysis, we embed this process in continuous time by assuming that customers arrive according to a Poisson process with rate $\frac{p}{2}$ over the time period $[0, \lceil 4n/p \rceil]$. At each arrival, the queue where the customer arrives is drawn from the uniform distribution over $\{1, 2, \dots, p\}$. Thus, by a basic property of the Poisson process, the arrival processes for the p queues are mutually independent, and each is Poisson with rate $\frac{1}{2}$. When a service phase is triggered, the service events occur immediately, with no lapse of time. It should be clear that letting customers arrive according to a Poisson process has no effect on the payoff received by the adversary in the course of the first n arrivals. Moreover, by the following Proposition 8(i),

the probability that the number of arrivals over the time period $[0, \lceil 4n/p \rceil]$ is less than n goes exponentially to zero as n tends infinity. Thus it suffices to prove that, in this continuous-time set-up, the adversary has an exponentially small chance of achieving a payoff greater than $d(n/p + h)$.

Proposition 8 *Let $N(t)$ be the number of arrivals in a Poisson process with rate μ over the time interval $[0, t]$. Then*

- (i) $\Pr[N(t) \leq \lfloor \frac{\mu t}{2} \rfloor] \leq \exp(-0.3 \lfloor \frac{\mu t}{2} \rfloor)$;
- (ii) $\Pr[N(t) \geq \lceil 2\mu t \rceil] \leq \exp(-0.09 \lceil 2\mu t \rceil)$.

Proof: Let $S_m = \sum_{i=1}^m X_i$, where X_i are independent identically distributed exponential random variables with mean $1/\mu$. Then

$$N(t) \leq m \iff S_m \geq t.$$

The following elementary large deviation bounds hold for an arbitrary random variable Z :

$$\Pr[Z \geq z] \leq \inf_{\theta > 0} e^{-\theta z} E e^{\theta Z} \quad (5.5)$$

$$\Pr[Z \leq z] \leq \inf_{\theta > 0} e^{\theta z} E e^{-\theta Z} \quad (5.6)$$

We apply these bounds to S_m , obtaining (i) with $\theta = \mu/2$ in (5.5) and (ii) with $\theta = 2\mu$ in (5.6) via routine calculations. \square

Now we define a modified continuous-time process that incorporates a mechanism, which we call *amortization*, to keep the queue lengths small. In addition to the service phases scheduled by the adversary, we schedule *random service events* according to a Poisson process with rate 1. At each random service event, one customer is removed from each nonempty queue, and the adversary receives one unit of payoff. The effect of these random service events on the total payoff is that we amortize some of the payoff that the adversary would otherwise receive in the service phases. We show next that the adversary is better off with the amortization, namely, the payoff the adversary gain from the random service events is at least as large as the payoff the adversary is deprived of by the random service events. On the other hand, by Proposition 8(ii), the probability that more than $\lceil 8n/p \rceil$ random service events occur over the time period $[0, \lceil 4n/p \rceil]$ goes exponentially to zero as

n tends infinity. Thus it suffices to prove that, in the modified continuous-time process in which random service events occur according to a Poisson process with rate 1, the adversary has an exponentially small chance of achieving a payoff greater than $d'(n/p + h)$, for some suitable constant d' .

Proposition 9 *Let U and \tilde{U} be the number of units of payoff the adversary receives in the process with and without random service events, respectively. Let R be the number of random service events that would occur. Then $\tilde{U} \leq U + R$, as random variables.*

Proof: The proof is similar to that of Proposition 7. We choose an arbitrary instance of arrivals and random service events and show that $\tilde{U} \leq U + R$ holds for the chosen instance. We assume that all the arrival events and the random service events occur at different times, as it holds with probability one.

Let Q and \tilde{Q} denote the set of queues associated with the processes without and with random service events, respectively. Let k , s_i , d_i and \tilde{d}_i , be defined as in Proposition 7. So $\tilde{U} = \sum_{1 \leq i \leq k} \tilde{d}_i$ and $U = \sum_{1 \leq i \leq k} d_i$. Let R_i be the number of random service events that occur before the arrival of s_i and let $r_1 = R_1$ and $r_i = R_i - R_{i-1}$ for $1 < i \leq k$. So $R = \sum_{1 \leq i \leq k} r_i$. We want to show that

$$\sum_{1 \leq i \leq k} \tilde{d}_i \leq \sum_{1 \leq i \leq k} d_i + \sum_{1 \leq i \leq k} r_i. \quad (5.7)$$

As in Proposition 7, let $Q_i = \langle q_i(1), \dots, q_i(p) \rangle$ and $\tilde{Q}_i = \langle \tilde{q}_i(1), \dots, \tilde{q}_i(p) \rangle$ denote the states of Q and \tilde{Q} respectively upon the arrival of s_i and set $\alpha_i = \max_{1 \leq j \leq p} \{\tilde{q}_i(j) - q_i(j)\}$. Notice that $\alpha_1 = 0$. We will show that

$$\tilde{d}_k \leq d_k + \alpha_k \quad (5.8)$$

and for $i = 1, 2, \dots, k-1$,

$$\alpha_{i+1} \leq \alpha_i - \tilde{d}_i + d_i + r_i. \quad (5.9)$$

From (5.8) and (5.9) one can easily derive (5.7).

The proof of (5.8) is similar to the corresponding one in Proposition 7. To prove (5.9), we fix i and let $Q'_i = \langle q'_i(1), \dots, q'_i(p) \rangle$ and $\tilde{Q}'_i = \langle \tilde{q}'_i(1), \dots, \tilde{q}'_i(p) \rangle$ denote the states of Q and \tilde{Q} respectively upon the time the service phase triggered by s_i is completed. Let

$$\alpha'_i = \max_{1 \leq j \leq p} \{\tilde{q}'_i(j) - q'_i(j)\}. \quad (5.10)$$

As before, let $a_i(j)$ be the number of arrivals to the j th queue between s_i and s_{i+1} exclusively. Then $a_i(j) = q_{i+1}(j) - q'_i(j)$, as no random service events occur in Q . Set $a'_i(j) = q_{i+1}(j) - \tilde{q}'_i(j)$. Then $a_i(j) \leq a'_i(j) + r_{i+1}$, as one random service event may serve at most one arrival in each queue. Hence, for $1 \leq j \leq p$,

$$\tilde{q}_{i+1}(j) - q_{i+1}(j) \leq \tilde{q}'_i(j) - q'_i(j) + r_{i+1},$$

which gives

$$\alpha_{i+1} \leq \alpha'_i + r_{i+1}. \quad (5.11)$$

Also we have $q'_i(j) = \max\{0, q_i(j) - d_i\}$ and $\tilde{q}'_i(j) = \max\{0, \tilde{q}_i(j) - \tilde{d}_i\}$. Hence,

$$\begin{aligned} \tilde{q}'_i(j) - q'_i(j) &\leq \max\{0, \tilde{q}_i(j) - q_i(j) - \tilde{d}_i + d_i\} \\ &\leq \max\{0, \alpha_i - \tilde{d}_i + d_i\} \\ &\leq \alpha_i - \tilde{d}_i + d_i, \quad [\text{by (5.8)}] \end{aligned}$$

which gives, by (5.10),

$$\alpha'_i \leq \alpha_i - \tilde{d}_i + d_i. \quad (5.12)$$

The combination of (5.11) with (5.12) gives (5.9), completing the proof. \square

The effect of random service events on the lengths of the queues is significant. For each queue, the arrival rate is $1/2$, which is less than the rate of random service events, which is 1. So the length of a queue is likely to be small, without even considering the effect of the service phases. Let M_k be the number of service phases in which the adversary receives a payoff of at least k . These service phases correspond to instants when the special customer arrives at a queue of length at least k . Let m_k be the number of service phases in which the adversary receives a payoff of exactly k . Then the total payoff received by the adversary is

$$\sum_{k=1}^{\infty} k m_k = \sum_{k=1}^{\infty} \sum_{i=k}^{\infty} m_i = \sum_{k=1}^{\infty} M_k. \quad (5.13)$$

Our analysis of the adversary's total payoff begins by studying the probability distribution of M_k for a fixed k . For this analysis we make the pessimistic assumption that the adversary's sole purpose is to maximize M_k , the number of times the special customer arrives at a queue of size at least k . In Section 7 we investigate the frequency with which queues of length at least k occur. This is preceded by Section 6, in which we determine how often the adversary can expect to arrive at

a queue of length at least k , knowing how frequently such queues occur. Section 8 combines the results of Sections 6 and 7 to show that the adversary's chance of achieving a payoff greater than $d(n/p + h)$ is small.

5.2.3 Shooting Gallery Game

We introduce a game called the *Shooting Gallery Game*. The player of this game is a marksman who possesses m targets and h bullets. Before each shot the marksman may set up any number of targets from 1 to p . If he sets up t targets, then his chance of success is t/p . If he succeeds then his score increases by one and the t targets are destroyed. If he fails, then no targets are destroyed. He is allowed h shots, and the total number of targets available is m . The goal of the marksman is to maximize his final score.

The shooting gallery game is intended to model the situation of an adversary who watches the fluctuations of the queues, with the goal of scheduling the arrivals of special customers at times when they are likely to arrive at a queues of length at least k . A shot in which t targets are set up is intended to represent the arrival of a special customer at a time when t of the p queues are of length k or greater. Thus m , the number of targets, corresponds to the number of moments when some queue reaches size k , and h , the number of shots, represents the number of special arrivals. Our analysis favors the adversary by giving him complete freedom to allocate the targets to shots, subject to a restriction on the total number of targets and the number of shots.

The marksman has a dilemma: setting up more targets gives him a better chance of success; but such a success would also destroy more targets. Suppose that a fixed number of targets, say a , are set up each time. Then m targets allows a score of at most m/a and h bullets achieves a score of ha/p on average. These two limits on the marksman's score become approximately equal when $a = \lceil \sqrt{mp/h} \rceil$. Such a choice enables the marksman to achieve an expected score close to $\sqrt{mh/p}$. We show that the marksman has an exponentially small chance of achieving a score that is substantially better than $\sqrt{mh/p}$ no matter how he sets up the targets.

Theorem 7 *Let S denote the marksman's final score. Then*

$$\Pr[S > 3\sqrt{mh/p}] \leq \exp(-\frac{1}{6}\sqrt{mh/p}),$$

no matter how the marksman selects the number of targets at each step.

Proof: If $h \leq 9m/p$, then $3\sqrt{mh/p} \geq h \geq S$ and the proposition trivially holds. We assume that $h > 9m/p$. Let $a = \frac{1}{2}\sqrt{mp/h} < p$.

We change the scoring rules in the marksman's favor as follows. Let us say that a shot is *of type 1* if more than a targets are set up, and *of type 2* if a or fewer targets are set up. Then we count each type 1 shot as a success, and give each type 2 shot a chance of success a/p . The number of successes in type 1 shots is at most $m/a = 2\sqrt{mh/p}$. Let $B(n, q)$ denote the number of successes in a Bernoulli process with n trials, each having a chance of success q . Then the number of successes in type 2 shots is stochastically dominated by $B(h, a/p)$. Then

$$\Pr[S > 3\sqrt{mh/p}] \leq \Pr[B(h, a/p) > \sqrt{mh/p}] \leq \exp(-\frac{1}{6}\sqrt{mh/p}),$$

where the final inequality follows from the Chernoff bound on the tail of the binomial distribution. \square

5.2.4 A Simple Process

Continuing our analysis of the random variable M_k , we investigate the frequency with which the length of a given queue is greater than or equal to k . Disregarding the service phases caused by the adversary, each queue has Poisson arrivals with rate $1/2$ and Poisson service with rate 1 . This queueing process is a simple *birth-and-death* process. Let an *event* be either an arrival event or a service event in the process. Associated with this process is a Markov chain $\{X_i\}$ where X_i is the number of customers in the queue after the i th event. The Markov chain $\{X_i\}$ is a *simple random walk* on nonnegative integers, started at $X_0 = 0$, with transition probabilities

$$p_{0,1} = \frac{1}{3} = 1 - p_{0,0}$$

and

$$p_{j,j+1} = \frac{1}{3} = 1 - p_{j,j-1}, \quad \text{for } j > 0,$$

where $p_{j,j'} = \Pr[X_{i+1} = j' | X_i = j]$. This Markov chain has a stationary distribution $\{\pi_i\}$, $i = 0, 1, \dots$, with

$$\pi_i = 1/2^{i+1}. \quad (5.14)$$

Let B_k be the set of states $\{i : i \geq k\}$. Let $U_k(m)$ denote the number of times that $\{X_i\}$ is in B_k up to step m , given that the state at time 0 is 0. The following

is the main theorem of this section. It gives an upper bound on the probability of a large deviation of $U_k(m)$ from $E[U_k(m)]$.

Theorem 8 For any fixed $k > 0$,

$$\Pr(|U_k(m) - E[U_k(m)]| \geq x) \leq \exp(-\frac{x^2}{32m}).$$

We need three lemmas. A sequence of random variables $\{Y_i : i = 0, 1, \dots\}$ is a *martingale* if, for $i \geq 0$, (i) $E[|Y_i|] < \infty$ and (ii) $E[Y_{i+1}|Y_0, \dots, Y_i] = Y_i$.

Lemma 1 (Martingale Tail Inequality [SS87]) Let $\{Y_i : i = 0, 1, \dots\}$ be a martingale such that $|Y_{i+1} - Y_i| \leq c$ for $0 \leq i < n$. Then

$$\Pr(Y_n \geq Y_0 + c\alpha\sqrt{n}) \leq e^{-\alpha^2/2}.$$

Let T_{a0} be the expected number of steps $\{X_i\}$ takes to reach state a from state 0. Notice that $T_{a0} = aT_{10}$.

Lemma 2 $T_{10} = 3$.

Proof: By conditioning on the next state,

$$T_{10} = 1 + \frac{T_{20}}{3} = 1 + \frac{2T_{10}}{3}.$$

□

Let $E_k^a(m)$ denote the expected number of times that $\{X_i\}$ is in B_k up to step m , given that the state at time 0 is a . By definition, $E_k^{a+1}(m) \geq E_k^a(m)$ and $E_k^a(m) \leq E_k^a(m+1) \leq E_k^a(m) + 1$.

Lemma 3 For $|a - b| \leq 1$,

$$|E_k^a(m) - E_k^b(m-1)| \leq T_{10}.$$

Proof: There are three cases:

- (i) $a = b$, $|E_k^a(m) - E_k^a(m-1)| \leq 1$;
- (ii) $b = a - 1$, $E_k^{a-1}(m-1) \leq E_k^a(m) \leq T_{10} + E_k^{a-1}(m-1)$;
- (iii) $b = a + 1$, $E_k^a(m) - 1 \leq E_k^{a+1}(m-1) \leq T_{10} + E_k^a(m)$. □

Proof of Theorem 8: For $0 \leq i \leq m$, let

$$Y_i = E(U_k(m) | X_0, X_1, \dots, X_i).$$

Then $\{Y_0, Y_1, \dots, Y_m\}$ is a martingale (see e.g. [Do53, p92]) with $Y_0 = E[U_k(m)]$ and $Y_m = U_k(m)$. By the Markovian property,

$$Y_i = U_k(i) + E_k^{X_i}(m - i).$$

Since $|U_k(i+1) - U_k(i)| \leq 1$,

$$|Y_{i+1} - Y_i| \leq 1 + |E_k^{X_{i+1}}(m - i - 1) - E_k^{X_i}(m - i)|. \quad (5.15)$$

But $|X_{i+1} - X_i| < 1$. By Lemmas 3 and 2,

$$|Y_{i+1} - Y_i| \leq 1 + T_{10} \leq 4 = c. \quad (5.16)$$

By setting $\alpha = \frac{1}{4} x m^{-1/2}$ in Lemma 1,

$$\Pr(U_k(m) - E[U_k(m)] \geq x) \leq \exp\left(-\frac{x^2}{32m}\right).$$

□

Proposition 10 $E[U_k(m)] \leq 2^{-k}m$.

Proof: By the ergodic theorem for Markov chains and (5.14),

$$\lim_{m \rightarrow \infty} \frac{E[U_k(m)]}{m} = \sum_{i=k}^{\infty} \pi_i = 2^{-k}. \quad (5.17)$$

We claim that $E[U_k(m)] \leq 2^{-k}m$ for all $m \geq 0$. Suppose on the contrary that $E[U_k(m)] > 2^{-k}m$ for some m_0 . Let $m = tm_0$ be a multiple of m_0 . Since $E_k^0(m_0) = E[U_k(m_0)]$ and $E_k^a(m_0) \leq E_k^b(m_0)$ for $b \geq a$,

$$\begin{aligned} \frac{E[U_k(m)]}{m} &= \frac{\sum_{j=0}^{t-1} \sum_{a=0}^{\infty} \Pr(X_{jm_0} = a) E_k^a(m_0)}{tm_0} \\ &\geq \frac{t E_k^0(m_0)}{tm_0} = \frac{E[U_k(m_0)]}{m_0} > 2^{-k}. \end{aligned}$$

Since m is an arbitrarily large multiple of m_0 , this contradicts (5.17). □

The following is an important corollary of Theorem 8 and Proposition 10.

Corollary 3

$$\Pr[U_k(m) \geq \frac{m}{2^{k-1}}] \leq \exp\left(-\frac{m}{2^{2k+5}}\right).$$

Proof:

$$\Pr[U_k(m) \geq \frac{m}{2^{k-1}}] \leq \Pr[U_k(m) - E[U_k(m)] \geq \frac{m}{2^k}] \leq \exp\left(-\frac{m}{2^{2k+5}}\right).$$

□

5.2.5 The Distribution of M_k

We shall study the distribution of M_k for a fixed k . Having fixed k , let us focus on the history of a particular queue i . The events affecting the queue are arrivals, the random service events, and the service phases triggered by the adversary.

In the absence of service phases caused by the adversary, the process in queue i becomes the birth-and-death process studied in the previous section. The events are arrivals, which occur with rate $1/2$, and random service events, which occur with rate 1. Thus events in queue i occur according to a Poisson process with rate $3/2$. There are p queues. Thus, by Proposition 8(ii), the probability that more than $m_0 = \lceil 12n/p \rceil$ events occur in some queue is at most

$$q_0 = p e^{-1.08n/p}. \quad (5.18)$$

We shall assume that no more than m_0 arrival and random service events occur in any queue, and study the distribution of M_k under this assumption.

A service phase is *k-profitable* if it results in a payoff of at least k for the adversary. The time interval between successive *k-profitable* service phases is called a *k-interval*. Queue i is said to be *k-eligible* during a given *k-interval* if, at some point during the interval, the length of queue i is at least k . Let the random variable $X(k, i)$ denote the number of *k-intervals* during which queue i is *k-eligible*.

Proposition 11

$$\Pr[X(k, i) > \frac{m_0}{2^{k-1}}] \leq \exp(-\frac{m_0}{2^{2k+5}}), \quad (5.19)$$

where $m_0 = \lceil 12n/p \rceil$.

Proof: We give the adversary extra power by assuming that the adversary observes the stream of arrivals and service events, is able to trigger service phases whenever he wishes, and has complete freedom to specify the payoff associated with each service phase. Then the adversary maximizes the number of *k-intervals* during which queue i is *k-eligible* by scheduling a service phase with payoff k every time the length of queue i reaches k , and otherwise leaving the queue alone. Under this policy for the adversary, the number of *k-intervals* during which the queue is *k-eligible* is just the number of times the queue length reaches k and drops instantaneously from k to zero, which in turn is no larger than the number of times that the state of the underlying birth-and-death process reaches a value greater than or equal to k .

Under our assumption that there at most m_0 arrival and random service events, $X(k, i)$ is stochastically no more than $U_k(m_0)$. By Corollary 3,

$$\Pr[X(k, i) \geq \frac{m_0}{2^{k-1}}] \leq \Pr[U_k(m_0) > \frac{m_0}{2^{k-1}}] \leq \exp(-\frac{m_0}{2^{2k+5}}).$$

□

Corollary 4 *Let $T_k = \sum_{i=1}^p X(k, i)$. Then*

- (i) $T_k \geq T_{k'}$ if $k \leq k'$;
- (ii) $\Pr[T_k > 24n2^{-k}] \leq p \exp(-\frac{n}{p} 2^{-(2k+1)})$.

Proof: The fact that $X(k, i) \geq X(k', i)$ if $k \leq k'$ implies (i) whereas (ii) is direct consequence of (5.19) by noting $m_0 = \lceil 12n/p \rceil$. □

Now we are ready to analyze the random variable M_k , which is the number of times the adversary achieves a payoff of at least k . We draw an analogy between the Shooting Gallery Game and our continuous-time model. The number of bullets h corresponds to the number of service phases that the adversary can trigger. The targets correspond to pairs (I, i) where I is a k -interval and i is a queue that is k -eligible during interval I . Therefore, at most $T_k = \sum_{i=1}^p X(k, i)$ targets are available to the marksman. The act of setting up t targets and taking a shot corresponds to executing a service phase at a time when t queues are of length at least k . A successful shot corresponds to the arrival of the special customer at a queue of length at least k . The marksman's score corresponds to M_k , the number of times the adversary receives a payoff of at least k .

Recall that the payoff received by the adversary is $\sum_{k=1}^{\infty} M_k$ by (5.13). The following theorem completes the proofs of Theorems 5 and 6.

Theorem 9 *Suppose that $n \geq p^3$. There exist constants α' , β' , γ' and d' such that*

$$\Pr[\sum_{k=1}^{\infty} M_k > d'(\frac{n}{p} + h)] \leq \gamma' p n \exp(-\frac{\beta'}{p} n^{\alpha'}).$$

Proof: We break $\sum_{k=1}^{\infty} M_k$ into three parts and each part separately. Let

$$\sum_{k=1}^{\infty} M_k \leq \sum_{k=1}^{\lfloor a \log n \rfloor} M_k + \sum_{k=\lfloor a \log n \rfloor}^{\lfloor n^b \rfloor} M_k + \sum_{k=\lfloor n^b \rfloor}^{\infty} M_k,$$

where $a < 1$ and $b < 1$ are positive constants to be specified later.

Let \mathcal{E}_1 be the event that $T_k > 24 n 2^{-k}$ for some k , $1 \leq k \leq \lfloor a \log n \rfloor$. By Corollary 4,

$$\begin{aligned} q_1 &= \Pr[\mathcal{E}_1] \\ &\leq \sum_{k=1}^{\lfloor a \log n \rfloor} p \exp\left(-\frac{n}{p} 2^{-2k-1}\right) \\ &\leq a p \log n \exp(-0.5 n^{1-2a}/p), \end{aligned} \quad (5.20)$$

which goes exponentially to zero, if

$$a < 1/2. \quad (5.21)$$

Let \mathcal{E}_2 be the event that for some k , $1 \leq k \leq \lfloor a \log n \rfloor$, a marksman with T_k targets and h bullets achieves a score of at least $3\sqrt{T_k h/p}$. Then by Theorem 7, the probability of event \mathcal{E}_2 given that event \mathcal{E}_1 does not occur is

$$\begin{aligned} q_2 &= \Pr[\mathcal{E}_2 \mid \mathcal{E}_1 \text{ does not occur}] \\ &\leq \sum_{k=1}^{\lfloor a \log n \rfloor} \exp\left(-\frac{1}{6} \sqrt{T_k h/p}\right) \\ &\leq a \log n \exp\left(-\frac{1}{6} \sqrt{T_{\lfloor a \log n \rfloor} h/p}\right) \\ &\leq a \log n \exp(-0.25 \sqrt{h n^{1-a}/p}), \end{aligned} \quad (5.22)$$

which goes exponentially to zero, since $a < 1$.

The probability that either event \mathcal{E}_1 or event \mathcal{E}_2 occurs is at most $q_1 + q_2$, which is exponentially small. We assume that neither event \mathcal{E}_1 nor event \mathcal{E}_2 occurs. By Theorem 7, viewing M_k as the score of the marksman,

$$\begin{aligned} \sum_{k=1}^{\lfloor a \log n \rfloor} M_k &\leq \sum_{k=1}^{\lfloor a \log n \rfloor} 3 \sqrt{\frac{T_k h}{p}} \\ &< \sqrt{\frac{n h}{p}} \sum_{k=1}^{\infty} 3 \sqrt{2^{-(k+1)} 3} \\ &\leq c' \left(\frac{n}{p} + h \right), \end{aligned} \quad (5.23)$$

where $c' < 9$ and the last inequality follows from the fact that $\sqrt{nh/p} \leq n/p + h$.

Since the score cannot be larger than the number of targets, $M_k \leq T_k$. Also $T_k \geq T_{k+1}$. Thus, by Corollary 4,

$$\sum_{k=\lfloor a \log n \rfloor}^{\lfloor n^b \rfloor} M_k \leq n^b T_{\lfloor a \log n \rfloor} \leq 24 n^b n^{1-a} = c'' \frac{n}{p}, \quad (5.24)$$

where $c'' = 24pn^{b-a}$ and the second inequality follows from our assumption that event \mathcal{E}_1 does not occur. Thus

$$c'' \leq 24, \quad \text{if } p \leq n^{a-b}. \quad (5.25)$$

Lastly we show that with high probability

$$M_k = 0, \quad (5.26)$$

for all $k \geq \lceil n^b \rceil$. Since $M_k \leq T_k \leq T_{k'}$ for $k \geq k'$,

$$\begin{aligned} M_k &> 0, \text{ for some } k \geq \lceil n^b \rceil \\ \Rightarrow T_{\lceil n^b \rceil} &> 0 \\ \Rightarrow X(\lceil n^b \rceil, i) &> 0, \text{ for some } i. \end{aligned}$$

But $X(\lceil n^b \rceil, i) \leq U_{\lceil n^b \rceil}(m_0)$,

$$\begin{aligned} q_3 &= \Pr\left[\sum_{k=\lceil n^b \rceil}^{\infty} M_k > 0\right] \\ &\leq p \Pr[U_{\lceil n^b \rceil}(m_0) > 0] \\ &\leq pE[U_{\lceil n^b \rceil}(m_0)] \\ &\leq 3n2^{-n^b}. \end{aligned} \quad (5.27)$$

To meet the constraints on a and b in (5.21) and (5.25), together with the assumption that $n \geq p^3$, we may take

$$a = 4/9 \quad \text{and} \quad b = 1/9$$

so that

$$\alpha' = 1/9, \quad \beta' \geq 1/4 \quad \text{and} \quad \gamma' \leq 5,$$

according to q_0, q_1, q_2 and q_3 in (5.18), (5.20), (5.22) and (5.27). By (5.23), (5.24) and (5.26),

$$d' = c' + c'' \leq 33.$$

□

The proof of Theorem 6 is complete.

5.3 PRAM Implementation

In this section we describe a PRAM implementation of our algorithm in the case that H is a binary tree and show that the Main Theorem holds for this PRAM implementation.

Our model is a CRCW PRAM using the ARBITRARY concurrent-write convention; this means that, if two or more processors try to write simultaneously into a cell of the shared memory, the processor that succeeds is chosen arbitrarily. We assume that a processor can detect whether it has succeeded in a write and that the value a processor writes is the value of a node together with the necessary information for further expansion of that node. There are p processors and the shared memory consists of $4p$ cells which are divided into p blocks of 4 cells each. For convenience, we shall allow a processor to attempt two writes simultaneously in one write step. A node is called *distributed* if the value of the node has been successfully written into the shared memory.

We modify our previous communication protocol as follows.

- (i) In the *Node Expansion Step*, processor i with a nonempty F_i executes the following segment, replacing the corresponding segment in the original protocol:

```

let  $v_i$  be the node of least cost in  $F_i$ ;
expand  $v_i$ ;
if  $v_i$  is a leaf then
     $B_i \leftarrow c(v_i)$ ;
     $F_i \leftarrow F_i \setminus \{v_i\}$ ;
else
    let  $w$  and  $w'$  be the children of  $v_i$ ;
    if neither  $w$  nor  $w'$  was distributed then
        select two random cells;
        write  $w$  into one cell and  $w'$  into the other;
    else /*one of  $w$  and  $w'$  is not distributed*/
        select one random cell;
        write the child of  $v_i$  who is not distributed into the cell;
    if both  $w$  and  $w'$  are distributed then  $F_i \leftarrow F_i \setminus \{v_i\}$ ;

```

- (ii) In the *Message Arrival Step*, processor i receives new nodes by reading the

newly written values from the 4 cells of block i .

The modified protocol maintains the property that each generated node is equally likely to be received by any processor.

We call a node expansion *successful* if both children of the expanded node are distributed after that expansion. There are at most $2p$ writes attempted simultaneously, each selecting a random cell from $4p$ cells. Thus the chance that a write is successful is at least $1/2$, independently of other writes. So the chance that a node expansion is successful is at least $1/4$. The *success number* of a node v is the number of times v is expanded until a successful node expansion occurs. Since the successive node expansions of v are independent of each other, the success number of v is stochastically no larger than a geometric random variable with mean 4.

We show that Theorem 1 holds with this PRAM implementation by reducing the analysis of the PRAM implementation to our previous analysis. Again our goal is to prove Theorem 2.

The modified protocol maintains the property that each processor expands its best node at each node expansion step. So we may focus on the nodes in \widetilde{H} . We call a node a *pre-node* if it causes pre-delays, or a *post-node* if it causes post-delays and a node expansion step a *special-step* if a special node is expanded at that step. We observe that the node expansions of the special nodes, the pre-nodes and the post-nodes all occur at different times, thus the node expansion processes of these nodes, maybe intertwined in time, are mutually independent. There are at most h special nodes and, by the Chernoff bound, the chance that there are more than $2n/p$ post-nodes is at most $e^{-n/3p}$. Given that there are at most $2n/p$ post-nodes the sum of post-delays and special-delays is stochastically no larger than a sum of $h + 2n/p$ independent geometric random variables with mean 4. But the latter sum, by the Chernoff bound again, has an exponentially small probability to exceed $d'(n/p + h)$ for a suitable constant d' .

It remains to bound the number of pre-delays. In the queueing model of our process, we observe that the strategy consistent with Rules 1 and 2 remains optimal for the adversary. This can be seen as follows. In the modified protocol, an instance consists of a sequence of arrivals with their destinations and their success numbers. An instance in the modified protocol can be viewed as an instance in the original protocol by replacing each arrival in the modified protocol by a multiple of the arrival itself with the same destination, where the multiple is the success number of

that arrival. The fact that the strategy consistent with Rules 1 and 2 is optimal for any instance in the original protocol implies that the same strategy remains optimal for any instance in the modified protocol.

Given the optimality of the strategy consistent with Rules 1 and 2, we may instead study our queueing process in continuous time by assuming that customers arrive according to a Poisson process with rate $1/8$ over the time period $[0, \lceil 16n/p \rceil]$, and further amortize the payoff of the adversary, the number of pre-delays, by scheduling random service events according to a Poisson process with rate 1.

To bound the additional payoff that the adversary may receive in the amortized process, we focus on the frequency with which the length of a fixed queue is large under the amortization. For this purpose, we may assume, to the advantage of the adversary, that the random service events are the only service events that occur and that the chance that a service is successful is exactly $1/4$. We fix a queue and let an *event* be either a random service event or an arrival at that queue. Let X_i be the length of the queue after the i th event. Then $\{X_i\}$, $i = 0, 1, 2, \dots$, is a Markov chain with $X_0 = 0$. By the fact that the process of arrivals is Poisson with rates $1/8$ and the process of services is Poisson with rates 1, the chance that the next event is an arrival or a service is $1/9$ or $8/9$, respectively. Hence, the transition probabilities of $\{X_i\}$ are

$$p_{0,1} = \frac{1}{9} = 1 - p_{0,0}$$

and for $j > 0$,

$$p_{j,j+1} = \frac{1}{9}, \quad p_{j,j} = \frac{7}{9} \quad \text{and} \quad p_{j,j-1} = \frac{2}{9},$$

where $p_{j,j'} = \Pr[X_{i+1} = j' | X_i = j]$.

It is easy to check that $\pi_i = 1/2^{i+1}$, $i \geq 0$, is the stationary distribution of the above Markov chain. The remaining analysis of this Markov chain follows exactly as we did in Section 5.2.4 and Section 5.2.5 with only minor changes, such as $T_{10} = 9$ in Lemma 2. We can conclude that the probability that the number of pre-nodes exceeds $d''(n/p + h)$ for some constant d'' goes to zero exponentially fast, which implies that the same conclusion holds regarding the number of pre-delays, thus proving Theorem 6.

5.4 Remaining Issues

In this section we specify some details omitted in our description of Local Best-First Search and discuss the consequences of allowing two nodes to have the same cost.

5.4.1 Some Omitted Details

In the code for Local Best-First Search given in Section 5.1, we left unspecified the designated destinations for a processor to send its message when its local frontier is empty and how messages are used to notify all processors of the minimum cost and turn the computation off. We now specify these details.

We configure the processors into a uniform binary tree of preassigned structure. When a processor has an empty local frontier, it sends the message containing its bound B_i to all the neighbors in the tree. Let τ be the time when all nodes of \widetilde{H} have been expanded and B be the minimum cost of a leaf of H . At time τ , at least one processor will possess the bound B . From time τ onward, each processor that has received the bound B will have an empty local frontier, and will use each subsequent node expansion step to send the bound B to its neighbors in the tree. Thus, all processors will receive the bound B by some later time σ , where $\sigma \leq \tau + 2 \log n$. From time σ onward, all local frontiers will be empty.

The computation will stop as soon as each process learns that a point has occurred at which all local frontiers are empty. In a PRAM, this information can be conveyed by executing occasional verification steps according to a preassigned schedule known to all processors. At each verification step, each processor with a nonempty local frontier tries to write its name in a designated cell. All the processors then read that cell to determine whether a value has been stored. On a message-passing network, the same objective can be achieved by scheduling occasional broadcast phases, in which each processor with a nonempty frontier sends its name to the root processor along the edges of the tree, and the root processor then broadcasts the information it has received to all nodes of the tree. The intervals between verification or broadcast steps can be chosen so that these steps have no appreciable influence on the overall execution time.

5.4.2 Consequence of Equal Costs

We have assumed that the nodes of H have distinct costs. When the nodes of H are allowed to have the same cost, the statement of Theorem 5 needs some modification, together with some minor changes in the Local Best-First Search.

The *level* of a node v in H , denoted by $l(v)$, is the distance from v to the root of H . Given a problem instance (H, c) , we define a new order " \prec " among the nodes of H such that $v \prec u$ if (i) $c(v) < c(u)$ or (ii) $c(v) = c(u)$ and $l(v) < l(u)$ or (iii) $c(v) = c(u)$ and $l(v) = l(u)$ and v is to the left of u in H . We say that the *priority* of v is higher than that of u if $v \prec u$. In the Local Best-First Search, the following is the selection rule: each processor selects the node of highest priority from its local frontier.

Let \hat{v} be the leaf of highest priority in H . Let \widehat{H} be the subtree consisting of those nodes in H whose priorities are no higher than the priority of \hat{v} . Notice that \hat{v} is a leaf of minimum cost. Let S be the set of nodes $u \in \widehat{H}$ such that $c(u) < c(\hat{v})$. Every node expansion algorithm that determines a leaf v' of minimum cost in H must expand every node in S and v' . Let n_1 be the number of nodes in S , and let n_2 be the number of nodes that are not in \widehat{H} and let h be the number of nodes in a longest root-leaf path in \widehat{H} . Then $\max\{(n_1 + 1)/p, h\}$ is an inherent lower bound on the execution time of any p -processor algorithm on the instance (H, c) . On the other hand, by the Main Theorem, the Local Best-First Search is likely to locate the leaf \hat{v} within $d(n/p + h)$ number of steps where d is a constant and $n = n_1 + n_2$.

5.5 Open Problems and Further Research

We discuss some open problems and possible further research related to the algorithm presented in this chapter.

1. The main open problem of this chapter is similar to that of Chapter 3. Like the previous parallel backtrack search algorithms, Local Best-First Search assumes a fully-connected network. One can implement the algorithm in a sparse network by routing messages. But this would lose some of the advantage of the algorithm. To avoid that message-routing delay, one can modify the algorithm so that, whenever a processor expands a node, it sends the children of that node to random neighbors, rather than to random processors anywhere

in the network. The effectiveness of this modified algorithm will depend critically on the interconnection structure. The main open problem is to analyse the modified Local Best-First Search on the hypercube, the butterfly or other interesting sparsely interconnected networks.

2. In practice, distributing the children of the expanded node after each node expansion may be excessive and unnecessary. It would be desirable to find some way to reduce the frequency of the interprocessor communications while not jeopardizing the effectiveness of the algorithm.
3. We have completely ignored the cost of managing the local frontiers. The frontier nodes of each processor can be kept in a priority queue to facilitate the selection of the node of least cost; each removal or insertion incurs a cost proportional to the logarithm of the size of the queue. The size of each queue depends very much on the cost function $c(v)$. In analyzing the costs of updating the priority queues, we can no longer disregard the nodes that are not in \widetilde{H} , as we did in analyzing the node expansion steps of the algorithm. The nodes that are not in \widetilde{H} can be kept in the queues, and thus influence the costs of updates of the queues.

Chapter 6

Game Tree Search

6.1 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 $\text{val}(v)$. 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.

The best known heuristic in practice for evaluating MIN/MAX trees is the α - β pruning procedure [KM75]. For 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 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 computation, and conse-

quently there has been a considerable research effort on parallel game-tree search over the past decade. Most of this research is concerned with implementation and experimentation with parallel game tree search. Apart from a recent paper [Alth89], which is discussed in Section 6.3, there has been little theoretical study in this area. We contribute to the study of the parallel game-tree evaluation algorithms by presenting a paradigm for parallelizing a class of parallel game-tree evaluation algorithms, including the left-to-right algorithm and the α - β algorithm.

6.2 Models of Computation

We shall base on our study of the game tree evaluation algorithms 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 execution time of an algorithm is the number of basic steps it requires to determine the value of the root. 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 execution 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.

These two models are different in nature. An algorithm that can be easily described in one model may not be easily described in the other model. However, for all the algorithms discussed in this chapter and the next, the description of an algorithm in one model can be translated straightforwardly into a counterpart

description of the algorithm in the other model.

6.3 Sequential Algorithms

In this section we discuss two well-studied sequential game-tree evaluation algorithms, the “left-to-right” algorithm for evaluating AND/OR trees and the α - β pruning algorithm for evaluating MIN/MAX trees.

There are a number of other game tree evaluation algorithms. The most notable ones include *SCOUT* [Pe80], which is a combination of the “left-to-right” algorithm and the α - β pruning algorithm, and *SSS** [Sto79], a best-first search in the space of solutions trees. There is also the interesting heuristic based on “MIN/MAX approximation” [Ri88]. As we do not intend to discuss the parallelization of these algorithms, we shall not describe them here.

We shall use $B(d, n)$ to denote the set of uniform d -ary AND/OR trees of height n and $M(d, n)$ to denote the set of uniform d -ary MIN/MAX trees of height n .

6.3.1 Sequential SOLVE

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 following are some basic facts about NOR-tree evaluation.

Fact 1 *Any deterministic algorithm for evaluating NOR-trees has to evaluate every leaf in the worst case.*

Proof: Given a fixed NOR-tree, the leaf-values of the tree can be assigned in such a way that the given deterministic algorithm is forced to look at every leaf before it can determine the value of the root. Let r be the root of the tree. Suppose that v is the leaf to be evaluated next. If v has unevaluated siblings, then assign the value of v to 0; otherwise, let u be the node closest to r on the root-leaf path ending at v such that u has only one unevaluated child, and assign the value of v to the value with which u is evaluated to 0. Under this assignment, every leaf will be evaluated.

□

The *total work* of an algorithm, sequential or parallel, 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. The following fact gives an inherent lower bound on the total work of any algorithm which evaluates any instance of $B(d, n)$.

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

Proof: 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 $d^{\lfloor n/2 \rfloor}$. \square

Let T be any rooted tree with root r . Let v be a node in T . The value of v is *determined* if $\text{val}(v)$ 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( $\text{val}(v)$ );
  else
    let  $u_1, u_2, \dots, u_d$  be the children of  $v$ ;
    for  $i = 1$  to  $d$  do
       $b \leftarrow \text{S-SOLVE}(u_i)$ ;
      if  $b = 1$  then
        return (0);
    return (1).

```

We shall study the properties of Sequential SOLVE. By Fact 1, any deterministic algorithm that evaluates AND/OR trees would have to evaluate all the leaves. To avoid this worst-case behavior, researchers have taken 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 a NOR-tree is determined by an independent coin flip with a fixed bias q , $0 \leq q \leq 1$. Let $I_A(d, n, q)$ denote the expected number of leaves evaluated by a deterministic algorithm A that evaluates random uniform d -ary NOR-trees of height n in the i.i.d. model where the bias is q . The quantity $R_A(d, q) = \lim_{n \rightarrow \infty} [I_A(d, n, q)]^{1/n}$ is called the *branching factor* of algorithm A . We abbreviate Sequential SOLVE as S-SOLVE. The following theorem can be found in [Pe84].

Theorem 10 *Let ξ_d be the unique positive root of the equation $x^d + x - 1 = 0$. Then*

- (i) *if $q = \xi_d$, $I_{S-SOLVE}(d, n, q) = [\xi_d/(1 - \xi_d)]^n$;*
- (ii) *if $q \neq \xi_d$, $R_{S-SOLVE}(d, q) = \sqrt[d]{d}$.*

Theorem 10 shows that Sequential SOLVE exhibits a “threshold” phenomenon in the i.i.d. model. When the bias does not coincide with the special value ξ_d , Sequential SOLVE has the best possible asymptotic performance by Fact 2; when the bias coincides with ξ_d , Sequential SOLVE performs worst, as one can show that $\xi_d/(1 - \xi_d) > \sqrt[d]{d}$. In particular, $\xi_2 = (\sqrt{5} - 1)/2 = 0.618\dots$, which is the “golden ratio” and greater than $\sqrt{2} = 1.414\dots$, and $I_{S-SOLVE}(2, n) = (1 + \xi_2)^n = (1.618\dots)^n$.

In the i.i.d. model with bias q , a deterministic algorithm A is said to be *optimal* for evaluating random uniform trees if for all d, n, q and any deterministic algorithm A' , $I_A(d, n, q) \leq I_{A'}(d, n, q)$. In 1983, M. Tarsi showed that Sequential SOLVE is optimal for all values of q , in particular, for the threshold value ξ_d . The proof technique of Tarsi is to show that for any algorithm A' , another algorithm A'' can be constructed inductively such that $I_{A''}(d, n, q) \leq I_{A'}(d, n, q)$ and A'' is equivalent to Sequential SOLVE.

Theorem 11 (Tarsi) *Sequential SOLVE is optimal in the i.i.d. model with any bias.*

Proof: See [Ta83]. \square

The optimality of Sequential SOLVE suggests that Sequential SOLVE be a good candidate for parallelization. A recent paper by Althöfer [Alth89] gives a probabilistic analysis of a certain algorithm for evaluating uniform binary AND/OR trees in the i.i.d. model where the bias q is equal to the critical value $(\sqrt{5} - 1)/2$. He states that, when the number of processors is moderate, the expected speed-up over Sequential SOLVE is proportional to the number of processors. We will show that Sequential SOLVE can be parallelized in such a way that, uniformly on every instance of a uniform tree, a linear speed up can be achieved. Consequently, our result holds in the i.i.d. model automatically.

Another probabilistic approach is through randomized algorithms. This approach avoids imposing any assumptions on inputs and hence is more robust. A *randomized* algorithm is allowed to flip coins, and the actions of the algorithm may depend on the outcomes of these flips. 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 Sequential SOLVE: repeatedly choose an unevaluated child of the root at random and evaluate the child recursively until the value of the root can be determined. It turns out that randomization helps achieve a substantial reduction in the number of leaves evaluated. Moreover, it has been shown that randomized Sequential SOLVE is optimal among randomized sequential algorithms for evaluating uniform NOR-trees. Precise statements of these results and their proofs can be found in [SW86].

6.3.2 α - β Pruning

The most well-known method for evaluating MIN/MAX trees is α - β pruning. McCarthy thought of this method in early 60's and later coined the name " α - β " pruning. In 1976, Knuth and Moore published a paper on the analysis of α - β pruning method, which remains to this day a definitive reference.

The power of α - β pruning lies in its ability to detect certain subtrees whose values can no longer influence the value of the root and, consequently, the algorithm prunes away those subtrees from further evaluation. The algorithm traverses the input tree in a depth-first search and backtracks when it prunes a subtree. During the search, it maintains two parameters, the α -*bound*, denoted by α , and the β -*bound*, denoted by β , with the invariant property that $\alpha < \beta$. The α -bound is updated by the

returned values on the MAX-nodes and the β -bound is updated by the returned values on the MIN-nodes. The interval $[\alpha, \beta]$ is called the *search-window*. The α - β pruning algorithm is described by the following procedure *alphabeta*(v, α, β) where $[\alpha, \beta]$ is the search window and v is the root of the subtree to be evaluated. To evaluate a MIN/MAX tree T with root r , the α - β pruning algorithm calls procedure *alphabeta*($r, -\infty, +\infty$). The *current value* of a MAX-node (MIN-node) v is the maximum (minimum) of the returned values of the evaluated children of v . Initially, the current value of a MAX-node (MIN-node) is $-\infty$ ($+\infty$).

```

alphabeta( $v, \alpha, \beta$ );
{
  if ( $v$  is a leaf) return(val( $v$ ));
  else {
    /*initialize current value*/
    if ( $v = \text{MAX}$ )  $s \leftarrow -\infty$ ;
    else  $s \leftarrow +\infty$ ;
    let  $u_1, u_2, \dots, u_d$  be the children of  $v$ ;
    for  $i = 1$  to  $d$  do {
       $t \leftarrow \text{alphabeta}(u_i, \alpha, \beta)$ ;
      if ( $v = \text{MAX}$ )
        if ( $t \geq \beta$ ) return( $t$ ); /* prune remaining children */
        else {  $\alpha \leftarrow \max\{\alpha, t\}$ ;  $s \leftarrow \max\{s, t\}$ ; }
      if ( $v = \text{MIN}$ )
        if ( $t \leq \alpha$ ) return( $t$ ); /* prune remaining children */
        else {  $\beta \leftarrow \min\{\beta, t\}$ ;  $s \leftarrow \min\{s, t\}$ ; }
    } /* end of for */
  } /* end of if */
  return( $s$ ).
}

```

A more compact form, called “negmax”, of the procedure *alphabeta*(v, α, β) can be found in [KM75].

The correctness of α - β pruning follows from the following proposition given in [KM75].

Proposition 12 (Knuth and Moore [KM75]) *Let t be the value returned by a procedure call $\text{alphabeta}(v, \alpha, \beta)$ where $\alpha < \beta$ and v is the root of the MIN/MAX tree. Then*

- (i) $t \leq \alpha$, if $\text{val}(v) \leq \alpha$;
- (ii) $t \geq \beta$, if $\text{val}(v) \geq \beta$;
- (iii) $t = \text{val}(v)$, if $\alpha < \text{val}(v) < \beta$.

In particular, $\text{alphabeta}(v, -\infty, +\infty)$ returns $\text{val}(v)$.

It is easy to construct instances of uniform MIN/MAX tree such that the α - β pruning algorithm would have to evaluate all the leaves. As for the AND/OR trees, one can make some probabilistic assumption on the input instance and study the expected number of leaves evaluated on such a random input. In the i.i.d. model, the leaf-values of a MIN/MAX tree are drawn independently from some common continuous distribution. Let $I_A(d, n, F)$ denote the expected number of leaves evaluated by a deterministic algorithm A on a random uniform d -ary MIN/MAX-tree of height n where the leaf-values are drawn independently from a common distribution F . The quantity $R_A(d, F) = \lim_{n \rightarrow \infty} [I_A(d, n, F)]^{1/n}$ is called the *branching factor* of algorithm A . The determination of the branching factor of the α - β pruning algorithm, $R_{\alpha-\beta}(d, F)$, was resolved by Pearl [Pe82]. Notice that $R_{\alpha-\beta}(d, F)$ does not depend on F .

Theorem 12 (Pearl) *For any continuous distribution F ,*

$$R_{\alpha-\beta}(d, F) = \frac{\xi_d}{1 - \xi_d},$$

where ξ_d is the unique positive root of the equation $x^n + x - 1 = 0$.

Proof: See [Pe82] or [Pe84]. \square

To verify that a continuous-valued MIN/MAX tree T has value $\text{val}(T) = c$, one needs to check that $\text{val}(T) \geq c$ and $\text{val}(T) \leq c$. The task of verifying the value of a continuous-valued MIN/MAX tree is equivalent to the task of evaluating two NOR-trees of identical structure. The evaluation of any NOR-tree can be seen as a part of verification of the evaluation of some continuous valued MIN/MAX tree. Since verification cannot be more complex than evaluation, the evaluation of NOR-trees cannot be more complex than the evaluation of continuous-valued MIN/MAX

trees. Therefore, Theorem 11, together with Theorem 10 and Theorem 12, establishes the asymptotic optimality of the α - β pruning algorithm for evaluating uniform MIN/MAX trees in the i.i.d. model among all deterministic MIN/MAX tree evaluation algorithms.

One can also randomize the α - β algorithm by letting the algorithm randomly select an unevaluated child to evaluate next in its depth-first search. Though we don't know whether this randomized α - β pruning algorithm is optimal among all randomized algorithms for evaluating uniform MIN/MAX trees, a randomized version of algorithm SCOUT was shown in [SW86] to possess such an optimality.

We end this section with a well-known fact about the MIN/MAX trees, which is the counterpart of Fact 2.

Fact 3 *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 number of leaves evaluated by any algorithm that evaluates T .*

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

6.4 Previous Work on Parallel Game-Tree Search

In this section we survey previous work on parallel game-tree search. As game-tree search has a prominent place in artificial intelligence, parallel game-tree search has accumulated a substantial literature and is currently an active field of research.

Most of the work on parallel evaluation of game-trees is to parallelize the α - β pruning method. An early approach is the "window decomposition" by Baudet in 1978 [Bau78]. The idea is to divide the initial search window $[-\infty, +\infty]$ into as many nonoverlapping subintervals as the number of processors available; in parallel, each processor evaluates the same input tree using each subinterval as its initial search window; the processor whose search window contains the value of the tree will return the value of the tree. It was hoped that the narrower initial search window would result in a much better speed-up for a single processor. Baudet found that this

approach offered only limited speed-up, typically, a factor of 5 or 6, regardless of the number of processors used. The reason is that no matter how narrow the search window containing the value of the input tree is, the minimal number of leaves that have to be evaluated is still large.

Later approaches are all under the general scheme of “tree decomposition”. The idea is to decompose the input tree into subtrees and let each processor work on a different subtree. This approach provides potentially unlimited parallelism. Finkel and Fishburn [FF82] proposed a method called “tree-splitting” that maps the input tree onto a tree of processors in such a way that the root of the input tree is mapped to the root-processor and the children of a node in the input tree is mapped to the children of the processor mapped to that node. As a result, each leaf-processor in the processor-tree evaluates a subtree of the input tree; when a leaf-processor finishes its subtree, it returns the value to its parent-processor which communicates this value to its other leaf-processors to update their search windows. The processor-tree is static with no processor-reallocation. The pruning effect is not considered in the construction of the processor-tree. The paper predicts a speed-up at least \sqrt{n} using n processors on uniform trees.

Akl *et al* [ABD82] proposed a method, later called “mandatory-work-first” by Finkel and Fishburn [FF83], based on the following observation: when the sequential α - β pruning algorithm evaluates a game-tree, it has to evaluate a certain “minimal tree” in any case. The idea of mandatory-work-first is to evaluate the minimal tree in parallel in the first phase; and in the second phase, the rest of the tree is evaluated using the information obtained in the first phase. This approach works well for the input tree with the “best-ordering” in which each node assumes the value of its leftmost branch, but may lose its advantage for the input tree with the worst-ordering. Another related approach is to study the behavior of the α - β pruning under the assumption that the input tree is “strongly ordered”, so that each node is like to assumes its value among its first few leftmost branches [MC82].

Parallel evaluation of AND/OR trees in connection with logic programming has also been extensively studied. A discussion of architectural issues concerning the execution of parallel logic programs can be found in [WLY85, Li85].

There are also works on parallelizing the SSS* algorithm for evaluating game trees. SSS* is a sequential best-first search algorithm for evaluating game-trees. It is as effective as the α - β method on random trees, but it requires a large amount

of space for its execution. Kumar and Kanal [KK84] have formulated SSS* as a branch-and-bound algorithm and presented two parallel implementations of SSS*.

As the field of distributed computing grows, there is a development in distributed game-tree search. The distributed program DIB of Finkel and Manber [FM87] discussed in the context of parallel backtrack search was used to perform game-tree search, but the speed-up obtained was not impressive, due to the exhaustive nature of DIB. Vornberger [Vo87b] implemented both the α - β pruning and SSS* in an asynchronous multiprocessor network in a chess application. Processors are allocated dynamically during the computation. A speed-up of 8.15 was obtained for the α - β pruning and only a speed-up of 1.9 over SSS* on a 16-processor network. The paper concluded that the α - β pruning method is much better suited for parallel processing than the method of SSS*. Recently, Vornberger *et al* [FMMV89] reported an impressive speed-up of 11.5 for the α - β pruning on a 16-processor network in a chess application. The improved speed-up is due to a new dynamic processor allocation scheme. Ferguson and Korf [FK88] developed a general distributed algorithm called Distributed Tree Search (DTS) and applied it to game-tree search. Dynamic processor allocation is used in DTS to achieve efficient load-balancing on irregular input trees. But the pruning effect was not considered when DTS is applied to the α - β pruning algorithm.

Chapter 7

Parallel Game-Tree Evaluation Algorithms

In this chapter we present some parallel game-tree evaluation algorithms. Our main algorithm is called Parallel SOLVE; it parallelizes Sequential SOLVE in a natural way. We prove that Parallel SOLVE achieves a linear speed-up on uniform trees when the number of processors used is about the height of the input tree. Parallel SOLVE is later extended to an algorithm called Parallel α - β that parallelizes the sequential α - β pruning algorithm for evaluating MIN/MAX trees.

All the parallel algorithms we present are based on the same strategy. This strategy is a general and effective paradigm for parallelizing a class of sequential tree search algorithms, and 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.

7.1 Parallel SOLVE

We first study the parallel evaluation of AND/OR trees. We present a parallel algorithm called Parallel SOLVE for evaluating AND/OR trees. Parallel SOLVE parallelizes Sequential SOLVE in a natural way and, on every instance of uniform AND/OR tree, it achieves a linear speed-up over Sequential SOLVE, when the number of processors used is close to the height of the input tree. We assume that AND/OR trees are represented as NOR-trees.

We first present another algorithm, called *Team SOLVE*, which 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 13 *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 2, Sequential SOLVE takes at least $d^{\lfloor k/2 \rfloor}$ steps to evaluate each super-leaf. The proposition follows. \square

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.

Our main contribution to the parallel evaluation of AND/OR trees is the following algorithm, called *Parallel SOLVE*, which 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. 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. Let

$T(v)$ be the subtree rooted at v . Let v be the root of the subtree to be evaluated and w the leftmost live leaf in the subtree. To illustrate the top-down view, the program P-SOLVE in Figure 7.1 describes Parallel SOLVE of width 1 on a binary NOR-tree in which P-SOLVE has v and w as its parameters.

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 13 [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 , on the order of $d \log d$, such that for any $T \in B(d, n)$ with $n > n_0$,*

$$\frac{S(T)}{P(T)} \geq cn,$$

where $c > 1/52$ and $n + 1$ is the number of processors used by Parallel SOLVE of width 1 on T .

Corollary 5 *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 , on the order of $d \log d$, such that for $n > \max\{n_0, 3\}$,*

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

where $c' < 104$.

Proof: At each step, at most $n + 1$ leaves are evaluated. So $W(T) \leq (n + 1)P(T)$. By Theorem 13, $P(T) \leq S(T)/cn$ where $c > 1/52$. Hence, $W(T) \leq 2c^{-1}S(T)$. \square

7.2 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

```

P-SOLVE( $v, w$ : node): boolean;

if  $v$  is a leaf then
    evaluate  $v$ ;
    return ( $\text{val}(v)$ );
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( $u_2, u$ );    /*finish evaluating  $T(u_2)$  in parallel*/
                return ( $1 - r$ );
        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 ( $\text{nor}(l, r)$ ).    /* "nor" is the NOR-function*/

```

Figure 7.1

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 14 *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. \square

By Proposition 14, Theorem 13 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 15 *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 , $i_0 \leq n$. Let $C(t+1) = (c'_1, c'_2, \dots, c'_n)$. Then

$$c'_{i_0} < c_{i_0}. \tag{7.1}$$

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

$$c'_1 \leq c_1, c'_2 \leq c_2, \dots, c'_{i_0-1} \leq c_{i_0-1}. \tag{7.2}$$

By (7.1) and (7.2), $C(t+1)$ 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 $R = \{i | 1 \leq i \leq n, \text{ and } c_i > 0\}$. 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_i 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 $t_{k+1}(H_T) \leq \sigma_k$. Hence, $t_{k+1}(H_T) \leq \binom{n}{k}(d-1)^k$. \square

By Proposition 15, 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 4 For $d \geq 2$, let

$$k_1 = \max\{k : \binom{n}{k} d^k \leq d^{\lceil n/2 \rceil}\}. \quad (7.3)$$

Then for $n > 13$,

$$k_1 > n/13.$$

Proof: For $x \geq 13$, $(2xe)^2 < 2^x$. Then, for $n \geq b = 13$, one can derive

$$(be)^{\lceil n/b \rceil} \leq 2^{\lceil n/2 \rceil - \lceil n/b \rceil}. \quad (7.4)$$

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

$$\binom{n}{m} \leq \left(\frac{ne}{m}\right)^m \leq (be)^m \leq d^{\lceil n/2 \rceil - m}, \quad (7.5)$$

where the last inequality is by (7.4) and the condition $d \geq 2$.

By (7.3) and (7.5), we have $k_1 \geq m \geq n/13$. \square

Lemma 5 For $d \geq 2$, let

$$k_2 = \max\{k : \sum_{i=0}^k (i+1) \binom{n}{i} (d-1)^i \leq d^{\lceil n/2 \rceil}\}. \quad (7.6)$$

Then for $n > n_0$, where n_0 is on the order of $d \log d$,

$$k_2 > n/13.$$

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,

$$\sum_{i=0}^k (i+1) \binom{n}{i} (d-1)^i < (k+1)^2 \binom{n}{k} (d-1)^k. \quad (7.7)$$

Let

$$\begin{aligned} x_0 &= \inf_{x>0} \{x | (x+1)^2 (d-1)^x \leq d^x\} \\ &= \inf_{x>0} \{x | \frac{1}{x} \log(x+1) \leq \frac{1}{2} \log(1 + \frac{1}{d-1})\}. \end{aligned} \quad (7.8)$$

Since $f(x) = \frac{1}{x} \log(x+1)$ is decreasing in $x > 0$,

$$(k+1)^2 (d-1)^k \leq d^k, \quad \text{if } k \geq x_0. \quad (7.9)$$

Let $n_0 = 13x_0 > 13$ as $x_0 > 1$. Then, by Lemma 4,

$$k_1 \geq n/13 \geq x_0, \quad \text{if } n \geq n_0. \quad (7.10)$$

By (7.7), (7.9) and (7.3), we have

$$\sum_{i=0}^{k_1} (i+1) \binom{n}{i} (d-1)^i \leq \binom{n}{k_1} d^{k_1} \leq d^{\lfloor n/2 \rfloor}. \quad (7.11)$$

Hence, by (7.6), (7.11) and (7.10),

$$k_2 \geq k_1 \geq n/13.$$

Finally, we show that $x_0 = O(d \log d)$, which implies $n_0 = O(d \log d)$. We only need consider the case that d is large. For small $x \neq 0$, $\log(1+x) \approx x$. Hence, for large d , $\log(1 + \frac{1}{d-1}) \approx 1/d$. Also, for large x , $\log(x+1) \approx \log x$. To make $2 \log x \leq x/d$, it is sufficient that $x = O(d \log d)$. Hence, by (7.8), $x_0 = O(d \log d)$. \square

Proposition 16 *For any $d \geq 2$, there is an n_0 , on the order of $d \log d$, such that for any $T \in B(d, n)$ with $n > n_0$,*

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

where $c < 52$.

Proof: The proof is a combination of Proposition 15 and Lemma 5. We have

$$P(H_T) = \sum_{i=1}^{n+1} t_i(H_T). \quad (7.12)$$

We maximize (7.12), 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 15 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 $i = 0, 1, \dots, k_0$,

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

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

where

$$k_0 = \max\{k : \sum_{i=0}^k (i+1) \binom{n}{i} (d-1)^i \leq S(T)\} \quad (7.13)$$

and x satisfies

$$\sum_{i=0}^{k_0} (i+1) \binom{n}{i} (d-1)^i + (k_0+2)x = S(T). \quad (7.14)$$

Hence, by (7.12) and (i)-(iii),

$$P(H_T) \leq \sum_{i=0}^{k_0} \binom{n}{i} (d-1)^i + \lfloor x \rfloor. \quad (7.15)$$

By Fact 2, $S(T) \geq d^{\lfloor n/2 \rfloor}$. Then, by Lemma 5 and (7.13), there is an n_0 , on the order of $d \log d$, such that for $n > n_0$,

$$k_0 > n/13. \quad (7.16)$$

Let

$$A = \sum_{i=\lfloor k_0/2 \rfloor}^{k_0} \binom{n}{i} (d-1)^i + \lfloor x \rfloor. \quad (7.17)$$

Thus, by (7.15) and (7.17),

$$P(H_T) \leq 2A. \quad (7.18)$$

By (7.14),

$$S(T) > \sum_{i=\lfloor k_0/2 \rfloor}^{k_0} (i+1) \binom{n}{k} (d-1)^i + (k_0+2)[x] \geq \frac{k_0}{2} A. \quad (7.19)$$

Hence, by (7.18), (7.19) and (7.16), for $n \geq n_0$,

$$P(H_T) \leq 2A \leq \frac{4}{k_0} S(T) \leq \frac{52 S(T)}{n}.$$

□

Theorem 13 follows immediately from Propositions 14 and 16.

The proof of Theorem 13 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 6 *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 13 holds for T for some absolute constant c .*

7.3 Parallel α - β

We turn to the problem of evaluating MIN/MAX trees in parallel. The strategy used in Parallel SOLVE applies to the evaluation of MIN/MAX trees. The resulted algorithm is called *Parallel α - β* which parallelizes the sequential α - β pruning algorithm.

We shall describe a general method for evaluating MIN/MAX trees, which includes the sequential α - β pruning algorithm and Parallel α - β as special case. 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 $\text{val}_{\tilde{T}}(v)$ denote the value of node v in \tilde{T} . This pruning process maintains the invariant property that $\text{val}_{\tilde{T}}(r) = \text{val}_T(r)$. 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 $\text{val}_{\tilde{T}}(v)$, 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 subtrees 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:

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

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

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

At each pruning step, the subtrees rooted at certain unfinished nodes are deleted by the following pruning rule.

Pruning Rules

Prune a subtree rooted at 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 14 *At any time, we have $\text{val}_{\tilde{T}}(r) = \text{val}_T(r)$. Hence, when root r is finished, the pruning process returns $\text{val}_T(r)$.*

Proof: Consider a pruning step at which a subtree H rooted at an unfinished node v is pruned from \tilde{T} . Without loss of generality, we assume that H is a maximal subtree pruned at that step and H is the only subtree pruned at that step. Let \tilde{T}' denote the tree obtained from \tilde{T} by deleting H . We want to show that $\text{val}_{\tilde{T}'}(r) = \text{val}_{\tilde{T}}(r)$.

By the pruning rule, $\alpha_{\tilde{T}}(v) \geq \beta_{\tilde{T}}(v)$. Thus, $\alpha_{\tilde{T}}(v) \neq -\infty$ and $\beta_{\tilde{T}}(v) \neq \infty$. Let u be a MAX-ancestor of v and w a MIN-ancestor of v such that $\alpha_{\tilde{T}}(v) \geq \beta_{\tilde{T}}(v)$.

$$f_{\tilde{T}}(u) \stackrel{\text{def}}{=} \max\{\text{val}_{\tilde{T}}(u') \mid u' \text{ is a finished child of } u\} = \alpha_{\tilde{T}}(v)$$

and

$$g_{\tilde{T}}(w) \stackrel{\text{def}}{=} \min\{\text{val}_{\tilde{T}}(w') \mid w' \text{ is a finished child of } w\} = \beta_{\tilde{T}}(v).$$

Hence,

$$f_{\tilde{T}}(u) \geq g_{\tilde{T}}(w). \quad (7.20)$$

By symmetry, we may assume that u is an ancestor of w . By the assumption that H is maximal, v must be a child of w . If $\text{val}_{\tilde{T}}(w) = \text{val}_{\tilde{T},(w)}$, then $\text{val}_{\tilde{T}}(r) = \text{val}_{\tilde{T},(r)}$. Assume that $\text{val}_{\tilde{T}}(w) \neq \text{val}_{\tilde{T},(w)}$. As w is a MIN-node, the deletion of a child of w would not increase the value of w . Hence,

$$\text{val}_{\tilde{T}}(w) < \text{val}_{\tilde{T},(w)}. \quad (7.21)$$

As the values of the finished children of w remain unchanged, $g_{\tilde{T}}(w) \geq \text{val}_{\tilde{T},(w)}$. Moreover, as u is a MAX-node, $\text{val}_{\tilde{T}}(u) \geq f_{\tilde{T}}(u)$. Then by (7.20),

$$\text{val}_{\tilde{T}}(u) \geq \text{val}_{\tilde{T},(w)}. \quad (7.22)$$

Let y be the MAX-ancestor of w closest to w such that $\text{val}_{\tilde{T}}(y) \geq \text{val}_{\tilde{T},(w)}$. By (7.22), y is well-defined. We show that $\text{val}_{\tilde{T}}(y) = \text{val}_{\tilde{T},(y)}$, which implies that $\text{val}_{\tilde{T}}(r) = \text{val}_{\tilde{T},(r)}$ as desired. There are two cases. Let $S(w)$ be the set of siblings of w .

Case 1: y is the parent of w . By the definition of y and (7.21), $\text{val}_{\tilde{T}}(y) > \text{val}_{\tilde{T},(w)}$. Thus, $\text{val}_{\tilde{T}}(y) = \max_{z \in S(w)} \text{val}_{\tilde{T}}(z) = \max_{z \in S(w)} \text{val}_{\tilde{T},(z)}$. Hence,

$$\text{val}_{\tilde{T},(y)} = \max\left\{\max_{z \in S(w)} \{\text{val}_{\tilde{T},(z)}, \text{val}_{\tilde{T},(w)}\}, \text{val}_{\tilde{T}}(y), \text{val}_{\tilde{T},(w)}\right\} = \text{val}_{\tilde{T}}(y),$$

where the last equality is by the definition of y .

Case 2: y is not the parent of w . Let z_0 be the child of y that is an ancestor of w . Let $t = \text{val}_{\tilde{T}}(z_0)$ and $t' = \text{val}_{\tilde{T},(z_0)}$. Let r (r') be the maximum of the value of a MAX-node between z_0 and w in T (T'). We have $t \leq r < \text{val}_{\tilde{T},(w)} \leq \text{val}_{\tilde{T}}(y)$, where the second strict inequality is by the choice of y . Hence, $\text{val}_{\tilde{T}}(y) = \max_{z \in S(w)} \text{val}_{\tilde{T}}(z) = \max_{z \in S(w)} \text{val}_{\tilde{T},(z)}$. Moreover, $t' \leq r' \leq \text{val}_{\tilde{T},(w)} \leq \text{val}_{\tilde{T}}(y)$. Thus

$$\text{val}_{\tilde{T},(y)} = \max\left\{\max_{z \in S(w)} \{\text{val}_{\tilde{T},(z)}\}, t'\right\} = \max\{\text{val}_{\tilde{T}}(y), t'\} = \text{val}_{\tilde{T}}(y). \quad \square$$

The sequential α - β pruning procedure, in the leaf-evaluation model, is the following *Sequential α - β* . The correctness of the algorithm is guaranteed by Proposition 12 and also by Theorem 14.

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 . The correctness of Parallel α - β is guaranteed by Theorem 14.

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.

One would certainly hope that Parallel α - β of width 1 would achieve a linear speed-up over Sequential α - β as the Parallel SOLVE vs. Sequential SOLVE in the case of AND/OR trees. But we are unable to prove this. The problem is that one cannot extend Proposition 14 to the MIN/MAX trees¹. Here we show what prevents such an extension. For a MIN/MAX tree T , let $\tilde{L}(T)$ be the set of leaves evaluated during the execution of Sequential α - β on T . So $\tilde{S}(T) = |\tilde{L}(T)|$. Let \tilde{H}_T denote the MIN/MAX tree obtained from T by deleting the nodes that are not ancestors of leaves in $\tilde{L}(T)$. Let $\tilde{P}_w(T)$ denote the running time of Parallel α - β of width w on a MIN/MAX tree T . The extension of Proposition 14 to the MIN/MAX trees would be the following statement: $\tilde{P}_w(T) \leq \tilde{P}_w(\tilde{H}_T)$ for any width w and any

¹We erroneously claimed such an extension in our report at the SPAA of 1989 [KZ89] and were led to conclude that Parallel α - β of width 1 achieved a linear speed-up over Sequential α - β .

MIN/MAX tree T . But this statement does not hold. In particular, it does not hold in the case of $w = 1$ which we are most interested in. The following is a small counter-example.

Counter-Example: $T \in M(2, 4)$ has 16 leaves whose values are 1, 2, 3, 4, 1, 2, 3, 4, 1, 5, 4, 3, 5, 5, 5, 5 in the left-to-right order. The root of T is a MAX-node. We show that $\tilde{P}_1(T) = 4 > \tilde{P}_1(\tilde{H}_T) = 3$.

Sequential α - β evaluates precisely those leaves that are not 5. In particular, the first leaf of 5 was pruned because the pruning condition was satisfied by the fact that its sibling leaf was evaluated to 1 and the left child of the root was evaluated to 3. Let T_1 and T_2 be the left and right subtrees of T , respectively. Let \tilde{H}_T^1 and \tilde{H}_T^2 be the left and right subtrees of \tilde{H}_T , respectively. We have $T_1 = \tilde{H}_T^1$. Let A be the algorithm Parallel α - β of width 1. A evaluates both T_1 and \tilde{H}_T^1 in 3 steps. \tilde{H}_T^2 has only 3 leaves. Hence, A evaluates \tilde{H}_T in 3 steps. On the other hand, during the evaluation of T by A , the first 3 leaves of T_2 , which are 1, 5, 4, are evaluated sequentially after T_1 is evaluated. The leaf of 5 was evaluated in this case because that T_1 was not evaluated yet to provide the pruning information. After 3 steps, T_2 is not evaluated yet and can not be pruned either. One more step is needed to complete the evaluation of T . So T is evaluated in 4 steps. \square

The above example shows what goes wrong. When Parallel α - β of width 1 evaluates T , the first leaf of 5 was evaluated because the pruning information, the left child of the root is evaluated to 3, is not available yet. The evaluation of that leaf delayed the whole evaluation by one step. In general, when Parallel α - β evaluates a MIN/MAX tree T , a subtree T' of T that are not in \tilde{H}_T may be evaluated until the information for pruning T' is obtained. But this information may come from the nodes that are at the top of the path from the root of T to T' and may not be obtained until the algorithm has already spent much time in T' . In other words, the algorithm may “trap” in T' and fail to make the progress it would make otherwise in \tilde{H}_T . On contrast, the pruning condition in case of AND/OR trees is “local”, which eliminates the problem.

In the case of AND/OR trees, Proposition 14 allows us to consider H_T instead of T so that the total work of Parallel SOLVE is at most that of Sequential SOLVE with respect to the evaluation of H_T . The key point is the ratio of the total work of the parallel algorithm and that of the sequential counterpart. From the proof of the Main Theorem, one can see that Parallel α - β of width 1 would achieve a linear

speed-up over the Sequential α - β if the following conjecture holds. Conjecture:

Let $W_1(T)$ be the total work of Parallel α - β of width 1 in evaluating T . Let $|\widetilde{H}_T|$ be the number of leaves in \widetilde{H}_T . Then there is an absolute constant c such that for every $T \in M(d, n)$,

$$W_1(T) \leq c|\widetilde{H}_T|.$$

We have tested this conjecture on some randomly generated game trees. The results of our tests, though limited, showed supporting evidence to the conjecture. We generated uniform binary trees T in which the leaf-values were drawn from the uniform distribution over certain range. The ratio of $W_1(T)$ over $|\widetilde{H}_T|$ quickly converged to somewhere in the interval between 1.6 and 1.7 as the height of the tree increased to 30.

7.4 Node-Expansion Model

The algorithms Parallel SOLVE and Parallel α - β can be extended in several ways. In this and the subsequent sections we shall discuss these extensions. We shall only consider the evaluation of AND/OR trees. The extensions to the MIN/MAX trees can be extrapolated accordingly.

So far we have been using the leaf-evaluation model in presenting and analyzing our algorithms. In this section we consider the node-expansion model introduced in Section 6.2. We shall 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 13 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 7.7, 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 program P-SOLVE*(v, g) in Figure 7.4 describes N-Parallel SOLVE of width 1 on a binary NOR-tree. P-SOLVE*(v, g) is similar to program P-SOLVE in Section 7.1. 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. Initially, g consists of only v . We assume that g carries with it the right-siblings of the nodes on g .

The following theorem is the counterpart of Theorem 13 in the node-expansion model.

Theorem 15 *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*

```

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
            return(1 - P-SOLVE*(u, g));
        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*( $u_2$ );
        return (1 - P-SOLVE*( $u_2$ , g));
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)).

```

Figure 7.4

on d , such that for any $T \in M(d, n)$ with $n > n_0$,

$$\frac{S^*(T)}{P^*(T)} \geq cn,$$

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 15 goes as that of Theorem 13. It can be easily checked that the counterpart of Proposition 14 in the node-expansion model holds without change. The only part in the proof that needs to be changed is Proposition 15. 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 7.2 consists of precisely those nodes of T that are expanded by N-Sequential SOLVE on T .

Proposition 17 For any $T \in B(d, n)$,

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

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

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 15, 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 17 is larger than the one in Proposition 15 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 15 holds.

7.5 Fixed Number of Processors

In this section we present a modified N-Parallel SOLVE in which the number of processors available is fixed. We shall use the terminology of Section 7.4.

There is a natural way to modify N-Parallel SOLVE of width 1 when only a fixed number of processors is available. Let F_1 be the set of frontier nodes whose pruning numbers are less than or equal to 1. All nodes in F_1 would be evaluated by N-Parallel SOLVE of width 1 at the next step. In case that there are only p processors available, at most p nodes in F_1 are evaluated at next step.

N-Parallel SOLVE with p processors

If $|F_1| \leq p$, evaluate all nodes in F_1 ; otherwise, evaluate the leftmost node in F_1 and $p - 1$ rightmost nodes in F_1 .

Let w be the leftmost frontier node. The *base path* is the path from the root to w . A *right-hanging subtree* is a subtree rooted at a right-sibling a node on the base path. A right-hanging subtree is *active* if it contains a frontier node of pruning number 1. The structure of N-Parallel SOLVE with p processors can be described as follows: let one processor perform N-Sequential SOLVE on each active hanging-subtree along the base-path from the root downwards until either (i) there is only one processor left, in that case, let the last processor expands w or (ii) there are more than one processor left, in that case, let any processor evaluate w . In other words, allocate the processors off the base path as the tree is being generated: when there is only one processor left, let the last processor evaluate the remaining subtree by itself.

To describe N-Parallel SOLVE with p processors in code, we will use the procedures S-SOLVE* and P-SOLVE* of Section 7.4 with some straightforward modifications such as adding a parameter to P-SOLVE* to indicate how many processors available to P-SOLVE*. We shall not omit the exact code.

The following theorem states that N-Parallel SOLVE with p processors achieves a linear speed-up in p over N-Sequential SOLVE on uniform trees.

Theorem 16 *For a NOR-tree T , let $P^*(T, p)$ be the number of steps N-Parallel SOLVE with p processors takes to evaluate T . Then, for any $d \geq 2$, there is an n_0 , depending on d , such that for any $T \in B(d, n)$ with $n > n_0$,*

$$\frac{S^*(T)}{P^*(T, p)} \geq cp,$$

where $c > 0$ is an absolute constant.

Proof: First of all, one can check that the counterpart of Proposition 14 holds for N-Parallel SOLVE with p processors. We need to a counterpart of Proposition 16.

Let $t_k^*(T, p)$ denote the number of steps of parallel degree k during the execution of N-Parallel SOLVE with p processors on T . By the same argument of Proposition 15 and Proposition 17, we have that for any $T \in B(d, n)$ and $1 \leq p \leq n + 1$, $t_{k+1}^*(H_T, p) \leq (n - k) \binom{n}{k} (d - 1)^k$ for $1 \leq k \leq p$. This is enough to prove the theorem by the argument of Proposition 16. \square

There does not seem to be a similar modification to N-Parallel SOLVE of widths greater than 1, given that there are fixed number of processors. This probably does not matter in practice. When the number of processors is limited, the width-one algorithm would be the choice anyway.

7.6 Randomization

In this section we briefly discuss the randomization of Parallel SOLVE. We assume the node-expansion model.

We have already described the randomized Sequential SOLVE in an early section. The description was top-down. So it would be more natural to randomize the node-expansion version of the Sequential SOLVE, namely, the N-Sequential SOLVE. The randomized N-Sequential SOLVE, called *R-Sequential SOLVE*, is as follows: 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. 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 the 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 17 *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 $n > n_0$,

$$\frac{E(S_R^*(T))}{E(P_R^*(T))} \geq cn.$$

where $c > 0$ is an absolute constant.

Proof: Follows from Theorem 15 by averaging. \square

7.7 Implementations

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 this section we discuss these implementational issues in the context of implementing the algorithms N-Parallel SOLVE of width 1 and N-Parallel SOLVE with p processors.

We propose an implementation of N-Parallel SOLVE of width 1 by the “level-allocation” method in which each processor is in charge of one level of the input tree. This method has the advantage of conceptual simplicity. But it is inflexible when the number of processors is fixed, and it works at its best under the assumption that each message is delivered in unit time. As a complement to the level-allocation method, we propose an implementation of N-Parallel SOLVE with p processors by the “dynamic-allocation” method in which only p processors are used. The dynamic-allocation method can also be adapted to a distributed environment where no assumption on message-delivery is made. Both implementations avoid passing complex data structures and maintain the linear speed-up of the original algorithms. For convenience in exposition we restrict ourselves to the case that the input NOR-tree is binary, i.e., each internal node has exactly two children.

In our implementations, the procedure S-SOLVE* of Section 7.4 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.

7.7.1 Level-Allocation Method

In this section we describe the level-allocation method for implementing N-Parallel SOLVE of width 1.

We shall implement procedure P-SOLVE* of Section 7.4 differently from its original description. In its original 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), $\text{val}(v) = 1$ and $\text{val}(v) = 0$. 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 $\text{val}(v)$ 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 “ $\text{val}(v) = 0$ ” to processor $d(v) - 1$ and halts; if the first message it receives is “ $\text{val}(w) = 0$ ” then it sends the message “P-SOLVE*(x)” to processor $d(v) + 1$. As soon as it has received the two messages “ $\text{val}(w) = 0$ ” and “ $\text{val}(x) = 0$ ” it sends the message “ $\text{val}(v) = 1$ ” 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 $S\text{-SOLVE}^*(v)$ 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 " $P\text{-SOLVE}^{**}(u)$ " to processor $d(u)$ and the message " $S\text{-SOLVE}^*(x)$ " 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 " $P\text{-SOLVE}^{***}(u)$ " to processor $d(u)$. If u is the terminal node of the path g then it sends the message " $P\text{-SOLVE}^*(u)$ " to processor $d(u)$. When the traversal of the path is complete and all the required messages have been sent, the execution of $P\text{-SOLVE}^*(v)$ terminates.

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

The message " $P\text{-SOLVE}^{***}(v)$ " is similar to " $P\text{-SOLVE}^{**}(v)$ " 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 " $\text{val}(x) = b$," where x is the right child of v . Upon receipt of this message it sends the message " $\text{val}(v) = 1 - b$ " to processor $d(v) - 1$.

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

As it is, the level-allocation method uses as many processors as the level of the input tree. One can simulate the method using p processors by dividing levels of the input trees into "zones" of p consecutive levels, and letting processor d be responsible for level d in each zone. To simulate, each processor has to divide its attention to its levels in different zones by multiplexing, which may not be efficient in practice.

We show that the "pre-emption" rule achieves the correct pruning behavior given the assumption that message-delivery takes unit of time. For processor d , it only receives messages containing nodes of the level d . We say that processor d receives

messages of type S-SOLVE*() in the *left-to-right* order if whenever processor d receives a message S-SOLVE*(v) and later receives a message S-SOLVE*(u) of the same type, then v is to the left of u at level d of the tree. The correct pruning behavior is ensured if each processor receives the messages of the same type in the left-to-right order. We show that this is indeed the case. The major case that needs to be checked is the case that processor $d(v)$ receives the message "P-SOLVE*(v)" after it has already received the message "S-SOLVE*(v)". In that case, processor $d(v)$ walks down its path g and sends messages to each of the processors at the levels along the path until reaching the end of the path. Among those processors receiving messages from processor $d(v)$, processor i will receive its messages before processor j does, if $j > i$, by the assumption that message-delivery takes unit of time. So processor i works on a node that is on the right of path g before processor j processes the message containing node w received from processor $d(v)$. Consequently, it is not possible for processor i to send a message containing node w' to processor j afterwards such that w' is to the left of w .

The assumption of unit-time message-delivery is not necessary if we are willing to pass in the message the *address* of a node, which is the binary string coding the path from the root of the input tree to the node. Using the address, a processor can ensure the correct pruning behavior by always working on the right-most node on its level received so far. The address-passing allows the level-allocation method to be implemented in a distributed environment.

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 17 of Section 7.4. Consequently, the conclusion of Theorem 15 holds for N-Parallel SOLVE, implemented as described. With each time step τ we associate a "base path" as follows. Let $X(\tau) = \{y \mid \text{P-SOLVE}^*(y) \text{ is being executed at time } \tau\}$. 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 17. By these properties, the base paths associated with the steps of the traversal will not increase the bounds stated in Proposition 17. One can conclude that our implementation does not compromise the linear speed-up of N-Parallel SOLVE over N-Sequential SOLVE.

7.7.2 Dynamic-Allocation Method

In this section we describe the dynamic-allocation method for implementing N-Parallel SOLVE with p processors. We shall mainly focus on the structure of the method rather than the details. A more detailed account of the method is contained in [YZ89].

Central to the dynamic-allocation method is a linear ordering of the p processors. Each processor that has a subtree to evaluate executes the procedure S-SOLVE*. A processor is *working* if it is evaluating some subtree; otherwise, it is *free*. N-Parallel SOLVE with p processors imposes a natural ordering on the working processors: the processor evaluating the top active right-hanging subtree of the base-path is the first in the ordering and the processor evaluating the end of the base-path is the last. We append the free processors, if any, to the end of the ordering of the working processors in an arbitrary way to form a complete linear ordering of p processors. We shall organize the p processors as a linked list, a "processor structure", according to this linear ordering. This linked list is maintained dynamically. Interprocessor communication occurs only between consecutive processors in the linked list.

The last working processor in the linked list has a special significance. It maintains the trailing part of the base path and can give away subtrees off the base-path to other free processors. Because of its role of giving away subtrees, we call this processor the *donor*. A free processor asks the donor for new work. The donor relinquishes the topmost right-hanging subtree for the requesting free processor, which subsequently becomes the parent of the donor in the linked list. For example, if processors A, B, C and D enter a computation alphabetically with A as the starting processor, the resulting order will be B-C-D-A, with B evaluating the top right-hanging subtree and A being the donor. After A donates to another free processor E, E becomes the parent of A and the child of D, and A remains as the donor. We observe that such a work assignment avoids passing complex data structures such

as the trailing part of the base path.

When a processor other than the donor finishes its subtree, its action depends on the value of its subtree. If the value is 1, then it reports the value to its parent and waits for new work; and in this case, all the subtrees of the processors below that processor should be pruned. If the value is 0, then that processor should “swap” down along the linked list to reach the donor for new work, and if it gets new work, it becomes the parent of the donor as indicated before. When a child-processor reports a value to its parent-processor, the parent interprets the value according to the parity of the distance between the subtree of the parent and the subtree of the child. The work assignments contain the parity information.

When the donor finishes its subtree, the base path moves to some right-hanging subtree along the base path. The new donor is the processor possessing that right-hanging subtree. The trailing section of the base-path now coincides with the local sequential search path of the new donor. The new donor walks down its local search path and gives away the right-hanging subtrees of its local search path as the new work assignments to the processors below it along the linked list; at the end of this process, the new donor becomes the child of the last processor to which it gives work, and it maintains the rest of the trailing part of the base path. In other words, the donor “swaps” downwards along the linked list of processors to give away new work assignments to the processors encountered. No complex data structure is passed in this process. For example, let $A-B-C-D$ be the linked list where D is the donor. Suppose that the subtree containing C and D is finished and the base path moves into the subtree of B . Then B becomes the new donor and it donates to C and D . The linked list becomes $A-C-D-B$ with B remaining the donor.

The problem with this method is that a processor cannot know whether it should take the role of the new donor without global knowledge of the computation. A processor can potentially be a new donor if the processor itself did not finish its subtree and its child-processor did. Our strategy is to let all such processors assume the role of the new donor. So there can be multiple donors at one time. As we have described, a processor gets a new subtree by either seeking a donor when it finishes its subtree or receiving a subtree from a donor. In the latter case, if the processor is free, it starts to evaluate the received subtree; if the processor is working, it has to decide whether the received subtree is “valid”, because of multiple donors, and if the subtree is valid, the processor abandons or “prunes” its current subtree and

works on the received subtree. We shall discuss how to decide whether a received subtree is valid.

To ensure a proper pruning behavior, a working processor needs a “pre-emption” rule to decide whether to prune its current subtree when it receives a new subtree. This pre-emption rule depends on our assumption about message-delivery. The simplest situation is to pass the absolute address of the subtree in the message. In this case, we can use the “rightmost subtree” rule: always work on the lexicographically rightmost subtree received so far. This clearly ensures the proper pruning behavior in any circumstances. Suppose that we are unwilling to pass the absolute address of a subtree in the consideration of efficiency and instead require that a message be delivered in unit time. In this case, we can use the “most-recent” rule: always work on the latest received subtree. A proof of the correctness of this rule follows the same line as the one for the level-allocation method. Suppose that A is above B in the linked list. If a processor receives a subtree from A at time t_A and a subtree from B at time t_B , then it is fairly clear that $t_A \leq t_B$. But it could happen that $t_A = t_B$ if we are not careful. To avoid this, we require that when a processor “swaps” downwards on the linked list of processors to give the swapped processors new work, it should give the work to a processor *before* swapping with that processor. This ensures that a processor will not receive two subtrees at the same step.

We now consider the distributed environment in which processors operate asynchronously and there are no assumptions on message-delivery. In this case, races can occur. A new subtree can arrive at a processor earlier than some older subtrees. Hence, proper synchronization is crucial. The address-passing can solve the problem but we assume no address-passing. The linked list structure of the processors gives us an additional parameter of a processor, namely, the “rank”. The *rank* of processor A is the number of processors preceding A in the linked list. It turns out that the rank is a convenient mechanism for synchronization. With the proper protocols, we can use the “lower-rank” rule: always work on the subtree received from a processor of a lower rank. A message contains the rank of the sender, which is $O(\log p)$ bits long. We shall not elaborate on the details of the proper protocols. The mechanism of rank can also solve other synchronization problems that arise in the distributed implementation.

There are two sources of major additional delays. The first is the situation in

which the topmost donor walks down its local search path as it assigns new work to other processors. This traversal is considered as instantaneous in the node-expansion model. This type of delay was analyzed in the level-allocation method and was shown to be incorporated into the path-counting argument.

The second source of delay is the situation in which a non-donor processor has evaluated its subtree to 0 and subsequently walks down the linked list to seek the donor for new work. This traversal is considered as instantaneous in the node-expansion model. These delays cause an increase in the number of “bad” steps. But we show that we can still use the upper bound given in the proof of Proposition 17 of Section 7.4. Let A be a non-donor processor that has evaluated its subtree T_A to 0. Let T'_A be the (left) sibling subtree of T_A . In the same effect as using the “coding” of the base path, we reflect the trailing part of the base path that lies in T'_A into the subtree T_A . As A walks down the base path, we associate with the current step the path from the root to where A is, and regard this new path as the base path of that step. This base path has the following properties: (i) the number of working processors at that step is at least as large as the parallel degree of the path in the node-expansion model; (ii) it has not been counted before and will not be counted later as a base path; (iii) it is a path counted in Proposition 17. Notice that property (i) means that the step is counted as a step with *worse* parallel degree. These properties imply that we can still use the bounds stated in Proposition 17 to reach the same conclusion of Theorem 16 for our implementation as for the original algorithm.

7.8 Open Problems and Further Research

There are many open problems and possible further topics to pursue. We list a number of them.

1. The main open problem is the conjecture that the work of Parallel α - β of width 1 is within a constant factor of the work of Sequential α - β on uniform MIN/MAX trees. This conjecture would imply a linear speed-up of Parallel α - β of width 1 over Sequential α - β on uniform MIN/MAX trees.
2. The major weakness of Theorem 13 is that linear speed-up is proved only in the case of width 1. When the width is 2 or 3, the number of processors used

on a uniform tree of height n is $O(n^2)$ and $O(n^3)$, respectively. We believe that the speed-up of Parallel SOLVE on uniform trees should remain linear in the number of processors used for any fixed width. It appears that new proof techniques are needed for the analysis of higher widths.

3. Our results are asymptotic in the height of the input tree. Theorem 13 requires the height of a d -ary uniform tree to be on the order of $d \log d$. This should be contrasted with the “wide-and-shallow” game trees encountered in chess programs, which have relatively large branching factor and limited depth.
4. In Theorem 13 the provable lower bound on constant c in the linear speed-up ratio is small ($c > 1/52$). Some simulations we did indicate that a larger lower bound on c is achievable. It would be highly desirable to establish this theoretically.
5. We have suggested two implementations of the width-one algorithms on NOR-trees. There is no conceptual difficulty to modify these implementations for MIN/MAX trees. But it is not clear how to extend these implementations to the higher widths.
6. There is almost a total lack of experimental results on the algorithms presented. To try out these algorithms on real applications would give valuable evaluations of the practical usefulness of these algorithms. We did a preliminary testing of a program implementing the dynamic allocation scheme on a local network of SUN's, and found that there was significant communication overhead [YZ89]. We suspect that our program would perform better in tightly coupled multiprocessing environments.
7. The evaluation of AND/OR trees is closely related to the execution of logic programming languages such as Prolog. However, parallel execution of Prolog programs gives rise to “binding conflicts”. It is not yet clear how useful Parallel SOLVE could be for parallel execution of Prolog programs.

Bibliography

- [ABD82] S.G. Akl, D.T. Bernard and R.J. Doran, Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm, *Proceedings of the 1980 international Conference on Parallel Processing*, IEEE, New York, 1980, 231-234.
- [Alth89] I. Althöfer, A parallel game tree search algorithm with a linear speedup, personal communication, 1989.
- [AV79] D. Angluin, L.G. Valiant, Fast probabilistic algorithms for Hamiltonian circuits and matchings, *J. Comput. System Sci.*, **19** (1979), 155-193.
- [Ba85] E. Balas, Branch and bound methods, in *The Traveling Salesman Problem*, edited by E.L. Lawler *et al*, John Wiley & Sons, 1985.
- [Bau78] G. Baudet, The design and analysis of algorithms for asynchronous multiprocessors, Computer Science Tech. Rept. CMU-CS-78-116, Carnegie-Mellon University, Pittsburgh, PA, 1978.
- [BRT87] A. de Bruin, A.H.G. Rinnooy Kan and H.W.J.M. Trienekens, A Simulation tool for the performance evaluation of parallel branch and bound algorithms, to appear in *Mathematical Programming*, June 1987.
- [CSW85] L. Carter, L. Stockmeyer and M. Wegman, The complexity of backtrack searches, *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, 1985, 449-457.
- [Do53] J.L. Doob, *Stochastic Process*, John Wiley & Sons, 1953.
- [FK88] C. Ferguson and R. Korf, Distributed tree search and its application to alpha-beta pruning, *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988 (AAAI-88).

- [FF82] R.A. Finkel and J.P. Fishburn, Parallelism in alpha-beta search, *Artificial Intelligence*, 19 (1982), 89-106.
- [FF83] R.A. Finkel and J.P. Fishburn, Improved speedup bounds for parallel alpha-beta search, *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-5(1), 1983.
- [FM87] R.A. Finkel and U. Manber, DIB — A distributed implementation of backtracking, *ACM Transactions on Programming Languages and Systems*, 9 (1987), 235-256.
- [FMMV89] Feldmann, Monien, Myslewietz and Vornberger, Distributed game-tree search, *International Computer Chess Association Journal*, 12(2), June 1989.
- [Ga76] Z. Galil, On enumeration procedures for theorem proving and for integer programming, *Automata languages and Programming*, edited by S. Michaelson and R. Milner, 355-382, 1976.
- [Gra87] T. Grandine, Some recent experiences in solving the asymmetric traveling salesman problem, Engineering Technology Applications Report ETA-TR-46, Boeing Computer Services, Seattle, Washington, March 1987.
- [IYF79] M. Imai, T. Fukumura and Y. Yoshida, A parallelized branch-and-bound algorithm: implementation and efficiency, *System Computer Controls*, 10(3), 1979, 62-70.
- [KSW86] R.M. Karp, M. Saks and A. Wigderson, On a search problem related to branch-and-bound procedures, *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 19-28, 1986.
- [KZ88] R.M. Karp and Yanjun Zhang, A randomized parallel branch-and-bound procedure, *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, 1988, 290-300.
- [KZ89] R.M. Karp and Yanjun Zhang, On parallel evaluation of game trees, *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 1989, 409-420.
- [KM75] D.E. Knuth and R.N. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence*, 6 (1975), 293-326.

- [KK84] V. Kumar, L.N. Kanal, Parallel branch-and-bound formulations for AND/OR tree search, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-6(6), Nov. 1984.
- [KRR88] V. Kumar, K. Ramesh and V.N. Rao, Parallel best-first search of state-space graphs: a summary of results, *Proceedings of the 1988 National Conference on Artificial Intelligence* (AAAI-88).
- [LS84] T.H. Lai and S. Sahni, Anomalies in paralel branch-and-bound algorithms, *Communications of the ACM*, 27 (1984), 594-602.
- [LW66] E.L. Lawler and D.E. Wood, Branch-and-bound methods: a survey, *Operations Research*, 14 (1966) 699-719.
- [Li85] G. Li, Parallel Processing of Combinatorial Search Problems, Ph.D. thesis, Department of Computer Science and Engineering, Purdue University, 1985.
- [Li85] G. Li and B. W. Wah, Computational efficiency of parallel approximate branch-and-bound algorithms, *Proceedings of the 1984 International Conference on Parallel Processing*, IEEE, New York, 1984.
- [Ma86] U. Manber, On maintaining dynamic information in a concurrent environment, *SIAM J. Comput.* 15 (1986), 1130-1142.
- [MC82] T.A. Marsland and M. Campbell, Parallel search of strongly ordered game trees, *Computing Survey* 14(4) (1982), 533-551.
- [PS82] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.
- [Pe80] J. Pearl, Asymptotic properties of minimax trees and game-searching procedures, *Artificial Intelligence*, 14(2) (1980) 113-138.
- [Pe82] J. Pearl, The solution for the branching factor of the alpha-beta pruning algorithm and its optimality, *Communication of ACM* 25(8) 1982, 559-564.
- [Pe84] J. Pearl, *Heuristics*, Addison-Wesley, 1984.
- [Ri88] R. L. Rivest, Game tree searching by min/max approximation, *Artificial Intelligence* 34 (1988) 77-96.

- [Ra89] A. Ranade, personal communication, 1989.
- [RND77] E. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice Hall, 1977.
- [SW86] M. Saks and A. Wigderson, Probabilistic Boolean Decision Trees and the Complexity of Evaluating, *27th Annual Symposium on Foundations of Computer Science*, 29-38, 1986.
- [SS87] E. Shamir and J. Spencer, Sharp Concentration of the Chromatic Number on Random Graphs $G_{n,p}$, *Combinatorica* 7(1), 1987.
- [Ta83] M. Tarsi, Optimal Search on Some Game Trees, *Journal of ACM* 30 (1983), 389-396.
- [Sto79] G.C. Stockman, A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence* 12 (1979), 179-196.
- [Vo87a] O. Vornberger, Parallel alpha-beta versus parallel SSS*, *Proceedings of the IFIP Conference on Distributed Processing*, Amsterdam, 1987.
- [Vo87b] O. Vornberger, Load Balancing in a Network of Transputers, *2nd International Workshop on Distributed Algorithms*, Amsterdam, July 1987.
- [YZ89] J. Yang, Y. Zhang, A distributed implementation of a game-tree evaluation algorithm, *manuscript*, May 1989.
- [WLY85] B.W. Wah, G. Li, C.F. Yu, Multiprocessing of Combinatorial Search Problems, *IEEE Computer*, June 1985.
- [WM84] B.W. Wah and E. Ma, MANIP— A multicomputer architecture for solving combinatorial extremum search problems, *IEEE Trans. Computers*, Vol. C-33, No.5, May 1984, 377-390.

