Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation

RICHARD M. KARP AND YANJUN ZHANG

University of California at Berkeley, Berkeley, California

Abstract. Universal randomized methods for parallelizing sequential backtrack search and branch-and-bound computation are presented. These methods execute on message-passing multiprocessor systems, and require no global data structures or complex communication protocols. For backtrack search, it is shown that, uniformly on all instances, the method described in this paper is likely to yield a speed-up within a small constant factor from optimal, when all solutions to the problem instance are required. For branch-and-bound computation, it is shown 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. These randomized methods demonstrate the effectiveness of randomization in distributed parallel computation.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms—*computation on discrete structures*; *sorting and searching*; F.1.2 [**Computation by Abstract Devices**]: Modes of computation—*parallelism and concurrency*; *probabilistic computation*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms*; G.3 [**Probability and Statistics**]: *probabilistic algorithms*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*backtracking*; *graph and tree search strategies*

General Terms: Algorithms

Additional Key Words and Phrases: Backtrack search, branch-and-bound, distributed parallel computation

1. *Introduction*

A *combinatorial search problem*, or simply *search problem*, is a problem of finding certain arrangements of some combinatorial objects among a large set of possible arrangements. Typical examples of search problems are enumerating the satisfying assignments of a Boolean formula or finding a minimum-cost tour through a set of cities. Computational resources required for solving search problems tend to grow exponentially in the size of problem instance; the explosive complexity limits the range of search problems that are solvable in practice.

Search problems are well suited for parallel computation—the set of possible arrangements can be searched simultaneously. Potential speed-up by paral-

The research for this paper was supported by National Science Foundation (NSF) grant DCR 84-11954 and by the International Computer Science Institute, Berkeley, California.

Authors' present addresses: R. M. Karp, Computer Science Division, University of California at Berkeley, Berkeley, CA 94720; Y. Zhang, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275

lel computation in solving search problems can be great. The study of parallel algorithms for solving search problems is of theoretical and practical significance.

In this paper, we study the parallel execution of two fundamental search methods: backtrack search and branch-and-bound computation. We present universal randomized methods for parallelizing sequential backtracking search and branch-and-bound computation. These methods execute on message-passing multiprocessor systems, and require no global data structures or complex communication protocols. For backtrack search we show that, uniformly on all instances, our method is likely to yield a speed-up within a small constant factor from optimal, when all solutions to the problem instance are required. For branch-and-bound computation, we show that, uniformly on all instances, the execution time of our method is unlikely to exceed a certain inherent lower bound by more than a constant factor. These randomized methods demonstrate the effectiveness of randomization in distributed parallel computation.

We shall formulate a search problem as a *tree search*. We assume that an algorithm for solving a certain search problem is given. This algorithm has a certain procedure for generating subproblems. When this procedure is applied to a subproblem A, it either solves A directly or derives from A a set of subproblems $A_1, A_2, \ldots, A_d$ such that the solution of A can be found from the solutions of $A_1, A_2, \ldots, A_d$. Given a problem instance, we associate with it a rooted tree in which the root of the tree corresponds to the given problem instance, an internal node corresponds to a subproblem of the given problem instance, the children of an internal node correspond to the set of subproblems derived from the subproblem represented by that internal node, and a leaf corresponds a subproblem that can be solved directly. The execution of the given algorithm corresponds to a search in the tree associated with the problem instance. The nodes of the tree are generated by the *node expansion* operation that, when applied to node v, 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 generate the nodes of the tree until a leaf or a set of leaves are identified as the solution. The execution time of a sequential search is the number of node expansions, all other computation being considered free.

We are interested in executing the tree search in parallel. Our model of parallel computation is a message-passing multiprocessor system with p processors. We require that at most p nodes are expanded at a single step. We assume that there is no global memory, and that a processor can send a message to any other processor in unit time, where a message has sufficient capacity to contain whatever the information is required for a node expansion operation. The processors are regarded as synchronous for the purpose of counting steps, but the algorithms themselves do not require global synchronization. The execution time of a parallel search is the number of steps at which node expansions are performed, all other computations being considered free. Given a problem instance, 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 instance. Our goal is to find parallel algorithms that achieve a speed-up close to the number of processors used, and yet are efficiently implementable in the chosen model of parallel machines.

2. *Backtrack Search*

Problems that yield to backtrack search have the property that it is possible to determine that some initial choices cannot lead to a solution. This property allows the search to terminate an unproductive exploration and then "backtrack" to a point where a new search can be started. An example of backtrack search is a way 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 cannot 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 previously attempted. This process is repeated until a solution is found or it is found that there is no solution. A backtrack search can be viewed as a search through a tree of partial solutions in which the root represents the empty solution, an internal node represents a partial solution, the children of an internal node v represent all possible minimal extensions to the partial solution represented by v, and a leaf of this tree represents either a solution or a partial solution that cannot be extended to a solution. The execution of a sequential backtrack search corresponds to a depth-first search in this tree using the node expansion operation. A study on the complexity of backtrack search can be found in [4].

In this paper, we shall consider the *all-solution* backtrack search that seeks all solutions to the given problem instance. By backtrack search, we shall always mean all-solution backtrack search. The fundamental property of all-solution backtrack search is that its corresponding tree search must expand every node of the tree. This is because the backtrack search cannot tell whether a partial solution can be extended to a solution until it determines that the partial solution corresponds to a leaf of the tree. As all nodes of the tree must be expanded, each node expansion performed is a useful work. An all-solution parallel backtrack search algorithm will achieve a good speed-up if processors are kept busy performing node expansions.

Given a backtrack search algorithm, let H be the rooted tree of partial solutions associated with the algorithm on some problem instance. Consider a parallel search algorithm that generates 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. The execution time of any algorithm that generates H 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 that generates H is at least n/p, since all n nodes in H must be expanded and the algorithm can expand at most p nodes at one step. Therefore, $\max\{n/p, h\}$ is an inherent lower bound on the execution time of any p-processor algorithm that generates H. Our goal is to find p-processor backtrack search algorithms whose execution time comes close to this lower bound.

3. *Parallel Backtrack Search*

In this section, we study parallel backtrack search. We give a generic description of the parallel backtrack search algorithms we wish to study. We then describe two specific algorithms, one deterministic and the other randomized;

the deterministic algorithm requires global control whereas the randomized one does not.

3.1. GENERIC PARALLEL BACKTRACK SEARCH ALGORITHMS. A *frontier node* is a node that has been generated but not expanded. The frontier nodes are distributed among the processors, with each belonging to exactly one processor. Let P_i denote processor i. The *local frontier* of P_i, denoted by F_i, is the set of frontier nodes possessed by P_i. A processor is *busy* if its local frontier is nonempty; otherwise, it is *idle*. A busy processor is *loaded* if its local frontier contains two or more nodes. The *level* of a node v is the number of nodes on the path from the root to v inclusively. A *top-node* of P_i is a node of minimum level among all nodes 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. For two nodes v and u of a tree that do not lie on the same root-leaf path, v is *to the left of* u if v is visited before u is visited in a depth-first traversal of the tree in which the children of a node are visited according to the ordering of the children.

A generic parallel backtrack search algorithm is given in Figure 1. 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 *pairing step* in which a set of loaded processors are designated as donating processors, each having a distinct idle processor as its receiving processor, and a *donation step* in which each donating processor transfers half of its top-nodes to its receiving processor. At each pairing step, the set R of pairs (i, j) such that processor i donates to processor j is called the *pairing set*.

By the description in Figure 1, a busy processor performs a depth-first traversal in some subtree, and finishes the traversal, with a possibility of donating some parts of the subtree to other processors, before it starts to traverse another subtree. The local frontier of each processor can be conveniently maintained by a stack. A processor with a nonempty stack removes the top node of its stack, expands it, and pushes its children, if any, onto the stack in the reversed order of the children. The following conditions are required for donation: (a) only idle processors may receive donations; (b) only loaded 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, and (e) a donating processor donates half of its top-nodes. Among these conditions, condition (e) of donating top-nodes is the essential rule for donation. Condition (d) guarantees that the top-nodes of a processor are consecutive siblings and appear at the bottom of the stack. A processor donates its top-nodes by removing them from the bottom of its stack; the donation message permits a succinct description of the donation as the donated top-nodes are consecutive siblings.

The code in Figure 1 omits the details of terminating the computation after H is completely generated. The computation can stop as soon as each processor learns that all local frontiers are empty. This objective can be achieved by scheduling occasional broadcast phases, in which the processors configure themselves into a uniform binary tree of preassigned structure. 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 the broadcast steps can be so chosen that these steps have no appreciable influence on the overall execution time.

```
Generic Parallel Backtrack Search
/*Initialization*/
F_1 = {r};
for i = 2,3,...,p, F_i ← ∅;
while some F_i ≠ ∅ do
  /*Node Expansion Step*/
  for i = 1,2,...,p in parallel do
    if F_i ≠ ∅ then
      let v_i be the leftmost node in F_i;
      expand v_i;
      F_i ← F_i \ {v_i};
      if v_i is not a leaf then F_i ← F_i ∪ Γ(v_i);
/*Pairing Step*/
determine a pairing set
R = {(i,j) | |F_i| > 1, |F_j| = 0, 1 ≤ i,j ≤ p} such that
if (i,j), (i',j') ∈ R, then either i = i', j = j' or i ≠ i', j ≠ j';
/*Donation Step*/
for i = 1,2,...,p in parallel do
  let T_i be the set of top-nodes in F_i;
  let D_i ⊆ T_i be a set of ⌈|T_i|/2⌉ nodes in T_i;
  if (i,j) ∈ R for some j then /*i donates D_i to j*/
    F_i ← F_i \ D_i;
    send message "i donates D_i" to j;
for j = 1,2,...,p in parallel do
  if j receives message "i donates D_i" then F_j ← D_i.
```

FIG. 1. Generic parallel backtrack search.

We prove some basic properties of the generic parallel backtrack search algorithm. The *level* of a busy processor is the level of its top-nodes. The level of a receiving processor after a donation will be the level of the donating processor. The *degree* of a tree is the maximum number of children of any internal node of the tree. A *unit* of work is one of the following three operations: *expand*, by which a processor expands a node, *donate*, by which a processor makes a donation, and *receive*, by which a processor receives a donation. The *total work* is the total number of work units performed.

PROPOSITION 1. *Let d be the degree of H. Then (i) a node can be donated at most $\lceil \log d \rceil$ times, (ii) the total work is at most $3n\lceil \log d \rceil$, and (iii) a busy processor with k top-nodes either increases its level or becomes idle after at most $\lceil \log k \rceil$ donations. In particular, for a binary tree, each node can be donated at most once, the level of a processor increases after each donation.*

PROOF. Let v be a node involved in a donation. The number of siblings of v that are together with v in a donation is reduced by half at 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 at the next step. This proves (i).

There are exactly n node expansions, as each node is expanded once. By (i), a node is donated at most $\lceil \log d \rceil$ times, and thus received at most $\lceil \log d \rceil$ times. Thus, the total work is at most $3n\lceil \log d \rceil$, which gives (ii).

To prove (iii), let P_i be a busy processor.

Case 1. P_i has only one frontier node u. Then P_i will expand u, increasing its level or becoming idle.

Case 2. P_i has two or more frontier nodes. Let k be the number of top-nodes of P_i. Each time P_i donates, the number of top-nodes of P_i is reduced by half, thus at most $\lceil \log k \rceil$ donations can be done before P_i increases its level or becomes idle. □

3.2. A DETERMINISTIC ALGORITHM. We now present a specific deterministic algorithm called *Full-Donation Backtrack Search* (FDBS). The strategy of FDBS is simple: Let as many idle processors receive donations as possible. In other words, choose the pairing set as large 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 there are more busy processors than the idle ones, one may take into account certain attributes such as the level number, size of local frontier, or length of local depth-first traversal path, in setting priorities for donation among busy processors. The rule of full-donation does not explore the potential computational advantages of setting donation priority among busy processors. The following theorem shows that FDBS is within a factor of $O(\lceil \log d \rceil)$ from the inherent lower bound $\max\{n/p, h\}$.

THEOREM 2. *The execution time of Full-Donation Backtrack Search on instance H is at most* $\lceil \log d \rceil (3n/p + h)$ *where d is the degree of H.*

PROOF. A step is *perfect* if every processor does one unit of work at that step, that is, it expands, donates, or receives; otherwise, the step is *imperfect*. By Proposition 1(ii), the total number of units of work is at most $3n\lceil \log d \rceil$. Hence, there can be no more than $3n\lceil \log d \rceil / p$ perfect steps.

We show that there are at most $h\lceil \log d \rceil$ imperfect steps. The *search-level* is the minimum level of all busy processors. We show that the search-level increases in each $\lceil \log d \rceil$ imperfect steps. Consider an imperfect step. There must be an idle processor that did not receive a donation in that step. By the maximality of the full-donation rule, every loaded processor must have donated in that step. Let P be a busy processor. If P is not loaded, then P either increases its level or becomes idle after one step. If P is loaded, then P donates at each imperfect step and, by Proposition 1(iii), increases its level after at most $\lceil \log d \rceil$ imperfect steps. Consider a processor Q that receives a donation from P at some step. As P halves its top-nodes for donation at each imperfect step, Q receives at most 2^k top-nodes from P if P has not increased its level after $\lceil \log d \rceil - k$ imperfect steps. Processor Q will increase its level, which is equal to the level of P, in k imperfect steps after receiving the donation. Hence, in $\lceil \log d \rceil$ imperfect steps, the level of any busy processor and the levels of processors that receive donations will increase. This implies that the search-level increases in each $\lceil \log d \rceil$ imperfect steps. The search-level is at most h. There can be at most $h\lceil \log d \rceil$ imperfect steps. □

3.3. A RANDOMIZED ALGORITHM. Though exhibiting optimality, Full-Donation Backtrack Search requires global control to implement donation. It turns out that the global control required by FDBS can be effectively replaced by randomization. We present a randomized algorithm called *Randomized Parallel Backtrack Search* (RPBS) in which random requests are used in

```
Pairing Step of RPBS
/*Requesting Message Step*/
for j = 1,2,...,p in parallel do
  if j is idle then
    dest(j) ← a random element of {1,2,...,p};
    send message "j wants new work" to dest(j);
/*Accepting Message Step*/
for i = 1,2,...,p in parallel do
  if i is loaded then
    let A_i = {j|i has received a message "j wants new work"};
    if A_i ≠ ∅ then
      select an arbitrary k ∈ A_i;
      send message "i has work to share" to k;  /*will donate to k*/
```

FIG. 2. Pairing step of RPBS.

donation. Randomized request was also proposed in a distributed implementation of backtrack search in [4a].

The donation of RPBS is as follows: Each idle processor initiates a request to a randomly chosen processor; a loaded processor that receives some requests selects one request from the received requests, and donates to the idle processor that made that request. Figure 2 contains the code describing the pairing step of RPBS. The corresponding modification in the donation step in Figure 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**". On each fixed instance H, the execution time of RPBS is a random variable.

The following theorem states that, for any instance H with degree d, with a probability approaching 1 exponentially fast as n increases, the execution time of RPBS is within a factor of $O(\log d)$ from the lower bound $\max\{n/p, h\}$. This shows that RPBS is a universal method for executing backtrack search algorithms efficiently in parallel without global control.

THEOREM 3. *Let the random variable $T(H)$ be the execution time of RPBS on H. Let n be the number of nodes in H and let h be the maximum number of nodes in a root-leaf path of H. Let d be the degree of H. Then, for any instance H and for any $p \geq 2$,*

$$Pr\left[T(H) > \lceil \log d \rceil \left(\frac{9n}{p} + 4h\right)\right] < n \exp\left(-\frac{n \log d}{4p^2}\right).$$

In particular, the probability bound is at most n^{-c} for any positive constant c if n is $\Omega(p^2 \log p)$.

Theorem 3 is an immediate consequence of the following theorem, combined with the fact that H has n nodes.

THEOREM 4. *Let the random variable $T(H, w)$ denote the number of steps of RPBS 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 $p \geq 2$,*

$$Pr\left[T(H, w) > \lceil \log d \rceil \left(\frac{9n}{p} + 4h\right)\right] < \exp\left(-\frac{n \log d}{4p^2}\right).$$

PROOF. At any point before w is expanded, let s be the current unique frontier node, generated but unexpanded, on path from the root to w. We call

the processor possessing s the *special processor*, denoted by S. When the special processor donates, either the donating processor remains as the special processor or the receiving processor becomes the special processor.

If $F_S = \{v\}$ at a donation step, then at the preceding node expansion step, either $F_S = \{v, u\}$ and u was expanded, or $F_S = \{u\}$ and v was generated from u. The latter case increases the level of S, and thus occurs at most h times. The former case implies $|F_S| > 1$ at the preceding donation step. A donation step at which $|F_S| > 1$ is called a *trial step*. At a trial step, the special processor S will donate if it receives a request from an idle processor. We show that, with the indicated probability bound, the number of trial steps is at most $\lceil \log d \rceil (9n/p + 3h)$, and the theorem follows.

We call a trial step *successful* if S donates at that step. A trial step is successful if S receives a request from an idle processor. 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*. As the total work is no more than $3n\lceil \log d \rceil$, there can be at most $6\lceil \log d \rceil n/p$ good trial steps. It is reduced to show that, with high probability, there are at most $3\lceil \log d \rceil (n/p + h)$ bad trial steps.

The probability that a bad trial step is successful is the probability that some idle processor requests to the special processor. The number of idle processors at a bad step is at least $p/2$. The probability that a bad trial step is successful is thus at least $1 - (1 - 1/p)^{p/2} > 1 - \exp(-1/2) > 1/3$. By Proposition 1(iii), the total number of donations by special processors is at most $h\lceil \log d \rceil$. So there can be at most $h\lceil \log d \rceil$ successful trial steps. Let $B(t, N, \rho)$ denote the probability that there are fewer than t successes in N independent Bernolli trials where the probability of success for each trial is ρ. We have

$$\Pr\left[\text{more than } 3\lceil \log d \rceil \left(\frac{n}{p} + h\right) \text{ bad trial steps}\right] \leq B\left(h\lceil \log d \rceil, 3\lceil \log d \rceil \left(\frac{n}{p} + h\right), \frac{1}{3}\right). \tag{1}$$

By a Chernoff-bound on binomial distribution [1], for $1 > \gamma > 0$,

$$B((1 - \gamma)\rho N, N, \rho) \leq \exp\left(-\tfrac{1}{2}\gamma^2 \rho N\right). \tag{2}$$

Let $(1 - \gamma)\rho N = h\lceil \log d \rceil$, $N = 3\lceil \log d \rceil (n/p + h)$ and $\rho = \frac{1}{3}$. Then

$$\gamma = \frac{n/p}{n/p + h}$$

and

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

To bound $\gamma^2 \rho N$ from below, consider two cases.

Case 1. $n/p \geq h$. In this case, $\gamma^2 \rho N \geq n\lceil \log d \rceil / 2p$.

Case 2. $n/p < h$. In this case, $\gamma^2 \rho N \geq n\lceil \log d \rceil / 2p^2$, as $n \geq h$. Hence

$$\gamma^2 \rho N \geq \frac{n\lceil \log d \rceil}{2p^2}. \tag{3}$$

By (1), (2), and (3), the probability of more than $3\lceil \log d \rceil (n/p + h)$ bad trial steps is at most $\exp(-\frac{1}{4}n \log d/p^2)$. The proof is complete. □

4. *Branch-and-Bound Method*

Branch-and-bound procedures are the most frequently used method in practice for the solution of combinatorial optimization problems. An introduction to branch-and-bound method can be found in [7] and an in-depth account in [3].

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 $A_1, A_2, \ldots, A_d$ such that an optimal solution to problem A can be found by solving each of $A_1, A_2, \ldots, A_d$ 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 subproblem A, and the lower bounds satisfy the monotonicity property that the lower bound on subproblem A is no larger than the lower bound on a subproblem derived from A. The 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. The computation may terminate when it finds a solution whose cost is not larger than the lower bounds of the remaining subproblems, and a solution of the least cost among the found solutions is the solution of minimum cost. Different rules can be used to decide the order in which subproblems are branched on. The "best-first" rule is to branch on the subproblem of least lower bound. The best-first rule tends to minimize the number of subproblems that are created, but also may need to maintain a large set of subproblems at a given time. The "depth-first" rule is to branch on the most recently generated subproblem. The depth-first rule tends to minimize the number of subproblems that are maintained, but may explore some subproblems unnecessarily. A recent study of time-space trade-offs in sequential branch-and-bound computation is given in [5].

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. The primitive step of the search is the expansion of a given node of the tree to produce its children and their cost bounds. We model a branch-and-bound computation as a rooted tree H in which each node has a finite number of children, together with a cost function c on the nodes of H such that $c(v)$ is the cost of an optimal solution to the subproblem associated v if v is a leaf, or the lower bound on the cost of the subproblem associated with v if v is an internal node. We require that the cost function c satisfy the conditions (i) if $v \neq w$, then $c(v) \neq c(w)$ and (ii) if w is a child of v, then $c(v) < c(w)$. Condition (i) of distinct costs is for convenience; condition (ii) is the monotonicity property of the lower bounds. We consider algorithms whose objective is to generate the leaf of least cost in H, using the node expansion operation. This model is similar to the one introduced in [5].

We are interested in executing a branch-and-bound computation in parallel. The fundamental challenge is to allocate the subproblems to the processors so

that they can all be performing useful work. A parallel branch-and-bound algorithm may not achieve an effective speed-up by merely keeping all processors busy; a successful solution must ensure that processors will not spend much time exploring useless subproblems, and that the overhead for interprocessor communication is not excessive.

5. *Parallel Branch-and-Bound*

The following is a generic description of the branch-and-bound algorithm we consider. Let $\Gamma(S)$ denote the set of 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.

Generic Branch-and-Bound 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;
  $F \leftarrow \{F \setminus S\} \cup \Gamma(S)$;
  $B \leftarrow \min(\{B\} \cup \{c(v)\colon v \in S \text{ and } v \text{ is a leaf}\})$;
  $F \leftarrow \{v \in F\colon c(v) \leq B\}$.

We think of the nodes in S 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 branch-and-bound algorithms described above. 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 v^*. Every node expansion algorithm to determine the minimum-cost leaf of H must expand every node of $\tilde{H}$. Let n be the number of nodes in $\tilde{H}$, and let h be the number of nodes in a longest root-leaf path in $\tilde{H}$. Notice that n and h are all concerned with $\tilde{H}$, not H. Then the execution of any algorithm is at least h, and the execution of any p-processor algorithm is at least n/p. Our goal is to design p-processor algorithms whose execution time comes close to the lower bound $\max\{n/p, h\}$ on all instances of (H, c).

Among algorithms that expand at most p nodes per step, the following "best-first" rule for selecting S is a direct extension of the sequential best-first rule. We call the algorithm implementing this rule Global 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

PROPOSITION 5. *The execution time of Global Best-First Search is at most* $(n/p) + h$.

PROOF. We show that all the nodes in $\tilde{H}$ will be expanded within $n/p + h$ steps. Let w be an arbitrary node in $\tilde{H}$. Let $P(w)$ be the path from the root of H to w and let v be the node on $P(w)$ that is currently in the frontier F. Consider the next node expansion step. If v is not expanded, then by the best-first rule, all p nodes expanded in this step are in $\tilde{H}$. As $|\tilde{H}| = n$, there can be at most n/p such steps. If v is expanded, the child of v on $P(w)$ will be in F. There can be at most h such steps, as the height of $\tilde{H}$ is at most h. Hence, w will be expanded within $n/p + h$ steps. □

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 the 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 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 $\min(\{B_i,\ i = 1, 2, \ldots, p\})$. The code for Local Best-First Search is contained in Figure 3.

The code of Figure 3 omits the details of how messages are used to notify all processors of the minimum cost and turn the computation off. Let τ be the time when all nodes of $\tilde{H}$ have been expanded. Let 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 a random processor. Thus, with high probability, all processors will receive the bound B by some later time σ, where $\sigma \leq \tau + O(\log p)$. From time σ onward, all local frontiers will be empty. By scheduling occasional broadcast phases, as discussed previously, each processor will learn that all local frontiers are empty, and stop its computation.

On each fixed problem instance (H, c), the execution time of the randomized algorithm Local Best-First Search is a random variable. We prove that there exists a universal constant α 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.

THEOREM 6. *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 $\tilde{H}$ and let h be the maximum number of nodes in a root-leaf path of $\tilde{H}$. Let the random variable $T(H, c)$ denote the execution time of Local Best-First Search on the*

```
Local Best-First Search
/*Initialization*/
F_1 = {r};
for i = 2,3,..., p, F_i ← ∅;
for i = 1,2,..., p, B_i ← ∞;
while some set F_i ≠ ∅ do
  /*Node Expansion Step*/
  for i = 1,2,..., p in parallel do
    if F_i ≠ ∅ then
      let v_i be the node of least cost in F_i;
      expand v_i;
      F_i ← F_i \ {v_i};
      if v_i is a leaf then B_i ← c(v_i)
      else
        for each child w of v_i do
          dest(w) ← a random element of {1,2,..., p};
          send w to dest(w)
    else
      send "a leaf of cost B_i" to a random element of {1,2,..., p};
  /*Message Arrival Step*/
  for i = 1,2,..., p in parallel do
    F_i ← F_i ∪ {w : dest(w) = i};
  for i = 1,2,..., p in parallel do
    B_i ← min(B_i ∪ {x : i has received a message "a leaf of cost x"});
  for i = 1,2,..., p in parallel do
    F_i ← {v ∈ F_i: c(v) ≤ B_i}.
```

FIG. 3. Local best = first search.

instance (H, c). *Then, for* $n \geq p$,

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

In particular, the probability bound is at most n^{-c} *for any positive constant* c *if* n *is* $\Omega(p(\log p)^{1/\alpha})$.

Theorem 6 is an immediate consequence of the following theorem, combined with the fact that $\tilde{H}$ has at most n nodes.

THEOREM 7. *For every instance* (H, c) *and for every node* w *in* $\tilde{H}$, *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. Then, for* $n \geq p$,

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

where α, β, γ, *and* d *are the constants stated in Theorem* 6.

Our goal is to prove Theorem 7. In the next section, we present a proof of Theorem 7 based on the proof outlined in [6]. Recently, Ranade [8] found a new, and simpler, proof. Nevertheless, the proof presented in this paper is of interest because of the probabilistic analysis techniques it uses.

6. *Proof of Theorem* 7

We present a proof of Theorem 7 in this section. The core of our analysis is to transform the problem into a cleanly stated problem about queuing systems. The analysis also gives rise to a simple game of strategy called the Shooting

Gallery Game and a study of large deviations in random walks. To enhance the continuity, some claims in this section are proved in the Appendix.

Recall that $\tilde{H}$ consists of all the nodes whose costs are less than or equal to that of the minimal-cost leaf in H. In analyzing the time required by Local Best-First Search to expand the nodes in $\tilde{H}$, we can disregard all nodes of H that do not lie in $\tilde{H}$, since no processor will ever choose to expand such a node when it has a node of $\tilde{H}$ available. We are interested in the time steps needed to expand a specific node w in $\tilde{H}$. We divide the nodes of $\tilde{H}$ 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 special node s present in some local frontier, 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 possessing a special node. Since the newly generated nodes are sent to random queues, 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 [1], the chance that the number of post-delays is greater than $2n/p$ is at most $\exp(-n/3p)$.

Thus, the crux of the proof of Theorem 7 lies in bounding the number of pre-delays. To approach this bound, we view the computation as a queuing process.

6.1. A Queuing Process. To describe the execution of Local Best-First Search as a queuing process, we view each processor as *server* and each node in $\tilde{H}$ 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 $2, 3, \ldots, p$ are empty. The system contains exactly one special customer and the queue containing the special customer is called the *special queue*. The queuing process alternates between *service steps*, in which the customer of least cost in each nonempty queue is served, and *arrival steps*, in which 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 queuing 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 $a_1, a_2, \ldots$ of elements from $\{1, 2, \ldots, p\}$ where a_i is drawn independently from the uniform distribution over $\{1, 2, \ldots, p\}$. The intended meaning is that a_k is the destination of the kth node of $\tilde{H}$ to be generated (the nodes generated simultaneously are arranged in order of increasing cost). It should be clear that the following data completely determines a run of the queuing process: an n-node tree $\tilde{H}$ 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 $\tilde{H}$, a node w in $\tilde{H}$ and a destination sequence. We may think of $\tilde{H}$, c, and w as being chosen by an *adversary* whose goal is to maximize the probability that the cost of the queuing process exceeds $d(n/p + h)$ for some suitable constant d.

We now modify the rules of the queuing process in favor of the adversary. We replace an arrival step at which k customers arrive with a sequence of k arrival steps at each of which one of those k customers arrives. A *step* is either a service step or an arrival step with a single customer. At each step, the adversary chooses one of five steps 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 $\{1, 2, \ldots, p\}$ 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 which customer is served from the special queue. 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; and (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 queuing process, the adversary preserves the ability to simulate an instance given by the triple $(\tilde{H}, c, w)$; that is, 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 $(\tilde{H}, c, w)$.

The adversary *succeeds* if the number of pre-events exceeds $d(n/p + h)$. Thus, we can prove Theorem 7 by showing that the adversary's chance of success in the modified queuing process is exponentially small. We show that the strategy that respects following rules is optimal for the adversary:

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 in the system.

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 occur during the period of these services. Post-events never occur. The cost of the process, that is, the number of pre-events, is simply the sum of the lengths of the special queues at the times the special customers arrive.

The sequence of events selected by the adversary gives rise to an *arrival pattern* $B = b_1, b_2, \ldots$, where $b_k \in \{r, s\}$ such that $b_k = s$ if the kth arrival is special and $b_k = r$ if the kth arrival 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 following proposition, proved in the Appendix, shows that the strategy respecting Rules 1 and 2 is optimal.

PROPOSITION 8. *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.*

6.2. A CONTINUOUS-TIME MODEL. The imposition of Rules 1 and 2 simplifies the queuing process and enables us to give the following clean description of it. A sequence of n customers (each corresponding to a regular customer) 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* (corresponding to the arrival of a special customer). When a service phase is triggered, a random queue is chosen (corresponding to the special queue). If the queue contains m customers, then the adversary receives a payoff of m and $m + 1$ service events occur (corresponding to m pre-events and one s-event); at each service event, one customer in each nonempty queue is served and deleted from its queue. No arrivals occur during a service phase. The total number of service phases is at most h. The adversary's goal is to maximize the probability that his total payoff exceeds $d(n/p + h)$. We wish to prove that its probability of achieving this goal is exponentially small.

To facilitate the analysis, we embed this process in continuous time by assuming that customers arrive according to a Poisson process. This poissonization allows us to use the tools of stochastic processes to analyze the payoff of the adversary. We shall assume that customers arrive according to a Poisson process with rate $p/2$ over the time period $[0, 4n/p]$. At each arrival, the queue where the customer arrives is drawn from the uniform distribution over $\{1, 2, \ldots, p\}$. Thus, the arrival process for a particular queue is Poisson with rate $1/2$ (and the arrival processes for all queues are mutually independent). 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. The following proposition is proved in the Appendix.

PROPOSITION 9. *Let $N(t)$ be the number of arrivals in a Poisson process with rate $\tilde{\mu}$ over the time interval $[0, t]$. Then (i) $Pr[N(t) < \lfloor \mu t/2 \rfloor] \leq exp(-0.3\lfloor \mu t/2 \rfloor)$ and (ii) $Pr[N(t) > \lfloor 2\mu t \rfloor] \leq exp(-0.19\mu t)$.*

By Proposition 9(i), the probability of fewer than n arrivals in a Poisson process with rate $p/2$ over the time period $[0, 4n/p]$ is at most $\exp(-0.3n)$. 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 $d(n/p + h)$.

Now we define a modified continuous-time process that incorporates an amortization mechanism 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. The following proposition, proved in the Appendix, states that the adversary is better off with the amortization.

PROPOSITION 10. *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 occurring. Then, $\tilde{U} \leq U + R$.*

On the other hand, by Proposition 9(ii), the probability that more than $\lfloor 8n/p \rfloor$ random service events occur over the time period $[0, 4n/p]$ is at most $\exp(-0.7n/p)$. 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.

The random service events, however, have a major impact on the lengths of the queues. For each queue, the arrival rate is $1/2$ whereas the rate of random service events is 1. One can expect that the length of a queue tends to be small, even without 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. M_k is nonincreasing in 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

$$\sum_{k=1}^{\infty} k m_k = \sum_{k=1}^{\infty} \sum_{i=k}^{\infty} m_i = \sum_{k=1}^{\infty} M_k. \tag{4}$$

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 6.4, we investigate the frequency with which queues of length at least k occur. This is preceded by Section 6.3, 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.

6.3. 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.

PROPOSITION 11. *Let S be the marksman's final score. Then, no matter how the marksman selects the number of targets at each step,*

$$Pr\left[S > 3\sqrt{\frac{mh}{p}}\right] \leq exp\left(-\tfrac{1}{6}\sqrt{\frac{mh}{p}}\right).$$

PROOF. If $h \leq 9m/p$, then $3\sqrt{mh/p} \geq h \geq S$. Now assume that $h > 9m/p$. Set $a = \frac{1}{2}\sqrt{mp/h}$. Then $a < p$. We 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. We change the scoring rules in the marksman's favor as follows: (i) count each shot of type 1 as a success and (ii) let the chance of success in each shot of type 2 be a/p. The shots of type 1 can generate at most $m/a = 2\sqrt{mh/p}$ successes. The number of successes generated by the shots of type 2 is stochastically dominated by the number of successes in a Bernoulli process with h trials, each having a chance of success a/p, denoted by $B(h, a/p)$. Hence, $\Pr[S > 3\sqrt{mh/p}] \leq \Pr[B(h, a/p) > \sqrt{mh/p}] \leq \exp(-\frac{1}{6}\sqrt{mh/p})$, where the last inequality is by the Chernoff-bound. □

6.4. A RANDOM WALK. Continuing our analysis of the random variable M_k, we investigate the frequency with which the length of a single queue is greater than or equal to k. To do so, we disregard the service phases triggered by the adversary. Then each queue is a Poisson process with arrival rate $1/2$ and service rate 1. Let X_i be the number of customers in a given queue after the ith event, which is either an arrival or a service. Then $\{X_i\}$ is a simple random walk on nonnegative integers, started at state 0, with a probability $1/3$ of going up by 1 at any state, and a probability $2/3$ of going down by 1 if not at state 0, or staying at state 0. The stationary distribution of $\{X_i\}$ is $\{\pi_i\}$ with $\pi_i = 1/2^{i+1}$ for $i = 0, 1, \ldots, \infty$.

We are interested in studying the random variable $U_k(m)$ that is the number of times that the random walk $\{X_i\}$, starting at 0, reaches a value greater than or equal to k in the first m steps.

PROPOSITION 12. $E[U_k(m)] \leq 2^{-k}m$.

PROOF. The random walk $\{X_i\}$ has the stationary distribution $\{p_i\}$ where $p_i = 2^{-(i+1)}$. By the ergodic theorem,

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

We claim that $E[U_k(m)] \leq 2^{-k}m$ for all $m \geq 0$. Suppose, on the contrary, that $E[U_k(m_0)] > 2^{-k}m_0$ for some $m_0 \leq 0$. Let $m = tm_0$ be a multiple of m_0. Let $E_k^a(m)$ be the expected number of times that the random walk $\{X_i\}$ reaches k or larger in the first m steps, started at a. Conditioning on the state of X_{jm_0} for $0 \leq j \leq t-1$,

$$E[U_k(m)] = \sum_{j=0}^{t-1} \sum_{a=0}^{\infty} \Pr\left[X_{jm_0} = a\right] E_k^a(m_0). \tag{5}$$

But $E_k^a(m_0) \geq E_k^0(m_0)$. Hence, (5) implies $E[U_k(m)] \geq tE_k^0(m_0) = tE[U_k(m_0)]$ and

$$\frac{E[U_k(m)]}{m} \geq \frac{tE[U_k(m_0)]}{tm_0} = \frac{E[U_k(m_0)]}{m_0} > 2^{-k},$$

contradicting to the fact that

$$\lim_{m\to\infty} \frac{E[U_k(m)]}{m} = 2^{-k},$$

as m is an arbitrarily multiple of m_0. □

A sequence of random variables $\{Y_i: i = 0, 1, \ldots\}$ is a *martingale* if, for $i \geq 0$, (i) $E[|Y_i|] < \infty$ and (ii) $E[Y_{i+1}|Y_0, \ldots, Y_i] = Y_i$. The following lemma is a special form of Azuma's martingale inequality [2].

LEMMA 13. *Let* $\{Y_i: i = 0, 1, \ldots\}$ *be a martingale such that* $|Y_{i+1} - Y_i| \leq c$ *for* $0 \leq i < n$. *Then*

$$Pr\left[Y_n \geq Y_0 + c\alpha\sqrt{n}\right] \leq exp(-\alpha^2/2).$$

PROOF. A simple proof can be found in [10, p. 55]. □

Let $E_k^a(m)$ denote the expected number of times that random walk $\{X_i\}$ reaches a state $\geq k$ in the first m steps, starting at state a.

LEMMA 14. *For* $|a - b| \leq 1$, $|E_k^a(m) - E_k^b(m-1)| \leq 3$.

PROOF. Let $T_{a,b}$ be the expected number of steps random walk $\{X_i\}$ takes to reach state b from state a. We show that $T_{1,0} = 3$. Suppose that we are at state 1. Conditioning on the next state, which is either 0 or 2, we have $T_{1,0} = 1 + T_{2,0}/3 = 1 + 2T_{1,0}/3$, as $T_{2,0} = 2T_{1,0}$. This gives $T_{1,0} = 3$. Note that $T_{a,a-1} = T_{1,0}$ for any $a > 0$. To prove the proposition, consider three cases.

Case 1. $a = b$. By definition, $E_k^a(m-1) \leq E_k^a(m) \leq E_k^a(m-1) + 1$ or equivalently $|E_k^a(m) - E_k^a(m-1)| \leq 1$.

Case 2. $b = a - 1$. Clearly, $E_k^{a-1}(m-1) \leq E_k^a(m)$. On the other hand, $E_k^a(m) \leq T_{a,a-1} + E_k^{a-1}(m-1) = 3 + E_k^{a-1}(m-1)$. Hence, $|E_k^a(m) - E_k^{a-1}(m-1)| \leq 3$.

Case 3. $b = a + 1$. Clearly, $E_k^a(m) \leq E_k^{a+1}(m-1) + 1$. On the other hand, $E_k^{a+1}(m-1) \leq T_{a+1,a} + E_k^a(m) = 3 + E_k^a(m)$. Hence, $|E_k^a(m) - E_k^{a+1}(m-1)| \leq 3$. □

The following proposition gives an upper bound on the probability that $U_k(m)$ is larger than its mean $E[U_k(m)]$ by an additive factor x. This probability bound is of general interest for large deviations in random walks.

PROPOSITION 15. *For $k > 0$,*

$$Pr[U_k(m) \geq E[U_k(m)] + x] \leq exp\left(\frac{x^2}{16m}\right). \tag{6}$$

PROOF. For $0 \leq i \leq m$, let $Y_i = E(U_k(m)|X_0, X_1, \ldots, X_i)$. Then, $\{Y_0, Y_1, \ldots, Y_m\}$ is a martingale. By the Markovian property,

$$Y_i = U_k(i) + E_k^{X_i}(m - i) \tag{7}$$

for $0 \leq i \leq m$. In particular, $Y_0 = E[U_k(m)]$ and $Y_m = U_k(m)$. Since $|U_k(i + 1) - U_k(i)| \leq 1$, by (7),

$$|Y_{i+1} - Y_i| \leq 1 + |E_k^{X_{i+1}}(m - i - 1) - E_k^{X_i}(m - i)|. \tag{8}$$

But $|X_{i+1} - X_i| \leq 1$. By Lemma 14 and (8), $|Y_{i+1} - Y_i| \leq 4 = c$. By Lemma 13, (6) follows by setting $\alpha = \frac{1}{4}xm^{-1/2}$. □

COROLLARY 16. *For $k \geq 1$, $Pr[U_k(m) \geq m/2^{k-1}] \leq exp(-m2^{-2k}/16)$.*

PROOF. By Proposition 12 and Proposition 15 with $x = m/2^k$. □

6.5. THE DISTRIBUTION OF M_K. We shall study the distribution of M_k for a fixed k. Having fixed k, we 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. By Proposition 9(ii), the probability that more than $\lfloor 12n/p \rfloor$ arrivals and random service events occur in some queue over the time period $[0, 4n/p]$ is at most $p \exp(-1.1n/p)$. We shall make the assumption that there are no more than $\lfloor 12n/p \rfloor$ arrivals and random service events in any queue.

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. Let $T_k = \sum_{i=1}^{p} X(k, i)$. $X(k, i)$ is nonincreasing in k and so is T_k.

PROPOSITION 17. *Let $m_0 = \lfloor 12n/p \rfloor$ and $\beta = 1/16$. Then*

$$Pr\left[X(k, i) \geq \frac{m_0}{2^{k-1}}\right] < exp\left(-\beta m_0 2^{-2k}\right).$$

PROOF. 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 associated random walk $\{X_i\}$ in Section 6.4 reaches a value greater than or equal to k. Under the assumption that there are no more

than $m_0 = \lfloor 12n/p \rfloor$ arrivals and random service events, $X(k,i)$ is stochastically no more than $U_k(m_0)$. By Corollary 16, $\Pr[X(k,i) \geq m_0/2^{k-1}] \leq \Pr[U_k(m_0) \geq m_0/2^{k-1}] < \exp(-\beta m_0 2^{-2k})$. □

COROLLARY 18. *For $n \geq p$,*

$$Pr\left[\sum_{i=1}^{p} X(k,i) > 24n2^{-k}\right] \leq p\,exp(-\beta 2^{-2k} n/p),$$

where β is the constant in Proposition 17.

PROOF. That $\sum_{i=1}^{p} X(k,i) > 24n2^{-k}$ implies that for some i, $X(k,i) > 24n2^{-k}/p \geq m_0/2^{k-1}$. The result follows from Proposition 17 and $m_0 = \lfloor 12n/p \rfloor \geq n/p$ if $n \geq p$. □

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. Recall that the payoff received by the adversary is $\sum_{k=1}^{\infty} M_k$ by (4). The following theorem completes the proof of Theorem 7.

THEOREM 19. *There exist positive constants α, β, γ, and d such that for $n \geq p$,*

$$Pr\left[\sum_{k=1}^{\infty} M_k > d\left(\frac{n}{p} + h\right)\right] \leq \gamma n\,exp\left(-\beta\left(\frac{n}{p}\right)^{\alpha}\right). \tag{9}$$

PROOF. We define $A = \lfloor a \log_2(n/p) \rfloor$ and $B = \lfloor (n/p)^b \rfloor$, where a and b are certain positive constants to be specified later. Then

$$\sum_{k=1}^{\infty} M_k \leq \sum_{k=1}^{A} M_k + \sum_{k=A}^{B} M_k + \sum_{k=B}^{\infty} M_k.$$

We shall bound each of the three summations separately. We first show that, with high probability, the last summation $\sum_{k=B}^{\infty} M_k$ is zero. Suppose that $\sum_{k=B}^{\infty} M_k > 0$. Then some queue must be of length at least $B = \lfloor (n/p)^b \rfloor$ at some time. Given that there are at most $m_0 = \lfloor 12n/p \rfloor$ arrival and random service events,

$$\Pr\left[\sum_{k=B}^{\infty} M_k > 0\right] \leq p \Pr[U_B(m_0) > 0] \leq pE[U_B(m_0)] \leq \gamma' n 2^{-(n/p)^b}, \tag{10}$$

where the last inequality is by Proposition 12.

We now consider the first summation $\sum_{k=1}^{A} M_k$. To bound M_k for a fixed 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.

Let $\mathscr{E}_1$ be the event that $T_k = \sum_{i=1}^{p} X(k,i) > 24n2^{-k}$ for some $k \le A = \lfloor a \log(n/p) \rfloor$. By Corollary 18, for some constant $\beta' > 0$,

$$\begin{aligned}\Pr[\mathscr{E}_1] &\le \sum_{k=1}^{A} p \exp\left(\frac{-\alpha' n 2^{-2k}}{p}\right) \\ &\le pA \exp\left(\frac{-\beta' n 2^{-2A}}{p}\right) \\ &\le \gamma'' n \exp\left(-\beta'\left(\frac{n}{p}\right)^{1-2a}\right). \end{aligned} \tag{11}$$

Assume that event $\mathscr{E}_1$ does not occur. Then, we may give the adversary the advantage of having $T_A = \lfloor 24n2^{-A} \rfloor$. Then, for properly chosen constants $c > 0$ and $c' > 0$,

$$cn\left(\frac{n}{p}\right)^{-a} \le T_A \le c' n \left(\frac{n}{p}\right)^{-a}. \tag{12}$$

Let $\mathscr{E}_2$ be the event that for some $k \le A$, a marksman with T_k targets and h bullets achieves a score larger than $3\sqrt{T_k h/p}$. We bound the probability of event $\mathscr{E}_2$ given that event $\mathscr{E}_1$ does not occur. By Proposition 11, noting that T_k is non-increasing in k and $A = \lfloor a \log_2(n/p) \rfloor$,

$$\begin{aligned}\Pr[\mathscr{E}_2 | \bar{\mathscr{E}}_1] &\le \sum_{k=1}^{A} \exp\left(-\frac{1}{6}\sqrt{\frac{T_k h}{p}}\right) \\ &\le A \exp\left(-\beta''\sqrt{h\left(\frac{n}{p}\right)^{1-a}}\right) \\ &< \gamma''' n \exp\left(-\beta''\left(\frac{n}{p}\right)^{(1-a)/2}\right), \end{aligned} \tag{13}$$

where the second last inequality uses the first inequality of (12).

Assume that neither event $\mathscr{E}_1$ nor event $\mathscr{E}_2$ occurs. Then viewing M_i as the score of the marksman,

$$\sum_{k=1}^{A} M_k < \sum_{k=1}^{A} 3\sqrt{\frac{T_k h}{p}} < \alpha' \sqrt{\frac{nh}{p}} \sum_{k=1}^{\infty} \sqrt{2^{-k}} \le d'\left(\frac{n}{p} + h\right),$$

where the last inequality is by the fact that $\sum_{k=1}^{\infty} \sqrt{2^{-k}}$ is bounded and that $\sqrt{xy} < x + y$ for $x > 0$ and $y > 0$.

Finally, we bound the middle summation $\sum_{k=A}^{B} M_k$. Given that neither of the events $\mathscr{E}_2$ and $\mathscr{E}_1$ occurs, we have $M_A \le 3\sqrt{T_A h/p}$ and $T_A \le c' n (n/p)^{-a}$ (by the second inequality of (12)). Since M_k is nonincreasing in k and $B = \lfloor (n/p)^b \rfloor$,

$$\sum_{k=A}^{B} M_k \le 3B\sqrt{\frac{T_A h}{p}} \le d''\sqrt{\left(\frac{n}{p}\right)^{1-a+2b} h} \le d''\left(\left(\frac{n}{p}\right)^{1-a+2b} + h\right), \tag{14}$$

where the last inequality is again by $\sqrt{xy} < x + y$.

Take $a = 2/5$ and $b = 1/5$. Then $a = 2b$, and from (14).

$$\sum_{k=A}^{B} M_k \le d''\left(\left(\frac{n}{p}\right)^{1-a+2b} + h\right) = d''\left(\frac{n}{p} + h\right).$$

By the probability bounds indicated in (9), (10), (11) and (13), take $\alpha = \min\{1 - 2a, (1 - a)/2, b\} = 1/5$, and constants β, γ, and d in (9) can be extracted from the proof. □

The proof of Theorem 7 is complete.

7. *Future Research*

We have made the strong assumption that any processor can send a message to any other processor in unit time. The algorithms can be implemented on a PRAM in which processors communicate via a shared memory. The algorithms can also 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. Recently, Ranade [9] has given a more sophisticated parallel backtrack search algorithm that runs on the Butterfly network and, with high probability, has an execution time within a small constant factor of the inherent lower bound. It remains open to analyze the modified branch-and-bound algorithm in sparsely interconnected networks such as the Hypercube and the Butterfly networks.

Appendix A. Some Proofs

PROOF OF PROPOSITION 8. 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, B and Rules 1 and 2. Let $\tilde{L}$ be any sequence of events consistent with A and B.

Let k be the number of special customers and let s_i be the ith special customer. Let d_i and $\tilde{d}_i$ be the number of pre-delays to s_i in L and $\tilde{L}$, respectively. We show that

$$\sum_{i=1}^{k} d_i \ge \sum_{i=1}^{k} \tilde{d}_i. \tag{15}$$

Let $q_i(j)$ and $\tilde{q}_i(j)$ be the length of queue j upon the arrival of s_i with L and $\tilde{L}$, respectively. Let

$$\alpha_i = \max_{1 \le j \le p} \{\tilde{q}_i(j) - q_i(j)\}, \tag{16}$$

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

$$\tilde{d}_k \le d_k + \alpha_k \tag{17}$$

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

$$\alpha_{i+1} \leq \alpha_i - \tilde{d}_i + d_i. \tag{18}$$

One can derive (15) from (17) by repeatedly applying (18). We first show that for $i = 1, 2, \ldots, k$,

$$\tilde{d}_i \leq d_i + \alpha_i, \tag{19}$$

which includes (17) as a special case.

We fix i and assume that s_i arrives at queue 1. By Rule 1, $d_i = q_i(1)$. Since there can be at most $\tilde{q}_i(1)$ pre-delays to s_i given $\tilde{L}$, $\tilde{d}_i \leq \tilde{q}_i(1)$. 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 (19).

To prove (18), let $a_i(j)$ be the number of arrivals to queue j between s_i and s_{i+1} exclusively. Then $q_{i+1}(j) = \max\{0, q_i(j) - d_i\} + a_i(j)$ by Rules 1 and 2, and $\tilde{q}_{i+1}(j) = \max\{0, \tilde{q}_i(j) + a_i(j) - \tilde{d}_i\}$. Since $\max\{a, b\} - c = \max\{a - c, b - c\}$,

$$\begin{aligned}
\tilde{q}_{i+1}(j) - q_{i+1}(j) &= \max\left\{0 - q_{i+1}(j), \tilde{q}_{j+1}(j) + a_i(j) - \tilde{d}_i - q_{i+1}(j)\right\} \\
&\leq \max\left\{0, \tilde{q}_i(j) - \tilde{d}_i - q_i(j) + d_i\right\} \\
&\leq \max\left\{0, \alpha_i - \tilde{d}_i + d_i\right\} \\
&\leq \alpha_i - \tilde{d}_i + d_i,
\end{aligned} \tag{20}$$

where the last two inequalities are by (16) and (19), respectively. By (16) and (20), $\alpha_{i+1} \leq \alpha_i - \tilde{d}_i + d_i$, which is (18) as desired. □

PROOF OF PROPOSITION 9. By Markov inequality, for an arbitrary random variable Z and $\theta > 0$,

$$\Pr[Z > z] \leq \exp(-\theta z) E \exp(\theta Z) \tag{21}$$

$$\Pr[Z < z] \leq \exp(\theta z) E \exp(-\theta Z). \tag{22}$$

Let $S_m = \sum_{i=1}^{m} X_i$, where X_i are independent and identically dependent exponential random variables with mean $1/\mu$. Then, $N(t) \geq m$, if and only if $S_m \leq t$, and

$$E \exp(sS_m) = (E \exp(sX_1))^m = \left(\mu(\mu - s)^{-1}\right)^m. \tag{23}$$

Taking $Z = S_m$, $z = t$ and $m = \lfloor \mu t/2 \rfloor$ in (21) and $\theta = s = \mu/2$ in (23),

$$\begin{aligned}
\Pr[N(t) < \lfloor \mu t/2 \rfloor] &= \Pr[S_{\lfloor \mu t/2 \rfloor} > t] \\
&\leq \exp(-\mu t/2) 2^{\lfloor \mu t/2 \rfloor} \leq \exp(-0.3 \lfloor \mu t/2 \rfloor).
\end{aligned}$$

Similarly, taking $Z = S_m$, $z = t$ and $m = \lfloor 2\mu t \rfloor$ in (22) and $\theta = -s = 2\mu$ in (23),

$$\begin{aligned}
\Pr[N(t) > \lfloor 2\mu t \rfloor] &\leq \Pr[S_{\lceil 2\mu t \rceil} \leq t] \\
&\leq \exp(2\mu t) 3^{-\lceil 2\mu t \rceil} \leq \exp(-0.19\mu t).
\end{aligned}$$

□

PROOF OF PROPOSITION 10. The proof is similar to that of Proposition 8. Let A be an instance of arrivals and let B be an instance of random service events. Let (A, B) denote the instance of arrival A with random service events

B. Let d_i and $\tilde{d}_i$ be the number of pre-delays to s_i with instance A and instance (A, B), that is, without and with random service events, respectively. Thus $\tilde{U} = \sum_{1 \le i \le k} \tilde{d}_i$ and $U = \sum_{1 \le i \le k} d_i$. 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 $1 < i \le k$. Thus, $R = \sum_{1 \le i \le k} r_i$. We show that

$$\sum_{1 \le i \le k} \tilde{d}_i \le \sum_{1 \le i \le k} d_i + \sum_{1 \le i \le k} r_i. \tag{24}$$

Let $q_i(j)$ and $\tilde{q}_i(j)$ be the length of queue j upon the arrival of s_i with A and (A, B), respectively. Let $\alpha_i = \max_{1 \le j \le p}\{\tilde{q}_i(j) - q_i(j)\}$. We show that

$$\tilde{d}_k \le d_k + \alpha_k \tag{25}$$

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

$$\alpha_{i+1} \le \alpha_i - \tilde{d}_i + d_i + r_{i+1}. \tag{26}$$

One can derive (24) and (25) by repeatedly applying (26). The proof of (25) is the same as that of (19). To prove (26), let $q'_i(j)$ and $\tilde{q}'_i(j)$ be the length of queue j upon the time of completing the service phase triggered by s_i on instances A and (A, B), respectively. Let $a_i(j)$ be the number of arrivals to the queue j between s_i and s_{i+1} exclusively. Then $a_i(j) = q_{i+1}(j) - q'_i(j)$, as A has no random service event s. Set $a'_i(j) = q_{i+1}(j) - q'_i(j)$. Then $a_i(j) \le a'_i(j) + r_{i+1}$, as each random service event serves at most one arrival in one queue. Hence, $\tilde{q}_{i+1}(j) - q_{i+1}(j) \le \tilde{q}'_i(j) - q'_i(j) + r_{i+1}$. Let $\alpha'_i = \max_{1 \le j \le p}\{\tilde{q}'_i(j) - q'_i(j)\}$. Then

$$\alpha_{i+1} \le \alpha'_i + r_{i+1}. \tag{27}$$

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

$$\begin{aligned} \tilde{q}'_i(j) - q'_i(j) &\le \max\{0, \tilde{q}_i(j) - q_i(j) - \tilde{d}_i + d_i\} \\ &\le \max\{0, \alpha_i - \tilde{d}_i + d_i\} \\ &\le \alpha_i - \tilde{d}_i + d_i, \end{aligned}$$

where the last inequality is by (25). Since $\alpha'_i = \max_{1 \le j \le p}\{\tilde{q}'_i(j) - q'_i(j)\}$, the last equality gives $\alpha'_i \le \alpha_i - \tilde{d}_i + d_i$. This, together with (27), proves (26). □

REFERENCES

1. ANGLUIN, D., AND VALIANT, L. G. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. Syst. Sci. 19* (1979), 155–193.
2. AZUMA, K. Weighted sums of certain dependent variables. *Tohoku Math. J. 3* (1967), 357–367.
3. BALAS, E. Branch and bound methods. In *The Traveling Salesman Problem*, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds. Wiley, New York, 1985.
4. CARTER, L., STOCKMEYER, L., AND WEGMAN, M. The complexity of backtrack searches. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*. (Providence, R.I., May 6–8). ACM, New York, 1985, 449–457.

4a. FINKEL, R. A., AND MANBER, U. DIB—A distributed implementation of backtracking. *ACM Trans. Prog. Lang. Syst. 9*, 2 (1987), 235–256.

5. KARP, R. M., SAKS, M., AND WIGDERSON, A. On a search problem related to branch-and-bound procedures. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1986, pp. 19–28.

6. KARP, R. M., AND ZHANG, Y. A randomized parallel branch-and-bound procedure. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. (Chicago, Ill., May 2–4), ACM, New York, 1988, pp. 290–300.
7. PAPADIMITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
8. RANADE, A. A simpler analysis of the Karp–Zhang parallel branch-and-bound method. Tech. rep. No. 586. Computer Science Division, Univ. California at Berkeley, Berkeley, Calif., 1990.
9. RANADE, A. Optimal speedup for backtrack on a butterfly network. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, 1991, pp. 44–48.
10. SPENCER, J. *Ten Lectures on the Probabilistic Method*. SIAM, Philadelphia, Pa., 1987.

RECEIVED OCTOBER 1990; REVISED APRIL 1992 AND MARCH 1993; ACCEPTED MARCH 1993