Fast Deterministic Parallel Branch-and-Bound

Kieran T. Herley[†] Andrea Pietracaprina[‡] Geppino Pucci[§]

Abstract

The *branch-and-bound* problem involves determining the minimum cost leaf in a cost-labelled tree, subject to the constraint that only the root is known initially and that children are revealed only by visiting their parent. We present the first efficient deterministic algorithm to solve the branch-and-bound problem for a tree T of constant degree on a p-processor parallel machine. Let c^* be the cost of the minimum-cost leaf in T, and let n and h be the number of nodes and the height, respectively, of the subtree $T^* \subseteq T$ of nodes of cost less than or equal to c^*. Our algorithm runs in $O\left(n/p + h\log^2(np)\right)$ time on an EREW-PRAM. Moreover, the running time faithfully reflects both communication and computation costs, unlike most of the previous results where the cost of local computation is ignored. For large ranges of the parameters, our algorithm matches the optimal performance of existing randomized strategies. The algorithm can be ported to any architecture for which an efficient implementation of Parallel Priority Queues [PP91] is available.

Keywords: Algorithms and data structures, theory of parallel and distributed computation, combinatorial optimization, dynamic load-balancing.

Contact Author: Geppino Pucci
Departimento di Elettronica e Informatica,
Università di Padova,
Via Gradenigo 6/A, I-35131 Padova, ITALY
e-mail: `geppo@artemide.dei.unipd.it`
fax: +39 49 8277826

[†]Department of Computer Science, University College Cork – National University of Ireland, Cork, Ireland.
email: `k.herley@cs.ucc.ie`

[‡]Dipartimento di Matematica Pura e Applicata, Università di Padova, Padova, Italy.
email: `andrea@artemide.dei.unipd.it`

[§]Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy.
email: `geppo@artemide.dei.unipd.it`

1 Introduction

The solution of a combinatorial optimization problem can often be obtained by the exploration of a tree, whose internal nodes correspond to partial solutions (growing progressively more refined with increasing depth) and whose leaves correspond to feasible solutions. Finding a solution of minimum cost involves searching the tree to identify a minimum-cost leaf. For many NP-hard optimization problems, an exhaustive search of the entire tree would be prohibitively expensive due to its size. In such cases great care needs to be exercised to explore only as much of the tree as is necessary to identify the minimum-cost leaf. One of the most popular and frequently-used strategies of this kind is known as *branch-and-bound*. This technique relies on a *branching procedure*, used to generate the children of the nodes, and a *bounding procedure* used to label each node with a cost which is a lower bound to the cost of the best solution (if any) residing in that node's subtree. The search for an optimal solution (i.e., the minimum-cost leaf) is guided by the costs associated with the nodes and tends to favour the exploration of subtrees rooted at nodes of small cost, which, intuitively, are more likely to lead to the optimal solution. (See [PS82] for an extensive treatment of branch-and-bound.)

The computational structure of any branch-and-bound strategy is captured by the following abstract search problem [KSW86]. Let T be an arbitrary tree of finite size, whose nodes are labelled with distinct integer-valued *costs*, the cost of each node being strictly less than the cost of its children. The tree structure is not known in advance but is revealed incrementally as its exploration unfolds. Initially, only a pointer to the root is available. Once a pointer to a node is available, the node can be *visited*, at which juncture pointers to its children are made available together with their associated costs. We assume that the degree of any node is constant and that visiting a node takes constant time. The *branch-and-bound problem* involves determining the cost, c^*, of the minimum cost leaf in T. Note that any correct algorithm for this problem must visit all those nodes whose costs are less than or equal to c^*. These nodes form a subtree T^* of T. Throughout the paper, n and h will denote, respectively, the size and the height of T^*.

There is a straightforward sequential algorithm for the branch-and-bound problem based on the *best-first* strategy, which visits nodes in order of increasing cost. Such a strategy can be implemented by means of a sequential priority queue where nodes which are available but not yet visited are stored (using cost as key) and subsequently extracted in order of increasing cost. The $O(n \log n)$ running time of this simple strategy is dominated by the cost of the $O(n)$ queue operations. In fact, it can be proved that any algorithm implementing a pure best-first strategy requires $\Omega(n \log n)$ sequential time [Puc93].

In [Fre93], Frederickson devised several sequential algorithms for the problem of selecting the k-th smallest item in a large heap. In particular, he provides a very elegant and simple algorithm which attains $O(k \log \log k)$ time, and a rather complex recursive strategy to bring the time bound down to the optimal $O(k)$ time. Both the algorithms comply with the restriction that a node is visited only after its parent has been visited. In fact, the latter algorithm can be adapted to yield an optimal $O(n)$ sequential algorithm for the branch-and-bound problem as follows. Let T be the branch-and-bound tree and observe that the costs of the nodes of T satisfy the heap order. The algorithm executes a number of iterations: in the i-th iteration, Frederickson's algorithm is invoked to select the 2^i-th smallest element in T, say x_i. Once x_i is known, the subtree of the 2^i smallest

elements in T can be easily enumerated in $O(2^i)$ time, using, for example, a modified depth-first search. If the subtree contains at least one leaf of T, then the leaf associated with the smallest cost is returned. Otherwise, the next iteration is started. Clearly, an overall $O(n)$ running time is attained.

In a parallel setting, the development of efficient branch-and-bound algorithms entails a delicate trade-off between an equitable distribution of the computational load (*i.e.*, node visits) among the processors and the minimization of "superfluous visits" (*i.e.*, visits to nodes in $T - T^*$). Moreover, the balancing of computational load must be accomplished on-line as the execution unfolds, thus introducing an additional cost which must be kept under control. The attainment of these goals is hampered by the unpredictable shape of T^* and by the fact that c^* is not known in advance, so that nodes in T^* cannot be easily identified by inspection. As a consequence, enforcing load balance might result in a higher number of superfluous visits, while, on the other hand, reducing the number of such visits requires a best-first-like exploration which may result in keeping processors unnecessarily idle.

A number of parallel branch-and-bound algorithms have been devised for a variety of distributed-memory architectures. It is easy to see that any algorithm for this problem requires at least $\Omega(n/p + h)$ time on any p-processor machine, since all n nodes of T^* need to be visited and at least h steps must be performed to visit the longest root-to-leaf path in T^*. Karp and Zhang present a randomized algorithm [KZ93], running in $O(n/p + h)$ (*i.e.*, optimal) time with high probability on a complete network of processors. (See [Ran90] for a simplified analysis of the algorithm.) The algorithm, however, features a collection of local priority queues whose maintenance costs are not accounted for. For $n > p^2 \log p$, optimal running time is also achieved by the randomized algorithm of [LAB93] designed on their atomic message passing model, a variant of the complete network, which assumes that interprocessor communication is controlled by a centralized FIFO arbiter. Kaklamanis and Persiano [KP95] present a deterministic branch-and-bound algorithm that runs in $O\left(\sqrt{nh}\log n\right)$ time on an n-node mesh. The clever mesh-specific techniques exploited in their algorithm rely on the assumption that the mesh size and the problem size are comparable. It is not clear whether the performance of the algorithm scales when the problem size increases. Several implementations of branch-and-bound on real machines, based on different load balancing strategies, are presented and compared in [LM92, DLR93]. In these works, however, performance is evaluated experimentally rather than analytically.

An easier variant of branch-and-bound is represented by the *backtrack search problem*, where no costs are associated with the nodes (or, equivalently, all costs are the same) and all nodes of the tree T must be visited. Clearly, any fast algorithm for branch-and-bound also yields a fast backtrack search algorithm, while the reverse is not true. Parallel algorithms for backtrack search can be found in [KZ93, Ran91, KP95, HPP96b].

In this paper, we present a deterministic parallel algorithm for the branch-and-bound problem based on a parallelization of the $O(k \log\log k)$ heap-selection algorithm of [Fre93], combined with the use of parallel priority queues. The algorithm can be run on any machine for which an efficient implementation of parallel priority queues is available. Based on the work of [PP91], we show how to implement the algorithm on an EREW-PRAM consisting of p synchronous RAM processors with direct access to a common memory, where in a single step each processor may read or write

a distinct memory cell [JáJ92]. The algorithm runs in $O\left(n/p + h\log^2(np)\right)$ time, which is optimal for $h = O\left(n/(p\log^2(np))\right)$. We remark that the running time faithfully reflects all computation and communication costs, some of which, such as those connected with the management of the local data structures, are disregarded in some of the aforementioned works (e.g., [KZ93, Ran90]).

In Section 2 we recall some of the ideas employed in Frederickson's sequential selection algorithm [Fre93], which will be adopted in our branch-and-bound algorithm. A general strategy for parallel branch-and-bound is presented in Section 3, while its PRAM implementation is described in Section 4. Section 5 closes with some final remarks and pointers to future research.

2 Frederickson's Strategy

Consider a bounded-degree tree T whose nodes are labelled with costs satisfying the heap property. In [Fre93], a *clan* is defined as a set of at most s nodes, where s is an integer parameter which will be specified by the analysis. The nodes in the clan are referred to as its *members*. A clan C is associated with two additional sets of nodes: the *offspring* $\mathit{Off}(C)$, which is the set of children of clan members which are not themselves members of C; and the *poor relations* $PR(C)$, which will be characterized later.

For a set of tree nodes V, let $\mathit{Best}(V)$ denote the set containing the s cheapest nodes among those in V and their descendants. (If V and their descendants number less than s, $\mathit{Best}(V)$ contains all of them.) Frederickson's selection algorithm works by partitioning the nodes of T into clans. Such a partition induces a binary *tree of clans* $\mathcal{C}(T)$ as follows. (The resulting clan tree is related to, but distinct from, the underlying branch-and-bound tree T.) Let r denote the root of T. The root of $\mathcal{C}(T)$ is defined as the clan $R = \mathit{Best}(\{r\})$. The corresponding set of poor relations is empty, *i.e.* $PR(R) = \emptyset$). Each clan $C \in \mathcal{C}(T)$ has two child clans, C' and C'', with

$$\begin{array}{rclcrcl} C' & = & \mathit{Best}(\mathit{Off}(C)) & \text{and} & PR(C') & = & \mathit{Off}(C) - \mathit{Best}(\mathit{Off}(C)) \\ C'' & = & \mathit{Best}(PR(C)) & \text{and} & PR(C'') & = & PR(C) - \mathit{Best}(PR(C)). \end{array}$$

Note that the root R of $\mathcal{C}(T)$ has only one child, since $PR(R) = \emptyset$. Also, since T has bounded degree, a simple argument shows that $|\mathit{Off}(C)| = \Theta(s)$ and $|PR(C)| = \Theta(s)$. If a clan C has exactly s members, its *cost*, denoted by $\mathit{cost}(C)$, is defined as the *maximum* cost of any of its members; if C has fewer than s members, $\mathit{cost}(C) = \infty$. Note that in this latter case, C is a leaf of $\mathcal{C}(T)$. It is easy to see that every node in a clan C costs less than every node in $\mathit{Off}(C) \cup PR(C)$, hence both $\mathit{cost}(C')$ and $\mathit{cost}(C'')$ are strictly greater than $\mathit{cost}(C)$.

Notice that since the determination of clan membership is largely driven by the cost of various nodes, it may be the case that the members of a clan are not close to one another in the tree T. As a consequence, the shape of $\mathcal{C}(T)$ may be substantially different from the shape of T. Furthermore, while a node may be a poor relation of one or more clans prior to becoming a member of some clan, every tree node is a member of exactly one clan, so the clans partition the nodes of T.

In [Fre93], it is shown that the k-th smallest node of T is a member of one of the $2\lceil k/s \rceil$ clans of minimum cost. These clans can be identified through a sequential exploration of $\mathcal{C}(T)$ in increasing order of clan cost using a sequential priority queue. By adopting this strategy, and

setting $s = O(\log k)$, the k-th smallest node of T can be found in nonoptimal $O(k \log\log k)$ running time. As discussed in the Introduction, Frederickson improves the running time to $O(k)$ through the application of a recursive strategy which, however, does not seem to lend itself easily to an efficient parallelization.

3 The Branch-and-Bound Algorithm

In this section, we present a parallel deterministic algorithm for solving the branch-and-bound problem on an arbitrary cost-labelled tree T of bounded degree. The algorithm is described for a generic p-processor machine, and relies on a parallel best-first exploration of the tree of clans $\mathcal{C}(T)$ defined in the previous section. The algorithm makes use of a *Parallel Priority Queue* (PPQ) [PP91], a data structure containing a number of items each labelled with an integer-valued key. Two main operations are provided by a PPQ: `Insert`, that adds a p-tuple of new items into the queue; `Deletemin`, that extracts the p items with the smallest keys in the queue. The PPQ Q is employed to store clans of $\mathcal{C}(T)$ generated during the execution, using their costs as keys.

For $1 \leq i \leq p$, let P_i denote the i-th processor of the machine. Processor P_i maintains a local variable ℓ_i which, at any time during the execution of the algorithm, gives the cost of the cheapest leaf visited by P_i up to that point. The processors also maintain two global variables, q and ℓ, whose values represent the minimum cost of a clan currently in the PPQ Q and the minimum of the ℓ_i quantities, respectively. At the beginning of the algorithm, Q is empty and a pointer to the root r of T is available. Also, each of the ℓ_i quantities is initialized to ∞, and so is ℓ. The algorithm is given below.

Algorithm BB:

1. Processor P_1 produces clan $R = \mathit{Best}(\{r\})$ and sets ℓ_1 to the cost of the minimum leaf in R, if any exists. Then, R is inserted into Q, and q and ℓ are set to the cost of R and to ℓ_1, respectively.

2. The following substeps are iterated until $\ell < q$.

   (a) `Deletemin` is invoked to extract the $k = \min\{p, |Q|\}$ clans $C_1, C_2, \ldots, C_k$ of smallest cost from Q. For $1 \leq i \leq k$, clan C_i is assigned to P_i.

   (b) For $1 \leq i \leq k$, P_i produces the two children of C_i, namely C_i' and C_i'', and updates ℓ_i accordingly.

   (c) `Insert` is invoked (at most twice) to store the newly produced clans into Q. The values ℓ and q are then updated accordingly.

3. The value ℓ is returned.

We say that a node is *visited* when the clan it belongs to is produced by Substep 2.(b). Recall that c^* denotes the cost of the minimum cost leaf, and T^* the subtree of T consisting of all nodes of cost less than or equal to c^*. Also recall that n and h denote the size and height of T^*, respectively.

Lemma 1 *Algorithm BB is correct.*

Proof: Since the cost of a clan C is strictly less than the cost of any node in $\mathit{Off}(C) \cup PR(C)$, it follows that at any time during the course of the algorithm all nodes in T with cost less than or equal to q have already been visited. Therefore, when ℓ becomes smaller than q, the algorithm has visited at least one leaf (the one with cost ℓ) and all nodes (and, in particular, all leaves) with cost less than or equal to $q > \ell$. This implies that $\ell = c^*$, hence the algorithm correctly identifies the minimum cost leaf in the tree. □

We say that a clan C is *good* if $cost(C) \leq c^*$, otherwise C is *bad*. Note that a good clan has exactly s members which all belong to T^*, while a bad clan either has less than s nodes or contains at least one node in $T - T^*$. The performance of the algorithm crucially relies on the following technical lemmas.

Lemma 2 *In each iteration of Step 2 at least one good clan is extracted from Q.*

Proof: Suppose that there is an iteration of Step 2 at the beginning of which the condition $\ell \geq q$ holds and such that no good clan is extracted from Q in Substep 2.(a). This implies that right before the iteration starts we have $q > c^*$. From the proof of Lemma 1 we know that all nodes with cost less than or equal to q have already been visited, hence we must have $\ell = c^* < q$, which is a contradiction. □

Lemma 3 *All good clans belong to the first $O(hs)$ levels of $\mathcal{C}(T)$.*

Proof: In fact, we prove that any clan containing nodes of T^* belongs to the first $O(hs)$ levels of $\mathcal{C}(T)$, which is a stronger claim. It is clear that a node of T is a member of only one clan C, but it may be a poor relation of a number of clans before C is produced. Notice, however, that the poor relations of a clan constitute a proper subset of either the poor relations or the offspring of the clan's parent in $\mathcal{C}(T)$. Therefore, a node may belong to the poor relations of at most $O(s)$ clans prior to becoming a member of a clan. If we number the levels of $\mathcal{C}(T)$ top to bottom starting at the root, we can see that if a node in T^* is member of a clan at level k in $\mathcal{C}(T)$, then its children must be members of clans at level $k + O(s)$ in $\mathcal{C}(T)$. Hence, it follows that the maximum level in $\mathcal{C}(T)$ of a clan containing nodes of T^* is $O(hs)$. □

Theorem 1 *The number of iterations of Step 2 required to reach the termination condition is $O(n/(ps) + hs)$.*

Proof: From Lemma 2 it follows that we only need to consider iterations in which at least one good clan is extracted from Q. We say that an iteration is *full* if p good clans are extracted from Q, and *partial*, otherwise. Since each good clan contains s members, which are all nodes of T^*, it is clear that there can be at most $n/(ps)$ full iterations. Consider now a partial iteration and notice that all the good clans present in the queue at that time are extracted. In this case, the minimum level in $\mathcal{C}(T)$ of any good clan in the queue increases of at least one by the end of the iteration. Since the level of any good clan is $O(hs)$, there cannot be more than $O(hs)$ partial iterations. □

4 PRAM Implementation

The PRAM implementation of Algorithm BB relies on the priority queue realization developed by Pinotti and Pucci in [PP91]. For a PPQ Q storing m items of constant size, the authors provide $O(\log m)$-time algorithms for `Insert` (insertion of p elements) and `Deletemin` (deletion of the p minima) on a p-processor EREW-PRAM*. Any clan produced by Algorithm BB is stored in the PRAM shared memory together with its offspring and poor relations. The clan is represented in the PPQ Q using constant space, by storing only its cost and a pointer to the shared-memory region where the clan resides.

Theorem 2 *The branch-and-bound problem for an arbitrary bounded-degree tree T can be solved on a p-processor EREW-PRAM in time*

$$O\left(\frac{n}{p} + h\log^2(np)\right).$$

Proof: It is easy to see that the running time of the algorithm is dominated by that of Step 2. By Theorem 1, there are $O(n/(ps) + hs)$ iterations in this step; hence, since $h \leq n$, the queue cannot contain more than $O(nsp)$ clans at any time. By using the PPQ algorithms of [PP91], Substeps 2.(a) and 2.(c) can then be executed in $O(\log(nsp))$ time. Given a set V of $O(s)$ nodes, $Best(V)$ can be computed by a single processor in $O(s)$ time, by a straightforward adaptation of Frederickson's linear-time selection algorithm, as follows. We incorporate the nodes of V into a temporary heap H, whose top $O(\log|V|)$ consist of $k = O(|V|)$ "dummy" nodes with cost $-\infty$ and whose lower levels are made up of the nodes of V and the subtrees rooted at those nodes. Note that the construction of H takes $O(s)$ time. Then, Frederickson's sequential, linear-time selection algorithm is invoked to select the $(k+s)$-th cheapest node in the heap, in $O(k+s) = O(s)$ time. Finally, a simple search is used to obtain, in $O(s)$ time, the $k+s$ nodes of minimum cost in H, the last s of which identify the set $Best(V)$. Therefore, Substep 2.(b) can be executed in $O(s)$ time. Thus, the overall complexity of the algorithm, as a function of s is

$$O\left(\left(\frac{n}{ps} + hs\right)(\log(nsp) + s)\right).$$

The theorem follows by choosing any $s = \Theta(\log(np))$. □

Note that the above choice of s requires that the algorithm knows the value n in advance, which is an unrealistic assumption in many practical contexts. In order to remove this assumption, we run the algorithm a number of times, choosing s based on exponentially increasing guesses for n. More precisely, we start by running the algorithm guessing the value $n_1 = p^2$ for n and setting $s_1 = \log(n_1 p) = O(\log p)$. If the algorithm does not terminate within time an_1/p, for a suitable constant a, we abort the execution and run the algorithm again guessing $n_2 = (n_1)^2$ and $s_2 = \log(n_2 p)$. In general, at the i-th guess, we run the algorithm with $n_i = (n_{i-1})^2$ and

*In fact, the PPQ implementation of [PP91] is for the CREW-PRAM and achieves a slightly better time bound for the Deletemin operation. The EREW implementation can be obtained by by simply replacing the use of the CREW merging routine in the PPQ algorithms with the EREW merging of [BN89].

$s_i = \log(n_i p)$ and abort the execution if it does not terminate within time an_i/p. This process is iterated until the first run for which the algorithm terminates within the chosen bound. Let j be the first index for which $n_j \geq n$. Since $h \leq n$ and $p \leq \sqrt{n_j}$, we have that

$$\left(\frac{n}{ps_{j+1}} + hs_{j+1}\right)(\log(ns_{j+1}p) + s_{j+1}) = O\left(\frac{n_{j+1}}{p}\right).$$

Therefore, if constant a is suitably chosen, the algorithm must terminate at some run $i \leq j+1$ and, upon termination, the current value of s is $O(\log(np))$. Moreover, the overall running time of the algorithm is dominated by that of the last run, which is within the stipulated bound.

Notice that the choice $s = \log(np)$ is not necessarily optimal. In fact, it can be proved that the optimal choice is $s = \left\lceil \sqrt{n/(ph)} \right\rceil$. For this value of s the running time of the algorithm becomes

$$O\left(\frac{n}{p} + h\log(np)\left\lceil\sqrt{\frac{n}{ph}}\right\rceil\right),$$

which is always $O\left(\frac{n}{p} + h\log^2(np)\right)$ and is asymptotically smaller for $h > n/(p\log^2(np))$. However, since h is usually not known, guessing the above optimal value for s may not be feasible in practice.

5 Further Research

In our branch-and-bound algorithm, interaction among the processors is mainly confined to the management of the PPQ. Therefore, the algorithm can be immediately ported to any distributed-memory architecture for which an efficient implementation of PPQ operations is available. Furthermore, the PRAM algorithms for PPQ operations devised in [PP91] rely on standard prefix and sorting routines, which are well studied and efficiently supported primitives on most architectures. However, these algorithms can guarantee fast running times only under the assumption that each PPQ item can be represented in constant space, while in our case the PPQ stores clans, whose size is nonconstant. Although this does not constitute a problem in a shared-memory environment such as the PRAM, where each PPQ item may simply store a pointer to a clan, in a distributed memory machine queue operations may involve the actual movement of clans among memory modules, which considerably increases the overhead associated with such operations.

An implementation of the general branch-and-bound algorithm was devised in [HPP96a] for the *Optically Connected Parallel Computer* (OCPC), a machine consisting of p processors, each with a private local memory module, communicating through a complete optical interconnection [AM88]. On such an architecture, the algorithm attains near-optimal performance but its implementation relies on rather involved shared memory simulation techniques, needed to enable the use of pointers in a distributed environment. Finding a simpler implementation for the OCPC or, in general, simple and efficient implementations for other distributed memory machines remains an interesting open question.

Another interesting direction for future work is to implement our theoretically efficient branch-and-bound algorithm, or some simplified version of it, on real parallel platforms and to compare its performance against that of other strategies, such as those proposed in [LM92, DLR93], in the

context of practical applications.

References

[AM88] R.J. Anderson and G.L. Miller. Optical communication for pointer based algorithms. Technical Report CRI 88-14, Computer Science Dept., Univ. of Southern California, Los Angeles, CA, 1988.

[BN89] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal algorithm for shared-memory machines. *SIAM J. on Computing*, 18(2):216–228, Apr. 1989.

[DLR93] R. Diekmann, R. Lüling, and A. Reinefeld. Distributed combinatorial optimization. In *Proc. of SOFSEM '93*, pages 33–60, Hrdoňov, Šumaca, Czech Republic, December 1993.

[Fre93] G. Frederickson. An optimal algorithm for selection in a min-heap *Information and Computation*, 104:197–214, 1993.

[HPP96a] K. Herley, A. Pietracaprina, and G. Pucci. Deterministic parallel algorithms for backtrack search and branch-and-bound. Technical Report TR-/96, Dipartimento di Matematica, Università di Padova, Padova, Italy, May 1996.

[HPP96b] K. Herley, A. Pietracaprina, and G. Pucci. Fast deterministic backtrack search. In *Proc. of the 23rd Int. Colloquium on Automata, Languages and Programming*, LNCS 1099, pages 598–609, July 1996.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading MA, 1992.

[KP95] C. Kaklamanis and G. Persiano. Branch-and-bound and backtrack search on mesh-connected arrays of processors. *Mathematical Systems Theory*, 27:471–489, 1995.

[KSW86] R.M. Karp, M. Saks, and A. Wigderson. On a search problem related to branch-and-bound procedures. In *Proc. of the 27th IEEE Symp. on Foundations of Computer Science*, pages 19–28, New York, 1986.

[KZ93] R.M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40:765–789, 1993. See also *Proc. of the 20th ACM STOC*, 1990, 290–300.

[LAB93] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message-passing. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 154–163, 1993.

[LM92] R. Lüling and B. Monien. Load balancing for distributed branch & bound algorithms. In *Proc. of the 6th International Parallel Processing Symposium*, pages 543–549, 1992.

[PP91] M.C. Pinotti and G. Pucci. Parallel priority queues. *Information Processing Letters*, 40:33–40, 1991.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[Puc93] G. Pucci. A lower bound on the "best-first" exploration of branch-and-bound trees. Manuscript, 1993.

[Ran90] A. Ranade. A simpler analysis of the karp-zhang parallel branch-and-bound method. Technical Report 586, Computer Science Division, Univerisy of California at Berkeley, Berkeley, CA 94720, 1990.

[Ran91] A.G. Ranade. Optimal speed-up for backtrack search on a butterfly network. In *Proc. of 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 40–48, 1991.