PARALLEL BRANCH-AND-BOUND ALGORITHMS: SURVEY AND SYNTHESIS

BERNARD GENDRON

University of Montreal, Montreal, Quebec, Canada

TEODOR GABRIEL CRAINIC

DSA, University of Quebec at Montreal and CRT, University of Montreal, Montreal, Quebec, Canada

(Received May 1993; revision received June 1994; accepted July 1994)

We present a detailed and up-to-date survey of the literature on parallel branch-and-bound algorithms. We synthesize previous work in this area and propose a new classification of parallel branch-and-bound algorithms. This classification is used to analyze the methods proposed in the literature. To facilitate our analysis, we give a new characterization of branch-and-bound algorithms, which consists of isolating the performed operations without specifying any particular order for their execution.

Branch-and-bound (BB) methods are well-known algorithmic tools for solving NP-hard optimization problems. For many of these inherently difficult problems, only small instances can be solved in a reasonable amount of time on sequential computers. Consequently, the use of parallelism to speed up the execution of BB algorithms has emerged as a way to solve larger problem instances, and has attracted many researchers in recent years. One objective of this paper is to give a detailed survey of what has been achieved in this field. Another objective is to introduce the reader to many of the challenges associated with adapting BB algorithms for parallel architectures. In particular, we present several strategies to exploit parallelism and we show, by using examples taken from the literature, that the choice of a strategy is greatly influenced by the design of the parallel machine used, as well as by the characteristics of the problem.

Surveys of parallel BB algorithms have been presented by Roucairol (1989a), Pardalos and Li (1990), and Trienekens and de Bruin (1992), and can also be found in papers dedicated to the general area of parallelism in combinatorial optimization and mathematical programming (Kindervater and Lenstra 1985, 1986, 1988, Ribeiro 1987, Roucairol 1989b, Pardalos, Phillips and Rosen 1992, Grama, Kumar and Pardalos 1993, Eckstein 1994a). Many new developments, however, have appeared since and, consequently, are not covered in these surveys. The present paper attempts to fill this gap. It also differs significantly from previous work in several aspects. First, it gives a new presentation of BB algorithms which consists of isolating the operations performed without specifying any particular order for their execution. The usefulness of this approach is shown by stating the convergence of exact BB methods under general conditions that hold for most sequential and parallel algorithms. Second, it proposes a new classification of parallel BB algorithms, which is used to characterize and analyze the methods proposed in the literature. Third, it is intended as a guide to the conceptual design of parallel BB algorithms. Our aim here is to diffuse as much pertinent information as possible to encourage further new developments. In particular, when describing a reported algorithm, we aim to identify the original contributions, specify the problem that was solved and the architecture being used, and give a summary of significant results.

Subject classifications: Programming, implementation on parallel architectures. Programming, integer, algorithms: survey.
Area of review: COMPUTING.

0030-364X/94/4206-1042 $01.25

The paper is organized as follows. Section 1 gives a presentation of BB algorithms, and introduces the related terminology used in the remainder of the text. Section 2 shows how parallelism can be exploited in BB algorithms, and characterizes many of the problems that one has to face when adapting these methods to parallel architectures. A detailed survey of the field, including a historical overview, is the subject of Section 3. Finally, the conclusion summarizes this work and proposes some research directions.

1. BRANCH-AND-BOUND ALGORITHMS

The characterization of BB algorithms has been studied by many researchers (Bertier and Roy 1964, Agin 1966, Lawler and Wood 1966, Balas 1968, Mitten 1970, Geoffrion and Marsten 1972, Kohler and Steiglitz 1974, 1976, Rinnooy Kan 1976, Sekiguchi 1981, Nau, Kumar and Kanal 1984, Ibaraki 1987, McKeown, Rayward-Smith and Turpin 1991), but most of the previous descriptions assume that the algorithms are executed in a sequential environment. In this section, we describe the operations involved in all BB algorithms without specifying their relative order of execution. Hence, our description can be shown to be valid for most BB algorithms executed in a parallel environment.

1.1. Branch-and-Bound to Find One Optimal Solution

BB may be seen as an implicit enumeration method for solving the optimization problem P: $Z(P) = \min_{x \in S} f(x)$, where f is a real-valued function, and S is a subset of a real vector space V. We assume that P can be solved by enumerating a finite number of points (not necessarily known in advance) in S, and that it is an NP-hard problem, which implies that no polynomial (in the dimension of V) algorithm is known to solve it. We also assume that the problem is either infeasible ($S = \emptyset$) or has a finite optimal value ($Z(P) > -\infty$).

When the problem is not tractable, a *divide-and-conquer* approach may be used to solve it. It consists of decomposing the set of feasible solutions S into n subsets $S_1, \ldots, S_n$, such that $\cup_{i=1}^n S_i \subseteq S$. One identifies $S^D = \cup_{i=1}^n S_i$ as the *decomposed set*, while its complement $S^E = S - \cup_{i=1}^n S_i$ is called the *excluded set*. Let P^D and P^E denote the optimization problems associated with the decomposed and the excluded sets, respectively. We assume that the optimal value of P^D is not worse than the optimal value of P^E, that is $Z(P^D) \leq Z(P^E)$. When $S^E = \emptyset$ (we then assume $Z(P^E) = +\infty$), the decomposition is called a *division* of S. If the subsets are also mutually disjoint ($S_i \cap S_j = \emptyset$, $i \neq j$), the decomposition is called a *partition* of S. Let P_i denote the optimization problem associated with subset S_i, and $Z(P_i)$ its optimal value ($i = 1, \ldots, n$). We then have $Z(P) = \min_{1 \leq i \leq n} Z(P_i)$ (with the convention that an infeasible problem has an infinite optimal value). If subproblems $P_1, \ldots, P_n$ are not solved directly, a similar decomposition, called a *branching operation*, may be applied to each of them. Successive decompositions may be performed until all subproblems obtained are easy to solve (we will see shortly, however, that in most cases, it is not necessary to solve all subproblems). It is assumed that the whole process generates a finite number of subproblems, because P can be solved by enumerating a finite number of points in S.

A subproblem Q obtained by performing branching operations need not be decomposed for one of two reasons:

Elimination Rule 1: The subproblem has been solved, that is, it is either infeasible, or an optimal solution has been found.

Elimination Rule 2: Another subproblem R is known to have an optimal value which is not worse than the optimal value of the given subproblem, that is, $Z(R) \leq Z(Q)$.

At first sight, it seems that to apply the second elimination rule, one must know the optimal values of the subproblems. However, it is sufficient to know only a lower bound on the optimal value of Q and an upper bound on the optimal value of R. This is the role of the *bounding operation*. It associates with each subproblem Q a lower bound $Z^l(Q)$, and an upper bound $Z^u(Q)$. Usually, a finite upper bound for a given subproblem Q also corresponds to a feasible solution to that subproblem, and hence, to the original problem P. Thus, if a given subproblem Q has a lower bound greater or equal to a known upper bound on the optimal value of the problem, then it need not be decomposed. This version of the second elimination rule is called the *lower bound test*. Another version of this rule, the *dominance test* (Kohler and Steiglitz 1974, Ibaraki 1977), compares directly two subproblems and, on the basis of problem-specific rules, determines if one has a better optimal value than the other (it is then said that it dominates the other). The dominance test is rarely used, while the lower bound test is universal in all BB algorithms.

The BB algorithm thus consists of performing branching and bounding operations, as well as testing the elimination rules, and can be described as the

process of building a tree, called a *BB tree*. The root of this tree is the original problem, while the sons of a given node (subproblem) Q are the subproblems obtained by decomposition of Q. The leaves of the tree are the subproblems that one does not decompose, and we distinguish between *leaves of type* 1, solved subproblems, and *leaves of type* 2, subproblems not decomposed due to the application of the second elimination rule. To each BB tree, we also associate a special tree, called *basic tree*, which is obtained by performing the same operations, but without testing the second elimination rule. Thus, two algorithms that perform the same branching and bounding operations on all subproblems, though possibly not in the same order, will have the same associated basic tree.

While the BB tree is built, the subproblems may be in one of the following three states: generated, evaluated, or examined. A subproblem is *generated* when it has been obtained from another subproblem by decomposition (initially, the original problem is the only generated subproblem). A generated subproblem is *evaluated* when a bounding operation has been applied to it, while it is *examined* if either a branching operation has been performed on it (in this case, we say it was *decomposed*), or the elimination rules have shown that it is not necessary to decompose it (in this case, we say it was *eliminated*). Let G denote the set of all generated subproblems, and define the corresponding partition:

$$G_{00} = \{Q \in G: Q \text{ is not evaluated and not examined}\},$$

$$G_{01} = \{Q \in G: Q \text{ is not evaluated and examined}\},$$

$$G_{10} = \{Q \in G: Q \text{ is evaluated and not examined}\},$$

$$G_{11} = \{Q \in G: Q \text{ is evaluated and examined}\}.$$

1.2. Convergence of the Branch-and-Bound Algorithm

When all generated subproblems have been examined, the algorithm stops and the best upper bound of all evaluated subproblems is the optimal value of the problem. This claim may be proven, provided that one uses a bounding operation and elimination rules that satisfy the following **Convergence Assumptions**:

1. A finite upper bound for any evaluated subproblem corresponds to a feasible solution to the original problem, i.e., the bound is obtained by evaluating f at a point of S.
2. A generated subproblem Q is solved if it is evaluated, and $Z^l(Q) = Z^u(Q)$ (when the subproblem is infeasible, we adopt the convention that $Z^l(Q) = Z^u(Q) = +\infty$).
3. Consider any list of subproblems $Q_0, R_0, \ldots, Q_k, R_k$, $k \geq 0$, where R_i is a leaf and either $Q_i = R_i$, or R_i is a descendant of Q_i such that $Z(Q_i) = Z(R_i)$ for $i = 0, \ldots, k$. Then, subproblem R_i is not eliminated, using the dominance test, by subproblem $Q_{(i+1 \bmod k+1)}$, for $i = 0, \ldots, k$.

The first assumption ensures that the best upper bound of all evaluated subproblems corresponds to a feasible solution (if one exists), while the second assumption gives a condition based only on the bounding operation for applying the first elimination rule. The third assumption further characterizes the second elimination rule when it is represented by a dominance test, and prevents deadlock situations, as we will see shortly in the proof of convergence. We are now ready to state and prove the convergence theorem:

Theorem. (Convergence of the BB Algorithm) *Suppose that the three convergence assumptions are satisfied by the BB algorithm. When all generated subproblems have been examined, one has:*

1. *There exists at least one evaluated subproblem* ($G_{11} \neq \emptyset$).
2. *The evaluated subproblem Q having the smallest upper bound among all evaluated subproblems* ($Z^u(Q) \leq Z^u(R)$ *for all* $R \in G_{11}$), *identifies an optimal solution of P, if one exists, and* $Z(P) = Z^u(Q)$.

To prove the first part of the theorem, suppose the contrary. This implies that all leaves are of type 2, since, by the second convergence assumption, all leaves of type 1 are evaluated. Now, consider a subproblem Q_0 which eliminated a given leaf R. By definition of the branching operation and the finiteness of the BB tree, there must exist a leaf R_0 which is either a descendant of Q_0 or Q_0 itself, such that $Z(Q_0) = Z(R_0)$. Since R_0 is a leaf of type 2, there exists a subproblem Q_1 which eliminated R_0. By the previous argument, we can identify a leaf R_1 which is either a descendant of Q_1 or Q_1 itself, such that $Z(Q_1) = Z(R_1)$. By applying the same argument a finite number of times, we will eventually reach leaf R, because there is a finite number of leaves. This would generate a list of subproblems $Q_0, R_0, \ldots, Q_k, R_k = R$, $k \geq 0$ that contradicts the third convergence assumption. Hence, there must exist at least one evaluated subproblem.

To prove the second part of the theorem, we distinguish two cases. In the first case, suppose that P is infeasible ($Z(P) = +\infty$). This implies that Q is infeasible also and $Z^u(Q) = +\infty$, so that $Z(P) = Z^u(Q)$, and the theorem is verified. In the second case, suppose that P is feasible ($Z(P) < +\infty$). We then prove the following assertion: for all $R \in G$, $Z(R) \geq Z^u(Q)$. This assertion implies that $Z(P) \geq Z^u(Q)$, so that Q has a finite upper bound value. Thus, by the first convergence assumption, $Z^u(Q)$ corresponds to a feasible solution of P, and $Z(P) \leq Z^u(Q)$. Putting the two inequalities together, we get the desired result and the feasible solution corresponding to $Z^u(Q)$ is also optimal.

To prove the assertion, suppose the contrary: $\exists\, R \in G$, $Z(R) < Z^u(Q)$. Without loss of generality, we can assume that R is a leaf because, if it is not, there exists a leaf which is a descendant of R having the same optimal value. Now, we distinguish two cases: R is a leaf of type 1 or R is a leaf of type 2. If R is a leaf of type 1, $Z(R) = Z^u(R)$ by the second convergence assumption. But, $Z^u(Q) \leq Z^u(R)$ by the definition of Q. Thus, we have $Z^u(Q) \leq Z^u(R) = Z(R) < Z^u(Q)$, a contradiction. If R is a leaf of type 2, then there exists a subproblem Q_0 which eliminated R and $Z(Q_0) \leq Z(R)$. By definition of the branching operation, there must exist a leaf R_0 which is either a descendant of Q_0 or Q_0 itself, such that $Z(Q_0) = Z(R_0)$. Again, we distinguish the same two cases. If R_0 is a leaf of type 1, we have $Z^u(Q) \leq Z^u(R_0) = Z(R_0) = Z(Q_0) \leq Z(R) < Z^u(Q)$, a contradiction. If R_0 is a leaf of type 2, we apply the same reasoning as above to identify subproblems Q_1 and R_1, such that $Z(R_1) = Z(Q_1) \leq Z(R_0)$, and R_1 is a leaf. By repeating the same argument a finite number of times, we will eventually reach either a leaf of type 1, in which case a contradiction is obtained in the same way as above, or leaf R. In that case, a list $Q_0, R_0, \ldots, Q_k, R_k = R$, $k \geq 0$, contradicting the third convergence assumption would have been generated by the algorithm, which yields a contradiction. This completes the proof of the assertion and, consequently, of the convergence theorem.

If the second elimination rule consists only of the lower bound test, the convergence is easier to prove, because in that case we no longer need the third convergence assumption. We first reformulate the second elimination rule in the following way:

Elimination rule 2 (by lower bound test): Q is not decomposed if it is evaluated, and another evaluated subproblem R has an upper bound which is not worse than the lower bound of Q, that is, $Z^u(R) \leq Z^l(Q)$.

With this new definition, the first part of the convergence theorem is trivially verified, because all leaves are now evaluated, and there exists at least one leaf, the number of generated subproblems being finite. To prove the second part, we proceed in the same way as above, with the exception that the assertion: for all $R \in G$, $Z(R) \geq Z^u(Q)$, is now easier to prove. Suppose the contrary: $\exists\, R \in G$, $Z(R) < Z^u(Q)$, and assume, without loss of generality, that R is a leaf. If R is a leaf of type 1, we obtain a contradiction in the same way as above. If R is a leaf of type 2, then there exists an evaluated subproblem R^* such that $Z^u(R^*) \leq Z^l(R)$. Thus, it follows that $Z^u(Q) \leq Z^u(R^*) \leq Z^l(R) \leq Z(R) < Z^u(Q)$, which is a contradiction. This proves the assertion and, hence, the convergence theorem.

1.3. Other Branch-and-Bound Algorithms

It is easy to modify the description given in subsection 1.1 to define an algorithm that identifies all optimal solutions. First, we redefine the branching operation such that P^E has an optimal value which is strictly worse than P^D, that is, $Z(P^D) < Z(P^E)$. Second, we impose a strict inequality to the second elimination rule. Then, when all generated subproblems have been examined, each evaluated subproblem with the smallest upper bound identifies an optimal solution, and by definition of the branching operation, all optimal solutions have been found. The proof of convergence proceeds exactly as above, with the exception that the third convergence assumption is redundant, because the type of list it prohibits is now impossible to obtain.

It is also possible to modify the method to find only an approximate solution. One may modify the branching operation in such a way that the excluded set may contain an optimal solution; relax the condition that all generated subproblems must be examined before the algorithm stops; or use an *ϵ-approximate algorithm* where the second elimination rule is modified such that a subproblem Q is eliminated by another subproblem R if $Z(R) - \epsilon \leq Z(Q)$, where $\epsilon \geq 0$ is a parameter that may change its value during the execution of the algorithm. The three approaches may be combined to obtain a wide variety of heuristic methods.

1.4. Sequential Branch-and-Bound Algorithms

We conclude Section 1 with some general remarks about the mechanism of the operations, and their order of execution in a traditional sequential environment (one process, with access to a single memory, performs the instructions sequentially).

To solve a given problem, it is possible to define several branching and bounding operations of varying degrees of efficiency. With regard to the bounding operation, experience in sequential environments has shown that "tighter is better" (Ibaraki 1987). Furthermore, branching and bounding operations may be interdependent, and may also depend on the history of the process, that is, the particular order in which the operations are performed. Finally, the time required to perform a given bounding operation may vary significantly from one subproblem to another. In recent years, the tendency has been to spend more effort on the original problem than on other subproblems. The idea, verified by experience, is to obtain as soon as possible very tight bounds, in particular, a good upper bound to reduce the number of generated subproblems, and hence, the total amount of work.

A possible order for the execution of operations in a sequential environment is given by the *best-first* BB paradigm. In this method, the value of the best upper bound found so far is kept in a variable *best_Z* for executing rapidly the lower bound test (we assume, for simplicity, that only the lower bound test is used for verifying the second elimination rule). Also kept in memory is a list L containing only evaluated but not yet examined subproblems (set G_{10}). Initially, *best_Z* is set to infinity and the original problem is evaluated and, if not solved, added to the list. At each step, if L is not empty, a subproblem Q in L with the smallest lower bound among all subproblems in L is selected (*selection operation*). The lower bound test is performed on Q, and if Q is not eliminated, it is decomposed according to the branching operation. Each newly generated subproblem is then evaluated, and if its upper bound is better than the current value of *best_Z*, it replaces it. The two elimination rules are also tested on the newly generated subproblems, which are added to the list if not eliminated. Another step is performed, until L becomes empty. The selection operation, called best-first selection, may be performed efficiently by managing the list as a heap data structure. The main advantage of this method is that, among all selection operations, the best-first selection is optimal with respect to the number of decomposed subproblems, when no ties occur among lower bounds, and the branching and bounding operations do not depend on the history of the process (Fox et al. 1978). A disadvantage is that the method may require a lot of memory space for storing list L.

The *depth-first* BB algorithm defines another order for the execution of operations. Here, the list L represents the set of generated subproblems not yet evaluated nor examined (set G_{00}). Initially, *best_Z* is set to infinity, and the original problem is evaluated and, if not solved, decomposed according to the branching operation. At each step, if the last examined subproblem was decomposed, all sons of this subproblem are added to the list, except one which is evaluated and examined. If the last examined subproblem was eliminated, the selection operation consists of choosing a subproblem in L among those which have been generated most recently. This subproblem is then evaluated and examined. Each time a subproblem is evaluated, the variable *best_Z* is updated if necessary. Another step is performed, until L is empty. A stack data structure may be used to perform efficiently the depth-first selection operation. There are three main advantages to the method. First, among all selection operations, it minimizes storage requirements (Ibaraki 1987). Second, when a subproblem is not eliminated, a significant part of the information generated by the last bounding operation is directly available to hasten the evaluation of the next subproblem (Nemhauser and Wolsey 1988). In comparison, such a reoptimization feature is more difficult to implement when the best-first selection operation is used. Third, feasible solutions are generally found more rapidly than with other selection operations (Ibaraki 1987, Nemhauser and Wolsey 1988). This is especially the case when upper bounds are computed only for subproblems that correspond to leaves of the basic tree. A disadvantage of the method is that it may generate a large number of subproblems. This disadvantage may be reduced by identifying a good upper bound in the early stages of the algorithm, thus strengthening the lower bound test, and consequently, reducing the number of generated subproblems.

Other selection operations and many variations on best-first and depth-first algorithms are also possible. The reader is referred to Ibaraki (1987) for a more complete treatment of this topic.

2. PARALLELISM IN BRANCH-AND-BOUND ALGORITHMS

Computer architectures strongly influence the design of parallel BB algorithms. Hence, before presenting a classification of parallel BB algorithms and discussing issues related to algorithmic design and performance measures, we give a description of parallelism at the hardware level. This is to be distinguished from parallelism at the software level, where several processes can simulate parallelism by sharing the resources of the same processor. Our description follows Bertsekas and Tsitsiklis (1989).

2.1. Parallelism at the Hardware Level

The *control* parameter refers to the presence or absence of a global control unit. Here, we only consider parallel architectures built according to the *control-flow* model: Each processor in the system is executing instructions in an order determined by a control unit. The other models proposed to date are the *data-flow* model, in which processors perform operations according to the availability of the input data, and the *demand-flow* model, in which processors execute operations in an order determined by the requirements for data (see Treleaven, Brownbridge and Hopkins 1982 for a more complete description of these models). Control-flow parallel architectures with only one control unit belong to the SIMD (Single Instruction Multiple Data) class, while systems with several control units (generally one per processor) belong to the MIMD (Multiple Instruction Multiple Data) class (Flynn 1966).

Synchronization refers to the presence or the absence of a global clock used to synchronize operations among processors. When there is only one clock, we speak of a *synchronous* system, while in the presence of several clocks, typically one per processor, the system is called *asynchronous*. SIMD computers are synchronous by definition, while MIMD systems are mainly asynchronous.

The *grain* indicates the amount of data each processor of the system can handle. In *fine-grained* systems, each processor can handle only a small amount of data, corresponding to scalar or small vector operations. At the other extreme, *coarse-grained* systems are characterized by the possibility of simultaneous treatment of large amounts of data.

The *communication* parameter refers to the way processors exchange information. There are two main possibilities: Processors may write and read in a common memory accessible to all (*shared-memory* systems), or may exchange messages (*message-passing* systems). Shared-memory systems may be further characterized by the presence of either a physically realized common memory, or a mechanism permitting access from each processor to any area of memory: *tightly* or *loosely coupled* systems, respectively. Message-passing systems are characterized by their *interconnection network topology*, which describes how processors are connected. The most common topologies are the ring, the tree, the mesh, and the hypercube (for details, see Bertsekas and Tsitsiklis).

Last, one considers the *number of processors*. *Massively parallel* systems are made of a large number of processors, on the order of thousands. Fine-grained systems are usually massively parallel, while coarse-grained ones have generally less processors, say on the order of tens (but the situation is changing rapidly, and there now exists some coarse-grained systems with more than one thousand processors). Note that tightly coupled shared-memory systems are generally restricted to a small number of processors, usually not more than twenty, due to the difficulty of implementing simultaneous access to a common memory without provoking bottlenecks.

The distinctions introduced at the hardware level are less strict when one considers the software level. Indeed, by adding appropriate software mechanisms, it is possible with some systems to simulate the behavior of other systems, although the efficiency of such simulations is questionable. For example, an MIMD system may simulate an SIMD system, or an asynchronous system may simulate a synchronous system. For the remainder of the text, we mainly consider parallelism at the software level, although there is a strong relationship between the conception of parallel algorithms at the software level and their actual implementation on parallel architectures.

2.2. Classification of Parallel Branch-and-Bound Algorithms

We identify three main approaches in designing parallel BB algorithms. *Parallelism of type* 1 introduces parallelism when performing the operations on generated subproblems. It consists, for example, of executing the bounding operation in parallel for each subproblem to accelerate the execution. Thus, this type of parallelism has no influence on the general structure of the BB algorithm and is particular to the problem to be solved. *Parallelism of type* 2 consists of building the BB tree in parallel by performing operations on several subproblems simultaneously. Hence, this type of parallelism may affect the design of the algorithm. This is also the case for *parallelism of type* 3, which implies that several BB trees are built in parallel. The trees are characterized by different operations (branching, bounding, testing for elimination, or selection), and the information generated when building one tree can be used for the construction of another.

The three types of parallelism may be combined either sequentially (Pekny and Miller 1992, for example, who perform in parallel the bounding operation at the root node, while parallelism of type 2 is exploited for the rest of the algorithm), or hierarchically. In the second case, for example, several BB trees are considered simultaneously, each of these trees is built in parallel, while, finally, treating each subproblem in

parallel (Miller and Pekny 1993). Most of the time, however, only one type of parallelism is exploited.

Parallelism of type 3, especially suited for implementation on coarse-grained asynchronous MIMD architectures, has been the object of very few studies. We briefly present three examples that illustrate its behavior. In the first example, the BB trees being built differ only in the branching operations (Pekny 1989, Miller and Pekny 1993). In the second example, only the selection operation differentiates the BB trees (Janakiram, Agrawal and Mehrotra 1988a, b, Janakiram et al. 1988). A variant of the depth-first operation, called randomized depth-first, is used. It randomly selects the next subproblem to evaluate and examine among the last generated subproblems. Thus, there exists an infinitesimal probability that two processes, each of them performing a randomized depth-first operation, build the same BB tree. Replication of work is possible though, particularly at the early stages of the algorithm. To avoid this, a global list of the status of the subproblems in the first k levels of the basic tree is maintained. Results of a simulation with ten processes show that this feature is essential to obtain good performance. They also suggest that an implementation of this algorithm on shared-memory systems is more appropriate than on message-passing ones. Finally, in the third example, the BB trees are differentiated only by the application of the lower bound test (Kumar and Kanal 1984). The main idea is to let each process perform the lower bound test with different values for the upper bound. At any time during the execution of the algorithm, one process (not necessarily the same) uses the best upper bound found so far by all processes, while the other processes take an "optimistic" view by subtracting $\epsilon > 0$ from the value of the best upper bound. Hence, this method essentially consists of performing concurrently several ϵ-approximate BB algorithms. Since, at any time, at least one process uses $\epsilon = 0$ to perform the lower bound test, the whole algorithm can be shown to converge to an optimal solution, if one exists. Note that this method is appropriate only for algorithms that frequently update the upper bound, in particular when the bounding operation gives weak bounds, or when a good starting upper bound is not known.

The second type of parallelism has been the object of abundant work. The approach is mostly appropriate for implementation on coarse-grained asynchronous MIMD systems, although some studies have been conducted on massively parallel fine-grained SIMD systems (Kindervater and Trienekens 1988, Dehne, Ferreira and Rau-Chaplin 1989a, b), on a pipeline computer, and on a data flow architecture (Kindervater and Trienekens). Fine-grained implementations, however, are appropriate only for algorithms that require small amounts of memory, particularly for the bounding operation. As for SIMD systems, all instructions must be the same for all processors, which is quite unnatural for general BB algorithms (as noted at the end of Section 1, the bounding operation may be quite different from one subproblem to another). Results on a fine-grained SIMD architecture and on a pipeline computer (Kindervater and Trienekens) show that the overhead due to the synchronization of all instructions is too costly, while the experimental nature of the data flow architecture, particularly the fact that only small instances could be solved (Kindervater and Trienekens), makes it difficult to draw general conclusions about its usefulness when compared to the classical control-flow approach.

To classify implementations of the second type of parallelism on asynchronous MIMD systems, we first distinguish between *synchronous* and *asynchronous* algorithms. A synchronous algorithm is divided into phases, such that in each phase the processes perform instructions independently of each other, while communications occur only between phases; thus, all processes must synchronize before communicating information. We further distinguish between *strictly* and *loosely* synchronous algorithms. In the first case, the communication protocol (which information to send and where) is assumed fixed and does not vary with different executions. These algorithms display a deterministic behavior in the sense that the processes follow exactly the same path for two different runs, unless each computing phase has a nondeterministic component. In a loosely synchronous algorithm, the processes may not follow the same path for different executions. For example, the communication protocol may depend on run-time information. When executing asynchronous algorithms, communications may occur at any time and are unpredictable. Thus, these algorithms have a nondeterministic behavior.

The second parameter used in our classification is based on the notion of *work pool*, which is a memory location where processes find and store their units of work (generated subproblems that are not yet examined). Typically, a process looking for work picks up a subproblem in a work pool, and evaluates, or examines it. When it finishes its action, the process usually stores the subproblems that are not yet examined in the same or in a different work pool. A process may also take actions independently of any work pool. For example, a subproblem newly generated by the process can be evaluated and even examined without first being stored and retrieved

from a work pool. Our classification distinguishes between *single* and *multiple* pool algorithms.

In the first case, there is only one memory location where units of work are stored. The sequential best-first and depth-first algorithms mentioned in subsection 1.4 are examples of single pool algorithms, where the pool is managed as a single list. Note that the pool may also be organized into two distinct lists (Miller and Pekny 1989, Pekny and Miller 1990, 1992, Kudva and Pekny 1993): One list contains subproblems that are not evaluated nor examined (set G_{00}), and another, subproblems that are evaluated but not yet examined (set G_{10}). Single pool algorithms are implemented mainly on shared-memory systems. On message-passing architectures, it is possible to implement them by using the *master-slave* paradigm: One process, called *master*, manages the work pool, and sends work units to other processes, called *slaves*, that send back results to fill the work pool.

In multiple pool algorithms, there are several memory locations where processes find and store their units of work. Several organization schemes are possible, the three most common being the *collegial*, the *grouped*, and the *mixed*. In a collegial algorithm, each work pool is associated with exactly one process. In a grouped organization, processes are partitioned, and each work pool is associated with a subset of this partition. Note that the collegial organization is a particular case of the grouped one, where the partition is made of singletons. In a mixed organization, each process has an associated work pool, but there is also a global work pool, shared by all processes.

In conclusion, we can classify parallel BB algorithms of type 2, intended to run on asynchronous MIMD architectures, as either *Synchronous Single Pool* (SSP), *Asynchronous Single Pool* (ASP), *Synchronous Multiple Pool* (SMP), or *Asynchronous Multiple Pool* (AMP) algorithms.

2.3. Algorithmic Design Issues

Other parameters may be used to characterize parallel BB algorithms of type 2; see, in particular, the classification proposed by Trienekens and de Bruin. However, instead of including these parameters in a taxonomy, we think it is more useful to introduce them as possible answers to problems of algorithmic design which occur naturally in this type of parallel BB algorithms.

One of these problems, the *initial generation and allocation of work units*, arises at the beginning of the execution of the algorithm, when only one subproblem, the root node of the tree, is available to all processes. Since it often happens that the branching operation generates few subproblems, a start-up phase where parallelism is not fully utilized seems difficult to avoid. On the other hand, using parallelism as soon as several work units become available may not be a good policy either. Consider, for example, a sequential depth-first algorithm using a dichotomous branching scheme. It is common to define a branching operation in such a way that only one of the two newly generated subproblems has a good probability of leading to an optimal solution. The depth-first selection operation would precisely choose this subproblem, while the evaluation and examination of the other subproblem would be delayed until the subtree with the selected subproblem at its root is completely examined. At this point, if an optimal solution has been found, there is a good probability that the other subproblem will be eliminated immediately. If the same branching strategy is used in a parallel environment, and subproblems are examined as soon as they become available, the two subproblems can be examined simultaneously. Then, the subtree rooted at the subproblem not selected by the sequential depth-first operation may now contain more generated subproblems, resulting in extra work compared to the sequential algorithm. This explains the so-called detrimental anomaly (Li and Wah 1984a), where a parallel algorithm is slower on a given instance than the corresponding sequential algorithm. The goal may thus be stated as follows: Use all processes as soon as possible, while avoiding giving them unpromising subproblems (that have only a small chance of leading to an optimal solution).

Several strategies have been proposed to address this issue: 1) assign the original problem to one process, and gradually broadcast among processes the units of work as they are created; 2) generate several subproblems by performing a special branching operation (the number of subproblems thus created being normally larger than the number of processes); 3) one process performs a sequential BB algorithm up to a point where a "sufficient" number of unexamined subproblems are available; 4) the processes perform a sequential phase in which the same tree is built by every process and, when the number of unexamined subproblems becomes at least equal to the number of processes, each process selects the subproblems on which it will subsequently work. The first three approaches can be used in all types of algorithms, while the last one is suited mainly for multiple pool algorithms. The choice of an appropriate strategy depends on practical considerations, such as the nature of the parallel architecture being used and the characteristics of the problem to solve.

Following the initial allocation of work units, designing a policy for subsequent *allocation and sharing of work units* among processes is another major issue. The objectives of such a policy should be to *balance the workload* among processes (all processes should do an approximately equal amount of work to fully utilize parallelism), and to feed them with promising units of work, to avoid a situation where the parallel algorithm would generate more subproblems than a corresponding sequential algorithm. Achieving these objectives is relatively easy in single pool algorithms. In multiple pool algorithms, however, the situation is more complex. One alternative is to dynamically create new processes that will take parts of fully loaded work pools (Schwan, Gawkowski and Blake 1988, Schwan et al. 1989a, b, Jansen and Sijstermans 1989). Another, more common, alternative is to have a fixed number of processes and to allow exchange of work units among pools. If this policy is used, we further distinguish between *static* and *dynamic* allocation strategies. In a static strategy, a given number of tasks (subproblems) are initially created, and processes subsequently share these tasks. This strategy is mainly used in mixed organization algorithms, where the global pool is used to keep the tasks, while each process performs a sequential BB algorithm (by using its own local pool) on one task selected from the global pool. Thus, in a static approach, there are no exchanges of subproblems among local work pools. Dynamic strategies, on the contrary, allow sharing of work units among local pools. Decisions on how to perform these exchanges are usually taken by the processes associated with the pools. We distinguish three classes of dynamic strategies (see Kröger and Vornberger 1990 for a similar classification):

1. **Strategy on request.** In this approach, a process with an (almost) empty work pool reacts by sending a request for work to another process. The request may be accepted, and a part of one work pool is transferred to the other, or rejected, in which case the process may decide to send another request. If the receiving process accepts the request, it must decide how many work units will be transferred and how they will be selected.
2. **Strategy without request.** Here, processes decide to share work units without being requested to by other processes. Before sending work units to another pool, each process must answer the following questions: How often to send work units? To which pool to send them? How many work units to send? How should they be selected?
3. **Combined strategy.** This approach combines the two previous ones. Processes exchange units of work without being asked for, and sends requests when the level of their work pool is too low.

Another design issue concerns the *application of the second elimination rule*. To apply it when it is defined by the lower bound test (see Wah, Li and Yu 1985 for a brief discussion of the application of dominance tests in a parallel environment), a process needs an upper bound on the optimal value of the given subproblem. Strategies for communicating upper bounds among processes depend mainly upon the type of algorithm and the architecture being used to implement it. For example, in an asynchronous algorithm on a shared-memory system, a variable accessible to all processes keeps the value of the best upper bound generated so far. When a process finds a better upper bound, it communicates it to other processes simply by updating the value of the variable. In an asynchronous algorithm on a message-passing architecture, each process may have access to a local variable that keeps the best upper bound it knows. When a process finds a better upper bound, it communicates it to other processes by sending its value. Usually, this communication step consists in a broadcast of the message to all processes (during the attempted broadcast, the message may be killed by some process which has found a better upper bound). The efficiency of such a broadcast is clearly dependent on the interconnection network topology.

The issue of *termination detection* is trivially solved for single pool and SMP algorithms. The real problem arises only for AMP algorithms on message-passing architectures, because it is not sufficient that all work pools be empty to declare termination. Indeed, some messages may still be traveling through the interconnection network. Of course, this problem is not specific to parallel BB algorithms, and methods for detecting termination of algorithms on distributed systems (without a central control) have been studied widely (see, for example, the paper by Dijkstra and Sholten 1980, and the references given in chapter 8 of Bertsekas and Tsitsiklis).

2.4. Performance Measures

Apart from problems of algorithmic design, another difficulty that arises naturally in a parallel environment is how to measure adequately the performance of an algorithm (see also the paper by Barr and Hickman 1993 on this subject). Related to parallel BB algorithms, several measures are possible:

Quality of the solution: This measure is only relevant in approximate BB algorithms. For example, one

may compare the best value obtained by a parallel algorithm with the one found by a sequential algorithm to investigate if parallelism can improve, and to what extent, the quality of the solution.

Number of generated subproblems: There are two variants of this measure: the total number of generated subproblems, and the number of subproblems generated before the best solution is found. This last measure introduces a timing notion, and may be difficult to estimate (in an AMP algorithm or an algorithm of type 3, for example). The first measure may be used in all types of algorithms, but it may not represent a fair evaluation of the total amount of work performed by the algorithm for two main reasons. First, the work performed may vary significantly from one subproblem to another (as noted in subsection 1.4). Second, it does not take into account additional work introduced by parallelism, such as the sharing of work units among processes, and the communication of an upper bound. Another problem related to this measure (and to other measures as well) is that it may vary significantly with different executions when the algorithm has a nondeterministic behavior, particularly if it is an asynchronous one. To obtain an adequate measure, it is then preferable to obtain statistics such as the mean and the standard deviation, derived from experimenting with several runs.

Speedup and efficiency: The speedup measure attempts to evaluate the improvement in time performance when more than one processor is used. Let $T(p)$ denote the time to solve a given problem on $p \geq 1$ processors. The speedup and the efficiency are then defined as $S(p) = T(1)/T(p)$ and $E(p) = S(p)/p$, respectively. The major difficulties with this definition consist of determining which algorithms should be used to measure the times, and how these should be measured. The answer to these questions depends mainly on practical considerations. For example, one definition for $T(1)$ is the time required by the best sequential algorithm. To evaluate the speedup of a given parallel algorithm based on this definition, time is measured with respect to the same parallel architecture for both the sequential and the parallel algorithms. However, the best sequential algorithm for all instances may not be known, as is often the case with problems solved by BB algorithms. One may then use as $T(1)$ the time obtained by a "good" sequential algorithm, or the time required by the parallel algorithm running on one processor. These are the methods of choice when studying the speedup performance of a parallel BB algorithm. In particular, a "good" sequential BB algorithm can be one in which the branching and bounding operations, as well as the elimination rules, are defined similarly as in the parallel algorithm (it is not clear how to generalize this definition of a "good" sequential algorithm for the case of parallel algorithms of type 3).

3. SURVEY

In this section, we present a survey of the literature on parallel BB algorithms of type 2, designed to be executed on coarse-grained asynchronous MIMD architectures (references to other types of algorithms have been given in the previous section). Our survey is based on a historical point of view, because the research interests seem to follow a pattern according to the period when researchers conducted their work. We distinguish three periods. In the early years (1975–1982), few parallel systems were available, and researchers are forced either to simulate parallelism or to use experimental architectures. Nevertheless, they discover interesting phenomena, in particular the possibility of superlinear speedups ($S(p) > p$). In the following years (1983–1986), researchers focus on the theoretical understanding of the performance of parallel BB algorithms. They mainly study speedup anomalies and derive expressions to evaluate the maximum speedup attainable by certain types of parallel BB algorithms. Since 1987, the focus is on the implementation of many types of algorithms on various parallel architectures.

3.1. Early Experiments (1975–1982)

The first simulation of a parallel BB algorithm was conducted by Pruul (1975; the results were published thirteen years later by Pruul, Nemhauser and Rushmeier 1988). The algorithm may be classified as AMP, because several processes operating asynchronously perform their own depth-first procedure. A coordinator is in charge of dispatching work units among processes. Initially, it assigns the original problem to one process. Subsequently, it answers requests from processes that emptied their work pool by giving them one subproblem taken from a nonempty work pool. The experiments are conducted with ten randomly generated 25-city instances of the asymmetric traveling salesman problem (TSP), on up to five processes. Results indicate that the number of generated subproblems decreases significantly when the number of processes is increased. Thus, average speedups on p processes are sometimes larger than p.

Results of another simulation experiment were published by Imai, Fukumura and Yoshida (1979), and Imai, Yoshida and Fukumura (1979). The algorithm belongs to the ASP class, because several

processes operating asynchronously share the same work pool, which is a list of subproblems not yet examined nor evaluated. A depth-first selection operation is used: A given process selects the next subproblem to evaluate and examine among the deepest nodes in the tree. Experiments are conducted with randomly generated trees and instances of the set covering problem, on up to 128 processes. The same tendency as in Pruul's study is observed and is called an "acceleration effect."

The first experiments on a parallel architecture appear to have been conducted at Carnegie-Mellon University in 1975 (Weide 1982), on a 5-processor loosely coupled shared-memory system, called C.mmp. The parallel algorithm, designed to solve 0–1 integer linear programming problems (ILPP), is very simple: Generate p subproblems and solve them independently using the same number of processes, each performing a sequential BB algorithm. Weide reports that average running times can sometimes be reduced by partitioning into more subproblems than there are processors, and by sharing the processors among the active subproblem-solving processes. Experiments with another parallel system designed at Carnegie-Mellon University were reported by Fuller et al. (1978). The parallel system, called Cm*, is a 10-processor loosely coupled shared memory architecture, in which each processor has its own local memory. The authors selected several algorithms to measure the efficiency of their architecture and its adaptability to various algorithmic designs. Among them is a BB algorithm for solving set partitioning problems. The parallel implementation is of the AMP type using a mixed organization. The algorithm starts by generating more than $k \cdot p$ subproblems, where k is a parameter arbitrarily fixed to a value of 10, and p is the number of processes used (one per processor). These subproblems are added to the global list where processes pick them, and perform sequential depth-first BB on each of them. The best upper bound found so far is kept in a global variable accessible to all processes. Results with five instances on up to eight processes show near-linear speedups, and even super-linear speedups for one data instance.

The first detailed study on the performance of BB algorithms implemented on a parallel architecture appeared in 1982. Mohan (1982, 1983, 1984) designed two single-list algorithms to solve the asymmetric TSP, and performed experiments on a 50-processor Cm*. The first algorithm is a synchronous master-slave approach, and thus belongs to the SSP class. At each phase, the master selects a subproblem according to a best-first criterion, and performs a branching operation which consists of creating p subproblems and sending one to each of the p slaves. The slaves perform bounding operations on the subproblems they receive and send back their results to the master. The second algorithm belongs to the ASP class. The processes pick subproblems in the list according to a best-first selection operation, perform branching and bounding operations, and add the newly generated subproblems to the list. Here, the branching operation is the classical dichotomous scheme. Results of a single run on a 30-city instance with up to 16 processors (one process per processor) show that the first algorithm is rather inefficient for two main reasons. First, the branching operation generates too many subproblems compared to the classical dichotomous rule. Second, waiting times on each processor due to synchronization and inexact workload balance significantly slow down the execution. Results also show that the speedup achieved by the second algorithm is rather limited due to contention of access to the single list.

Other early experiments on loosely-coupled shared-memory systems are mentioned in Møller-Nielsen and Staunstrup (1984), while parallel BB algorithms of the AMP class designed for message-passing architectures are presented in El-Dessouki and Huen (1980), Burton et al. (1982), DeWitt, Finkel and Solomon (1984), and Lavallée and Roucairol (1985). Another approach for implementing parallel BB algorithms consists of designing specialized hardware. MANIP, a specialized parallel system, was first presented in 1981 (Wah and Ma 1981), and subsequent improvements to the original design were proposed in the following years (Wah and Ma 1984, Wah, Li and Yu 1984, 1985). The architecture was never realized, but the efficiency of the design was analyzed by performing simulations on a sequential computer. Other specialized architectures were also proposed by Harris and Smith (1977), and Desai (1977, 1978, 1979).

3.2. Theoretical Studies (1983–1986)

Many researchers have studied the theoretical behavior of SSP algorithms with the following characteristics: 1) there are p processes performing branching and bounding operations; 2) the best upper bound found so far is kept in a global variable called *best_Z*, accessible to all processes; 3) the subproblems are stored in a single list L; 4) the algorithms are synchronous, and may be described as follows. Initially, the original problem is added to list L (possibly once a bounding operation has been performed). At each phase, $\min(|L|, p)$ subproblems in L are chosen according to a given selection criterion. Each selected

subproblem is treated by exactly one process. This process performs branching and bounding operations, and tests for elimination, on both the subproblem and the new ones obtained by decomposition, in an order that varies with the algorithm. The branching operation is assumed to be fixed and does not depend on the selection criterion. At the end of each phase, *best_Z* has been updated by all processes, and the generated subproblems which have not been eliminated are added to the list. The algorithm stops when the list is empty at the beginning of a phase.

The selection criteria commonly used are similar to the sequential ones. They can be characterized by a selection function v, which associates with each subproblem Q a value $v(Q)$, such that the subproblems selected to be examined in priority have the smallest v values. Thus, in a best-first algorithm, $v(Q) = Z^l(Q)$, and in a depth-first algorithm $v(Q) = -d(Q)$, where $d(Q)$ is the depth of subproblem Q in the BB tree. Ties among the v values of several subproblems are possible (in this case, they are broken arbitrarily when performing the selection operation), unless the selection function is one-to-one. It is possible to modify a selection function to ensure that it is one-to-one (details can be found in Li and Wah 1984a).

The selection function plays a central role in understanding two anomalous behaviors displayed by parallel BB algorithms of the type described above. Let $I(p)$ be the number of phases performed by the parallel BB algorithm using p processes to solve a given problem instance. We have a *detrimental anomaly* if there exists a problem instance such that $I(p_1)/I(p_2) < 1$ and $p_1 < p_2$. We have an *acceleration anomaly* if there exists a problem instance such that $I(p_1)/I(p_2) > p_2/p_1$ and $p_1 < p_2$.

Anomalous behaviors of parallel best-first algorithms (the selection function not necessarily being one-to-one) were first studied by Lai and Sahni (1982, 1983, 1984), and later on by Lai and Sprague (1985a, b, 1986). In particular, assuming that $Z^l(Q) \leq Z^l(R)$ whenever R is a descendant of Q in the basic tree, they show that when $p_1 = 1$ detrimental and acceleration anomalies cannot occur if all internal nodes of the basic tree have a lower bound different from the optimal value. However, they also show that detrimental and acceleration anomalies are possible for arbitrary values of p_1, p_2. For parallel depth-first algorithms, Quinn (1983) shows a similar result, and Li and Wah (1986b) provide sufficient conditions for detrimental anomalies not to occur, and necessary conditions for acceleration anomalies. In spite of the possibility of anomalous behaviors, results of simulations of best-first algorithms to solve the 0–1 knapsack problem and the TSP (Lai and Sahni 1984), and of a depth-first algorithm to solve the TSP (Quinn 1983), show that anomalies, particularly detrimental ones, are very rare.

Anomalous behaviors for more general selection functions were studied by Burton et al. (1983), Li and Wah (1984a, b, c, 1986a, 1990), Li (1985), and Wah, Li and Yu (1984, 1985). In particular, when $p_1 = 1$, they show that detrimental anomalies are impossible if the selection function is one-to-one and $v(Q) \leq v(R)$ whenever R is a descendant of Q in the basic tree. Li and Wah (1984c) also analyze the behavior of algorithms that use dominance tests, and of ϵ-approximate algorithms. The basic results on anomalous behaviors are difficult to extend to more general models of parallel BB algorithms. Li and Wah (1986a) briefly analyze the case where multiple lists are used, while Trienekens (1989a, 1990) considers a class of algorithms that may include asynchronous ones. However, the condition defining this class of algorithms can be verified a priori only for synchronous algorithms. Finally, Mans and Roucairol (1993) study anomalies occurring in parallel best-first algorithms in which ties among lower bounds are broken using a second selection function.

Another related research subject is the derivation of lower and upper bounds on the speedup obtained by parallel BB algorithms. Quinn and Deo (1983, 1986) and Huang and Davis (1987) give upper bounds on the speedup of an ASP algorithm using a single list and a best-first selection operation, while Li and Wah (1984b, 1986b) give bounds on the speedup of best-first and depth-first SSP algorithms.

3.3. Experiments on Parallel Systems (Since 1987)

Since 1987, researchers have focused on the design of parallel BB algorithms and their implementation on general-purpose parallel systems (one exception being the work by Cheng and Wang 1990, where a specialized architecture is proposed for implementing an AMP mixed organization algorithm). In this subsection, algorithms are grouped according to our classification and, for each class we select representative algorithms and give both general descriptions, and an overview of their most significant results.

3.3.1. Synchronous Single Pool Algorithms

As indicated previously, the SSP model has been used mainly to study theoretical properties of parallel BB algorithms. Few researchers have actually implemented this approach on a parallel system. All implementations of this approach were realized on message-passing architectures by using a master-slave model.

Quinn (1990) implements an SSP algorithm, similar to the model described in subsection 3.2, on a 64-processor NCUBE/7 hypercube. The master process, running exclusively on one processor, manages the single list and selects subproblems to send to each slave according to a best-first criterion. At every iteration, each slave receives one subproblem, performs branching and bounding operations, and sends back to the master the newly generated subproblems. Experiments are performed on ten 30-vertex instances of the TSP. A model is also proposed for predicting the speedup performance of the algorithm. Using this model, it is shown that the algorithm performs reasonably well (being competitive with some AMP algorithms), when the time required to perform operations on one subproblem largely dominates the time to communicate it.

McKeown et al. (1991; see also Rayward-Smith, Rush and McKeown 1993, McKeown, Rayward-Smith and Turpin 1991) implement a similar algorithm on a network of transputers (using up to eight processors), and report experiments on one instance of the Steiner tree problem using a depth-first selection operation. The work pool is stored in the memory of the master processor, but contains only limited information about the subproblems generated so far: The priority of each subproblem (the value of the selection function), its lower bound (to perform the lower bound test before sending the subproblem), and the processor on which the information about the subproblem is kept. When the master selects a subproblem, it is assigned in priority to the processor which keeps all the information concerning it. This scheme clearly minimizes communication times when implementing a single pool algorithm in a message-passing environment. The algorithm appears, however, to be less efficient than a corresponding ASP one.

Gendron and Crainic (1993b) (see also Gendron 1991) choose an SSP approach to implement a parallel version of a depth-first sequential approximate algorithm for solving the multicommodity location problem with balancing requirements. The algorithm stops when a fixed number of subproblems have been examined. In this context, it is crucial to avoid situations where the parallel algorithm finds a worse solution than the one obtained by the sequential algorithm. To attain this objective, the authors divide the parallel algorithm into two phases: a sequential phase, where the master process performs the sequential method up to N iterations, and a parallel phase. In this phase, the master selects subproblems according to their depths, sends them to the slaves, performs bounding and branching operations on one subproblem, and examines the subproblems sent to the slaves. The slaves only perform bounding operations on the subproblems they receive. This scheme is justified by the fact that, for most data instances, the time to perform the bounding operation on one subproblem clearly dominates the time for other computations or communications. Experiments on a network of up to 16 transputers are reported.

3.3.2. Asynchronous Single Pool Algorithms

Except for being asynchronous, almost all ASP algorithms proposed in the literature have the same characteristics as the SSP algorithms described previously: a fixed number of processes, a single list of unexamined subproblems, and a global variable that keeps the best upper bound found so far. These algorithms were implemented both on shared-memory systems and on message-passing architectures. In the second case, either the operating system's primitives are used to simulate a shared-memory or a master-slave approach is implemented. A theoretical analysis of this type of implementation based on a queueing network model, is given by Boxma and Kindervater (1991). The model is used to analyze the effect of variations in the number of slaves, or in the speed of the master and the slaves.

Roucairol (1987a, b) implements an ASP best-first algorithm on a 4-processor Cray X-MP 48 (a shared-memory architecture) for solving the quadratic assignment problem (QAP). Experiments are also reported on an emulator and comparisons are made with depth-first and random selection operations. Near-linear speedups are observed. A similar implementation on a 4-processor Cray2 is also used by Plateau and Roucairol (1989) to solve the 0–1 multiknapsack problem. To the basic approach of Roucairol, the authors add a mechanism to improve workload balancing: A process accessing the pool is allowed to treat a subproblem only if its current load (measured from the beginning of the execution) does not exceed $t\%$ of the mean load (best results were obtained for $t = 50$). Essentially the same approach as in Roucairol's paper is also used by Boehning, Butler and Gillett (1988) for solving ILPP with a simplex-based bounding procedure. Experiments were performed on three shared-memory systems with up to 20 processors. Superlinear speedups were observed for some instances. Different approaches for implementing ASP algorithms on shared-memory architectures can also be found in Barr and Stripling (1992) and Mohamed (1992).

Kumar, Ramesh and Nageshwara Rao (1988) also implement a best-first ASP algorithm similar to Roucairol's, but use a concurrent heap data structure

(Nageshwara Rao and Kumar 1988a, b) to manage the list of subproblems. This structure allows multiple heap insertions and deletions to be performed concurrently. It is shown on two instances of the TSP that this data structure dramatically improves performances over the traditional heap structure. The authors report experiments on many instances of the TSP and the vertex cover problem (VCP) by using a 100-processor loosely coupled shared-memory system, the BBN Butterfly computer. In particular, when the concurrent heap is used, a near-linear speedup is observed on a 25-city instance of the TSP. It is also shown that an ASP approach may not be as efficient for solving the VCP, because the time to perform a bounding operation for the VCP is significantly less than for the TSP (for the tested data instances). In this case, contention for access to the single list, even when it is managed as a concurrent heap, limits the performance of the algorithm. Le Cun, Mans and Roucairol (1993) (see also Mans 1992a) performed an extensive comparative study of concurrent priority queue data structures used in best-first BB algorithms. Their experiments on a shared-memory system of nine processors confirm that the concurrent heap structure, used by Kumar, Ramesh and Nageshwara Rao, exhibits good speedup performance. However, they also show that data structures other than the classical heap may be more efficient in a sequential environment. Consequently, although concurrent access to some of these data structures may be difficult, they could prove effective in reducing execution time, even in a parallel environment.

In Cannon and Hoffman (1990; see also Cannon 1988), a shared-memory system is simulated on a network of eight DEC VAX stations by using operating system functions. The algorithm is designed to solve large-scale 0–1 ILPP by a simplex-based strong cutting plane approach (also called branch and cut). Since the bounding procedure requires significant computing time, but also provides sharp bounds thus generating few subproblems for most instances, a direct adaptation of a sequential approach would not utilize the computing resources efficiently, particularly at the early stages of the execution. The authors thus implement a parallel starting phase that generates 32 subproblems, instead of only one, to initialize the work pool (managed as a shared file). They also implement mechanisms to "pause" the treatment of a subproblem when it appears that it may not lead to an optimal solution, and to "resume" processing on it at a later time. Experimental results are reported on seven instances. For one of these instances, the sequential algorithm using the parallel starting phase is shown to be more efficient than the traditional sequential algorithm that begins execution by treating the original problem.

Trienekens (1986, 1989b) implements a best-first master-slave ASP algorithm on a distributed network of heterogeneous computers. The master manages the work pool and sends a subproblem to a slave as soon as one becomes idle. Each slave receives a subproblem from the master, performs the branching operation, and sends back the newly generated subproblems, after evaluating them. Experiments are reported on several instances of the TSP with up to six processors. Results of simulations of this algorithm are also given by de Bruin, Rinnoy Kan and Trienekens (1988). A similar approach is implemented on a 64-processor NCUBE/6 hypercube multicomputer by Abdelrahman and Mudge (1988; see also Abdelrahman 1988) to solve the 0–1 ILPP. The authors observe that the speedup is limited by two main factors: the communication overhead, which increases with the size of the cube, and a poor workload balancing, because processors nearer to the master receive more subproblems than those farther away. The algorithm is also shown to be inferior to an AMP approach. Lüling and Monien (1989) report experiments with a similar algorithm implemented on a network of 64 transputers connected by a tree topology. They solve many instances of the VCP and report nearly linear speedups. This is in contrast with one of the conclusion of Kumar, Ramesh and Nageshwara Rao (1988), who remark that an ASP approach may not be appropriate for solving the VCP. This apparent contradiction is easy to explain by the fact that Lüling and Monien show results for instances with 150 nodes, while Kumar, Ramesh and Nageshwara Rao report experiments on graphs with between 50 and 80 nodes.

McKeown et al. present extensive results of an ASP algorithm, similar to their SSP approach, particularly in the way it manages the single work pool. The algorithm was tested on a network of up to 32 transputers, by solving several problems including instances of the Steiner tree problem, the TSP, and the Chinese postman problem. For the first two problems, the bounding operation is nontrivial, particularly for the TSP, and the algorithm shows near-linear speedups for reasonably large instances. For the last problem, however, the bounding operation is rapidly performed and poor speedups are observed. When compared with an AMP approach, the algorithm is shown to be competitive when solving the TSP, but is completely outperformed when solving the Chinese postman problem.

Eckstein (1994b, c) implements an ASP algorithm on the CM-5, an asynchronous message-passing system. The code bears some resemblance to commercial sequential packages for solving general mixed

integer programming problems. The global work pool is managed in a similar way as in McKeown et al., where the master process knows only the priority of each subproblem, and the processor and memory address of the remaining information. This distributed-memory scheme leads to frequent asynchronous communications between slaves and makes heavy use of the speed and general-purpose topology of the CM-5 architecture. Experiments on several real-world data instances with up to 128 processors show near-linear speedups for many hard problems, or run times reduced to the order of seconds for easier problems. Preliminary results obtained with an AMP implementation are also given and demonstrate the superiority of the ASP approach on most instances.

Pekny and Miller (1990; see also Pekny 1989, Miller and Pekny 1989, Pekny and Miller 1990, Balas et al. 1991, Kudva and Pekny 1993) propose an algorithm that differs from the basic model described at the beginning of this section. The global work pool is here implemented by using two lists: a list of generated subproblems not yet evaluated nor examined, and a list of evaluated subproblems not yet examined. A process looking for work will give priority to unevaluated subproblems. This scheme minimizes memory requirements, and also permits early identification of a good upper bound, because an efficient upper bounding procedure is called when evaluating every subproblem. Experiments are reported on a 10-processor BBN Butterfly system for solving the asymmetric TSP. Instead of focusing on the speed-up performance of their algorithm, the authors emphasize solving notably difficult very large instances that could not be solved previously in a reasonable amount of time by using traditional BB algorithms in a sequential environment. They solve randomly generated instances of up to 10,000 cities (one instance of this size being solved in about twenty minutes), and notably difficult instances of up to 3,000 cities that are designed to confound neighborhood search heuristics.

3.3.3. Synchronous Multiple Pool Algorithms

Very few algorithms of this type have been implemented. Pardalos and Rodgers (1989, 1990; see also Rodgers 1989) propose a collegial algorithm where processes synchronize after each has performed *MAXV* iterations, or has emptied its work pool. During the computation phase, the processes execute a depth-first procedure using the subproblems in their pool. During the communication phase, the processes first exchange their status, "free" if the associated work pool is empty, "busy" otherwise; then, when a "busy" process identifies a "free" one, a subproblem exchange is attempted. The algorithm is used to solve 0–1 unconstrained quadratic problems, and is implemented on two shared-memory systems, the four-processor Cray X-MP/48 and the six-processor IBM 3090-600E, and two message-passing hypercubes, a 32-processor iPSC/1 and a 16-processor iPSC/2. The authors experimentally observe that many synchronizations occur without any exchange of subproblems, and modify the algorithm to define a nonpreemptive version. When a process empties its work pool, it asks the other processes for a synchronization phase; processes enter in a synchronization phase only if they have performed at least *MAXV* iterations since the last synchronization phase, or if they have emptied their work pool. Contrary to the original version, the nonpreemptive version is not strictly synchronous, because processes may follow different paths, if the algorithm is not run under the same conditions for two consecutive executions.

Pargas and Wooster (1988) propose another loosely synchronous collegial algorithm for solving a job scheduling problem. These authors implement an initial work allocation strategy in which all processes build the same tree, and subsequently split tasks by allocating themselves some subproblems when their number becomes sufficiently large. Each process then performs its own depth-first procedure on the subproblems assigned during the initial phase. Processes synchronize at regular intervals to exchange bounds, and also for load balancing purposes. During the communication phase, if a process has no more units of work, it may take one of the subproblems initially allocated to another process. This scheme does not require any explicit exchange of subproblems, because the initial tree is accessible in the local memory of each process. The implementation of the algorithm on a 16-transputer FPS T-20 architecture must be termed loosely synchronous, because the communication and computation procedures are executed as concurrent processes on each transputer. Consequently, it is possible that two consecutive runs will not result in the same computations.

Laursen (1993b; see also Laursen 1991, 1993a) presents an SMP algorithm where processes (one per processor) not only exchange bounds and subproblems, but also information about their local times in order to minimize loss due to synchronization. At each step, every process communicates with a different neighbor (an interconnection network topology formed of perfect edge-disjoint matchings is assumed) and adjusts its local time to be the average of the two local times. In addition, two protocols for exchanging subproblems are presented. Both aim at equalizing

the workloads (measured as the number of subproblems in the local pool) of each pair of communicating processes. In one protocol, the heaviest process sends subproblems to the other without any further restrictions, while in the second protocol, exchange of subproblems occurs only when one of the processes has less than two subproblems in its pool. Experiments in solving many instances of the graph partitioning problem, the weighted VCP and the QAP on a network of 17 transputers are presented. They show that the two protocols are equally effective, but the second is to be preferred because it minimizes the amount of communication.

3.3.4. Asynchronous Multiple Pool Algorithms

In this section, we consider only AMP algorithms with a fixed number of processes (see Jansen and Sijstermans, for an example of an algorithm with a variable number of processes). Among the three main organization schemes for the location of work pools (subsection 2.2), the collegial approach has received much attention. However, a mixed organization appears to be an attractive alternative, particularly when the global work pool is used to share information among processes to speed up the search for an optimal solution, as verified by Kumar, Ramesh and Nageshwara Rao. The authors present a dynamic allocation strategy without request that is suited for implementation on shared-memory systems. In this strategy, there is a shared-memory location, called a blackboard, through which subproblems are exchanged among processes. After selecting a subproblem in its local work pool by using a best-first criterion, a process performs the branching operation only if the lower bound of the subproblem is within a "tolerable" limit of the best subproblem stored in the blackboard. If the selected subproblem is much better than the best subproblem in the blackboard, the process transfers some of its good subproblems to the blackboard. If the selected subproblem is much worse than the best subproblem in the blackboard, the process transfers some good subproblems from the blackboard to its local work pool. Results of experiments with many instances of the TSP and the VCP on a BBN Butterfly with up to 100 processors show the superiority of the blackboard strategy over two collegial algorithms based on dynamic allocation strategies without request. Other mixed organization algorithms that use a dynamic allocation strategy are presented by Kindervater (1989) and Mans, Mautor and Roucairol (1993), while mixed organization algorithms with a static allocation strategy are proposed by many authors: Zariffa (1986), Mraz and Seward (1987), Altmann, Marsland and Breikreutz (1988), Pardalos and Crouse (1989), Rost and Maehle (1988), Li and Pardalos (1992), Laursen (1991, 1993a, b). Very few algorithms that propose a grouped organization scheme were implemented: Gulyanitskii, Sergienko and Khodzinskii (1989), Schwan, Gawkowski and Blake (1988), Schwan et al. (1989a, b), Kawaguchi and Maeda (1990), McKeown, Rayward-Smith and Turpin (1991), McKeown et al. (1991), Rayward-Smith, Rush and McKeown (1991).

Among collegial algorithms, we distinguish between those that use a coordinator process for communication and load balancing purposes, and those that are fully distributed (all processes being identical). Yang and Das (1991) use a coordinator to implement a static allocation strategy. The coordinator executes a sequential best-first algorithm until there are N unexamined subproblems. These subproblems are then distributed among N processes, each performing a best-first BB procedure. In Schwan, Gawkowski and Blake (1988) and Schwan et al. (1989a, b), the initial allocation strategy is similar, but the algorithm then switches to a dynamic allocation policy on request. When a process empties its list of unexamined subproblems, it sends a request to the coordinator which chooses a busy process and asks it to share work by sending the subproblem with the smallest lower bound in the associated work pool. The selection operation used by each process is a mixture of best-first and depth-first, and is shown to be more efficient than a pure best-first selection operation, particularly when the number of processes increases. Experimental results on an Intel IPSC/1 32-node hypercube are reported for TSP instances with up to 30 cities.

Gendron and Crainic (1993a) present another collegial algorithm that uses a coordinator to ease load balancing and termination detection. The initial allocation strategy consists of a synchronous procedure which may be seen as a distributed version of Mohan's SSP algorithm. At each step, p subproblems are created, one subproblem being assigned to each working process. After bounding operations are performed, the subproblem with the smallest lower bound among all evaluated subproblems is determined in a distributed fashion. Branching is performed on this subproblem and another step is executed unless the problem is solved or a given maximum number of steps is attained. The initial phase is typically run for a small number of steps, and the algorithm then switches to a completely asynchronous phase, where each working process performs its own depth-first search of a subtree. During this phase, when a process runs out of work, it sends a request to

the coordinator, which uses a round robin strategy to identify a granting process. The coordinator also periodically receives from each process the number of subproblems stored in their local memory. This information is used to eliminate as possible granting processes those that have insufficient workloads. The authors report on the results of experiments in solving the multicommodity location problem with balancing requirements on two message-passing architectures, a network of 16 transputers and a distributed system of five workstations, and show, in particular, that the initial phase eases load balancing.

In all fully distributed collegial algorithms proposed in the literature, a dynamic work allocation strategy is used. According to our classification, we distinguish between algorithms that use a strategy on request, without request, or a combined strategy.

Finkel and Manber (1987) propose a distributed implementation of backtracking, which may be used for depth-first BB algorithms. Each process performs its own backtracking procedure. When it runs out of work, it sends a request to another process. This process may be selected randomly, or by using a cyclic order (assuming there are p processes): Initially, each process i ($1 \leq i \leq p$) requests work from its successor, process $j = (i \bmod p) + 1$. If process j cannot grant the request, it forwards it to its own successor. When the request is granted, the successor of process i becomes the successor of the process that granted the request. In a variation of this strategy, each process sending a request to its successor does not wait until the request is granted. Instead, after a given amount of time, it sends a request to another process. Strategies are also tested with regard to the number of subproblems to transfer, and how to select them. The authors report preliminary results for the TSP (and other backtracking applications) on a network of 20 VAX-11/750 computers connected by a ring topology.

Nageshwara Rao and Kumar (1987) present a parallel depth-first algorithm, which may be used to implement BB algorithms, and that is similar in many aspects to that of Finkel and Manber. Initially, one process starts a depth-first procedure using the original problem. When a process has an empty work pool, it sends a request to another process, the choice of this process being architecture-dependent (it is assumed that one process is running per processor). When a process receives a request, it sends about half of all the nodes that are above a certain "cutoff" depth (to avoid sending subproblems that can be solved rapidly). The algorithm was implemented on two shared-memory systems, the 30-processor Sequent Balance and the 120-processor BBN Butterfly, and on a 128-node hypercube message-passing iPSC architecture, which was also used to simulate the behavior of 1- and 2-ring topologies (Kumar, Nageshwara Rao and Ramesh 1988). The only problem tested was the 15-puzzle problem which is often used to test the effectiveness of search methods that arise in artificial intelligence. Kumar and Nageshwara Rao (1987) analyze theoretically the efficiency of this general work allocation strategy as a function of the architecture being used, and evaluate how the size of the problem, defined as the total number of generated subproblems (assumed to be constant), should grow to maintain a given efficiency when the number of processes is increased. This is called the isoefficiency measure. The original analysis was subsequently expanded (Kumar and Nageshwara Rao 1989, 1990, Grama, Kumar and Nageshwara Rao 1991, Kumar, Grama and Nageshwara Rao 1991), and the algorithm was tested on many architectures and for other (not necessarily optimization) problems: The test pattern generation problem on a 128-processor Symult s2010 (a 2-D mesh topology) and on a network of 4 Sun workstations (Arvindam et al. 1991); the floorplan optimization problem in VLSI circuits on a 1024-node Ncube/10, a 128-processor Symult, and on a network of 16 Sun workstations (Arvindam, Kumar and Nageshwara Rao 1989); the tautology verification problem on the Ncube/2 and the Ncube/10 (Arvindam, Kumar and Nageshwara Rao 1990, Grama, Kumar and Nageshwara Rao 1991, Kumar, Grama and Nageshwara Rao 1991). This algorithmic framework was also used as a basis for studying the average speedup that can be obtained, and, in particular, the occurrence of superlinear speedups (Nageshwara Rao and Kumar 1988c, 1990). Abdelrahman and Mudge (1988) propose a collegial algorithm similar to that of Nageshwara Rao and Kumar, except that each process runs a best-first algorithm. It was tested on a 64-processor NCUBE/6 to solve 0–1 ILPP.

Vornberger (1986; see also Monien and Vornberger 1987) presents a distributed collegial algorithm implemented on a ring of sixteen Intel 8088 processors. The initial allocation strategy has each process building the same tree until p unexamined subproblems are available. Each process then selects one subproblem and starts its own best-first procedure. When its work pool becomes empty, a process sends a request to its left neighbor in the ring (one process being run per processor), which, if its work pool contains "reasonable" subproblems, gives subproblems to its right neighbor. A process also probes its neighbor at regular intervals, by sending the value of its best lower bound. If the neighbor finds that its own lower bound

is much better, it shares a part of its work pool. Other collegial distributed algorithms that use an allocation strategy on request are given by Ma, Tsung and Ma (1988) and Mans (1992b).

Quinn (1987, 1990) presents several allocation strategies without request for a distributed collegial best-first algorithm implemented on a hypercube topology. Initially, one process has the original problem in its work pool. At every iteration (which consists of a branching operation on one subproblem and bounding operations on the newly generated subproblems), each process sends to one of its neighbors one subproblem from its work pool according to a given criterion. Four criteria are proposed: Any one of the newly generated subproblems; the newly generated subproblem with the smallest lower bound; the subproblem with the second smallest lower bound among all subproblems in the work pool; the subproblem with the smallest lower bound among all subproblems in the work pool. The neighbor to which process j ($0 \leq j \leq 2^d$) sends a subproblem at iteration i is obtained by inverting bit ($i \bmod d$) of j. The four strategies were tested on an NCUBE/7 64-node hypercube with ten 30-vertex instances of the TSP. The third and fourth strategies prove to be best, because they tend to distribute evenly the amount of useful work performed by all processes.

Several allocation strategies without request are proposed by Troya and Ortega (1988, 1989a, b). In one of them, processes send each of their newly generated subproblems to a process randomly selected (see also Felten 1988 who uses the same strategy, and Karp and Zhang 1988, and Ranade 1990 who propose theoretical analyses of this load balancing approach). In other strategies, each newly generated subproblem is given a value $j \in \{0, 1\}$; a dichotomous branching scheme is assumed. One strategy then assigns subproblem j to the same process at every iteration, while another ensures, by a cyclical allocation scheme, that subproblem j will not be transferred to the same process at every iteration. Simulations show that the random strategy is inferior to the others.

Another type of strategy without request uses information obtained from other processes to decide on how to share work units. Vornberger (1987) introduces an allocation scheme in which a process i gives its best subproblem (a best-first strategy being used) to each neighbor j at every L iterations, where L depends on the value of the smallest lower bound most recently communicated by j. If the best subproblem in the work pool of process i has a smaller value than the one in the work pool of process j, L is set to a "low" value, otherwise it is set to a "high" value. Experimental results with ten instances of the VCP on a network of 32 transputers are presented. Another strategy without request that uses information from other processes to take decisions is presented by Anderson and Chen (1987).

Lüling and Monien (1989) were among the first to propose a combined strategy. Each process starts initially with one subproblem and performs a best-first procedure using the subproblems in its work pool. A "quality" measure associated with each work pool, called *weight*, is used to govern the sharing of work units among processes. The load balancing strategy makes use of several parameters that are assigned values at the beginning and remain fixed during the whole computation. Experiments with five instances of the VCP are performed on a network of 60 transputers and on a ring of 32 transputers. The number of generated subproblems in the work pool is used to measure the weight. Kröger and Vornberger (1990) use the same load-balancing strategy to solve the two-dimensional cutting stock problem on a network of 32 transputers. However, the weight is measured differently. Assume that $Q_1, \ldots, Q_n$ are the subproblems in the work pool of a given process i and *best_Z* is the best upper bound known by this process. The weight is then given by $w_i = \sum_{i=1}^{n} (best\_Z - Z^l(Q_i))^2$. The same weight measure is also used by Lüling and Monien (1992) in a subsequent paper. The authors add some ingredients to their basic strategy, but their main contribution is to introduce a controller process that modifies the values of the parameters during execution. They give results on two instances of the VCP and three instances of the weighted VCP on a network of 256 transputers. It results that this weight measure makes sense only when the lower bounds are distributed in large intervals, otherwise the number of generated subproblems appears to be more adequate.

Clausen and Träff (1988, 1991) present two distributed collegial algorithms for solving the graph partitioning problem. Two different bounding operations are used, one being qualified as "easy" and generating large trees, and the other being qualified as "tight" and generating small trees. Results of experiments on a 32-node iPSC hypercube reveal that a combined load balancing technique is more efficient than a strategy on request. They also show that the easy bounding operation performs better in a parallel environment than the tight bounding operation. McKeown et al. propose another combined work allocation strategy. A process exhausting its work pool sends a request to a neighboring process. When a process receives a request, it sends several subproblems to the requesting process if it has (or will soon have) subproblems in its work pool. Otherwise, it sends a message

indicating that it is idle. In addition to this basic strategy, each process sends its best subproblem to a neighboring process (it cycles between each of its neighbors) every *count* iterations. Results of experiments with the TSP and the Chinese postman problem on a network of up to 32 transputers show near-linear speedups, when the parameter *count* is adjusted adequately.

4. CONCLUSION

We have presented a state-of-the-art survey of parallel BB algorithms for solving NP-hard optimization problems. Our work differs in many aspects from previous contributions to the area. First, our survey is more complete and up-to-date. Second, we have proposed a new presentation of the BB algorithm, where the operations are isolated without specifying any order for their execution. The usefulness of this approach has been shown by stating a convergence theorem based on properties that hold for most exact sequential and parallel algorithms. Third, we have proposed a new classification of parallel BB algorithms which differs in many aspects from previous classifications proposed by Roucairol (1989) and Trienekens and de Bruin (1992). Roucairol considers asynchronous algorithms of two types: vertical and horizontal. Vertical algorithms correspond to the distributed collegial algorithms of our classification, while horizontal algorithms correspond to the ASP class of algorithms. Trienekens and de Bruin classify algorithms by using the notion of knowledge base, which is an entity that may contain, among other things, generated subproblems (examined or not), upper bounds, and feasible solutions found by the algorithm. The basic difference between this notion and our notion of work pool, is that a work pool contains only generated subproblems that are awaiting some treatment, while a knowledge base may contain any knowledge generated by the algorithm.

With regard to the design of BB algorithms for solving a given problem on particular parallel architectures, we can draw the following conclusions:

- Implementations on massively parallel fine-grained SIMD machines are appropriate only for problems with trivial bounding operations performed in constant time.
- Synchronization appears unnecessary in most cases.
- ASP algorithms are appropriate only for problems with a nontrivial bounding operation, and parallel architectures having a relatively small number of processors.
- Implementations of ASP algorithms on shared-memory architectures should benefit from the work of Nageshwara Rao and Kumar (1988a, b) and Le Cun, Mans and Roucairol (1993) on concurrent priority queue data structures, and of Pekny and Miller (1992) on two-list implementations.
- In AMP algorithms, dynamic combined strategies appear most promising for allocating work units among processes. In particular, the notion of weight of a work pool, introduced by Lüling and Monien (1989), seems particularly useful.
- In AMP algorithms, a mixed organization scheme for the location of work pools appears to be an attractive alternative to a collegial approach, particularly if the global work pool is used to improve the search for an optimal solution.

As possible extensions to the present study, we suggest the following:

- Study the impact of parallelism of type 3, particularly along the lines suggested by Miller and Pekny (1993), the idea being to use parallelism to diversify the search.
- Study algorithmic schemes that combine the three types of parallelism.
- Perform comparative studies of single pool and multiple pool algorithms for several problems (see, in particular, the pioneering work of McKeown et al. on the implementation of kernels on transputer networks).
- Perform comparative studies of several initial work allocation strategies.
- Implement new schemes for the dynamic allocation of work units, and perform comparative studies of them.

ACKNOWLEDGMENT

Financial support for this project was provided by the N.S.E.R.C. of Canada and the Fonds F.C.A.R. of the Province of Québec. Special thanks are due to Sylvie Hétu for her help in building the bibliography.

REFERENCES

ABDELRAHMAN, T. S. 1988. Parallel Best-First Branch and Bound Algorithms on Distributed Memory Multiprocessors. Ph.D. Thesis, University of Michigan, Ann Arbor.

ABDELRAHMAN, T. S., AND T. N. MUDGE. 1988. Parallel Branch-and-Bound Algorithms on Hypercube Multiprocessors. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol.* II: *Applications*, 1492–1499.

AGIN, N. 1966. Optimum Seeking With Branch-and-Bound. *Mgmt. Sci.* **13,** B-176–B-185.

ALTMANN, E., T. A. MARSLAND AND T. BREIKREUTZ. 1988. Accounting for Parallel Tree Search Overheads. In *Proceedings of the 1988 International Conference on Parallel Processing*, 198–201.

ANANTH, G. Y., V. KUMAR AND P. M. PARDALOS. 1993. Parallel Processing of Discrete Optimization Problems. In *Encyclopedia on Microcomputers*. Marcel Dekker.

ANDERSON, S., AND M. C. CHEN. 1987. Parallel Branch-and-Bound Algorithms on the Hypercube. In *Hypercube Multiprocessors*, M. T. Heath (ed.). SIAM Press, Philadelphia, 309–317.

ARVINDAM, S., V. KUMAR AND V. NAGESHWARA RAO. 1989. Floorplan Optimization on Multiprocessors. In *Proceedings of IEEE International Conference on Computer Design*, 109–114.

ARVINDAM, S., V. KUMAR AND V. NAGESHWARA RAO. 1990. Efficient Parallel Algorithms for Search Problems: Applications to VLSI CAD. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, 166–169.

ARVINDAM, S., V. KUMAR, V. NAGESHWARA RAO AND V. SINGH. 1991. Automatic Test Pattern Generation on Parallel Processors. *Paral. Comput.* **17,** 1323–1342.

BALAS, E. 1968. A Note on the Branch-and-Bound Principle. *Opns. Res.* **16,** 442–445.

BALAS, E., D. MILLER, J. PEKNY AND P. TOTH. 1991. A Parallel Shortest Augmenting Path Algorithm for the Assignment Problem. *J. ACM* **38,** 985–1004.

BARR, R. S., AND B. L. HICKMAN. 1993. Reporting Computational Experiments With Parallel Algorithms: Issues, Measures and Experts' Opinions. *ORSA J. Comput.* **5,** 2–18.

BARR, R. S., AND W. STRIPLING. 1992. A Parallel Mixed-Strategy Branch-and-Bound Approach to the Fixed-Charge Transportation Problem. Technical Report 92-CSE-19, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas.

BERTIER, P., AND B. ROY. 1964. *Procédure de résolution pour une classe de problèmes pouvant avoir un caractère combinatoire*. Cahiers du Centre d'études de recherche opérationnelle **6**(4), 202–208.

BERTSEKAS, D. P., AND J. N. TSITSIKLIS. 1989. *Parallel and Distributed Computation, Numerical Methods*. Prentice-Hall, Englewood Cliffs, N. J.

BOEHNING, R. L., R. M. BUTLER AND B. E. GILLETT. 1988. A Parallel Integer Linear Programming Algorithm. *Eur. J. Opnl. Res.* **34,** 393–398.

BOXMA, O. J., AND G. A. P. KINDERVATER. 1991. A Queueing Network Model for Analyzing a Class of Branch-and-Bound Algorithms on a Master-Slave Architecture. *Opns. Res.* **39,** 1005–1017.

BURTON, F. W., G. P. MCKEOWN, V. J. RAYWARD-SMITH AND M. R. SLEEP. 1982. *Parallel Processing and Combinatorial Optimization, Combinatorial Optimization III*, L. B. Wilson, C. S. Edwards and V. J. Rayward-Smith (eds.). University of Stirling, U.K., 19–36.

BURTON, F. W., M. M. HUNTBACH, G. P. MCKEOWN AND V. J. RAYWARD-SMITH. 1983. Parallelism in Branch-and-Bound Algorithms. Internal Report CSA/3/1983, School of Computing Studies and Accountancy, University of East Anglia, Norwich, U.K.

CANNON, T. L. 1988. Large-Scale Zero-One Linear Programming on Distributed Workstations. Ph.D. Thesis, Department of Operations Research and Applied Statistics, George Mason University, Fairfax, Virginia.

CANNON, T. L., AND K. L. HOFFMAN. 1990. Large-Scale 0–1 Linear Programming on Distributed Workstations. *Ann. Opns. Res.* **22,** 181–217.

CHENG, K. H., AND Q. WANG. 1990. An Asynchronous Multiprocessor Design for Branch-and-Bound Algorithms. In *Proceedings of the Frontiers of Massively Parallel Computation*, 65–68.

CLAUSEN, J., AND J. L. TRAFF. 1988. Parallel Graph Partitioning Using Branch-and-Bound With Dynamic Distribution of Subproblems. DIKU Report 88/18, Department of Computer Science, University of Copenhagen, Denmark.

CLAUSEN, J., AND J. L. TRAFF. 1991. Implementations of Parallel Branch-and-Bound Algorithms—Experiences With the Graph Partitioning Problem. *Anns. Opns. Res.* **33,** 331–349.

DE BRUIN, A., A. H. G. RINNOOY KAN AND H. W. J. M. TRIENEKENS. 1988. A Simulation Tool for the Performance of Parallel Branch and Bound Algorithms. *Math. Prog.* **42,** 245–271.

DEHNE, F., A. G. FERREIRA AND A. RAU-CHAPLIN. 1989a. Parallel Branch-and-Bound on Fine-Grained Hypercube Multiprocessors. *Tools for Artif. Intell.*, 616–622.

DEHNE, F., A. G. FERREIRA AND A. RAU-CHAPLIN. 1989b. Parallel Branch-and-Bound on Fine-Grained Hypercube Multiprocessors. *Paral. Comput.* **15,** 201–209.

DESAI, B. C. 1977. A Parallel Processing System to Solve 0–1 Programming Problem. Ph.D. Thesis, McGill University, Montreal, Canada.

DESAI, B. C. 1978. The BPU: A Staged Parallel Processing System to Solve the Zero-One Problem. In *Proceedings of ICS*, Taipei, Taiwan, December 1978, 802–817.

DESAI, B. C. 1979. A Parallel Microprocessing System. In *Proceedings of the 1979 International Conference on Parallel Processing*, p. 136.

DE WITT, D., R. FINKEL AND R. SOLOMON. 1984. The Crystal Multicomputer: Design and Implementation Experience. *IEEE Trans. Soft. Engin.* **SE-13,** 953–967.

DIJKSTRA, E. W., AND C. S. SHOLTEN. 1980. Termination Detection for Diffusing Computations. *Infor. Proc. Letts.* **11,** 1–4.

ECKSTEIN, J. 1994a. Large-Scale Parallel Computing, Optimization, and Operations Research: A Survey. *ORSA CSTS Newsletter* **14,** (2), 1–28.

ECKSTEIN, J. 1994b. Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5. Technical Report TMC-257, *SIAM J. Optim.* (to appear).

ECKSTEIN, J. 1994c. Parallel Branch-and-Bound for Mixed Integer Programming. *SIAM News* **27**(1), 12–15.

EL-DESSOUKI, O. I., AND W. H. HUEN. 1980. Distributed Enumeration on Network Computers. *IEEE Trans. Comput.* **C-29,** 818–825.

FELTEN, E. W. 1988. Best-First Branch-and-Bound on a Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. II: Applications*, 1500–1504.

FINKEL, R., AND U. MANBER. 1987. DIB—A Distributed Implementation of Backtracking. *ACM Trans. Prog. Lang. and Syst.* **9**(2), 235–256.

FLYNN, M. J. 1966. Very High-Speed Computing Systems. *Proc. IEEE* **54,** 1901–1909.

FOX, B. L., J. K. LENSTRA, A. H. G. RINNOOY KAN AND L. E. SCHRAGE. 1978. Branching From the Largest Upper Bound: Folklore and Facts. *Eur. J. Opnl. Res.* **2,** 191–194.

FULLER, S. H., J. K. OUSTERHOUT, L. RASKIN, P. I. RUBINFELD, P. J. SINDHU AND R. J. SWAN. 1978. Multi-Microprocessors: An Overview and Working Example. *Proc. IEEE* **66**(2), 216–228.

GENDRON, B. 1991. *Implantations parallèles d'un algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage*, M.Sc. Thesis, Publication No. 761, Centre de recherche sur les transports, University of Montreal, Canada.

GENDRON, B., AND T. G. CRAINIC. 1993a. A Parallel Branch-and-Bound Algorithm for Multicommodity Uncapacitated Location With Balancing Requirements. Publication No. 924, Centre de recherche sur les transports, University of Montreal, Canada.

GENDRON, B., AND T. G. CRAINIC. 1993b. Parallel Implementations of a Branch-and-Bound Algorithm for Multicommodity Location With Balancing Requirements, *INFOR* **31**(3), 151–165.

GEOFFRION, A. M., AND R. E. MARSTEN. 1972. Integer Programming: A Framework and State-of-the-Art Survey. *Mgmt. Sci.* **18,** 465–491.

GRAMA, A., V. KUMAR AND V. NAGESHWARA RAO. 1991. Experimental Evaluation of Load Balancing Techniques for the Hypercube. *Proceedings of the Parallel Computing 91 Conference*, 497–514.

GRAMA, A. Y., V. KUMAR AND P. M. PARDALOS. 1993. Parallel Processing of Discrete Optimization Problems. In *Encyclopedia of Microcomputers*, Marcel Dekker, New York, 129–147.

GULYANITSKII, L. F., I. V. SERGIENKO AND A. N. KHODZINSKII. 1989. Discrete Optimization Methods for Multiprocessor Computer Systems. *Cybernetics* **24,** 418–427.

HARRIS, J. A., AND D. R. SMITH. 1977. Hierarchical Multiprocessor Organizations. *Proceedings of the 4th Annual Symposium on Computer Architecture*, 41–48.

HUANG, S.-R., AND L. DAVIS. 1987. A Tight Upper Bound for the Speedup of Parallel Best-First Branch-and-Bound Algorithms. Technical Report, Center on Automation Research, University of Maryland, College Park.

IBARAKI, T. 1977. The Power of Dominance Relations in Branch-and-Bound Algorithms. *J. ACM* **24,** 264–279.

IBARAKI, T. 1987. Enumerative Approaches to Combinatorial Optimization. *Anns. Opns. Res.*, 10–11.

IMAI, M., T. FUKUMURA AND Y. YOSHIDA. 1979. A Parallelized Branch-and-Bound Algorithm, Implementation and Efficiency. *Syst. Comput. Controls* **10**(3), 62–70.

IMAI, M., Y. YOSHIDA AND T. FUKUMURA. 1979. A Parallel Searching Scheme for Multiprocessor Systems and its Application to Combinatorial Problems. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, 416–418.

JANAKIRAM, V. K., D. P. AGRAWAL AND R. MEHROTRA. 1988a. A Randomized Parallel Branch-and-Bound Algorithm. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. III: Algorithms and Applications*, 69–75.

JANAKIRAM, V. K., D. P. AGRAWAL AND R. MEHROTRA. 1988b. A Randomized Parallel Backtracking Algorithm. *IEEE Trans. Comp.* **37**(12), 1665–1676.

JANAKIRAM, V. K., E. F. GEHRINGER, D. P. AGRAWAL AND R. MEHROTRA. 1988. A Randomized Parallel Branch-and-Bound Algorithm. *Int. J. Paral. Prog.* **17**(3), 277–301.

JANSEN, J. M., AND F. W. SIJSTERMANS. 1989. Parallel Branch-and-Bound Algorithms. *Future Generation Comp. Syst.* **4,** 271–279.

KARP, R. M., AND Y. ZHANG. 1988. A Randomized Parallel Branch-and-Bound Procedure. *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, 290–300.

KAWAGUCHI, T., AND T. MAEDA. 1990. A Parallel Branch-and-Bound Algorithm for a Torus Machine. *Syst. and Comp. in Japan* **21**(3), 101–108.

KINDERVATER, G. A. P. 1989. Exercises in Parallel Combinatorial Computing. Ph.D. Thesis, Centre for Mathematics and Computer Science, Amsterdam.

KINDERVATER, G. A. P., AND J. K. LENSTRA. 1985. Parallel Algorithms. In *Combinatorial Optimization: Annotated Bibliographies*, M. O'hEigeartaigh, J. K.

Lenstra and A. H. G. Rinnooy Kan (eds.). John Wiley, New York, 106–128.

KINDERVATER, G. A. P., AND J. K. LENSTRA. 1986. An Introduction to Parallelism in Combinatorial Optimization. *Discr. Appl. Math.* **14,** 135–156.

KINDERVATER, G. A. P., AND J. K. LENSTRA. 1988. Parallel Computing in Combinatorial Optimization. *Anns. Opns. Res.* **14,** 245–289.

KINDERVATER, G. A. P., AND H. W. J. M. TRIENEKENS. 1988. Experiments With Parallel Algorithms for Combinatorial Problems. *Eur. J. Opnl. Res.* **33,** 65–81.

KOHLER, W. H., AND K. STEIGLITZ. 1974. Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems. *J. ACM* **21**(1), 140–156.

KOHLER, W. H., AND K. STEIGLITZ. 1976. Enumerative and Iterative Computational Approaches. In *Computer and Job-Shop Scheduling Theory*, E. G. Coffman, Jr. (ed.). John Wiley, New York, 229–287.

KROGER, B., AND O. VORNBERGER. 1990. A Parallel Branch-and-Bound Approach for Solving a Two-Dimensional Cutting Stock Problem. Technical Report, Department of Mathematics and Computer Science, University of Osnabrück, Germany.

KUDVA, G. K., AND J. F. PEKNY. 1993. A Distributed Exact Algorithm for the Multiple Resource Constrained Sequencing Problem. *Anns. Opns. Res.* **42,** 25–54.

KUMAR, V., A. GRAMA AND V. NAGESHWARA RAO. 1991. Scalable Load Balancing Techniques for Parallel Computers. Technical Report TR-91-55, Computer Science Department, University of Minnesota, Minneapolis.

KUMAR, V., AND L. N. KANAL. 1984. Parallel Branch-and-Bound Formulations for AND/OR Tree Search. *IEEE Trans. Pattern Anal. and Mach. Intel.* **PAMI-6**(6), 768–778.

KUMAR, V., AND V. NAGESHWARA RAO. 1987. Parallel Depth First Search. Part II. Analysis. *Int. J. Paral. Prog.* **16**(6), 501–519.

KUMAR, V., AND V. NAGESHWARA RAO. 1989. Load Balancing on the Hypercube Architecture. *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, 603–608.

KUMAR, V., AND V. NAGESHWARA RAO. 1990. Scalable Parallel Formulations of Depth-First Search. In *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan and L. Kanal (eds.). Springer-Verlag, New York, 1–41.

KUMAR, V., V. NAGESHWARA RAO AND K. RAMESH. 1988. Parallel Depth First Search on the Ring Architecture. Technical Report TR-88-16, Department of Computer Sciences, The University of Texas at Austin.

KUMAR, V., K. RAMESH AND V. NAGESHWARA RAO. 1988. Parallel Best-First Search of State-Space Graphs: A Summary of Results. *Proceedings of the Seventh National Conference on Artificial Intelligence* **1,** 122–127.

LAI, T.-H., AND S. SAHNI. 1982. Anomalies in Parallel Branch-and-Bound Algorithms. Technical Report, University of Minnesota, Minneapolis.

LAI, T.-H., AND S. SAHNI. 1983. Anomalies in Parallel Branch-and-Bound Algorithms. *Proceedings of the 1983 International Conference on Parallel Processing*, 183–190.

LAI, T.-H., AND S. SAHNI. 1984. Anomalies in Parallel Branch-and-Bound Algorithms. *Commun. ACM* **27,** 594–602.

LAI, T.-H., AND A. SPRAGUE. 1985a. Performance of Parallel Branch-and-Bound Algorithms. *Proceedings of the 1985 International Conference on Parallel Processing*, 194–201.

LAI, T.-H., AND A. SPRAGUE. 1985b. Performance of Parallel Branch-and-Bound Algorithms. *IEEE Trans. Comp.* vol. **C-34**962–964.

LAI, T.-H., AND A. SPRAGUE. 1986. A Note on Anomalies in Parallel Branch-and-Bound Algorithms With One-to-One Bounding Functions. *Infor. Proc. Letts.* **23,** 119–122.

LAURSEN, P. S. 1991. New Parallel Branch and Bound for the Quadratic Assignment Problem. Master Thesis, DIKU-Report 91-9-7, Department of Computer In Science, University of Copenhagen, Denmark.

LAURSEN, P. S. 1993a. Simple Approaches to Parallel Branch and Bound. *Paral. Comput.* **19,** 143–152.

LAURSEN, P. S. 1993b. A Parallel Branch and Bound Algorithm With Synchronized Communication. Research Report, Department of Computer Science, University of Copenhagen, Denmark.

LAURSEN, P. S. 1994. Can Parallel Branch and Bound Without Communication be Effective? *SIAM J. Optim.* **4**(2), 288–296.

LAVALLEE, I., AND C. ROUCAIROL. 1985. Parallel Branch and Bound Algorithms. Rapport Interne MASI No. 112, Université Paris VI.

LAWLER, E. L., AND D. E. WOOD. 1966. Branch-and-Bound Methods: A Survey. *Opns. Res.* **14,** 699–719.

LE CUN, B., B. MANS AND C. ROUCAIROL. 1993. Comparison of Some Concurrent Priority Queues for Branch and Bound Algorithms. Working Paper, INRIA, France.

LI, G.-J. 1985. Parallel Processing of Combinatorial Search Problems. Ph.D. Thesis, Purdue University, West Lafayette, Indiana.

LI, G.-J., AND B. W. WAH. 1984a. Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms. Technical Report TR-84-6, School of Electrical Engineering, Purdue University, West Lafayette, Indiana.

LI, G.-J., AND B. W. WAH. 1984b. Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms. *Proceedings of the 1984 International Conference on Parallel Processing*, 473–480.

LI, G.-J., AND B. W. WAH. 1984c. How to Cope With Anomalies in Parallel Approximate Branch-and-Bound Algorithms. *Proceedings of the National Conference on Artificial Intelligence*, 212–215.

LI, G.-J., AND B. W. WAH. 1986a. Coping With Anomalies in Parallel Branch-and-Bound Algorithms. *IEEE Trans. Comp.* **C-35**(6), 568–573.

LI, G.-J., AND B. W. WAH. 1986b. How Good Are Parallel and Ordered Depth-First Searches? *Proceedings of the 1986 International Conference on Parallel Processing*, 992–999.

LI, G.-J., AND B. W. WAH. 1990. Computational Efficiency of Parallel OR-Tree Searches. *IEEE Trans. Soft. Engin.* **16,** 13–31.

LI, Y., AND P. M. PARDALOS. 1992. Parallel Algorithms for the Quadratic Assignment Problem. In *Advances in Optimization and Parallel Computing*, P. M. Pardalos (ed.). Elsevier Science Publishers, 177–189.

LÜLING, R., AND B. MONIEN. 1989. Two Strategies for Solving the Vertex Cover Problem on a Transputer Network. Third International Workshop on Distributed Algorithms, Lecture Notes in Computer Science, No. 392, 160–170.

LÜLING, R., AND B. MONIEN. 1992. Load Balancing for Distributed Branch and Bound Algorithms. *Proceedings of the Sixth International Parallel Processing Symposium*, 543–548.

MA, R. P., F.-S. TSUNG AND M.-H. MA. 1988. A Dynamic Load Balancer for a Parallel Branch and Bound Algorithm. *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. II: Applications*, 1505–1513.

MANS, B. 1992a. *Contribution à l'algorithmique non numérique parallèle: parallélisation de méthodes de recherche arborescentes*. Ph.D. Thesis, Université Paris VI, Paris, France.

MANS, B. 1992b. *Un algorithme d'équilibrage de charge dynamique et autoadaptatif pour le branch and bound*. Technical Report MASI 92.67, Laboratoire de Méthodologie et Architecture des Systèmes Informatiques, Institut Blaise Pascal, Paris, France.

MANS, B., AND C. ROUCAIROL. 1993. Performances of Parallel Branch-and-Bound Algorithms With Best-First Search. Working Paper, INRIA-MASI, Paris, France.

MANS, B., T. MAUTOR AND C. ROUCAIROL. 1993. A Parallel Depth First Search Branch and Bound for the Quadratic Assignment Problem. Working Paper, INRIA-MASI, Paris, France.

MCKEOWN, G. P., V. J. RAYWARD-SMITH AND H. J. TURPIN. 1991. Branch-and-Bound as a Higher-Order Function. *Anns. Opns. Res.* **33,** 379–402.

MCKEOWN, G. P., V. J. RAYWARD-SMITH, S. A. RUSH AND H. J. TURPIN. 1991. Using a Transputer Network to Solve Branch-and-Bound Problems. *Transputing 91, Proceedings of the World Transputer User Group Conference*, 781–800.

MILLER, D. L., AND J. F. PEKNY. 1989. Results From a Parallel Branch-and-Bound Algorithm for the Asymmetric Traveling Salesman Problem. *O. R. Letts.* **8,** 129–135.

MILLER, D. L., AND J. F. PEKNY. 1993. The Role of Performance Metrics for Parallel Mathematical Programming Algorithms. *ORSA J. Comput.* **5,** 26–28.

MITTEN, L. G. 1970. Branch-and-Bound Methods: General Formulation and Properties. *Opns. Res.* **18,** 24–34.

MOHAMED, R. A. K. 1992. Parallel Branch and Bound for Mixed Integer Programming. Technical Report 92-CSE-10, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas.

MOHAN, J. 1982. A Study in Parallel Computation: The Traveling Salesman Problem. Report CMU-CS-82-136(R), Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn.

MOHAN, J. 1983. Experience With Two Parallel Programs Solving the Traveling Salesman Problem. *Proceedings of the 1983 International Conference on Parallel Processing*, 191–193.

MOHAN, J. 1984. Performance of Parallel Programs: Model and Analyses. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn.

MØLLER-NIELSEN, P., AND J. STAUNSTRUP. 1984. Experiments With a Multiprocessor. Report DAIMI PB-185, Computer Science Department, Aarhus University, Aarhus, Denmark.

MONIEN, B., AND O. VORNBERGER. 1987. Parallel Processing of Combinatorial Search Trees. *Proceedings of the International Workshop on Parallel Algorithms and Architectures*, Lecture Notes in Computer Science, No. 269, 60–69.

MRAZ, R. T., AND W. D. SEWARD. 1987. Performance Evaluation of Parallel Branch-and-Bound Search With the Intel iPSC Hypercube Computer. *Proceedings of Supercomputing '87* **III,** 82–91.

NAGESHWARA RAO, V., AND V. KUMAR. 1987. Parallel Depth First Search. Part I. Implementation. *Int. J. Paral. Prog.* **16**(6), 479–499.

NAGESHWARA RAO, V., AND V. KUMAR. 1988a. Concurrent Insertions and Deletions in a Priority Queue. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. III: Algorithms and Applications*, 207–211.

NAGESHWARA RAO, V., AND V. KUMAR. 1988b. Concurrent Access of Priority Queues. *IEEE Trans. Comp.* **37**(12), 1657–1665.

NAGESHWARA RAO, V., AND V. KUMAR. 1988c. Superlinear Speedup in State-Space Search. *Proceedings of the 1988 Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, No. 338.

NAGESHWARA RAO, V., AND V. KUMAR. 1990. On the Efficiency of Parallel Depth-First Search. Technical Report TR-90-55, Computer Science Department, University of Minnesota, Minneapolis.

NAU, D. S., V. KUMAR AND L. KANAL. 1984. General Branch-and-Bound, and its Relation to A* and AO*. *Artif. Intel.* **23,** 29–58.

NEMHAUSER, G. L., AND L. A. WOLSEY. 1988. *Integer and Combinatorial Optimization*. John Wiley, New York.

PARDALOS, P. M. (ED.). 1992. *Advances in Optimization and Parallel Computing*. North-Holland, Amsterdam.

PARDALOS, P. M., AND J. V. CROUSE. 1989. A Parallel Algorithm for the Quadratic Assignment Problem. *Proceedings of Supercomputing '89*, 351–360.

PARDALOS, P. M., AND G. P. RODGERS. 1989. Parallel Branch and Bound Algorithms for Unconstrained Quadratic Zero-One Programming. In *Impacts of Recent Computer Advances on Operations Research*, R. Sharda, B. L. Golden, E. Wasil, O. Balci and W. Stewart (eds.). Elsevier Science Publishing, 144–157.

PARDALOS, P. M., AND G. P. RODGERS. 1990. Parallel Branch-and-Bound Algorithms for Quadratic Zero-One Programs on the Hypercube Architecture. *Anns. Opns. Res.* **22,** 271–292.

PARDALOS, P., AND X. LI. 1990. Parallel Branch-and-Bound Algorithms for Combinatorial Optimization. *Supercomputer* **7**(5), 23–30.

PARDALOS, P. M., A. T. PHILLIPS AND J. B. ROSEN. 1992. *Topics in Parallel Computing in Mathematical Programming*. Science Press, New York.

PARGAS, R. P., AND E. D. WOOSTER. 1988. Branch-and-Bound Algorithms on a Hypercube. *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. II: Applications*, 1514–1519.

PEKNY, J. F. 1989. Exact Parallel Algorithms for Some Members of the Traveling Salesman Problem Family. Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Penn.

PEKNY, J. F., AND D. L. MILLER. 1990. A Parallel Branch-and-Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems. *Proceedings of the ACM Eighteenth Annual Computer Science Conference*, 56–62.

PEKNY, J. F., AND D. L. MILLER. 1992. A Parallel Branch-and-Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems. *Math. Prog.* **55,** 17–33.

PLATEAU, G., AND C. ROUCAIROL. 1989. A Supercomputer Algorithm for the 0–1 Multiknapsack Problem. In *Impacts of Recent Computer Advances on Operations Research*, R. Sharda, B. L. Golden, E. Wasil, O. Balci and W. Stewart (eds.). Elsevier Science Publishing, 144–157.

PRUUL, E. 1975. Parallel Processing and a Branch-and-Bound Algorithm. M.Sc. Thesis, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, N. Y.

PRUUL, E., G. L. NEMHAUSER AND R. A. RUSHMEIER. 1988. Branch-and-Bound and Parallel Computation: A Historical Note. *O. R. Letts.* **7,** 65–69.

QUINN, M. J. 1983. On the Speedup of Parallel Depth-First Branch-and-Bound Algorithms. Technical Report 83-9, Department of Computer Science, University of New Hampshire, Durham, N. H.

QUINN, M. J. 1987. Implementing Best-First Branch-and-Bound Algorithms on Hypercube Multicomputers. In *Hypercube Multiprocessors*, M. T. Heath (ed.). SIAM Press, Philadelphia, 318–326.

QUINN, M. J. 1990. Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer. *IEEE Trans. Comp.* **39**(3), 384–387.

QUINN, M. J., AND N. DEO. 1983. An Upper Bound for the Speedup of Parallel Branch-and-Bound Algorithms. *Proceedings of the Third Conference on Foundations of Software Technology and Theoretical Computer Science*, 488–504.

QUINN, M. J., AND N. DEO. 1986. An Upper Bound for the Speedup of Parallel Best-Bound Branch-and-Bound Algorithms. *BIT* **26,** 35–43.

RANADE, A. 1990. A Simpler Analysis of the Karp-Zhang Parallel Branch-and-Bound Method. Report UCB/CSD 90/586, Computer Science Division, University of California at Berkeley, California.

RAYWARD-SMITH, V. J., S. A. RUSH AND G. P. MCKEOWN. 1993. Efficiency Considerations in the Implementation of Parallel Branch-and-Bound. *Anns. Opns. Res.* **43,** 123–145.

RIBEIRO, C. C. 1987. Parallel Computer Models and Combinatorial Algorithms. *Anns. Discr. Math.* **31,** 325–364.

RINNOOY KAN, A. H. G. 1976. On Mitten's Axioms for Branch and Bound. *Opns. Res.* **24,** 1176–1178.

RODGERS, G. 1989. Algorithms for Unconstrained Quadratic Zero-One Programming on Contemporary Computer Architectures. Ph.D. Thesis, The Pennsylvania State University, University Park, Penn.

ROST, J., AND E. MAEHLE. 1988. Implementation of a Parallel Branch-and-Bound Algorithm for the Traveling Salesman Problem. *Proceedings of CONPAR 88*, 152–159.

ROUCAIROL, C. 1987a. *Du séquentiel au parallèle: la recherche arborescente et son application à la programmation quadratique en variables 0–1.* Thèse d'État, Université Paris VI, France.

ROUCAIROL, C. 1987b. A Parallel Branch-and-Bound Algorithm for the Quadratic Assignment Problem. *Discr. Appl. Math.* **18,** 211–225.

ROUCAIROL, C. 1989a. Parallel Branch-and-Bound Algorithms—An Overview. In *Parallel and Distributed Algorithms*, M. Cosnard, Y. Robert, P. Quinton and M. Raynal (eds.). Elsevier Science Publishers, Amsterdam, 153–163.

ROUCAIROL, C. 1989b. Parallel Computing in Combinatorial Optimization. *Comp. Phys. Reps.* **11,** 195–220.

SCHWAN, K., J. GAWKOWSKI AND B. BLAKE. 1988. Process and Workload Migration for a Parallel Branch-and-Bound Algorithms on a Hypercube Multicomputer. *Proceedings of the Third Conference on Hypercube Multiprocessors, Vol. II: Applications,* 1520–1530.

SCHWAN, K., B. BLAKE, W. BO AND J. GAWKOWSKI. 1989a. Global Data and Control in Multicomputers: Operating System Primitives and Experimentation With a Parallel Branch-and-Bound Algorithm. *Concurrency: Prac. and Exper.* **1**(2), 191–218.

SCHWAN, K., W. BO, B. BLAKE AND J. GAWKOWSKI. 1989b. OS Primitives for the Implementation of Distributed Objects in Multicomputers: Experimentation With a Parallel Branch-and-Bound Algorithm. *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications, Vol. II,* 785–791.

SEKIGUCHI, Y. 1981. A Unifying Framework of Combinatorial Optimization Algorithms: Tree Programming and its Validity. *J. Opns. Res. Soc. Japan* **24**(1), 67–93.

TRELEAVEN, P. C., D. R. BROWNBRIDGE AND R. P. HOPKINS. 1982. Data-Driven and Demand-Driven Computer Architectures. *ACM Comput. Surv.* **14,** 93–143.

TRIENEKENS, H. W. J. M. 1986. Parallel Branch and Bound on an MIMD System. Report 8640/A, Econometric Institute, Erasmus University, Rotterdam, The Netherlands.

TRIENEKENS, H. W. J. M. 1989a. Parallel Branch and Bound and Anomalies. Report EUR-CS-89-01, Computer Science Department, Faculty of Economics, Erasmus University, Rotterdam, The Netherlands.

TRIENEKENS, H. W. J. M. 1989b. Computational Experiments With an Asynchronous Parallel Branch and Bound Algorithm. Report EUR-CS-89-02, Computer Science Department, Faculty of Economics, Erasmus University, Rotterdam, The Netherlands.

TRIENEKENS, H. W. J. M. 1990. Parallel Branch and Bound Algorithms. Ph.D. Thesis, Erasmus University, Rotterdam, The Netherlands.

TRIENEKENS, H. W. J. M., AND A. DE BRUIN. 1992. Towards a Taxonomy of Parallel Branch and Bound Algorithms. Report EUR-CS-92-01, Computer Science Department, Faculty of Economics, Erasmus University, Rotterdam, The Netherlands.

TROYA, J., AND M. ORTEGA. 1988. A Parallel Best Bound First Branch-and-Bound Scheme. *Mini and Microcomp. and Their Applic.* 509–512.

TROYA, J., AND M. ORTEGA. 1989a. Live Nodes Distribution in Parallel Branch-and-Bound Algorithms. *Microproc. and Microprog.* **25,** 301–306.

TROYA, J., AND M. ORTEGA. 1989b. A Study of Parallel Branch-and-Bound Algorithms With Best-Bound-First Search. *Paral. Comput.* **11,** 121–126.

VORNBERGER, O. 1986. Implementing Branch-and-Bound in a Ring of Processors, *Proceedings of CONPAR 86 Conference on Algorithms and Hardware for Parallel Processing*, Lecture Notes in Computer Science, No. 237, 157–164.

VORNBERGER, O. 1987. Load Balancing in a Network of Transputers. Second International Workshop on Distributed Algorithms, Lecture Notes in Computer Science, No. 312, 116–126.

WAH, B. W., G.-J. LI AND C.-F. YU. 1984. The Status of MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems. *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 56–63.

WAH, B. W., G.-J. LI AND C.-F. YU. 1985. Multiprocessing of Combinatorial Search Problems. *IEEE Comp.*, June 1985, 93–108.

WAH, B. W., AND Y. W. MA. 1981. MANIP—A Parallel Computer System for Implementing Branch and Bound Algorithms. *Proceedings of the 8th Annual Symposium on Computer Architecture*, 239–262.

WAH, B. W., AND Y. W. MA. 1984. MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems. *IEEE Trans. Comp.* **C-33**(5), 377–390.

WEIDE, B. W. 1982. Modeling Unusual Behavior of Parallel Algorithms. *IEEE Trans. Comp.* **C-31**(11), 1126–1130.

YANG, M. K., AND C. R. DAS. 1991. A Parallel Branch-and-Bound Algorithm for MIN-Based Multiprocessors. *Perf. Eval. Rev.* **19**(1), 222–223.

ZARIFFA, N. 1986. Implementation and Analysis of Three Parallel Branch-and-Bound Algorithms for the Vertex Covering Problem. M.Sc. Thesis, School of Computer Science, McGill University, Montreal, Canada.