DIMACS Technical Report 95–03

Parallel Best-First Branch-and-Bound in Discrete Optimization: a Framework[1]

by

Ricardo Corrêa[2]
LMC – IMAG
46, av. Félix Viallet, 38031
Grenoble Cedex, France
Ricardo.Correa@imag.fr

Afonso Ferreira[3]
CNRS – LIP – ENS-Lyon
46, allée d'Italie, 69364
Lyon Cedex 07, France
ferreira@lip.ens-lyon.fr

[1]This work was partially supported by DIMACS NSF grant STC-91-199999 and the NJ Commission on Science and Technology, by DRET, by the project Stratagème of the French CNRS, and by the Human Capital and Mobility project SCOOP of the European Union. A revised version of this report is to appear in *Solving Combinatorial Optimization Problems in Parallel I: Methods*, Springer-Verlag.
[2]Partially supported by a CNPq (Brazil) fellowship, grant 201421/92-5 (BC).
[3]Visiting member.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

In this report, parallelism is used as a way to solve *discrete optimization problems*. We search for an *optimal solution* $x^* \in S^*$, where S^* is the set of all best solutions in a *domain* S, defined as the discrete set of all vectors x in the solution space that satisfy a set of constraints. Improving the search efficiency is of considerable importance since exhaustive search is often impracticable. The method called *branch-and-bound* (noted B&B) is a tree search algorithm often used as an intelligent search in this context. Its principle lies in successive decompositions of the original problem in smaller disjoint subproblems until an optimal solution is found. The algorithm consists of a heuristic iterative search that avoids visiting some subproblems which are known not to contain an optimal solution.

Parallel processing has been widely studied as an additional source of improvement in search efficiency in discrete optimization. We review in this report the literature pertinent to *parallel branch-and-bound algorithms*. The focus is on *distributed memory* parallel systems, which are composed of a set of processors connected by a physical network, each one with its own local memory. The communications between two different processors are implemented through the exchange of messages over links of the network connecting the two processors. Given that an attractive feature is that disjoint subproblems can be decomposed simultaneously and independently, the challenge is how to use the set of processors to improve the search efficiency of B&B algorithms, concurrently decomposing several subproblems at each iteration. In general terms, the potential to be explored consists of a linear – on the number of processors – reduction on the number of iterations. However, the reduction on the number of iterations can deviate considerably from linear due to possible *speedup anomalies*.

Parallel B&B is traditionally considered as an irregular parallel algorithm due to the fact that the structure of the search tree is not known beforehand. It may result in unnecessary work if a subproblem that does not contain an optimal solution is chosen and assigned to a processor to be decomposed. Therefore, the use of a distributed memory parallel system incurs a number of overheads, including communication overheads and idle time due to workload imbalance and contention on common data structures. Several special techniques have been developed to address these problems, essentially related to the amount of "necessary" work assigned to each processor. This report reviews these techniques, attempting to tie the area of parallel B&B under a common, uniform terminology. It can be profitable either to the parallel processing or to the operations research community members.

1 Introduction

The use of parallel algorithms for solving NP-hard problems becomes attractive as massively parallel systems consisting of a number of powerful processors offer large computation power and memory storage capacity (for a theory of NP-hardness, see [17]). Even though parallelism will not be able to overdue the assumed worst case exponential time or memory complexity of those problems (unless an exponential number of processors is used) [8], the average time complexity of heuristic search algorithms for many NP-hard problems is polynomial [56] (also, there are some heuristic search algorithms which find *suboptimal solutions* in polynomial time [59]). Consequently, those parallel systems, possibly with hundreds or thousands of processors, give us the perspective of efficiently solving relatively large instances of NP-hard problems, being a motivation for using parallel processing.

There is another important motivation for using parallel processing. In several industrial, research or other real-world environments, mathematical programmers must face up to moderate instances of NP-hard problems for which an *exact optimal solution* is highly desirable (e.g. VLSI floor-plan optimization [3]). In such circumstances, parallel processing can bring the time of solution from some days or months, that is typical when one workstation is used, to some minutes or seconds. This is crucial in some applications that require real time solutions (e.g. robot motion planning and speech understanding). These two motivations imply that it is of major importance the study of ways to parallelize existing sequential methods used for solving NP-hard problems.

Among the universe of NP-hard problems, in this report[1] *parallel branch-and-bound* is treated as a way to solve *discrete optimization problems*[2], which are stated as follows. Let n be a positive integer, $\mathbb{Z}^n$ be the *solution space* (nonnegative integral n-dimensional vectors)[3], S be a *domain* defined as the discrete set of all vectors x in the solution space that satisfy a set of constraints, and $f : S \to R$ be the *objective function*[4] from S onto a completely ordered set R. A typical constraint is written as $\sum_{i=1}^{n} a_i x_i \leq b$, where a_i and b are real numbers. We call $x \in S$ a *feasible solution*[5] and $f(x)$ the *value*[6] of x. A feasible solution x is *better* than $x' \in S$ if $f(x) < f(x')$. We search an *optimal solution*[7] $x^* \in S^*$ (we note $f^* = f(x^*)$), where S^* is the set of all best feasible solutions. Maximization problems can be stated and dealt with similarly. We assume S finite and non-empty.

The search for optimal solutions constitutes the most important problem in the scope of discrete optimization. Improving the search efficiency is of considerable importance since

[1] We use the notation most currently encountered in the literature. In case several different terms are used to designate identical concepts, we define the term we use and indicate the other terms as a footnote. We also highlight the definitions in order to make easy their identification in the text. A notation sheet can be found at the end of the report.

[2] Or *linear (pure) integer programming problem* [45, 46].

[3] If $\mathbb{Z}^n$ is replaced by $\{0,1\}^n$ we have a *combinatorial optimization problem*.

[4] Or *criterion function* [30].

[5] Or *answer node* [30].

[6] Or *cost* [30].

[7] Or *least-cost answer node* [30].

exhaustive search is often impracticable. The method called *branch-and-bound* (noted B&B) is a tree search algorithm often used as an intelligent search in this context. Its principle lies in successive decompositions of the original problem in smaller disjoint subproblems until an optimal solution is found. Each subproblem is represented by its domain, which is partitioned in a decomposition. A *search tree*[8] $T = (\mathcal{V}, \beta)$ describes these subproblems (the set of vertices $\mathcal{V}$, rooted at S) and the decomposition process (the set of edges β). The algorithm consists of a heuristic iterative search in T that avoids visiting some subproblems which are known not to contain an optimal solution, and is detailed in section 2. Backtracking, dynamic programming, A* and AND-OR tree search can be viewed as variations of B&B algorithms [20, 28, 43, 44].

Parallel processing has been widely studied as an additional source of improvement in search efficiency in discrete optimization. A variety of interconnection networks and parallel algorithms have been proposed for processor-processor or processor-memory communication, as ways for parallelizing sequential algorithms [27]. We review in this report the literature pertinent to modeling, performance characterization and implementation of *parallel branch-and-bound algorithms* for discrete optimization problems. The focus is on *distributed memory* parallel systems. These systems are composed of a set of p processors connected by a physical network, each one with its own local memory and with no physically shared memory. The communications between two different processors are implemented through the exchange of messages over links of the network connecting the two processors. Intel Paragon, Cray T3D, IBM SP-2, CM-5 and network of workstations are examples of distributed memory parallel systems.

From a parallel computing point of view, the challenge is how to use a set of processors to improve the search efficiency of B&B algorithms, given that an attractive feature is that disjoint subproblems can be decomposed simultaneously and independently. The principle of the (high level) parallelization considered in this report goes in this direction. It consists of concurrently decomposing several subproblems at each iteration [11, 30, 31, 34, 48, 57]. Other sources of parallelism exist, but they have not been widely explored [7, 57]. In general terms, the potential to be explored consists of a linear – on the number of processors – reduction on the number of iterations and, as a consequence, a linear improvement on the search efficiency of B&B algorithms. However, the reduction on the number of iterations can deviate considerably from linear due to possible *speedup anomalies* (details in section 4).

Parallel B&B is traditionally considered as an irregular parallel algorithm due to the fact that the structure of the search tree is not known beforehand [14]. The search in T involves heuristic choices of paths which determine subproblem(s), depending on the number of available processors, to decompose at each iteration. The efficiency of the algorithm depends on these choices, because they may result in unnecessary work if a subproblem that does not contain an optimal solution is chosen and assigned to a processor to be decomposed. Therefore, a distributed memory parallel system, executing a parallel B&B algorithm, incurs a number of overheads, including communication overheads and idle time due to workload imbalance and contention on common data structures. These overheads can degrade the

[8] Or *problem tree* [57].

performance. As a consequence, several special techniques have been developed to address the problems related to the irregularity of the search tree or of the parallel tree searching process, essentially related to the amount of "necessary" work assigned to each processor.

Unfortunately, an inevitable proliferation of terminology and parallel approaches came along with the numerous proposals in parallel B&B encountered in the literature. Thus, from the point of view of algorithm development, it is difficult to analyze the relative differences and merits of alternative schemes in a meaningful way. It is also difficult to focus common effort on approaches and areas of study which seem most likely to prove fruitful. Therefore, from the users viewpoint, it is not immediate to migrate neither from one scheme to another, nor from one parallel system to another, nor from one application to another. For these reasons, this report attempts to tie the area of parallel B&B under a common, uniform terminology. It can be profitable either to the parallel processing or to the operations research community members.

The remainder of the report reviews the literature in parallel best-first B&B, in its theoretical and practical aspects, and is organized as follows. The definition of the sequential B&B adopted is set in section 2, where the main differences in notation are underlined. Passing to the parallel approach, section 3 is devoted to models of parallel B&B which will be used in the sequel to classify the implementations. Before the practical approach, the main theoretical results are reviewed in sections 4 (speedup anomalies) and 5 (efficiency measures). Then, a large overview of published implementations is encountered in section 6, relating each implementation to its corresponding theoretical model. Finally, some open problems and challenges for further developments are presented in section 7. A large number of references is provided.

2 Preliminaries on sequential B&B

A sequential B&B algorithm consists of a sequence of *iterations*, in which some operators are applied over a *data structure* $\mathcal{D}$. This *data structure* keeps a priority list of subproblems [8, 40]. These operators choose a subproblem from $\mathcal{D}$, decompose it and eventually insert new subproblems into the *data structure*, chosen among those generated by the decomposition. Feasible solutions are found during the search in two situations. First, a decomposition may generate subproblems simple enough to be solved directly, determining a feasible solution for S. Also, given a subproblem v, one can find a feasible solution of S in v (not necessarily the best solution of v). The best feasible solution found so far is kept, as well as its value. At some points of the execution, some subproblems shown not to contain a feasible solution better than the current best known feasible solution are eliminated [11, 18, 23, 24, 40, 42, 57].

We call *open subproblems*[9] the subproblems that contain a feasible solution and that, at any point of the execution, were generated but neither decomposed nor eliminated. At each iteration, the *data structure* $\mathcal{D}$ is composed of a collection of open subproblems, called

[9] Or *live nodes* [30], *active subproblems* [57], *unexamined nodes* [52], *frontier nodes* [26].

openset[10], an *upper bound*[11] U, which is the best value found so far, and a feasible solution whose value is U. The execution of the algorithm starts with the data structure in the initial state $\mathcal{D}_0 = (\{S\}, U_0)$, where $U_0 \geq f^*$ is a constant corresponding to the initial upper bound (possibly infinite), and ends at the final state $(\emptyset, f^*)$. The operators applied at each iteration are four, namely, *selection*, *evaluation*, *insertion* and *decomposition*. We say that a processor *selects*, *decomposes* and *inserts* when it performs, respectively, a *selection*, a *decomposition* and an *insertion* on $\mathcal{D}$ when executing a sequential B&B algorithm.

The first operation of an iteration is *selection*. It consists of the rule for choosing new open subproblems to decompose, given a collection of subproblems.

Def. 1 (Selection) *Let V be the set of open subproblems at the beginning of an iteration. The* selection *function, applied to V, is any δ such that if $V \neq \emptyset$ then $\delta(V) \neq \emptyset$ (liveliness condition) and $\delta(\emptyset) = \emptyset$.*

The liveliness condition above points out that if there is an open subproblem at the beginning of an iteration, then at least one such a subproblem will be chosen for decomposition. A selected subproblem is called an *E-node* [30]. To each subproblem generated during the execution is associated a *lower bound*[12], defined in the following. It is calculated when the subproblem is generated by a *decomposition*.

Def. 2 (Lower bound) *The* lower bound *function l associates to each open subproblem v a value smaller than or equal to the value of the best feasible solution in v. The following conditions must hold:*

1. $l(v) \leq \min_{x \in v}\{f(x)\}$;
2. $l(v) = f(x)$, *if $v = \{x\}$ and x is a feasible solution;*
3. $l(v) \geq l(t)$, *if v is generated by the decomposition of t.*

Let h be a *heuristic priority function* that applied to an open subproblem determines its priority for selection. The subproblems of $\mathcal{D}$ are available in a nondecreasing order of h. We concentrate on *best-first*[13] search, where $h(v) = l(v)$ and δ selects the subproblem with smallest priority.

A search tree $T = (\mathcal{V}, \beta)$, with the domain S, the objective function f and a lower bound function l, is called a *BB-tree* [31]. The leaves of T are *solution nodes*[14], which solved directly in a decomposition. Let *worthwhile subproblem*[15] be a subproblem v so that $l(v) < f^*$ [52].

[10] Or *liveset* [31], *active set* [57].
[11] Or *incumbent* [31].
[12] Or *bounding function* [30].
[13] Or *least-cost branch-and-bound* [30], *best-bound-first* [31], *best-bound* [24, 52, 53]. The maximization version is also known as *branching from the largest upper bound* [16].
[14] Or *bottom nodes* [24], *terminal nodes* [40].
[15] Or *critical node* [30, 31].

The worthwhile subproblems of a BB-tree T form the *critical subtree* T^* of T [31, 40]. It is easy to see that all worthwhile subproblems must be selected during a B&B execution. A *critical path* is a path in T from the root to a solution node housing an optimal solution, and its length is its number of edges. A critical path with minimum length is a *minimum critical path*.

After the selection of a subproblem v, a *decomposition* creates a set of new subproblems.

Def. 3 *The* decomposition *operation of an E-node v in a B&B algorithm corresponds to:*

- *a partition of v, if v is not a solution node;*
- *a solution to optimality of v, otherwise.*

The definition above imposes that T is a tree[16]. For each subproblem generated by the decomposition, a lower bound is calculated. At an iteration, a new upper bound can be generated just after a decomposition in two ways. First, with a decomposition of a solution node. Second, a feasible solution can be found during the computation of a partition in a decomposition.

The goal of the third operation, *evaluation*, is to allow an intelligent search in S that avoids considering subproblems known not to be leading to an optimal solution of the original problem. We use the following lemma to describe it [11, 18, 23].

Lemma 1 *At an iteration, if $l(v) \geq U$ then v cannot lead to a feasible solution which is better than the current best known feasible solution.*

Def. 4 (Evaluation) *For each iteration, the* evaluation *eliminates all subproblems v such that $l(v) \geq U$.*

Finally, the inclusion of new open subproblems is accomplished by *insertion* operations.

Def. 5 (Insertion) *An open subproblem is included in $\mathcal{D}$ in such a way that the set of open subproblems remains sorted by the heuristic priority function h.*

In particular, in the sequential version, all the open subproblems generated by a decomposition are inserted.

Example 1 *Figure 1 illustrates a sequential execution over a BB-tree. The iteration where a subproblem is selected and decomposed is shown in the corresponding vertex of the tree. The two optimal solutions are also shown.*

Subproblem eliminations are also possible if the available information on subproblems can be used to show that an open subproblem v_j cannot include a better feasible solution for S than that obtainable from another open subproblem v_i. In this case, a *dominance relation* is verified, in the sense that v_i dominates v_j. Then, v_j can be eliminated of further decompositions [24].

---

[16] Although not in the scope of this report, there are other search algorithms, known as *graph search algorithms* (as A^* and its derivations), for which T is a graph because replication of subproblems are allowed [4, 49].

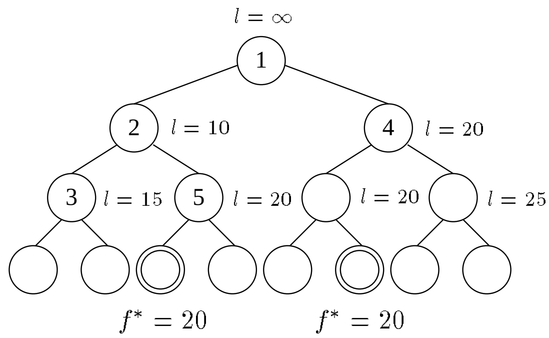

Figure 1: Example of a sequential B&B.

3 Analytic models for parallel B&B

In this section, we first establish our models for global data organization. Then we recall the definition of synchronous parallel B&B from the literature [9, 11, 30, 31, 34, 48], and we describe two general asynchronous models of parallel B&B. The difference between these two asynchronous models is the parallel computation model assumed, although both of them imply implementations that are modularly expandable [9, 58].

Essentially, the only two informations in a parallel B&B algorithm to be considered by all processors are the value of the current best solution and the heuristic order of open subproblems. The synchronous model uses these global informations. As it will be seen later, the heuristic order of open subproblems is used to ensure an efficient *qualitative workload sharing* at each iteration. However, even if at an iteration an incorrect current best solution is used by some processor as the upper bound or a subproblem is selected in an incorrect order, the algorithm still finds the optimal solution provided the value used as the current best solution is the value of some feasible solution. The asynchronous models seize these facts, relaxing the constraints of the synchronous model.

3.1 Parallel global data organization models

Two approaches will be used to model interprocessor interactions in order to treat global data: *shared data-object model (SDOM)* and *distributed data model (DDM)*. The SDOM allows processors to communicate *only* through a *global data structure*, which possesses exactly one priority list of subproblems [33]. On the other hand, in the DDM, each processor has its own *local data structure*, and the processors communicate by exchanging subproblems among the *local data structures*, each one with one priority list of subproblems. Some examples of implementations are summarized in section 6.

Distributed Data Model The DDM is a fully distributed approach, briefly described in figure 2. Its simple nature might lead the reader to believe that parallel B&B is trivial in this case: each processor sequentially searches on different and disjoint regions of T. Recall, however, that the structure of T is not known beforehand, and since subproblems are generated and selected in an unpredictable way in parallel B&B algorithms, irregularities appear during the parallel search. It is likely that a distributed algorithm will produce *search overhead* compared to the sequential algorithm because a processor can be assigned to a region of T which is not considered in sequential. In addition, a processor can eventually run out of work due to an evaluation.

Special techniques must be used to address the problems related to the irregularity of the search tree or of the parallel tree searching process. These problems are essentially related to *dynamic workload sharing*, where the amount of work must be evenly distributed over the processors. The amount of work can be defined in several different forms, but it is always related to subproblem decompositions. Several experiments have shown that, in many situation of interest, this dynamic workload sharing must take into account the heuristic function h (*qualitative workload sharing*) to minimize search overhead [1, 2, 12, 13, 19, 26, 27, 37]. Subproblems can migrate from one processor to another, or from a local data structure to another.

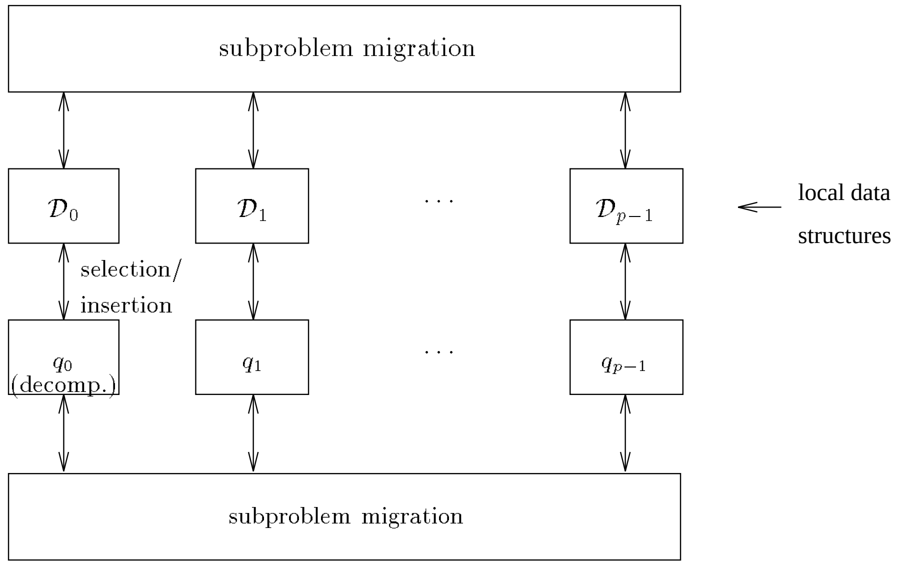

Figure 2: Overall description of the DDM with p processors.

Shared Data-Object Model The SDOM is a high-level approach, where the structure of the search tree is not taken into account, avoiding the troubles inherent to irregularity (figure 3). The global data structure $\mathcal{D}$ is consistently shared by all processors, e.g., whenever any processor invokes an operation on the global data structure, any other processor that subsequently operates the global data structure will retrieve the result of the operation just performed. Its implementation can be *centralized* or *distributed* (see section 6). There is a latency time associated to every such operation and mutual exclusive accesses are provided implicitly. The processors communicate by means of applying the operators (selection, evaluation and insertion) over the shared data structure. The operations are executed indivisibly, e.g., the model guarantees serializability of operation invocations; if two operations are applied simultaneously, then the result is as if one of them is executed before the other; the order of invocation, however, is non-deterministic [9, 10]. The SDOM guarantees an upper bound on the number of iterations (section 5), and allows implicit qualitative and quantitative workload sharing [9].

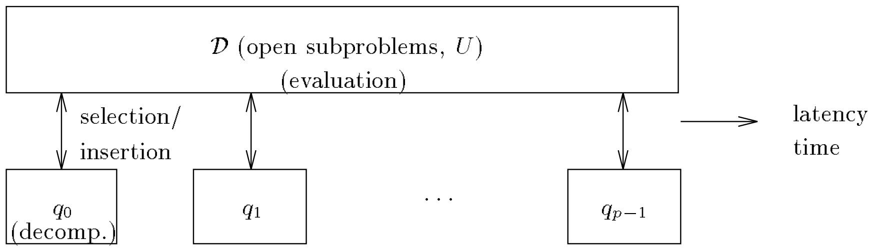

Figure 3: Overall description of the SDOM with p processors.

3.2 Synchronous model under SDOM

For the sake of convenience and without loss of generality, we assume that the synchronous model applies uniquely under the SDOM. In the synchronous formulation, the p processors work iteratively and synchronously, where each iteration consists of five synchronous steps:

S1. each processor selects (non-empty selections, unless there are not enough open subproblems);

S2. each processor decomposes its selected subproblem independently from the others (eventually generating feasible solutions);

S3. the processors that generated feasible solutions in step S2 update the new upper bound together (some subproblems may be eliminated);

S4. each processor inserts all subproblems generated in step S2 and not eliminated in step S3.

We note that this synchronous model determines that *a selection is not performed before all the insertions of the previous iteration have finished* (see figure 4). Such a behavior implies the wasted times, indicated in figure 4, determined by the waiting states for insertion and selection, and for synchronization (idle states). Therefore, synchronous B&B severely restricts the set of possible optimizations of operations over $\mathcal{D}$. In general, this yields high latency times. On the other hand, the open subproblems are *totally ordered*, i.e., the several subproblems selected at each iteration are guaranteed to be the current open subproblems with smallest priority function values h. This fact ensures the best workload sharing, in terms of the heuristic function values, at each iteration.

Synchronous implementations are only useful in special cases or with a few processors and lead generally to communication bottlenecks. Once an operation request has been issued to $\mathcal{D}$, the local algorithms remain idle due to the latency time associated to $\mathcal{D}$. This is a bottleneck of the model that can only be avoided if the computation time for one decomposition operation is much longer than the communication time to transfer one subproblem from one processor to another.

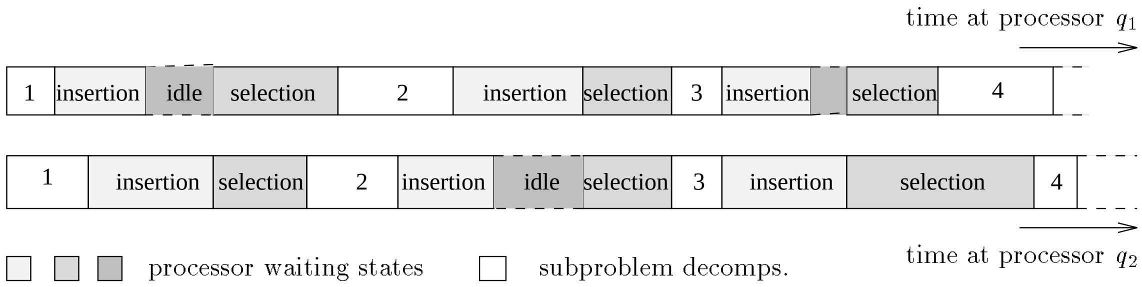

Figure 4: An example of a synchronous execution with 2 processors.

3.3 Asynchronous models

Contrary to synchronous B&B, the main characteristic of an asynchronous algorithm, under SDOM or DDM, is that the local algorithms do not have to wait at predetermined points for predetermined data to become available. Each of the p processors works in its own pace, only limited by the time needed to operate the data structure, carrying out their sequences of operations independently from the others. The objective is a better use of the computational resources than in the synchronous version. Its application under the DDM is immediate.

Under the SDOM, the asynchronous model allows some processors to operate over $\mathcal{D}$ more frequently than others, and we allow the latency time to be substantial and unpredictable. This avoids synchronization restrictions and represents executions allowing one or more selections from a given processor q_i after another processor q_j has selected but before it has accessed $\mathcal{D}$ in order to perform the corresponding evaluation or a corresponding subproblem insertion.

Under both SDOM and DDM, asynchronism may yield the following main advantages. A reduction of the synchronization penalty, overlap between communication and computation, and a potential speedup over synchronous algorithms in most of applications. Further, it may provide a smaller number of accesses to $\mathcal{D}$. On the other hand, it may accomplish more unnecessary subproblem decompositions, and the detection of termination tends to be somewhat more difficult to implement.

3.3.1 Shared Data-Object Model

Although each processor works in a sequence of iterations, there is no synchronization among processors. In this case, an iteration is characterized by:

SA1. each processor selects (at least one non-empty selection);

SA2. each processor decomposes its selected subproblem independently from the others;

SA3. sequential execution on non-inserted open subproblems until some condition holds (e.g., there are no non-inserted open subproblems);

SA4. each processor having found a feasible solution updates the upper bound;

SA5. each processor inserts some of its best open subproblems (possibly none).

In spite of the asynchronism, the model guarantees that the open subproblems are *partially ordered*. By *partially ordered* we mean that whenever a subproblem v is selected, there is no other subproblem with smaller priority than v that was inserted and not yet selected. In other words, if there exist subproblems with smaller priority than v already generated until the moment of v's selection, then they will be either inserted in the future or not inserted at all. This fact can be seen as an advantage over other asynchronous approaches (that do not keep any order on the subproblems) in terms of the quality of the subproblems selected [26, 27, 37]. Therefore, the global data structure is used to avoid irregularities related to the distribution of the search tree, which is not known beforehand, over the processors.

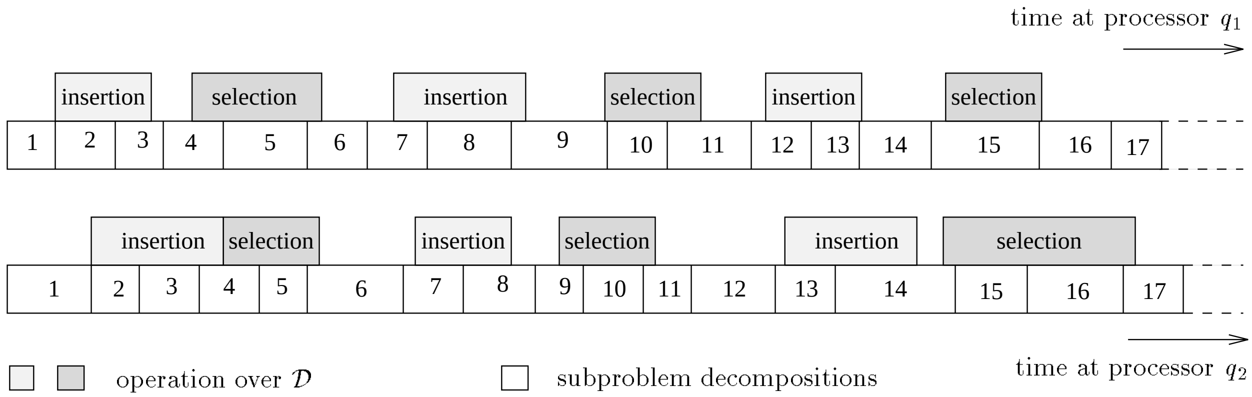

Figure 5: An example of an asynchronous execution with 2 processors.

The example in figure 5 illustrates the capacity of the asynchronous model to avoid waiting states. In this case, in contrast with the synchronous example in figure 4, all the processors remain working on subproblem decompositions while operating on $\mathcal{D}$. This supposes that each processor keeps a number of local subproblems not inserted in $\mathcal{D}$. This asynchronous behavior stems from the inherent non-determinism of the operations of a B&B algorithm, i.e., different time durations for operations in $\mathcal{D}$ as well as different time durations for decomposition and computation of lower bounds.

The notion of *distance* introduced below gives a comparative parameter between synchronous and asynchronous executions. In order to establish such a comparative parameter, we note that the existence of *local opensets* gives a distributed and more flexible nature to an asynchronous execution compared to a synchronous one. For the sake of simplicity, we use the notation *iteration* in all cases, corresponding to a *synchronous iteration* in case of synchronous B&B, or an *asynchronous iteration* in case of asynchronous B&B.

Def. 6 (Local openset) *At an iteration k, the* local openset *in a processor q_i, $V_i(k)$, is the set of subproblems whose last selected ancestors were selected by the processor q_i and which will be inserted in future iterations.*

On the other hand, an asynchronous execution risks search overhead more than a synchronous execution does (details about search overhead in next section on *speedup anomalies*). This fact suggests a measure that indicates the "degree of asynchronism" in an asynchronous execution.

Def. 7 (Distance) *Let k' be the iteration in which the last selected ancestor of a $v \in V_i(k)$ was selected by the processor q_i. The* distance *of the processor q_i at the iteration $k \geq k'$ is given by $d_i(k) = \max_{V_i(k)}\{k - k'\}$, and $d(k) = \{d_0(k), d_1(k), \cdots, d_{p-1}(k)\}$.*

We recall that a selection may be empty. In order to guarantee that all processors work, we assume the following, as it was done in [5] in the context of systems of equations.

Assumption 1 (Partial asynchronism) *There exists a positive integer D such that every processor selects at least once in every interval of $D + 1$ iterations, and a positive integer $\Delta = \max_{i,k}\{d_i(k), D\}$, named* asynchronism measure.

3.3.2 Distributed Data Model

A theoretical (asynchronous) model, as defined under the SDOM, is more difficult to establish under the DDM. The major difficulty is the lack of a global reference, as the global data structure of the SDOM. Actually, it is possible to establish such a model in some special cases, although it is not very useful because its parameters would be difficult to define meaningfully or to measure in practice. We adopt an informal definition instead, as follows. Initially, one local data structure contains the problem S, and the others are empty. Then, each processor iterates through a sequence of three steps:

DA1. sequential execution on its local data structure until some condition holds (e.g., local data structure is empty);

DA2. each processor having found a feasible solution x better than the current best solution broadcasts x, as well as its value;

DA3. subproblem migration for workload sharing purposes.

A processor selects open subproblems from and inserts the generated subproblems into its local data structure (step DA1), and uses a heuristic to send/receive some of its open subproblems to/from other processors (step DA3). This heuristic is designed such that the subproblem migration provides or restores a workload sharing. These subproblem migrations characterizes a parallelism at the search level. Therefore, subproblems can migrate among the local data structures, characterizing a parallelism at the data structure level.

Table 1 summarizes the positive features of each model.

parallel B&B		
SYNCHRONOUS	**ASYNCHRONOUS**	
SDOM	**SDOM**	**DDM**
		fully distributed
open subproblems are totally ordered, inducing the best workload sharing at each iteration	open subproblems are partially ordered	
	reduction of synchronization penalty	no synchronization penalty
	overlap of communication and computation	overlap of communication and computation
	hidden memory latency times	no memory latency times
possibly less decompositions	possibly less operations on $\mathcal{D}$	

Table 1: Synchronous and asynchronous positive features.

4 Speedup anomalies

Appropriately ordering the search and restricting the region searched are the key ideas behind B&B algorithms. The heuristic nature of a B&B algorithm appears when it must decide which path, among several different paths (represented by the open subproblems), has the higher probability to be a minimum critical path in a selection. Considering a sequential B&B, a direct consequence of definition 4 (page 5) is that the number of iterations is at least as large as the number of worthwhile subproblems. However, several subproblems in the search tree may have the same priority, or the same lower bound since we are considering best-first search. A situation that is not unusual in BB-trees is $l(v) = f^*$ for subproblems that are near ancestors of solution nodes in T. Related to these features, many authors noted, in the middle of the 80's, the existence of *speedup anomalies*, that are due to bad selections when several subproblems have a lower bound that equals f^* [30, 31] (for a historical note, see [51]). Indeed, Fox et al. had already observed, in the late 70's, that, even in the case of a single processor, when several nodes have the same lower bound, then some best-first strategies are non-optimal [16]. We summarize in this section the most important known results concerning the anomalies in synchronous B&B algorithms.

Let *isoergotic parallel algorithms* (or simply *isoergotic algorithms*) define parallel algorithms whose total amount of work to be accomplished (without considering system tasks as memory or input/output management) remains the same no matter what is the number of processors employed. Notice that parallel algorithms involving heuristic choices, notably parallel B&B algorithms as considered is this report, can be non-isoergotic.

Let $I(p, \Delta)$ denote the number of iterations carried out with p processors and asynchronism measure Δ. For synchronous parallel B&B, we denote $I(p) = I(p, 0)$. The *speedup* with respect to the number of iterations accomplished is defined as $A(p) = I(1)/I(p)$ (in the general case, $A(p, \Delta) = I(1)/I(p, \Delta)$). Generally speaking, the speedup $A(p)$ is uniquely determined by the difficulty of the problem (size and shape of T) since the execution is completely defined by the synchronous model. In practice, when several processors are used, situations where $1 < A(p) < p$ are usually unavoidable, mainly when the number of processors is large with respect to the difficulty of the problem. The *speedup anomalies* are of two types: *acceleration anomaly*, where $A(p) > p$; and *detrimental anomaly*, where $A(p) < 1$ [30, 31, 34]. Obviously, it is desirable to preserve acceleration anomalies and avoid detrimental anomalies.

4.1 Conditions for preventing detrimental anomalies

Several authors have analyzed detrimental anomalies in the general case [30, 31]. Lai and Sahni have shown that if $l(v) \neq f^*$ whenever v is not a solution node, then synchronous parallel B&B with $p > 1$ does not increase the number of iterations accomplished in sequential [30]. The reason is that, in these particular cases, $I(p)$ is bounded by the number of worthwhile subproblems. However, in the general case, distinct heuristic values must be defined for the subproblems to resolve the ambiguity among subproblems such that $l(v) < f^*$, i.e., a consistent selection strategy must be used. We assume that, like in the case studied originally by Li and Wah [34], the priority function h satisfies the following assumption, where a consistent selection order is used to avoid detrimental anomalies.

Assumption 2 *The heuristic priority function is such that:*

i. $h(v) \neq h(v')$ *if* $v \neq v'$, $v, v' \in V$; *and*

ii. $h(v) \leq h(v')$, *if* v' *is a descendant of* v.

An unambiguous heuristic as in assumption ii is a sufficient condition because it guarantees that at least one subproblem that would be decomposed in sequential with the same heuristic is decomposed in each iteration of the synchronous execution, and that once all the subproblems that would be decomposed in sequential with the same heuristic are decomposed or eliminated, the synchronous execution terminates. The proofs for the synchronous case can be found in [34]. In the next section, a similar result is presented for the asynchronous case.

The existence of anomalies when increasing the number of processors used in parallel was also studied [30, 31]. The conditions above are no longer sufficient when the comparisons consider $p > 1$ as the reference case. Lai and Sprague showed that $\{ \; l(v) \neq f^* \text{ whenever } v$ is not a solution node $\}$ is a sufficient condition to prevent detrimental anomalies when the number of processors is doubled [31]. They mainly studied the importance of the average width of T, and concluded that if T is sufficiently wide, then no detrimental anomalies occur.

4.2 Conditions for preserving acceleration anomalies

Acceleration anomalies are a consequence of the fact that synchronous B&B can be more efficient in its heuristic selection because it decomposes simultaneously several subproblems at each iteration. In [30], it is demonstrated that, given a problem, an instance of its BB-tree can be found such that an acceleration anomaly occurs for a certain p. On the other hand, it is also demonstrated in [30] that, if one assumes that $l(v) \neq f^*$ whenever v is not an optimal solution, then acceleration anomalies do not occur. However, one can always find at least one instance such that $A(p_1)/A(p_2) > p_2/p_1$, $1 < p_1 < p_2$, meaning that when $p_1 > 1$ processors are used, an addition of processors can *increase* the speedup more than linearly, but cannot exceed p_2.

Another necessary condition for acceleration anomalies were stated in [34]. Such condition is based on the *complete consistency* of the heuristic function, in the sense that h should not be *completely consistent* with l. The heuristic priority function h is *completely consistent* with l if there exist two subproblems v_i and v_j such that $h(v_i) > h(v_j)$ and $l(v_i) \leq l(v_j)$. Note that if there are nodes with equal lower bounds and assumption 2 is verified, then h is not completely consistent with the lower bound function.

4.3 Remarks

In practical terms, additional processors may enable us to improve the upper bound quickly, thereby curtailing the expansion of some of the subproblems that might get expanded without these processors. The state space tree T often shows exponential growth. The critical subtree of T occasionally, especially in worst case situations, also does. It is implicitly assumed that if the problem in question is "sufficiently large", the number of worthwhile open subproblems at any time is large enough to keep all processors busy working with subproblems which have to be decomposed [7]. However, the exact number of iterations required depends upon the shape of the search tree. Previous results on speedup anomalies showed unsurprisingly that dense search trees with short critical paths yield to parallel attack more readily than sparse trees with long critical paths [31].

Theoretical analysis and experiments have shown that anomalies are rare when optimal solutions are sought using best-first searches and the problem is sufficiently large. Simulations with the knapsack and traveling salesperson problems are shown in [30]. Of the 105 randomly generated instances run, only six exhibited anomalous behavior, all in the knapsack problem. There was only one case of detrimental anomaly. Other simulations with the knapsack problem, where the instances randomly generated were hard instances, produced no anomalies [11]. Similar results exist in the literature for the traveling salesperson problem [50, 51, 52], for the vertex-cover problem [58], for knapsack, integer programming and vertex-cover problem [59], and for the quadratic assignment problem [35, 47, 54].

These results must be analyzed with some caution. If the problem possesses a good bound computation scheme producing tight bounds, then parallelizing the subproblem investigation might produce an algorithm which scales poorly and may make meaningful use of only a few processors [7]. It is important to keep in mind that they concern "academic problems", with randomly generated instances. If we think of "industrial strength" problems , with real-world instances, pathological cases exist such that anomalies can be frequent (for an illustration, see [12, 13]).

5 Efficiency: counting iterations and time

Consider, initially, the synchronous case. Let us define *efficiency* as $e(p) = A(p)/p$. The speedup as defined in section 4 and the efficiency just defined are considered as simple and expressive ways to measure the overall performance of isoergotic parallel algorithms. They appeal to those interested in theoretical aspects of parallel algorithm design and performance. In isoergotic algorithms, the total quantity of work accomplished remains the same in the two cases (sequential and parallel), and $e(p)$ is a good measure of efficiency in this case, since the total amount of work can be computed as $p.e(p)$ [30, 34, 51, 57].

The notion of computational parallel efficiency is always related to the idea of *velocity* of a parallel algorithm, often as a comparative parameter with respect to a sequential counterpart (see section 2). However, it seems generally impossible to *predict* what search ordering will lead to a maximum amount of subproblem elimination due to evaluations. Therefore, it seems impossible to determine beforehand the minimum amount of work to be accomplished. This phenomenon is also one of the explanations for speedup anomalies (see section 4). Gustafson cites the parallel B&B as a counterexample to the affirmation that "the best sequential algorithm defines the least work necessary", which he considers as a myth of performance measurement [21]. The definition of $e(p)$ considers this myth as a fact to measure the computational parallel efficiency of parallel B&B algorithms.

In the following, we shall define parallel efficiency measures which take into account the two factors involved in the efficiency of parallel synchronous B&B algorithms (or, in general, of non-isoergotic parallel algorithms): the *parallel work of the processors* and the *quality of the heuristic selection function* [11]. The following two sections are devoted to the asynchronous case.

5.1 Measuring effectiveness

In this subsection, we revise the definitions presented in [11]. The slight difference we introduce here concerns the definition of "useful work" below. We first recall the measure comparing parallel and sequential versions. From now on, let Φ^i represent the set of subproblems decomposed during an execution using i processors.

Def. 8 (Adjusted utilization) $e_u(p) = \frac{A(p)}{p} \frac{|\Phi^p|}{|\Phi^1|}$

This measure consists of a representation of the time in which each processor *does* work, and detects overhead due to parallel processing management. If the processors work most of the time during the execution of an algorithm, its adjusted utilization goes to one, even if it eventually does a lot of useless decompositions due to bad selections.

In order to define a second fraction of the efficiency, we need to precisely define the concept of "useful work". Given that a worthwhile subproblem can house an optimal solution, it is reasonable to say that the useful work is exclusively composed of the decompositions of worthwhile subproblems and the ones leading the algorithm to the solution [16, 40]. In other words, let C_m be the minimum over all of the minimum critical paths of each of the trees of $T - T^*$ (the resulting forest when we delete T^* from T). In the case of an iterative algorithm for tree searching, the useful work is defined as $I^* = |T^*| + |C_m|$, where $|T^*|$ and $|C_m|$ indicate the number of vertices in each case.

Def. 9 (Selection efficiency) $e_s(p) = \frac{I^*}{pI(p)}$

Finally, the *effectiveness* is measured as the rate of processor utilization for doing useful work [11].

Def. 10 (Effectiveness) $\epsilon(p) = e_u(p).e_s(p)$

The effectiveness is a simple and general framework for the analysis of parallel B&B algorithms. There are two factors on the *effectiveness* that behave symmetrically. From the traditional point of view, the processor utilization tends to decrease when the number of processors increases. On the other hand, as a heuristic method, the B&B is affected by the choice of the search order. Therefore, a parallel execution can be different from the sequential one. This is the reason why the parallel B&B affects the efficiency of the heuristic, which tends to increase when the number of processors increases.

5.2 Speedup bounds

Some works in speedup bounds tried to count the number of iterations in synchronous B&B. Li and Wah, in their work in speedup anomalies [34], also calculated the lower bound on the speedup given by $A(p) \geq 1$, under assumption 2 (page 14). In [40], lower and upper bounds on $A(p)$ are presented, also in the synchronous case. In that work, there is a characterization of T in different regions, depending on l and f^*. Thus, bounds on the number of iterations are calculated in sequential, as well as in each region of T. The bounds in parallel are functions of these parameters.

Now, let us consider asynchronous B&B, of which we are interested in counting the number of iterations of parallel B&B. In this context, the analysis of anomalies and speedup bounds requires a causality among some special subproblems (namely *basic subproblems* [9] or *basic nodes* [34]), which is well represented by the iterations in the SDOM. On the other hand, under DDM, the concept of asynchronous iteration is not precisely defined, making such analysis difficult. We concentrate, then, on the SDOM. The results also apply to synchronous executions, which correspond to asynchronous ones with $\Delta = 0$.

As cited in a previous section, the main goal of asynchronous implementations is to avoid synchronization delays between iterations. Such delays transform the total order of the synchronous execution in a partial order (see subsections 3.2, page 8, and 3.3, 9), and may have negative effects over the adjusted utilization or selection efficiency measures. Intuitively, this partial order may induce bad selections, the asynchronism may vary the number of iterations accomplished, and may also vary the average iteration time with respect to its synchronous counterpart.

An upper bound on the speedup is presented in [53]. Let *level* of a subproblem v in T define the number of edges of the shortest path in T from its root to v. In that work, assuming the same time for decomposition of subproblems at a same level of T, they also calculate time bounds. They compare sequential and parallel versions where the ambiguity of subproblems with same lower bound is resolved by choosing one of the deepest subproblems in T. In this case, they state tight bounds by calculating the number of subproblems to be decomposed by level of T.

It is shown in [9] that there is a lower bound for $A(p, \Delta)$ that depends on the asynchronism measure Δ and on the number of processors employed, under the assumptions 1 (page 11) and 2 (page 14). These results generalize the ones in synchronous B&B in [34] and shows that, as for the synchronous case, those are sufficient conditions to bound the total number of iterations. It is also shown that if an acceleration anomaly occurs in the synchronous execution, then the lower bound on $A(p, \Delta)$ indicates that an acceleration anomaly can also occur in the asynchronous execution. Furthermore, the detrimental anomaly is bounded. With respect to the selection efficiency, we should search for asynchronous implementations with low asynchronism measure Δ in order to guarantee a large lower bound for $A(p, \Delta)$.

5.3 Counting time

Considering asynchronous executions, the time of execution of each iteration is unpredictable. Thus, we extend the effectiveness defined in the previous section to consider the time. We treat separately the adjusted utilization and the selection efficiency. Firstly, concerning the adjusted utilization, it is sufficient to substitute the "number of iterations" by "execution time" in the definition of speedup. Secondly, we reconsider the concept of "useful work" in order to redefine the selection efficiency.

Let τ^* be the execution time of the useful work. It is composed of two terms, respectively, the time executing decompositions of worthwhile subproblems and that for executing decompositions of subproblems whose lower bound equals f^*. Let $\mathcal{C}$ be the set of critical paths of the trees of $T - T^*$, C be a critical path of $\mathcal{C}$ and $\tau(\mathcal{C})$ ($\tau(T^*)$) the execution time of the subproblems in $\mathcal{C}$ (T^*). Note that this definition considers that the accesses to the data structure are not taken as useful work. Then,

$$\tau^* = \tau(T^*) + \min_{C \in \mathcal{C}} \tau(C).$$

Therefore, for measuring effectiveness in terms of time, it suffices to let τ^* replace I^* and $I(p)$ be replaced by the execution time with p processors in the definition of speedup, adjusted utilization and selection efficiency.

6 Overview of implementations

As a summary of the theoretical aspects approached in the previous sections, parallel B&B algorithms differ from other applications in science and engineering in the following points:

1. data sets are irregular and not constant in size;
2. the workload generated is not uniform;
3. parallel B&B is non-isoergotic.

These features provide a challenge for distributed memory system implementations of high level parallel B&B, which have been investigated from several points of view. A number of parallel implementations have been reported in the literature. Their origins lay in the simulations already cited in previous sections and in parallel implementations of related techniques, as divide-and-conquer [22] and backtracking [15]. These implementations involve the classical communication versus computation tradeoff. Briefly speaking, the amount of decomposed subproblems can be reduced at the cost of increased communication, and vice-versa.

The vast majority of published implementations have focused on specialized algorithms for "pure" combinatorial problems. These are taken as example because they can be implemented and tested in order to capture the essence of parallel B&B algorithms without too many distracting details specific to the problems. We give below a list (non exhaustive) of such applications and their references. More details about the distributed memory implementations can be found in the following subsections, relating each implementation to its corresponding theoretic model.

Problem	reference
Integer linear programming	shared memory [6], distributed memory [1]
Knapsack	distributed memory [9]
Quadratic assignment (QAP)	shared memory [54], distributed memory [32, 47]
Scheduling	distributed memory [39]
Traveling salesperson (TSP)	distributed memory [29, 41, 52, 55]
Vertex cover (VCP)	distributed memory [29, 38]

Also, some "industrial strength" implementations, bearing some resemblance to commercial packages, begin to emerge:

Problem	reference
Mixed integer programming	distributed memory [12, 13]

In the following, a set of distributed implementation is presented which stress the most important conclusions obtained empirically about parallel B&B. These implementations are classified according to the parallel global data organization model used (section 3).

6.1 Shared data-object model

The key points in this model are the implementation of the global data structure $\mathcal{D}$ and the selection operation. The simplest way to implement $\mathcal{D}$ assigns it to a given central processor, and is called *centralized strategy* [27]. However, this strategy limits performance since the latency time at each iteration grows linearly with the number of processors. Moreover, at least two communications are necessary between each processor and the central processor per decomposition (corresponding to the selection and insertions). To remedy these inconvenients, the *distributed strategy* tries to use the set of processors to implement $\mathcal{D}$ and the selection operation.

Centralized strategy

T. Abdelrahman and T. Mudge: their paper describes an implementation of a synchronous parallel B&B that utilizes the 0-1 integer linear programming problem as application. The sequential B&B algorithm used is such that, in parallel, the ratio of the communication overhead to the computation time for a subproblem decomposition is high. For this reason, good speedups were obtained only with a few processors. No analyses concerning the amount of useful work are provided [1].

J. Eckstein: this CM-5-oriented asynchronous implementation for the mixed integer programming problem used the centralized strategy to assess the seriousness of the resulting bottlenecks. During the execution of the algorithm, a number of additional informations are associated to each subproblem in order to improve the performance of the decomposition operation. However, he used the *active message* facility of the CM-5 to prevent useless migration of subproblems and excessive amount of memory consumed at the processor housing $\mathcal{D}$. Problems from MIPLIB, a public collection of problems derived from real industrial applications, were used to check the efficiency of the algorithm in a "industrial strength" environment. It was used up to 128 processors. The benefits of parallelism tend to persist at least until there are about 100 subproblems for every processor, which is related to the coarse grained feature of the problem. Impressive speedup values were obtained, mainly for a problem whose estimated sequential time was 6.6 days and that was solved in only 34.9 min with 128 processors [12]. The results were extended in [13] with more efficient decompositions and lower bound calculations. With this improvement in the sequential algorithm and a better scalar performance of the version of the CM-5 used, most of the superlinear speedups disappeared [12, 13].

M. Quinn: in order to predict the performance of the synchronous implementation described in his paper to solve the traveling salesperson problem, the author describes a model. This model predicts, and the experimental results confirm, an eventual drop in speedup caused by an increase in communication overhead. The sequential algorithm uses an old decomposition heuristic based on the assignment problem, but that does not use the special structure of the TSP [36]. The 30 tested instances were generated randomly from a uniform distribution [52].

Distributed strategy

R. Corrêa and A. Ferreira: the synchronous and asynchronous implementations described in their paper approach the question of the relation between the order of selections and the possibility of overlap between communication and computation. A distributed algorithm is used in both the implementations to select the p (number of processors) smallest priority subproblems in $\mathcal{D}$. In the synchronous case, the latency time associated to selection and insertion degrades the execution time. However, it has a good order of subproblem decompositions, where the total number of subproblems decomposed decreases almost linearly with the number of processors. This order is then taken as a reference in the sense that it is desirable to obtain an asynchronous execution (without synchronization points) that follows this order. With this purpose, an asynchronous algorithm is proposed that is represented by two waves propagating in the search tree. One wave corresponds to the subproblems in $\mathcal{D}$ and the other wave corresponds to the subproblems generated but not inserted in $\mathcal{D}$. The decompositions of subproblems not inserted in $\mathcal{D}$ tend to move the corresponding wave away from the other wave. The distance between the two waves is controlled such that if it becomes greater the a predefined threshold, then decompositions of subproblems in $\mathcal{D}$ are done, bringing the two waves closer to each other. The application used is the knapsack problem, with randomly generated hard instances. Besides the fine grained nature of the problem, good speedup were obtained up to 32 processors [9].

V. Kumar, K. Ramesh and V. Nageshwara Rao: two SDOM implementations were tested by these authors, being one synchronous and other asynchronous. Both implementations were done in a BBN Butterfly composed of up to 256 processors. The global data structure is physically distributed over the local memories of the processors, and the processors use the facility, provided by the BBN Butterfly computer, to access the local memory of other processor through a packet switched network. The asynchronous version is called *blackboard*, and its principle is similar to the asynchronous implementation above. The problems tested are the traveling salesperson problem and the vertex cover problem, whose instances were randomly generated. In the synchronous version, a concurrent heap is used in order to improve the performance of the global data structure operations. For a sufficiently large instance of the traveling salesperson problem, this synchronous algorithm attains good speedups. The vertex cover problem, as a fine grain problem, revealed inadequate. The asynchronous implementations exhibited good speedups for both application problems [29].

D. Miller and J. Pekny: they also used a BBN Butterfly computer to implement their algorithm (14 processors). It is an asynchronous parallel B&B under SDOM. The application used is the traveling salesperson problem. The sequential B&B algorithm used is sophisticated, and involves the solution of an assignment problem relaxation to compute the lower bound of subproblems and subtour elimination heuristics to decompose a subproblem. The decomposition operation is divided in two steps, the *division* of the subproblem in a number of new subproblems and the *lower bound computation*. The global data structure, in this case, keeps two priority lists of subproblems; a list of subproblems to be divided (open subproblems) and a list of subproblems to compute the lower bound (*solved* subproblems). During a selection, the global data structure provides an open subproblem if there is at least one; otherwise, it provides a solved subproblem. Moreover, a patching algorithm is used from time to time to find a feasible solution in a subproblem, which can update the upper bound. The instances were generated randomly. The speedup values tended to increase with the number of subproblems decomposed. For reduced numbers of decomposed subproblems, the speedup tended to one, and deceleration speedup anomalies were detected. For instances whose the number of subproblems is sufficiently large, the speedups tended to p, and acceleration anomalies were verified [41, 50].

6.2 Distributed data model

The most important question to be addressed in the context of DDM is the dynamic workload sharing. Three choices must be done. The first choice consists of deciding between *quantity* or *quality* workload sharing. In the first case, the amount of work defines the workload of the processors (e.g., the number of open subproblems in a local data structure). In the second case, it is the quality of the work (generally given by the priority of the open subproblems) that defines the workload. The second choice is whether to prevent or to recover workload imbalance. Finally, the workload sharing can be local or global.

Implementations

T. Abdelrahman and T. Mudge: the objective of their DDM implementations is the investigation of the importance of a specialized dynamic workload sharing. The version without this workload scheme exhibited very poor results in terms of speedups. Even the comparison with the SDOM, centralized strategy, is unfavorable. The explanation for that is the difficulty to correctly predict the workload at the beginning of the execution in order to initially determine the workload of each processor for the rest of the execution. Using a dynamic quantitative workload sharing algorithm, which is based on the principle that an idle processor requests work from another processor, good speedups were obtained, for the instances tested, up to 32 processors. These results were all better than the SDOM, centralized strategy, ones [1].

J. Eckstein: in order to use a DDM to extend its previous SDOM, centralized strategy, implementation, this author uses a randomized workload sharing scheme proposed by Karp and Zhang [25]. In this scheme, each processor randomly chooses, from time to time, a processor to send a just generated subproblem. Comparing the results obtained with this scheme with the results with the SDOM, he noticed an increasing in the parallel execution time of about 30 percent in average. Investigating the total amount of work accomplished by the processors in both implementation, he surprisingly noticed that this was typically smaller under the DDM, indicating that a certain amount of randomness can be beneficial. An in-depth investigation of this fact showed that the Karp-Zhang scheme does a fairly good job of balancing workload quality, but is less effective at evening out its quantity, since more processors idleness was detected. Using the improved sequential implementation (see subsection 6.1), this scheme also exhibited a degradation on the parallel execution time (36 percent in average). The amount of work increased less than the parallel execution time (25 percent in average). However, using in addition a specialized workload sharing scheme that organizes some of the processors in a binary tree, the speedup values became competitive with the corresponding values using the centralized strategy. An interesting point to remark is that this scheme used solely is worse than the randomized in terms of speedup (in average). Apparently, randomization removes the "easy" parts of the task of workload sharing, and the specialized workload sharing scheme improves the performance taking care of the difficult aspects [12, 13].

V. Kumar, K. Ramesh and V. Nageshwara Rao: besides the SDOM implementations, these authors also experimented two DDM implementations. In both implementations, specialized workload sharing schemes were used. They are the random strategy of Karp and Zhang, and a ring strategy. In the ring strategy, the processors are assumed to be connected in a virtual ring. Each processor periodically sends the newly generated subproblems to one of its neighbors in the ring. In order to interpret the results obtained, we note that the main difference between the two problems used (TSP and VCP) is that for the VCP, nearly 75 percent of the subproblems have the same lower bound, while these values are more distributed in the TSP. Thus, a quality workload sharing is more important for the TSP than for the VCP. The results confirm this affirmation. For the TSP, both DDM strategies are worse than the SDOM strategies. For the VCP, the SDOM implementation also yields better speedups, but the randomized scheme also gives good results [29].

P. Laursen: this author uses the QAP problem to compare asynchronous parallel B&B implementations under DDM, with and without specialized dynamic workload sharing. The emphasis in his work was the simplicity. He measures the average processor utilization rate to determine efficiency. The QAP being a very hard optimization problem, the sequential algorithm tends to decompose a large number of subproblems. In the first approach, the original problem is initially decomposed in an appropriate number and subproblems. Then, the amount of work associated to each subproblem is estimated, and the subproblems are distributed according to these estimations such the total amount of work is evenly distributed over the processors. In the second approach, subproblems are exchanged between neighbor processors at fixed periods of time for dynamic workload sharing purposes. Good results were obtained in both cases. Even the version without dynamic workload sharing exhibited average processor utilization rates between 80 and 90 percent, indicating that the tree search is quite "regular" in this case. The quantitative workload sharing used is simple, and bring the average processor utilization to 98 percent [32].

R. Lüling and B. Monien: the VCP was also used by these authors as an application in order to investigate DDM implementations. The emphasis of their proposal is a quality workload sharing specialized scheme. They proposed some *weight functions* to express the workload of a processor. The workload is locally controlled in such a way that a processor and its neighbors have nearly the same workload. A feedback scheme is used to dynamically update the admissible difference of workload between processors, to avoid trashing effects and to control the amount of communication. Using a variation of the VCP, known as *weighted VCP*, the instances were randomly generated with weights in a large range. Two communication topologies were used, namely de Bruijn and ring. For local workload sharing strategy, low diameter networks are well adapted, thus the implementations in the de Bruijn network presented better results. Impressive good speedups were obtained to sufficiently large instances, even up to 256 processors [38].

R. Ma, F. Tsung and M. Ma: they studied two implementations of an asynchronous parallel B&B under DDM. In the first implementation, there is no specialized workload sharing scheme. In this case, subproblems are allocated to the processors, and the processors work independently. On the other hand, in the second implementation, when a processor runs out of work, it requests work from another busy processor. The busy processor estimates its remaining work in order to decide whether or not to transfer some work to the idle processor, which characterizes a quantitative workload sharing scheme. Good speedup values were attained with dynamic workload sharing and a resource scheduling problem as target application (no details about the instance chosen are provided). The workload imbalance caused bad speedup values in the first implementation [39].

P. Pardalos and J. Crouse: these authors used a sophisticated sequential B&B algorithm to find all optimal solutions of a quadratic assignment problem, which prevents the possibility of speedup anomalies (see section 4). The asynchronous parallel B&B algorithm is based on this sequential one, under DDM. A sufficiently large number of subproblems is initially generated and evenly distributed over the processors. The speedup is measured for two classes of test problems used to evaluate the algorithm. Good speedups were obtained only for sufficiently large instances [47].

6.3 General remarks

In the light of the large empirical investigation on parallel B&B algorithm found in the literature, it is clear that parallelism is useful to improve its search performance. However, an universal parallelization strategy seems not to exist. The algorithmic solutions for the major problems posed by the parallelism when one tries to parallelize a sequential B&B algorithm for a given problem seems to be problem-dependent, and not rarely, instance-dependent. The SDOM favors the qualitative distribution of the subproblems over the processors, but tends to incur more overheads in data structure operations. It is not indicated when qualitative workload sharing is not crucial. However, even when DDM is more indicated, specialized workload sharing is often necessary, but its nature depends on the shape of the searched tree. When lower bounds varies in a large range, quality workload sharing is more important, to which some randomness appears to be useful. Otherwise, quantity workload sharing appears to be sufficient. Therefore, in the context of "industrial strength" problems, the instances tend to present several different search trees, and quality workload sharing seems to be necessary as a general strategy.

Another crucial question that emerged was relative to the impact of the sequential algorithm in the performance of its parallel counterpart. It appears that increasing the performance of the sequential algorithm tends to decrease the performance of its parallel algorithm. In order to obtain good speedups, the number of decomposed subproblems in sequential must be large enough. Deceleration anomalies tend to be more frequent in small problems. With large problems, on the other hand, acceleration anomalies tend to be more frequent. Finally, the relation between the parallel performance and machine characteristics, such as communication and computation power, is not yet established in a clear and systematic manner. The question of completely model parallel B&B, with performance predictions, remains open. Stochastic models exist, but they do not capture the particularities of real-world instances.

7 Challenges for further research

The great challenge for parallel branch-and-bound researchers is the design of a parallel algorithm that reduces considerably the time needed in sequential to obtain an optimal solution for discrete optimization problems. Important foundations in this direction have already been laid, since several parallel implementations exist that attained good performance results in "academical" experiments. Hence, the next barrier to be tackled relates to real-world problems. Then, a model that incorporates the technology and performance of parallel systems and the state-of-the-art of sequential codes (eventually including advanced techniques, as cutting planes and other polyhedral methods) should be established. This model must present an easy adaptation to new technologies and sequential codes to permit analyses of the relations between the technology and problem characteristics. Clearly, such methodology must be general enough to incorporate the most recent advances in technologies and sequential codes. Finally, convivial environments to users are required in order to easily incorporate existing sequential codes and disseminate the parallelism within the operations research community.

References

[1] T. Abdelrahman and T. Mudge. Parallel branch and bound algorithms on hypercube multiprocessors. In *1988 ACM Conf. on Lisp and Funct. Prog.*, pages 1492–1499. ACM Press, 1988.

[2] G. Ananth, V. Kumar, and P. Pardalos. Parallel processing of discrete optimization problems. *Encyclopedia of Microcomputers*, 13:129–147, 1993.

[3] S. Arvindam, V. Kumar, and V. N. Rao. Efficient parallel algorithms for search problems: Applications in VLSI CAD. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October 1990.

[4] A. Barr and E. Feigenbaum. *The Handbook of Artificial Intelligence*, volume I. William Kaufmann, Inc., 1981.

[5] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall International Editions, 1989.

[6] R. Boehning, R. Butler, and B. Gillett. A parallel integer LP algorithm. *European Journal of Operations Reasearch*, 34:393–398, 1988.

[7] J. Clausen. Do inherently sequential branch-and-bound algorithms exist? *Parallel Processing Letters*, 1994. To appear.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, McGraw-Hill, 1990.

[9] R. Corrêa and A. Ferreira. A distributed implementation of asynchronous parallel branch-and-bound. In *IRREGULAR '94*, Aug. 1994. To appear.

[10] R. Corrêa and A. Ferreira. Modeling parallel branch-and-bound for asynchronous implementations. In *'94 DIMACS Workshop on Parallel Processing*. DIMACS Center, Rutgers University, Apr. 1994. To appear.

[11] R. Corrêa and A. Ferreira. On the effectiveness of synchronous branch-and-bound algorithms. 1994. Submitted for publication.

[12] J. Eckstein. Control strategies for parallel mixed integer branch and bound. In I. C. S. Press, editor, *Supercomputing '94*, 1994. To appear.

[13] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. Technical Report TMC-257, Thinking Machines Corporation, 1994. To appear in the *SIAM Journal on Optimization*, 1994.

[14] A. Ferreira and J. Rolim, editors. *Solving Irregular Problems in Parallel: State of the Art*. Kluwer Academic Publisher, Boston (USA), 1995.

[15] R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, Apr. 1987.

[16] B. Fox, J. Lenstra, A. Rinnooy Kan, and L. Schrage. Branching from the largest upper bound: folklore and facts. *Europ. Journal of Op. Res.*, pages 191–194, 1978.

[17] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. F. Freeman, 1979.

[18] M. Gengler. Axiomatique de branch and bound. Rapport interne 77, LITH - Ecole Polytechnique Fédérale de Lausanne, 1992.

[19] M. Gengler and G. Coray. A parallel best-first B&B with synchronization phases. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Parallel Processing: CONPAR 92 - VAPP V*, LNCS 634, pages 515–526. Springer-Verlag, Sept. 1992.

[20] S. Gnesi, U. Montanari, and A. Martelli. Dynamic programming as graph searching: An algebraic approach. *Journal of the ACM*, 28(4):737–751, Oct. 81.

[21] J. Gustafson. The consequences of fixed time performance measurement. In *25th Hawaii Int. Conf. on Syst. Sc.*, volume III. IEEE, Jan. 1992.

[22] E. Horowitz and A. Zorat. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers*, c-32(6):582–585, June 1983.

[23] T. Ibaraki. Theoretical comparisons of search strategies in branch-and-bound algorithms. *Int. Journal of Computing and Information Sc.*, 5(4):315–344, 1976.

[24] T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *Journal of the ACM*, 24(2):264–279, Apr. 1977.

[25] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Symposium on Theory of Computing*, pages 290–300, 1988.

[26] R. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computations. *Journal of the ACM*, 40(3):765–789, July 1993.

[27] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[28] V. Kumar and L. Kanal. The CDP: a unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In *National Conf. on A.I.*, 1983.

[29] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conf. on Artificial Intelligence*, pages 122–127, Aug. 1988.

[30] T. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27:594–602, 1984.

[31] T. Lai and A. Sprague. Performance of parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, C-34(10):962–964, Oct. 1985.

[32] P. S. Laursen. Simple approaches to parallel Branch and Bound. *Paral. Comp.*, 19:143–152, 1993.

[33] W. Levelt, M. Kaashoek, H. Bal, and A. Tanenbaum. A comparision of two paradigms for distributed shared memory. *Software – Practice and Experience*, 22(11):985–1010, Nov. 1992.

[34] G. Li and B. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, C-35(6):568–573, June 1986.

[35] Y. Li and P. Pardalos. Parallel algorithms for the quadratic assignment problem. In P. Pardalos, editor, *Advances in Optimization and Parallel Computing*, pages 177–189. North-Holland, 1992.

[36] J. Little, K. Murty, D. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[37] R. Lüling and B. Monien. Load balancing for distributed branch & bound algorithms, 1992.

[38] R. Lüling and B. Monien. Load balancing for distributed branch & bound algorithms. Technical Report Nr. 114, Universität Gesamthochschule Paderborn, Feb. 1993.

[39] R. Ma, F. Tsung, and M. Ma. A dynamic load balancer for a parallel branch and bound algorithm. In *1988 ACM Conf. on Lisp and Funct. Prog.*, pages 1505–1513. ACM Press, 1988.

[40] B. Mans. *Contribution à l'Algorithmique Non Numérique Parallèle : Parallélisation de Méthodes de Recherche Arborescente.* PhD thesis, Université Paris VI, June 1992.

[41] D. Miller and J. Pekny. Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. *Operations Research Letters*, 8:129–135, June 1989.

[42] L. Mitten. Branch-and-bound methods: General formulation and properties. *Operations Research*, 18:24–34, 1970.

[43] T. Morin and R. Marsten. Branch-and-bound strategies for dynamic programming. *Operations Research*, 24(4):611–627, Aug. 1976.

[44] D. Nau, V. Kumar, and L. Kanal. General branch and bound, and its relation to A* and AO*. *Artificial Intelligence*, (23):29–58, 1984.

[45] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons Interscience, 1988.

[46] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall Inc., Englewood Cliffs, NJ, 1982.

[47] P. Pardalos and J. Crouse. A parallel algorithm for the quadratic assignment problem. In *Proceedings Supercomputing 1989*, pages 351–360. ACM Press, 1989.

[48] P. Pardalos and X. Li. Parallel branch-and-bound algorithms for combinatorial optimization. *Supercomputer*, pages 23–30, Sept. 1990.

[49] J. Pearl. *Heuristics - Intelligent Search Strategies for Computer Problem Solving.* Reading, MA: Addison-Wesley, 1984.

[50] J. Pekny and D. Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. *Mathematical Programming*, 55:17–33, 1992.

[51] E. Pruul, G. Nemhauser, and R. Rushmeier. Branch-and-bound and parallel computation: a historical note. *Op. Res. Letters*, 7(2):65–69, Apr. 1988.

[52] M. Quinn. Analysis and Implementation of Branch and Bound Algorithms on a Hypercube Multicomputer. *IEEE Trans. on Comp.*, 39(3):384–387, March 1990.

[53] M. Quinn and N. Deo. An upper bound for the speedup of parallel best-first branch-and-bound algorithms. *BIT*, (26):35–43, 1986.

[54] C. Roucairol. A parallel branch and bound algorithm for the quadratic assignment problem. *Discrete Applied Mathematics*, 18, 1987.

[55] K. Schawan, J. Gawkowski, and B. Blake. Process and workload migration for a parallel branch-and-bound algorithm on a hypercube multicomputer. In *1988 ACM Conf. on Lisp and Funct. Prog.*, pages 1520–1530. ACM Press, 1988.

[56] D. R. Smith. Random trees and the analysis of branch and bound. *Journal of the ACM*, 31(1):163–188, 1984.

[57] H. Trienekens. *Parallel Branch and Bound Algorithms.* PhD thesis, Erasmus University, Rotterdam, 1990.

[58] B. Wah and Y. Eva. MANIP - A multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, C-33(5):377–390, May 1984.

[59] B. Wah and C. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Transactions on Software Engineering*, SE-11(0):922–934, Sept. 1985.

Symbol	Usage
$A(p)$	speedup with p processors
B&B	branch-and-bound algorithm
D	maximum interval without selection from a given processor
$\mathcal{D}$	global data structure
$I(p)$	number of iterations with p processors
S	domain of the problem
S^*	set of optimal solutions
SDOM	shared data-object model
T	state space tree
T^*	critical subtree of T
V	set of vertex of T
$V_i(k)$	local openset, processor q_i, iteration k
$d(k)$	distance, iteration k
$d_i(k)$	distance, processor q_i, iteration k
f	objective function
h	heuristic priority function
k	iteration
l	lower bound function
p	number of processors
n	dimension of S
v, v'	subproblems
x	element of S
x^*	optimal solution
Δ	asynchronism measure
β	decomposition process
δ	selection function

Table 2: Symbols used in this report.