CHAPTER 1

Parallel Branch-and-Bound Algorithms

TEODOR GABRIEL CRAINIC

Département de management et technologie École des Sciences de la Gestion Université du Québec à Montréal and CIRRELT, Canada

BERTRAND LE CUN and CATHERINE ROUCAIROL

Laboratoire PR*i*SM, Université de Versailles (France)

1.1. INTRODUCTION

In the beginning of the twenty-first century, large unsolved combinatorial optimization problems have been solved exactly.

Two impressive cases should be mentioned. First are two instances of the famous Symmetric Traveling Salesman problem (TSP) with >10,000 cities (respectively, 13,509 and 15,112 cities; instances usa13509, d15112) by Applegate *et al*. (1). Second are instances of the Quadratic Assignment Problem (QAP) up to size 32, Nugent 30 (900 variables) and Krarup 32 (1024 variables) by Anstreicher, Brixius, Goux, and Linderoth (2,3). The results on the instance Nugent 30 have been announced in American (Chicago Tribune, Chicago Sun Times, HPCWire, WNCSA Access Magazine) and French (InfoScience, Le Monde, Transfert) newspapers. This impressive media frenzy reflected the impact of the achievement, which was deemed of the same order of magnitude as the victory of IBMs parallel computer DeepBlue over the chess world champion Kasparov.

Several factors combined to bring on these achievements. A first reason is the scientific progress in Operations Research, in particular regarding the quality of the lower bounds for these problems (cutting plane techniques for the TSP and convex quadratic programming for the QAP). The computation of these new bounds is very time consuming, however. Moreover, the bounds are computed at each node of a tree whose size is huge (several billions of nodes). The progress in processor computing power certainly contributed.

Parallel Combinatorial Optimization, edited by El-Ghazali Talbi

These two reasons would not have been sufficient, however. The utilization of *parallel branch-and-bound* (B&B) strategies on large computer clusters and grids with advanced programming tools, including multithreading and fault tolerance functionalities, is the third factor of success. Indeed, the TSP instance usa13509 required 48 workstations (DECAlpha, Pentium II, Pentium Pro, and Ultrasparc) that explored in parallel a tree of 9539 nodes. The instance Nugent 30 (30 firms to be assigned to 30 sites) needed the exploration of a tree with 11,892,208,412 nodes and a network of 2510 heterogeneous machines with an average number of 700 working machines (Pc, Sun, and SGI Origin2000). These machines were distributed in two national laboratories (Argonne, NCSA), five American universities (Wisconsin, Georgia tech, New Mexico, Colombia, Northwestern), and was connected to the Italian network INFN. The time spent was ~1 week! But, the equivalent sequential time on a HP-C3000, for example, was estimated at 218,823,577 s or 7 years!

The resolution of these problems by using parallel B&B illustrates the interest of this methodology. Not all NP-hard combinatorial optimization problems or problem instances may be equally well addressed, however. An honest analysis of the above results shows that these performances are also due to some characteristics of the problems, particularly the existence of very good upper bounds. Moreover, the tree search strategy used was equivalent to a brute force exploration of the tree. In most cases, bounds are less tight and more advanced parallel tree exploration strategies must be used. The goal of this chapter is to discuss some of these challenges and present some of the parallel B&B strategies that may be used to address them. We will show how parallelism could help to fight against the combinatorial burst and to address efficiently (linear speed up) and accurately (optimal solutions) combinatorial optimization problems of considerable size. A general review of parallel B&B methodology and literature may be found in Gendron and Crainic (4).

The chapter is organized as follows. Section 1.2 briefly recalls the sequential B&B algorithm. Section 1.3 presents the different sources of parallelism that can be exploited in a B&B algorithm. Section 1.4 discusses the performances that can be obtained from a theoretical and experimental point of view. Section 1.5 presents different parallelization strategies, with their respective advantages, issues, and limits. Section 1.6 briefly reviews B&B libraries proposed to help the user to implement the different parallelization strategies and to benefit from advanced programming tools, including multithreading and fault tolerance functionalities. To illustrate these concepts, the application of parallel B&B to the QAP is presented in Section 1.7. Section 1.8 contains concluding remarks.

1.2. SEQUENTIAL B&B

Let us briefly recall the main components of a B&B algorithm. Suppose the following combinatorial optimization problem is to be solved: Given a finite

discrete set X, a function $F: X \to \mathbb{R}$, and a set S, where $S \subseteq X$, and an *optimal* solution $x^* \in S$, such that $f(x^*) = \min\{f(x) | \forall x \in S\}$. The set S is usually a set of constraints and is called feasible domain, all elements $x \in S$ being *feasible* solutions. We assume that S is finite or empty.

A **branch-and-bound** (B&B) solution method consists in implicitly enumerating S by examining a subset of feasible solutions only. The other solutions are eliminated when they cannot lead to a feasible or an optimal solution.

Enumerating the solutions of a problem consists in building a B&B tree whose nodes are subsets of solutions of the considered problem. The size of the tree, that is, the number of generated nodes, is directly linked to the strategy used to build it.

Synthetically, the B&B paradigm could be summed up as follows:

Building the Search Tree

- A *branching scheme* splits X into smaller and smaller subsets, in order to end up with problems we know how to solve; the B&B nodes represent the subsets, while the B&B edges represent the relationship linking a parent-subset to its child-subsets created by branching.
- A *search* or *exploration strategy* selects one node among all pending nodes according to priorities defined *a priori*. The priority of a node or set S_i, $h(S_i)$, is usually based either on the depth of the node in the B&B tree, which leads to a depth-first tree-exploration strategy, or on its presumed capacity to yield good solutions, leading to a best-first strategy.

Pruning Branches

- A *bounding function* υ gives a *lower bound* for the value of the best solution belonging to each node or set S_i created by branching.
- The *exploration interval* restricts the size of the tree to be built: Only nodes whose evaluations belong to this interval are explored, other nodes are eliminated. The smallest value associated to a pending node in a current tree, the upper bound UB, belongs to this interval and may be provided at the outset by a known feasible solution to the problem, or given by a heuristic. The upper bound is constantly updated, every time a new feasible solution is found (value of the best known solution, also called the *incumbent*).
- *Dominance relationships* may be established in certain applications between subsets S_i, which will also lead to discard nondominant nodes.

A Termination Condition

This condition states when the problem is solved and the optimal solution is found. It happens when all subproblems have been either explored or eliminated.

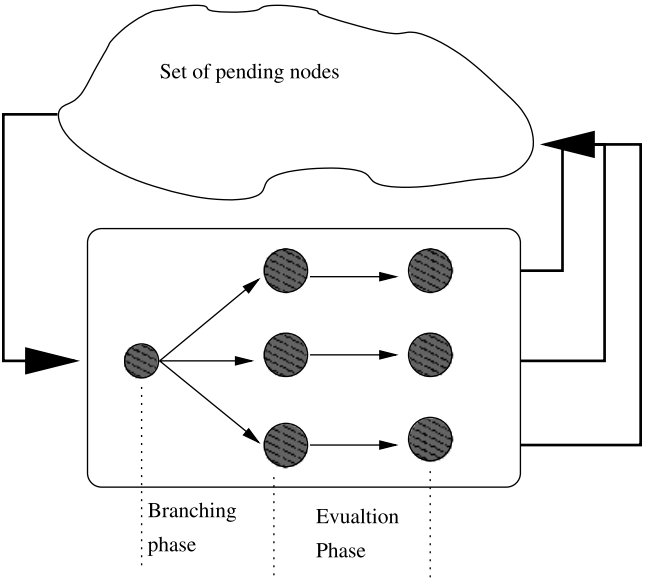


Fig. 1.1. Graphical view of a sequential B&B algorithm.

From an algorithmic point of view, illustrated in Fig. 1.1, a B&B algorithm consists in carrying out a series of basic operations on a pool of nodes (set of nodes of varying or identical priority), usually implemented as a priority queue: *deletemin* (select and delete the highest priority element), *insert* (insert a new element with predefined priority), *deletegreater* (delete elements with higher priority than a given value).

1.3. SOURCES OF PARALLELISM

Two basic, by now classic, approaches are known to accelerate the B&B search:

1. *Node*-based strategies that aim to accelerate a particular operation, mainly at the node level: Computation in parallel of lower or upper bound, evaluation in parallel of sons, and so on.
2. *Tree*-based strategies that aim to build and explore the B&B tree in parallel.

Node-based strategies aim to accelerate the search by executing in parallel a particular operation. These operations are mainly associated to the sub-problem, or node, evaluation, bounding, and branching, and range from "simple" numerical tasks (e.g., matrix inversion), to the decomposition of computing intensive tasks (e.g., the generation of cuts), to parallel mathematical programming (e.g., simplex, Lagrangean relaxation, capacitated multicommodity network flow) and meta-heuristic (e.g., tabu search) methods used to

compute lower bounds and to derive feasible solutions. This class of strategies has also been identified as *low-level* (or *type 1*) parallelization, because they do not aim to modify the search trajectory, neither the dimension of the B&B tree nor its exploration. Speeding it up is the only objective. It is noteworthy, however, that some node-based approaches may modify the search trajectory. Typical examples are the utilization of parallel Lagrangean relaxation or parallel simplex, particularly when multiple optima exist or the basis is transmitted by the nodes generated by the branching operation.

Other strategies, for example, *domain decomposition* (decompose the feasible domain and use B&B to address the problem on each of the components of the partition) and *multisearch* (several different B&B explore the same solution domain in parallel with or without communications), hold great promise for particular problem settings, but have not yet been studied in any particular depth. Note that these strategies are not mutually incompatible. Indeed, when problem instances are particularly hard and large, several strategies may be combined into a comprehensive algorithmic design. Thus, for example, node-based strategies could initiate the search and rapidly generate interesting subproblems, followed by a parallel exploration of the tree. Or, a multisearch approach may be set up, where each B&B search is using one or more parallel strategies. Tree-based strategies have been the object of a broad and most comprehensive research effort. Therefore, in this chapter, the focus is on tree-based strategies.

Tree-based parallelization strategies yield *irregular algorithms* and the corresponding difficulties have been well identified [e.g., Authié *et al*. (5) and the STRATAGEMME project (6)]:

- Tasks are created dynamically in the course of the algorithm.
- The structure of the tree to explore is not known beforehand.
- The dependency graph between tasks is unpredictable: no part of the graph may be estimated at compilation or runtime.
- The assignment of tasks to processors must be done dynamically.
- Algorithmic aspects, such as sharing and balancing the workload or transmitting information between processes, must be taken into account at run time in order to avoid overheads due to load balancing or communication.

Furthermore, parallelism can create tasks that are redundant or unnecessary for the work to be carried out (*research overhead*), or which degrade performances, the so-called *speedup anomalies*. Some of these issues are examined in the next section.

1.4. CRITICAL B&B TREE AND SPEEDUP ANOMALIES

The success of the parallelization of a B&B algorithm may be measured experimentally by the **absolute speedup** obtained with p processors, defined as the

```
Node root; // the root node.
PriorityQueue set; // the priority queue that  stores the
                   // set of pendings nodes
Node incumbent; // the Incubent.

 set.Ins(root);
 while (! set.empty() ) {
    Node n =  set.deletemin();
    foreach ( Node s son of n) { // Branching
        s.Evaluate(); // Evaluation of the node s
        if ( s.IsSolution() && incumbent.Cost()>s.Cost() ){
              incumbent= son;
      pq.Deletegreater(incumbent.Cost());
        }
        else if ( s.Feasible() && incumbent.Cost()>s.Eval() )
              set.Insert(s);
    }
 }
```

Fig. 1.2. Pseudo-code of a sequential B&B algorithm.

ratio of the time taken by the best serial algorithm over that obtained with a parallel algorithm using p processors, for one instance of a problem. For the sake of simplicity, **a relative speed-up** is often used, defined as the ratio of the time taken by a serial algorithm implemented on one processor over the time required by parallelizing the same algorithm and implementing it on p processors. *Efficiency* is a related measure computed as the speedup divided by the number of processors.

For parallel tree-based B&B, one would expect results that show almost linear speedup, close to p (efficiency close to 100%). Yet, the relative speedup obtained by a parallel B&B algorithm may sometimes be quite spectacular, $>p$, while at other times, a total or partial failure (much $<p$) may be observed. These behavioral anomalies, both positive and negative, may seem surprising at first (7). They are mainly due to the combination of the speedup definitions and the properties of the B&B tree where priorities, or the bounding function, may only be recognized *a posteriori*, once the exploration is completed. These issues have been the subject of a great deal of research in the 1980s (we would like to quote Refs 8, 9 and refer to Ref. 6 for a more comprehensive literature review).

In fact, the time taken by a serial B&B is related to the number of nodes in the fully developed tree. The size of the B&B tree—where the branching rule, the bounding function υ, and the node-processing priority h have been defined *a priori* (prior to execution)—depends on the search strategy and the properties of υ.

Four different types of nodes may be defined in a B&B tree (6,7), illustrated in Section 1.3:

1. Critical nodes, set C, representing incomplete solutions with a value strictly smaller than the optimum solution f^*.
2. Undecidable nodes, set M, which are nonterminal, incomplete, solutions with a value equal to f^*.
3. Optimal nodes, set O, with the value f^*.
4. Eliminated nodes, set E, with a value strictly $>f^*$.

As a B&B algorithm is executed according to a *best-first* strategy, it develops all the critical nodes, some undecidable nodes, and one optimal node. Certain nodes belonging to E can be explored when using other strategies. Any strategy will always develop the set of critical nodes, also called the **critical tree**. Several executions may correspond to the same developed B&B tree, according to the choices made by the strategy between nodes of equal priority, which may be very numerous.

The **minimal tree** is defined as the tree which, regardless of the exploration strategy, must be built to prove the optimality of a feasible solution and that has the minimal number of nodes. Notice that the critical tree (critical node) is included in the minimal tree. In parallel processing, p processors must explore all nodes in the minimal tree. In serial processing, speedup is always linear (lower than or equal to p) if the minimal tree has been built. Therefore, the serial time is that of the *best* possible serial algorithm.

The fact that it is not always possible to define *a priori* the search strategy to construct the minimal tree shows that speedups may be favorable (the serial tree is very large) or unfavorable, and therefore proves the existence of these anomalies. It is interesting to note that parallelism may have a corrective effect in serial cases, where the minimal tree has not been built.

In a theoretical synchronous context, where one iteration corresponds to the exploration of p nodes in parallel by p processors between two synchronizations, a suitable condition for avoiding detrimental anomalies is h to be discriminating (9). One necessary condition for the existence of favorable anomalies is that h should not be completely *consistent* with υ (strictly higher priority means a lower or equal value). This is the case of the breadth and depth-first strategies.

The *best-first* strategy constructs the minimal tree if (sufficient condition) there are no undecidable nodes ($M = \emptyset$), and if the bounding function is *discriminating*, which means that υ is such that no nodes have equal priority. The best-first strategy has proved to be very *robust* (10). The speedup it produces varies within a small interval around p. It avoids detrimental anomalies if υ is discriminating, as we have already seen, or if it is consistent (at least one node of the serial tree is explored at each iteration) (11).

Rules for selecting between equal priority nodes have been studied (11), with the aim of avoiding unfavorable anomalies in best-first B&B algorithms, thus eliminating processing overheads or memory occupation, which is not the case with the other proposals (12). Three policies, based on the order in which nodes are created in the tree, have been compared: (*1*) newest (the most

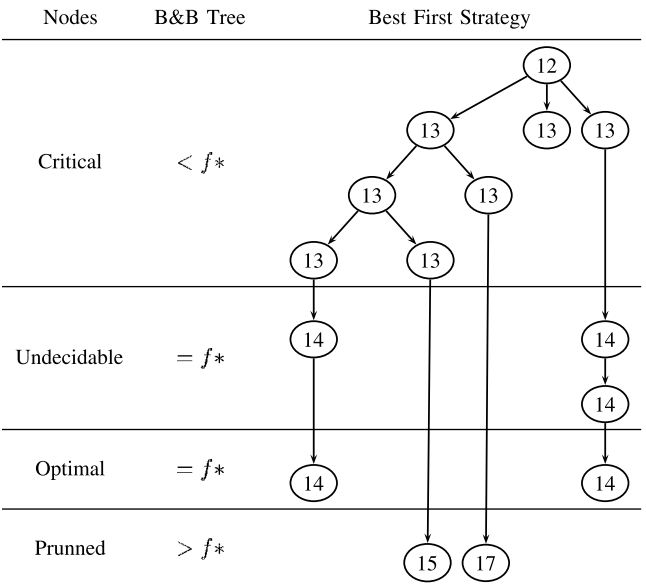


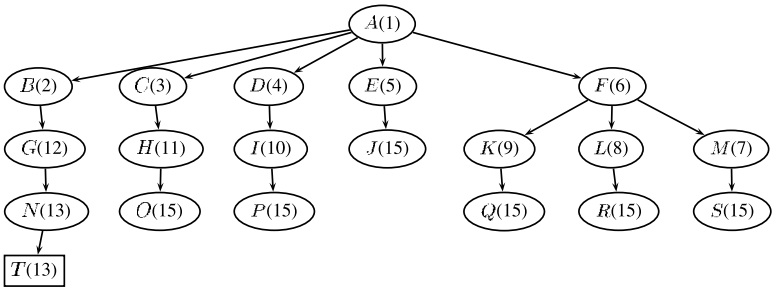
Fig. 1.3. Example of nodes classification of a B&B tree.

recently generated node); (*2*) leftmost (the leftmost node in the search tree); and (*3*) oldest (the least recently generated node).

Bounds calculated on expected speedups show that the "oldest" rule is least likely to produce anomalies. It is therefore interesting to study the data structures which, unlike the well-known heaps, can implement the "oldest" rule. As similar results (condition of existence of anomalies) may be demonstrated when the number of processors is increased (growth anomalies or non-monotonous increases in speed up), some researchers worked on defining a measurement, the isoefficiency function $iso(p)$ (13), based on general results on the "scalability" of parallel algorithms: in order to maintain an efficiency of e with p processors, the size of the problem processed should grow according to a function $iso(p)$.

Despite the obvious advantage of keeping acceleration anomalies possible, the interest in avoiding detrimental anomalies has been emphasized. In this context, Li and Wah (9) presented a special condition on the nodes with same priority that is sufficient to avoid degradation. Their method is attractive for depth-first search strategies where anomalous behavior is quite usual, and has been improved technically by Kalé and Saletore (12).

An example of anomaly with best-first strategy is given in Fig. 1.4. Since the cost of unfavorable anomalies needs to be compared with the price to forbid them, it may be worthwhile to consider the design and analysis of basic best-first search strategies, which deal with active nodes of same priority (same value), without either processing or memory overhead due to the removal of unfavorable anomalies.



(*a*)

Iterations	1	2	3	4	5	6	7	8	9	10	11	12	13
p_1	A	B	C	D	E	F	M	L	K	I	H	G	N

(*b*)

iterations	1	2	3	4
p_1	A	B	F	K
p_2		C	G	L
p_3		D	H	M
p_4		E	I	N

(*c*)

Iterations	1	2	3	4	5
p_1	A	B	K	G	N
p_2		C	L		
p_3		D	M		
p_4		E	H		
p_5		F	I		

(*d*)

Fig. 1.4. Example of Anomaly on a B&B. (*a*) The B&B tree. (*b*) Sequential scheduling with best-first search. (*c*) Parallel scheduling with 4 processors. (*d*) Parallel scheduling with five processors.

However, it is very important during the design process of a B&B algorithm to be able to define different priorities for subproblems (nodes) to be explored and to deal with nodes of equal priorities.

1.5. STRATEGIES FOR PARALLELIZATION

Most parallel B&B algorithms implement some form or another of tree exploration in parallel. The fundamental idea is that in most applications of interest the size of the B&B enumeration tree grows, possibly rapidly, to unmanageable proportions. So, if the exploration of the search tree is done more quickly by several processes, the faster acquisition of knowledge during the search (communication between processes) will allow for pruning more nodes or for eliminating more branches of the tree.

To describe the possible strategies, we start from a representation of the sequential B&B where a number of operations are performed on the data structure containing the work to be done (e.g., nodes in various states) and

the information relative to the status of the search (e.g., the incumbent value). Transposed in a parallel environment, the *tree management* strategies are presented as *search control* strategies implying *information* and *pool management* strategies. It is noteworthy that some of these strategies induce a tree exploration different from the one performed by the sequential method.

Recall from Section 1.2 that sequential B&B is fundamentally a recursive procedure that extracts a node (i.e., a subproblem) from the *pool*, thus deleting it from the data structure, performs a series of operations (evaluation, computation of upper bound, branching, etc.), and completes the loop by inserting one or several nodes (i.e., the new subproblems yielded by the branching operation) into the same pool.

Nodes in the pool are usually kept and accessed according to their *priority* based on various node attributes (e.g., lower and upper bound values, depth in the tree) and the search tree exploration strategy (e.g., best or depth-first). Node priorities thus define an order on the nodes of the pool, as well as the *sequential scheduling* of the search. An evident property of the sequential B&B search is that each time a node is scheduled, the decision has been taken with a complete knowledge of the information of the state of the search, which is the global view of all the pending nodes generated so far.

In a parallel environment, both the search decisions, including node scheduling ones, and the search information may be distributed. In particular, not all the relevant information may be available at the time and place (i.e., the processor) a decision is taken. Thus, first we examine issues related to the storage and availability of information in a parallel exploration of a B&B tree. We then turn to the role processors may play in such an exploration. The combination of various alternatives for these two components yield the basic strategies for parallel B&B algorithm design.

Two issues have to be addressed when examining the search information in a parallel context: (*1*) how is this information stored; (*2*) what information is available at decision time.

The bulk of the search information is made up of the pool of nodes, and *pool management* strategies address the first issue with respect to it: if and how to decompose the pool of nodes. A *centralized* strategy keeps all the nodes in a central pool. This implies that this unique pool serves, in one form or another, all processors involved in the parallel computation. Alternatively, in a *distributed* strategy, the pool is partitioned, each subset being stored by one processor. Other relevant information (e.g., global status variables like the value of the incumbent) is of limited size. Consequently, the issue is not whether it is distributed or not, but rather whether it is available in an up-to-date form when decisions are taken.

In a parallel B&B algorithm, more than one processor may decide, more or less simultaneously, to process a node. Their collective action corresponds to the *parallel scheduling* of the search, that is, to the management of the entire, but often distributed, pool and the corresponding distribution of work among processors.

The scheduling of nodes is based on the node priorities, defined as before in the sequential context. We define as *search knowledge*, the pool of nodes with their priorities, plus the incumbent value and the other global status variables of the search. The search knowledge may be *complete* or *partial*. If the search knowledge is complete, the resulting scheduling is very close to the sequential scheduling. Indeed, when at each step, the processor that has to make up a scheduling decision has an exact and complete knowledge of all the pending nodes to process, its decision is almost the same as in the sequential case. When only partial information is known to a processor, the scheduling could be really different compared to the sequential one.

When information is distributed, parallel scheduling must also include specific provisions to address a number of particular issues:

- The definition of a parallel initialization phase and of the initial work allocation among processors.
- The updating of the global status variables (e.g., the value of the incumbent).
- The termination of the search.
- The minimization of the idle time.
- The maximization of the meaningful work.

Search control strategies specify the role of each processor in performing the parallel search, that is, decisions relative to the extraction and insertion of nodes into the pool(s), the exploration strategy, the work to perform (e.g., total or partial evaluation, branching, offspring evaluation), the communications to undertake to manage the different pools, and the associated search knowledge.

From a search control point of view, we distinguish between two basic, but fundamental, types of processes: *master*, or *control*, and *slave* processes. Master processes execute the complete range of tasks. They specify the work that slave processes must do and fully engage in communications with the other master processes to implement the parallel scheduling of the nodes, control the exchange of information, and determine the termination of the search. Slave processes communicate exclusively with their assigned master process and execute the prespecified tasks on the subproblem they receive. Slave processes do not engage in scheduling activities.

The classical *master-slaves*, or *centralized control*, strategy makes use of these two types in a two-layer processor architecture, one master process and a number of slave processors (Fig. 1.5). This strategy is generally combined to a centralized pool management strategy. Thus, the master process maintains global knowledge and controls the entire search, while slave processes perform the B&B operations nodes received from the master processor and return the result to the master.

At the other end of the spectrum, one finds the *distributed control* strategy combined to a distributed pool management approach. In this case, sometimes also called *collegial* and illustrated in Fig. 1.6, several master processes

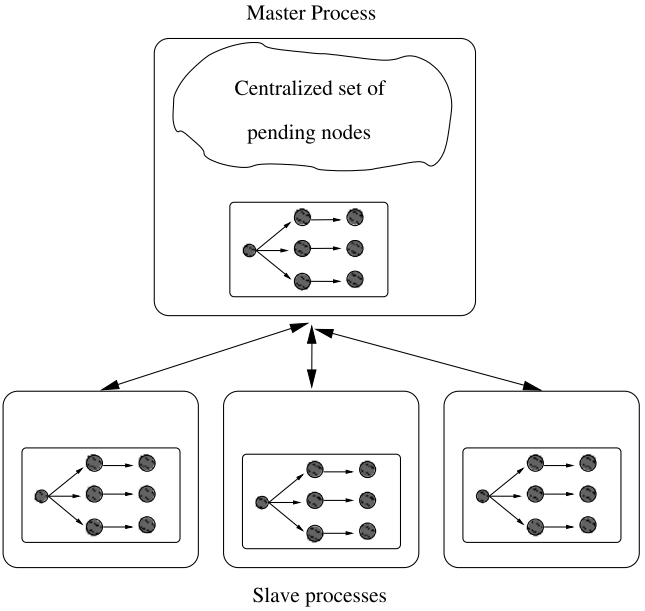


Fig. 1.5. Master-slave B&B algorithm.

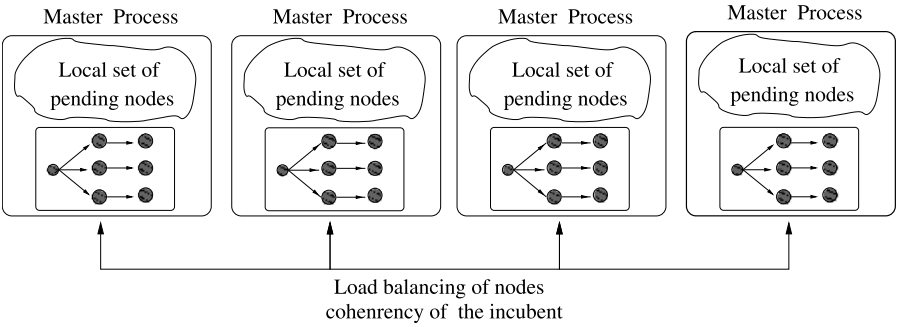


Fig. 1.6. Collegial B&B algorithm.

collegially control the search and the associated information. In this basic approach, there are no slave processes. Each master process is associated to one of the local pools that stores a subsets of the currently existing nodes (there is no sharing of local pools among several masters). It then performs the B&B operations on its local pool based on this partial node information, as well as on information transmitted by the other processes. The all-master processes combined activities thus makes up the partial-knowledge parallel scheduling.

The pool distribution and thus the search information distribution often results in uneven workloads for the various processes during the search. A *load balancing* strategy must then be implemented to indicate how the infor-

mation relative to the processor workloads circulates and how the corresponding load balancing decisions are taken. Information updating the global status variables (incumbent value, termination status, etc.) must also be exchanged. This communication policies set enforces the global control over the search. From a control point of view, two approaches are again available: either the decision is collegially distributed among the processes or it is (more or less) centralized. Thus, in the basic collegial strategy introduced above, all master processes may exchange messages (according to various communication topologies and policies) and decide collectively on how to equilibrate loads. Alternatively, one of the processors acts as load balancing master. It collects the load information and decides on the data exchanges to perform.

These strategies and types of processes are the basic building blocks that may be combined to construct more complex, hierarchical parallel strategies, with more than one level of information (pool) distribution, search control, and load balancing. Thus, for example, a processor could depend on another for its own load balancing, while controlling the load balancing for a group of lower level master processes, as well as the work of a number of slave processes.

1.6. BRANCH-AND-BOUND SOFTWARE

The range of purpose for B&B software is quite large, so it stands to reason that the numbers of problem types, user interface types, parallelization types, and machine types are also large.

This chapter focuses on software for which there exists an interface to implement Basic B&B, which means that one can use this type of library to implement one's own bounding procedure and one's own branching procedure. Several Libraries or applications exist, which are dedicated to one type of problems, commercial linear solvers like CPLEX, XPress-MP, or Open Source Software like GLPK (14) or lp_Solve are dedicated to solve Mixed Integer Linear program, but the B&B part is hidden in a "Black-Box". It could not be customized to implement an efficient parallel one.

However, these solvers could be used using a callable library. An application that implements a Parallel Basic B&B could use this kind of Solvers to compute the Bound. In this case, only the linear solver is used, the B&B part is ignored.

We first explain the designing tips used to develop this kind of software. Then we try to discuss the different Frameworks in terms of User algorithms they provide, the proposed Parallelization strategies, and finally the target machines.

1.6.1. Designing of a B&B FrameWork

When we consider the really basic form of B&B, *framework* is the only possible form of software. A Framework in this context is an implementation of

a B&B, with hooks that allow the user to provide custom implementations of certains aspects of the algorithm. For example, the user may wish to provide a custom branching rule, or custom bounding function. The customization is generally accomplished either through the use of C language callback functions or through a C++ interface in which the user must derive certain base classes and override default implementations for the desired function. In most cases, base classes and default implementations are abstract. The main algorithm, for example, the B&B loop, is written as a *skeleton* of algorithm. A skeleton uses abstract classes or abstract functions that must be redefined to obtain a concrete application.

In a Basic B&B, the abstractions are the following: node type, which includes the data in order to compute the bound of the subproblem and also stores the partial solution of the subproblem; solution type, which includes the value and the solution itself (the value of the variables), sometimes the type could be the same for the solution and for the subproblem; a function to evaluate the node, which according to the data stored in the node computes the evaluation of the partial solution, or the cost of a complete solution; a function used for branching, which must implement the method by which a subproblem is divided into subproblems according to the problem; a function to generate the root subproblem; and a function to generate the initial solution.

The Framework generally offers the following functionalities: the priority of a node that decides the order in which the subproblems will be explored. The choice of the node priority defines the search strategy; the management of the pending nodes, which is generally made using priority queues. As said in Section 1.5 about the parallelization strategy, the framework could store one or several priority queues, the managing of the pending nodes is then centralized or distributed; the management of the incumbent when a process finds a better solution, the framework must update it automatically in the other processes; and the main B&B loop that is executed by each process.

Let us present a modified version of Fig. 1.2 in order to present the data or function that must must be redefined by the user and the parts that are provided by the Framework. This design is not the unique possible design for a Framework. Authors of such a framework surely present theirs in a different way. But we believe that this design is what correponds more to reality.

The type Node represents a subproblem. As already mentioned, it must be defined by the user, in a Object Oriented Framework (written in C++, e.g.), the user defines its own type by derivating the type offered by the Framework.

Here, the type of the Solution (incumbent) is the same as the type of a subproblem; as said before, some Frameworks defined different types for the solution and for the nodes.

The type ParaPriorityQueue is provided by the Framework. It is used by the main loop to obtain a new node to be explored and to insert a new generated nodes.

```
// User Defined
procedure Branch(Node current, ParaPriorityQueue set) {
   foreach ( Node s son of current) { // Branching
      ... // fill the data of s according to the problem
      Evaluate(s); // Evaluation of the node s
                   // Redefined by the user
      if ( s.IsSolution() && incumbent.Cost()>s.Cost() ){
         incumbent= son;
         Update(incumbent); // FrameWork defined
      }
      else if ( s.Feasible() \&\& incumbent.Cost()>s.Eval() )
         set.Insert(s); // Framework defined
    }
}

// Framework defined
procedure MainLoop() {
   Node root;              // the root node.
   ParaPriorityQueue set; // the priority queue that  stores the
                           // set of pendings nodes
   Node incumbent;         // the Incubent.

   // Only executed by one processor
   if ( I am Master processor ) {
      Initialize(Root)  // User Defined
      set.Ins(root); // Framework defined
   }
   // The main loop
   while (! set.empty() ) { // Framework defined
      Node n =  set.deletemin();// Framework defined
      Branch(n,set); // User defined
   }
```

Fig. 1.7. Skeleton of a parallel B&B algorithm.

According to the parallelisation strategy, this type could have different implementations. First, in a centralized strategy, it is just an interface to receive or send nodes from and to the master process. Second, in a distributed strategy the ParaPriorityQueue stores a local priority queue, and also executes a load balancing procedure in order to ensure that each process has enough interesting nodes. This Data structure, or more generally this interface, could be seen as a high level communication tool. The main loop does not know where the nodes are really stored, but it just knows that this tool could be used to obtain and to insert nodes. The IsEmtpty method or function of the ParaPriorityQueue interface will be true only if no more Nodes exist in the entire application. In a distributed parallel machine like a cluster, for example, this interface will use a message passing library, like MPI or PVM, to transmit Nodes from one process to another.

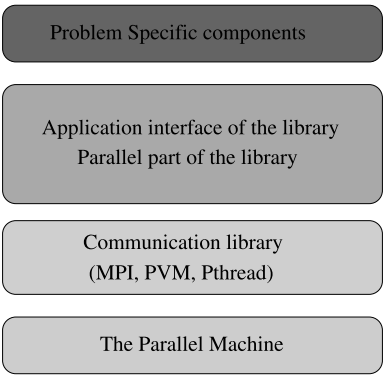


Fig. 1.8. Graphical view of a Framework and its interactions.

The Update function or method is used to broadcast to each process, the value or the new best solution found. The same communication library as the ParaPriorityQueue will be used. The management of the incumbent could be merged with the management of the Pending nodes.

Then we could see the Framework as a composition of different levels as presented in the Fig. 1.8. On the higher level, the user defines types and functions. On the low level the target machine.

There are many Frameworks for parallel B&B including:

1. PUBB (Univ. of Tokyo) (15).
2. BOB++ (Univ. of Versailles) (16).
3. PPBB-Lib (Univ. of Paderborn) (17).
4. PICO (Sandia Lab. & Rutgers University) (18).
5. FATCOP (University of Wisconsin) (19).
6. MallBa (Consortium of Spanish universities) (20).
7. ZRAM (Eth Zürich) (21).
8. BCP (CoinOR) (22).
9. ALPS/BiCePS (Leghih University, IBM, Clemson University) (22,23).
10. MW Framework (Leghih University) (24).
11. Symphony (Rice Univ.) (22,25,26).

They differ by the type of B&B they are able to solve, by the type of parallelization they propose, and then by the cibling machine.

Some of them propose a C interface [Bob (27), PPBB-Lib, MallBa, ZRAM, PUBB], although the other ones propose a C++ interface (Bob++, ALPS/BiCePS, PICO, MW).

1.6.2. User Algorithms

As already discussed, there exist a large variety of B&B: Basic B&B with a huge set of bounding procedures, Branch and Price, Branch and Cut. One could also consider that a simple divide and conquer algorithm is a base for a B&B algorithm.

With this multiplcity of methods, to make Framework easy to maintain, easy to use, and as flexible as possible, a possible design is a multilayered class library, in which the only assumptions made in each layer about the algorithm being implemented are those needed for implementing specific functionnality efficiently. Each of the proposed frameworks proposes a subset of these layers in core.

1.6.2.1. The Low-Level Algorithm: The Divide and Conquer. From a design point of view, a Simple tree search procedure like Divide and Conquer could be a base for a B&B procedure. Both are tree search procedures, but the B&B has additional functionalities, like prunning where the evaluation of the subproblem is used to discard a branch of the tree. For the parallelism, the differences of these two algorithms do not imply many modifications. If a parallel B&B is implemented, a parallel Divide and Conquer could also be implemented without a big overcost. For example, ZRAM, Bob++, ALPS, PUBB and MallBa implement simple tree search algorithms like backtracking. ZRAMand MallBA proposes Divide and Conquer and B&B has two different methods. Although, Bob++, ALPS, and PUBB, try to modelize Divide and Conquer as a base class of a B&B.

1.6.2.2. The Basic B&B. Bob, Bob++, PUBB, PPBB, ZRAM, PICO, MallBa, ALPS, BCP, and MW, propose an interface for a basic B&B. The interface are quite similar, and correspond to the design we presented in the previous section.

1.6.2.3. Mixed-Integer Linear Programming. A Mixed-Integer Linear Program could be solved using a B&B where the bounding operation is performed using tools from linear programming and polyhedral theory. Since Linear Programming does not accommodate the designation of integral variables, the integrality constraints are relaxed to obtain a linear programming relaxation. Most of time, this formulation is augmented with additional constraints or cutting planes, that is, inequalities valid for the convex hull of solutions to the original problem. In this way, the hope is to obtain an integral (and hence feasible) solution.

Each Framework that proposes a Basic B&B could also be used to solve Integer Linear Programs. The evaluation function and the Branching function could be written using a linear solver. The Coin Open Solver Interface (OSI) is very usefull for that. The OSI consists of a C++ base class with containers for storing instance data, as well as a standard set of problem import, export

modification, solution, and query routines. Each supported solver has a corresponding derived class that implements the methods of the base class. A nonexhaustive list of supported solver is CPLEX, XpressMP, lp_solve, GLPK, CLP.

However, an important feature that must be included in Frameworks to solve such problems efficiently, is a way to globally store the set of already generated cutting planes. One cutting plane that has been generated for one subproblem could be valid for another subproblem. Then, to avoid redundant computation, cutting planes should be stored in a "global data structure", from which processes could obtain an already generated cutting plane without regenerating it. The ALPS/BiCePS, PICO, SYMPHONY, and BCP Frameworks propose such data structure. The ALPS framework introduces the notion of Knowledge Base (KB).

1.6.2.4. Branch and Price and Cut. The Frameworks that propose a native interface for Branch and Price and Cut are Symphony and ALPS/BiCePS. As SYMPHONY is written in C, its user interface consists in implementing callback funtions including cutting-planes, generation, management of cut pool, management of the LP relaxation, search and dividing strategies, and so on.

ALPS/BiCePs, which seems to be the Symphony replacement should also propose an interface for Branch, Price, and Cut. Unlike Symphony, ALPS/BiCeps is layered allowing the resolution of sevral types of B&B. Symphony seems to be only for Branch and Cut.

1.6.2.5. Other User Methods. Another very interesting method that could also be offered by this kind of Framework is a graph search procedure. Bob++, ZRAM and MallBa propose an interface to develop Dynamic Programming application. In this kind of method, the difficulty is the parallel management of the state space. In a parallel environment the state space must be global, thus an algorithm to maintain the coherency of the state space between all the processes must be proposed.

1.6.3. Parallelization Strategies

Symphony and MW, use the master-worker paradigm (see Section 1.5) to parallelize the application. The nodes management is thus centralized. One process, the master, controls all the aspects of the search. In the other processes, the workers do the work (an exploration of one or several nodes) provided by the master. But as the task unity is the subtree and not the subproblem, each worker performs a search on a subtree, thus it could be considered that the worker controls its search. Hence, the control of the search is distributed. MW (24) has an interesting feature in a sense that it is based on a Fault Tolerant communication library Condor (28). As said before, this strategy works well for a small number of processors, but does not scale well, as

the central pool inevitably becomes a computational and communications bottleneck.

ALPS (23) and PICO (18) propose to use the Master-Hub-Worker paradigm, to overcome the drawbacks of the master-Worker approach. A layer of middle management is inserted between the master and worker process. In this scheme, "a cluster" consists of a hub, which is responsible for managing a fixed number of workers. As the number of processes increases, more hubs and cluster of workers are simply added. This decentralized approach maintains many advantages of global decision making while reducing overhead and moving some computational burden from the master process to the hubs.

The other libraries propose one or several distributed strategies. Some of them have used PVM (29) (PUBB, Bob, PPBB-Lib, ZRAM), but the modern ones use MPI (Bob++, ALPS, PICO, MallBa). PPBB-Lib proposes a fully distributed parallel B&B, where each process stores a local pool of subproblems. Several Load balancing strategies were proposed in order to ensure that each local pool has enough work.

In the vast majority of these frameworks, there is no easy way to extend the parallel feature. Only one parallel algorithm and one communication layer are proposed. As its ancestor Bob, Bob++ also proposes an interface to extend the parallel part of the library. Master–Slave, fully distributed, or mixed strategy could be implemented using this interface. Master–Slave, distributed versions of parallelisations exist for the Bob Library.

1.6.4. The Target Machines

According to the accessibility of the authors to specific machine, Frameworks have been ported to various types of machines. For example, Bob has been ported on PVM (29), MPI, PM^2 (30), Charm++ (31), Athapascan (32), POSIX threads, using shared memory machines, and then using distributed machines. The recent architecture being clusters of shared memory machines (SMP) or grid-computers, the current frameworks have proposed versions for these machines. MallBA, PICO, and ALPS use MPI as their message passing library. Bob++, MallBa, PICO, and ALPS run very well on a massively parallel distributed memory system. Bob++ (Athapascan based version) could also run on a cluster of SMPs, where a multithreading programming paradigm is used on each SMP node. Bob++ has been succesfully tested on a Grid (see Section 1.7).

As far as we know, only Bob++, MW, PICO, ALPS, MallBa, and Symphony are still being maintained.

1.7. ILLUSTRATIONS

Parallel B&B algorithms have been developped for many important applications, such as the Symmetric Traveling Salesman problem [Applegate, Bixby,

Chvatal, and Cook (1)], the Vehicle Routing problem [Ralphs (33), Ralphs, Ladány, and Saltzman (34)], and the multicommodity location and network design [Gendron and Crainic (35), Bourbeau, Gendron, and Crainic (36)], to name but a few. In this section, we illustrate the parallel B&B concepts introduced previously by using the case of the Quadratic Assignment Problem. The description follows the work of (37).

The Quadratic Assignment Problem (QAP) consists to assign n units to n sites in order to minimize the quadratic cost of this assignment, which depends on both the distances between the sites and the flows between the units. It can be formulated as follows:

Given two $(n \times n)$ matrices,

$F = (f_{ij})$ where f_{ij} is the flow between units i and j,
$D = (d_{kl})$ where d_{kl} is the distance between sites k and l

find a permutation p of the set $N = \{1,2, \cdots ,n\}$, which minimizes the global cost function:

$$Cost(p) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i)p(j)}$$

Although the QAP can be used to formulate a variety of interesting problems in location, manufacturing, data analysis, and so on, it is a *NP*–hard combinatorial problem, however, and, in practice, extraordinarily difficult to solve even for what may seem "small" instances. This is very different compared to other *NP*–hard combinatorial optimization problems, for example, the Traveling Salesman problem, for which, in practice, very large instances have been successfully solved to optimality in recent years. To illustrate this difficulty, the B&B for QAP develops huge critical trees. Thus, for example, the tree for Nugent24 has 48,455,496 nodes with bounds for values in [3488, 3490], while for Nugent30, the tree has 12,000,000,000 nodes for 677 values in interval [5448, 6124]. This extreme difficulty in addressing the QAP has resulted in the development of exact solution methods to be implemented on high-performance computers (7,38–43).

The parallel B&B presented in this section is based on the serial algorithm of Mautor and Roucairol (11) and uses the lower bound procedure (DP) of Hahn-and-Grant (44). This lower bound is based upon a new linear formulation for the QAP called "level_1 formulation RLT". A new variable is defined $y_{ijkl} = x_{ij}x_{kl}$ and the formulation becomes

$$(QAP:) Min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1 k \neq i}^{} \sum_{l=1 l \neq j}^{n} C_{ijkl} y_{ijkl} + \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\sum_{k=1k\neq i}^{n} y_{ijkl} = x_{ij} = 0,1 \qquad \forall (i,j), l \neq j \tag{1.1}$$

$$\sum_{l=1l\neq j}^{n} y_{ijkl} = x_{ij} = 0,1 \qquad \forall (i,j), k \neq i \tag{1.2}$$

$$y_{ijkl} = y_{klij} \qquad \forall (i,j), k \neq i, l \neq j \tag{1.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1, \ldots, n \tag{1.4}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1, \ldots, n \tag{1.5}$$

$$y_{ijkl} \geq 0 \qquad \forall (i,j,k,l), k \neq i, l \neq j \tag{1.6}$$

The Linear programming relaxation of this program provides a lower bound to the QAP. But, as the number of variables and constraints is huge, it cannot be computed by commercial softwares. Hahn-and-Grant proposed a procedure called DP based on a successive dual decomposition of the problem (44). The DP bound is obtained iteratively by solving $\tilde{n}2 + 1$ linear assignment problems (using the Hungarian method), instead of solving the large linear program given above.

A "polytomic" branching strategy is used (45). This strategy extends a node by creating all assignments of an unassigned facility to unassigned locations based upon the counting of *forbidden* locations. A forbidden location is a location where the addition of the corresponding leader element would increase the lower bound beyond the upper bound. At a given unfathomed node, we generate children according to one of the following schemes

- *Row Branching*. Fix $i \in I$. Generate a child problem for each $j \in J$ for which the problem with $X_{ij} = 1$ cannot be eliminated;
- *Column Branching*. Fix $j \in J$. Generate a child problem for each $i \in I$ for which the problem with $X_{ij} = 1$ cannot be eliminated.

Several different branching strategies to choose the candidate for next generation i (row) or j (column) have been considered:

1. SLC: Choose row i or column j, which maximizes the sum of leaders.

$$Max(\sum C_{ijij})$$

2. SLC_v2: Add the nonlinear elements of each submatrix to the associated leader and choose row or column with maximal sum.

$$Max\left(C_{ijij} + \sum_{k=1}^{n}\sum_{l=1}^{n} C_{ijkl}\right)$$

3. SLC_v3: Choose a row or column with a maximal sum of elements in their associated submatrices.

The implementation of the B&B algorithm is made with Bob++ framework (16) on top of the Athapascan environment. Athapascan is a macrodata-flow application programming interface (API) for asynchronous parallel programming. The API permits to define the concurrency between computational tasks that make synchronization from their accesses to objects into a global distributed memory. The parallelism is explicit, and functional, the detection of the synchronization is implicit. The semantics is sequential and an Athapascan program is independent from the target parallel architecture (cluster or grid). The execution of the program relies on an interpretation step that builds a macrodata-flow graph. The graph is direct and acyclic (DAG) and it encodes the computation and the data depencies (read and write). It is used by the runtime support and the scheduling algorithm to compute a schedule of tasks and a mapping of data onto the architecture. The implementation is based on using a light-weight process (thread) and one-side communication (active message).

In the context of B&B, an Athapascan task is a subproblem. The execution of the task yields to the exploration of a subtree of the B&B tree. The set of ready tasks (the subproblems) is distributed on the processors. Each processor stores a list of ready tasks locally. As Athapascan performs load balancing on the lists to insure maximal efficiency, each processor has a subproblem to work on. Each Athapascan's list of ready tasks is equivalent to a local priority queue that stores a subproblem in a parallel B&B. Then, according to the strategies listed in Section 1.5, the Bob++/Athapascan Framework proposes a fully distributed strategy.

First, we present results on a small cluster (named COCTEAU, located at the PRiSM laboratory, University of Versailles): 7 DELL workstations (Xeon bi-proc, 2.4 GHz, 2 GB RAM, 5 Go memory, Red Hat 7.3). The cluster was not dedicated, but was very steady. Table 1.1 displays the results.

TABLE 1.1. Runtime of Parallel QAP on COCTEAU Cluster of Size 10

Instance	Cocteau Cluster, SLC		
	Sequential Time, min	Parallel Time, min	Efficiency
Nug18	23.31	2.43	0.959
Nug20	246.98	24.83	0.994
Nug21	548.31	54.86	0.999
Nug22	402.43	44.23	0.909
Nug24	10764.2	1100.56	0.978

Performances are very good, as the efficiency is close to 1. The performance decreases slightly as the size increases up to 24. To show that the results are not machine dependant, we display the results of experiments with the same number of processors on a larger cluster (located in Grenoble and named I-cluster 2): 104 HP nodes (bi-Itanium-2900 MHz, 3 GB RAM, Intel icc 8 Compiler) interconnected by a Myrinet network. As displayed in Table 1.2, with 14 processors, the performances are similar, while for two problems, Nugent 18 and 20, favorable anomalies appear with a speedup >14.

Tables 1.1 and 1.2 show the results with the SLC Branching strategy presented above. We compare different branching strategies (SLC, SLC_v2 SLC_v3) using the small cluster (14 processors) implementation. Results are displayed in Table 1.3. The total number of explored nodes is comparable with a small advantage to strategy SLC_v3. Notice that the optimal solution is known for Nugent problems of size up to 30, and thus the upper bound is initialized to this value plus one unit. The parallel branch and bound algorithm developed the critical tree (critical nodes C representing incomplete solutions with a value strictly smaller than the optimum solution f^*) and explored some nodes in the set of undecidable nodes M, nonterminal or incomplete solutions with a value equal to f^*, and one node in O (optimal nodes with value f^*). Indeed, the set E of eliminated nodes (with a value strictly $>f^*$) is empty. But

TABLE 1.2. Performance of Parallel QAP on I-Cluster 2 of Size 14

Instance	Running time, min			Performances	
	Sequential	Parallel(7CPUs)	Parallel(14CPUs)	Speedup	Efficiency
Nugent18	54.24	7.54	3.39	16	114.28
Nugent20	588.46	84.29	41.96	14.02	100.17
Nugent21	1208.9	173.53	87.14	13.87	99.09
Nugent22	959.7	138.14	69.45	13.81	98.7

TABLE 1.3. Comparison of Branching Strategies

Instance	SLC_v2		SLC_v3	
	Nodes	Time	Nodes	Time
Nug12	11	01.01	10	01.03
Nug14	121	04.04	122	04.04
Nug15	248	14.16	239	10.13
Nug17	1,653	173.74	1,674	107.07
Nug18	4,775	593.90	4,744	369.70
Nug20	42,223	5,248.15	42,232	4,892.72
Nug22	47,312	7,235.75	47,168	7,447.54
Nug24	266,575	73,821	265,687	69,365

TABLE 1.4. Runtime of Parallel B&B for QAP on a Large Cluster with 80 Processors

Instance	I-Cluster2, SLC				
	1CPU	20CPUs	40CPUs	50CPUs	80CPUsb
Nug18	11.07	1.07	1.6	0.53	n.a
Nug20	149.06	5.78	4.56	3.38	4.46
Nug21	61.28	2.41	2.11	1.55	2.98
Nug22	214.7	8.26	5.81	4.56	6.01
Nug24	n.a	82.18	44.35	37.25	27.96

the set M could be very large and, according to the branching strategy, the size of the explorated tree could be very close to the size of the minimal tree. For Nugent 24, at the root node, the search interval is very thin in comparison with the number of explored nodes, while, later, many nodes have equal evaluation.

In addition, tests have been conducted on a higher number of processors, on the I-cluster 2. Table 1.4 shows that the results for the problem Nugent 24 up to 80 processors are very good. The other problems are too small to have the time to use all the 80 processors. This is the reason why the times with 80 processors is slightly higher than those with 50 processors.

To complete this illustration, we discus the implementation of the parallel B&B algorithm for the QAP using *Grid Computing*. Grid computing (or metacomputing or using a computational grid) is the application of the resources of many geographically distributed, network-linked, heterogeneous computing resources to a single problem at the same time. The main features of metacomputing are

- Dynamically available computing resources: Machines may join the computation at any time.
- Unreliability: Machines may leave without warning due to reboot, machine failure, network failure, and so on.
- Loosely coupling: Huge network, "low" communication speed, communication latency are highly variable and unpredictable.
- Heterogeneous resources: Many different machines (shared workstations, nodes of PC clusters, supercomputers) and characteristics (memory, processors, OS, network latency, and so on.

Programs should therefore be self-adaptable and fault tolerant. Grid computing requires the use of software based upon resource management tools provided by projects like Globus (46), Legion (47), and Condor (28). The goal of these tools is to assign processors to a parallel application. These software do not perform load balancing between the processes, however.

The advantage of such a platform compared to a traditional multiprocessor machine is that a large number of CPUs may be assembled very inex-

pensively. The disadvantage is that the availability of individual machines is variable and communication between processors may be very slow.

We experimented our parallel QAP algorithm, using the Athpascan environment, on a French grid initiative, called e-Toile, where six locations with clusters or supercomputers were linked by a very high-speed network. The middleware used was the e-Toile Middleware that was an evolution of the Globus middleware. We obtained and proved the optimal solution to the Nugent 20 instance in a time of 1648 secondes (>4 h), using a cluster of 20 machines (AMD MP, 1800+) at one location, while only 499 s (8,31 min) were required with the following 84 machines located in different sites:

- 7*bi-Xeons 2.4 GHz, 2 Go (Laboratory PriSM-Versailles).
- 10*bi-AMD MP 1800, 1 Go (CEA-Saclay).
- 10*bi-Xeons, 2.2 GHz, 2 Go (EDF-Clamart).
- 15*bi-PIII, 1.4 GHz, 1 Go (ENS-Lyon).

A relative speedup of $T_{20}/T_{84} = 3.1$ was thus achieved.

1.8. CONCLUSION

We have presented a summary of basic concepts for parallel B&B methodology applied to hard combinatorial optimization problems. This methodology is achieving very impressive results and is increasinly "accessible" given the continuously decrease in the costs of computers, networks, and communication devices.

We have also presented several software tools that help to implement Parallel B&B algorithms. These tools offer a large variety of interfaces and tools that allow the user to implement algorithms from the basic B&B to the most sophisticated branch and cut. However, a lot of work has to be done to include fault-tolerance, self-adaptibility, multiapplication, heterogeneity in these tools.

We also present the resolution of the Quadratic assignement problem,for which the parallelism is really a great feature to solve instances that could never been approach before.

REFERENCES

1. D. Applegate, R.E. Bixby, V. Chvatal, and W. Cook. On the solution of traveling salesman problem. *Doc. Math.*, **ICM**(III):645–656 (1998).
2. K.M. Anstreicher and N.W. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. *Math. Prog.*, **89**(3):341–357 (2001).
3. J.P. Goux, K.M. Anstreicher, N.W. Brixius, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Math. Prog.*, **91**(3):563–588 (2002).

4. B. Gendron and T.G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Oper. Res.*, **42**(6):1042–1066 (1994).
5. G. Authié *et al*. *Parallélisme et Applications Irrégulières*. Hermès, 1995.
6. C. Roucairol. A parallel branch and bound algorithm for the quadratic assignment problem. *Discrete Appl. Math.*, **18**:211–225 (1987).
7. B. Mans, T. Mautor, and C. Roucairol. A parallel depth first search branch and bound algorithm for the quadratic assignment problem. *Eur. J. Oper. Res. (Elsevier)*, **3**(81):617–628 (1995).
8. T.-H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communication A.C.M.*, **27**:594–602 (June 1984).
9. G. Li and B.W. Wah. Coping with anomalies in parallel branch-and-bound. *IEEE Trans. Comp.*, **C-35**(6):568–573 (June 1986).
10. C. Roucairol. Recherche arborescente en parallèle. RR M.A.S.I. 90.4, Institut Blaise Pascal—Paris VI, 1990. In French.
11. B. Mans and C. Roucairol. Theoretical comparisons of parallel best-first search branch and bound algorithms. In H. France, Y. Paker, and I. Lavallée, eds., *OPOPAC, International Workshop on Principles of Parallel Computing*, Lacanau, France, November 1993.
12. L.V. Kalé and Vikram A. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *Inter. J. Parallel Prog.*, **19**(4):251–293 (1990).
13. Anshul Gupta and Vipin Kumar. Scalability of parallel algorithms for matrix multiplication. *Proceedings of the 1993 International Conference on Parallel Processing*, Vol. III—Algorithms & Applications, CRC Press, Boca Raton, FL, 1993, pp. III-115–III-123.
14. Andrew Makhorin. Glpk (gnu linear programming kit). Available at http://www.gnu.org/software/glpk/glpk.html.
15. Y. Shinano, M. Higaki, and R. Hirabayashi. A generalized utility for parallel branch and bound algorithms. *Proceedings of the 7nd IEEE Symposium on Parallel and Distributed Processing (SPDP '95)*, 1995, pp. 858–865. Available at http://al.ei.tuat.ac.jp/yshinano/pubb/.
16. B. Le Cun. Bob++ framework: User's guide and API. Available at http://www.prism.uvsq.fr/blec/Research/BOBO/.
17. S. Tschoke and T. Polzer. Portable parallel branch-and-bound library user manuel, library version 2.0. Technical Report, University of Paderborn, 1998.
18. J. Eckstein, C.A. Phillips, and W.E. Hart. Pico: An object-oriented framework for parallel branch-and-bound. Technical Report 40-2000, RUTCOR Research Report, 2000.
19. Q. Chen and M.C. Ferris. FATCOP: A fault tolerant condor-PVM mixed integer program solver. Technical Report, University of Madison, Madison, w1, 1999.
20. E. Alba *et al*. Mallba: A library of skeletons for combinatorial optimisation. In B. Monien and R. Feldman, eds., *Euro-Par 2002 Parallel Processing*, Vol. 2400 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Heidelberg, 2002, pp. 927–932.
21. A. Brunnger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench zram and its applications. *Ann. Oper. Res.*, **90**:45–63 (1999).
22. The COIN-OR Team. The coin-or organization. http://www.coinor.org/.

23. Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. Alps: A framework for implementing parallel search algorithms. *The Proceedings of the Ninth INFORMS Computing Society Conference* (2005).
24. J. Goux, J. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm, Technical Report, Angcnne National Laboratory. 1999.
25. T.K. Ralphs and M. Guzelsoy. The symphony callable library for mixed integer programming. *The Proceedings of the Ninth INFORMS Computing Society Conference*, San Francisco, USA, (2005).
26. T.K. Ralphs. Symphony 3.0.1 user's manual. http://www.branchandcut.org/.
27. M. Benaichouche *et al*. Bob: une plateforme unifiée de développement pour les algorithmes de type branch-and-bound. RR 95/12, Laboratoire PR*i*SM, Université de Versailles—Saint Quentin en Yvelines, May 1995. In french.
28. M. Litzkow, M. Livny, and M.W. Mutka. Condor—a hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS '88)*, 1988, pp. 104–111.
29. V.S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, **2**(4):315–340 (1990).
30. J.F. Mehaut and R. Namyst. *PM^2: Parallel Multithreaded Machine. A multithreaded environment on top of PVM*. *Proceedings of EuroPVM'95*, Lyon, September 1995.
31. L.V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, eds., *Parallel Programming using C++*, MIT Press, 1996, pp. 175–213. Boston, MA.
32. T. Gautier, R. Revire, and J.L. Roch. Athapascan: an api for asynchronous parallel programming. Technical Report, INRIA RT-0276, 2003.
33. T.K. Ralphs. Parallel Branch and Cut for Capacitated Vehicle Routing. *Parallel Comp.*, **29**:607–629 (2003).
34. T.K. Ralphs, L. Ladány, and M.J. Saltzman. Parallel Branch, Cut, and Price for Large-Scale Discrete Optimization. *Math. Prog.*, **98**(13):253–280 (2003).
35. B. Gendron and T.G. Crainic. A Parallel Branch-and-Bound Algorithm for Multicommodity Location with Balancing Requirements. *Comp. Oper. Res.*, **24**(9):829–847 (1997).
36. B. Bourbeau, B. Gendron, and T.G. Crainic. Branch-and-Bound Parallelization Strategies Applied to a Depot Location and Container Fleet Management Problem. *Parallel Comp.*, **26**(1):27–46 (2000).
37. A. Djerrah, V.D. Cung, and C. Roucairol. Solving large quadratic assignement problems, on clusters and grid with bob++/athapascan. *Fourth International Workshop of the PAREO Working Group on Parallel Processing in Operation Research*, Mont-Tremblant, Montreal, Canada, January 2005.
38. J. Crouse and P. Pardalos. A parallel algorithm for the quadratic assignment problem. *Proceedings of Supercomputing 89*, ACM, 1989, pp. 351–360.
39. A. Brüngger, A. Marzetta, J. Clausen, and M. Perregaard. Solving large-scale QAP problems in parallel with the search library ZRAM. *J. Parallel Distributed Comp.*, **50**(1–2):157–169 (1998).
40. V-D. Cung, S. Dowaji, Cun B. Le T. Mautor, and C. Roucairol. Concurrent data structures and load balancing strategies for parallel branch-and-bound/a*

algorithms. *III Annual Implementation Challenge Workshop, DIMACS*, New Brunswick NJ, October 1994.

41. B. Le cun and C. Roucairol. Concurrent data structures for tree search algorithms. In J. Rolim A. Ferreira, ed., *IFIP WG 10.3, IRREGULAR94: Parallel Algorithms for Irregular Structured Problems*, Kluwer Academic, Gereva, Swizterland, September 1994, pp. 135–155.
42. Y. Denneulin, B. Lecun, T. Mautor, and J.F. Mehaut. Distributed branch and bound algorithms for large quadratic assignment problems. *5th Computer Science Technical Section on Computer Science and Operations Research*, Dallas, TX, 1996.
43. Y. Denneulin and T. Mautor. Techniques de régulation de charge—applications en optimisation combinatoire. *ICaRE'97, Conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, Aussois, 1997, pp. 215–228.
44. P. Hahn and T. Grant. Lower bounds for the quadratic assignment problem based upon a dual formulation. *Oper. Res.*, **46**:912–922 (1998).
45. T. Mautor and C. Roucairol. A new exact algorithm for the solution of quadratic assignment problems. *Discrete Appl. Math.*, **55**:281–293 (1994).
46. Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *Inter. J. Supercomp. Appl. High Performance Compu.*, **11**(2):115–128 (Summer 1997).
47. A. Natrajan *et al*. The legion grid portal. *Grid Computing Environments 2001, Concurrency and Computation: Practice and Experience*, 2002. Vol. 14 No 13–15 pp. 1365–1394.