BOB : a Unified Platform for Implementing Branch-and-Bound like Algorithms

B. Le Cun,
C. Roucairol and The PNN Team

N ° 95/16

BOB : a Unified Platform for Implementing Branch-and-Bound like Algorithms*

B. Le Cun†
C. Roucairol† and The PNN Team

Thème/Theme 1 — Parallèlisme/Parallelism
Équipe/Team PNN‡

Rapports de Recherche n ° 95/16 — — 21 pages

Abstract: In this report, we propose the library BOB for an easy development of the Branch-and-Bound applications (min/maximization). This library has the double goal of allowing on the one hand the Combinatorial Optimization community to implement their applications without worrying about the architecture of the machines and benefiting the advantages provided by parallelism. On the other hand, BOB offers to the community of Parallelism a set of benchmark composed by the efficient algorithms of Combinatorial Optimization for its parallelization methods and/or tools.

To achieve this double goal, the BOB library is founded on the notion of global priority queue which makes the parallelization methods independent from the applications, and vice-versa. We describe for this global priority queue different implementation models (asynchronous, synchronous, client/server, ...) according to the type of used machine (serial, parallel with shared or distributed memory).

A set of serial and concurrent data structures (D-Heap, Skew-Heap, Implicit-Heap, Funnel-Tree, Splay-Trees, ...) is provided to achieve the global priority queue.

We also emphasis on the conception of BOB and its main components (user functions, kernel functions and monitor), in particular the management of the global upper/lower bound in parallel environment. Finally, we show with an example how easy to develop Branch-and-Bound applications with this library.

Key-words: Branch-and-Bound method, parallel and distributed algorithms, library of functions, data structures, priority queues, concurrency, load balancing.

The PNN team members : M. Benaïchouche, V.-D. Cung, S. Dowaji, T. Mautor.

(Résumé : tsvp)

‡Email : pnn@prism.uvsq.fr

*This work was partially supported by the project Stratagème of the french CNRS.

†May also be contacted at *INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, FRANCE.*

Laboratoire PRiSM
Université de Versailles - Saint Quentin en Yvelines
78035 Versailles Cedex, FRANCE.
Téléphone/Phone : (33 1) 39 25 40 56 – Télécopie/Fax : (33 1) 39 25 40 57

BOB : une Plate-forme Unifiée de Développement pour les Algorithmes de type Branch-and-Bound[‡]

Résumé : Nous proposons, dans ce rapport, une bibliothèque d'aide au développement d'applications de type Branch-and-Bound BOB (min/maximisation). Cette bibliothèque a le double objectif d'une part, de permettre à la communauté de l'Optimisation Combinatoire d'implémenter ses applications sans se soucier de l'architecture des machines et de profiter des avantages du parallélisme; et d'autre part, d'offrir à la communauté de Parallélisme un banc d'essai, composé d'algorithmes performants de l'Optimisation Combinatoire, pour ses méthodes et/ou outils de parallélisation.

Pour réaliser ce double objectif, la bibliothèque BOB est fondée sur la notion de file de priorité globale qui permet de rendre transparentes les méthodes de parallélisation vis à vis des applications, et vice-versa. Nous décrirons pour cette file de priorité globale, différents modèles d'implémentation (asynchrone, synchrone, client/serveur, ...) suivant le type de machine utilisée (séquentiel, parallèle à mémoire partagée ou à mémoire distribuée).

Un ensemble de structures de données séquentielles et concurrentes (D-Heap, Skew-Heap, Implicite-Heap, Funnel-Tree, Splay-Trees, ...) est proposé pour la réalisation de la file de priorité globale.

Nous détaillerons également la conception de BOB et ses principales composantes (fonctions utilisateurs, fonctions noyau et moniteur), notamment la gestion en environnement parallèle de la borne sup/inférieure globale. Puis, nous montrons, à l'aide d'un exemple, la facilité de développement des applications de type Branch-and-Bound, avec cette bibliothèque.

Mots-clé : Méthode Branch-and-Bound, algorithmique parallèle et distribuée, bibliothèque de fonctions, structures de données, files de priorité, concurrence, équilibrage de charge.

[‡]Ce travail est partiellement financé par le projet Stratagème du CNRS.

Contents

1 The need for a Branch-and-Bound library

All Branch-and-Bound applications use the same components [4, 15, 19, 22, 26, 28, 29, 33]. When someone writes Branch-and-Bound algorithms to resolve real-life problems, often he rewrites (implements) these components : Management of the priority queue, management of the upper bound, etc. He can not concentrate his work for the essential parts of the application (generation, evaluation). Moreover, the center of his interest being on the Combinatorial Optimization side, he often makes too little effort to optimize the implementational parts of his application like the priority queue. For example, he might use a priority queue in the form of a D-Heap (or implicit-Heap) while other forms of priority queues might be much more efficient. The total execution time of the application would be penalized. In sequential implementation, a library should then facilitate the development of an application of the type Branch-and-Bound and should offer efficient components.

Moreover, parallelism seems to be an interesting option to resolve larger problems more quickly [3, 5, 12, 16, 24, 36]. However, when someone writes an efficient sequential Branch-and-Bound application, he rarely ports it on a parallel machine. Although the use of such machine would retard the combinatorial explosion. We will demonstrate that in the case of the parallelization of a search, it is possible to make the parallelism transparent for the application. That facilitates enormously the development and the portability of an application using the Branch-and-Bound method on parallel machines.

Another good reason for the presence of a library is to be able to validate the parallelizations of Branch-and-Bound on concrete examples. As a matter of fact, researchers working on the parallelization of an algorithm work little on the resolution of the problem from a Combinatorial Optimization point of view. They often use a simple example of a Branch-and-Bound application which does not necessarily reflect a good execution model of their parallelization [21, 30].

Writing a library is also a good mean to try to give a standard to write Branch-and-Bound applications in order to facilitate the exchange of code between researchers. Our library BOB is a way to join the two communities : Combinatorial optimization and parallel algorithmic.

In our knowledge, there exist similar libraries. However, they are not as much dedicated to Branch-and-Bound methods [11, 16] and subsequently they offer lesser facilities to develop applications. They do not neither cover all types of machines. The targeted machines are only distributed-memory ones [35]. Moreover, they do not offer to the user a set of data structures as general as BOB.

2 Principle of functioning of the library

First of all, we will describe our own conception of a Branch-and-Bound program by giving a unified vision of the algorithm on serial machines, parallel machines with shared or distributed memory or even on a network of workstations.

Then we will isolate the components which are to be implemented by the programmer of the application and those being able to be implemented by the BOB library in function of the architecture of the machine.

2.1 A Branch-and-Bound program

A Branch-and-Bound program is composed of three parts or principle procedures.

Initialization : The goal of this procedure is to initialize certain variables like the best known solution, its cost (the lower bound in a minimization and the upper bound in a maximization), and the different informations proper to the application like matrices, vectors, etc.

Search : This procedure is intended to visit the search tree. It includes the generation (with the principle of branching) and the evaluation of the children nodes (subproblems). In order to give the possibility to choose between different search strategies, we have isolated in this procedure a subprocedure. This subprocedure generates and evaluates the children nodes of exactly one node passed to it via a parameter.

The exploration of a child node is either postponed by inserting it into a priority queue, or immediate by a recursive call to this subprocedure. Hence, Best First and Depth First search strategies could be tested for the same application. Moreover, that would permit mixed strategies where certain subtrees, chosen by the best first strategy, are explored in a depth first manner for a certain height. Therefore, a regulation of the "granularity" of computation between two accesses to the priority queue is made possible. This procedure makes certain operations on a priority queue. This priority queue contains the nodes waiting to be explored. This procedure should also be able to access the lower/upper bound in order to update it when a new solution has been found or to make sure that a certain node must be explored.

Thus, the components presented here are the functions of generation and evaluation (Branching and Bounding), the priority queue and the lower/upper bound.

End : This procedure displays the found solution and different measures like the execution time, the total number of explored nodes, the number of operations on the priority queue, etc.

2.2 The parallelism

In the case of a parallel Branch-and-Bound processes execute the search procedure. They must communicate in order to get work (nodes to be processed) and in order to maintain a global knowledge of the lower/upper bound. In the literature, there has been proposed many parallel algorithms which permit the management of work between processes. For example, in the case of a shared memory machine, processes concurrently access a common priority queue [6, 14, 17, 20, 27, 32]. For a distributed memory machine each process locally manages a priority queue and maintains a workload by using a load balancing strategy [7, 18, 34]. Other algorithms [1, 8, 13, 25] are based on the client/server

model. A specialized process - the server - loads the nodes in a local priority queue. The other processes - the clients - ask the server for nodes to be explored and send back the generated subtrees.

From a conceptual point of view, all of these algorithms propose a management of a global priority queue. Processes exploring nodes get their workload by executing an algorithm participating in the management of the global priority queue (GPQ).

The notion of GPQ encapsulates two components :

1. the parallel management algorithm (concurrent access protocol, load balancing, etc),
2. the local or shared priority queue(s), founded on the sequential algorithms of priority queues.

It is this important notion of global priority queue which has made the conception of our BOB library possible. It permits an abstraction of the architecture of machines from the Branch-and-Bound algorithm point of view. That yields to a unified search procedure for the sequential as well as for the parallel exploration. Also, we should introduce the notion of global lower/upper bound (GL/UB). The value of GL/UB must be known by all the processes. Whenever it is updated, this value should be diffused to all the other processes in order to avoid redundant work. Quite often the parallel management of GL/UB is tightly linked to the GPQ algorithm.

2.3 The critical components of an application

This model of Branch-and-Bound program needs the five components described before.

1. **The initialization and the generation/evaluation procedure**. They directly depend on the nature of the problem to be treated. Therefore, the user of the library has to write these functions.
2. **The global priority queue**. It could use different algorithms according to the architecture of the parallel machine and to the choice of the user. This components encapsulates the use of local as well as of shared priority queue algorithms.
3. **The local or shared priority queue(s)**. They are dependent on one side on the choice of the algorithm (stack, search tree, etc), and on the other side of the parallelization selected (concurrent access, local access, particular operations, etc).
4. **The global lower/upper bound**. The strategy of its management depends on the architecture of the machine and on the choice of parallelization.
5. **The statistics and measures linked to the behavior of the algorithm**.

Only the functions of the first component can not be provided by the library because they depend on the problem to be treated. They must be developed by the user and can be implemented without worrying about the problems of parallelization. Whereas the functions involved in the other components (2., 3., 4. and 5.) are independent of the

architecture of the machine and the choice of the user. For a given problem, they simply need to know the structure of the nodes of the search tree (the `BobNode` type). They could be implemented in the BOB library. Their use is done via calls to predefined functions. So, the different types of management of parallelism are made transparent for the user.

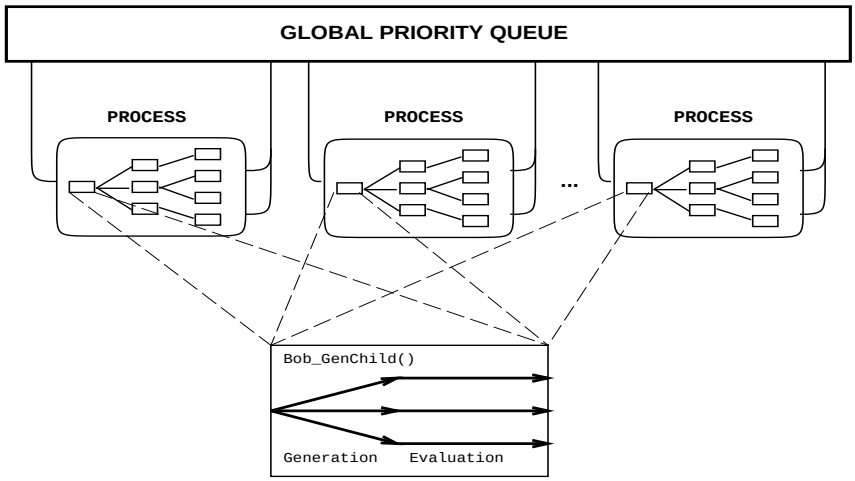

Figure 1: A parallel Branch-and-Bound scheme.

Figure 1 shows the interaction between the global priority queue, the processes and the recursive calls of the generation/evaluation (branching/bounding) procedure.

Another component managed by the library is the notion of priority. It could be either the simple evaluation, either the evaluation and the depth of the node, either the potentiality [8]. This list is not exhaustive. The developer's guide contains all the explanations to define a new notion of priority. Once a new notion developed, it could be used in other applications.

For the GL/UB, the library is conceived with the management of minimization and maximization in mind. The user indicates in his application the bound he seeks to achieve (minimization or maximization).

One of the goals of the library is to provide different types of GPQ and of GL/UB in order to test their efficiency. Subsequently, the types that are best adapted to a particular machine and a particular type of problems can be isolated. Thus, the BOB library can be considered as a collection of libraries. Each of these libraries takes into account the choice of the user and the architecture of the machine. When compiling his application, the user indicates the model of his machine as well as the options precising the type of parallelization. We have chosen this method to avoid an overhead of execution due to the choice of parallelization and to avoid an overhead in the size of the executable due to unused functions. The library takes also into account the handling of processes from the UNIX point of view as well as from a light process point of view (threads).

We have defined the parameters of an executable program in BOB. They permit to specify the number of processes, the "granularity" of computation, and the options for displaying the results and statistics. (cf. sections 5 and 6.5).

2.3.1 The global priority queue

In this section we detail the choices concerning the classification of global priority queues proposed in BOB.

On the first level, we classify the architectures of machines according to three families : serial, parallel with shared memory and parallel with distributed memory.

In serial, the notion of global priority queue makes use but of one particular algorithm of local priority queue (heap, search tree, etc).

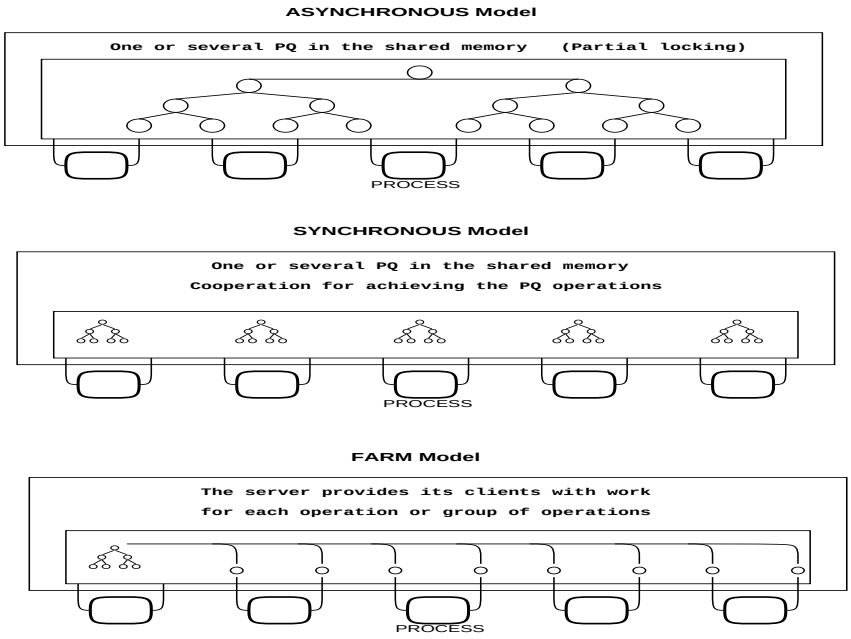

Figure 2: Classification of GPQ on a shared memory machine.

In the literature, there has been proposed many models to handle global priority queues in parallel. We will detail three models for each parallel architecture.

Asynchronous : Processes execute themselves the operations on local or shared priority queues. They do not cooperate to realize one or many operations. The GPQ can be seen as a priority queue with concurrent access (shared memory machines) or as a set of local priority queues with a queue per process and a load balancing strategy (distributed memory machines).

Synchronous : Processes cooperate to realize a parallel operation (simultaneous insertions or deletions). The cooperation can exist in each operation. Particularly, in the delete operation, processes can handle the deletion of nodes with the highest priority in the GPQ.

Farm (of processes) : In this model, one process (server) is specialized in the management of the GPQ. The "client" processes do the search and communicate with this server to obtain workload (nodes).

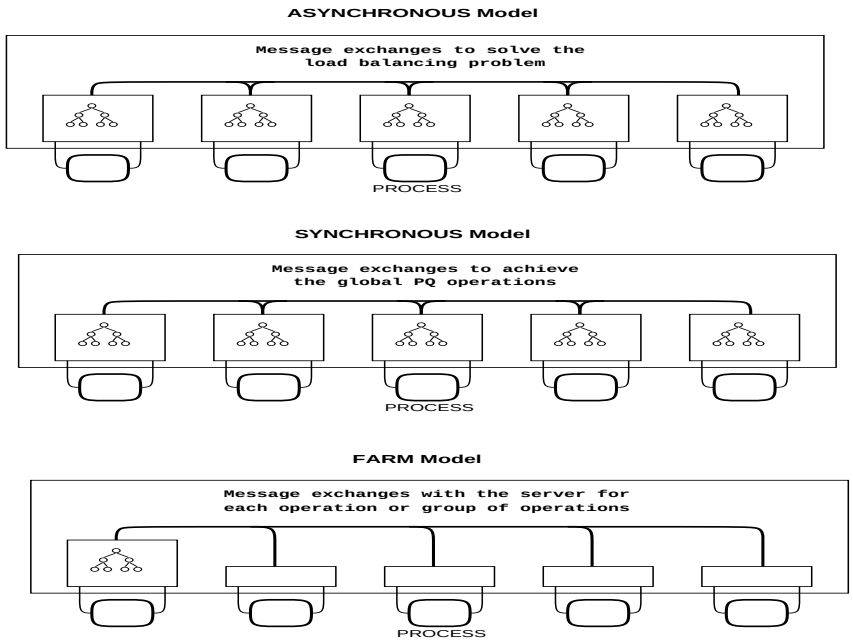

Figure 3: Classification of GPQ on a distributed memory machine.

Figures 2 and 3 represent graphically the classification of GPQ's on the two architectures.

These parallelizations are different of each other not only in function of the architecture of the target machine but also in function of the semantics attached to the delete operation. In certain cases, the deleted node is not the best-valued node in the whole GPQ. A relaxation of the notion of "best-valued" is often preferred on a very restrictive time of the delete operation. The user must make his choice by taking into account this compromise.

2.3.2 Local or shared priority queue

All of GPQ models need priority queues which are either local (sequential) or shared (capable of handling many partial locking protocols). BOB proposes different types of local/shared priority queues :

Heap D-Heap, Skew-Heap, Pairing-Heap, Leftist-Heap,

Search trees Splay-tree (Semi-Tree, Simple-Tree, Single-Tree, Single-Simple-tree),

Funnels Funnel-Tree, Funnel-Table.

This non-exhaustive list indicates the available data-structures. However, anyone can develop his own data-structures which can be totally new ones. In the case of shared memory machines, the priority queue(s) must handle concurrent access. We have defined three mutual exclusion protocols for this case. In the first protocol, a process reserves an exclusive access to the priority queue for each operation. The two others are partial

locking protocols [2, 9, 10, 23, 31]. Some priority queue algorithms might not handle certain protocols. The user must choose the type of priority queue and the locking protocol he wants to use.

3 Conceptual structure of the library

Figure 4 presents the conceptual structure of BOB. The user must provide four functions describing his method of resolution. The definition of these functions is detailed in section 4.1. All other functions necessary for the Branch-and-Bound algorithm are handled by the kernel of the library. If users do not seek to provide BOB with new priority queues, load balancing strategies or parallelization methods, they do not need to know the functions of the kernel. A developer's guide of BOB is provided with the library.

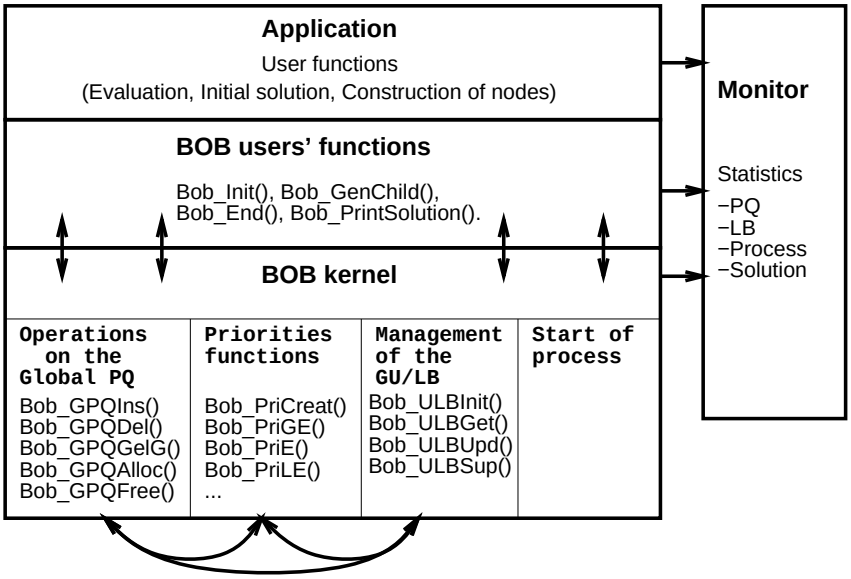

Figure 4: Functional structure of BOB.

The main goal of the monitor is to provide statistics on the different objects manipulated by an application (priority queues, workload of processes, number of load balancing operations, ...). These informations make it possible for the user to analyze the different parameters of his application (efficiency of the evaluation function, the branching criterion, ...) and to improve the choice of his parameters.

4 Objects in the application

The objects manipulated by the application are of two origins. The ones are defined by the user for his application, the others are defined in the BOB library.

4.1 Types and functions defined by the user

The user can freely choose the definition of these objects for his application; Nevertheless, he should take into account the names of these types and functions.

The types

`BobNode` This type must be a structure which defines a node of the search tree depending on the application. The user is supposed to fill this structure with the variables which define an element of the search space proper to his application.

`BobSolution` This type must represent the solution to the problem.

The functions

Initialization : `void Bob_Init(BobArgc, BobArgv)`

This function regroups all the initializations that a Branch-and-Bound would need : the initial lower bound, the initial upper bound, the root of the search tree, the size of the problem and the best initial known solution.

The parameters of this function represent the parameters passed to the C `main()` function. (classical form of `argc` and `argv`).

The end : `void Bob_End()`

This function offers to the user the possibility to execute instructions specific to his application at the end of the execution of the program.

Display of the solution : `void Bob_PrintSolution(s)`

In general we write a Branch-and-Bound algorithm in order to find a solution to an optimization problem. Hence, we have a function to display the best current known solution (`BobSolution *s`). This function will be executed each time a request of displaying statistics is invoked.

Generation of subproblems (child nodes) :
`void Bob_GenChild(BobCurrNode, BobDepth, BobExpCt)`

In this function, the user writes the generation via the branching criterion of his choice. He also writes the evaluation of child sub-problems of the *current* subproblem `BobCurrNode` of type `BobNode`. It makes part of the parameters list. The parameter `BobDepth` is the current exploration depth. The parameter `BobExpCt` is a pointer to a structure of type `BobTExpCt` giving among other informations the depth of the exploration (cf. paragraph 4.2).

4.2 The types, variables and functions provided by the library

These predefined objects in the library can be used by the user in order to write his application. He does not need to develop them.

The types

`BobTPri` This type defines the priority of a node. The user can choose between different notions of priority for the best first search. The priority of a node is either its evaluation or its evaluation and its depth in the search tree. The notion of "potentiality" is expected among these choices [8].

`BobTid` This type represents the identity of a process. This type will be different according to the type of the target machine. The variable associated to this type is `BobId`.

`BobTExpCt` This type regroups the useful informations for the control of the granularity of exploration.

The variables Many global variables, that the user must initialize in the function `Bob_Init()` are offered by the library :

`BobRoot` : A pointer to a node of type BobNode. It represents the root of the search tree. The library accomplishes the task of inserting this node.

`BobPbSize` : The initial size of the problem (optional).

The functions The library proposes functions to handle the upper bound and the insertion in the data structure(s) containing nodes to be explored.

1. Functions for the upper/lower bound :

   `void Bob_ULBInit(BobULB,s)` : initialization of the global upper bound. The value of the variable `BobULB` is given by the user. He must provide the variable `s` containing the initial solution (*BobSolution **).

   `int Bob_ULBGet()` : returns the value of the global upper bound.

   `int Bob_ULBSup(BobEval)` : tests if `BobEval` is lower/higher than the global lower/-upper bound BobULB.
   It returns
   `1` if `BobULB` > `BobEval`,
   `0` otherwise.

   `int Bob_ULBUpd(BobNewULB, BobNewSol)` : this function tries to update the global solution.
   It returns
   `1` if `BobULB` has been modified,
   `0` otherwise.
   This function deletes the obsolete nodes from the global structure, that is nodes having their evaluations strictly higher/lower than `BobULB`.

2. Function to control the granularity : `int Bob_ExpCtrl(BobDepth, BobExpCt)`

   Tests if the maximal depth of the exploration, allowed in the structure pointed to by `BobExpCt`, has been attained by the process calling this function.

   It returns
   `1` if `BobDepth` is equal to the maximal depth of the exploration, `0` otherwise.

   In the case of parallel machines, this function tests also if the number of nodes in the global data structure is sufficient in order to give workload to all of the processes.

3. Functions for the global data structure :

   `void Bob_GPQIns(BobCurrNode)` : inserts the current node in the global priority queue.

   `BobNode *Bob_GPQDel()` : deletes the node of type `BobNode` with the highest priority in the global priority queue or in a part of the global priority queue (the case for certain parallelizations).

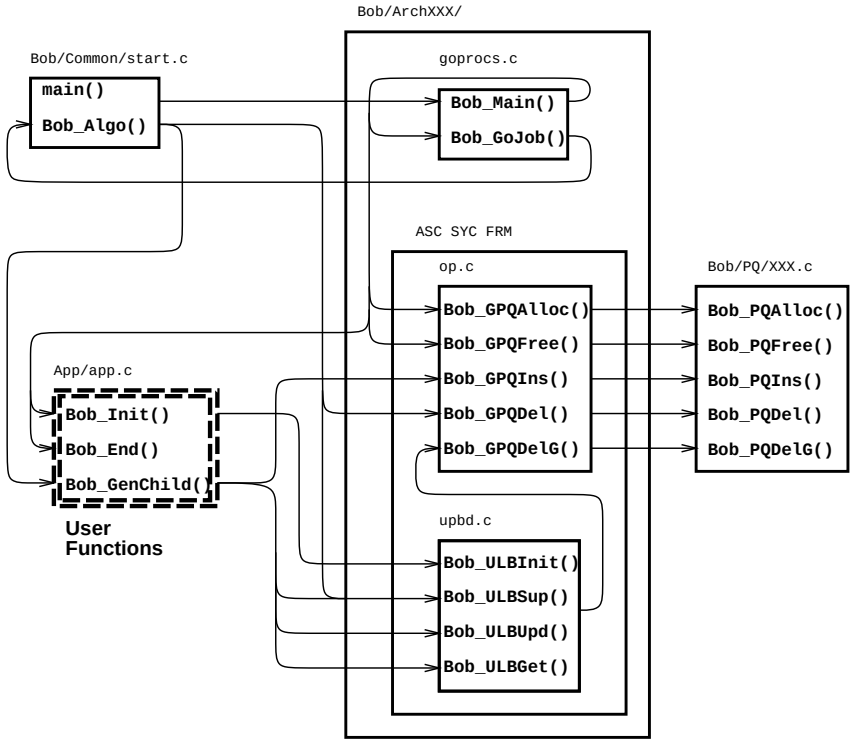

Figure 5: Relations of BOB functions.

Figure 5 gives another view of functions in the library. The arrows translate the relations calling-called between functions.

5 Monitor

The library proposes a "real-time" monitor giving statistics on the behavior of the application. Displaying the statistics as well as the current best solution is done by calling the function `Bob_PrintStat()`. Statistics are divided into two groups : Statistics on the activity of one or many processes, and statistics on the state of the local/shared priority queue. At the end of the execution, all of the statistics as well as the execution times will be displayed automatically.

5.1 Activities of processes

For each of the architectures, statistics give informations on :

- the number of generated (evaluated) nodes,
- the number of non inserted nodes (with evaluation higher/lower than the bound GU/LB),
- the number of inserted nodes,
- the number of deleted nodes,
- the number of deleted nodes by `DeleteGreater` (for which the bound GU/LB becomes lower/higher than their evaluation),
- the priority of the last deleted node.

But in the case of parallelism, additional statistics depending on the selected parallelization are provided. For example, in the case of asynchronous GPQs on a distributed architecture (local priority queues with load balancing), the statistics provide the user with the number of nodes sent, received, deleted and treated before finding a solution.

5.2 State of local or shared priority queues

Statistics on local or shared priority queues give informations on :

- the number of nodes at a given moment.
- the maximal number of nodes.
- the priority of the last deleted node.

Like for other objects, in the case of a parallel execution, additional informations concerning the parallelization are provided.

6 Architecture of the library

6.1 Organization of directories

The source code of the library is in the directory `Bob`. At the same level, we find directories of different applications. (Ex : `Qap`, `TspLittle`, `TspDicho`, `TspPoly`, `Vcp`, etc).

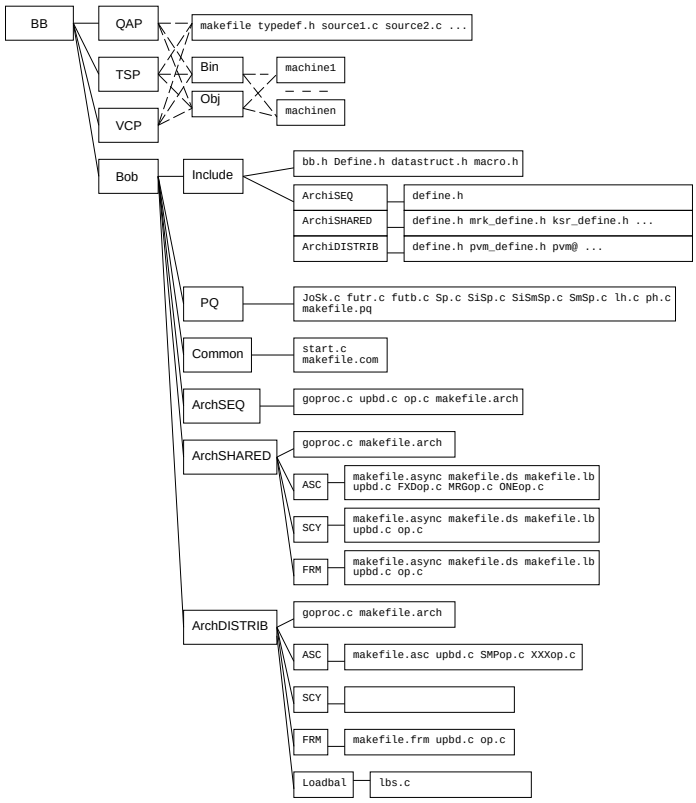

Figure 6: Directories of the library.

6.2 The `Bob` directory

In `Bob`, we find six directories and one makefile (`makefile.bb`).

`Bob/Include` contains the header files (`.h`) common to all of the Branch-and-Bound.

`Bob/Common` contains the file `start.c`, this file contains the following functions :

- function `main()`,
- function of study of parameters.

It also contains the makefile specific to this file.

`Bob/PQ` contains the source files of priority queues.

`Bob/ArchSEQ` source for a serial run-time.

`Bob/ArchSHARED` source for a parallel run-time with shared memory.

`Bob/ArchDISTRIB` source for a parallel execution with distributed memory.

`makefile.bb` This makefile directs the compilation towards the convenient architecture.

6.3 Directories of the architectures : `ArchSEQ`, `ArchSHARED` and `ArchDISTRIB`

These directories contain the source code handling the upper bound (`upbd.c`), the global data structure (`op.c`), as well as the source code for the parallel management of processes (when it could have place) and for the statistics (`goproc.c`).

Serial : `Bob/ArchSEQ` This directory contains the three files listed above as well as the makefile `makefile.arch`.

Parallel : `Bob/ArchSHARED`, `Bob/ArchDISTRIB` Each of these two parallel directories contains the `goproc.c` file, a makefile called `makefile.arch` and three directories corresponding to the models of parallelization :

`ASC` Directory containing the source code for handling the priority queue and the GU/LB in asynchronous mode.

`SYC` Directory containing the source code for handling the priority queue and the GU/LB in synchronous mode.

`FRM` Directory containing the source code for handling the priority queue and the GU/LB in the FARM mode where the priority queue(s) are handled by one server.

Each of these three directories contains one or many source codes for algorithms of management of GPQs (`op.c`) and of GU/LB (`upbd.c`).

Moreover, the directory `Loadbal` in `Bob/ArchDISTRIB`, concerns a distributed implementation. This directory contains two functions associated to the load balancing; One function controls the workload of a processes (`Bob_LBalCtrl()`), and the second one regulates this workload (`Bob_LBal()`) for a better exploitation of the distributed machine (cf. Developer's guide for more details).

6.4 Portability of the library

BOB must be portable. The serial part of BOB does not cause any problem since it uses the standard environment of UNIX. However, this is not the case for the parallel parts of Bob. BOB necessitates, on a shared memory machine, the use use of mutual exclusion primitives. BOB uses a set of C macros, allowing the generation of these primitives. The correspondence between this set of macros and the functions present on the parallel machine is done via a configuration file. Actually, the library has been ported on KSR1 and on Sequent Balance.

The distributed part of BOB works on PVM which seems to be a standard for distributed memory machines and for networks of workstations. But in order not to limit BOB to this library of exchange of messages, BOB can work on a set of macros, which can be corresponding PVM calls or functions implementing PVM calls on top of a particular library. This is used for the primitives of sending/receiving messages. In what concerns launching processes, the code can be written in the `goproc.c` file without being forced to simulate the PVM calls for handling processes. Let us note that BOB has been

ported on the distributed machine PARAGON, with a communication library proper to the INTEL parallel machines. The developer's guide contains a complete explanation on the procedure to follow in order to port BOB on a particular parallel machine.

6.5 Compilation and execution

6.5.1 Compilation

The compilation of an application is done via the command `make`. Since BOB is composed of a set of distinct and/or dependent sub-libraries, the process of compilation in BOB is a sequence of "recursive" calls to the command `make`. Each call handles one component of the library. The command `make` invoked by the user in the application directory, will call another `make` command in the selected or given architecture directory in order to compile the GPQ and the GU/LB requested. Then the file `Common/start.c` will be compiled. Finally, this last one will call another `make` command on the file `PQ/makefile.pq` which will generate the executable.

By default, all the possible executables of the target machine will be generated. We do not recommend this method since the number of different executables will be enormous given the number of different possible combinations. It is possible to define a set of executables to be generated via the environment variables passed to the command `make`. We can not detail here all the environment variables corresponding to the different possible combinations. Please refer to the documentation provided with the library in order to know the choices provided. Hereafter, we will describe the principle variables :

`PRTP` notion of priority `EVAL` (evaluation), `EVDP` (evaluation+depth), `POTE` (potentiality).

`PQTP` local or shared priority queues algorithm `SKEW`, `FUNNEL`, `SPLAY`, `HEAP`, `OTHERS`. If the queue is shared, the locking protocol `PLTP` could be `EXC`, `LCK`, `MRK`.

`ARTP` type of the architecture : `SEQ`, `SHARED` or `DISTRIB`. For a parallel data structure :

- `MDTP` type of the method used `ASC`, `SYC`, `FRM`.
- `DSTP` representation of the structure.
- `LBTP` load balancing strategy

The command executed in the directory `Qap` on KSR1:

```
make PRTP=EVAL ARTP=SHARED MDTP=ASC DSTP=FXD LBTP=2 PLTP=MRK PQTP=SKEW
```

will generate the executable corresponding to a parallel Branch-and-Bound resolving the QAP on KSR1 with an asynchronous data structure of type `FXD` with the load balancing strategy 2; The priority queue used will be the Skew-Heap, with the partial locking protocol which uses the marking technique. The executable will be in the directory `Qap/Bin/ksr1` and will be called `EVALASCFXD5MRKskh`.

Even though this type of names for executables is rather complicated, it has the advantage of differentiating the executables according to the choice of the user.

6.5.2 The executable

We have defined the parameters that an application BOB can take.
`exec [ -p NbProc[:NbDS] ] [ -v Stat ] [ -d MaxDepth[:TypeDepth] ] ...`

`-p NbProcs[:NbDS]` defines the number of processes and the number of data structures (reserved to the parallelism).

`-v Stat` defines the level of statistics.

`-d MaxDepth[:TypeDepth]` defines the granularity of computation (in number of levels) and the type of computation (each process can have a different granularity calculated according to a function).

`...` parameters reserved to the application.

7 The development of an application

We present a kernel of a Branch-and-Bound program using the library BOB. The files cited must exist in the application directory. This example is taken and detailed in the user's guide provided with the library.

7.1 The header file `typedef.h`

All the definitions of macros and structures must be regrouped in a header file in order to be accessible by the different source files of the library. We call it typedef.h but another name could be chosen as well.

If the optimization problem consists of maximizing a cost function, the file `typedef.h` must contain the following line :

```
#define ORTP MAXIMISATION
```

The default configuration of BOB is for the minimization.

The type `BobNode` must be defined in the header file. The two mandatory fields are `Pri` of type `BobTPri` and `BobNdInfo` of type `BobTNdInfo`. The names of these fields can not be altered. Otherwise, BOB does not recognize them.

BOB needs also a type called `BobSolution` which describes the solution of the optimization problem being handled.

```
/*--- BobNode structure for the simulation of a Branch and Bound ---*/
typedef struct {
  BobTNdInfo BobNdInfo; /* Node informations (read only) */
  int Depth;            /* Node depth */
  ...                   /* Informations relative to the application */
  ...
  BobTPri Pri;          /* Node priority */
} BobNode;

typedef struct {
  int Sol[100];
} BobSolution;
```

7.2 The source file(s)

In this application, all of the functions have been grouped into one source file that we call `app.c`.

```
/*------------------- Include the header files ------------------*/
#include <stdio.h>
#include <math.h>
#include "../Bob/Include/bb.h"


/*-------------------------------------------------------------*/
void Bob_GenChild(an,Depth,ExpCt)
BobNode *an;
int Depth;
BobTExpCt *ExpCt;
{
  BobNode *n;
  int Eval,dp;

  dp = an->Depth+1;                 /* Sons depth */
  for (i=0; i<NBSONS ; i++) {       /* Generation loop for the NBSONS sons */
    n = (BobNode *)Bob_NodeAlloc(0); /* Memory allocation for the sons */
    if ( n==NULL ) {fprintf(stderr,"Memory error\n");return;}
    Branch(i,an,n);
    Eval = Evaluate(n);
    Bob_PRICREAT(n->Pri,Eval,0,dp);  /* Creation of the son's priority  */
    Bob_STEXP();                    /* Number of the evaluated nodes */
    if ( Bob_ULBSup(n->Pri) ) {     /* Compare to the GU/LB */
       if ( Bob_ExpCtrl(prof+1,ExpCt) ) {/* Compare to the maximal depth */
          Bob_GenChild(n,prof+1,ExpCt);  /* Depth-first traversal */
          Bob_NodeFree(n);          /* Free the son node */
       } else
          Bob_GPQIns(n);            /* Insert into the priority queue */
    } else {
       Bob_STNIS();                 /* Number of the unexplored nodes */
       Bob_NodeFree(n);             /* Free the son node */
    }
  }
}


/*-------------------------------------------------------------*/
void Bob_Init(n,v)
int n;
char **v;
{
  int bs;
  int Eval;
  BobSolution Sol;

  if ( n!=NbParameters ) {
     fprintf(stderr,"Parameter error : ... ... ...\n");
     exit(1);
  }
  BobPbSize = ProblemSize();        /* Problem's size */
  bs = InitialSolution(&Sol);       /* Initialization of the initial solution */
  Bob_ULBInit(bs,&Sol);               /* and the initial GU/LB */
  BobRoot=(BobNode *)Bob_NodeAlloc(0); /* Allocation of the root node */
  if ( BobRoot==NULL ) {fprintf(stderr,"Memory error\n");return;}
  Eval = Evaluate(BobRoot);
  Bob_PRICREAT(BobRoot->Pri,Eval,0,0); /* Creation of the son's priority */
```

```
}

/*-------------------------------------------------------------*/
void Bob_End() {
  printf("The End !\n");
}

/*-------------------------------------------------------------*/
void Bob_PrintSolution(s)
BobSolution *s;
{
int i;
  for (i=0;i<BobPbSize ; i++ ) {
     printf("%2d ",s->Sol[i]);
  }
  printf("\n");
}
```

7.3 The file `makefile`

The `makefile` in the application directory is the first one called in order to construct the executable. The variables to be defined are :

`OBJAPP` containing all the objects (`.o`) of the application.

`INC` containing the header file of your application.

For each one of the source files, the commands concerning its compilation must be written on three different lines. In our case, the file `makefile` is:

```
LDIR  = ../Bob
OBJ   = Obj/$(HOSTTYPE)/$(PRTP)
OBJAPP= $(OBJ)app.o
DEBUG  =
APPDEBUG = $(DEBUG)
INC = ../../App/typedef.h
DSFLAG = $(ARTP) ARTP=$(ARTP) PRTP=$(PRTP) OBJAPP="$(OBJAPP)" INC=$(INC) DEBUG=$(DEBUG)

all : $(OBJAPP)
        make -f $(LDIR)/makefile.bb  $(DSFLAG)

$(OBJ)app.o : app.c typedef.h
        cc -c app.c  $(APPDEBUG)  -DAPPINC=\"$(INC)\" -DPRTP=$(PRTP)
        mv app.o $(OBJ)app.o
```

8 Conclusion and perspectives

In this report we have presented a library to help the user willing to develop applications using the Branch-and-Bound method. We think that BOB can be used by the vast majority of researchers in the Combinatorial Optimization and in the Parallel Algorithmic fields. But there is still a lot of work to be done from provided parallelizations and resolved problems point of view. However, what seems to be more urgent for BOB is a graphics interface. Statistics are presented for the moment in a textual format (ASCII) and only upon the request of the user. A graphics interface representing the evolution

of the activity of processes and the state of different data structures, under the form of curves, would be interesting.

The compilation of the application remains a difficult task. We are planning to write a graphical interfacing tool which permits to choose easily between the different compilation parameters.

We plan too to implement different strategies to control the granularity. At the moment the granularity of exploration (see section 2), is fixed for an execution. We plan to implement strategies where the granularity is updated while the execution depending on the depth of the current node, the load of the different process and the gap between the evaluation of the current node and the value of best known solution.

With little learning, BOB gives the possibility to write Branch-and-Bound programs by concentrating only on the "interesting" parts of the resolution method. With very little effort, the program can be executed on a shared or distributed memory machine or on a simple network of workstations. This can help resolve more quickly larger problems. If the library BOB is used by a large number of researchers in the Combinatorial Optimization domain, it can become a mean to test and compare the parallelizations developed on a great number of real problems. While for the time being, parallelizations are being developed on simple problems from the literature. Thus the BOB library can facilitate the exchange between the two fields of Combinatorial Optimization and Parallel Algorithmic, an exchange that we hope beneficial.

At this point, the sequential part of the BOB Library and the QAP, TSP, VCP applications could be reach at *http://www.prism.uvsq.fr/english/parallelcr/bob_us.html* The bob team could also be reach by email at *bobteam@prism.uvsq.fr*.

References

[1] Benaïchouche (M.). – Méthodes de recherche arborescentes parallèles : Branch & bound distribué. *In : 7ème Journnées Internationales des Sciences Informatiques, JISI'94, Tunis Tunisie*, pp. 266–282. – 1994.

[2] Calhoun (J.) et Ford (R.). – Concurrency control mechanisms and the serializability of concurrent tree algorithms. *In : of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* – Waterloo Ontario, Avr. 1984. Debut de la theorie sur la serializability.

[3] Corrêa (R.) et Ferreira (A.). – *Parallel best-first branch-and-bound in discrete optimization : a framework.* – Rapport technique n° 95–03, DIMACS, Rutgers University, Mars 1995.

[4] Cung (V.-D.). – *Contribution à l'Algorithmique Non Numérique Parallèle : Exploration d'Espaces de Recherche.* – 4, Place Jussieu, 75252 Paris Cedex 05, FRANCE, Thèse de PhD, Université Pierre et Marie Curie – Paris VI, Avr. 1994. In French.

[5] Cung (V.-D.), Dowaji (S.), Le Cun (B.), Mautor (T.) et Roucairol (C.). – Concurrent data structures and load balancing strategies for parallel branch-and-bound/a* algorithms. *In : DIMACS Challenge 3 Workshop.* – Oct. 1994.

[6] Cung (V.-D.) et Le Cun (B.). – An efficient implementation of parallel a*. *In : Parallel and Distributed Computing : Theory and Practice. Proceedings of the First Canada-France Conference on Parallel Computing*, éd. par Cosnard (M.), Ferreira (A.) et Peters (J.), pp. 153–168. – Mai 1994.

[7] Dowaji (S.). – *Équilibrage de Charge (EC) en Milieu Distribué.* – Rapport technique n° 92.101, Labo. MASI - Université PARIS VI - 4, Place Jussieu - 75252 PARIS CEDEX 05, Laboratoire MASI, décembre 1992.

[8] Dowaji (S.) et Roucairol (C.). – Influence of priority of tasks on load balancing strategies for distributed branch-and-bound algorithms. *In : 9th International Parallel Processing Symposium (IPPS'95) - Workshop on Solving Irregular Problems on Distributed Memory Machines*, pp. 83–90. – Santa Barbara (USA), April 25-28 1994.

[9] Ellis (C.). – Concurrent search and insertion in 2-3 trees. *Acta Informatica*, vol. 14, 1980, pp. 63–86.

[10] Ellis (C.). – Concurrent search and insertion in avl trees. *IEEE Trans. on Cumputers*, vol. C-29, n° 9, Sept. 1981, pp. 811–817.

[11] Finkel (R.) et Manber (U.). – Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, vol. 9, n° 2, Avr. 1987, pp. 235–256.

[12] Grama (A. Y.) et Kumar (V.). – A survey of parallel search algorithms for discrete optimization problems. – Personnal communication, 1993.

[13] Hajian (M.), Hai (I.) et Mitra (G.). – *A Distributed Processing Algorithm For Solving Integer Programs Using a Cluster Of Workstations.* – Rapport technique n° TR/14/94, Department of Mathematics and Statistics, 1994.

[14] Hiromoto (U.), Masafumi (Y.), Masaharu (I.) et Toshihide (I.). – Parallel searches of game trees. *Systems and Computers in Japan*, vol. 18, n° 8, 1987, pp. 97–109.

[15] Ibaraki (T.). – Game solving procedure h* is unsurpassed. *In : Discrete Algorithms and Complexity*, éd. par Johnson (D. S.) et al., pp. 185–200. – 1987.

[16] Kalé (L.) et Saletore (V. A.). – Parallel state-space search for a first solution with consistent linear speedups. *International Journal of Parallel Programming*, vol. 19, n° 4, 1990, pp. 251–293.

[17] Kanal (L. N.) et Kumar (V.). – Parallel implementation of a structural analysis algorithm. *IEEE Pattern Recognition and Image Processing*, 1981, pp. 452–458.

[18] Kumar (V.), Ananth (G. Y.) et Rao (V. N.). – *Scalable Load Balancing Techniques for Parallel Computers.* – TR n° 91-55, University of Minnesota, Nov. 1991.

[19] Kumar (V.) et Kanal (L. N.). – A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, vol. 21, n° 1, 2, 1983, pp. 179–198. – Reprinted in the book Search and Heuristics edited by Judea Pearl.

[20] Kumar (V.) et Kanal (L. N.). – Parallel branch-and-bound formulations for and/or tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, n° 6, Nov. 1984, pp. 768–778.

[21] Kumar (V.), Ramesh (K.) et Rao (V. N.). – Parallel best-first search of state-space graphs : A summary of results. *The AAAI Conference*, 1987, pp. 122–127.

[22] Lawler (E.) et Wood (D.). – Branch-and-bound methods : a survey. *Operations Research*, vol. 14, 1966, pp. 699–719.

[23] Le Cun (B.), Mans (B.) et Roucairol (C.). – *Opérations concurrentes et files de priorité.* – RR n° 1548, INRIA-Rocquencourt, 1991.

[24] Li (G.) et Wah (B. W.). – Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transaction on Computers*, vol. 6, n° C-35, Juin 1986, pp. 568–573.

[25] Li (T.). – Parallel imprecise iterative deepening for combinatorial optimization. *International Journal of High Speed Computing*, vol. 3, n° 1, 1991, pp. 63–76.

[26] Mans (B.). – *Contribution à l'Algorithmique Non Numérique Parallèle : Parallélisation de Méthodes de Recherche Arborescente.* – Thèse de doctorat, Université Paris 6, Juin 1992.

[27] Mans (B.) et Roucairol (C.). – *Concurrency in priority queues for branch and bound algorithms.* – RR n° 1311, INRIA-Rocquencourt, Oct. 1990.

[28] Nilsson (N. J.). – *Principles of Artificial Intelligence.* – Tioga Publishing Co., 1980.

[29] Pearl (J.). – *Heuristics.* – Addison-Wesley, 1984.

[30] Powley (C.), Ferguson (C.) et Korf (R. E.). – Parallel heuristic search : Two approaches. *In : Parallel Algorithms for Machine Intelligence and Vision*, pp. 42–65. – Kumar and Kanal, 1990.

[31] Rao (V.) et Kumar (V.). – Concurrent insertions and deletions in a priority queue. *IEEE proceedings of International Conference on Parallele Processing*, 1988, pp. 207–211.

[32] Rao (V. N.), Kumar (V.) et Ramesh (K.). – *Parallel Heuristic Search on Shared Memory Multiprocessors : Preliminary Results.* – Rapport technique n° AI85-45, Artificial Intelligence Laboratory, The University of Texas at Austin, Juin 1987.

[33] Roucairol (C.). – *Recherche arborescente en parallèle.* – RR n° M.A.S.I. 90.4, Institut Blaise Pascal - Paris VI, 1990. In French.

[34] Talbi (E.). – *Allocation dynamique de processus dans les systèmes distribués et parallèles : Etat de l'art.* – Rapport technique n° 162, Laboratoire d'Informatique Fondamentale de Lille, 1995.

[35] Tschöke (S.). – A portable parallel branch-and-bound library. *In: DIMACS-Challenge 3 Workshop.* – Oct. 1994.

[36] Wah (B. W.), jie Li (G.) et Yu (C. F.). – Multiprocessing of combinatorial search problems. *IEEE Computer*, Juin 1985, pp. 93–108.

Laboratoire PRiSM, Université de Versailles - Saint Quentin en Yvelines,
45 avenue des État-Unis, 78035 Versailles Cedex, FRANCE.