Bob++ : a Framework for Exact Combinatorial Optimization Methods on Parallel Machines

François Galea and Bertrand Le Cun
PR*i*SM Laboratory, University of Versailles
45 avenue des Etats-Unis, 78035, Versailles, France
Francois.Galea@prism.uvsq.fr, Bertrand.Lecun@prism.uvsq.fr

Abstract— **The aim of this article is to propose the object-oriented design of the Bob++ framework. Bob++ is a framework for implementing solvers for combinatorial optimization problems on parallel and sequential machines. Several similar frameworks have been proposed in the last decade but each of them only focuses in one method, said Branch-and-Bound, Divide-and-Conquer, etc.. and proposes also one parallelization, which is very difficult to extend. We propose a software design where: first, several exact combinatorial optimization methods are made available to the user to solve a problem, and second, an interface to facilitate the implementation of a parallelization is also provided. Parallelizations may use POSIX threads as well as MPI, or more specialized libraries such as Athapascan/Kaapi.**

Keywords— **Combinatorial Optimization, Search algorithms, Branch-and-Bound, Parallelism, Cluster, Grid Computing**

I. Introduction

Large scale decision and optimization problems belong to the class of the best applications for parallel machines and also for computational grids. The problems are in the NP-Hard complexity class and may require an exponential computational time in the worst case. It is natural to consider the parallelization of the search process when a solution of a large-scale problem is out of reach when using a single-processor computer.

The tremendous attention that the parallelization of these methods as Branch-and-Bound has received in the literature gives some indication of its importance in many research areas.

But as in many domains, several software frameworks have been proposed, establishing the interface between the users and the parallel machine. These tools include Bob++ [21], BCP [13], PICO [3], ALPS [20], [12], [18], Bob [9], PUBB [15], [14], PPBB [19] ....

It is possible to classify these different existing frameworks according to two major criteria:
1. The **node search algorithm** involved in the search process. These algorithms include Branch-and-Bound (B&B), Divide-and-Conquer (D&C), A* and Dynamic Programming (DP).
2. The **programming environment** they use to implement the parallelization. Some of the available programming environments are POSIX threads, MPI, PVM, and Athapascan/Kaapi.

Many of the available parallel search algorithm frameworks are specialized at the same time in the algorithm they implement, and for a specific programming environment. For example, BCP [13] is an implementation of the Branch-and-Price-and-Cut algorithm, which runs on the MPI programming environment. PICO [3] is a Mixed-integer solver, which implements B&B, and also runs on MPI.

Some other projects tend to diversify some aspects of the solver framework. SYMPHONY [17], for example, solves mixed-integer programming (MIP) problems using PVM for distributed memory machines or OpenMP for shared memory machines. ALPS [20], [12], [18], which in some way is a successor of SYMPHONY [17] and BCP [13], generalizes the node search to any tree search, which of course enables B&B search, among others. Though, the only available programming environment for ALPS [20], [18] is MPI. In a similar manner, PEBBL [4] integrates the B&B search from PICO [3], allowing the implementation of a larger variety of solvers than MIP solvers.

The old version Bob [9], [11] focus on parallel Branch and Bound. Others methods like A* have been added but with awful hacks. What Bob++ proposes is to provide different search algorithm classes, while being able to use different possible parallelization methods. The goal is to propose a single framework for most classes of combinatorial optimization problems, which can be solved on as many different programming environments as possible. Figure 1 shows how Bob++ interfaces between high-level applications (QAP, TSP, ...) and different possible parallel programming environments. However, Bob++ is still under developement.

Bob++ has been developed in C++ language, and proposes a C++ API, composed of basic classes which are extendable by the user.

Most real-life tests have been done using the Branch-and-Bound search algorithm, showing the robustness of the Branch-and-Bound application interface. Most of other developed applications are just validation tests for the design of some of the node search algorithms, such as a simple N-Queens problem solver which uses Divide-and-Conquer.

Most of recent work has been focused on Branch-and-Bound. Dynamic Programming and A* are currently unavailable, due to changes we made in the Bob++ structure when developing Branch-and-Bound and Divide-and-Conquer, even though these changes have been done with the addition of Dynamic Programming and A* in mind. This is why the only application interface we will talk about in the following sections of this article is Branch-and-Bound.

The next section deals with the Bob++ application interface. The parallel interface is presented in the sec-

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

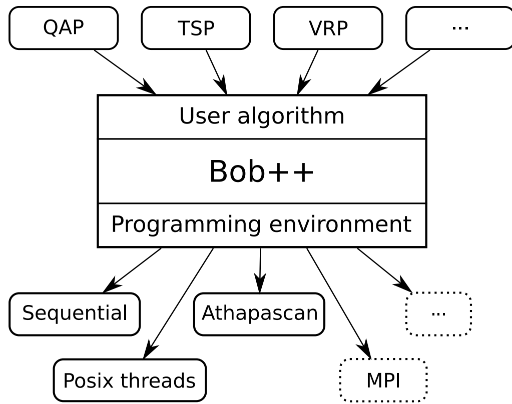

Fig. 1. The structure of Bob++ applications

tion III. The section IV shows two applications. Concluding remarks and future work are presented in the section V.

II. Application Interface

The idea behind the Bob++ application interface is to provide the programmer with an easy interface for programming parallel node search algorithms, based on the following classical methods : A*, Divide-and-Conquer, Dynamic programming and Branch-and-bound. At this time, the only implemented methods are Divide-and-Conquer and Branch-and-Bound.

In Bob++, each of the mentioned methods is implemented by four classes. The base classes for each of them are the following:

- The `Instance` class is used for the storage of all the global data of the application. This data is initialized at the beginning of the resolution. It is not allowed to modify its contents during the execution of the resolution.
- The `Node` class enables the storage of the data and contains the methods associated to a node in the search space.
- The `GenChild` class contains the method used to generate the child nodes of a given node.
- The `Algo` class contains the execution code of the main loop of the resolution.

Each of these base classes are derived into specific classes which are specialized for the resolution of the different possible search algorithms.

The specialized `Instance`, `Node` and `GenChild` classes are called "the user classes", meaning that these classes must be customized by the user to implement the resolution of its own problem. The virtual methods that the user needs to redefined, are different according to the choosen method (B&B, D&C, ...).

In order to perform strong type checking at compilation time and to avoid downcast, Bob++ makes an exhaustive use of templates in the class definition. The Bob++ classes are parametrized by a `Trait` class which only contains types definitions. The `Trait` must contain the definition for the `Node`, the `Instance`, the `GenChild` and the `Algo` but also for the `Stat` class and `Priority` class. This modelization, is widely used in

```
class MyTrait {
  public:
    typedef MyNode Node;
    typedef MyInstance Instance;
    typedef MyGenChild GenChild;
    typedef Bob::BBAlgo<MyTrait> Algo;
    typedef Bob::BestEPri<MyNode> PriComp;
    typedef Bob::BBStat Stat;
};

class MyNode : public Bob::BBIntMinNode {
    ....
};

class MyInstance : public Bob::BBInstance<MyTrait> {
    ....
};

class MyGenChild: public Bob::BBGenChild<MyTrait> {
    ....
};
```

Fig. 2. Example of a `Trait` class

C++ Standard Template Library (STL). The figure 2 shows a short example of this.

A `Stat` instance stores all the activities of an associated `Algo`. It is used for monitoring the execution of the resolution, either in a offline or online way. The user can extend the definition of the default `Stat` class to add statistics or monitored values which correspond to its specific needs.

The `Priority` class contains the rules to schedule the search. Bob++ provides default classes derived from `Priority`. These default classes can be used when a standard node selection rule (depth first, best evaluation first, etc.) is enough. The user can also choose to create a specific `Priority`-derived class.

A. The log system

Using the `Stat` class, or any of its derived classes, makes possible to the user to get a lot of information from the execution of the algorithm.

The `Stat` is instanciated only once in sequential and shared memory environments, hence the generated log data is global in this case. In distributed memory environments, one `Stat` instance is associated to each running `Algo` instance, which generally corresponds to a processor of the parallel machine. Thus, the different `Stat` objects only store locally-generated data.

While the algorithm is running, a `Stat` object generates statistics about general data such as the number of generated nodes, evaluated nodes, or the number of calls to the generation method of the `GenChild`, as well as problem-specific information. It is possible to redirect the output of a `Stat` object to a file, hence the execution evolution can be analyzed after the execution is finished, or during the execution.

It is also possible to view the data from the file when the execution is not finished. Of course, in this case, all the processors must output their logging information to the same file, thus this behaviour is limited to the sequential and multi-threaded programming environments.

In shared memory programming environments, the

analysis of the contents of a log file allows to obtain both global and processor-specific information about the execution of the algorithm. In distributed memory programming environments, only processor-specific data is available in the different log files. This why the log system offers the possiblilty to configure a `Stat` instance to output its data to a network host machine. This host simply runs a *boblistener* program, which listens for incoming log information from the different parallel computer nodes, and gathers them to a single log file. This way, it remains possible to have real-time access to centralized global and processor-specific information, even in distributed memory environments.

In order to display the generated log information, a tool called *bobview* has been developed. This tool can display the contents of a log file, either after the execution of the algorithm, or while it is running. As the log information contains both global and processor-specific data, *bobview* offers different view modes for the analysis of the execution.

As an example, figure 3 shows the displayed information which is generated by the execution of the MIP solver running on a dual processor shared memory computer, using the multi-threaded programming environment. The first view displays global information about the global priority queue activities, basically the evolution of the number of priority queue insertions, deletions, and number of stored nodes. The two other plots display thread-specific information.

This log system has been developed with the idea that the user should be able to easily get information feedback from the behaviour of its algorithms. This information allows easy tuning of an application without having to wait for the end of its execution.

B. The B&B abstract solver

A solver using the B&B method is written by extending the 3 following “user” classes : `BBNode`, `BBInstance` and `BBGenchild`.

To be able to compare two `BBNode` objects according to their evaluation, the `BBNode` class is parametrized with the sense of optimization (maximisation or minimsation) and the type of the evaluation (int, float, double, etc..). Shortcuts have been predefined, such that the user can choose one the different available default node classes i.e. `BBMinIntNode`, `BBFloatMaxNode` ... as a base class for its own `Node` class. In Bob++, only one type, one class is used to represent the subproblem or a solution of the problem.

The `BBInstance` class must be extended to enable the storage of all the information needed for the resolution of the specific problem. For example when solving a MIP problem, one may plan to include the original MIP problem in the `BBInstance`-derived object, in order to have constant access to the knowledge about integer and continuous variables.

The `BBGenChild` class is not really different from the basic `GenChild` class. The user must extend the `BBGenChild` class to redefine the `operator()` which performs the branching operation from a `Node` object.

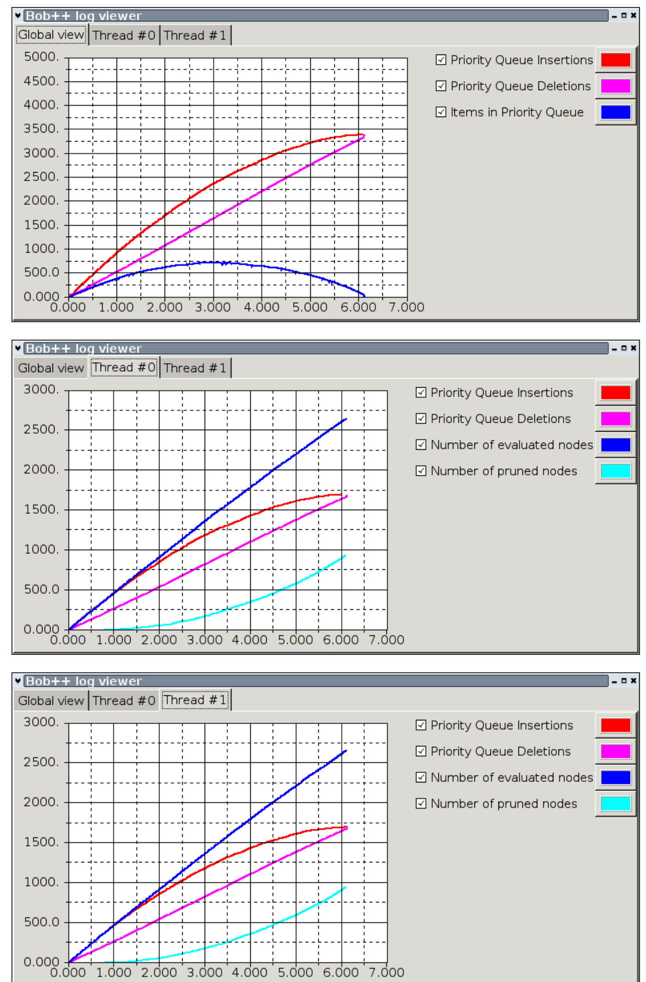

Fig. 3. Execution Analysis of a multi-threaded B&B application

The user code must ensure that each newly-generated child `Node` object which is suitable to be explored is inserted into the priority queue if it is a subproblem.

A `Priority` class is associated to the specialized `Node` class. The `Priority` class must be defined in the `Trait` class.

The figure 4 shows the UML sequence diagram of the search procedure i.e. the `BBAlgo::operator()`.

III. Parallel Interface

Unlike the other frameworks, the Bob++ design integrates a programming interface to implement all kinds of tree search parallelizations. Many of the existing frameworks propose only one parallelization. For example, PICO [3] and ALPS [20], [12], [18] propose a Master-Hub-Worker parallelization, SYMPHONY [17] uses a Master/Worker paradigm using PVM.

There exists a large variety of machines and a large variety of problems. One could be able to use a dual-core processor, as well as cluster of workstations. Bob++ could be extended using this interface by adding a new *Programming environment*. At this time, the proposed environments are sequential, multi-threaded (making use of POSIX threads) and Kaapi-based environments. An MPI version is currently under development.

A. The Programming Environment principles

The main idea leading the Bob++ design to obtain parallel solvers is a generalization of the *Global priority Queue (GPQ)* used to implement the old Bob [9], [11] library. The principle is to have different instances of `Algo` that are executed on different processors. These different `Algo` communicate, and are synchronized using high level communication tools. These tools are mainly the data structures that store the subproblems, the solutions or other information needed by a specific method. These *Global Data Structures* are equivalent to the Knowledge Pool introduced by ALPS [20], [12], [18].

For example in the context of a B&B, these "parallel" instances of `BBAlgo` will mainly execute a loop where at each iteration the `operator()` of an instance of the user `BBGenChild` is executed on a pending node obtained from a *Global priority Queue* `GPQ`. The new generated nodes are re-inserted in `GPQ`. In the same way if a solution is found, the data structure called *Global Solution* `GSol` will be updated with this new solution.

The different `GPQ` and `GSol` must communicate to ensure that each `Algo` has enough work to do and has the latest updated Solution.

The `GPQ` and `GSol` can be considered from the `Algo` point of view as high level communication tools since the different `Algo` instances only communicate through these tools.

This design does not constraint the algorithm used to manage the nodes or broadcast a new solution value. It does not imply a specific parallel strategy. The goal is to be able to implement, a Master/Hub/Worker strategy used by PICO [3], PEBBL [4] and ALPS [20], [12], [18], but also simple Master/Slave strategy using MPI, or different parallelization strategies using the POSIX threads on a shared memory machine. One could also implement a parallelization where the load balancing strategy takes into account the heterogeneity of a Grid. The figure 4 shows the UML sequence diagram of the `BBAlgo::operator()`, which performs the main loop of the algorithm. The `GPQ` and the `GSol` are respectively called `ThrPQ` and `ThrSol` which are concrete classes of the mutlithreaded environment.

An interesting property of this design is also that the parallel and sequential `Algo` implementations are exactly the same. The only difference is the different implementation of the *Global Data Structures*.

Abstract classes are proposed in the Bob++ Framework, to define the interfaces needed by the `Algo`. The `PQInterface` defines the interface used by the B&B algorithm to store the pending nodes. Concrete classes are also defined. For example the `PQSkew` class extends the `PQInterface` interface. In this case, the algorithm used to store the nodes according to their priorities is the Skew-Heap [16].

The *Threaded* programming environment defines another concrete class which extends the `PQInterface` called the `ThrPQ` (see figure 4). The goal of this class is to enable the access to the `PQSkew` in mutual exclusion mode. The figure 5 shows the UML sequence diagram

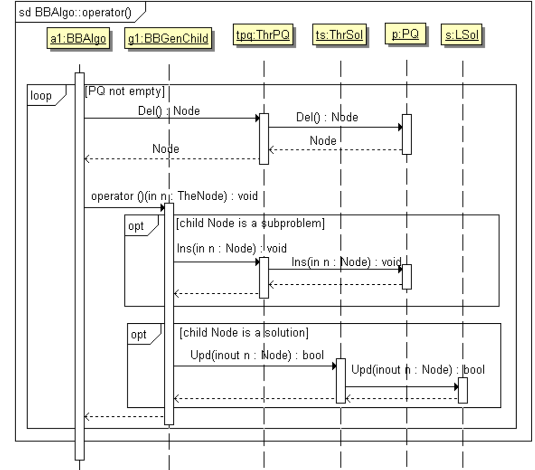

Fig. 4. The UML sequence diagram of a search

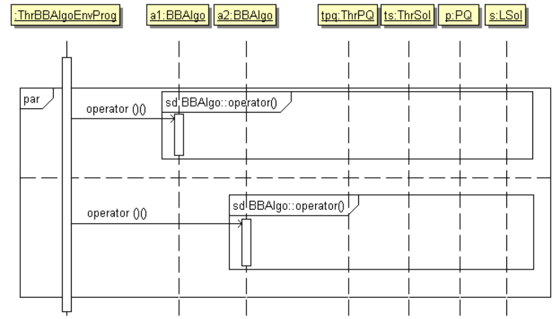

Fig. 5. The UML sequence diagram of multithreaded search

of the threaded environment, which calls in parallel the Search sequence diagram (see figure 4 on 2 differents instances of the `BBAlgo` class. In this diagram, the two instances of the BBAlgo use the same instance of `ThrPQ` and the same instance of `ThrSol`.

Therefore, the initialisation of an `Algo` e.g. the initialisation of an instance of the `Algo` itself and the initialization of the data structure it uses, are performed by the *Programming Environment*. The Bob++ Programming Environment class which realizes all this initialization is the `AlgoEnvProg` class.

More general initialization than which is made on the `AlgoEnvProg` is often necessary for a specific environment. For example, in a context of a multithreaded environment (called *Thr*), the different threads must be created, but after the parsing of the command line. These initializations are usually made in *static* methods. The choice between the environments is made in the *main* function. Fig. 6 shows an example of a *main* function that can be used to generate different solvers for the same problem, making use of different programming environments.

The multithreaded environment has been proposed in order to propose an easy and powerful use of modern dual- or quad-core processors. It also constitutes the first step for the implementation of a hybrid algorithm for clusters of shared-memory machines.

```
int main(int argc, char **argv) {
#ifdef Atha
   Bob::AthaBBAlgoEnvProg<MyTrait> env;
   Bob::AthaEnvProg::init(n,v);
   Bob::core::Config(n,v);
#elif defined(Threaded)
   Bob::ThrBBAlgoEnvProg<MyTrait> env;
   Bob::ThrEnvProg::init();
   Bob::core::Config(n,v);
   Bob::ThrEnvProg::start();
#else
   Bob::SeqBBAlgoEnvProg<MyTrait> env;
   Bob::core::Config(n,v);
#endif
   MyInstance *Instance=new MyInstance( );
   env(Instance);
#ifdef Atha
   Bob::AthaEnvProg::end();
#elif defined(Threaded)
   Bob::ThrEnvProg::stop();
#endif
   Bob::core::End();
   delete Instance;
}
```

Fig. 6. Example of a `main` function

B. The Athapascan Environment

We use the Athapascan/Kaapi library [23] as a parallel environment for Bob++. Athapascan/Kaapi is a high level parallel programming tool. Its aim is to schedule a set of tasks which are created dynamically using Athapascan's *Fork* primitive. Athapascan/Kaapi has been developed in such a way that one does not have to worry about the specific machine architecture or the optimization of load balancing between processors. The created tasks are scheduled on the different processors in order to complete the work.

In the context of B&B, the Athapascan/Kaapi library is a very interesting tool to parallelize the search procedure. A strategy consists in creating a task to explore a subtree rooted on a node. The leaves of the subtree which are subproblems become new tasks. The idea is to have enough tasks to ensure that all processors have enough work. In Bob++, the task that explores a subtree is a full instance of a Bob++ `Algo`. The Athapascan primitive used to `fork` a task is called in the `GPQ`'s *insert* method. Different strategies are defined in order to have different sizes of subtrees according to the position of the root node in the tree. When the node is very close to the root node of the B&B tree, the size of the subtree explored by a task must be very small, to produce work very quickly. When the root node of a subtree is on the middle of the B&B tree, the size of the subtree could increase. The Athapascan/Kaapi runtime stores the list of waiting tasks in a list which is local to each process. If the process runs on a machine that has several processors, the list is shared among the threads that are running on each processor and that execute the tasks. The next section shows some results with this environment.

IV. Bob++ Applications

This section describes the more advanced applications we designed on top of Bob++.

Instance	# Procs	Time (mn)
Nugent17	168	0.33
Nugent20	50	5.69
Nugent20	168	3.58
Nugent20	184	3.32
Nugent21	50	11.5
Nugent21	168	7.19
Nugent22	50	9.8
Nugent24	140	153.7

TABLE I: Tests of the QAP code

A. The QAP solver

The first Branch-and-Bound application is a solver for the Quadratic Assignment Problem. The QAP was formulated by Koopmans and Beckmann (1957) [8] as the task to find a permutation π of $\{1, ..., n\}$ that minimizes:

$$min \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} C_{ijkl}x_{ij}x_{kl} + \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \quad (1)$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i = 1, ..., n \quad (2)$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad j = 1, ..., n \quad (3)$$

$$x_{ij} = 0, 1 \quad i, j = 1, .., n \quad (4)$$

where C_{ijkl} denotes the cost incurred by locating facility i on location j and facility k on location l. It represents the product of f_{ik} (flow between the facilities i and k) and d_{jl} (distance between the locations j and l). The QAP Solver on top of Bob++ is based upon the dual procedure of Hahn and Grant [6] to compute the lower bound and the polytomic branching strategy of Mautor and Roucairol [10]. This code has been tested on various machines [2], [1] using the Athapascan/Kaapi parallelization. In [7] this code with Bob++ has been tested with a fault tolerant version of Athapascan/Kaapi. Table I shows the results of the QAP code on the some Nugent instances from the QAPLIB. The runs have been realized on a Itanium-2 cluster of the university of Grenoble. The CPU times show that the algorithm is scalable.

B. The MIP solver

The second Branch-and-Bound application which has been developed is a simple Mixed-Integer Programming (MIP) solver. The solver can take the description of a problem from a *CPLEX* LP file, then solve it using the chosen parallel environment. The LP solver used during the node evaluation can be chosen from different existing solvers : *CPLEX* from Ilog or *Xpress-MP* by Dash Optimization are the possible available commercial solvers, but solvers from the open source community can also be used : *GLPK*, *LP_solve* and *Clp*.

For the moment, the integrated MIP solver is at a very early stage of development. It uses simple

branching methods, and does not make use of generic cuts. Therefore, it cannot be compared to the available efficiency-proven MIP solvers. However, we obtain very good speed-up improvements using the POSIX threads environment, on multi-processor shared-memory architectures. Our objective for the near future is to perform heavy testings on distributed memory environments, as soon as the necessary code is added to the MIP solver.

The flexibility for the LP solver choice has been made possible by using the *Glop* library which we developed ([5], [22]). *Glop* is a free-software application programming interface (API), which wraps solver-specific LP or MIP solver function calls. Its base idea is to provide the user with a generalized API which allows to suppress the dependence of the code to a specific solver. This way, it is for example possible to compare the performance of the different solvers by using the same optimization code.

Another solver interface from the open-source community already exists : *Osi* from the COIN OR project [13]. *Glop* differs from *Osi* to the fact that it is only a lightweight wrapper to the solver API calls, whereas *Osi* is a C++ interface which allows the problem modelization and resolution in an object-oriented way, which is not the usual way solver APIs usually work.

V. Summary and Future Work

In this paper, we have presented the main features of the Bob++ Framework. We described the User Programming Interface and the Parallel Programming Interface, called the Programming Environment interface. We explain how is it possible to use a high level parallel library in Bob++. We show the good performance we obtained on the QAP problem using the Athapascan/Kaapi library. We also quickly present the MIP solver that uses the *Glop* library using the Multithreaded environment. Its development is still in progress, this is why no good results are currently available. The current solver can not be compared to other existing solvers as it still lacks cut generation.

We present the log system of Bob++, which provides the ability to analyze the execution of the parallel algorithm. The statistics are available during program execution, even in distributed memory environments. They can also be analyzed after the execution.

Current work includes the addition of generic cut generation to the MIP solver, which as we expect should provide significant performance increase.

Several other features could be proposed for the Application Interface and to increase the Parallel Interface. As said in the introduction, we would like Bob++ to propose interfaces for other methods as Dynamic Programming and A*. Higher level interfaces could be proposed to facilitate the programming of more specialized algorithms built on top of B&B, like Branch-and-Price and Branch-and-Cut. An old version of Bob++ used to include these features: we have to port them on this current version. The parallel interface could be extended to propose other parallel library ports like MPI.

References

[1] A. Djerrah. *Résolution Exacte d'un problème d'optimisation combinatoire NP-difficile sur grilles de machines.* PhD thesis, Université de Versailles-Saint-Quentin, july 2006. In French.

[2] A. Djerrah, S. Jafar, V.-D. Cung, and P. Hahn. Solving QAP on cluster with a bound of reformulation linearization techniques. In *In the 17th IMACS World Congress Scientific Computation, Applied Mathematics and Simulation IMACS2005*, Paris, France, July 2005.

[3] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch and bound. In E. Scientific, editor, *Proceedings of the Workshop on Inherently Parallel Algorithms in Optimization and Feasibility and their Applications*, Studies in Computational Mathematics, pages 219–265, 2001.

[4] J. Eckstein, C. A. Phillips, and W. E. Hart. PEBBL 1.0 User Guide. RRR 19-2006, RUTCOR, August 2006.

[5] F. Galea. *Problèmes d'optimisation en curiethérapie.* PhD thesis, Université de Versailles–Saint-Quentin-en-Yvelines – UVSQ, 45, Avenue des Etats-Unis, 78035 Versailles Cedex, FRANCE, Sept. 2006. In French.

[6] P. Hahn and T. Grant. Lower bounds for the quadratic assignment problem based upon a dual formulation. *Operations Research*, 46:912–922, 1998.

[7] S. Jafar. *Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité.* PhD thesis, INP de Grenoble, June 2006. In French.

[8] T. Koopmans and M. Beckman. Assignment Problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[9] B. Le Cun, C. Roucairol, and the PNN team. Bob : a unified platform for implementing branch-and-bound like algorithms. RR 95/16, Laboratoire PR*i*SM, Université de Versailles - Saint Quentin en Yvelines, Sept. 1995.

[10] T. Mautor and C. Roucairol. Difficulties of Exact Methods for Solving the Quadratic Assignment Problem. In P. M. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, volume 16 of *Discrete Mathematics and Theoretical Computer Science*, pages 263–274. DIMACS, American Mathematical Society, May 1994.

[11] M.Benaichouche, V.Cung, S.Dovaji, B. Cun, T.Mautor, and C.Roucairol. *Building a Parallel Branch and Bound Library*, volume 1024 of *LNCS State-of-the-Art Survey*, pages 201–231. Springer-Verlag, 1996.

[12] T. Ralphs, L. Ladnyi, and M. Saltzman. A Library Hierarchy for Implementing Scalable Parallel Search algorithms. *The Journal of Supercomputing*, 28(2):215–234, may 2004.

[13] M. J. Saltzman. *COIN-OR: An Open Source Library for optimization.* Kluwer, Boston, 2002.

[14] Y. Shinano, M. Higaki, and R. Hirabayashi. An Interface Design for General Parallel Branch-and-Bound Algorithms. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 277–284, 1996.

[15] Y. Shinano, M. Higari, and R. Hirabayashi. Generalized utility for parallel branch-and-bound algorithms. In *Proceedings of the 1995 Seventh Symposium on Parallel and Distributed Processing*, number 392, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[16] D. Sleator and R. Tarjan. Self-Adjusting Heaps. *SIAM J. Comput.*, 15(1):52–69, Feb. 1986.

[17] T.K.Ralphs and M. Guzelsoy. The SYMPHONY Callable Library for Mixed Integer Programming. In *In proceedings of the Ninth INFORMS Computing Society Conference*, 2005.

[18] T.K.Ralphs, L. Ladányi, and M. Saltzman. Parallel Branch, Cut, and Price for Large-scale Discrete Optimization. *Mathematical Programming*, 98(253), 2003.

[19] S. Tschoke and T. Polzer. Portable parallel branch-and-bound library user manual, library version 2.0. Technical report, Department of Computer Sciences, University of Paderborn., 1996.

[20] Y. Xu, T.K.Ralphs, L. Ladnyi, and M. Saltzman. ALPS: A Framework for Implementing Parallel Search Algorithms. In *In proceedings of the Ninth INFORMS Computing Society Conference*, 2005.

[21] Bob++: Framework to solve Combinatorial Optimization

Problems
http://bobpp.prism.uvsq.fr/.
[22] Glop, the Generic Linear Optimization Package.
http://glop.prism.uvsq.fr/.
[23] Kaapi: a library for high performance parallel computing based on an abstract representation to adapt the computation to the available computing ressources.
http://kaapi.gforge.inria.fr/.