A Grid-enabled Branch and Bound Algorithm for Solving Challenging Combinatorial Optimization Problems*

M. Mezmaz, N. Melab and E-G. Talbi
LIFL, CNRS UMR 8022
INRIA Futurs - DOLPHIN
Université des Sciences et Technologies de Lille
59655 - Villeneuve d'Ascq cedex - France

E-mail: {mezmaz,melab,talbi}@lifl.fr

Abstract

Solving optimally large instances of combinatorial optimization problems requires a huge amount of computational resources. In this paper, we propose an adaptation of the parallel Branch and Bound algorithm for computational grids. Such gridification is based on new ways to efficiently deal with some crucial issues, mainly dynamic adaptive load balancing, fault tolerance, global information sharing and termination detection of the algorithm. A new efficient coding of the work units (search sub-trees) distributed during the exploration of the search tree is proposed to optimize the involved communications. The algorithm has been implemented following a large scale idle time stealing paradigm (Farmer-Worker). It has been experimented on a Flow-Shop problem instance ($Ta056$) that has never been optimally solved. The new algorithm allowed to realize a success story as the optimal solution has been found with proof of optimality, within 25 *days using about* 1900 *processors belonging to* 9 *Nation-wide distinct clusters (administration domains). During the resolution, the worker processors were exploited with an average of* 97% *while the farmer processor was exploited only* 1.7% *of the time. These two rates are good indicators on the efficiency of the proposed approach and its scalability.*

Keywords: *Branch and Bound, Parallel Computing, Grid Computing, Flow-Shop Problem, Performance Evaluation.*

*This work is part of the *CHallenge in Combinatorial Optimization (CHOC)* project supported by the *National French Research Agency (ANR)* through the *High-Performance Computing and Computational Grids (CIGC)* programme.

1 Introduction

Combinatorial optimization addresses problems for which the resolution consists in finding the (near-)optimal configuration(s) among a large finite set of possible configurations. In practice, most of these problems are naturally NP-hard and complex. The Branch and Bound (B&B) algorithm is one of the most popular methods to solve exactly this kind of problems. This algorithm allows to reduce considerably the computation time required to explore all the solution space associated with the problem being solved. However, the exploration time remains considerable, and using parallel processing is one of the major and popular ways to reduce it. Many parallel B&B approaches have been proposed in the literature [4, 2, 5]. A taxonomy of associated parallel models is presented in [7]. Four models are mainly identified and studied within the context of grid computing. These models are briefly surveyed in the next section. In this paper, we focus on the most used of them: the parallel exploration of the search tree.

We are interested in the exploitation and deployment of this model on a computational grid. The model poses several issues related to the characteristics of the model itself and the properties of the grids. The major of these issues are load balancing, fault tolerance, termination detection and global information sharing (best solution found so far). Indeed, the irregular nature of the tree explored by the algorithm and the volatile and large scale nature of the grid involves a large amount of load balancing and checkpointing operations. These operations require an exorbitant communication and storage cost associated with the work units (collections of nodes) dynamically generated. In addition, the large scale nature of the grid

makes it difficult to efficiently exploit the synchronous mode for the exploration. The asynchronous mode is thus required which makes hard the termination detection of the algorithm and the global information sharing particularly in a large scale grid.

In this paper, we propose a grid-oriented approach based on special codings of the explored tree and the work units. These codings allow to optimize the dynamic distribution and checkpointing mechanisms on the grid, and to implicitly detect the termination of the algorithm and efficiently share the global information. The algorithm has been applied to the Flow-Shop scheduling problem, one of the hardest challenging problems in combinatorial optimization. The problem consists roughly to find a schedule of a set of jobs on a pool of machines that minimizes the total execution time (makespan). The jobs must be scheduled in the same order on all machines, and each machine can not be simultaneously assigned to two jobs. Using our grid-based algorithm, the problem instance (50 jobs on 20 machines) has been optimally solved for the first time. The method allows not only to improve the best known solution for the problem instance but it also provides a proof of the optimality of the provided solution. The experiments have been performed on a grid of over 1889 processors belonging to Grid5000 and different educational networks of Universit de Lille1 (Polytech'Lille, IEEA, IUT "A").

The rest of the paper is organized as follows. *Section 2* gives an overview of the B&B algorithm and its parallelization on the computational grid. The proposed parallel approach is described in *Section 3*. *Section 4* presents its implementation based on the farmer-worker paradigm. *Section 5* describes the experiments performed for solving a Flow-Shop problem instance that has never been solved optimally. *Section 6* draws some conclusions and perspectives of this work.

2 The B&B algorithm and Grid Computing

Solving a problem in combinatorial optimization consists in exploring a search space to provide a (near-)optimal solution. To each candidate solution of the search space is associated a cost. Solving exactly a combinatorial optimization problem consists in finding the solution having the optimal cost. For this purpose, the B&B algorithm is based on an implicit enumeration of all the solutions of the considered problem. The search space is explored by dynamically building a tree whose root node represents the problem being solved and its whole associated search space. The leaf nodes are the potential solutions and the internal nodes are subspaces of the total solution space. The size of these subspaces is increasingly reduced as one approaches the leaves.

The construction of such a tree and its exploration are performed using four operators: *branching*, *bounding*, *selection* and *elimination*. The algorithm proceeds in several iterations during which the best solution found so far is progressively improved. The generated and not yet treated nodes are kept in a list whose initial content is limited to only the root node. The four operators intervene at each iteration of the algorithm. The B&B makes it possible to reduce considerably the computation time necessary to explore the whole solution space. However, this remains considerable and parallel processing is thus required to reduce the exploration time.

In [7], four parallel models are identified for B&B algorithms: (1) *the parallel multi-parametric model*, (2) *the parallel tree exploration*, (3) *the parallel evaluation of the bounds*, and (4) *the parallel evaluation of a single bound*. The model (1) consists in launching simultaneously several B&B processes. These processes differ by one or more operators, or have the same operators, but parameterized differently. The trees explored in this model are not necessarily the same. Model (1) guarantees the implicit exploration of the whole solution space. Like model (1), model (2) also consists in launching several B&B processes. However, all the processes in model (2) are similar, and explore simultaneously the same tree. Among the four models, this model is the most popular and studied one. Unlike the two previous models, models (3) and (4) suppose the launching of only one B&B process. They do not allow to parallelize the whole B&B algorithm as both models (1) and (2) do, but they parallelize only the bounding operator. In model (3), each process evaluates the bounds of a distinct pool of nodes, while in model (4) a set of processes evaluate in parallel the bound of a single node.

In [7], an analysis of these different parallel models is presented within the context of grid computing. This analysis takes into account the characteristics of the computational grid: its multi-administrative domain nature, the dynamic availability and heterogeneity of its resources, and its large scaleness. In this paper, we focus on the parallel model (2) of the B&B algorithm. In addition to the properties of the computational grid, the characteristics of the model must be considered: the irregular nature of the explored search tree and its asynchronous exploration mode. All these characteristics of the grid and the parallel model make more challenging several issues: load balancing, fault tolerance, termination detection and global information sharing (best solution found so far). In the next section, we propose a new approach to deal with these issues.

3 The Proposed Approach: Concepts and Operators

The proposed approach is based on the *parallel tree exploration model* using a depth first *search strategy*. In this approach, a list of active nodes is considered. Active nodes are nodes generated during the exploration but not yet visited. During the search, this list evolves continuously and the algorithm stops once this list becomes empty. The list of active nodes covers a set of tree nodes made up by all the nodes which can be potentially explored from each node of the list. The principle of the approach is based on the assignment of a number to each node of the tree. The numbers of any set of nodes, covered by a list of active nodes, always form an interval (two numbers). The approach thus defines a relation of equivalence between the concept of *list of active nodes* and the concept of *interval*. Knowing one the two should make it possible to deduce the other. As its size is reduced, the interval is used to optimize communication and check-pointing operations, while the list of active nodes is used for exploration. In order to switch from one concept to the other, the approach defines two additional operators: the *fold operator* and the *unfold operator*. The fold operator deduces an interval from a list of active nodes, and the unfold operator deduces a list of active nodes from an interval. To define these two operators, three new concepts are introduced: the node *weight*, the node *number* and the node *range*.

3.1 Node weight

The weight of a given node n, noted $weight(n)$, is the number of leaves of the sub-tree of which n is the root node. Equation (1) defines the weight of a node in a recursive way. The weight of a leaf is equal to 1, and the weight of an internal node is equal to the sum of the weights of its node children. This definition is at the same time general and inapplicable. Indeed, it is general since it defines the weight of a node for any tree, and inapplicable since the size of the tree is exponential.

$$weight(n) = \begin{cases} 1 \;\; if \;\; children(n) = \phi \\ \sum_{i \in children(n)} weight(i) \quad otherwise \end{cases} \tag{1}$$

However, the knowledge of the structure of a tree makes it possible to simplify this definition. Equations (2) and (3) define in a simpler way the weight of a node for respectively a *binary tree* and a *permutation tree*. In these two definitions, the depth of a node n, noted $depth(n)$, is the number of nodes which separate it from the root node. P is the depth associated with the leaves. A binary tree is a tree where except the leaves each node has two children. A permutation tree is the tree associated with problems where the goal is to find a permutation among a finite set of elements.

$$weight(n) = 2^{(P - depth(n))} \tag{2}$$

$$weight(n) = (P - depth(n))! \tag{3}$$

Many combinatorial optimization problems can be modeled as a permutation tree. In such tree, any node n, except the root node, satisfies the condition (4).

$$|children(n)| = |children(parent(n))| - 1 \tag{4}$$

In a binary tree, a permutation tree or any other tree of regular structure, the nodes of the same depth have the same weight. Consequently, instead of associating the weights to the nodes, it is simpler to associate them to the depths, and to deduce the weight of a node from its associated depth. At the beginning of the B&B algorithm, a vector which gives the weight associated with each depth is calculated. Using this vector, it is possible to find the weight of a node knowing its depth. Figure 1 gives an example of the weights associated with the depths in a permutation tree.

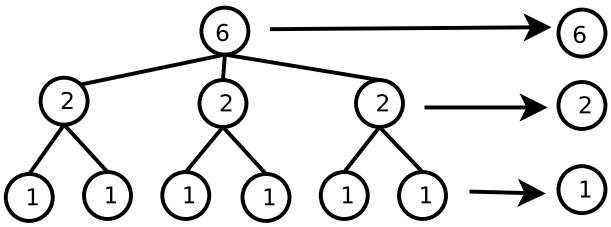

Figure 1. Weight of a node

3.2 Node number

To each node n of the tree is assigned a number noted $number(n)$. As equation (5) indicates it, the number of a node n can be obtained using the nodes of its path. The path of a node n, noted $path(n)$, is the set of nodes between the root node and the node n. The node n and the root node are always included in $path(n)$. To find the number of a node n, it is sufficient to know the "$predecessors$" of each node in $path(n)$. The "$predecessors$" of a node n, noted $predecessors(n)$, is the set of sibling nodes of n which are generated before n.

$$number(n) = \sum_{i \in path(n)} \sum_{j \in predecessors(i)} weight(j) \tag{5}$$

The definition (5) can be applied to any tree independently of its structure. Equation (6) gives a simpler definition for the trees of regular structure such as the binary or

permutation tree. This definition is based on the fact that, in this kind of trees, nodes of the same depth have a similar weight. It is sufficient to know the path of a node and the rank of each node of this path. The rank of a node n, noted $rank(n)$, is the position of n among its sibling nodes. During the generation of the children of a given node, the rank of the first generated node is 0, the $rank$ of the second generated node is 1, and so on. Figure 2 gives an example of numbers obtained in a permutation tree. These numbers are inside their associated nodes.

$$number(n) = \sum_{i \in path(n)} rank(i) * weight(i) \quad (6)$$

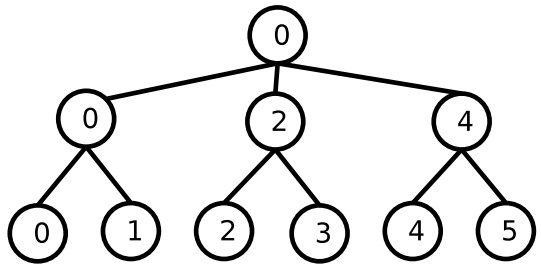

Figure 2. Illustration of the node number

3.3 Node range

The range of a node n, noted $range(n)$, is the interval to which belong the node numbers of the sub-tree of which n is the root node. Figure 3 gives an example of the range associated with each node of a permutation tree. As indicated in equation (7), the beginning of the range of a node is equal to its number, and its end is equal to the sum of its number and its weight.

$$range(n) = [number(n), number(n) + weight(n)[\quad (7)$$

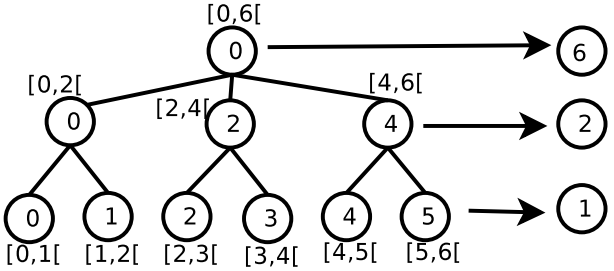

Figure 3. Illustration of a node range

3.4 Fold operator

The role of Fold operator is to deduce from a list N of active nodes the interval to which belong the numbers associated to the nodes which are potentially explored using the nodes of N. In this paper, the interval of a list N of active nodes is noted $interval(N)$. This is equivalent to the union of all the ranges of the nodes N (see (8)).

$$interval(N) = \cup_{i \in N} \ interval(i) \quad (8)$$

In B&B, the position of the nodes in an active list N depends on the search strategy adopted by the selection operator. Let N_1, N_2 ... N_k be the order by which these nodes appear in the list, and $[A_1, B_1[$, $[A_2, B_2[$ ... $[A_k, B_k[$ be their respective ranges. Condition (9) is always checked when the search strategy adopted is depth first. Therefore, $interval(N)$ can be found without knowing all ranges of the active nodes. As (10) indicates it, it is sufficient to know the ranges of N_1 and N_k. Or more simply, it is enough to know the numbers associated with these two nodes and the weight of N_k. Figure 4 gives an example illustrating the folding of a list of active nodes into an interval.

$$\forall \ i < k, \ \ B_i = A_{i+1} \quad (9)$$

$$interval(N) = [number(N_1), number(N_k) + weight(N_k)[\quad (10)$$

Indeed, in Figure 4 the grey colored active nodes 1, 2 and 4 are considered. The union of ranges associated to these nodes is $[1, 5[$. This corresponds exactly to the result returned by equation (10) i.e. $[number(1), number(4) + weight(4)[= [1, 4 + 1[= [1, 5[$

3.5 Unfold operator

The Unfold operator deduces from an interval $[A, B[$ a list of active nodes noted $nodes([A, B[)$. As indicated in (11), $nodes([A, B[)$ is composed by the nodes of the tree whose range is included in $[A, B[$, and for which the range of their parent is not included in $[A, B[$. These two conditions guarantee that $nodes([A, B[)$ is an unique and minimal list. Indeed, it is impossible to find another list whose length is lower than $nodes([A, B[)$, and which allows to explore the nodes with numbers belonging to $[A, B[$.

$$nodes([A, B[) = \left\{ \begin{array}{c} n/ \\ range(n) \subseteq [A, B[\\ range(p) \not\subset [A, B[\\ p = parent(n) \end{array} \right\} \quad (11)$$

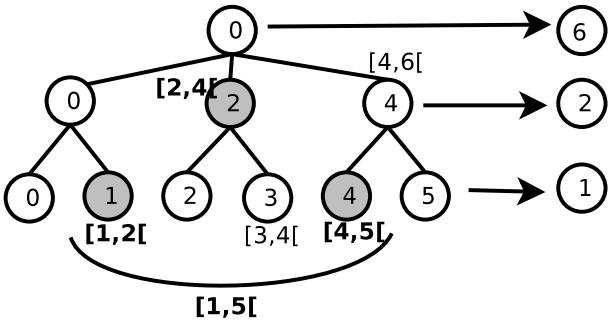

Figure 4. Illustration of active nodes and interval

$$elim(n) = (range(n) \subseteq [A, B[\ \ or \ \ (range(n) \cap [A, B[) = \phi) \quad (12)$$

Finding $nodes([A, B[)$ can be done using a B&B algorithm in which operators, except the elimination operator (elim), are the same ones as those of the B&B algorithm. This algorithm is based on the range of a node to choose between an elimination and a branching operator. As (12) indicates it, a node n is eliminated when its range is included in $[A, B[$, or when its range and $[A, B[$ are completely disjointed. Otherwise, the node n is decomposed. In a tree with a maximum depth P, the B&B performs less than P decompositions. This guarantees the low cost of the unfold operator. As equation (13) indicates it, the list of $nodes([A, B[)$ is made up by all the eliminated nodes which their interval is included in $[A, B[$. Figure 4 gives an example illustrating the transformation of an interval in a list of active nodes.

$$nodes([A, B[) = \{n / elim(n) \ \ and \ \ range(n) \subseteq [A, B[\} \quad (13)$$

4 The Proposed Grid-enabled Parallel B&B

Fold and Unfold operators can be used for the parallelization of the B&B according to different paradigms. In this paper, the farmer-worker paradigm is used. In this paradigm, only one processor plays the role of the farmer, and all the other processors are workers. This paradigm is relatively simple to use but the farmer can constitute a bottleneck. However, communicating and handling intervals instead of list of active nodes allow to significantly reduce the communication costs and the farmer load.

In the adopted farmer-worker approach, the workers perform the B&B processes, and the farmer hosts the coordinator. Each B&B process explores an interval of node numbers, and manages the local best solution found so far. On the other hand, the coordinator keeps a copy of all the not yet explored intervals, and manages the global best solution found so far. The copies of the intervals are kept in a set noted $INTERVALS$, and the global best solution in a variable noted $SOLUTION$. Figure 5 gives an example with three B&B processes and a coordinator. In this example, three intervals are being explored, and the fourth one is waiting for a free available B&B process.

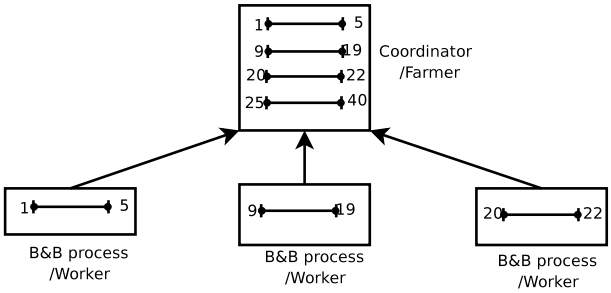

Figure 5. An example with B&B processes and a coordinator

In addition to balancing the load between B&B processes, other issues must be taken into account. Indeed, the B&B processes make three assumptions about the workers. They suppose that they are likely to break down and can be behind fire-walls. Consequently, these processes are fault tolerant, are launched according to the cycle stealing model, and exchange their messages according to the pull model. The only assumption of the coordinator about the farmer is that it can fail. The coordinator manages only the possible failures of the farmer. The following sections explain how this farmer-worker approach deals with the quoted issues.

4.1 Fault Tolerance

The coordinator manages a possible failure of the farmer by periodically saving, in two files, the contents of $INTERVALS$ and $SOLUTION$. If a farmer failure occurs, the coordinator initializes $INTERVALS$ and $SOLUTION$ by the contents of these files. The B&B processes manage the worker failures by regularly updating the copy of their local interval in $INTERVALS$, and by informing the coordinator each time a new solution is found. When a worker fails the last copy of its interval is either entirely given by the coordinator to another B&B process, or shared between several B&B processes.

B&B processes update the copy of their interval using the *intersection* operator. This operator updates the interval being explored and its copy in $INTERVALS$. Let $[A, B[$ be an interval being explored in a B&B process, and

$[A', B'[$ its copy in $INTERVALS$. During the exploration process, the two intervals $[A, B[$ and $[A', B'[$ evolve continuously. Indeed, a B&B process, by exploring $[A, B[$, increments the value of A and does not change the value of B. While the load balancing mechanism, explained in the following section, decrements the value of B' and does not change the value of A'. The beginning of an interval is likely also to be incremented by several B&B processes. This occurs when the load balancing mechanism attributes the same interval to several processes. As indicated in (14), the intersection between two intervals is done by considering the maximum of their beginnings and the minimum of their ends.

$$[A, B[\cap[A', B'[= [max(A, A'), min(B, B')[\qquad (14)$$

4.2 Load balancing

In this approach, the work unit of a B&B process is the exploration of an interval. A B&B process requests an interval when it joins the calculation for the first time, and when it finishes the exploration of its interval. The role of the coordinator is then to assign an interval to it. This is done using two interval operators: the *selection operator* and *partitioning operator*. The role of the former is to select an interval from $INTERVALS$, and the role of the latter is to divide the selected interval. The partitioning operator divides an interval $[A, B[$ into two intervals $[A, C[$ and $[C, B[$. The *holder process*, the one to which belongs $[A, B[$, keeps $[A, C[$ since it already explores it starting from A, while the *requesting process*, the one which requests a new interval, obtains $[C, B[$. After a certain time, the holder process is also informed to limit its exploration to $[A, C[$ instead of $[A, B[$. Indeed, as mentioned in the previous subsection, the B&B processes regularly contact the coordinator to update their intervals.

After a partitioning operation, the $INTERVALS$ set is updated by replacing $[A, B[$ with $[A, C[$, and by adding $[C, B[$ to $INTERVALS$. Both intervals $[A, C[$ and $[C, B[$ do not have necessarily the same length. Indeed, the requesting and the holder processes are deployed in an environment where the hosts are heterogeneous, volatile and not dedicated. Consequently, the lengths of the two intervals must be proportional to the participation of each one in the calculation. The choice of the partitioning point C depends on the power and the availability of the processors which host the holder and the requesting processes.

In this approach, it is possible that some intervals are not assigned to any process. This occurs at the beginning of a calculation or after a farmer or worker failure. Such intervals are managed by supposing that they belong to a virtual process which has a null power. Consequently, the partitioning point C of these intervals is equal to A. They are thus assigned entirely to the requesting process. To avoid obtaining intervals of small size (granularity issue), the partitioning operator is parameterized by a threshold. An interval which has a length lower than this threshold is duplicated instead of being divided. The coordinator keeps only one copy of a duplicated interval, even if it is assigned to several processes. Duplication is very important during the termination phase of the algorithm. Indeed, if some intervals are assigned to slower processors their duplication on more powerful processors will allow to speed up the search.

In addition to the choice of the partitioning point, the choice of the interval $[A, B[$ to be divided is another issue taken into account by the coordinator. The goal is to assign, to the requesting process, the greatest available interval. The selection operator does not choose the greatest interval $[A, B[$ of $INTERVALS$, but the one which produces the greatest interval $[C, B[$. Such interval is sufficiently long (coarse-grained work unit) to be assigned to a new requesting process.

4.3 Termination detection

The management of the termination detection is another issue which is crucial in a grid. In this approach, the resolution stops once $INTERVALS$ becomes empty. At the beginning, $INTERVALS$ contains only one interval which corresponds to the whole tree. The beginning of this interval is equal to 0, the smallest number of the tree, and its end is equal to the greatest number of the tree. In other words, $INTERVALS$ is initialized by the range of the root node.

During the exploration process, the cardinality of $INTERVALS$ is almost equal to the number of the B&B processes, while its size continuously decreases. The cardinality of $INTERVALS$ is the number of intervals that it contains, and the size of $INTERVALS$ is the sum of the lengths of its intervals. This size corresponds to the number of the not yet explored solutions of the search space.

Any empty interval (with a beginning higher than the end) of $INTERVALS$ is automatically removed. During the resolution, the size of $INTERVALS$ decreases constantly. Calculation stops once $INTERVALS$ becomes empty, and its size is thus equal to 0. When $INTERVALS$ is empty, a B&B process, which contacts the coordinator to update/ask for an interval, is informed by the coordinator that it must terminate.

4.4 Solution sharing

In this farmer-worker approach, the solution sharing is done by using $SOLUTION$. As mentioned in Section 4, $SOLUTION$ contains the global optimal solution. In addition to this variable, each B&B process manages its own local optimal solution found so far. The objective of the solution sharing is to reflect any improvement of any local optimal solution in all the other B&B processes. This is achieved using three rules: A B&B process (1) initializes its local optimal solution by $SOLUTION$; (2) immediately informs the coordinator each time its local solution is improved ; (3) regularly reads $SOLUTION$ to update its local optimal solution.

5 Experimentation

5.1 The permutation Flow-Shop problem

The Flow-Shop problem is one of the numerous scheduling problems that has received a great attention given its importance in many industrial areas. The problem can be formulated as a set of N jobs $J_1, J_2, \ldots, J_N$ to be scheduled on M machines. The machines are critical resources as each machine can not be simultaneously assigned to two jobs. Each job J_i is composed of M consecutive tasks $t_{i1}, \ldots, t_{iM}$, where t_{ij} designates the j^{th} task of the job J_i requiring the machine m_j. To each task t_{ij} is associated a processing time p_{ij} and a starting time s_{ij}. The problem being tackled here is more exactly the *permutation Flow-Shop problem*. In this problem, jobs must be scheduled in the same order on all the machines. The objective is to minimize the total completion time called *makespan* and noted C_{max}. This problem is NP-hard[8], and can be formulated as follows:

$$C_{max} = Max\{s_{iM} + p_{iM} | i \in [1 \ldots N]\} \qquad (15)$$

The application of the proposed approach to the Flow-Shop problem has been experimented on one of the instances proposed by [3]. More exactly, it is the sixth instance generated for problems of 50 jobs on 20 machines called Ta056[1]. This instance has never been solved optimally. The best known solution of Ta056 has a cost of 3681. It is found by [9] using a meta-heuristic (near-optimal resolution method).

5.2 The experimentation grid platform

The grid-based B&B algorithm has been experimented using the experimental grid detailed in Table 1 and illustrated by Figure 6. It is made up of 1889 processors belonging to 9 clusters. Three clusters belong to three different education departments of Universit de Lille1 (IUT-A, Polytech'Lille, IEEA-FIL), and six clusters belong to Grid'5000[2]. Grid'5000 is a nation-wide experimental grid composed by 9 clusters distributed over several French universities (Bordeaux, Lille, Rennes, Sophia, Toulouse, Orsay, Lyon, Grenoble and Nancy). Only the first six clusters are exploited in our experiments. Unlike the university machines, which are mono-processors, all the machines of Grid'5000 are bi-processors.

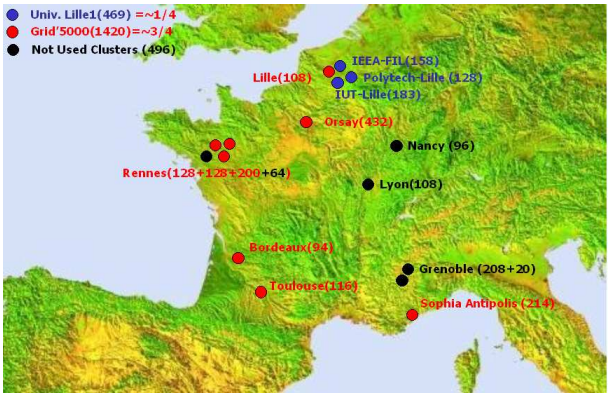

Figure 6. The experimental nation-wide grid

Inside all clusters, except that from IUT-A, the machines are interconnected by a Gigabit Ethernet. The IUT-A cluster uses a 100 Megabit Ethernet connection. The three university clusters are interconnected by a Gigabit connection. The Grid5000 clusters and the clusters of the university are interconnected, using the 2.5 Gigabit RENATER[3] nation-wide network.

5.3 Experimental results

The Taillard's instance Ta056 of the Flow-Shop problem is difficult, CPU time intensive and has never been solved exactly. Two runs have been performed, and at each time the optimal solution is found with a cost equal to 3679. This solution is the following schedule:

(14, 37, 3, 18, 8, 33, 11, 21, 42, 5, 13, 49, 50, 20, 28, 45, 43, 41, 46, 15, 24, 44, 40, 36, 39, 4, 16, 47, 17, 27, 1, 26, 10, 19, 32, 25, 30, 7, 2, 31, 23, 6, 48, 22, 29, 34, 9, 35, 38, 12).

In the first experiment, the algorithm has been initialized with an upper bound equal to 3681. The run lasted about

[1] http://www.eivd.ch/ina/Collaborateurs/etd/default.htm

[2] http://www.grid5000.fr

[3] http://www.renater.fr

CPU (GHz)	Domain	No.
P4 1.70	IEEA-FIL (Lille1)	24
P4 2.40		48
P4 2.80		59
P4 3.00		27
AMD 1.30	Polytech'Lille (Lille1)	14
Celeron 2.40		35
Celeron 0.80		14
Celeron 2.00		13
Celeron 2.20		28
P3 1.20		12
P4 3.20		12
P4 1.60	IUT-A(Lille1)	22
P4 2.00		18
P4 2.80		45
P4 2.66		57
P4 3.00		41
AMD 2.2	Bordeaux(Grid5000)	2x47
AMD 2.2	Lille(Grid5000)	2x54
Xeon 2.4	Rennes(Grid5000)	2x64
AMD 2.2		2x64
AMD 2.0		2x100
AMD 2.0	Sophia(Grid5000)	2x107
AMD 2.2	Toulouse(Grid5000)	2x58
AMD 2	Orsay(Grid5000)	2x216
Total		**1889**

Table 1. The computational pool

one month and three weeks using an average of 500 processors with a peak of 1245 processors. The goal of the second run is to get more statistics in order to learn more about the behavior of the proposed approach. In the second experiment, the algorithm has been initialized with an upper bound equal to 3680. Table 2 summarizes the most important statistics recorded during the second run, and Figure 7 plots the evolution of the number of exploited processors over the time.

As reported in Table 2, the resolution lasted *25 days* with an average of 328 processors, a maximum of $1,195$ available processors, and a cumulative computation time of about *22 years*. During the resolution, the B&B processes requested the load balancing mechanism approximately 130 thousand times. More than 2 million checkpointing operations were done by the B&B processes, while the coordinator does its checkpoint every 30 minutes. About 6 billion nodes were explored. In the proposed approach, some nodes can be explored by several B&B processes. This occurs mainly when an interval is duplicated. Table 2 shows that the rate of the redundant explored nodes is lower than 0.4%. The worker processors were exploited with an average of 97% while the farmer processor was exploited 1.7%.

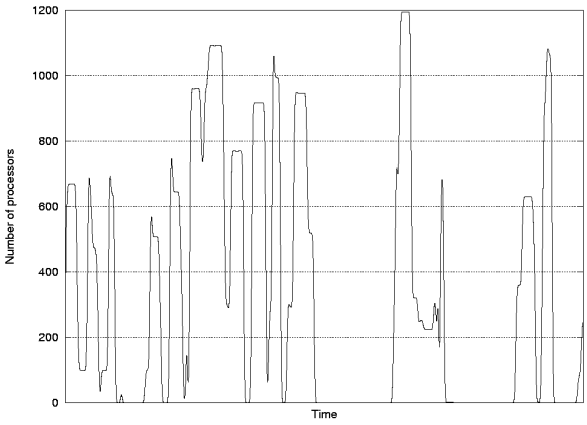

Figure 7. The evolution of the number of available processors

These two rates are good indicators on the parallel efficiency of this approach and its scalability. In the farmer-worker paradigm, a good approach must maximize the exploitation rate of the worker processors and must minimize the exploitation rate of the farmer processor.

Running wall clock time	25 days
Total cpu time	22 years
Average number of workers	328
Maximum number of workers	1,195
Worker CPU exploitation	97%
Coordinator CPU exploitation	1.7%
Checkpoint operations	4,094,176
Work allocations	129,958
Explored nodes	6,50874 e+12
Redundant nodes	0.39%

Table 2. The execution statistics

In terms of the exploited computational power, the second resolution of Ta056 ranks second. Table 3 compares and gives the positions of the most known success stories. More information about these stories can be found in [6] and [1]. To the best of our knowledge, Sw24978 is the only instance in the combinatorial optimization history which requires more computational power than Ta056. Sw24978 is an instance of the Traveling Salesman Problem. It consists in finding the shortest path to visit $24,978$ towns of Sweden. The cumulative time of the resolution of Sw24978 is equivalent to about 84 years of the computational power of an Intel Xenon 2.8 GHz.

Order	Problem	Instance	Description	Computation power
1	TSP	Sw24978	24,978 towns of Sweden	84 years/Intel Xeon 2.8 GHz
2	Flow-Shop	Ta056	50 jobs on 20 machines	22 years
3	TSP	D15112	15,112 towns of Germany	22 years/Compaq Alpha 500 MHz
4	QAP	Nug30		7 years/HP-C3000 400MHz
5	TSP	Usa13509	13,509 towns of USA	4 years

Table 3. The comparison of the most known resolutions

6 Conclusions and Future Works

Solving exactly large instances of combinatorial optimization problems requires a huge amount of computational resources. Parallel B&B algorithms based on the parallel exploration of the search tree have successfully been applied to solve these problems. However, experiments are often limited to few tens of processors. Designing and implementing B&B algorithms for a large scale computational grid is a great research challenge as several crucial issues must be tackled: dynamic load balancing of an irregular tree, fault tolerance due to the volatile nature of the grid, termination detection of asynchronous processes, global information sharing in a large scale context.

In this paper, we have proposed a new grid-enabled B&B algorithm together with new approaches allowing to efficiently tackle the problems quoted above. The approach consists in an efficient coding associated with the explored tree and work units (collections of nodes). Each node of the explored tree is assigned a node number. A work unit is delimited by two leaves of the explored tree, and thus represented by an interval whose beginning and end are the numbers associated with the two leaves. Such coding approach allows to optimize the communications induced by the load balancing, checkpointing-based fault tolerance and global information sharing operations. The termination detection is naturally ensured by the load balancing mechanism and no additional communication is required. The proposed approach includes two operators allowing to convert a collection of nodes into an interval and conversely.

The approach has been implemented following a large scale idle time stealing Farmer-Worker paradigm. It has been experimented on a Flow-Shop problem instance ($Ta056$) that has never been solved optimally. The instance has been solved within 25 days on 1889 processors, belonging to 9 Nation-wide clusters (administration domains). During the resolution, the worker processors were exploited with an average to 97% while the farmer processor was exploited 1.7%. These two rates are good indicators on the parallel efficiency of this approach and its scalability.

These experimental results must be "enforced" with a theoretical analysis. An analytical study is currently in progress in order to evaluate the complexity of the proposed approach and then compare it with the principal methods of the literature. It is also planned to use the approach with a peer to peer paradigm. This paradigm makes it possible to push far the scalability limits of the method. The objective is to exploit more and more processors and to solve more and more complex instances.

Acknowledgment

We would like to thank the technical staffs of the Grid'5000, IEEA-FIL, IUT-A and Polytech'Lille for making their clusters accessible and fully operational.

References

[1] D. Applegate, R. E. Bixby, V. Chvatal, and W. Cook. On the solution of travelling salesman problem. In *Documenta Mathemitica*, pages 645–656, 1998.

[2] V. D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Parallel and distributed branch-and-bound/A* algorithms. Technical Report 94/31, Laboratoire PRISM, Université de Versailles, 1994.

[3] E.Taillard. Banchmarks for basic scheduling problems. *European Journal of European Research*, pages 23:661–673, 1993.

[4] B. Gendron and T.G. Crainic. Parallel Branch and Bound Algorithms: Survey and Synthesis. *Operations Research*, 42:1042–1066, 1994.

[5] A. Iamnitchi and I. Foster. A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems. In *Intl. Conf. on Parallel Processing*, pages 4–14, 2000.

[6] Goux J.P., Anstreicher K.M., Brixius N.W., and Linderoth J. Solving large quadratic assignment problems on computational grids. In *Mathematical Programming*, pages 341–357, 2001.

[7] Nouredine Melab. *Contributions à la rsolution de problèmes d'optimisation combinatoire sur grilles de calcul*. PhD thesis, LIFL, USTL, Novembre 2005.

[8] Garey M.R., Johnson D.S., and Sethi R. The complexity of flow-shop and job-shop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

[9] R. Ruiz and T. Stutzle. A simple and effective iterated greedy algorithm for the flowshop scheduling problem. In *Technical report, submitted to EJOR*, 2004.