Grid'BnB: A Parallel Branch & Bound Framework for Grids

Alexandre di Costanzo[1], Laurent Baduel[2], Denis Caromel[1], and Satoshi Matsuoka[2]

[1] INRIA - I3S - CNRS - UNSA, France
[2] Tokyo Institute of Technology, Japan

Abstract. This article presents *Grid'BnB*, a parallel branch and bound framework for grids. Branch and bound (B&B) algorithms find optimal solutions of search problems and NP-hard optimization problems.
Grid'BnB is a Java framework that helps programmers to distribute problems over grids by hiding distribution issues. It is built over a master-worker approach and provides a transparent communication system among tasks. This work also introduces a new mechanism to localize computational nodes on the deployed grid. With this mechanism, we can determine if two nodes are on the same cluster. This mechanism is used in *Grid'BnB* to reduce inter-cluster communications. We run experiments on a nationwide grid. With this test bed, we analyze the behavior of a communicant application deployed on a large-scale grid that solves the flow-shop problem.

1 Introduction

Branch and bound (B&B) algorithm is a technique for solving search problems and NP-hard optimization problems. B&B aims to find the optimal solution and to prove that no ones are better. The algorithm splits the original problem into sub-problems of smaller size and then, for each sub-problem, the *objective function* computes the lower/upper bounds.

Because of the large size of handled problems (enumerations size and/or NP-hard class), finding an optimal solution for a problem can be impossible on a single machine. However, it is relatively easy to provide parallel implementations of B&B. Many previous work deal with parallel B&B as reported in [1].

Grids gather large amount of heterogeneous resources across geographically distributed sites to a single virtual organization. Resources are usually organized in clusters, which are managed by different administrative domains (labs, universities, etc.). Thanks to the huge number of resources grids provide, they seem to be well adapted for solving very large problems with B&B. Nevertheless, grids introduce new challenges such as deployment, heterogeneity, fault-tolerance, communication, and scalability.

We present *Grid'BnB*, a parallel B&B framework for grids. *Grid'BnB* aims to hide grid difficulties to users, especially fault-tolerance, communication, and scalability problems. The framework is built over a master-worker approach and provides a transparent communication system among tasks. Local communications between processes optimize the exploration of the problem. *Grid'BnB* is implemented in Java within the *ProActive* [2] Grid middleware. Our second contribution is an extension of the *ProActive* deployment mechanism to localize computational resources on grids. We detect locality at runtime providing the grid topology to applications in order to improve scalability and performance.

2 *Grid'BnB*: Branch and Bound Framework

2.1 Principles

Branch and bound is an algorithmic technique for solving optimization problems. B&B aims to solve problems by finding the optimal solution and by proving that no other ones are better. The original problem is split in sub-problems of smaller sizes. Then, the *objective function* [3] computes the lower/upper bounds for each sub-problem. Thus for an optimization problem the objective function determines how good a solution is. The upper bound is the worst value for the potential optimal solution, the lower bound is the best value. Therefore, if V is the optimal solution for a given problem and $f(x)$ the objective function, then *lower bound* $\leq f(V) \leq$ *upper bound*. Problems aim to minimize or maximize the objective function, in this paper we assume that problems minimize.

B&B organizes the problem as a tree, called *search tree*. The root node of this tree is the original problem and the rest of the tree is dynamically constructed by sequencing two operations: *branching* and *bounding*. Branching consists in recursively splitting the original problem in sub-problems. Each node of the tree is a sub-problem and has as ancestor a branched sub-problem. Thereby, the original problem is the parent of all sub-problems: it is named the root node. The second operation, bounding, computes for each tree node the lower/upper bounds. The entire tree maintains a *global upper bound* (GUB): this is the best upper bound of all nodes. Nodes with a lower bound higher than GUB are eliminated from the tree because branching these sub-problems will not lead to the optimal solution; this action is called *pruning*. Conceptually it is relatively easy to provide parallel implementations of B&B. Many previous work use the master-worker paradigm [4, 5].

The optimization problem is represented as a dynamic set of tasks. A first task (the root node of the search tree) is passed to the master and branched. The result is a set of sub-tasks to branch and to bound. Even in parallel generating and exploring the entire search tree leads to performance issues. Parallelism allows to branch and to bound a large number of feasible regions at the same time, but the pruning action seriously impacts the execution time. The efficiency of the pruning operation depends on the GUB updates. The more GUB is close to the optimal solution, the more sub-trees are pruned. The GUB's updates are determined by how the tree is generated and explored. Therefore, a framework for grid B&B has to propose several exploration strategies such as *breadth-first search* or *depth-first search* (more details in Section 2.2).

Other issues related to pruning in grids are concurrency and scalability. All workers must share the GUB as a common global data. GUB has multiple parallel accesses in read (get the value) and write (set the value). A solution for sharing GUB is to maintain a local copy on all workers and when a better upper bound than GUB is found the worker broadcasts the new value to others.

In addition, for grid environments, which are composed of numerous heterogeneous machines and which are managed by different administrative domains, the probability of having faulted nodes during an execution is not negligible. Therefore, a B&B for grids has to manage fault-tolerance. A solution may for instance be that the master handles worker failures and the state of the search tree is frequently saved in a file.

2.2 Architecture

Grids lead to scalability issues owing to the large number of resources. Aida and al. [6] show that running a parallel B&B application based on a hierarchical master-worker architecture scales on grids. For that reason we choose to provide *Grid'BnB* with a hierarchical master-worker. Our hierarchical master-worker is composed of four kind of entities: *master*, *sub-master*, *worker*, and *leader*.

The *master* is the unique entry point: it receives the entire problem to solve as a single task (it is the *root task*). At the end, once the optimal solution is found, the master returns the solution to the user. Thus, the master is responsible for branching the root task, managing task allocation to sub-masters and/or workers, and handling failures. *Sub-masters* are intermediary entities whose role is to ensure scalability. They are hierarchically organized and forward tasks from the master to workers and vice versa by returning results to the master (or their sub-master parent). The role of the *workers* is to execute tasks. They are also the link between the tasks and the master. Indeed when a task does branching, sub-tasks are created into the worker that sent them to the master for remote allocation. *Leader* is specific role for workers. Leaders are in charge of forwarding messages between clusters (more details further).

Users who want to solve problems have to implement the task interface provided by the *Grid'BnB* API. Figure 1 shows the task interface and the worker interface implemented by the framework. The task interface contains two fields: `GUB` is a local copy of the global upper bound; and `worker` is a reference on the associated local process, handling the task execution. The objective function that users have to implement is `explore`. The result of this method must be the optimal solution for the feasible region represented by the task. `V` is a Java 1.5 generics: the user defines the real type. The branching operation is implemented by the `split` method. In order to not always send to the master all branched sub-problems, the *Grid'BnB* framework provides, via the `worker` field, the method `availableWorkers`, which allows users to check how many workers are currently available. Depending on the result of this method, users can decide to do branching and to locally continue the exploration of the sub-problem. To help users to structure their codes, we introduced two methods to initialize bounds: `initLowerBound` and `initUpperBound`. These two methods

```
public abstract class Task<V> {
    protected V GUB;
    protected Worker worker;
    public abstract V explore(Object[] params);
    public abstract ArrayList<?extends Task<V>> split();
    public abstract void initLowerBound();
    public abstract void initUpperBound();
    public abstract V gather(V[] values);    }

public interface Worker {
        public int availableWorkers();    }
```

Fig. 1: The task and worker Java interfaces.

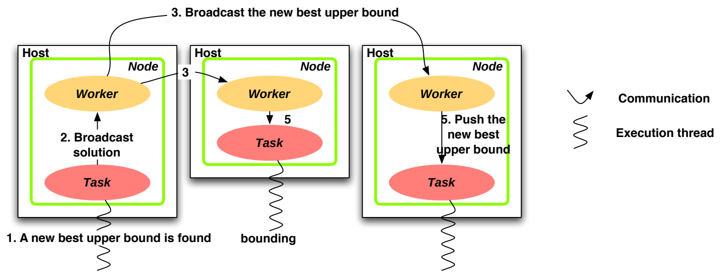
are called for each task just before the objective function `explore`, and they are not mandatory. The last method to implement is `gather`: the (sub-)master calls this method when all its tasks are solved. The method returns the best results from all tasks, *i.e.* the optimal solution.

The root task is passed to the master that performs the first branching. Then when a task is allocated to a worker that starts to explore it. As soon as a worker is available, a new task can be allocated. The worker starts by heuristic methods to initialize lower/upper bounds for the current feasible region, then it calls the objective function. Within the objective function, the user can decide whenever to branch the current region with the help of the `availableWorkers` method, which returns the current number of free workers.

The master and the search tree strategy handle task allocation; thereby the master works as a queue for task scheduling. The exploration algorithm of the search tree is important regarding performances. Therefore, *Grid'BnB* allows users to choose adapted algorithms to solve their problems. We propose four algorithms: *breadth-first search* explores the tree in larger, *depth-first search* explores all branches one by one, *first-in-first-out* (FIFO) explores the tree following the order tasks have been sent to the master, and *priority* explores in priority branches that updated the GUB the most frequently. If none of those algorithms satisfy the problem, users can implement their owns.

The tasks produce new GUB candidates while they are computed by workers. The GUB must be available to all tasks to prune the maximum of none promising branches of the search tree. The strategy for sharing GUB is to use a local copy of GUB on all workers and to broadcast updated value. Figure 2 shows the process of updating GUB when a worker finds a new better upper bound. To be efficient, a B&B framework has to broadcast the GUB as fast as possible. With a large number of workers, directly broadcasting GUB to every worker cannot scale. For that reason *Grid'BnB* organizes workers in groups.

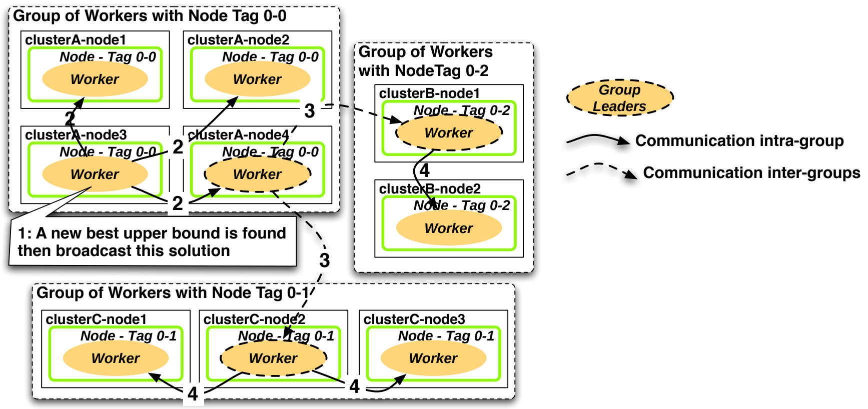
Groups are sets of workers, which can efficiently broadcast GUB between them. The master is in charge of building groups. Thus, the main criterion to put workers in the same group is their localization on the same cluster. Clusters



3. Broadcast the new best upper bound
Host
Node
Worker
2. Broadcast solution
Task
1. A new best upper bound is found
3
5
bounding
5. Push the new best upper bound
Communication
Execution thread

Fig. 2: Update the best global upper bound.

usually provide a high performance environment for communication. The master elects a worker as *leader* in each group. This leader has a reference to all other group leaders. When a leader receives a communication from outside its group, it broadcasts the communication to its group. Inversely when the leader receives a communication from a member of its group, it broadcasts the communication to the other leaders but only if the new upper bound is better than its own GUB value. Figure 3 shows an example of broadcasting GUB between groups.



Group of Workers with Node Tag 0-0
clusterA-node1
clusterA-node2
clusterA-node3
clusterA-node4
Node - Tag 0-0
Worker
Group of Workers with NodeTag 0-2
clusterB-node1
clusterB-node2
Node - Tag 0-2
1: A new best upper bound is found then broadcast this solution
Group of Workers with Node Tag 0-1
clusterC-node1
clusterC-node2
clusterC-node3
Node - Tag 0-1
Group Leaders
Communication intra-group
Communication inter-groups
2
3
4

Fig. 3: Broadcasting solution between groups of workers.

Within the user code, errors can occurs, such as uncaught exceptions. Workers handle user exceptions. When a worker catches an exception, it forwards it to the master, and then the master stops the whole computation and returns the exception to the user.

The last feature of *Grid'BnB* is the fault-tolerance. Fault-tolerance is a real issue of grid environments; the large number of resources that are distributed on different administration domains implies a high probability of faults, such as hardware failures, networks down time, or maintenance.

Master and sub-masters hierarchically manage infrastructure failures, such as host failures. The monitoring consists of frequently pinging entities. When the ping call fails (communication timeout, network errors, etc.), the remote host is considered as unreachable and down. In that case, the master re-allocates the task to an available worker. If for the same task several results are returned to the master (worker considered down for network problem and come back), only the first one is kept, others are flushed. Masters handle the fault of their sub-masters: if a sub-master does not answer to a ping call, the master chooses a free worker and re-instantiates it as a sub-master. Masters also handle the fault of leaders; the master frequently pings leaders. When a leader is unreachable, the master elects a new leader in the group.

The master must be deployed on a stable machine, because it is at the top of the monitoring hierarchy. As opposed to sub-masters and workers, master host failures cannot be dynamically handled by the framework but require users intervention. The status of the current execution (GUB and all tasks) is frequently saved on disk. Thus for long-running problem, if the master node faults the user can restart the solving at a recent state of the execution.

Grid'BnB provides a high level-programming model for solving problems with parallel B&B. From the users points of view, the framework handles all issues related to distribution/parallelism and fault-tolerance.

2.3 Implementation

Grid'BnB is designed for grids and is implemented with Java, which allows to use a large kinds of resources, operating systems, and machine architectures. More of Java, *Grid'BnB* is implemented within the *ProActive* Grid middleware.

ProActive [2] is a Java library for concurrent, distributed and mobile computing.*ProActive* features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronization mechanisms, and migration of active objects with pending calls. As *ProActive* is built on top of standard Java APIs, neither does it require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified Java Virtual Machine (JVM). A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Method calls sent to active objects are asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as wait-by-necessity.*ProActive* provides typed *group communication*, an important feature for high-performance and grid computing. The group communication [7] extends the *ProActive* elementary mechanism for asynchronous remote method invocation and automatic futures.

In *Grid'BnB*, master, sub-masters, and workers are active objects. Each active object serves remote calls in FIFO order. Master manages futures on current executing tasks. Then, groups of workers are *ProActive* groups. Leaders are also member of a *ProActive* group. Thereby, hierarchical *ProActive* groups represent workers. A hierarchical group is indeed a group of groups. Finally, to optimize communication between workers to solve more rapidly problems, the management of workers in groups lay to the *ProActive* deployment framework. *ProActive* features a system for the deployment of applications on grids. The next section explains the deployment mechanism and how we improved it to manage organization of workers in groups of communications.

3 Grid Node Localization

The key principle of *ProActive* deployment [8] is to eliminate from the source code the following elements: *machine names*, *creation protocols*, *registry*, and *lookup protocols*. It allows to deploy any application anywhere without modifying the source code. The deployment sites are called *nodes* and correspond to JVMs, which host active objects. The deployment framework uses *Virtual Nodes* (VNs). VNs are the deployment abstractions for the applications; they are defined in the program source and after activation they are mapped to a set of nodes. The deployment framework relies on XML descriptors. They are composed of two parts: mapping and infrastructure. The VN, which is the deployment abstraction for applications, is mapped to nodes in the deployment descriptors, and nodes are mapped to physical resources, *i.e.* to the infrastructure. Nodes are created using remote connection and creation protocols. Deployment descriptors allow combining these protocols in order to seamlessly create remote JVMs.

In Section 2 we proposed to organize workers in groups for optimizing communication. The selection criterion for group acceptance for a worker is its physical localization on a cluster. Therefore, the node localization on the grid is important for an efficient implementation of our *Grid'BnB* framework. The *ProActive* deployment framework provides a high-level abstraction of the underlying physical infrastructure. Once deployed, the application cannot easily access to the topology of the physical infrastructure. For instance, programmers have to compare node addresses for determining if two nodes are deployed on the same cluster. Nevertheless, two nodes may have the same sub-net address on different clusters, with network of NATs. Hence, programmers may use metrics, such as latency, to determine if nodes are "close". Consequently, organizing workers in group by clusters and optimizing communication between clusters is a very difficult and complicated task. For that reason we introduced a new mechanism in the *ProActive* deployment framework to identify nodes, which are deployed on the same cluster or even on the same machine.

The creation of a node is the result of a deployment graph (a directed acyclic graph: DAG) with connection protocols. This deployment graph is specified within the XML deployment descriptor. Our deployment node tagging mechanism aims to tag nodes in regard of the deployment graph on which they are mapped in the deployment descriptor. This tag will allow the application to organize groups in regard to the deployment process that created nodes. With this mechanism, all deployed nodes are tagged with an identifier at deployment time. Nodes that have the same tag value have been deployed by the same deployment process. As a result, they have a high probability to be located in the same the same local network.

Figure 4 shows the process of tagging nodes. The tag is built by a concatenation of identifiers at each level of the deployment graph. At the beginning of the deployment, a new tag is instantiated for each virtual node. For leaf nodes of the DAG, which are JVM creations, no identifier is added. Therefore, all nodes deployed with the same path in the DAG have the same tag.

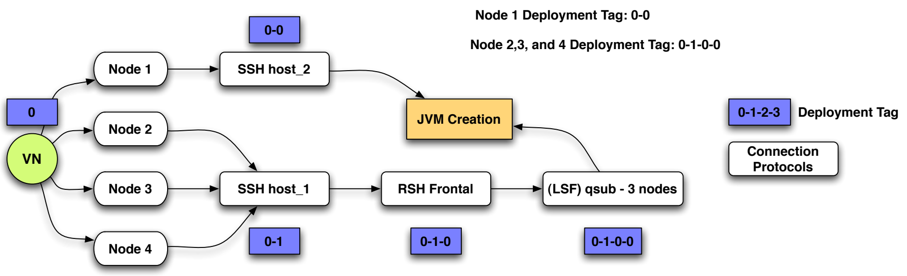


Fig. 4: Deployment tag mechanism.

The tag is an abstraction of the physical infrastructure; it provides more information about how nodes have been deployed. It is now possible to know at the application level that the same deployment graph has deployed two nodes. The deployment tag can be used for instance by applications to optimize communication between nodes or to do data localization. More especially *Grid'BnB* uses the deployment tag to dynamically organize worker communications between clusters. Figure 3 shows the deployment result of a single virtual node on three clusters. The deployment has returned nine nodes: four nodes on *clusterA*, two on *clusterB*, and three on *clusterC*. The node tag mechanism has tagged the nodes *0-0* on *clusterA*, *0-1* on *clusterC*, and *0-2* on *clusterB*. Tags are finally used to organize workers in groups of communication to optimize communication between clusters.

4 Experiments

4.1 The Flow-Shop Problem

Flow-shop is a NP-complete permutation optimization problem. The flow-shop problem consists in finding the optimal schedule of n jobs on m machines. The set of jobs is represented by $J = \{j_1, j_2, \ldots j_n\}$, each j_i is a set of operations $j_i = \{o_{i1}, o_{i2}, \ldots o_{im}\}$ where o_{im} is the time taken on machine m and the set of machines is represented by $M = \{m_1, m_2, \ldots m_m\}$.

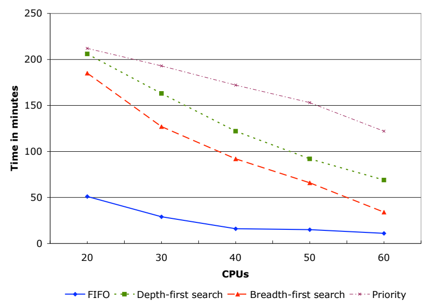
The operation o_{ij} must be processed by the machine m_j. The sequence of jobs are the same on every machines, e.g. if j_3 is treated in position 2 on the first machine, j_3 is also executed in position 2 on all machines.

We consider the mono-objective case, which aims to minimize the overall completion time of all jobs, *i.e. makespan*. The makespan is the total execution time of a complete sequence of jobs. Thus, the mono-objective goal is to find the sequence of jobs that takes the shortest time to complete.

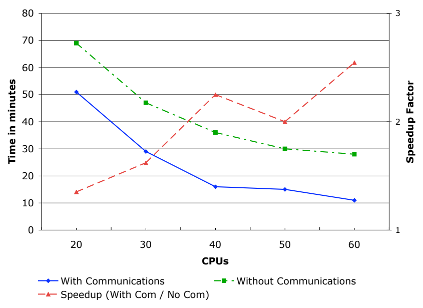
4.2 Single Cluster Experiments

These experiments aim to choose the best search strategy and to determine the impact on performances of dynamically sharing GUB with communications. We use a 32 nodes cluster at INRIA Sophia lab, powered by dual-processors AMD Opteron with a speed of 2 GHz and connected via Gigabit Ethernet.

Figure 5a shows results of applying different search strategies (described in section 2.2) to flow-shop. The selected instance of flow-shop is 16 jobs / 20 machines. Results show that FIFO is the fastest for all those experiments; the speedup between 20 CPUs and 60 CPUs is 4.63. This is a super linear speedup owing to increase the total of CPUs allows a larger generation of the search tree in parallel and thereby, improving the GUB faster to prune more branches. Breadth-first search scales with a very good speedup, the speedup between 20 CPUs and 60 CPUs is 5.44, also super linear. The high speedup is normal because more breadth-first search is deployed on nodes the more the tree is explored in parallel. Depth-first search speedup is linear, 3.00, and for priority search the speedup is 1.73. The speedup is particularly high with all these experiments, because with 60 CPUs the chosen flow-shop instance can be widely explored in parallel whatever the search strategy. The built search tree rapidly provides the best solution as upper bound, thus each process can delete many branches.



(a) Benchmarking search tree strategies



(b) Dynamic GUB sharing vs. no sharing

Fig. 5: Single cluster experiments: flow-shop $n = 16$, $m = 20$.

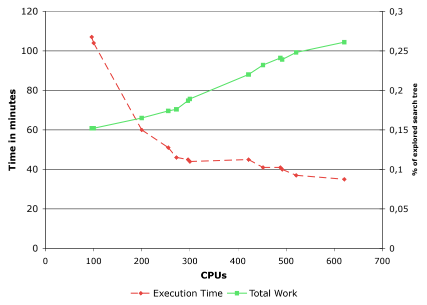
With the same instance of flow-shop and with the FIFO strategy, we now benchmark the impact of dynamically sharing GUB with communications. We benchmark flow-shop with communications between workers for sharing GUB and without dynamically sharing GUB between workers (no communication). In the case of no communication, the master keeps the GUB up-to-date with all results from computed tasks; and when a task is allocated to a worker by the master, it sets the current GUB value to the task. Figure 5b shows the results. Using communications to share GUB improves performance. But the speedup, $\frac{T\ No\ Communication}{T\ Communications}$, is lower for 50 CPUs than 40 CPUs, this decrease comes from the fact that since 40 CPUs this flow-shop instance has enough CPUs to explore the whole tree in parallel, *i.e.* it is the optimal deployment.

These experiments on a single cluster show that dynamically sharing GUB with communications between workers improve execution time, and that choosing the right search strategy considerably affects performances.

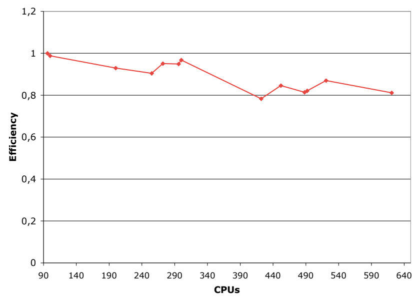
4.3 Large Scale Experiments

In order to experiment *Grid'BnB* on grids, we used a large-scale nationwide infrastructure for grid research, *Grid'5000* (G5K) [9]. The G5K project aims at building a highly reconfigurable, controllable and monitorable experimental grid platform gathering 9 sites geographically distributed in France currently featuring a total of about 3000 CPUs. G5K is composed of a large number of machines, which have different kinds of CPUs (dual-core architecture, AMD Opteron 64 bits, PowerPC G5 64 bits, Intel Itanium 2 64 bits, Intel Xeon 64 bits), of operating systems (Debian, Fedora Core 3 & 4, MacOs X, *etc.*), of supported JVMs (Sun 1.5 64 bits and 32 bits, and Apple 1.4.2), and of network connection (Gigabit Ethernet and Myrinet).

Grid experiments run with the same implementation of flow-shop, as previous single cluster experiments. The instance of flow-shop is now a larger problem: 17 jobs / 17 machines. The search tree strategy is FIFO and communications are used to dynamically share GUB. Results of experiments with G5K are summarized in Figure 6a and Table 1.



(a) Results



(b) Efficiency

Fig. 6: Large scale experiments: flow-shop $n = 17$, $m = 17$.

Table 1: Large scale experiments results.

CPUs	**Sites**	**Execution time**	**Tasks**	**% of explored search tree**	**Gathered time**
100	1	104 m	1567	0.152%	167 h
200	1	60 m	2515	0.165%	181 h
300	2	44 m	3729	0.189%	196 h
492	4	40 m	5447	0.239%	251 h
621	5	35 m	6968	0.261%	267 h

The broken line in Figure 6a shows that the execution time strongly decreases until 272 CPUs, the speedup between 96 CPUs and 272 CPUs is 2.32. From 272 to 621 CPUs the execution time is almost constant, the speedup between 272 and 621 CPUs is 1.31. Then, the global speedup, between 96 and 621 CPUs, is 3.01. Our *Grid'BnB* flow-shop scales well up to 272 (close to linear speedup). However, for more than 272 CPUs, the execution time decreases slowly. Nevertheless, the solid line shows the percentage of branches explored in the search tree, *i.e.* total number of tested permutations, this line increases with the number of CPUs. This line is indeed the total work done by the computation.

Figure 6b shows the efficiency E, this value estimates how CPUs are utilized for the computation. Values of E are between 0 and 1, a single-processor computation and linear speedup have $E = 1$. Here, we consider the execution time (T) efficiency corrected with the work (W: total number of tested permutations) because *Grid'BnB* computes more work with increasing CPUs. Thus, the efficiency for n CPUs: $E_n = \frac{T_n/T_{96} * W_{96}/W_n}{96/n}$. The figure shows that between 96 and 300 CPUs, E is close to 1 (0.9), which is very good. However, for 422 and more, E decreases to 0.8, it is still a good value. This decrease can be explain by the fact that for experiments with less than 422 CPUs are done on 1 or 2 grid sites and for 422 and more 3 up to 5 sites nationally-distributed. In addition, grid sites are heterogeneous in regards of CPUs power and inter-site network connections.

Experiments on single cluster and large scale grid show that it is better to use communications to dynamically share GUB, and that it is important for users to choose the adapted search tree strategy to their problems to solve. Large experiments also show that *Grid'BnB* can be used on grid environments, we deploy flow-shop on a nationwide grid of five clusters gathering a 621 CPUs.

5 Related Work

Branch and Bound: Many work reported by the survey in [1] are based on a centralized approach with a single manager, which maintains the whole tree and hands out tasks to workers. This kind of approach clearly does not scale for grid environments.Aida and al. [5] present a solution based on hierarchical master-worker to solve scalability issues. Workers do branching, bounding, and pruning on sub-problems, which are represented by tasks. The supervisor handles the sharing of the best current upper bound. Supervisor and sub-masters gather results from workers and are in charge to hierarchically update the best upper bound on all workers. We show in section 4.2 that using dynamic communications rather than using the master to share GUB allows to complete the computation faster. In [6] Aida and Osumi propose a study of their hierarchical master-worker framework implemented using GridRPC middleware [10] and Ninf-G [11]. The authors discuss the granularity of tasks, notably when tasks are fine-grain the communication overhead is too high compared to the computation of tasks. Thereby, *Grid'BnB* introduces a method to check how many workers are available. This method helps users to program tasks and to dynamically determine the most appropriate granularity of the tasks.

Iamnitchi and Foster [12] proposes a solution to do B&B over grids that differs from *Grid'BnB* and others because it does not base on master-worker paradigm, but on a decentralized architecture that manages resources through a membership protocol. Each process maintains a pool of problems to solve. When the pool is empty, the process asks for work to other processes. The sharing of the best upper bound is handled by circulating a message among processes. The fault-tolerance issue is addressed by propagating all completed sub-problem to all processes. This approach may result in significant overhead, in terms of both duplicated work and messages.

ParadisEO [13] is an open source framework for flexible parallel and distributed design of hybrid meta-heuristics. Moreover, it supplies different natural hybridization mechanisms mainly for meta-heuristics including evolutionary algorithms and local search methods. All these mechanisms can be used for solving optimization problem. Like *Grid'BnB*, the grid version of ParadisEO is based on the master-worker paradigm. ParadisEO splits the optimization problem in tasks. Then, the task allocation is handled by MW [4], a tool for scheduling master-worker applications over Condor [14], which is a grid resource manager. Unlike *Grid'BnB*, ParadisEO just provides mechanisms for searching algorithms.

Skeletons: The common architecture used for B&B on grids is "master-worker". For parallel programming, the master-worker pattern is called *farm skeleton* [15]. Muskel [16] is a Java skeleton framework for grids that provide farm. Skeleton frameworks usually provide task allocation and fault-tolerance. Thus, skeletons seem well adapted for implementing B&B for grids. Like *Grid'BnB* users just have to focus on the implementation of the problem to solve all other issue related to grid and tasks managing are handled by the framework. However in farm skeletons tasks cannot share data, such as a global upper bound to prune more promising branches of the search tree to find more rapidly the optimal solution. In addition, another skeleton that fits B&B algorithm is the *divide-and-conquer skeleton*. This skeleton allows to dynamically split task, *i.e.* branching, but like farm it is not possible to share the global upper bound between task.

Divide-and-Conquer: Conceptually, B&B technique fits the divide-and-conquer paradigm. The search tree can be divided into sub-trees, and each sub-tree is then assigned to an available computational resource. This is done recursively until the task is small enough to be solved directly.

Satin [17] is a system for divide-and-conquer programming on grid platforms. Satin express divide-and-conquer parallelism entirely in the Java language itself, without requiring any new language constructs. Satin uses so-called marker interfaces to indicate that certain method invocations need to be considered for parallel execution, called spawned. A mechanism is also needed to synchronize with spawned method invocations. Satin can be used directly to implement B&B. Thus, users can mark branching methods to be executed in parallel. Like *Grid'BnB*, Satin is in charge to distribute sub-problems through grids. But unlike our framework, Satin does not provide any mechanisms for sharing global upper bound and more generally no mechanism for communication between parallel executed sub-problems.

6 Conclusion and Perspectives

We described *Grid'BnB* a parallel B&B framework for grids. *Grid'BnB* provides a framework to help users to solve optimization problems hiding grids, parallelism, and distribution related issues. It is based on a hierarchical master-worker architecture enhanced with communications between processes to share the best global upper bound thus exploring less parts of the search tree and decreasing the execution time. Because grids provide a large-scale parallel environment, we propose to organize workers in groups of communications. Groups reflect grid topology. This feature aims to optimize inter-cluster communications and to update more rapidly the global upper bound on all processes. *Grid'BnB* proposes different search tree algorithms to help users to choose the most adapted one for the problem to solve. Finally, the framework allows fault-tolerance for long-running executions. In addition, we introduced a new mechanism, *deployment node tagging*, to localize deployed nodes on grids. The deployment node tagging allows *Grid'BnB* to identify nodes, which are on the same cluster, and to optimize group communications between processes. This mechanism is integrated in the deployment framework of the *ProActive* grid middleware.Experiments show that *Grid'BnB* scales on a real nationwide grid, such as Grid'5000. We were able to deploy a permutation optimization problem, flow-shop, on up to 621 CPUs distributed on five sites.

In future work, we plan to improve our flow-shop implementation with a better objective function, such as the technique proposed by Lageweg [18]. Likewise, we want to run larger scale experiments on a worldwide grid, by mixing clusters located in France and Japan. We believe that *Grid'BnB* can used for more than B&B. Without modification of the framework it may be used to do divide-and-conquer or as farm skeleton. *Grid'BnB* is framework for parallel programming that targets all embarrassingly parallel problems.

References

1. Gendron, B., Crainic, T.: Parallel Branch-And-Bound Algorithms: Survey and Synthesis. Operations Research **42**(6) (1994) 1042–1066

2. Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: Proactive: an integrated platform for programming and running applications on grids and p2p systems. Computational Methods in Science and Technology **12**(1) (2006) 69–77
3. Atallah, M.: Algorithms and theory of computation handbook. CRC Press (1999)
4. Goux, J., Kulkarni, S., Linderoth, J., Yoder, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid. Proc. 9th IEEE Symp. on High Performance Distributed Computing (2000)
5. Aida, K., Natsume, W., Futakata, Y.: Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on (2003) 156–163
6. Aida, K., Osumi, T.: A Case Study in Running a Parallel Branch and Bound Application on the Grid. Proc. IEEE/IPSJ The 2005 Symposium on Applications & the Internet (SAINT2005) (2005) 164–173
7. Baduel, L., Baude, F., Caromel, D.: Efficient, Flexible, and Typed Group Communications in Java. In: Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, ACM Press (2002) 28–36 ISBN 1-58113-559-8.
8. Baude, F., Caromel, D., Mestre, L., Huet, F., Vayssière, J.: Interactive and descriptor-based deployment of object-oriented grid applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, IEEE Computer Society (2002) 93–102
9. Cappello, F., Caron, E., Dayde, M., Desprez, F., Jeannot, E., Jegou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Richard, O.: Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In: Grid'2005 Workshop, Seattle, USA, IEEE/ACM (2005) to appear.
10. Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing. 3rd International Workshop on Grid Computing, November (2002)
11. Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. Journal of Grid Computing **1**(1) (2003) 41–51
12. Iamnitchi, A., Foster, I.: A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems. 29th International Conference on Parallel Processing (ICPP), Toronto, Canada, August (2000) 21–24
13. Cahon, S., Talbi, E.G., Melab, N.: Paradiseo: A framework for parallel and distributed metaheuristics. In: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society (2003) 144.1
14. Litzkow, M., Livny, M., Mutka, M.: Condor - a hunter of idle workstations. In: Proc. of the 8th International Conference of Distributed Computing Systems. (1988)
15. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge, MA, USA (1991)
16. Danelutto, M.: Qos in parallel programming through application managers. In: PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), Washington, DC, USA, IEEE Computer Society (2005) 282–289
17. van Nieuwpoort, R.V., Maassen, J., a nd Thilo Kielmann, G.W., Bal, H.E.: Satin: Simple and efficient java-based grid programming. Accepted for publication in Journal of Parallel and Distribute d Computing Practices (2004)
18. Lageweg, B., Lenstra, J., Kan, A.: A General Bounding Scheme for the Permutation Flow-Shop Problem. Operations Research **26**(1) (1978) 53–67