Parallel Skeletons for Tabu Search Method*

Maria J. Blesa Lluís Hernàndez Fatos Xhafa

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Campus Nord C6
E-08034, Barcelona, Spain

E-mail: {mjblesa,ba-pfc1,fatos}@lsi.upc.es

December 22, 2000

Abstract

In this paper we present two parallel skeltons for Tabu Search method –a well known meta-heuristic designed for finding approximate solution of combinatorial optimization problems. Our parallel skeletons are designed and implemented using the generic parallel programming paradigm. The first skeleton is based on independent runs while the second in the classical master-slave model. Our starting point is the design and implementation of a sequential skeleton that is used later as basis for the two parallel skeletons. Both skeletons provide the user with the followings: (a) permit to obtain parallel implementations of tabu search method for concrete combinatorial optimization problems from existing sequential implementations; (b) there is no need for the user to know neither parallel programming nor communication libraries; (c) the parallel implementation for a concrete problem is obtained automatically from a sequential implementation of the problem. The skeletons, however, require from the user a sequential instantiation of tabu search method for the problem at hand. The skeletons are implemented in C++ using MPI as a communication library and offer several properties such as a genericity, flexibility, component reuse, robustness and time savings, mainly due to the generic programming and object oriented programming paradigms. We have instantiated the two skeletons for the 0-1 Multidimensional Knapsack problem and have observed the claimed properties of the skeletons.

Keywords: *Optimization, Tabu Search, Parallelism, Generic Programming, Components.*

1 Introduction

Many interesting combinatorial optimization problems are shown *NP-hard* [10] and hence unlikely to be solvable within a reasonable amount of time. Heuristics have proved a good alternative to cope in practice with such hard problems. Among these heuristics there is also the tabu search (TS), a meta-heuristic designed to find approximate solution of combinatorial optimization problems. TS was introduced by Glover [11] (for its current form see [12]) and has been used to sub-optimally solve a wide range of important optimization problems from theory and practice. TS has been extensively studied in both sequential and parallel settings, actually, there exists an abundant literature for the method dealing with theoretical aspects and implementation issues.

In the *sequential setting* considerable efforts have been done by the researchers and as a result there is a long list of problems to which the TS has been successfully applied such as scheduling problems (e.g. [18, 31, 6, 24]), graph problems (e.g. k-Cardinality Tree [15]), resource allocations (e.g. [25, 28]), layout problems (e.g. [26, 9, 17]), to name a few. After a careful and almost exhaustive revision of such implementations, we have observed that all these implementations

*This research was partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and the CICYT project TIC1999-0754-C03.

are *ad hoc* and quite dependable on the problem at hand. This approach has, at least, two drawbacks. First, one has to implement the method from scratch for any problem of interest, and, second, it is difficult to introduce even small changes in the code since it would require the modification of most of the implementation. We remark here that different implementations of the tabu search for the same problem can be generated by varying the definition and structure of several underlying entities of the search, so it would be quite interesting to generate new implementations for a problem from existing ones with as many few changes as possible.

Generic programming and reuse of components have already proved to be very important to the software process in general. In combinatorial optimization we are often encountered with *methods* for sub-optimally solving a large number of optimization problems and such methods apply almost in the same way to all problems. Consider, for example, the tabu search method, the ingredients of such method are the same for any problem to which one would like to apply the method. It is, therefore, quite interesting to have a *generic program* or a kind of *template* for tabu search from which one could derive instantiations for any problem of interest. Many authors have pointed out that TS may be viewed as an engineering design approach. In this spirit, we dealt with the design and (sequential) implementation issues of tabu search from a generic programming paradigm[1]. Our objective was twofold: in one hand to obtain a powerful engine from which sequential implementations of tabu search could be obtained, and, more importantly, to use the sequential implementation in order to automatically obtain parallel implementations, as we will explain later in detail. These properties are achieved by a careful design of the skeleton identifying the common entities of the TS method.

Tabu Search have also been considered in the *parallel setting* to exploit the advantages of the parallelism. Parallelism permits, in one hand, to use more resources to obtain good solutions in reasonable computing times, and in the other, parallel versions of heuristics tends to yield more robust implementations than the sequential ones. Those ideas have been investigated both in the context of TS and that of other meta-heuristics. In [5] the authors present a whole taxonomy of parallel TS strategies while in [4] they present fundamental ideas to design parallel strategies for meta-heuristics in a general context. Those ideas are applicable to a wide range of problems, yet the existing parallel implementations for different problems (e.g. [29, 7, 23]) are *ad hoc* implementations. So, while from a theoretic point of view there is almost clear how to exploit parallelism in the TS method in general, in practice the known parallel strategies are applied using specific knowledge of the problem at hand. This fact, in a sense, constitutes a gap between generic parallel programming and concrete parallel implementations for TS.

In this paper our objective is to reduce the above mentioned gap. To this aim we have designed and implemented in C++ using MPI Library [14] two parallel skeletons for TS using the paradigm of generic parallel programming. The concept of generic programming has turned out to be very useful to parallel programming in a manner that allows good expressibility, reuse, robustness yet maintaining the efficiency. Parallel generic programming paradigm has been used in several other contexts (e.g. [20, 27, 16, 19, 13, 30]). Generic programming clearly separates algorithms from the data structures due to a powerful abstraction. This paradigm, applied to the TS method, allows us to obtain a general enough procedure that permits the instantiation of the TS method for any optimization problem.

Our first skeleton exploits the *independent run* parallelism. If the user disposes of a sequential implementation of TS for a given problem(via our sequential TS skeleton), then he can automatically obtain a parallel implementation for the problem based on independent runs. In this model, at the beginning, there is a distinguished processor that distributes the same data to the processors, then each processor runs the TS independently, and at the end, the distinguished processor receives the solutions found by different processors and reports the best solution as well as other statistical information. This kind of parallelism though trivial is especially important in the context of TS since a good implementation for TS method requires extensive experimenting so as to identify adequate parametres that guide the search in the right way. Such a fine tuning results, most of the times, time consuming, hence the independent runs parallelism permits a better use of resources and clear time savings.

[1] We will give an overview of the obtained sequential skeleton in Section 2, for a complete exposition the reader is referred to [2]

The second skeleton is based on a classical master-slave model. In this model, there is a master processor that controls the search. At the beginning, the master sends the same data to the rest of processors and waits until they finish a *tabu search iteration* consisting in *computing* a new solution in the solution space through the exploration of neighborhood of the current solution. Once each processor reports its new solution, the master selects the best among all of them, sends it back to the processors and the search is relaunched from this new solution. The process is repeated until a stopping criteria is reached. We notice that in this model different processors may follow different trajectories while exploring the neighborhood of a given solution.

Importantly, the skeletons offer an automatic parallel implementation of the TS method for any given problem from an existing sequential implementation. For instance, since we dispose of a sequential implementation of TS for the 0-1 Multidimensional Knapsack obtained via the sequential skeleton, we have automatically obtained two parallel implementations of TS for the problem, by recombining our parallel skeletons and the sequential implementation fro this problem. We consider this very important since there is no need for the user to know neither parallel programming techniques nor communication libraries (MPI, in our case). Interestingly, we have observed from the experimental results that for both skeletons, in spite of being generic, there is no loose in efficiency. We have applied our parallel skeletons to 0-1 Multidimensional Knapsack and have executed these parallel programs over a cluster of PC at our department (www.lsi.upc.es/~mallba/BA-Cluster/). Our results are quite satisfactory and promising.

The paper is organized as follows. In Section 2 we give an overview on TS and present the sequential skeleton. Next, in Section 3.1 we give the parallel skeleton based on independent runs, show how to instantiate it for a concrete problem –the 0-1 Multidimensional Knapsack– and present some experimental results. The parallel skeleton based on master-slave implementation is given in Section 3.2 together with the instantiation for the 0-1 Multidimensional Knapsack and some experimental results. We conclude in Section 4 with some remarks and outline future work.

2 Preliminaries

2.1 Overview on Tabu Search

Tabu Search belongs to the family of local search algorithms but here the search is done in a *guided* way in order to overcome the local optima. Roughly speaking, the method starts from an initial solution and jumps from one solution to another one in the solution space but tries to avoid cycling by forbidding or penalizing moves which take the solution, in the next iteration, to solutions previously visited (called *tabu*). To this aim, TS keeps a *tabu list* which is historical in nature and constitutes the tabu search memory. The role of the memory can change as the algorithm proceeds. At initialization the goal is to make a coarse examination of the solution space and further on the search is focused to produce local optima solutions in a process of *intensification* or make a *diversification* in order to explore new regions of the solution space.

Main Entities of Tabu Search. Here we identify the basic entities participating in the tabu search method. In order to abstract these entities and functionalities we did a very careful review of different known *ad hoc* implementations for the method in the literature. As we will see, a right abstraction will be easily translated into a programming language using object oriented and generic programing paradigm.

Recall that we are given an optimization problem consisting of: (a) a set of instances $\mathcal{I}$; (b) to a given instance $x \in \mathcal{I}$, there corresponds a set of feasible solutions $\mathcal{S}(x)$ (hereafter denoted by $\mathcal{S}$); (c) a cost function $f: \mathcal{S} \rightarrow \mathbb{R}$ associating a cost $f(s)$ to solutions $s \in \mathcal{S}$. We want to find an optimum solution $s^* \in \mathcal{S}$ with respect to some criteria (minimization or maximization).

For example, in 0-1 Multidimensional Knapsack we are given a set of n items, each of them is associated a benefit, and there are m linear restrictions on the capacities. The objective is to find a subset of items (to be included to the knapsack) that maximize the whole benefit. In this case, the instance consists of benefits c_j, $j = 1 \ldots n$ of the items, the constraint $m \times n$ matrix and the capacities b_i, $i = 1 \ldots m$. A feasible solution is any subset of items that satisfy the capacity restrictions and its cost is the sum of the benefits of the items. Clearly, the direction is Maximize.

Hence, we have the following entities:

Problem. Represents an instance of the problem to be solved.

Solution. Represents a feasible solution to the problem. Tabu search is an heuristic method so, the acceptability criteria for TS is to find a feasible solution $s' \in \mathcal{S}$ of cost $f(s')$ as close as possible to the optimum cost $f(s^*)$.

Neighborhood. Given a solution s, the set of all possible solutions that are reached from s in a single step (called *move*), is referred to as the neighborhood of s, usually denoted by $\mathcal{N}(s)$. TS moves from s to $s' \in \mathcal{N}(s)$ which is *the best* among all (or part of) possible solutions in $\mathcal{N}(s)$. The choice of the next solution s' is crucial to the whole process. Once a solution is visited it is considered tabu for some time. A solution in the neighborhood that is marked tabu will not be considered, so cycling is avoided by not considering recently visited solutions.

Move. A transition from a feasible solution to another feasible solution is called *move*. Typically in local search algorithms a move performs some local perturbation over the solution it is applied to. A move may be described by one or more attributes, and these attributes can become the foundation for creating an attribute based memory. Considering the number of attributes representing a move we may distinguish single-attribute moves and multi-attribute moves. The reason behind this is that, in general, solutions are extremely impractical to keep track of the exploration process and therefore it is better to describe the exploration process in terms of moves. Moves are given the *tabu status* if they lead to previously visited solutions. However TS establishes an *aspiration criteria* so that tabu moves can be accepted if they satisfy such a criteria.

Tabu list and efficient use of memory. Applying a move to a given solution may result in an improvement or a deterioration of the objective function value. Without additional control, however, such a process can cause a locally optimal solution to be re-visited immediately after moving to a neighbor, or in a future stage of the search process, respectively. To prevent the search from cycling between the same solutions, TS uses a short term memory –the so-called tabu list– to the aim of representing the trajectory of solutions encountered. Based on certain restrictions the tabu list implicitly keeps track of moves by recording their attributes. These attributes will be forbidden from being embodied in moves selected in at least one subsequent iteration because their inclusion might lead back to a previously visited solution. The goal is to permit "good" moves in each iteration without re-visiting solutions already encountered. The key point to the TS procedure lies in the tabu list management.

Intensification. While exploring a region of solution space it seems reasonable to *intensify* the search if we had evidences that such a region may contain good solutions. To this aim, TS incorporates an intensification procedure that consists in rewarding solutions having features in common with the current solution and penalizing solutions that are far from the current solution.

Diversification. As it happens with other local search methods, during the exploration process the method may get stuck in a region where no good solutions are found even a reasonable number of iterations is done. In such a case, the TS method launches the *diversification* procedure to spread out the search in another region. The diversification may be *softly* meaning that a region close to the current one is going to be explored, or *strongly* when a completely different region is considered.

Main procedure. There is no a standard main procedure for TS in the literature. We have checked out several existing TS programs and have observed that the main procedure looks different in different implementations of TS since the authors adapt it to the problem at hand. Since we wanted a skeleton for TS such that any problem could *fit in*, we had to deal with the design process of a component. This component was going to be the principle engine of any program for TS obtained by instantiating the skeleton. In such a design process we had to abstract from a large number of different implementations for TS and we came up with the component, called Solver. For simplicity, we give the diagram of Figure 1 where we can easily observe the main functionalities of such a component.

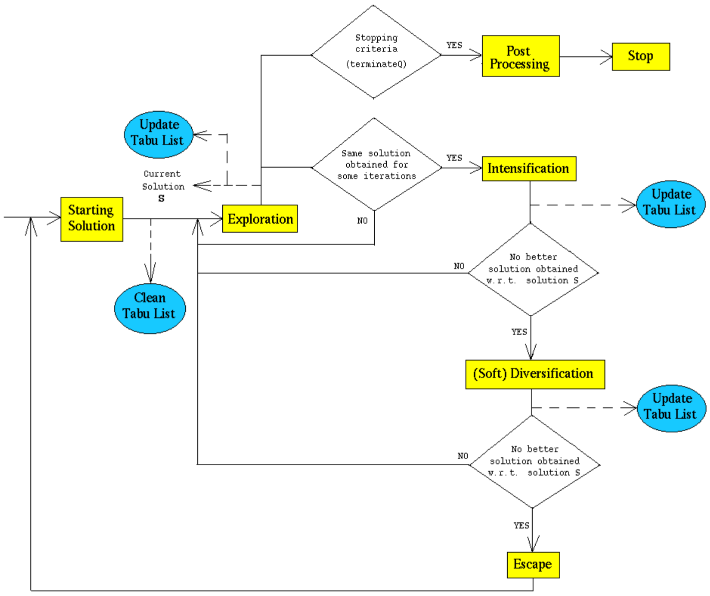

Figure 1: Main procedure of Tabu Search method.

2.2 Overview on the Sequential Skeleton for Tabu Search

The main entities mentioned above are easily introduced into either C++ classes or methods according to a logical definition in the context of the tabu search method. Some of these elements have directly become C++ classes (e.g. problem, solution and move) while others have been introduced into classes as methods (e.g. intensification and diversification).

Note that for the entities defined above, except the main procedure, their concrete representation depends strongly on the problem to be solved. The basic idea behind the sequential skeleton is to allow the user to instantiate any combinatorial optimization problem of interest by only defining the most important problem-dependant features. Elements related to the inner algorithmic functionality of the method are hidden to the user.

C++ Implementation of the Skeleton. The classes forming the skeleton are structured according to their *"availability"*. The classes implementing inner functionalities of the method (e.g. the main procedure) are completely provided by the skeleton, whereas there are some classes whose implementation is required to be particularized by the user. Therefore, the classes forming the skeleton are classified into two groups: *provided* and *required* classes.

Provided Classes. These classes implement the tabu search method itself and the rest of inner functionalities. Actually there are only two *provided classes* in the skeleton: the class Solver and the class Setup. The class Solver implements the main procedure of the TS and maintains the state of the exploration. The class Setup contains the setup parameters needed to run the method (e.g. number of independent runs to perform, number of iterations per independent run, etc.). The user can consult the state of the search and he can also inquire other information related to the exploration process. To this end the skeleton offers a transparent interface, i.e. definition, of the provided part to the user. In a certain sense, the *provided classes* can be seen as a "private" part of the skeleton.

Required Classes. These classes represent the rest of the entities and functionalities involved in the TS method (see Section 2.1) whose representation/implementation depends on the problem being solved. The requirements needed over these entities also depend on the problem. We have been able to abstract the necessities of each entity but the way they are carried out when solving a problem depends strongly on the problem itself. This allowed us to define C++ classes with a fixed interface but no implementation, so the Solver can use them in a "blind and generic way" (i.e. as black boxes) when implementing the TS method but they need a concrete implementation when instantiating a concrete problem.

Instantiation process. The *instantiation* of the sequential skeleton to solve a concrete problem (e.g. 0-1 Multidimensional Knapsack Problem) is the process of completing the requirements of the classes labelled as *required* with the features of the problem at hand. More precisely, to complete a *required class* means: (1) introduce data types for representing the entities and, (2) implement the methods of the class according to the chosen data types.

We have separated the C++ classes of the skeleton in three parts: (1) the definition (interface) of the classes; (2) the implementation of the *provided classes* and, (3) the implementation of the *required classes*. Talking in terms of files, the definition of the classes is located in the file `TabuSearch.hh`, the implementation of the *provided classes* is located in the file `TabuSearch.pro.cc` and finally the implementation of the *required classes* has to be done in the file `TabuSearch.req.cc` (see Figure 2). All the classes, defined and implemented in these files, are comprised under a unique namespace, called *TabuSearch*.

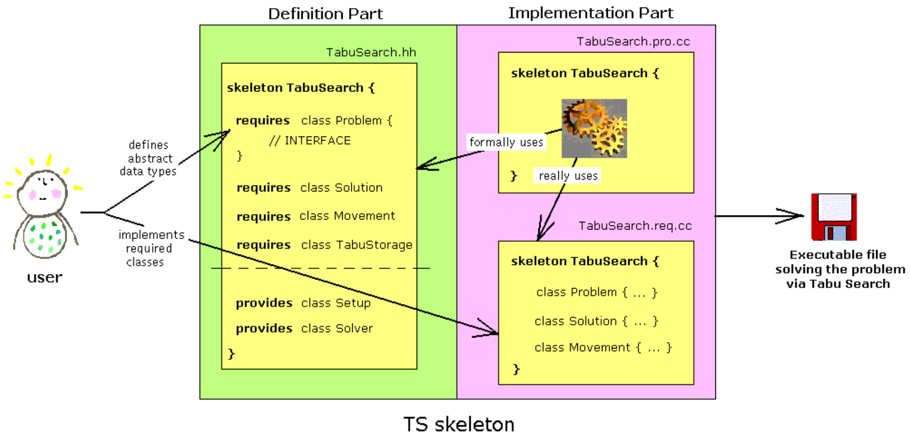

Figure 2: File composition of the TS-skeleton.

Following we show in more detail how the entities and concepts abstracted in Section 2.1 have been translated into classes and methods. The *provided class* `Solver` is reviewed due to its important functionalities, but we will mainly review the interfaces of the *required classes* and some examples of how to instantiate the skeleton since we consider this part of real interest to the user (for a complete description the reader is referred to [2]).

2.2.1 The provided class `Solver`

This class represents the main procedure of TS method and all the internal features related to the search. We have introduced intensification and diversification into the exploration process in order to construct a generic standarized algorithm for the TS method (see Figure 1). The user can decide if he wants to apply them and, if so, which will be their effect over the neighborhood-based search (see for example, the class `Solution` in 2.3).

The class `Solver` also collects information about the state of the search being performed. The state basically consists of information about the best solution found so far and other attributes describing the current point in the search process. The search is divided into three levels, so the state is too: (1) the *global level*, (2) the *independent-run level* and, (3) the *phase level*. The whole search (global level) is composed of some independent runs[2]. In an independent run the Tabu Search algorithm is executed, i.e. the method from an initial solution until a final solution is reached via successive neighborhood explorations (see Figure 1). This is what we call the independent-run level. Initially this should be the lowest search level, but we have introduced a lower one in order to allow a detailed control of the search. We have called this lowest level a *phase*. A phase in the Tabu Search method is considered to be an atomic transition from a current solution $s_i \in \mathcal{S}$ to the next solution $s_{i+1} \in \mathcal{S}$. Note that s_{i+1} can be either the best solution in the neighborhood $\mathcal{N}(s)$, either the solution obtained after instensification or the one obtained after diversification (depending on the conditions that trigger these procedures). In the first case a phase would coincide with a search iteration, but in the other cases it would be composed of some search iterations[3].
The interface for class `Solver` now follows:

```
class Solver {

public:

    Solver (const Problem& pbm, const Setup& setup);
    virtual ~Solver ();
    const Problem& problem () const;
    const Setup&   setup () const;

    // Execution and Hybridization
    virtual void run () =0;
    virtual void perform_one_independent_run () =0;
    virtual void perform_one_phase () =0;
    virtual void set_current_solution (const Solution& sol);
    virtual void set_current_solution (const Solution& sol, const double cost);

    // Global information - state
    virtual int       independent_run () const;
    virtual float     time_spent () const;
    virtual Solution  best_solution () const;
    virtual int       independent_run_best_found () const;
    virtual int       iteration_best_found () const;
    virtual float     time_best_found () const;
    virtual double    best_cost () const;
    virtual double    worst_cost () const;

    // Independent run information - state
    virtual Solution  current_solution () const;
    virtual double    current_cost () const;
    virtual int       current_iteration () const;
    virtual Solution  current_initial_solution () const;
    virtual double    current_initial_cost () const;
    virtual float     current_time_spent () const;
    virtual Solution  current_best_solution () const;
    virtual int       current_best_solution_iteration () const;
    virtual float     current_best_solution_time () const;
    virtual double    current_best_cost () const;
    virtual double    current_worst_cost () const;

    // Phase information
    virtual int  nb_iterations_performed_in_phase () const;
    virtual bool intensification_in_phase () const;
    virtual bool soft_diversification_in_phase () const;
    virtual bool hard_diversification_in_phase () const;

protected:
```

[2]The user configures how many independent runs compose the global search and how many iterations conform an independent run by menas of a setup parameter.

[3]More precisely, it would consist of $1 + ((nb_intensifications) + nb_diversifications)$, where `nb_intensifications` and `nb_diversification` are setup parameters provided by the user to configure the duration of the intensification and diversification procedures.

```
    const Problem& _problem;
    const Setup&   _setup;
    const bool     _minimizing;

    int      _independent_run;
    float    _time_spent;
    Solution _best_solution;
    int      _independent_run_best_found;
    int      _iteration_best_found;
    float    _time_best_found;
    double   _best_cost;
    double   _worst_cost;

    Solution _current_solution;
    double   _current_cost;
    int      _current_iteration;
    Solution _current_initial_solution;
    double   _current_initial_cost;
    float    _current_time_spent;
    Solution _current_best_solution;
    int      _current_best_solution_iteration;
    float    _current_best_solution_time;
    double   _current_best_cost;
    double   _current_worst_cost;
    int      _nb_iterations_performed_in_phase;
    bool     _intensification_in_phase;
    bool     _soft_diversification_in_phase;
    bool     _hard_diversification_in_phase;

    virtual void  update_state ();
};
```

Note that almost all the methods are virtual. This is due to the hierarchy defined over the class `Solver` in order to diferentiate between the sequential solver and the parallel solvers. The superclass `Solver` defines the requirements of a generic solver and the subclasses detail the different implementations.

2.2.2 The required class `Problem`

This class represents an instance of the problem to be solved. The internal implementation of the class only needs to create, serialize and get the direction (i.e. minimization or maximization), so the requirements over this class are quite few and its interface is quite simple. Note that serialization is important when an object of this class has to be sent to other processes in a parallel execution. This is exactly the standard interface supplied by the skeleton (note also that the *required classes* are labelled with the keyword `requires` before the class definition itself):

```
requires class Problem {
public:
       Problem ();
       ~Problem();
       friend ostream& operator<< (ostream& os, const Problem& pbm);
       friend istream& operator>> (istream& is, Problem& pbm);
       friend opacket& operator<< (opacket& op, const Problem& pbm);
       friend ipacket& operator>> (ipacket& ip, Problem& pbm);
       Direction direction () const;
};
```

When instantiating the skeleton for a concrete problem the user has to complete this interface (in the `.hh` file) by defining the data types to represent the instance problem. Then, the methods of the class have to be implemented (in the `.req.cc` file) according to the chosen representation. For example, to instantiate the 0-1 Multidimensional Knapsack Problem (0-1 MKNP) it is needed a representation for the instance, i.e. a representation for the benefits, the capacities and the constraint matrix. This representation may be added to the definition of the generic class (the one above) as private attributes:

```
private:
       int nb_items, nb_constraints;

       array2<double> _constraints;
       array<double>  _benefits;
       array<double>  _capacities;
```

The direction in this case is `Maximize`, so the implementation of the method `direction()` in the `.req.cc` file should be as follows:

```
Direction Problem::direction () const
{
        return Maximize;
}
```

2.3 The required class `Solution`

This class represents a feasible solution to the problem. The implementation of the TS method works with generic solutions by simply fixing the interface of the class. Again, in a concrete instantiation, a representation using data types is needed.

The constructor, destructor and serialization methods and operators are required. In this case it is needed to specify how to construct a solution from a problem and from an existing solution (copy constructor), and also define some useful operators such as `==`, `!=`.

```
Solution (const Problem&  pbm);
Solution (const Solution& sol);
~Solution();

Solution&   operator=  (const Solution& sol);
friend bool operator== (const Solution& sol1, const Solution& sol2);
friend bool operator!= (const Solution& sol1, const Solution& sol2);

friend ostream& operator<< (ostream& os, const Solution& sol);
friend istream& operator>> (istream& is, Solution& sol);
friend opacket& operator<< (opacket& op, const Solution& sol);
friend ipacket& operator>> (ipacket& ip, Solution& sol);
```

To any solution there is associated a cost so the class `Solution` includes a method for evaluating the cost of an object of the class. This method in charge of calculating the cost of the solution is named `fitness()`. While performing the exploration, the solutions in the neighborhood of the current solution are evaluated in order to choose the best one as next solution. Depending on the problem, the neighborhood of a solution can be really large and therefore evaluating the cost of solutions may result in time consuming. That is why in some cases it is easier to calculate the differential cost between the next solution and the current solution without applying the move that leads from one solution to the other. When it is possible, this differential cost can be implemented in the method `delta()` but it is not strictly necessary, it is only for the sake of efficiency. Note that, to evaluate the cost of a solution in the neighborhood, the method `fitness` needs the movement to be applied first to the current solution but the method `delta` does not.

```
double fitness () const;
double delta (const Movement& move) const;
```

As in other local search methods, Tabu Search starts exploration from an initial solution generated by some other procedure. Typically the initial solution is constructed randomly or by a greedy algorithm. The initial solution in the TS skeleton must be offered by the method `set_initial()`. Thinking on the parallel implementation, if a deterministic (greedy) procedure is used, all the different searches will start with the same initial solution (i.e. at the same point of the solution space). In some cases we may want to perturbate the solution in order to introduce nondeterminism in consecutive or parallel searches. This will guarantee that different searches over the solution space will start from different points. The method `perturbate_randomly()` changes locally and randomly a given solution.

```
void set_initial ();
void perturbate_randomly ();
```

In local search methods there exists the concept of neighborhood. Given a solution s, its neighborhood $\mathcal{N}(s)$ is the set of all possible solutions that are reached from s by applying a single move. A reachable solution s' in the neighborhood can be described in terms of the source solution s and the move m that leads to that neighbor solution after being applied to the source ($s \xrightarrow{m} s'$). The method `apply()` transforms a solution into a neighbor solution by applying a movement, so the neighborhood will be explored by generating the movements that lead to solutions in it and apply them to the current solution. Because the target solution is changed after applying a movement a method `unapply()` is needed to explore the rest of the neighborhood. Again, note that it is not needed to apply the movement when exploring the neighborhood if `delta` function is used.

```
void apply ( const Movement& move );
void unapply ( const Movement& move );
```

Tabu Search will choose the best solution in the neighborhood (actually the best move that leads to it) to be the next solution and continue the exploration. Moves in TS are given the *tabu status* if they lead to previously visited solutons. This tends to avoid cycling. Tabu Search also establishes an *aspiration criteria* so that tabu moves can be accepted if they satisfy such a criteria. The method `aspiration()` checks this criteria over the current solution in relation to a given movement.

```
bool aspiration (const Movement& move, const TabuStorage& tstore, const Solver& solver) const;
```

Tabu Search incorporates an intensification procedure to intensify the search if it had evidences that the region being explored may contain good solutions. The intensification is done by rewarding solutions having features in common with the current solution, and then solutions that are far from the current solution are penalized indirectly. The method `reward()` has to describe how the current solution (actually the items forming the solution) are rewarded. This method will be invoked just before the intensification starts. After intensifying the search, the solutions has to loose the reward, so method `unreward()` has to describe how to change back the solution to its original weight. Similarly, the methods `penalize()` and `unpenalize()` allow a soft diversification of the search, and the method `escape()` performs a hard diversification (see Section 2.1 and Figure 1).

```
void reward ();
void unreward ();
void penalize ();
void unpenalize ();
void escape ();
```

These are all the methods of the interface of the class `Solution`. Let us see how to complete this class for the 0-1 Multidimensional Knapsack Problem. First, a user needs to represent how is a solution to the problem. Given the information about the benefits, the capacities and the contraints related to the knapsack problem, a solution is a choice for each item deciding if it should be accepted in the knapsack or not. This can be easily represented by an array of binary values, where the i-th position indicates if the i-th item is included in the knapsack or not. Then the user would complete the definition of the class in the `.hh` file adding the private representation:

```
requires class Solution {
public:
        ...
private:
        array<int> _inclusions;4
}
```

Next the user has to implement the methods of the class according to their functionality. This is done in the `.req.cc` file. For example, the method `fitness()` has to calculate the cost associated to the solution, i.e. the sum of the benefits of the items included in the knapsack.

[4] Note that the user may need more auxiliar data types depending on the complexity of the problem to be solved and the efficiency of his implementation.

```
double Solution::fitness () const
{
    double cost = 0;
    for(int item=0; item<_problem.nb_items; item++)
        cost += _problem._benefits[item] * _inclusions[item];5

    return cost;
}
```

2.3.1 The required class `Movement`

This class represents the entity *movement*. A movement m is applied on a soluton s in order to reach a neighbor solution $s' \in \mathcal{N}(s)$. As always, constructor, destructor and serialization methods are needed in this class. Comparison operators are also needed in tabu search because of the management of the *tabu list*. This management requires also the movement to have a "tabu life", i.e. the time since a movement have been considered tabu.

Given a movement m, a solution s and the solution s' reached after applying m to s ($s \xrightarrow{m} s'$), the inverse movement m' is a movement such that it allows to recover the source solution ($s' \xrightarrow{m'} s$). In the Tabu Search method it is needed to invert a given movement because after applying a movement to the current solution, its inverse form is made tabu in order to avoid falling into an already visited solution. The interface of the class `Movement` follows:

```
requires class Movement {
public:
        Movement (const Problem& pbm, const Solution& sol);
        Movement (const Movement& move);
        Movement ();
        ~Movement ();

        Movement& operator= (const Movement& move);
        friend bool operator== (const Movement& move1, const Movement& move2);
        friend bool operator!= (const Movement& move1, const Movement& move2);

        friend ostream& operator<< (ostream& os, const Movement& move);
        friend istream& operator>> (istream& is, Movement& move);
        friend opacket& operator<< (opacket& op, const Movement& move);
        friend ipacket& operator>> (ipacket& ip, Movement& move);

        void generate ();
        void invert ();
        int  tabulife () const;
        void set_tabulife (int i);

private:
        int _tabulife;
};
```

For the 0-1 Multidimensional Knapsack Problem, a movement is a set of items that are dropped from the knapsack and a set of items to be added. The items to be dropped are selected according to the saturation of the different dimensions of the knapsack. The items with the lowest benefit of the most saturated dimensions are candidates to be dropped. A candidate should not be tabu, because intuitively that would mean that they were inserted recently. The items to be added are selected according to the benefit they contribute. Those with better benefit and not belonging to the current knapsack configuration are candidates to be added. The capacities of the knapsack must also be controled and updated when selecting the items to be added. The user may represent a movement like this with an array of integers (each integer is the index of an item in the array representing the knapsack) and two integer values indicating the number of items to be dropped and the number of items to be added. So the standard generic interface for the class Movement provided by the skeleton may be filled in as follows:

```
private:
        array<int> _items;
        int        _nb_drop, _nb_add;
```

[5] The attribute `_problem` is a reference to the representation of the instance, and the attribute `_benefits` of the problem is the representation of the benefits of each item.

2.3.2 The required class `TabuStorage`

Tabu Search maintains information related to the exploration process. Such information will be used to guide the search from a solution s to next one s' in the neighborhood by forbidding moves to some neighbor solutions. More precisely, the real aim of this "history" is to avoid reaching already visited solutions, at least for "some time". The simpler way to maintain this information is using a list T of the last $|T|$ solutions visited in order to prevent cycles of length at most $|T|$. Then from a current solution s only those solutions in $\mathcal{N}(s) - T$ have to be considered as candidates for the next solution. However, this list may be extremely impractical to use. As comented previously, it is better to describe the exploration process in terms of moves. So instead of keeping a list T of te last $|T|$ solutions visited, we may simply keep track of the last $|T|$ moves or of the last $|T|$ reverse moves associated with the moves actually performed. It is clear that this restriction is a loss of information and that it does not guarantee that no cycle of length at most $|T|$ will occur. Another drawback of the simplification of the tabu list, due to replacement of solutions by moves, is the fact that we may forbid visiting solutions that may be unvisited so far. We are then compelled to some relaxation of the tabu status; the tabu status will be overruled when some tabu conditions look attractive. This will be performed by *aspiration criteria*.

The class `TabuStorage` represents the structure to manage tabu movements. The skeleton provides a list as a default implementation but the user can implement a more efficient or just appropriate structure depending on the problem.

```
requires class TabuStorage {
public:
        TabuStorage (const Solver&);
        ~TabuStorage ();
        bool is_in_tabu_storage (const Movement& move, const Solution& sol) const;
        bool is_tabu (const Movement& move, const Solution& sol) const;
        void make_tabu (Movement& move, const Solution& sol);
        void make_tabu_inv (Movement& move, const Solution& sol);
        void update ();
        int  size () const;

private:
        const Solver&  _solver;
        list<Movement> _tlist;          // Default standard implementation
};
```

The structure needs a `Solver` to be created because there is information about setup parameters (particularly three of them `tabu_size`, `max_tabu_status` and `min_tabu_status`). They influence the definition and the management of the tabu storage structure. The `Solver` also keeps information about the state of the exploration that can be useful to decide if a movement must be considered tabu or not.

Basically, there are a couple of consultant methods: one method to consult if a movement is physically in the tabu storage (`is_in_tabu_storage`), and another one to consult if a movement is considered tabu or not (`is_tabu`). Note that being tabu may be different than being in the tabu storage. There are also a couple of methods to make movements tabu (one of them make tabu the inverse move). From time to time the structure may need to be updated, typically movements considered tabu for much time (usually more than `max_tabu_status` iterations) are eliminated from the structure. This purge must be implemented in the method `update()`. Finally, the method `size()` returns the size of the structure.

For the 0-1 Multidimensional Knapsack Problem, the tabu storage is more complicated than a simple list. Neither the standard implementation as a list of `Movement` is useful. Three structures are needed to perform an efficient management of the tabu movements when instantiating this problem:

- A tabu list implemented as an array of dimension n (the number of variables participating in the knapsack). The jth component stores the number of iteration where the jth item is made tabu. Experiments has shown that for the case of the 0-1 MKNP, the size of the tabu list should be randomly and uniformly chosen in the interval [3..15].

- An array of dimension n such that the jth component stores the number of iterations that the jth item has been accepted.
- An array maintaining a small but fixed number of solutions. Those solutions are the ones with best benefit found so far.

Again, these structures should be added to the definition of the standar generic class (`.hh` file) and their management via the methods described above should be implemented (`.req.cc` file). This can be done as follows:

```
private:
        array <int>      _TL;
        array <int>      _history;
        array <Solution> _best_solutions;
```

2.3.3 Others Methods

After the organization in classes of the main entities and functionalities participating in the Tabu Search method, only two other procedures are needed: one describing how to explore the neighborhood, and one describing how to terminate and independent run when executing the TS method. None of them are specially involved in any concrete class, so we have decided to introduce them in the TS-skeleton as independent required methods.

Exploring the neighborhood. As we already explained, the neighborhood of the current solution is explored by applying correct movements to it. The next solution of the exploration process will be the one obtained by applying the "best" move to the current solution. The user decides what does "best" mean according to his criterias. Usually these decision is made according to cost values and aspiration criterias of the solutions in the neighborhood. The user also decides whether the hole neighborhood or only a part of it needs to be explored. This decision can be very influential in the final efficiency of the instantiation. The method in the skeleton to specify how the neighborhood is explored is named `choose_best_move`, and its signature follows:

```
requires bool choose_best_move (const Problem& pbm, const Solution& sol,
                 const TabuStorage& tstore, const Solver& solver, Movement& move);
```

The parameters of this method are generic-enough to allow the exploration of the neighborhood independently of the problem to be instantiated. That is why objects from class `Problem`, `Solver` and `TabuStorage` are needed as input parameters. The method should return a boolean indicating if a move could be chosen[6] and, if so, that movement is returned as an output parameter.

Terminating the algorithm. We consider as *independent run* a complete Tabu Search procedure execution, i.e. the procedure from a starting solution up to the final one. This includes some "atomic explorations", i.e. transitions from s_i to s_{i+1} in the search process. Note that s_{i+1} can be a solution in $\mathcal{N}(s_i)$ reached after applying the best move, or it can be the solution obtained after intensifying or diversifying the search[7] around s_i.

An independent run may finish when some criteria has been reached, usually after performing some iterations but this is not the only possible condition. In order to allow the user to specify any termination criteria a problem-independant required method (named `terminateQ`) has been introduced in the TS-skeleton. Its signature is as follows:

```
requires bool terminateQ (const Solver& solver);
```

Again the parameters of this method should be useful enough to allow whichever implementation the user wants to do. The termination condition can depend on two main entities: the setup parameters and the current state of the search. Both of them are included in the class

[6] A solution is considered to have an empty neighborhood if there is no movement to be applied to it or if all the movement that can be applied to it are tabu and no-aspiration ones.

[7] Remember that intensification and diversification are clever variations of the search process that allow a deep/wide exploration of the search space near the current solution (see Section 2.1 and Figure 1).

Solver, so only an object of this class is needed as input parameter. Typically an independent run terminates when an specified number of iterations have been performed, but the user can implement it according to a maximum execution time etc.

2.3.4 Running the sequential skeleton

The skeleton is run through the class Solver and the final state of the search is also provided by this class. As seen above, to instantiate a problem using the TS-skeleton the user only has to give a concrete definition for the required classes (via abstract data types) and implement the methods of these classes according to the chosen definition and the functionalities of these methods. Once the instantiation is completed, a user may run it with a program like this:

```
#include "TabuSearch.hh"

int Main (int argc, char** argv)
{
    using skeleton TabuSearch;

    Problem problem;                          // Read the problem instance.
    cin >> problem;
    Setup setup;                              // Read the setup parameters.
    cin >> setup;

    Solver_Seq solver(problem,setup);         // Run the sequential TS method.
    solver.run();

    cout << solver.best_solution() << endl;   // Report best solution found.
    cout << solver.best_cost()     << endl;

    return 0;
}
```

3 Parallel Skeletons for Tabu Search

The sequential skeleton has a required part whose requirements are specified by the skeleton through classes with fixed interfaces. The user instantiates them by providing a concrete representation for the classes (via abstract data types) and by implementing the methods therein. The skeleton also has a completely provided part that implements the TS method itself and maintains information on the exploration process. Due to this separation, a generic parallel implementation of the TS method can be obtained by simply parallelizing the provided part. So, the user's instantiation can also run in parallel effortlessly.

Tabu Search can be parallelized in multiple ways. By now, we have implemented two different parallel versions of the skeleton. In Subsection 3.1 the direct parallelization by independent runs is explained. The Master-Slave parallel model applied to TS is presented in Subsection 3.2. The essentials of the two implementations consist in: (a) the main parallelized part is the method itself, i.e. provided part. So the parallelism remains hidden to the user and his "sequential-thought" instantiations must run in parallel without any effort for him; (b) only by defining how the participating classes are represented and by implementing the methods in them, a user can obtain either a sequential implementation or a parallel one to solve his problem via Tabu Search. So the user does not need to "think in parallel".

3.1 The Independent Runs Model

This model consists of simultaneous and independent executions of the same program (i.e. the tabu search method) in order to lately select the best solution among all the solutions obtained. With a parallel model like this, many different search paths can be explored simultaneously using the same time that a sequential model would use to explore only a unique path. In particular, we have implemented a slight variation of this model in which, the coordinator processor generates *strategies* (as many as there are processors) sends them to the processors and then each processor executes the TS program according to its strategy. A strategy consists mainly of an initial solution and other parameters related to tabu search such as tabu list size, number of instensifications etc. Clearly, in this model the main gain is in computation time. There is no relation between the number of indepedent runs and the quality of the solution obtained by each of them but usually a better global solution is found with this parallel model compared to the sequential one.

3.1.1 Implementation Issues

We have implemented and tested the Independent Runs model over an homogeneous PC cluster using Linux as Operating System and MPI as communication library. Concretely, we have used the MPICH implementation. The main drawback of this implementation (due to MPI) is that it spends much time initializing the parallel processes. Each independent run takes place in a different processor. Full control of the search is delegated to each of them, without any distinguished processor. But in order to start the whole process and gather the solutions found by different processors a coordinator processor is mantained. The coordinator processor distributes the input to all the processors, it waits until they finish the search and finally it collects the solutions in order to select the best one. Note that the coordinator processor is not a master processor (in the sense of the Master-Slave model). This model has a coarse grain parallelism.

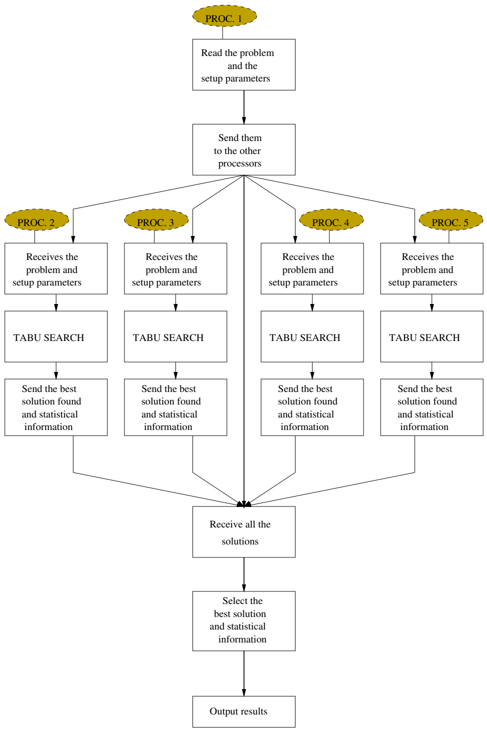

Figure 3: Work of the processors in the Independent Runs parallelization.

3.2 The Master-Slave Model

In the Master-Slave model (M-S) there are two distinguished kind of processors: a *master* processor and identical processors called *slaves*. The main work (the control) is performed by the master and the slaves are subordinated to the master in some point of the master's work.

The master processor spawns slaves processors[8], initialize them, assigns subtasks and collects their results. Then it computes a final result from the results obtained by the slaves and incorporates this final result (in some sense) to its work. So the master controls the whole process

[8] Ideally the slave processes are created dynamically by the master whenever they are needed and destroyed after but, depending on the library of communications used, this dynamic management is allowed or not. If it is not allowed, then the slave processes must be created at the beginning of the parallel program and later reused.

and uses slaves to its own benefit. On the other hand, the slaves are only expected to provide a service to the master. So the Master-Slave model is a one-to-many utilization relation.

The M-S model has some clear advantages: (a) flexibility and scalability, e.g. the slaves processors can be implemented in many different ways and they can be easily added; (b) separation of concerns, i.e. master does coordination and the main work and slaves do specific subtasks; (c) efficiency of the parallelism itself. We can also consider that, in some cases, the M-S model can have some disadvantages, e.g. machine dependency and no-feasibility[9].

3.2.1 Implementation Issues

We have implemented and tested the Master-Slave model over an homogeneous PC cluster using Linux as Operating System and MPI as communication library. As said before, the implementation of MPI used does not allow dynamic process creation, so the processors to be used are determined at the beggining of the parallel execution and the master is not allowed to create and destroy dinamically the slaves (the slaves are initialized and remain slept while they are not working). The master and each of the slaves processes run in a different processor.

The master processor runs the Tabu Search method and uses slaves to choose the "best" move that leads to a solution (the "best" one) in the neighborhood of the current one. So the master has the complete control of the search and the slaves are only used in order to perform in parallel the exploration of the neighborhood.

3.3 Instantiation Process

The instantiation process is the process of adapting the pattern provided by the TS-skeleton to solve a concrete problem. This adaptation consist of two main points: (a) the identification and representation of the entities participating in the method and, (b) the implementation of the functionalitites that these entities are expected to have according to the representation chosen. As we reviewed in Section 2.1, the entities and functionalities participating in the method have been translated into classes and methods in a generic enough form.

The representation of the entities correspond to the identification and representation of the data types implementing the required classes of the skeleton. These data types have to be included in the definition of the classes (in the `.hh` file). The implementation of the functionalitites is done through the implementation of the methods in the required classes. Both the concrete representation of the classes and the implementation of the methods are done "thinking in sequential". The parallelization remains completely hiden to the user and does not influence his instantiation. This is possible because the parallelized part is the method itself, so the same "sequential" instantiation will run in parallel without any more effort to the user, so the user has to instatiate the sequential skeleton and the same instantiation serves for both sequential an parallel execution.

3.4 Running the parallel skeletons

In Subsection 2.3.4 we explained how the sequential implementation of the TS-skeleton can be run after being correctly instantiated. There we saw that only an object of the sequential implementation of class `Solver` needs to be declared and, after this, the method `run` is called.

The same strategy is used to run the parallel implementations. The user only needs to declare an object of the class `Solver` implementing TS method via IR or M-S model and to call the `run` method. So the process is identical to the sequential instantiation and execution only changing the kind (sublclass) of `Solver`. So, once the required classes have been completed with the concrete features of the problem to some (and this is done "thinking in sequential") the Independent Runs or the M-S parallellization of the TS method applied to the instantiated problem is obtained without any more effort.

Let us show it by example. The parallel execution using the Independent Runs model of an instantiated TS-skeleton may be like this:

[9] The definition of subtasks may not always be possible (ir depends on the problem), but in most case it is.

```
#include "TabuSearch.hh"

int Main (int argc, char** argv)
{
    using skeleton TabuSearch;

    Problem problem;                          // Read the problem instance
    cin >> problem;
    Setup setup;                              // Read the setup parameters
    cin >> setup;

    Solver_IR solver(problem,setup);          // Run the parallel Independent-Runs model
    solver.run();

    cout << solver.best_solution() << endl;   // Report best solution found
    cout << solver.best_cost()     << endl;

    return 0;
}
```

3.5 Experimental Results for 0-1 Multidimensional Knapsack Problem

We have instantiated the parallel skeletons for 0-1 Multidimensional MultiKnapsack Problem. For the Independent Runs model, the obtained exeperimental results are given in Table 1 (more results are given in Table 3 and Table 4 in Appendix A). These results are obtained for six instances from the OR-Library [1][10] under the independent runs model. The execution is done with different maximum execution times and different number of processors. In this table, instance referes to the instance name, n and m are respectively the number of variables (items) and the number of restrictions; best known cost column gives the best known cost in the literature up to now [3] for the instance obtained by other specific implementations and best cost found indicates the best cost found by our parallel implementation, p indicates the number of processors (this table corresponds to the execution with $p = 4$ processors.)

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	total time (s)
Instances executed with $max_time = 600$s								
OR5x100-00	100	5	24381	24318	24205.4	0.0025	526785.0	610.1
OR5x100-29	100	5	59965	59620	59458.0	0.0058	729686.0	610.3
OR10x100-00	100	10	23064	22644	22414.4	0.0182	343089.6	611.2
OR10x100-29	100	10	60633	60629	60603.6	0.0001	453284.6	611.6
OR30x100-00	100	30	21946	21651	21595.2	0.0134	129800.4	614.0
OR30x100-29	100	30	60603	60503	60420.4	0.0017	171800.2	613.1
Instances executed with $max_time = 900$s								
OR5x250-00	250	5	59312	56980	56290.4	0.0393	359956.0	912.0
OR5x250-29	250	5	154662	153684	153483.8	0.0063	449184.0	911.1
OR10x250-00	250	10	59187	56727	56416.4	0.0416	224706.6	915.6
OR10x250-29	250	10	149704	148280	148083.2	0.0095	286595.6	914.7
OR30x250-00	250	30	56693	54792	54715.2	0.0335	70720.6	921.0
OR30x250-29	250	30	149572	148621	148427.8	0.0064	100949.4	919.6
Instances executed with $max_time = 1200$s								
OR5x500-00	500	5	120130	114741	114187.4	0.0449	242854.0	1212.6
OR5x500-29	500	5	299904	297383	297111.4	0.0084	298442.6	1212.9
OR10x500-00	500	10	117726	112199	111742.6	0.0470	144402.8	1221.1
OR10x500-29	500	10	307014	304354	303823.8	0.0087	185546.6	1217.5
OR30x500-00	500	30	115868	111648	111118.6	0.0364	42970.4	1228.8
OR30x500-29	500	30	300460	298753	298612.0	0.0057	61519.8	1229.3

Table 1: Results obtained for big instances of the 0-1 MKNP with Independent-Runs Strategy for $p = 4$.

We have also experimented for the same problem and the same instances under the master-slave model. The experimental results for $p = 4$ processors are given in Table 2 (more results are given in Table 5 and Table 6 in Appendix B) .

[10] http://mscmga.ms.ic.ac.uk/info.html

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	total time (s)
Instances executed with $max_time = 600$s								
OR5x100-00	100	5	24381	24071	23869.6	0.0127	3884.4	609.9
OR5x100-29	100	5	59965	59596	59448.0	0.0062	3927.2	610.0
OR10x100-00	100	10	23064	22357	22241.8	0.0307	2107.8	611.6
OR10x100-29	100	10	60633	60538	60451.8	0.0016	2095.6	610.8
OR30x100-00	100	30	21946	21378	21295.6	0.0259	491.6	625.6
OR30x100-29	100	30	60603	60288	60206.6	0.0052	810.6	614.1
Instances executed with $max_time = 900$s								
OR5x250-00	250	5	59312	57134	56615.2	0.0367	4757.0	911.6
OR5x250-29	250	5	154662	153713	153550.8	0.0061	4730.0	911.9
OR10x250-00	250	10	59187	56734	56276.8	0.0414	1373.4	914.2
OR10x250-29	250	10	149704	148148	148066.4	0.0104	1296.0	914.3
OR30x250-00	250	30	56693	54561	54455.8	0.0376	511.0	925.9
OR30x250-29	250	30	149572	148489	148351.8	0.0072	512.4	920.0
Instances executed with $max_time = 1200$s								
OR5x500-00	500	5	120130	115134	113655.2	0.0416	3139.2	1212.4
OR5x500-29	500	5	299904	297259	296578.6	0.0088	3128.0	1213.8
OR10x500-00	500	10	117726	111998	111387.2	0.0487	418.8	1226.8
OR10x500-29	500	10	307014	303647	303448.4	0.0110	459.0	1228.2
OR30x500-00	500	30	115868	110925	110536.0	0.0427	140.0	1242.1
OR30x500-29	500	30	300460	298448	298159.6	0.0067	147.2	1246.5

Table 2: Results obtained for big instances of the 0-1 MKNP with the Master-Slave Strategy for $p = 4$.

4 Conclusions and Future Work

In this paper we have presented two parallel skeletons for tabu search method designed and implemented using the generic parallel programming paradigm. The first skeleton is based on independent runs while the second in the master-slave model. In order to achieve these implementations we have first designed and implemented a sequential skeleton for the TS method that is used later as basis for the two parallel skeletons. Both skeletons represent several advantages to the user, namely, (a) the user can obtain parallel implementations of TS method for concrete combinatorial optimization problems from existing sequential implementations; (b) there is no need for the user to know neither parallel programming nor communication libraries; (c) the parallel implementation for a concrete problem is obtained automatically from a sequential implementation of the problem. The skeletons, however, require from the user a sequential instantiation of TS method for the problem at hand. We have instantiated the two skeletons for the 0-1 Multidimensional Knapsack problem and have observed that the parallel skeletons are easy to use, permit considerable time savings, and other properties such as robustenss, genericity due to the generic programming and object oriented programming. Importantly, in spite of being generic, there is no loose neither in efficiency nor in the quality of solutions, as it can be seen from our experimental results. We are actually testing our skeletons on other combinatorial optimization problems, concretely for Quadratic Assignement Problem and k-Cardinality Tree Problem.

We plan to implement other variants of parallel skeltons for tabu search that explore other degrees of parallelism based such as Master-Slave with strategies and Master-Slave with neighborhood partition, again from the generic parallel programming paradigm.

As for the actual versions, we plan to simplify even more the execution process of the parallel programs, in a sense, hide even more the parallelism. We think that the parallelism should be really automatic and tranparent, without needing any interaction from the user to the point that by only checking the underlying system where the execution will take place, the skeleton will be expected to configure and run a parallel execution automatically. The configuration of the parallel execution should be accurate-enough in order to take the maximum profit from the underlying system according to the number of available processors, their load and so on.

References

[1] J.E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail, *J. of Op. Res. Soc.* 41:11,1069–1072, 1990.

[2] M.J. Blesa and F. Xhafa. A C++ Implementation of a Skeleton for Tabu Search Method. Tech. Rep. LSI-00-47-R, Dept. de LSI, UPC, 2000.

[3] P.C. Chu and J.E. Beasley. A Genetic Algorithm for the Multidimensional Knapsack Problem, 1997. *Working Paper*

[4] T.G. Crainic and M. Toulouse. Parallel Metaheuristics. Tech. Rep., Dépt. des Sci. Admin. (Université du Québec a Montréal), 1997.

[5] T.G. Crainic, M. Toulouse, and M. Gendreau. Towards a Taxonomy of Parallel Tabu Search Heuristics. Tech. Rep., Dépt. d'Inf. et de Recherche Opérat. Université de Montréal, 1995.

[6] M. Dell'Amico and M. Trubian. Applying Tabu Search to the Job-Shop Scheduling Problem. *Ann. of Op. Res.*, 41:231–252, 1986.

[7] C.N. Fiechter. A Parallel Tabu Search Algorithm for Large Travelling Salesman Problem. *Disc. App. Math.*, 51:243–267, 1994.

[8] C. Fleurent and J. Ferland. Genetic Hybrids for the Quadratic Assignment Problem. In P. Pardalos and H. Wolkowicz, eds., *Quadratic Assignment and Related Problems.* Amer. Math. Soc., 1994.

[9] R.L. Francis and J.A. White. *Facility Layout and Location.* Prentice-Hall, 1974.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[11] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[12] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Op. Res.*, 5:533–549, 1986.

[13] M. Sato, J. Gerlach. Generic Programming for Parallel Mesh Problems. In Marydell Tholburn et al., eds., *3rd Int. Symp. on Comp. in Obj. Orient. Parallel Environments, ISCOPE'99*, vol. 1732 of *Lecture Notes in Comp. Sci.*, pages 108–119. Springer Verlag, 1999.

[14] W. Gropp and E. Lusk and N. Doss and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Par. Comp.*, 22:6, 789–828, 1996.

[15] K. Jörnsten and A. Løkketangen. Tabu Search for Weighted k-Cardinality Trees. *Asia-Pacific J. of Op. Res.*, 14(2):9–26, 1997.

[16] E. Kaltofen and A. Diaz. Generic Programming in the POOMA Framework. In *Generic Programming Followup to a Dagstuhl Seminar*, 1998.

[17] J. Krarup and P.M. Pruzan. Computer-aided Layout Design. *Math. Prog. Study*, 9:75–94, 1978.

[18] M. Laguna, J.W. Barnes, and F. Glover. Tabu Search Methodology for a Single Machine Scheduling problem. *J. of Int. Manufacturing*, 2:63–74, 1991.

[19] A. Lumsdaine and J.G. Siek. Generic Programming for High Performance Numerical Linear Algebra. In *Int. Workshop on Modern Software Tools for Scientific Computing SciTools'98, Electronic Proceedings*, 1998.

[20] T.K. Ralphs M. Eso, L. Ladanyi and L.E. Trotter Jr. Fully Parallel Generic Branch-and-Cut Framework. In *Proc. of the 8th SIAM Conf. on Parallel Proc. for Scientific Comp.*, 1997.

[21] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms, and Martial Arts: Towards Memetic Algorithms. Tech. Rep. No. 790, California Inst. of Tech., 1989.

[22] P. Moscato. Memetic Algorithms: A Short Introduction. In M. Dorigo et al. eds, *New Ideas in Opt.*. McGraw-Hill, 1999.

[23] C.S. Porto and C. Ribeiro. Parallel Tabu Search message-passing Synchronous Strategies for Task Scheduling Under Procedence Constraints. *J. of Heuristics*, 1(2):207–223, 1996.

[24] S. Porto and C. Ribeiro. A Tabu Search Approach to Task Scheduling on Heterogeneous Processor under Precedence Constraints. *Int. J. of High-Speed Comp.*, 7, 1995.

[25] J. Skorin-Kapov. Tabu Search Applied to the Quadratic Assignment Problem. *ORSA J. on Comp.*, 2(1):33–45, 1990.

[26] L. Steinberg. The Backboard Wiring Problem : A Placement Algorithm. *SIAM Review*, 3:37–50, 1961.

[27] N. Sundaresan. Generic Parallel Programming in C++. Colloquia 1997, University of California, Davis, 1997.

[28] E. Taillard. Robust Tabu Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[29] E. Taillard. Parallel Iterative Search Methods for Vehicle Routing Problem. *Networks*, 23:661–673, 1993.

[30] Leslie Trotter. Generic Parallel Implementation for Integer Programming. DONET Spring School on Computational Combinatorial Optimization, Dagstuhl, 2000.

[31] M. Widmer. The Job-shop Scheduling with Tooling Constraints: A Tabu Search Approach. *J. Op. Res.*, 42:75–82, 1991.

A Experimental Results from Independent Runs Parallelization of the TS method.

Table 3 and Table 4 present further experimental results obtained from the execution of the independent runs parallel implementation for 0-1 Multidimensional Knapsack. The notation is the same as explained previously in Section 3.5

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	total time (s)
Instances executed with $max_time = 600$s								
OR5x100-00	100	5	24381	24083	23981.6	0.0122	177222.0	606.3
OR5x100-29	100	5	59965	59646	59497.0	0.0053	247579.4	606.4
OR10x100-00	100	10	23064	22557	22337.2	0.0220	114937.4	607.4
OR10x100-29	100	10	60633	60626	60489.2	0.0001	159064.8	607.6
OR30x100-00	100	30	21946	21614	21546.8	0.0151	44897.0	609.6
OR30x100-29	100	30	60603	60503	60341.8	0.0017	57533.4	609.4
Instances executed with $max_time = 900$s								
OR5x250-00	250	5	59312	56380	55681.6	0.0494	119641.6	908.0
OR5x250-29	250	5	154662	153526	153227.6	0.0073	151808.6	908.2
OR10x250-00	250	10	59187	56414	56136.6	0.0469	75015.2	909.5
OR10x250-29	250	10	149704	148252	147977.2	0.0097	96986.0	909.3
OR30x250-00	250	30	56693	54578	54533.0	0.0373	24314.4	915.7
OR30x250-29	250	30	149572	148357	148232.0	0.0081	34610.0	915.6
Instances executed with $max_time = 1200$s								
OR5x500-00	500	5	120130	113326	112776.6	0.0566	71190.0	1215.3
OR5x500-29	500	5	299904	297679	297074.4	0.0074	86169.8	1210.4
OR10x500-00	500	10	117726	111974	111193.4	0.0489	40109.6	1214.2
OR10x500-29	500	10	307014	303614	303487.6	0.0111	53254.8	1213.9
OR30x500-00	500	30	115868	111358	111081.8	0.0389	10612.4	1226.6
OR30x500-29	500	30	300460	298569	298463.6	0.0063	16284.0	1228.8

Table 3: Results for big instances of the 0-1 MKNP under independent runs with $p = 2$ processors.

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	total time (s)
Instances executed with $max_time = 600$s								
OR5x100-00	100	5	24381	24211	24162.2	0.0070	1192054.6	619.8
OR5x100-29	100	5	59965	59667	59533.6	0.0050	1633099.8	620.8
OR10x100-00	100	10	23064	22478	22419.4	0.0254	753246.6	630.2
OR10x100-29	100	10	60633	60629	60627.2	0.0001	1001095.8	625.8
OR30x100-00	100	30	21946	21649	21623.0	0.0135	275664.4	624.9
OR30x100-29	100	30	60603	60503	60483.8	0.0017	364593.2	624.3
Instances executed with $max_time = 900$s								
OR5x250-00	250	5	59312	56759	56490.2	0.0430	804849.0	939.1
OR5x250-29	250	5	154662					
OR10x250-00	250	10	59187	56664	56405.8	0.0426	490034.4	926.0
OR10x250-29	250	10	149704	148414	148257.0	0.0086	623402.4	932.2
OR30x250-00	250	30	56693	54855	54762.4	0.0324	151330.6	936.5
OR30x250-29	250	30	149572	148617	148482.2	0.0064	222286.4	931.4
Instances executed with $max_time = 1200$s								
OR5x500-00	500	5	120130	114469	114070.0	0.0471	526232.0	1348.8
OR5x500-29	500	5	299904	297707	297453.8	0.0073	644637.2	1229.1
OR10x500-00	500	10	117726	112983	112254.0	0.0403	326128.0	1243.2
OR10x500-29	500	10	307014	304122	303849.2	0.0094	416856.2	1232.3
OR30x500-00	500	30	115868	111786	111464.2	0.0352	87063.6	1248.9
OR30x500-29	500	30	300460	298746	298687.6	0.0057	129835.6	1241.3

Table 4: Results obtained for big instances of the 0-1 MKNP under independent runs model with $p = 8$ processors.

B Experimental Results from Master-Slave Parallelization of the TS method.

Table 5 and Table 6 present further experimental results obtained from the execution of the Master-Slave parallel implementation for 0-1 Multidimensional Knapsack.

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	time total (s)
Instances executed with $max_time = 600$s								
OR5x100-00	100	5	24381	24381	24238.8	0	10441.6	598.9
OR5x100-29	100	5	59965	59945	59855.2	0.0003	6902.4	601.4
OR10x100-00	100	10	23064	22944	22822.0	0.0052	10340.4	600.6
OR10x100-29	100	10	60633	60633	60592.6	0	9335.6	600.6
OR30x100-00	100	30	21946	21719	21656.8	0.0103	5362.0	602.5
OR30x100-29	100	30	60603	60603	60349.6	0	4338.8	604.2
Instances executed with $max_time = 900$s								
OR5x250-00	250	5	59312	58900	58469.8	0.0069	1970.8	905.5
OR5x250-29	250	5	154662	154309	153998.8	0.0023	1402.6	910.4
OR10x250-00	250	10	59187	58076	57680.8	0.0188	1659.6	601.0
OR10x250-29	250	10	149704	148868	148292.6	0.0056	901.4	909.5
OR30x250-00	250	30	56693	55946	55490.2	0.0132	1672.2	905.7
OR30x250-29	250	30	149572	148761	148579.0	0.0054	768.6	920.3
Instances executed with $max_time = 1200$s								
OR5x500-00	500	5	120130	116969	115622.4	0.0263	426.2	1205.8
OR5x500-29	500	5	299904	297013	295104.6	0.0096	295.4	1226.4
OR10x500-00	500	10	117726	115919	114188.2	0.0153	592.2	1204.8
OR10x500-29	500	10	307014	303417	302680.8	0.0117	205.0	1214.0
OR30x500-00	500	30	115868	112898	112028.0	0.0256	759.6	1214.5
OR30x500-29	500	30	300460	298634	297754.4	0.0061	170.0	1207.5

Table 5: Results obtained for big instances of the 0-1 MKNP under Master-Slave model with $p = 2$ processors.

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	time total (s)
Instances executed with $max_time = 600$s								
OR5x100-00	100	5	24381	24282	24052.2	0.0041	1838.0	618.8
OR5x100-29	100	5	59965	59896	59792.6	0.0012	1723.0	620.4
OR10x100-00	100	10	23064	22717	22670.0	0.0150	1079.8	622.5
OR10x100-29	100	10	60633	60629	60507.4	0.0001	1040.4	619.5
OR30x100-00	100	30	21946	21616	21493.8	0.0150	430.4	626.4
OR30x100-29	100	30	60603	60123	60072.2	0.0079	387.2	629.0
Instances executed with $max_time = 900$s								
OR5x250-00	250	5	59312	58660	58274.2	0.0110	771.0	921.5
OR5x250-29	250	5	154662	154105	153643.0	0.0036	554.8	928.7
OR10x250-00	250	10	59187	57674	57227.4	0.0256	492.4	929.1
OR10x250-29	250	10	149704	148151	147805.2	0.0104	368.2	937.7
OR30x250-00	250	30	56693	55323	54937.8	0.0242	234.5	937.5
OR30x250-29	250	30	149572	148582	148259.4	0.0066	199.0	960.2
Instances executed with $max_time = 1200$s								
OR5x500-00	500	5	120130	115552	112096.8	0.0381	281.4	1233.3
OR5x500-29	500	5	299904	295432	293429.4	0.0149	191.0	1272.5
OR10x500-00	500	10	117726	114161	113312.2	0.0303	253.6	1224.6
OR10x500-29	500	10	307014	303469	303322.6	0.0115	187.2	1272.8
OR30x500-00	500	30	115868	110133	109422.2	0.0495	141.0	1240.0
OR30x500-29	500	30	300460	297681	297614.4	0.0092	97.6	1289.4

Table 6: Results obtained for big instances of the 0-1 MKNP under Master-Slave model with $p = 8$ processors.