Parallel Skeletons for Tabu Search Method Based on Search Strategies and Neighborhood Partition*

Maria J. Blesa† Lluís Hernàndez Fatos Xhafa

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

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

Abstract

In this paper we present two parallel skeletons for Tabu Search method –a well known meta-heuristic for approximately solving combinatorial optimization problems. Our parallel skeletons are designed and implemented using the generic parallel programming paradigm. The first skeleton is based on *independent runs model with search strategies* while the second is a *master-slave model with neighborhood partition*. Our starting point to obtain these skeletons was the design and implementation of a sequential skeleton that was 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 the existing sequential implementation of the problem. 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 and object oriented programming paradigms. We have instantiated the two skeletons for the 0-1 Multidimensional Knapsack problem and report extensive experimental results.

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

1 Introduction

Many interesting combinatorial optimization problems are shown *NP-hard* [GJ79] 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 [Glo77] (for its current form see [Glo86]) 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. [LBG91, Wid91, DT86, PR95]), graph problems (e.g. k-Cardinality Tree [JL97]), resource allocations (e.g. [SK90, Tai91]), layout problems (e.g. [Ste61, FW74, KP78]), to name a

*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.

†The work of this author was partially supported by the Catalan 2001FI-00659.

few. After a careful and almost exhaustive revision of such implementations, we have observed that all these implementations 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. 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. In [CTG95] the authors present a whole taxonomy of parallel TS strategies while in [CT97] 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. [Tai93, Fie94, PR96a]) 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.

A first approach to reduce this gap was done in [BHX01] where two simple generic parallel skeletons were presented for classical independent runs and master-slave models. Parallel generic programming paradigm has been used in several other contexts (e.g. [ELRT97, Sun97, KD98, LS98, JG99, Tro00]). 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, either sequential or parallel.

The first skeleton presented here is based on the classical independent runs model, but different search strategies are given to the independent runs. In this model there is a distinguished processor that distributes the same data to the processors, then each processor runs the TS independently parameterized with a different search strategy. At the end, the distinguished processor receives the solutions and reports the best solution as well as other statistical information. The classical independent run model, though trivial, is especially important in the context of TS since, in one hand, a good implementation for TS method requires extensive experimenting so as to identify adequate parameters that guide the search in the right way, and, in the other, running different search strategies simultaneously allows for a deeper exploration of the search space. The second skeleton is based on a classical master-slave model in which the work performed by a slave is to search a portion of the neighborhood of the current solution. There is a master processor that controls the search. At the beginning, it sends the same data to the rest of processors and waits until they finish a *tabu search iteration* consisting in *computing* a new solution by exploring the neighborhood of the current one. In the classical master-slave model every slave explores the whole (or a great part of) the neighborhood. Our approach makes this model more efficient since now the neighborhood is explored in parallel. Interestingly, we could design a generic way for partitioning the neighborhood of a given solution (this idea have been previously used in [Tai91, PR96b] though for concrete problems.) Afterwards, once each processor reports its best solution corresponding to the portion of the neighborhood explored and the master selects the best among all of them, hence the best solution in the neighborhood. Next, the master split the neighborhood of this new current solution up the slaves and the search is launched from this new solution.

Importantly, the skeletons offer automatic parallel implementations of the TS method for any given problem from an existing sequential implementation of it obtained from instantiating our sequential skeleton. For instance, since we dispose of a sequential implementation of TS for the 0-1 Multidimensional Knapsack obtained via the sequential skeleton, we automatically obtain these two parallel implementations of TS for the problem, by recombining the core of each parallel skeleton and the sequential implementation of the entities participating in the method. 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/).

The paper is organized as follows. We give some preliminaries on Tabu Search in Section 2 as to help readability of the paper. Our sequential skeleton –the basis for th two parallel skeletons– is given in Section 3. Then, we present the two parallel skeletons in Section 4. Experimental results are given in Section 4.4 (see also Appendices A and B), and finally, some conclusions and possible extensions of this work are given in Section 5.

2 Preliminaries 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.

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 $f: \mathcal{S} \to \mathbb{R}$ 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).

Main Entities of Tabu Search. In order to design a generic skeleton for the Tabu Search we had to identify the basic entities participating in the method. To this end, 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 programming paradigm. The goal is to obtain a generic skeleton for the tabu search method independently of the problem to be solved.

Problem. The instance of the problem to be solved.

Solution. Represents a feasible solution to the problem. Since TS is an heuristic method, 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. It is the set of all possible solutions, denoted by $\mathcal{N}(s)$, that are reached from a given solution s in a single step (called *move*). TS moves from s to $s' \in \mathcal{N}(s)$ which is *the best* among all (or part of) $\mathcal{N}(s)$.

Move. It is a transition between feasible solutions, more precisely it performs some local perturbation over the solution it is applied to and thus yielding another solution. Moves are given the *tabu status* if they lead to previously visited solutions, but TS also establishes an *aspiration criteria* so that tabu moves can be accepted if they satisfy it.

Tabu list. To prevent the search from cycling among the same solutions, TS uses a short term memory –the so-called tabu list– to represent the trajectory of solutions already visited by keeping track of already performed moves. Such moves will be forbidden from being selected in at least one subsequent iteration.

Intensification. It seems reasonable to *intensify* the search if we had evidence that the region being explored may contain good solutions. To this aim, TS incorporates an intensification procedure that allows the region to be deeply explored.

Diversification. TS method launches the *diversification* procedure to spread out the search in another region when no better solutions are found in the current (intensified) region. This allows the TS method to escape from local optima.

Main procedure. We have checked out several existing TS implementations and have observed that the main procedure is not standard since the authors implement it differently, using specific knowledge of 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 by abstracting from a large number of different implementations for TS. This component (called Solver) was going to be the principle engine of any program for TS obtained by instantiating the skeleton. The main procedure of the TS method uses the previous defined entities and concepts. This use can be completely specified (and hence the method itself) if a generic-enough interface is defined for each entity involved. Moreover it allows the method to be parallelized only by performing a parallel implementation of the main procedure.

3 Overview on the Sequential Skeleton for Tabu Search

In this section we show the sequential skeleton for Tabu Search. This skeleton is designed and implemented with the objective to be used later as a basis for parallel implementations although it is also suited as a stand-alone skeleton for sequential implementations of Tabu Search. In what follows we concentrate on implementation issues of the skeleton and will remark its particular functionalities that will be used by the parallel skeletons.

The main entities identified in Section 2 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-dependent 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 instantiated 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) 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`. All the classes defined and implemented in these files are comprised under a unique namespace, called *TabuSearch*.

Following we show in more detail how one of the entities–the Solution– abstracted in Section 2 has been translated into a C++ class.

3.1 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. The interface of class solution is the following:

```
requires class Solution {
  public:
    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);
    double fitness () const;
    double delta (const Movement& move) const;
    void   set_initial ();
    void   perturb_randomly ();
    void   apply ( const Movement& move );
    void   unapply ( const Movement& move );
    bool   aspiration (const Movement& move, const TabuStorage&tstore,const Solver& solver) const;
    void   reward ();
    void  unreward ();
    void   penalize ();
    void   unpenalize ();
    void   escape ();
    int  size () const;
    item operator[] (int i_th) const;
};
```

This generic but fixed interface is obtained by abstracting all the necessities we need at representing and implementing any feasible solution. 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. In some cases, however, it is easier to calculate the differential cost between the next solution and the current solution without applying the move that leads from one 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.

Tabu Search starts exploration from an initial solution generated. Typically, the initial solution is constructed randomly or by a greedy algorithm. The initial solution in the TS skeleton is obtained by implementing the method `set_initial()`. Thinking on the parallel implementation, if a deterministic (greedy) procedure is used, all the different searches would start from the same initial solution (i.e. at the same point of the solution space). In some cases we may want to perturb 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 `perturb_randomly()` changes locally and randomly a given solution. This method is used, in particluar, in the Independent Runs Model in order to compute different initial solutions (which in turn form part of different search strategies) to be send to the processors.

As in other local search methods, in Tabu Search 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 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.

Tabu Search will choose the best solution in the neighborhood (actually the best move that leads to it) as the next solution and continue the exploration. Moves in TS are given the *tabu status* if they lead to previously visited solutions. 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.

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 (method `unreward()`). Similarly, the methods `penalize()`, `unpenalize()` and `escape()` perform soft and hard diversification.

The last two methods of the class Solution, namely `size ()` and `operator[]` are functionalities to be used later on by the parallel implementation in order to partition the neighborhood of the solution. More concretely, a solution to a combinatorial optimization problem can be seen as a collection of elements of the same type, e.g. a permutation (collection of integers), a tree (set of edges), etc. Usually the exploration of the neighborhood uses the information contained in a Solution. It is useful to represent a Solution as a collection of items in order to allow a partial exploration of the neighborhood (e.g. the combinatorial neighborhood that can be generated by combining some items of a permutation). When running in parallel this will allow different processors to explore different parts of a unique neighborhood. It is needed to know the size of a Solution in terms of items as well as to extract an item from the solution in order to decide how to select the part of the neighborhood to explore. Note that implementing these two methods is not necessary for the sequential implementation, it is however necessary for the parallel implementation that uses neighborhood partition.

How should the user complete the class Solution, for its problem of interest, say for the 0-1 Multidimensional Knapsack Problem? First, the user needs to represent a feasible solution to the problem. Given the information about the benefits, the capacities and the constraints 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;[1]
}
```

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.

```
double Solution::fitness () const
{
    double cost = 0;
    for(int item=0; item<_problem.nb_items; item++)
        cost += _problem._benefits[item] * _inclusions[item];[2]

    return cost;
}
```

or the `operator[]`

```
item Solution::operator[] (int i_th) const
{
       return _inclusions[i_th];
}
```

3.2 Running the sequential skeleton

The class `Solver` is completely provided by the skeleton, so the user only uses it but he is not expected to do anything more. The skeleton is run through this class 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 functionality these methods are supposed to do. Once the instantiation is completely done, 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;
}
```

4 Our Parallel Skeletons for Tabu Search

The sequential TS skeleton has a *required part* whose requirements are specified through classes with fixed interfaces. The user instantiates them by providing a concrete representation for the

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

[2]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.

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 and using the required part as a black box. So, once the user has completed the instantiation of the sequential skeleton, his instantiation can also run in parallel almost effortlessly. We remark however (see Section 3) that the user has to implement, some functionalities of the sequential skeleton that are used exclusively by the parallel skeletons.

Tabu Search can be parallelized in several ways (see [CT97]) for a complete taxonomy of parallel tabu search heuristics). We present here two parallel implementations based on two different parallel models. The first implementation, namely, the direct parallelization by independent runs with search strategies is given in Subsection 4.1. The second implementation, given in Subsection 4.2, is based on the Master-Slave model in which the neighborhood exploration is done in parallel by the slaves, each slave exploring a part of the neighborhood.

The essentials of the two implementations, as far as the user is concerned, 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 additional 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".

4.1 Independent Runs Model with Autonomous Search Strategies

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 resulting solutions. Even in the simple case where all the processors have the same configuration, i.e. same starting solution and the same parameters such as size of tabu list etc., in this model many different search paths can *potentially* be explored simultaneously using the same time that a sequential model would use to explore only a unique path. In order to give autonomy to the processors and hence assure that each processor explores its own search path, a distinguished processor, called the coordinator processor, generates different *"strategies"* (as many as there are processors), sends them to the processors and then each processor executes the TS program according to the strategy assigned to it. A strategy consists mainly of an initial solution and other parameters related to tabu search such as tabu list size, number of intensifications, number of diversifications, minimum number of iterations a movement remains tabu, maximum number of iterations a movement remains tabu, and maximum number of repetions for a solution[3].

Clearly, in this model the main gain is in computation time. There is no relation between the number of independent 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 a sequential model. An important benefit of this model is that it allows an efficient fine tuning of parameters. Clearly, the success of Tabu Search depends on a careful choice of search parameters. These parameters are found through an experimental process and it is, in general, very time consuming. Hence, the Independent Runs model with search strategies is much more suited for an efficient fine tuning of parameters.

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 (we have used the MPICH implementation). The main drawback of this MPI implementation 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. But in order to start the whole process and gather the solutions found by different processors a coordinator processor is maintained. The

[3] All these parameters are taken from the class `Setup` of the sequential skeleton and are perturb according to a method implemented by the user in the sequential skeleton.

coordinator processor generates as many search strategies as there are processors and distributes them together with the input to all the processors. It waits until they finish the whole search and finally it collects the solutions found by the processors in order to select the best one and other statistical information. Note that the coordinator processor is not a master processor (in the sense of the Master-Slave model) and it doesn't participate in the search process. This model has a coarse grain parallelism.

4.2 Master-Slave Model with Neighborhood Partition

In the classic 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[4], 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 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 (the definition of subtasks may not always be possible.)

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 beginning of the parallel execution and the master is not allowed to create and destroy dynamically 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 solution. To this end, the master partition the current solution into parts (as many parts as slaves are) each of them of equal number of items. This is possible due to the methods `size()` and the `operator[]` of the class Solution (see Section 3). Then, the master sends one part to each slave who later using this part and the current solution explores just part of the neighborhood[5]. 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. Given that the neighborhood exploration is very time-consuming, this model endowed with the neighborhood partition has a better time efficiency.

4.3 Instantiating and Running the Parallel Skeletons

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 steps: (a) the identification and representation of the entities participating in the method and, (b) the implementation of the functionalities that these entities are expected to have according to the representation chosen. 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 functionalities is done through the implementation of the methods in the required classes. Both

[4]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–this is our case.

[5]Implementation detail will be given in a full version of the paper.

the concrete representation of the classes and the implementation of the methods are done "thinking in sequential". The parallelization remains completely hidden 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 instantiate the sequential skeleton and the same instantiation serves for both sequential an parallel execution.

The user only needs to declare an object of the class `Solver` implementing TS method via Independent Run or Master-Slave model and call the `run` method. Let us show it by example. The parallel execution using the Independent Runs model of an instantiated TS-skeleton can be done as follows:

```
#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;
}
```

4.4 Experimental Results for 0-1 Multidimensional Knapsack Problem

We have instantiated the parallel skeletons for the 0-1 Multidimensional Knapsack Problem. We give in Table 1 the experimental results under the Independent Runs model (more results are given in Table 3 and Table 4 in Appendix A). These results are obtained for six instances from the OR-Library [Bea90][6]. The execution is done with different maximum execution times and different number of processors. In the table, instance refers 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 [CB97] for the instance obtained by other specific implementations and best cost obtained indicates the best cost found by our parallel implementation, p indicates the number of processors.

instance	n	m	best cost known	best cost obtained	cost average obtained	dev.	iters. (average)	time total (s)
Instances solved with $max_time = 900$s								
OR5x250-00	250	5	59312	58417	57698.8	0.0151	3192.0	900.5
OR5x250-29	250	5	154662	153666	153297.0	0.0064	1360.6	902.1
OR10x250-00	250	10	59187	56376	55757.0	0.0475	1566.4	902.2
OR10x250-29	250	10	149704	148077	147831.6	0.0109	1152.8	901.9
OR30x250-00	250	30	56693	55412	55055.2	0.0226	1575.6	902.6
OR30x250-29	250	30	149572	148010	147720.4	0.0104	398.0	908.9

Table 1: Results obtained for middle-sized instances of the 0-1 MKNP under an Independent Runs Model with autonomous strategies using 4 processors.

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) .

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

instance	n	m	best cost known	best cost obtained	cost average obtained	dev.	iters. (average)	time total (s)
Instances solved with $max_time = 900$s								
OR5x250-00	250	5	59312	57838	57059.6	0.0249	747.2	912.5
OR5x250-29	250	5	154662	153518	153079.8	0.0074	484.4	916.1
OR10x250-00	250	10	59187	55352	54912.8	0.0648	278.4	917.8
OR10x250-29	250	10	149704	147865	147771.0	0.0123	363.6	913.0
OR30x250-00	250	30	56693	55082	54112.4	0.0284	142.8	927.0
OR30x250-29	250	30	149572	148142	148019.2	0.0096	129.6	915.2

Table 2: Results obtained for middle-sized instances of the 0-1 MKNP under a Master-Slave Model with neighborhood partition using 4 processors.

5 Current and Future Work

We have tested our parallel skeletons instantiated for the 0-1 Multidimensional Knapsack Problem with instances from OR-Library. We are currently testing with other instances from OR-Library and will test with the benchmark from Freville and Plateau [FP90]. We will also test our parallel skeletons instantiated with other problems and will compare our results (both quality of solutions and time-efficiency) with other known results in the literature.

We plan to simplify more the parallel executions so that by simply checking the underlying system where the execution will take place, the skeleton will be expected to configure and run a parallel execution automatically.

References

[Bea90] J.E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *J. of Op. Res. Soc.* 41:11,1069–1072. http://mscmga.ms.ic.ac.uk/info.html, 1990.

[BHX01] M. Blesa, Ll. Hernandez, and F. Xhafa. Parallel Skeletons for Tabu Search Method. In *Proceedings of International Conference on Parallel and Distributed Systems, ICPADS'01*, IEEE, 2001. *To appear.*

[CB97] P.C. Chu and J.E. Beasley. A Genetic Algorithm for the Multidimensional Knapsack Problem. working paper, 1997.

[CT97] T.G. Crainic and M. Toulouse. Parallel metaheuristics. Technical report, Dépt. des sciences administratives (Université du Québec a Montréal) and School of Computer Science (University of Oklahoma), 1997.

[CTG95] T.G. Crainic, M. Toulouse, and M. Gendreau. Towards a Taxonomy of Parallel Tabu Search Heuristics. Technical report, Dépt. d'informatique et de recherche opérationnelle. Université de Montréal, 1995.

[DT86] M. Dell'Amico and M. Trubian. Applying Tabu Search to the Job-Shop Scheduling Problem. *Ann. of Op. Res.*, 41:231–252, 1986.

[ELRT97] M. Eso, L. Ladanyi, T.K. Ralphs, and L.E. Trotter. Fully Parallel Generic Branch-and-Cut Framework. In *Proc. of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[Fie94] C.N. Fiechter. A Parallel Tabu Search Algorithm for Large Travelling Salesman Problem. *Discrete Applied Mathematics*, 51:243–267, 1994.

[FW74] R.L. Francis and J.A. White. *Facility Layout and Location.* Prentice-Hall, 1974.

[FP90] A. Freville and G. Plateau. Hard 0-1 Multiknapsack Testproblems for Size Reduction Methods *Investigacion Operativa*, 1:251–270, 1990.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[Glo77] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[Glo86] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Op. Res.*, 5:533–549, 1986.

[JG99] M. Sato J. Gerlach. Generic programming for parallel mesh problems. In Tholburn, Matsuoka, Oldehoeft, editor, *3d Int. Symp. on Comp. in Object-Oriented Parallel Environments, ISCOPE '99*, volume 1732 of *LNCS*, 108–119. Springer Verlag, 1999.

[JL97] K. Jörnsten and A. Løkketangen. Tabu Search for Weighted k-Cardinality Trees. *Asia-Pacific J. of Op. Res.*, 14(2):9–26, 1997.

[KD98] E. Kaltofen and A. Diaz. Generic Programming in the POOMA Framework. In *Generic Programming Followup to a Dagstuhl Seminar*, 1998.

[KP78] J. Krarup and P.M. Pruzan. Computer-aided Layout Design. *Math. Prog. Study*, 9:75–94, 1978.

[LBG91] 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.

[LS98] A. Lumsdaine and J. Siek. Generic Programming for High Performance Numerical Linear Algebra. *Workshop on Modern Software Tools for Sci. Comp.*, 1998.

[PR95] S. Porto and C. Ribeiro. A Tabu Search Approach to Task Scheduling on Heterogeneous Processor under Precedence Constraints. *J. of High-Speed Comp.*, 7, 1995.

[PR96a] C.S. Porto and C. Ribeiro. Parallel Tabu Search message-passing Synchronous Strategies for Task Scheduling Under Procedence Constraints. *Journal of Heuristics*, 1(2):207–223, 1996.

[PR96b] S. Porto and C. Ribeiro. A Case Study on Parallel Synchronous Implementations of Tabu Search Based on Neighborhood Decomposition. *Investigacion Operativa*, 5:233–259, 1996.

[SK90] J. Skorin-Kapov. Tabu Search Applied to the Quadratic Assignment Problem. *ORSA J. on Comp.*, 2(1):33–45, 1990.

[Ste61] L. Steinberg. The Backboard Wiring Problem : A Placement Algorithm. *SIAM Review*, 3:37–50, 1961.

[Sun97] N. Sundaresan. Generic parallel programming in c++. Colloquia 1997,University of California, Davis, 1997.

[Tai91] E. Taillard. Robust Tabu Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[Tai93] E. Taillard. Parallel Iterative Search Methods for Vehicle Routing Problem. *Networks*, 23:661–673, 1993.

[Tro00] Leslie Trotter. Generic parallel implementation for integer programming. DONET Spring School Computational Combinatorial Optimization, Dagstuhl, 2000.

[Wid91] M. Widmer. The Job-shop Scheduling with Tooling Constraints: A Tabu Search Approach. *J. Op. Res.*, 42:75–82, 1991.

A Experimental Results from IR with Strategies

instance	n	m	best cost known	best cost obtained	cost average obtained	dev.	iters. (average)	time total (s)
Instances solved with *max_time* = 900s								
OR5x250-00	250	5	59312	57645	57223.8	0.0281	1238.4	900.5
OR5x250-29	250	5	154662	153871	152940.8	0.0051	782.8	900.6
OR10x250-00	250	10	59187	56513	55685.0	0.0452	956.4	900.6
OR10x250-29	250	10	149704	147781	147739.4	0.0128	416.6	902.2
OR30x250-00	250	30	56693	55346	54872.2	0.0238	674.2	901.2
OR30x250-29	250	30	149572	148147	147761.8	0.0095	223.2	903.2

Table 3: Results obtained for middle-sized instances of the 0-1 MKNP under an Independent Runs Model with autonomous strategies using 2 processors.

instance	n	m	best cost known	best cost obtained	cost average obtained	dev.	iters. (average)	time total (s)
Instances solved with *max_time* = 600s								
OR5x250-00	250	5	59312	58106	57713.2	0.0203	8922.8	901.1
OR5x250-29	250	5	154662	154188	153972.0	0.0031	4407.8	901.3
OR10x250-00	250	10	59187	56496	56198.8	0.0455	4435.4	905.3
OR10x250-29	250	10	149704	149030	148840.2	0.0045	3762.8	903.3
OR30x250-00	250	30	56693	55725	55494.6	0.0170	4764.8	902.3
OR30x250-29	250	30	149572	149034	148969.6	0.0036	2231.4	903.1

Table 4: Results obtained for middle-sized instances of the 0-1 MKNP under an Independent Runs Model with autonomous strategies using 8 processors.

B Experimental Results from Master-Slave with Neighborhood Partition

instance	n	m	best cost known	best cost obtained	cost average obtained	dev.	iters. (average)	time total (s)
Instances solved with *max_time* = 900s								
OR5x250-00	250	5	59312	56258	55362.2	0.0515	271.0	909.1
OR5x250-29	250	5	154662	152793	152473.6	0.0121	252.2	911.1
OR10x250-00	250	10	59187	54510	54135.6	0.0790	114.0	923.6
OR10x250-29	250	10	149704	147777	147590.0	0.0129	209.4	911.4
OR30x250-00	250	30	56693	54317	53820.0	0.0419	95.0	921.3
OR30x250-29	250	30	149572	147155	147111.6	0.0162	57.0	948.5

Table 5: Results obtained for middle-sized instances of the 0-1 MKNP under a Master-Slave Model with neighborhood partition using 2 processors.

instance	n	m	best cost known	best cost obtained	cost average obtained	dev.	iters. (average)	time total (s)
Instances solved with *max_time* $= 900$s								
OR5x250-00	250	5	59312	58085	57923.0	0.0207	1223.8	919.5
OR5x250-29	250	5	154662	153636	153283.2	0.0066	746.4	922.7
OR10x250-00	250	10	59187	55374	54941.8	0.0644	505.4	916.5
OR10x250-29	250	10	149704	148220	147954.2	0.0099	669.2	915.4
OR30x250-00	250	30	56693	54951	54386.4	0.0307	269.0	917.5
OR30x250-29	250	30	149572	148424	148099.2	0.0077	221.4	921.5

Table 6: Results obtained for middle-sized instances of the 0-1 MKNP under a Master-Slave Model with neighborhood partition using 8 processors.