Parallel Skeletons for Tabu Search Method*

Maria J. Blesa Lluís Hernàndez Fatos Xhafa
Dept. de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Jordi Girona 1-3 C6, E-08034 Barcelona, Spain
{mjblesa,ba-pfc1,fatos}@lsi.upc.es

Abstract

In this paper we present two generic parallel skeletons for Tabu Search method –a well known meta-heuristic for approximately solving combinatorial optimization problems. 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 of Tabu Search for a concrete problem is obtained automatically from a sequential implementation of Tabu Search for 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 communication library and offer genericity, flexibility, component reuse, robustness and time savings. We have instantiated the two skeletons for the 0-1 Multidimensional Knapsack problem, among others, for which we report computational results.

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. It was introduced by Glover ([11], [12]) and has been used to sub-optimally solving a wide range of important optimization problems from theory and practice, in both sequential and parallel settings.

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. [16, 26, 7, 19]), graph problems (e.g. [14]), resource allocations (e.g. [20, 23]), layout problems (e.g. [21, 9, 15]), to name a few. 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.

In combinatorial optimization we are often encountered with *methods* for sub-optimally solve optimization problems and such methods apply almost in the same way to all problems (for example, the ingredients of Tabu Search 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* 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 TS from a generic programming paradigm (see [3]). 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. These properties are achieved by a careful design of the skeleton identifying the common entities of the TS method.

TS have also been considered in the *parallel setting* to exploit the advantages of the parallelism. Parallelism permits to use more resources to obtain good solutions in rea-

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

0-7695-1153-8/01 $10.00 © 2001 IEEE

sonable computing times. Parallel versions of heuristics also 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 [6] the authors present a whole taxonomy of parallel TS strategies while in [5] 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. [24, 8, 18]) are *ad hoc* implementations. So, while from a theoretical 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. To reduce this gap we have designed and implemented (in C++ using MPI Library) two parallel skeletons for TS from 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. [17, 22, 13, 25]). 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. From a sequential implementation of TS for a given problem (via our sequential TS skeleton), the user 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 and statistical information. This kind of parallelism though trivial is especially important in the context of TS since it requires extensive experimenting so as to identify adequate parameters that guide the search. Such a fine tuning usually results in time consuming, hence the independent runs parallelism permits a better use of resources and clear time savings.

The second skeleton is based on the 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 *TS iteration* consisting in *computing* a new solution in the solution space by exploring the neighborhood of the current solution. Once each processor reports its proposed next solution, the master selects the best among all of them, sends it back to the processors and the search is launched from this new solution.

Importantly, the skeletons offer an automatic parallel implementation of the TS method for any given problem from an existing sequential implementation. There is no need for the user to know neither parallel programming techniques nor communication libraries. 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 instantiated our parallel skeletons to 0-1 Multidimensional Knapsack and have executed them over an homogeneous PC cluster at our department (www.lsi.upc.es/~mallba/BA-Cluster/), using Linux and MPI as communication library. 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 together with some details of the instantiation for the 0-1 Multidimensional Knapsack. In Section 3 we present the parallel skeletons, the one based on independent runs model (Subsection 3.1) and that of master-slave model (Subsection 3.2). Some experimental results are also given. We conclude in Section 4 with some remarks and outline future work.

2. Sequential Skeleton for Tabu Search

TS 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 moves which take the solution , in the next iteration, to solutions previously visited (called *"tabu"*). To this aim, TS keeps a memory (*tabu list*) which is historical in nature.

2.1. Main Entities of Tabu Search.

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 $\mathcal{S}(x)$; (c) a cost function $f : \mathcal{S} \rightarrow \mathbb{R}$ associating a cost $f(s)$ to solutions $s \in \mathcal{S}$. The goal is to find an optimum solution $s^* \in \mathcal{S}$ with respect to f and an optimization 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. 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. The objective is to find a solution that maximizes the whole benefit. Hence, we have the following entities:

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.

2.2. Design and Implementation

The main entities mentioned above are easily introduced into either C++ classes or methods according to a logical definition in the context of the TS method and Object Oriented Programming paradigm. Some of them 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, except for the main procedure, the concrete representation of the entities defined above depends 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.

The classes forming the skeleton are structured and labelled according to their *"availability"*. The ones implementing inner functionalities of the method (e.g. the main procedure) are completely provided by the skeleton, whereas there are some classes whose implementations are required to be instantiated by the user. Therefore, the classes forming the skeleton are classified into two groups:

(a) Provided Classes: They implement the TS method itself and the rest of inner functionalities. There are only two in the skeleton: the class Solver and the class Setup. The class Solver implements the main procedure of the TS, maintains the state of the exploration and supplies methods to allow the parallelization of the skeleton (see Subsections 3.1 and 3.2). The class Setup contains the parameters needed to run the method (e.g. number of intensifications, tabu list size, etc.). Moreover, the user can consult the state of the exploration process.

(b) Required Classes: They represent the rest of the entities and functionalities involved in the TS method whose implementation depends on the problem being solved. We have abstracted the necessities of each entity but the way they are carried out when solving a problem depends strongly on the problem itself. All this leads us to define C++ classes with a fixed interface but no implementation, so the Solver can use them in a "blind and generic way."

We have separated the C++ classes of the skeleton in three parts (see Figure 1) sharing a unique name-space: (1) class interfaces at file `TabuSearch.hh`, (2) implementation of the *provided classes* at `TabuSearch.pro.cc` and, (3) implementation of the *required classes* at `TabuSearch.req.cc`.

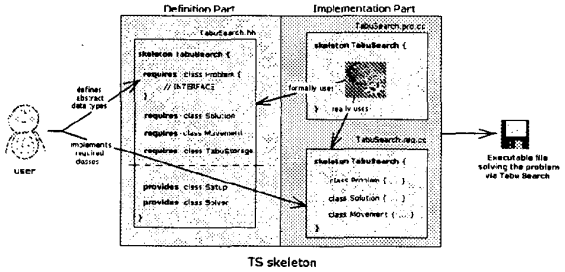

Figure 1. File composition of the TS-skeleton.

The *instantiation* of the skeleton for a given problem is the process of completing the requirements of the classes labelled as *required* with the features of the problem at hand. That means choosing data types for representing the entities and implementing the methods of the *required* classes according to the chosen data types. We show how some of the entities and concepts abstracted in Section 2 have been translated into classes and methods, and also some details about how to instantiate the skeleton for the Knapsack problem.

The provided class Solver represents the TS Method itself and all the internal features related to the search. A hierarchy is defined over class Solver in order to differentiate among sequential and parallel solvers. Part of the interface for class Solver follows:

```
provides class Solver {
  public:
      Solver (const Problem& p, const Setup& s);
      virtual ~Solver ();
      const Problem& problem () const;
      const Setup&   setup () const;

      virtual void run () =0;
      virtual void perform_one_independent_run () =0;
      virtual void perform_one_phase () =0;
      void set_current_solution (const Solution& s);
      ... };
```

The required class Problem represents the instance of the problem to be solved.

```
requires class Problem {
  public:
    Problem ();
    ~Problem();
    ostream& operator<<(ostream& o,const Problem& p);
    istream& operator>>(istream& i,Problem& p);
    opacket& operator<<(opacket& o,const Problem& p);
    ipacket& operator>>(ipacket& i,Problem& p);
    Direction direction () const;
};
```

To instantiate the 0-1 Multidimensional Knapsack Problem it is needed a representation for the benefits, the capacities and the constraint matrix. This representation is then added to the previous class as private attributes.

```
private:    int _nb_items, _nb_constraints;
            array2<double> _constraints;
            array<double>  _benefits, _capacities;
```

The direction of the problem is maximization, then:

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

Running the sequential skeleton. Once the instantiation is completed, the user may run it with a simple program that declares a sequential solver and calls the run method:

```
#include "TabuSearch.hh"

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

    Problem problem;
    ifstream f1(argv[1]);
    f1 >> problem;          // Problem from a file
    Setup setup;
    cin >> setup;           // Setup from the stdin

    Solver_Seq solver(problem,setup);
    solver.run();
    cout << solver.best_solution() << endl;
    cout << solver.best_cost()     << endl;
    return 0;
}
```

3. Parallel Skeletons for Tabu Search

Due to the separation of *required* and *provided* classes, 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.

TS can be parallelized in multiple ways. By now, we have implemented two different parallel versions of the skeleton: the direct parallelization by independent runs (IR) and the Master-Slave (M-S) model. 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 TS. So the user does not need to "think in parallel". (See the complete version of the paper [2] for more details on implementation issues and full experimental results).

3.1. Independent Runs Model

This model consists of simultaneous and independent executions of the same program in order to lately select the best solution among all the solutions obtained. In this model many different search paths can be explored simultaneously using the same time that a sequential model would use to explore a unique path. Clearly, 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 compared to a sequential model. In particular, we have implemented a slight variation of this model in which, the coordinator processor generates *strategies* (an initial solution and some setup parameters) and then each processor executes the TS program according to its strategy. We have implemented and tested the IR model over the BA-Cluster. The main drawback of the MPI implementation used (MPICH) is that it spends much time initializing the parallel processes. Neither it allows dynamic process creation, so the processors to be used are determined at the beginning of the parallel execution. 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 a coordinator is maintained. The coordinator processor distributes the input to all the processors and 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 M-S model). The IR model has a coarse grain parallelism.

3.2. The Classic Master-Slave Model

In the classic Master-Slave model 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's work. The master processor spawns slaves processors, initializes them, assigns subtasks and collects their results. Then it computes a final result from the results obtained by the slaves and incorporates it (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 this model is a one-to-many utilization relation.

The M-S model has some clear advantages: (a) flexibility and scalability, e.g. the slaves 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. The M-S model, however, can represent some disadvantages,such as machine dependency and no-feasibility, the definition of subtasks may not always be possible.

We have implemented and tested the M-S model over the BA-Cluster. The master is not allowed to create and destroy dynamically the slaves due to the MPI implementation used (MPICH). The master processor runs the TS method and uses slaves to explore in parallel the neighborhood and to choose the move that leads to the "best" next solution (note that the exploration should not be deterministic). Different M-S models have been implemented according to the nature of moves and the way the neighborhood is explored.

Notice that there is no a proper instantiation to the parallel skeletons. Both, the concrete representation of the classes and the implementation of their methods are done "thinking in sequential". The parallelization remains hidden to the user and does not influence his instantiation. The user has to instantiate the sequential skeleton and the same instantiation also serves for parallel implementations.

Running the parallel skeletons. The parallel skeleton is run as explained in Subsection 2.2. The user only needs to declare an object of the class Solver implementing TS method via IR or M-S model (i.e. to select the appropriate subclass of Solver) and to call the run method.

```
#include "TabuSearch.hh"
...
int Main (int argc, char** argv) {
    ...
    Solver_IR solver(problem,setup);  // IR model solver
    solver.run();
    ...
}
```

instance	n	m	best cost known	best cost found	average cost found	dev.	iters. (average)	total time (s)
Instances executed with *max_time* = 600s								
OR5x100-00	100	5	24381	24329	24244.6	0.0021	25710.0	601.0
OR5x100-29	100	5	59965	59955	59919.4	0.0002	24355.2	600.6
OR10x100-00	100	10	23064	23011	22931.0	0.0023	23355.8	601.1
OR10x100-29	100	10	60633	60633	60553.6	0	18109.0	601.7
OR30x100-00	100	30	21946	21946	21780.2	0	17297.4	601.3
OR30x100-29	100	30	60603	60603	60410.8	0	11961.6	602.1
Instances executed with *max_time* = 900s								
OR5x250-00	250	5	59312	58908	58636.8	0.0068	6784.8	904.3
OR5x250-29	250	5	154662	154446	154132.3	0.0014	3362.8	908.0
OR10x250-00	250	10	59187	58090	58015.6	0.0185	5204.4	907.3
OR10x250-29	250	10	149704	149009	148767.0	0.0046	2782.6	915.1
OR30x250-00	250	30	56693	55909	55648.2	0.0138	4404.8	908.2
OR30x250-29	250	30	149572	148901	148621.0	0.0045	1326.8	934.3
Instances executed with *max_time* = 1200s								
OR5x500-00	500	5	120130	116561	113983.6	0.0297	1219.4	1244.7
OR5x500-29	500	5	299904	296956	295761.0	0.0098	813.0	1265.8
OR10x500-00	500	10	117726	115241	114707.0	0.0211	1740.0	1225.5
OR10x500-29	500	10	307014	302425	302196.2	0.0149	606.0	1205.2
OR30x500-00	500	30	115868	113453	112673.2	0.0208	1945.0	1226.5
OR30x500-29	500	30	300460	298193	297565.0	0.0075	487.0	1277.4

Figure 2. Results for MKNP instances under IR model using 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* = 600s								
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* = 900s								
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* = 1200s								
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

Figure 3. Results for 0-1 MKNP instances under M-S model using 4 processors.

3.3. Experimental Results for 0-1 Multidimensional Knapsack Problem

We have instantiated both parallel skeletons for 0-1 Multidimensional Knapsack Problem. Part of the experimental results using four processors are given in Table 2 (for IR model) and Table 3 (for M-S model). These results are obtained for six instances from the OR-Library [1][1] and have been executed with different maximum execution times and different number of processors. The column header 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 [4] for the instance and best cost found indicates the best cost found by our parallel implementation. More results can be found at [2] and www.lsi.upc.es/~mjblesa/TSExperiments/knapsack.html.

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

4. Conclusions and Future Work

In this paper we have presented two parallel skeletons for TS 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 first designed and implemented a sequential skeleton. The user obtains parallel implementations of TS method for concrete combinatorial optimization problems from existing sequential implementations. There is no need for the user to know neither parallel programming nor communication libraries. 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 robustness and genericity due to the generic programming and object oriented programming. In spite of being generic, there is no loose neither in efficiency nor in the quality of solutions. We are actually testing our skeletons on other combinatorial optimization problems, concretely for Quadratic Assignment Problem, k-Cardinality Tree Problem, Resource-Constrained Project Scheduling Problem, among others, and comparing both the quality of solutions and efficiency of our implementations as compared to ad hoc implementations for these problems.

We plan to implement other variants of parallel skeletons for TS such as Master-Slave with strategies and Master-Slave with neighborhood partition, again from the generic parallel programming paradigm. We also plan to simplify even more the execution process of the parallel programs, in a sense, hide even more the parallelism. A cleverer parallelism can be obtained by checking the underlying system where the execution will take place. We want our skeletons to automatically configure the execution according to this underlying system. This auto-configuration should be accurate-enough in order to take the maximum profit from the number of available processors, their load and so on.

References

[1] J. 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.

[2] M. Blesa, L. Hernàndez, and F. Xhafa. Parallel Skeletons for Tabu Search Method. TechRep LSI-00-81-R, Dept. de LSI, UPC, 2000.

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

[4] P. Chu and J. Beasley. A Genetic Algorithm for the multidimensional knapsack problem. working paper, 1997.

[5] T. Crainic and M. Toulouse. Parallel metaheuristics. TechRep, Dépt. des sciences administratives (Université du Québec a Montréal) and Centre de recherche sur les transports (Université de Montréal), 1997.

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

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

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

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

[10] M. Garey and D. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freeman, 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. S. Jens Gerlach. Generic programming for parallel mesh problems. In M. T. Satoshi Matsuoka, R. R. Oldehoeft, editor, *3d Int. Symp. on Comp. in OO Parallel Environments*, v.1732 of *LNCS*, 108–119. Springer Verlag, 1999.

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

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

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

[17] T. R. M. Eso, L. Ladanyi and L. T. Jr. Fully parallel generic branch-and-cut framework. In *Procs. of the 8th SIAM Conf. on Parallel Proc. for Scientific Comp.*, 1997.

[18] C. 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.

[19] 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.

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

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

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

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

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

[25] L. Trotter. Generic parallel implementation for integer programming. DONET Spring School Dagstuhl, 2000.

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