
ELSEVIER

Parallel Computing 24 (1998) 2003–2019

PARALLEL COMPUTING

A parallel adaptive tabu search approach

E.G. Talbi *, Z. Hafidi, J-M. Geib

LIFL URA-369 CNRS, Université de Lille 1, Bâtiment M3 59655, Villeneuve d'Ascq Cedex, France

Received 15 April 1997

Abstract

This paper presents a new approach for parallel tabu search based on adaptive parallelism. Adaptive parallelism was used to dynamically adjust the parallelism degree of the application with respect to the system load. Adaptive parallelism demonstrates that high-performance computing using a hundred of heterogeneous workstations combined with massively parallel machines is feasible to solve large optimization problems. The parallel tabu search algorithm includes different tabu list sizes and new intensification/diversification mechanisms. Encouraging results have been obtained in solving the quadratic assignment problem. We have improved the best known solutions for some large real-world problems. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Tabu search; Adaptive parallelism; Quadratic assignment problem

1. Motivation and goals

Many interesting combinatorial optimization problems are NP-hard, and then they cannot be solved exactly within a reasonable amount of time. Consequently, heuristics must be used to solve real-world problems. Tabu search (TS) is a general purpose heuristic (meta-heuristic) that has been proposed by Glover [1]. TS has achieved widespread success in solving practical optimization problems in different domains (such as resource management, process design, logistic and telecommunications). Promising results of applying TS to a variety of academic optimization problems (traveling salesman, quadratic assignment, time-tabling, job-shop scheduling, etc.) are reported in the literature [2]. Solving large problems motivates the development of a parallel implementation of TS.

* Corresponding author. E-mail: talbi@lifl.fr

PII: S0167-8191(98)00086-6

The proliferation of powerful workstations and fast communication networks (ATM, Myrinet, etc.) with constantly decreasing cost/performance ratio have shown the emergence of heterogeneous workstation networks and homogeneous clusters of processors (such as DEC Alpha farms and IBM SP/2) [3,4]. These parallel platforms are generally composed of an important number of machines shared by many users. In addition, a workstation belongs to an owner who will not tolerate external applications degrading the performance of his machine. Load analysis of those platforms during long periods of time showed that only a few percentage of the available power was used [5,6]. There is a substantial amount of idle time. Therefore, dynamic adaptive scheduling of parallel applications is essential.

Many parallel TS algorithms have been proposed in the literature. In general, they don't use advanced programming tools (such as load balancing, dynamic reconfiguration and checkpointing) to efficiently use the machines. Most of them are developed for dedicated parallel homogeneous machines.

Our aim is to develop a parallel adaptive TS strategy, which can benefit greatly from a platform having combined computing resources of massively parallel machines (MPPs) and networks of workstations (NOWs). For this purpose, we use a dynamic scheduling system (MARS [1]) which harnesses idle time (keeping in mind the ownership of workstations), and supports adaptive parallelism to dynamically reconfigure the set of processors hosting the parallel TS.

The testbed optimization problem we used is the quadratic assignment problem (QAP), one of the hardest among the NP-hard combinatorial optimization problems. The parallel TS algorithm includes different tabu list sizes and intensification/diversification mechanisms based on frequency based long-term memory and restricted neighborhood.

The remainder of the paper is organized as follows. In Section 2, we describe existing parallel TS algorithms. The parallel adaptive TS proposed will be detailed in Section 3. Finally, Sections 4 and 5 will present respectively the application of the proposed algorithm to the QAP and results of experiments for several standard instances from the QAP-library.

2. Classification of parallel TS algorithms

We present in this section, respectively the main components of a sequential TS algorithm, and a classification of parallel TS algorithms. A new taxonomy dimension has been introduced.

2.1. Sequential tabu search

A combinatorial optimization problem is defined by the specification of a pair (X, f), where the search space X is a discrete set of all (feasible) solutions, and the

[1] Multi-user Adaptive Resource Scheduler.

objective function f is a mapping $f : X \to R$. A neighborhood N is a mapping $N : X \to P(X)$, which specifies for each $S \in X$ a subset $N(S)$ of X of neighbors of S.

The most famous local search optimization method is the *descent method*. A descent method starts from an initial solution and then continually explores the neighborhood of the current solution for a better solution. If such a solution is found, it replaces the current solution. The algorithm terminates as soon as the current solution has no neighboring solution of better quality. Such a method generally stops at a local but not global minimum.

Unlike a descent method, TS uses an adaptive memory H to control the search process. For example, a solution S' in $N(S)$ may be classified tabu, when selecting a potential neighbor of S, due to memory considerations. $N(H, S)$ contains all neighborhood candidates that the memory H will allow the algorithm to consider. TS may be viewed as a variable neighborhood method: each iteration redefines the neighborhood, based on the conditions that classify certain moves as tabu.

At each iterations, TS selects the best neighbor solution in $N(H, S)$ even if this results in a worst solution than the current one. A form of short-term memory embodied in H is the tabu list T that forbid the selection of certain moves to prevent cycling.

To use TS for solving an optimization problem, we must define in the input the following items:

- An initial solution S_0.
- The definition of the memory H.
- The stopping condition: there may be several possible stopping conditions [7]. A maximum number *nbmax* of iterations between two improvements of f is used as the stopping condition.

The output of the algorithm represents the best solution found during the search process. The following is a straightforward description of a sequential basic TS algorithm (Fig. 1) [8].

A tabu move applied to a current solution may appear attractive because it gives, for example, a solution better than the best found so far. We would like to accept the move in spite of its status by defining *aspiration conditions*. Other advanced techniques may be implemented in *a long-term-memory* such as intensification to encourage the exploitation of a promising region in the search space, and diversification to encourage the exploration of new regions [2].

2.2. Parallel tabu search

Many classifications of parallel TS algorithms have been proposed [9,10]. They are based on many criteria: number of initial solutions, identical or different parameter settings, control and communication strategies. We have identified two main categories (Fig. 2).

Domain decomposition: Parallelism in this class of algorithms relies exclusively on:

(i) **The decomposition of the search space:** the main problem is decomposed into a number of smaller subproblems, each subproblem being solved by a different TS algorithm [11].

Step 1 : Initialization

- Choose an arbitrary initial solution $S_0 \in X$ $(S = S_0)$
- $nbiter = 0$ /* current iteration */
- $bestiter = 0$ /* iteration where the best solution has been found */
- $bestsol = S_0$ /* best solution found */
- $H = \emptyset$ /* The memory is empty */

Step 2 : Iteration

- **While** $(nbiter - bestiter < nbmax)$ **Do**
  - $nbiter = nbiter + 1$
  - Choose a solution S^* such that $f(S^*) = min(f(x)/x \in N(H, S))$
  - Update the memory H
  - Update N(H,S)
  - **If** $f(S^*) < f(bestsol)$ **Then** $bestsol = S^*$ $bestiter = nbiter$;
  - $S = S^*$

Fig. 1. A basic sequential tabu search algorithm.

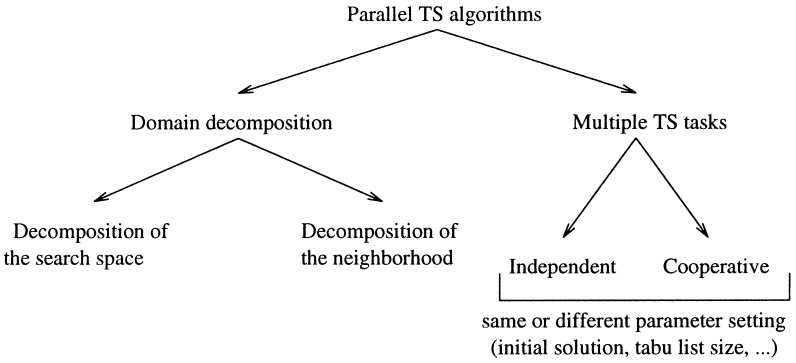

Fig. 2. Hierarchical classification of parallel TS strategies.

(ii) **The decomposition of the neighborhood:** the search for the best neighbour at each iteration is performed in parallel, and each task evaluates a different subset of the partitioned neighborhood [12,13].

A high degree of synchronisation is required to implement this class of algorithms.

Multiple tabu search tasks: This class of algorithms consists in executing multiple TS algorithms in parallel. The different TS tasks start with the same or different parameter values (initial solution, tabu list size, maximum number of iterations, etc.). Tabu tasks may be independent (without communication) [14,15] or cooperative. A cooperative algorithm has been proposed in [10], where each task performs a

given number of iterations, then broadcasts the best solution. The best of all solutions becomes the initial solution for the next phase.

Parallelizing the exploration of the search space or the neighborhood is problem-dependent. This assumption is strong and is met only for few problems. The second class of algorithms is less restrictive and then more general. A parallel algorithm that combines the two approaches (two-level parallel organization) has been proposed in [16].

We can extend this classification by introducing a new taxonomy dimension: the way scheduling of tasks over processors is done. Parallel TS algorithms fall into three categories depending on whether the number and/or the location of work (tasks, data) depend or not on the load state of the parallel machine (Table 1):

Non-adaptive: This category represents parallel TS in which both the number of tasks of the application and the location of work (tasks or data) are generated at compile time (static scheduling). The allocation of processors to tasks (or data) remains unchanged during the execution of the application regardless of the current state of the parallel machine. Most of the proposed algorithms belong to this class.

An example of such an approach is presented in [17]. The neighborhood is partitionned in equal size partitions depending on the number of workers, which is equal to the number of processors of the parallel machine. In [13], the number of tasks generated depends on the size of the problem and is equal to n^2, where n is the problem size.

When there are noticeable load or power differences between processors, the search time of the non-adaptive approach presented is derived by the maximum execution time over all processors (highly loaded processor or the least powerful processor). A significant number of tasks are often idle waiting for other tasks to complete their work.

Semi-adaptive: To improve the performance of the parallel non adaptive TS algorithms, dynamic load balancing must be introduced [17,16]. This class represents applications for which the number of tasks is fixed at compile-time, but the locations of work (tasks, data) are determined and/or changed at run-time (as seen in Table 1). Load balancing requirements are met in [17] by a dynamic redistribution of work between processors. During the search, each time a task finishes its work, it proceeds to a work-demand. Dynamic load balancing through partition of the neighborhood is done by migrating data.

However, the parallelism degree in this class of algorithms is not related to load variation in the parallel system: when the number of tasks exceeds the number of idle

Table 1
Another taxonomy dimension for parallel TS algorithms

	Tasks or Data	
	Number	Location
Non-adaptive	Static	Static
Semi-adaptive	Static	Dynamic
Adaptive	Dynamic	Dynamic

nodes, multiple tasks are assigned to the same node. Moreover, when there are more idle nodes than tasks, some of them will not be used.

Adaptive: A *parallel adaptive program* refers to a parallel computation with a dynamically changing set of tasks. Tasks may be created or killed function of the load state of the parallel machine. Different types of load state dessimination schemes may be used [18]. A task is created automatically when a processor becomes idle. When a processor becomes busy, the task is killed. [2] As far as we know, no work has been done on parallel adaptive TS.

3. A parallel adaptive tabu search algorithm

In this paper, a straightforward approach has been used to introduce adaptive parallelism in TS. It consists in parallel independent TS algorithms. This requires no communication between the sequential tasks. The algorithms are initialized with different solutions. Different parameter settings are also used (size of the tabu list).

3.1. Parallel algorithm design

The programming style used is the master/workers paradigm. The master task generates work to be processed by the workers. Each worker task receives a work from the master, computes a result and sends it back to the master. The master/workers paradigm works well in adaptive dynamic environments because:

- when a new node becomes available, a worker task can be started there,
- when a node becomes busy, the master task gets back the pending work which was being computed on this node, to be computed on the next available node.

The master implements a central memory through which passes all communication, and that captures the global knowledge acquired during the search. The number of workers created initially by the master is equal to the number of idle nodes in the parallel platform. Each worker implements a sequential TS task. The initial solution is generated randomly and the tabu list is empty.

The parallel adaptive TS algorithm reacts to two events (Fig. 3):

Transition of the load state of a node from idle to busy: If a node hosting a worker becomes loaded, the master *folds up* the application by withdrawing the worker. The concerned worker puts back all pending work to the master and dies. The pending work is composed of the current solution, the best local solution found, the short-term memory, the long-term memory and the number of iterations done without improving the best solution. The master updates the best global solution if it's worst than the best local solution received.

Transition of the load state of a node from busy to idle: When a node becomes available, the master *unfolds* the application by staring a new worker on it. Before

[2] Note that before being killed, a task may return its pending work (best known solution, short and long-term memory).

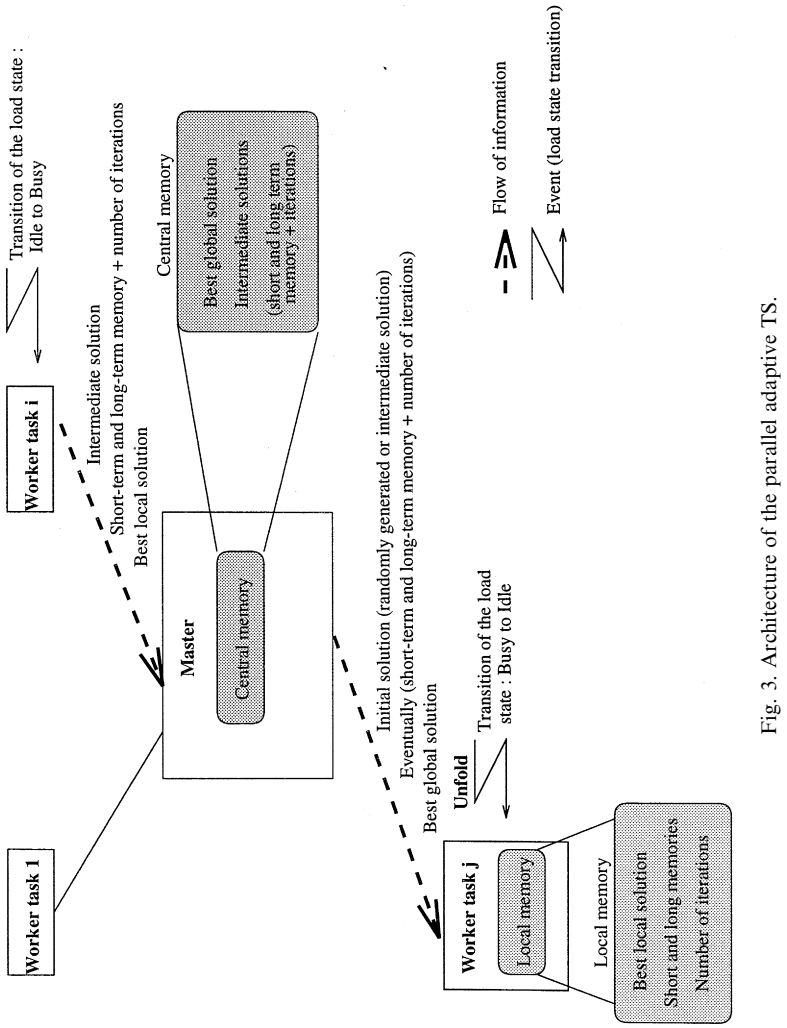

Fig. 3. Architecture of the parallel adaptive TS.

starting a sequential TS, the worker task gets the values of the different parameters from the master: the best global solution and an initial solution which may be an intermediate solution found by a folded TS task, which constitute a ''good'' initial solution. In this case, the worker receives also the state of the short-term memory, the long-term memory and the number of iterations done without improving the best solution.

The local memory of each TS task which defines the pending work is composed of (Fig. 3): the best solution found by the task, the number of iterations applied, the intermediate solution and the adaptive memory of the search (short-term and long-term memories). The central memory in the master is then composed of (Fig. 3): the best global solution found by all TS tasks, the different intermediate solutions with the associated number of iterations and adaptive memory.

3.2. Parallel algorithm implementation

The parallel run-time system to be used has to support dynamic adaptive scheduling of tasks, where the programmer is totally preserved from the complex task of managing the availability of nodes and the dynamics of the target machine. Piranha (under Linda) [19], CARMI/Wodi (under PVM/Condor) [20], and MARS [21] are representative of such scheduling systems. We have used the MARS dynamic scheduling system.

The MARS system is implemented on top of the UNIX operating system. We use an existing communication library which preserves the ordering of messages: PVM. [3] Data representations using XDR are hidden for the programmer. The execution model is based on a preemptive multi-threaded run-time system: PM^2. [4] The basic functionality of PM^2 is the Lightweight Remote Procedure Call (LRPC), which consists in forking a remote thread to execute a specified service.

It is very important for the MARS scheduling system to quantify node idleness or node availability. This is highly related to both load indicators chosen to define it and owner behavior. Several load indicators are provided: CPU utilization, load average, number of users logged in, user memory, swap space, paging rate, disk transfer rate, /tmp space, NFS performance, etc. Owner activity is detected by controlling its keyboard and mouse idle times. For our experiments based on many parallel applications, a node is considered idle if the one, five and ten minutes load average are below 2.0, 1.5 and 1.0 respectively and the keyboard/mouse are inactive for more than five minutes. Two conflicting goals emerge when setting the thresholds: minimize the overhead of the evaluation and the fluctuation of the load state, and exploit a node as soon as it becomes idle.

A MARS programmer writes a parallel application by specifying two multi-threaded modules: the *master module* and the *worker module*. The master module is composed mainly of the *work server thread*. The worker module acts essentially as

[3] Parallel Virtual Machine.
[4] Parallel Multi- threaded Machine.

a template for the *worker threads*. When the parallel application is submitted, the master module is executed on the home node. The number of "worker threads" is function of the available idle nodes. The MARS run-time scheduling system handles transparently the adaptive execution of the application on behalf of the user.

In the application, we have to define two coordination services: *get_work* and *put_back_work*. The first coordination service specifies the function to execute when an unfolding operation occurs and the second one for the folding operation.

When a processor becomes idle, the MARS node manger communicates the state transition to the MARS scheduler, which in turn communicates the information to the application through the master using the RPC mechanism. Then, the master creates a worker task. Once the worker is created, it makes a LRPC to the *get_work* service to get the work to be done. Then, the worker creates a thread which execute a sequential TS algorithm (Fig. 4).

When a processor becomes busy or owned, the same process is initiated in the MARS system. In this case, the worker makes a LRPC to the *put_back_work* service to return the pending work and dies (Fig. 5).

4. Application to the QAP

The parallel adaptive TS algorithm has been used to solve the quadratic assignment problem (QAP). The QAP represents an important class of combinatorial optimization problems with many applications in different domains (facility location, data analysis, task scheduling, image synthesis, etc.).

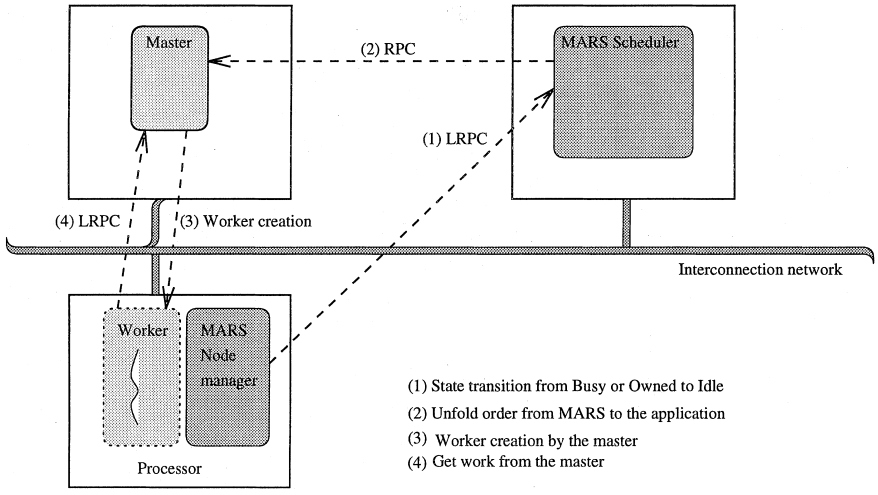

Fig. 4. Operations carried out when a processor becomes idle.

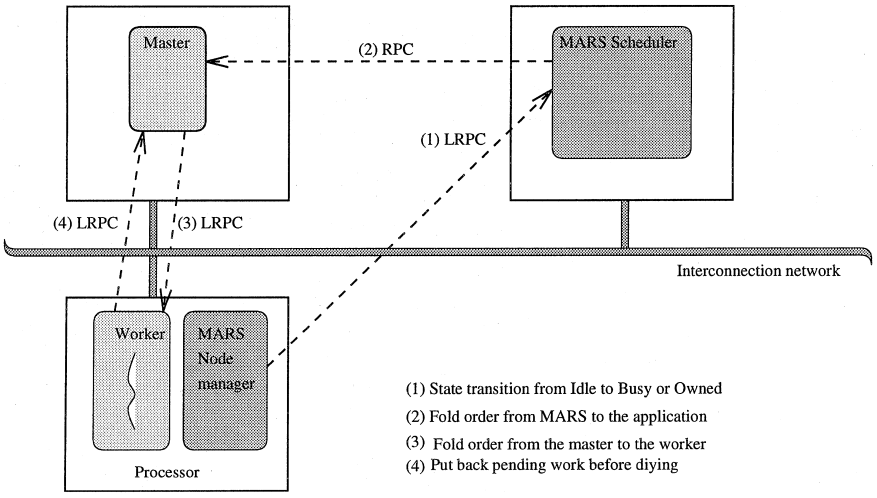

Fig. 5. Operations carried out when a processor becomes busy or owned.

4.1. The quadratic assignment problem

The first formulation was given by Koopmans and Beckmann in 1957 [22]. The QAP can be defined as follows:

Given:

- a set of n objects $O = \{O_1, O_2, \ldots, O_n\}$,
- a set of n locations $L = \{L_1, L_2, \ldots, L_n\}$,
- a flow matrix C, where each element c_{ij} denotes a flow cost between the objects O_i and O_j,
- a distance matrix D, where each element d_{kl} denotes a distance between location L_k and L_l,

find an object-location bijective mapping $M : O \rightarrow L$, which minimizes the objective function f,

$$f = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}\ d_{M(i)M(j)}.$$

The QAP is NP-hard [23]. Finding an ϵ-approximate solution is also NP-complete [24]. This fact has restricted exact algorithms (such as branch and bound) to small instances ($n < 22$) [25]. An extensive survey and recent developments can be found in [26].

4.2. Tabu search for the QAP

To apply the parallel adaptive TS to the QAP, we must define the neighborhood structure of the problem and its evaluation, the short-term memory to avoid cycling and the long-term memory for the intensification/diversification phase.

4.2.1. Neighborhood structure and evaluation

Many encoding schemes may be used to represent a solution of the QAP. We have used a representation which is based on a permutation of n integers:

$$s = (l_1, l_2, \ldots, l_n),$$

where l_i denotes the location of the object O_i. We use a pair exchange move in which two objects of a permutation are swapped. The number of neighbors obtained by this move is $(n(n-1))/2$.

We use the formulae reported in [27] to efficiently compute the variation in the objective function due to a swap of two objects. The evaluation of the neighborhood can be done in $O(n^2)$ operations.

4.2.2. Short-term memory

The tabu list contains pairs (i,j) of objects that cannot be exchanged (recency-based restriction). The efficiency of the algorithm depends on the choice of the size of the tabu list. Our experiments indicate that choosing a size which varies between $n/2$ and $(3n)/2$ gives very good results. The number of parallel TS tasks is set to the problem size n, and each TS task is initialized with a different tabu list size from $n/2$ to $(3n)/2$ with an increment of 1.

The aspiration function allows a tabu move if it generates a solution better than the best found solution. The total number of iterations depends on the problem size, and is limited to $1000n$.

4.2.3. Long-term memory

We use as a long-term memory a frequency-based memory which complements the information provided by recency-based memory. A matrix $F = (f_{i,k})$ represents the long-term memory. Let S denote the sequence of all solutions generated. The value $f_{i,k}$ represents the number of solutions in S for which $s(i) = k$. This quantity identifies the number of times the object i is mapped on the location k. The different values are normalized by dividing them by the average value which is equal to $(1 + nb\_iterations)/n$, given that the sum of the n^2 elements of the matrix F is equal to $n(1 + nb\_iterations)$.

If no better solution is found in $100n$ iterations, the *intensification* phase is started. The intensification phase starts from the best solution found in the current region and an empty tabu list. The use of the frequency-based memory will penalize non-improving moves by assigning a larger penalty to swaps with greater frequency values.

A simulated-annealing like process is used in the intensification phase. The relevance of our approach compared to pure simulated-annealing is that it exploits memory of TS for selecting moves. For each move $m = (i, j)$, an incentive value P_m is introduced in the acceptance probability to encourage the incorporation of good attributes (Fig. 6). The value of P_m is initialized to $\mathrm{Max}(f_{i,s(i)},\ f_{j,s(j)})$. Therefore, the probability that a move will be accepted diminishes with small values of P_m.

The *diversification* phase is started after $100n$ iterations are performed without any improvements of the restarted TS algorithm. Diversification is performed in $10n$

Step 1 : Initialization

- Choose the best found solution S_0 ($S = S_0$)
- $nbiter = 0$ /* current iteration */
- $T := T_{max}$ /* Initial temperature $T_{max} = 10$ */

Step 2 : Iteration

- **Repeat**
  - nbiter := nbiter + 1
  - **For** j:=1 **To** NB_MAX **Do** /* $NB\_MAX = n$ */
    - Generate a random move m which transforms S to S'
    - $\triangle S = f(S') - f(S)$
    - **If** $(\triangle S < 0)$ or $(random(0,1) < exp(-\frac{\triangle S}{T * P_m}))$ **Then** $S := S'$
  - **End For**
  - $T := a.T$ /* annealing schedule with $a = 0.9$ */
- **Until** $T < T_{min}$ /* $T_{min} = 0.5$ */

Fig. 6. Simulated annealing for the intensification phase.

iterations. The search will be forced to explore new regions by penalizing the solutions often visited. The penalty value associated to a move $m = (i, j)$ is $I_m = \text{Min}(f_{i,s(i)}, f_{j,s(j)})$. A move is tabu-active if the condition $I_m > 1$ is true. Therefore, we will penalize moves by assigning a penalty to moves with greater frequency. The number of iterations is large enough to drive the search out of the current region. When the diversification phase terminates, the tabu status based on long-term memory is dropped.

5. Computational results

For our experimental study, we collected results from a platform combining a network of heterogeneous workstations and a massively parallel homogeneous machine (Fig. 7). The network is composed of 126 workstations (PC/Linux, Sparc/Sunos, Alpha/OSF, Sparc/Solaris) owned by researchers and students of our University. The parallel machine is an Alpha-farm composed of 16 processors connected by a crossbar switched interconnection network. The parallel adaptive TS competes with other applications (sequential and parallel) and owners of the workstations.

5.1. Adaptability of the parallel algorithm

The performance measures we use when evaluating the adaptability of the parallel TS algorithm are execution time, overhead, the number of nodes allocated to the

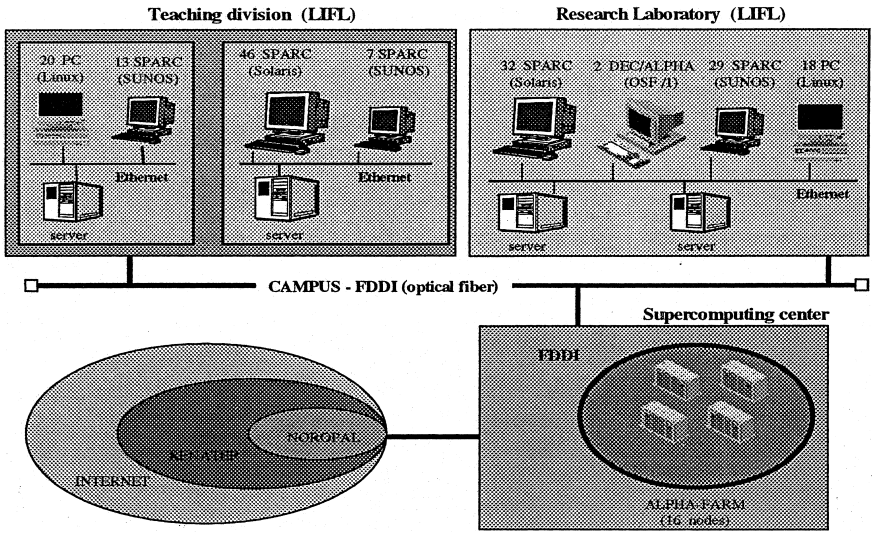

Fig. 7. The meta-system used for our experiments.

application, and the number of fold and unfold operations. The overhead is the total amount of CPU time required for scheduling operations. Table 2 summarizes the results obtained for 10 runs.

The average number of nodes allocated to the application does not vary significantly and represents 71% of the total number of processors. However, the high number of fold and unfold operations shows a significant load fluctuation of the different processors. During an average execution time of 2 h 25 mn, 79 fold operations and 179 unfold operations are performed. This corresponds to one new node every 0.8 mn and one node loss every 2 mn. These results demonstrate the significance of the adaptability concept in parallel applications.

The parallel algorithm is efficient in terms of the scheduling overhead due to the adaptability. The overhead is low comparing to the total execution time (0.09% of the total execution time). We see also that the deviation of the overhead is very low (0.24% for 10 runs). The classical speedup measure cannot be applied to our application which executes on a heterogeneous multi-user non-dedicated parallel

Table 2
Experiment results obtained for 10 runs of a large problem (Sko100a) on 100 processors (16 processors of the Alpha-farm, 54 Sparc/Solaris, 25 Sparc/SunOs, 5 PC/Linux)

	Mean	Deviation	Min	Max
Execution time (mn)	145.75	23.75	124	182
Overhead (sec)	8.36	0.24	8.18	8.75
Number of nodes allocated	71	15.73	50	92
Number of fold operations	79	49.75	24	149
Number of unfold operations	179	45.55	120	248

platform. Unfortunately, quantitative analysis of heterogeneous dynamic parallel systems still in its infancy [28].

5.2. QAP results

To evaluate the performance of the parallel TS in terms of solution quality and search time, we have used standard QAP problems of different types (QAP library):

- random and uniform distances and flows: Tai35a, Tai100a,
- random flows on grids: Nug30, Sko100a-f, Tho150, Wil100,
- real-life or real-life like problems: Bur26d, Ste36b, Esc128, Tai150b, Tai256c.

The parallel TS algorithm was run 10 times to obtain an average performance estimate. Table 3 shows the best known, best found, worst, average value and the standard deviation of the solutions obtained for the chosen small instances ($n < 50$). The search cost was estimated by the wall-clock time to find the best solution, and hence account for all overheads. Considering that the best solution may not be the last visited solution, the measured time is not necessarily the time of the complete execution of the algorithm.

We always succeed in finding the best known solutions for small problems. This result shows the efficiency and the robustness of the parallel TS.

Table 4 shows the best results obtained for large problems. For random-grid problems, we found the best known solutions (for *Sko 100c*) or solutions very close to best known solutions. Our results in terms of search time are smaller than those presented in [29] with better solution quality. The most difficult instance for our algorithm is the random-uniform *Tai100a*, in which we obtain a gap of 0.32% above the best known solution. For this class of instances, it is difficult to find the best known solution but simple to find "good" solutions.

For the third class of instances (real-life or real-life like), the best known solutions for *Tai150b* and *Tai256c* (generation of grey patterns) have been improved.

According to the results, we observe that the algorithm is well fitted to a large number of instances but its performance decreases for large uniform–random instances (Tai100a). Work is still to be done to improve the efficiency of the algorithm by introducing intensification mechanisms based on *path relinking*, where S represents a small subset of elite solutions [2].

Table 3
Results for small problems ($n < 50$) of different types

Instance	Best known	Best found	Worst	Average	Standard deviation	Average search time (s)
Tai35a	2 422 002	2 422 002	2 422 002	2 422 002	0	566
Nug30	6124	6124	6124	6124	0	337
Bur26d	3 821 225	3 821 225	3 821 225	3 821 225	0	623
Ste36b	15 852	15 852	15 852	15 852	0	763

Table 4
Results for large problems ($n \geqslant 50$) of different types

Instance	Best known	Best found	Gap	Search time (mn)
Tai100a	21 125 314	21 193 246	0.32%	117
Sko100a	152 002	152 036	0.022%	142
Sko100b	153 890	153 914	0.015%	155
Sko100c	147 862	147 862	0%	132
Sko100d	149 576	149 610	0.022%	152
Sko100e	149 150	149 170	0.013%	124
Sko100f	149 036	149 046	0.006%	125
Wil100	273 038	273 074	0.013%	389
Tho150	8 134 030	8 140 368	0.078%	287
Esc128	64	64	0%	230
Tai150b	499 348 972	499 342 577	−0.0013%	415
Tai256c	44 894 480	44 810 866	−0.19%	593

6. Conclusion and future work

The dynamic nature associated to the load of a parallel machine (cluster of processors and network of workstations) makes essential the adaptive scheduling of tasks composing a parallel application. The main feature of our parallel TS is to adjust, in an adaptive manner, the number of tasks with respect to available nodes, to fully exploit the availability of machines. The parallel algorithm can benefit greatly from a platform having combined computing resources of MPPs and NOWs.

An experimental study has been carried out in solving the QAP. The parallel TS algorithm includes different tabu list sizes and intensification/diversification mechanisms (frequency based long-term memory, etc.). The performance results obtained for several standard instances from the QAP-library are very encouraging in terms of,

Adaptability: The overhead introduced by scheduling operations is very low, and the algorithm reacts very quickly to changes of the machines load. It's a worthwhile parallelization because the parallel TS application is coarse-grained.

Efficiency and robustness: The parallel algorithm has always succeeded in finding the best known solutions for small problems ($n < 50$). The best known solutions of large real-life problems "charts of grey densities" (*Taixxxc*) and the real-life problem *Tai150b* have been improved. The parallel algorithm often produces best known or close to best known solutions for large random problems. Other sophisticated intensification and diversification mechanisms to improve the results obtained for large random–uniform problems (Taixxa) are under investigation.

The parallel adaptive TS may be used to solve other optimization problems: set covering, independent set, multiconstraint knapsack. We plan to improve our

framework to provide adaptive parallelism to other metaheuristics (genetic algorithms and hybrid algorithms) and for exact algorithms (IDA* and branch and bound).

Solving very large optimization problems can take several hours. The aspects of fault tolerance must be taken into account. A checkpointing mechanism which periodically save the context of the application is under evaluation [30].

References

[1] F. Glover, Tabu search – Part I, ORSA Journal of Computing 1 (3) (1989) 190–206.
[2] F. Glover, M. Laguna, Tabu search, in: C.R. Reeves (Ed.), Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publications, Oxford, 1992, pp. 70–150.
[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W. Su, Myrinet – A gigabit-per-second local-area network, IEEE Micro (1995) 29–36.
[4] P.R. Woodward, Perspectives on supercomputing: Three decades of change, IEEE Computer (1996) 99–111.
[5] D.A. Nichols, Using idle workstations in a shared computing environment, ACM Operating System Review 21 (5) (1987) 5–12.
[6] M.M. Theimer, K.A. Lantz, Finding idle machines in a workstation-based distributed system, IEEE Transactions on Software Engineering 15 (11) (1989) 1444–1458.
[7] F. Glover, E. Taillard, D. de Werra, A user's guide to tabu search, Annals of Operations Research 41 (1993) 3–28.
[8] A. Hertz, D. de Werra, The tabu search metaheuristic: How we use it? Annals of Mathematics and Artificial Intelligence (1989) 111–121.
[9] S. Voss, Tabu search: Applications and prospects, Technical report Technische Hochshule Darmstadt, Germany, 1992.
[10] T.D. Crainic, M. Toulouse, M. Gendreau, Towards a taxonomy of parallel tabu search algorithms, Technical Report CRT-933, Centre de Recherche sur les Transports, Université de Montreal, 1993.
[11] E. Taillard, Parallel iterative search methods for vehicle routing problem, Networks 23 (1993) 661–673.
[12] E. Taillard, Robust taboo search for the quadratic assignment problem, Parallel Computing 17 (1991) 443–455.
[13] J. Chakrapani, J. Skorin-Kapov, Massively parallel tabu search for the quadratic assignment problem, Annals of Operations Research 41 (1993) 327–341.
[14] M. Malek, M. Guruswamy, M. Pandya, H. Owens, Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem, Annals of Operations Research 21 (1989) 59–84.
[15] C. Rego, C. Roucairol, A parallel tabu search algorithm using ejection chains for the vehicle routing problem, in: Proc. of the Metaheuristics Int. Conf., Breckenridge, 1995, pp. 253–295.
[16] P. Badeau, M. Gendreau, F. Guertin, J.-Y. Potvin, E. Taillard, A parallel tabu search heuristic for the vehicle routing problem with time windows, RR CRT-95-84, Centre de Recherche sur les Transports, Université de Montréal, 1995.
[17] S.C.S. Porto, C. Ribeiro, Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints, Journal of heuristics 1 (2) (1996) 207–223.
[18] T.L. Casavant, J.G. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, IEEE Transactions on Software Engineering 14 (2) (1988) 141–154.
[19] D.L. Kaminsky, Adaptive parallelism in Piranha, Ph.D. thesis, Department of Computer Science, Yale University, RR-1021, 1994.
[20] J. Pruyne, M. Livny, Parallel processing on dynamic resources with CARMI, in: Proc. of the Workshop on Job Scheduling for Parallel Processing IPPS'95, Lecture Notes On Computer Science, No.949, Springer, Berlin, 1995, pp. 259–278.

[21] Z. Hafidi, E.G. Talbi, J.-M. Geib, MARS: Adaptive scheduling of parallel applications in a multi-user heterogeneous environment, in: European School of Computer Science ESPPE'96: Parallel Programming Environments for High Performance Computing, Alpe d'Huez, France, 1996, pp. 119–122.
[22] T.C. Koopmans, M.J. Beckmann, Assignment problems and the location of economic activities, Econometrica 25 (1957) 53–76.
[23] M. Garey, D. Johnson, Computers and Intractability: A guide to the theory on NP-completeness, Freeman, New York, 1979.
[24] S. Sahni, T. Gonzales, P-complete approximation problems, Journal of the ACM 23 (1976) 556–565.
[25] A. Brungger, A. Marzetta, J. Clausen, M. Perregaard, Joining forces in solving large-scale quadratic assignment problems in parallel, in: A. Gottlieb (Ed.), 11th Int. Parallel Processing Symposium, Geneva, Switzerland, Morgan Kaufmann, Los Altos, CA, 1997, pp. 418–427.
[26] P.M. Pardalos, F. Rendl, H. Wolkowicz, The quadratic assignment problem: A survey and recent developments, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 16 (1994) 1–42.
[27] E. Taillard, Comparison of iterative searches for the quadratic assignment problem, Location Science 3 (1995) 87–103.
[28] M.M. Eshagian, Heterogeneous computing, Artech House, MA, 1996.
[29] C. Fleurent, J.A. Ferland, Genetic hybrids for the quadratic assignment problem, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 16 (1994) 173–188.
[30] D. Kebbal, E.G. Talbi, J.-M. Geib, A new approach for check pointing parallel applications, in: Int. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA'97, LasVegas, USA, 1997, pp. 1643–1651.