Scheduling with Taboo Search on Parallel Architectures

Malek RAHOUAL[1] Rachid SAAD[2]

[1] LaMI, Université d'Evry Val d'Essonne, 91000 Evry - France.
Email: mrahoual@lami.univ-evry.fr

[2] Laboratoire d'Informatique Fondamentale et Appliquée
Université de Boumerdès, 35000 Algérie
Email: rachid_saad2003@yahoo.com

Abstract. Scheduling static tasks on parallel architectures is a basic problem arising in the design of parallel algorithms. This NP-complete problem has been widely investigated in the literature and remains one of the most challenging questions in the field. Among the resolution methods for this type of problems, the taboo search technique is of particular interest. Based on this technique, two algorithms are proposed and tested on a sample of instances in order to be compared experimentally with other well known algorithms. The results clearly indicate good overall performances of our algorithms. Next, some NP-completeness results are established showing that this problem is intractable for approximation, even for some restricted cases bearing a clear relation to the instances treated experimentally in this work.

Key-Words. Scheduling, Taboo search, NP-complete, parallelization.

1. Introduction

Responding to an ever-increasing demand of computing power, parallelism [8][23][27] appears over the years, as a realistic and promising solution in the design of computer systems. However, such a computing power is bought at a price.

For one thing, scheduling concurrent processes on multiprocessor architectures (which is our main concern in this paper) is difficult [25], that is NP-hard, except for a few particular cases. In fact, most of the assisting tools for programming parallel architectures [6][16][26] compel the user to define by himself his own mapping [26] by means of specific primitives, which is not an easy task. Should the program be executed on another configuration of the interconnection network, it would have to be rewritten, which questions the portability of the programs and highligts the importance of achieving an automatic schedule by some heuristic means.

There are two different approaches to scheduling, depending on the time when the mapping decision is taken. While a static schedule determines mapping at compile time, a dynamic schedule performs its mapping at execution. In both cases,

scheduling consists of assigning a processor and a start time to each task. It also entails the verification of the precedence constraints. The assignment should maximize the use of parallelism by loading the different processors as fairly and equally as possible. Moreover, it should minimize the communication costs [7][18][20] in the system.

Static assignment [3][18][26][27], dealt with in our present work, can be thought of as a starting point for a dynamic assignment algorithm. It is used for real time and graphic applications as well. Static assignments have many applications in scientific computation and linear algebra, such as computing the product of two matrices or solving linear systems of equations [22].

An instance of the static scheduling problem is describely by the following inputs, whose full details will be given subsequently:

- the set of tasks or processes
- the processing time of each of the tasks
- the precedence constraints to which these tasks are subjected
- the transfer times of outputs and how communication is taken into account
- the topology of the architecture on which the application will run.

Our main tool for tackling this scheduling problem is the well known Taboo Search (TS) method. TS is one of the many resolution methods for this type of problem. It is a general iterative method in combinatorial optimization, which was introduced by Glover [11][12] and extended later to a broader context. Taboo search is a meta-heuristic which is well known for its ability to escape from local optima. Due to the success of these methods as regards optimization problems, TS was the subject of many works in the last decade. It has also been successfully used in combination with other heuristics and methods for the scheduling problems [14], resources allocation problems [1] and telecommunication problems [12] as well.
Our choice of this particular method from among the many other methods is motivated by the following features of TS [12]:

- Utilization of a short term and a long term memory (the taboo list)
- Utilization of symbolic data. Just as is the case with genetic algorithms, TS is not constrained by the type (integer or not) of the variables used.
- Minimal a priori information is necessary: TS requires only that some means is available to evaluate the quality of a solution
- Introduction of diversification and intensification mechanisms [9]

Most of the algorithms proposed for the scheduling problem in the literature are sequential [2][15][17][20][24][26].
The aim of this work is to extend previous results by applying the taboo method to the static scheduling problem. Our assumptions throughout are:

- The tasks are non-preemptive (no duplication is allowed)
- The architecture is that of a multiprocessor, shared memory structure [24].

The objective is to minimize the total makespan. To this end, two taboo algorithms are designed, a sequential and a parallel one. The latter explores the space of solutions in parallel. These search processes automatically and dynamically compute the search

parameters (referred to as strategies): the size of the taboo list and the maximum number of iterations between two improvements of the solution.

The proposed parallel algorithm is independent of the architecture on which it is executed. Moreover, no assumption is made on the structure of the application. The implementation and the evaluation of the performance of our algorithm have shown an almost linear speed-up together with good overall performances (in terms of the quality of the solution and the execution time).

The second section of this paper is devoted to the presentation and formulation of the problem under study. The third section presents two sequential taboo search algorithms. As for the fourth section, it presents the parallel algorithm. The fifth section presents the different experimental tests undertaken along with their analysis. The sixth section provides a comparative study between the different algorithms proposed. Last, but not least, we conclude with a study of the complexity of our problem. We prove that the problem under investigation is intractable for approximation even for the particular case when the "topology of the architecture" is that of a complete graph with as many vertices as tasks (unlimited supply of processors with all the possible links available). This latter result justifies in our view the use of heuristic methods to deal with our problem.

2. Presentation and formulation of the scheduling problem

A computer processing is a sequence of elementary transformations of an initial set of inputs. These transformations may be grouped to form a set of partial processings referred to as tasks. Tasks are treated as entities by the system.
If one of those tasks outputs results to another one, it must precede it, which induces a partial order relation on the set of tasks.

2.1 Notations and definitions

G_T	: tasks graph; V_T: set of vertices of G_T with $\|V_T\|=n$; E_T: set of arcs of G_T
T_i	: denotes a task
e_i	: duration of T_i (the time needed to execute T_i once it is started)
r_i	: starting time of T_i
Y_{ij}	: number of units exchanged or transferred between T_i and T_j
G_p	: graph representing the architecture; V_p: set of vertices (processors) of G_p with $\|V_p\|=m$
E_p	: set of edges of G_p
P_i	: denotes a processor
C_{kl}	: transfer time of one unit of information from P_k to P_l (distance from P_k to P_l in weighted graph G_p)
C_{max}	: total execution time of the application

ftt(i)	: completion time of T_i.
ftp(i)	: completion time for processor P_i.
Proc_source	: source processor of the task being moved. It is the processor on which the task was mapped before the move
Num_task	: task concerned by the move
Proc_dest	: target processor (or destination) of the move
AS(P_i)	: set of tasks currently mapped on processor P_i
Ind	: position (or topological rank) of *num_task* in the new solution. This position refers to a rank of *num_task* in *AS(proc_dest)* after the move. *ind* can be computed by any topological sorting algorithm
size_list_taboo	: size of the taboo list
nbmax	: maximum number of iterations between two improvements of the best solution s^*
normal_size_taboo	: normal or small size of the taboo list
nb_local1	: maximum number of iterations of the intensification phase
nb_local2	: number of iterations of the diversification phase
nbr_search_iter	: number of iterations of the process of local intensification and global diversification
large_size_taboo	: large tenure with which the diversification process is initialized
tenure_step	: incrementing step of tenure during diversification
S	: a solution
*S**	: best solution encountered between two improvements
T	: taboo list

2.2 The task model

The basic concept for representing a parallel algorithm is the precedence graph of the tasks, where an elementary task is a set of instructions defined by its inputs, outputs and its execution time. Transfer of outputs induces a precedence relation on the set of tasks. A task receives its inputs from its immediate predecessors, performs computations on them and outputs the results to its immediate successors.
Transfer of output from task T_i to task T_j takes some (non-negligible) time if the two tasks are assigned to different processors. This communication cost depends on the number of words transferred as well as on the distance between the processors assigned to T_i and T_j [2][3][5][7][18].

An application is described by an oriented graph $G_T=(V_T, E_T)$. This is a connected circuit free graph, where:
- The set of vertices V_T corresponds to the set of tasks (including a start task and an end task). An arc from T_i to T_j means that T_j cannot start before T_i is completed.
- The value associated to T_i refers to the execution time e_i of T_i. The value associated to an arc (T_i, T_j) indicates the number of units Y_{ij} transferred from T_i to T_j.

2.3 The architecture model

MIMD, which stands for Multiple Instruction Multiple Data, is a type of multi-processor computer architecture, where each processor has its own built-in Data and Program memory. In this model, each processor can execute different processes, and communicates with the other processors by sending messages, the inter-processor communication being asynchronous. The processors are connected by fixed links provided by a network of a topology that could be either fixed (grid, tree, hypercube...) or reconfigurable (Crossbar's, Clos', Benes' network) [30].
The most powerful parallel machines presently on the computer market are essentially of the distributed memory MIMD type, such as Cray T3D, Intel Paragon, IBM SP2 or NEC Cenju.

The interesting point in this model lies in the fact that the networks of work-stations (which are growing so pervasive nowadays) operate basically on the MIMD mode, where communication occurs through the forwarding of messages (typically controlled by PVM: Parallel Virtual Machine) on a local or interconnection network. In fact, a network of work stations often consists of a hybrid architecture where a combination of any amount of work-stations and multiprocessors (managing their own their internal parallelism by a shared memory or by a bus or network device) are connected in a local network.

The class of architectures of interest to us is one in which many identical processors work in parallel, each of which is made of a control unit, a processing unit and a local random access memory. These processors communicate by exchanging messages through an interconnection network. This class of architectures is referred to as a distributed memory MIMD [21][29][30]. A classical example is a network of workstations communicating through an interconnection network. The latter characterizes the communication facilities provided for the processors and has therefore a crucial impact on the performances of the architecture.

The configuration of the system is described by a non oriented connected graph $G_p=(V_p,E_p)$. The set of vertices V_p corresponds to a set of processors. There is an edge between two vertices if the two processors are physically connected. An edge $(k,l)\in E_p$ is weighted by C_H. The representation of these values by a $|V_p| \times |V_p|$ matrix is straightforward.

2.4 Formulation of the scheduling problem

To formulate our problem, we need one more definition. Let (T_i, T_j) be an arc of G_T. Let us say that a processor P_k knows (T_i, T_j) at time t if either P_k has already executed T_i or the output necessary to execute T_j from T_i has already been transferred to P_k at time t. Now, we can state our static assignment problem as follows:

Given a set of tasks and a network G_P of processors, to schedule all the given tasks in a minimum makespan, subject to the following constraints:

(1) The tasks are non pre-emptive
(2) A processor can be in either one of the following mutually exclusive states: (a) Processing a task, (b) Idling, (c) Transfering outputs to another processor
(3) A processor P can execute a task T_j only if P knows all the arcs internally incident to T_j (that is, all arcs (T_i, T_j))
(4) A processor can not communicate with more than one processor simultaneously
(5) If T_i is assigned to P_k and T_j is assigned to P_m, the communication cost of transfering T_i to T_j is: $Y_{ij} * C_{km}$, where C_{km} is the distance from P_k to P_m in the weighted graph G_P.

The following example of a feasible schedule illustrates how communication is taken into account in our type of schedule.

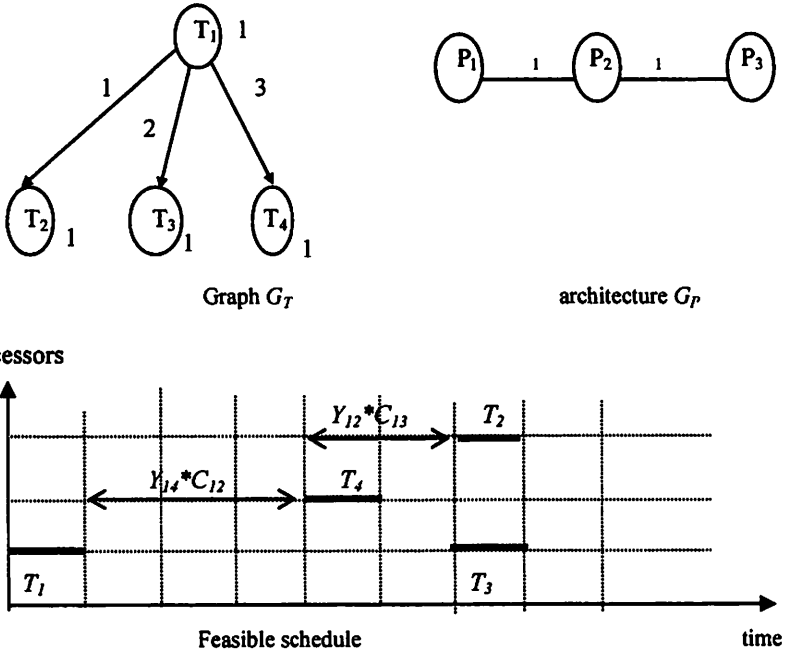

Fig. 1. Example of a feasible schedule for a particular instance

3. Application of the taboo method

This technique guides an iterative optimization process which, starting from an initial solution, searches the best solution encountered in a complex solutions space using local information to progress [11][14].
In our case, this local information consists of a triple (T_i, P_r, r_i) of the current solution such that T_i is the task whose execution on processor P_r starts at time r_i, for all task T_i.

Given a function f to optimize, TS iteratively moves from one solution to another until some specified condition is satisfied. Thus, every iteration of TS gives rise to an elementary move denoted by move. In our case, a move may be viewed as an elementary transformation operating on local information.
For every solution S_i, we define the neighborhood $V(S_i)$ of S_i to be the set of all the solutions that can be reached from S_i in a single move (that is, in one iteration of TS). TS adopts a strategy of modifying $V(S_i)$ as search proceeds, thus giving rise to a new set $V^*(S_i)$. To carry out this modification, TS uses special structures of memory (see figure 2).

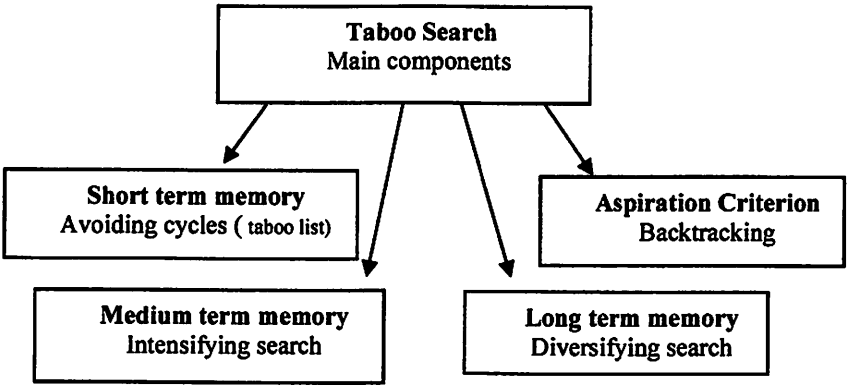

Fig. 2. The main components of TS

Other ingredients are built in our algorithm to allow for search to either intensensify or slacken in more or less promising regions. A diversification ingredient is also provided to cover the most part of the search space. Both ingredients are addressed in the forthcoming paragraphs.

3.1 Determination of the parameters of TS

3.1.1 Determination of an initial solution: a list_scheduling Algorithm

To start taboo search, we need to generate an initial solution. For this purpose, we use a straightforward parallelization of the well known critical path method. In a first step, we rank the tasks according to some order of priority which is specified in our algorithm as the increasing order of their late start times, as if they were executed by a single machine. Next, at any one given time, we use a sort of parallelization of the well known list scheduling algorithm (which is a greedy algorithm) to schedule, at any one given time, the tasks that are ready to be executed in parallel.

```
List_scheduling Algorithm
begin
Step1: - Ens1=∅ /*the set of tasks that are ready to be
         executed in parallel */
       - Ens2= {tasks of the application} /*set of tasks
         not assigned yet*/
       - Initialisation and data entry;
       - Determine the minimum (sequential) makespan of
         the set of tasks
Step2: Determine the (sequential) late start time of each
       task and order the tasks in the increasing order L
       of their start times
Step3: while  Ens2<>∅              Do
       - Determine the tasks that are ready to be executed
         in parallel;
       - Add these tasks to Ens1 ;
       - While (Ens1<>∅ and ∃ available processors) DO
            * Pick the task T_i in Ens1 with the highest
              priority in the priority list L and  Assign
              T_i to a processor,
            *  Update the state of the processors (a
              processor p completes its current task T_j
              and becomes free at time:  legth of T_j +
              start time of T_j)
            *  Ens2 : =Ens2-{ T_i };
            *  Ens1 : =Ens1-{ T_i };
         Done
    Done
End
```

3.1.2 Determination of the neighborhood

The neighborhood *V(s)* of a solution s is defined via the transformation move encoded as *move=(proc_source, num_task, proc_dest,ind).* A move consists of transferring task *num_task* mapped on *proc_source* to *proc_dest* at position *ind,* where *ind* is the topological rank of *num_task* in *AS(proc_dest)* (set of tasks currently mapped on processor *proc_dest*).
The neighborhood *V(s)* describing all the feasible moves from s is too large in general and the only way to determine the solution *s'* minimizing the objective function on *V(s)* is to explore the set *V(s)* entirely, which is too costly in terms of time and space. In order to narrow down the set of solutions examined in a given iteration, we use the so-called strategy of the candidate list: at any point in time, only a fraction of the current *V(s)* is examined and those "candidates" of *V(s)* are carefully chosen. In view of the importance of these strategies, efficient rules should be judiciously selected in order to generate and evaluate the good candidates. A few definitions are necessary before introducing our candidate list. At any point in time, let *p* be a processor and *t* be a task mapped on *p*. Let us say that *t* is a high level task in *p* if *t* has no predecessor among the tasks (currently) mapped on *p*. Now, given current solution *s*, our candidate list is generated by all the feasible moves in the form *move=(proc_source, num_task, proc_dest,ind),*where *num_task* is a high level task for *proc_source*. Moves of this type are, intuitively, among the best moves we can hope for in a neighborhood in a given iteration, and represent therefore the elements of our candidate list.

In order to avoid cycling, a memory of visited solutions is maintained, forming the so-called taboo list. It has limited size and each of its elements represents a series of attributes characterizing the solution.
The active taboo state is assigned to the attributes occurring in recently visited solutions. The taboo solutions are those that either contain or share taboo attributes, which prevents recently visited solutions from occurring in the neighborhood. For the scheduling problem under consideration, every feasible solution is described by a set of attributes whose attribute structure is *(processor, task, position).*
If a move *move*(P_l, T_t, P_h, ind) is applied to a solution s, all moves from T_t to P_l are forbidden in the next t iterations, which is supposed to prevent cycling. Parameter t is then referred to as the taboo tenure.
A move is said to be taboo if it creates a solution having one attribute in common with taboo list *T*. Every time a move *move*(P_l, T_t, P_h, ind) is applied to a solution *s*, the attribute $(P_l, T_t, position_{tl})$ is inserted in *T*, where $position_{tl}$ stands for the rank of T_t with respect to P_l. The aspiration criterion is an important element of the flexibility of taboo search. The taboo state of a solution is not absolute and may be ignored subject to some specified conditions referred to as aspiration criteria.
If a move produces the best solution found so far in terms of its cost, its taboo state is ignored and the solution is accepted. This aspiration criterion is called "objective aspiration".
The so-called aspiration by default is activated when all the moves available in the candidate list are taboo and not eligible for the aspiration by objective. This criterion causes the first move to lose its taboo state. An aspiration function denoted by *(Asp)* is defined practically on all the values of the objective function. When a solution s' is

part of T and satisfies the aspiration (that is: $f(s') < Asp(f(s))$), the taboo state of this solution is lifted and, as a result, s' besoms a candidate when the best neighbor of s is selected. In general, $Asp(f(s))$ assumes the best value s^* currently encountered.

Taboo search is stopped after a certain number *nbmax* of iterations between two consecutive improvements of the best solution s^* encountered.

```
Neighborhood determination algorithm:
begin
for ( all P_l of G_p ) do
   - determine the lowest level min_niv of the tasks
     assigned to P_l ;
   - for ( all T_i mapped on P_l at level min_niv ) do
      for ( all other processors P_l of G_p) do
       for(all subscripts ind (topological rank)of P_k) do
            * Generate the tuple depl =(P_l , T_i , P_k , ind)
            * if (feasible (depl) ) /*tests the feasibility
              of the solution generated by the move depl
             then insert depl in the candidate list endif
         done ;
       done ;  done ;  done ;
end.
```

```
Algorithm Taboo_aspiration

  Step1 : select solution s in S ;   s* :=s ; k := 0;
  Step2 : k:=k+1; Generate a subset S* of solutions in V(s)
  Step3 : select  a best solution j in S* such that
          f(j)<=Asp(f(s))   or  j ∉ T;  s:=j;
  Step4 : if f(s)<f(s*)  then  s*:=s;
  Step5 :  Update  conditions for taboo  and aspiration.
  Step6 :  if the halting condition is satisfied then
  stop else  go to  Step 2.
  END.
```

3.2 Intensification and diversification

The intensification strategy is based on the modification of the selection rules in order to promote the attributes that proved good historically. De Werra and Hertz [9] define the compromise to be held between intensification and diversification as: "an intelligent search, which should not only explore the regions of good solutions entirely but should also have an overall view of the search space and make sure that no region has been ignored". Diversification refers therefore to the set of ways in which new regions are explored. Those mechanisms often consist of the modification of the selection rules in order to introduce attributes rarely used otherwise. Those attributes may be introduced either partially or totally or as a result of a penalization of other solutions with more frequently used attributes. The time when diversification

takes place is also crucial, for instance when the search process is getting close to a local optimum or after a given number of iterations. Taboo search can also utilize the tenure of the taboo attributes as a diversification means: when the search slows down, that tenure is increased.

In our case, we have chosen the strategy that uses the short-term memory. We have also introduced the aspiration criteria (by default and by objective) designed to allow for search to depart from local optima.
The proposed function passes through two steps. The first one corresponds to the intensification process, which consists of applying the iterative process of TS with a usual tenure for a maximum of *nb_local1* improvements of s^*. A test is then carried out to check whether progress in the current region has significantly slowed down (that is, whether we are converging to a local optimum). If the test is positive, the diversification step is triggered. This latter step is a recursive function allowing for search to depart from the current region.
If after *nb_iter* iterations search has not detected a strong aspiration of s^* and all the moves of the list are not taboo (which accounts for the fact that the current large tenures are not "aggressive" enough), then the tenure will be increased and the number of iterations will be decreased. This recursive process will be stopped when a strong aspiration is satisfied or all the moves of the neighborhood are taboo.

```
Algorithm diversif_intensif
  BEGIN
  Step 1: /*Initialization
         - List_scheduling Algorithm(s) /*gene. initial solution
         - s* := s ;
         - f* := f(s);
         - tenure:=normal_size_taboo /*Initial. taboo list
  Step 2: /*iterative process of intensification and
           diversification
           for (i:=1 to nbr_search_iter)  do
           -  /*intensification process
                Taboo_intensif (tenure,nb_local1);
           - /* encounter of a local optimum (slow progression)
           - /* initialization of a large tenure
                large_tenure := large_size_taboo ;
           - /*diversification process
              nb_iter=nb_local2 ;
              Taboo_diversif(large_tenure,nb_iter) ;
           - /* setting taboo list tenure back to normal
                tenure = normal_size_taboo ;
           done
  END
```

4. Parallelization of TS

Classification of the approaches to parallelization of TS can be done in many ways, depending on the criteria taken into account, e.g., the number of initial solutions, the constant or variable feature of the search parameters such as the taboo

list size, tenure of the taboo attributes..., or the strategies of control and communication.
Talbi, Hafidi and Geib [28] classify these approaches into two broad categories: Domain decomposition and multiple task TS (figure 3).

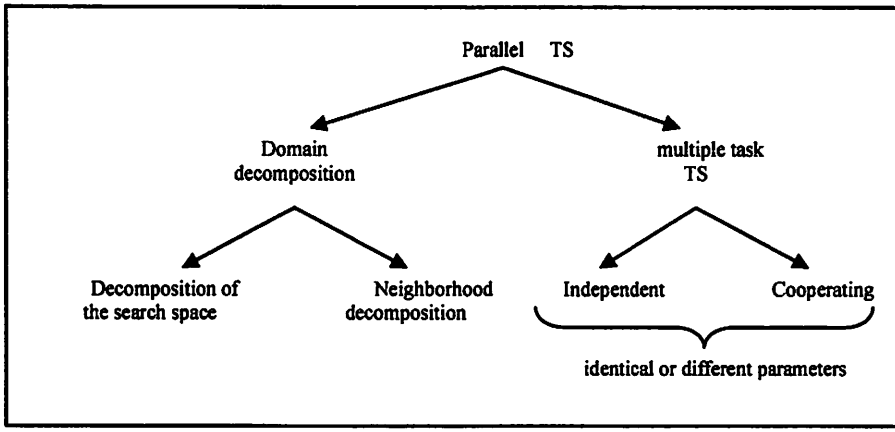

Fig. 3. Classification of the parallel TS algorithms

The implemented parallel algorithm, Parallel taboo, starts a parallel search of the space of the solutions thanks to search processes that automatically and dynamically compute the parameters of search (referred to as strategies): the size of the taboo list *size_taboo_list* and the maximum number *nbmax* of iterations between two consecutive improvements of s^*.
The parallelization in the master/slave mode proposed, consists of creating different parallel independent processes called parallel search paths. A path is an execution of *Taboo_aspiration* with an initial solution s_0 and a strategy given by *(size_list_taboo, nbmax)*. One of the processor has the master status. It controls the other (slave) processors by modifying dynamically the parameters that govern the search. This type of parallelization has the advantage of minimizing the communication costs between the different processes and is well suited to our distributed memory MIMD architecture.

The master process executes an iterative program. At each iteration, P slave processes are started. In the beginning of search, the master provides each slave with an initial solution (generated by the implemented list heuristic) along with a strategy to intensify the search in the different regions reached. When progress in the region explored by, say, process *i* slows down or gets stuck, the master changes strategy st_i to diversify the search and it does so by using *size_list_taboo* with a larger tenure (*large_size_taboo*) and by decreasing $nbmax_i$ to *nb_local*. The increase of tenure and the decrease of the maximum number of iterations will continue until a strong aspiration of the best solution found s_i^* will be satisfied.
At that time, the diversification phase is stopped and the intensification phase takes over in the new region. A normal sized value is then again assigned to tenure

size_taboo_list$_i$ and the value nb_iter$_i$ is assigned to *nbmax*$_i$. The initial solution of iteration *k* is set to the best solution found in iteration *k-1*.

Each slave process executes *Algorithm Taboo_aspiration* with aspiration with the initial solution and the strategy supplied by the parent process. Thus, *Algorithm Taboo_aspiration* is used as a diversification function for some strategies and as a intensification function for others.
Convergence is ensured by the two types of tenure: the small and the normal one. The master process has to run two slave processes, one with strategy *st*$_1$=(*small_tenure_taboo, nb_iter*$_1$) and the other with *st*$_2$=(*normal_tenure_taboo, nb_iter*$_2$).

```
Master process
begin
Step 1: /* generation of the initial solution of the P slaves
            algorithm _liste(s*) ;
            f* = cout (s*) ;
Step 2: /* initialization of the solutions of the P slaves
            s_i = s_i*,    ∀ i = 1..P
Step 3: /* initialization of search parameters (strategy) of
             the P slaves
            st_1 = (taille_liste_tabou_1 , nbmax_1) ;
            st_2 = (taille_liste_tabou_2 , nbmax_2) ;
            …
            st_p = (taille_liste_tabou_p , nbmax_p) ;
Step 4:   for (i=1 to nb_ search_iter) do
Step 4.1: if (i ≠1) then
          - Generate the strategies of theP slaves;
          - Assign the new initial solutions: s_i = s_i* ∀i=1..P;
          endif
Step 4.2: send the new solutions and the solutions to the slaves
Step 4.3: Recover the best solutions found by the slaves s_i*
Step 4.4:  Determine the solution (s_min) of minimum cost
            (cout_min) among the solutions (s_i*);
           If (cout_min < f*)   then   s* =s_min;
           f* =cout_min; endif
done
End.
```

5. Experimentation

In order to analyze the performances of our algorithms, a series of tests have been performed on instances of our problem, some of which were drawn from literature [5][7][22][24]. The tests were carried out on a network of workstations (PC of type Pentium 133 MHZ, 16.0 Mo RAM) operating under C/PVM. Due to the lack of theoretical results regarding our problem, we had no choice but to set the parameters of our algorithms empirically. They are indeed set as a result of our experiments. The values selected are those for which a compromise has been observed between the quality of the solution and the execution time of our algorithms.

Thus, a sample from our population of instances is used to (empirically) tune our parameters before our taboo search is run.

5.1 Comparison between the heuristics

Our benchmark of instances is that of [22]. The parameters used in *diversif_intensif* are : *large_size_taboo* = 7, *step_tenure* = 5 , *normal_size_taboo* = 5. For the sake of conciseness, the Y_{ij}'s are omitted. The values assumed by the remaining parameters are as follows:

	Instances				
n°	G_T: $\lvert V_T \rvert ; \lvert E_T \rvert$	G_P	*nb_local1*	*nb_local2*	*nbre_search_iter*
1	15;16	Hypercube of 8	2	2	3
2	15;14	Torus of 9	2	5	4
3	16;15	Complete of 4	2	7	5
4	16;15	Ring of 7	2	4	2
5	17;23	Grid of 9	2	4	4
6	31;30	Torus of 12	3	3	1
7	9;14	Grid of 9	2	2	4
8	13;24	Torus of 12	5	4	2
9	13;24	Hypercube of 4	2	4	3
10	11;12	Ring of 7	3	7	5
11	17;21	Torus of 16	5	8	5
12	9;8	Complete of 6	3	8	3
13	11;10	Complete of 6	3	10	1
14	10;9	Complete of 6	2	5	2
15	9;14	Complete of 4	2	6	5
16	12;12	Complete of 4	2	6	2
17	8;7	Complete of 4	2	5	4

Table 1. The parameters of algorithm diversif_intensif

The implemented heuristics have been tested on a certain number of instances and compared with the results obtained with genetic algorithms and simulated annealing [24]. The results are displayed in table 2.

n° Ins	diversif_intensif		Parallel_taboo		Genetic Algorithm [24]		simulated annealing [24]	
	cost_sol	T_{cpu} (s)	cost_sol	T_{cpu} (s)	cost_sol	T_{cpu} (s)	cost_sol	T_{cpu} (s)
1	7	0,27	8	0,20	7	0,68	7	0,81
2	17	1,64	20	5,70	20	0,62	20	1,67
3	30	0,43	30	0,36	31	0,61	32	0,57
4	30	0,16	30	0,78	32	0,63	35	1,18
5	35	1,97	94	0,05	41	0,82	58	2,94
6	10	1,86	10	11,34	11	2,05	11	22,08
7	14	0,16	16	0,07	15	0,47	15	0,75
8	31	0,82	33	1,69	31	0,72	33	2,50
9	33	0,27	36	0,02	33	0,66	33	0,64
10	9	1,31	14	0,37	11	0,58	13	0,99
11	16	14,17	17	11,17	19	0,95	21	6,76

Table 2. The results obtained from the different heuristics

Comparison between the sequential and the parallel version of Taboo on one side and between either the genetic algorithms or simulated annealing on the other side, show that the best results (both in terms of the quality of the solution and the execution time) are those of *Algorithm diversif_intensif.* For most of our tests:

- *Parallel_taboo* yields better costs than those of the genetic algorithms and simulated annealing. Its execution time is better than that of simulated annealing and is close to that of genetic algorithms.
- The genetic algorithm gives better results than simulated annealing as far as the quality of the solution is concerned.

With these tests, we note the significant effect of topology and the inter-processors communication costs on the total makespan of our scheduling problem.
Moreover, communication time between tasks also has a considerable effect on the total makespan of the application. For instance, Instance 11 executed on the torus of 16 is the benchmark having given the largest execution times (Tcpu and Tcpu *Parallel_taboo*) as compared with the other benchmarks.
We can conclude that the results depend on the nature of the precedence constraints and the inter-tasks communications. Indeed, whenever the communication costs were small, we have noticed that the tasks were fairly spread on the different processors. In all the other cases, the tasks are often executed on a limited number of processors.

In view of the diversity of settings in which our algorithms most favorably stood the comparison with other heuristics, we can safely conclude that our ingredients of diversification and intensification, along with the idea of tuning the appropriate parameters by sampling, were pertinent and explain to a large extent the good performances of our algorithms. Although these observations should not be considered as a proof of the good behavior of our algorithms, they do provide some

statistical support to our assertion that the ingredients involved in our algorithm are pertinent indeed.

The next two paragraphs provide additional support to our conclusion.

5.3 Comparison with instances whose optimal solutions are known

We have run our algorithms on a few instances of small size (the number of tasks ranges from 9 to 12) whose optimal solutions were found in [5][7][22]. Our results are listed in table 3 below. We notice that:

- *diversif_intensif* finds the optimum for all these instances
- *parallel_taboo* finds the optimum for some of these instances. For other instances, small magnitude deviations from optimum are observed.

n° Instances	*diversif_intensif*		*Parallel_taboo*		*Known optimal solution*
	cost_sol	T_{cpu} (s)	*cost_sol*	T_{cpu} (s)	
12	6	0,49	8	0,53	6
13	14	0,49	16	0,60	14
3	30	0,38	30	0,36	30
14	15	0,16	17	0,70	15
15	14	0,21	15	0,22	14
16	9	0,16	12	0,69	9
17	8	0,05	8	0,18	8

Table 3. The results obtained on instances with known optimal solutions

5.4 Tests on random graphs

Some tests were performed on different instances whose precedence graphs were randomly generated on many architectures (complete networks, rings, torus), in order to evaluate the effect of topology on the total makespan of our scheduling problem. The number of tasks is set to a multiple of 10 and increases uniformly from one generated instance to another.

Instances			cost_sol	T_{cpu}
G_T with	$\|V_T\|$	G_P	diversif_intensif	diversif_intensif
	10	Hypercube of 8	7	0,49
	20	Ring of 5	10	1,59
	30	Torus of 12	13	3,68
	10	Complete of 6	7	0 ,38
	20	Complete of 6	8	0,87
	30	Complete of 2	16	0,71
	30	Complete of 4	11	1,37
	30	Complete of 6	10	1,70
	30	Complete of 8	10	2,04
	30	Hypercube of 8	11	17,41
	30	Ring of 5	11	1,42
	30	Torus of 12	13	3,68

Table 4. Quality and execution time of algorithm diversif_intensif for the different random instances

- The execution time of the taboo search *algorithm diversif_intensif* increases as a function of the size of the precedence graph (number of tasks) and the number of the processors of the architecture.
- Increasing the number of processors of the target architecture would allow to improve the cost of the solution as it would also lead to an increase of the response time of the Taboo algorithm.
- The chosen topology has a considerable impact on the cost of the solution found, as well as on the response time of the taboo search algorithm.

6. NP-completeness results

This section deals with the hardness issues. The problem of deciding whether it is possible to schedule on a given network of processors G_P a given set of tasks endowed with a given precedence relation G_T within a given time bound t is denoted here by $SC(G_T,G_P,t)$. The related optimization problem is denoted by $SC(G_T,G_P)$ (or simply *SC* if the instance (G_T,G_P) is understood). In a first result, we establish the NP-completeness of $SC(G_T,G_P,t)$ for some restricted cases bearing on either the architecture or the precedence relation, thus shedding some light on how difficult our problem is in general. Next, we investigate the case when the graph of the precedence constraints has constant depth, from which we conclude that *SC* admits no polynomial approximation scheme unless P=NP. It should be emphasized that the restricted versions of *SC* established as NP-complete in this section all have a clear connection to the instances treated experimentally in this work.

6.1 NP-completeness results

SC is easily seen to be NP-hard. It is in fact NP-hard for all architectures, provided that the number of processors is large enough. To see this, consider the following reduction from the 3-partition problem. Let $a_1,a_2,...,a_n$ be an instance of 3-Partition. For all a_i associate a path C_i of length a_i representing a chain of a_i tasks each with a unit of execution time. The collection of these n independent paths constitutes a directed acyclic graph G_T. We assume unit communication time and execution time. Consider G_P to be any graph on $n/3$ vertices. Then, clearly, $a_1,a_2,...,a_n$ admits a 3-partition if and only if the so constructed instance of *SC* can be scheduled within time $t = 3 * (\sum a_i / n)$ for $i=1..n$.
Based on this reduction, we can also prove that the restricted case of *SC* when G_T is a rooted tree and unit execution and unit communication time is assumed is also NP-hard, at the expense of complicating the proof.

6.2 Hardness of SC as regards Approximation

The aim of this subsection is to establish the NP-completeness of $SC(G_T,G_p,t)$ when t is constant. Let G be any graph of order n, where n is a multiple of 3. Denote by H the directed acyclic graph depicted in figure 4. Assume all tasks of H and all communication to be of unit length. Then H has the following property: it can be scheduled within time $t \leq 6$ if and only if it is executed by three processors inducing a path of length two in the corresponding architecture. The "only if" part of the assertion is clear. Now, the "if " part. Let P_1, P_2, P_3 be three processors such that $P_1P_2P_3$ is a chain of length 2 in G_p. H is made of an upper, a middle and a lower path of respective lengths 5, 5 and 6 as indicated in the figure. Processor P_1 executes the upper path of H in time 6 including a communication time with P_2 ending at $t=2$. Similarly, P_2 executes the middle path of H in time 6 including a communication time with P_3 that ends at $t=3$. All those communications take a unit time because both the distances between P_2 and P_1 and between P_2 and P_3 are 1 in G_p. Thus, P_3 executes the lower path of T of length 6 within 6 units of time, as it has no communication to do and its fifth task is ready to start at time $t=4$. Now, construct the graph G_T such that G_T is the (vertex-disjoint) union of $n/3$ independent copies of H. Then, from the preceding remark, $SC(G_T,G)$ admits a schedule within time 6 if and only if G admits a partition into paths of length two, which is known to be an NP-complete problem.

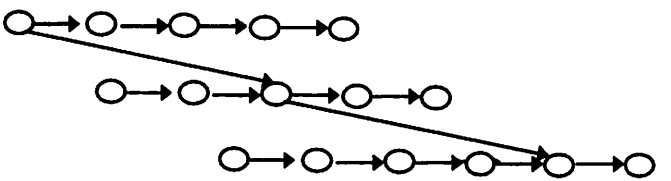

Fig. 4. The component H

Hence, the following theorem and its corollary:

Theorem: *SC($G_T, G_P, 6$)* is NP-complete.

Corollary: SC admits no polynomial approximation ratio less than 7/6 unless P=NP
Thus, *SC* is hard for approximation, in the sense that no polynomial approximation scheme can be found for it unless P=NP. In the remaining part of this section, we are going to prove that this result still holds true when $G_P=K_{|T|}$, the complete graph on $|T|$ vertices, where T is the set of tasks. This particular case is important in its own right because to suppose that $G_P=K_{|T|}$ is tantamount to saying that we have an unlimited supply of processors and an unlimited supply of links between them.
For the purpose of the proof, we define *3-SAT'* to be the restricted version of *3-SAT* in which each variable appears at most three times. We admit the following lemma whose proof is in [10].

Lemma: *3-SAT'* is NP-complete.

Now, we can state the main result of this section.

Theorem: *SC(G_T, $K_{|T|}$, 21)* is NP-complete.

Proof: The reduction is from *3-SAT'*. As the number of processors is the same as the number of tasks in our instance of *SC(G_T, $K_{|T|}$, 21)* to be constructed, we will suppose throughout that we have an unlimited supply of (pairwise connected) processors. Let f be a logical formula instance of 3–SAT'. Let us label its literals by $x_1, x_2, .., x_n$. Represent each literal x_i by a component X_i which is the directed graph constructed as follows:

1. $V(X_i)=\{a\}\cup B\cup X'_i$; B is of cardinality 3 and its vertices are labeled b_1, b_2, b_3. X'_i has three vertices labeled x_i, $\overline{x_i}$ and q_i.
2. $(b_1, x_i)\in E(X_i)$; $(b_2, \overline{x_i})\in E(X_i)$; and $(b_3, q_i)\in E(X_i)$.
3. For all $1\le j\le 3$, $(a, b_j)\in E(X_i)$

This completes the description of the component X_i. As a directed acyclic graph, X_i can be viewed as a set of tasks obeying the underlying precedence constraints.
Every task of X'_i is defined to be of length 1. Task a is of length 2 and every task of B is defined to be of length 3. Communication between any two tasks gives rise to an overhead of 3 units of time. Notice that, if we consider X_i as a precedence relation on tasks, then all its tasks can be scheduled within time t=12 (which is optimal) but then only one task at most from X'_i (that is, either x_i, $\overline{x_i}$ or the "dummy" task q_i) is executed at time less than 12 and its execution time is then necessarily t=9. If the task labeled x_i is the one that is executed at time t=9 in an optimal schedule of X_i, we say that x_i is "ahead of schedule" and "has a lead of 3 units of time over $\overline{x_i}$".
Observe that any of x_i and $\overline{x_i}$ can be made ahead of schedule in an optimal schedule of X_i. The figure below displays X_i along with an optimal schedule. In this schedule, x_i

is ahead of schedule. The assignment of the tasks to the processors is omitted as it can be guessed from the completion times (given from left to right in Figure 5).

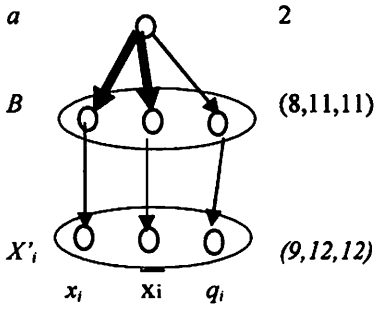

Fig. 5. The component X_i and an optimal schedule Bold arcs denote a communication

Represent each clause C by the component H_C depicted in Figure 6. More formally, Hc is the directed graph defined as follows:

1. $V(H_C)=\{c\}\cup C_1\cup C\cup\{c'\}\cup C'\cup\{c''\}\cup C''\cup D_1\cup D$; each of C, C_1,C',C'',D_1,D has three vertices labeled c1,c2,c3 (respectively c^1_1,c^1_2,c^1_3 $c'_1,c'_2,c'_3;c''_1,c''_2,c''_3;d^1_1,d^1_2,d^1_3$).
2. For all $1\le j\le 3$, $(c,c^1_j)\in E(H_C)$, $(c',c'_j)\in E(H_C)$, $(c'',c''_j)\in E(H_C)$
3. For all $1\le j\le 3$, $(c'_j,d^1_j)\in E(Hc)$; $(c''_j,d^1_j)\in E(H_C)$; $(c^1_j,c_j)\in E(H_C)$.
4. For all $1\le j\le 3$, $(d^1_j,d_j)\in E(H_C)$ and $(c_j,d_j)\in E(H_C)$

Define the length of all tasks of H_C -{c}to be 3, the length of the task c to be 6 and the communication cost between any pair of tasks scheduled on different processors to be 3.

Moreover, for every clause c of f, if variable x_i is the first (resp. second, third) variable appearing in c, then join the vertex labeled x_i in the component X_i to the first (resp. second, third) vertex of the subset C of H_C by an arc and define the communication time between any such pair of tasks to be 1.
This completes the description of our instance I of $SC(G_T,K_{|T|},21)$.

Before proceeding further with the proof, let us mention the following properties of H_C:

(i) As a set of tasks, the optimal makespan for H_C is 21.

(ii) For any optimal schedule of H_C, all three tasks c_1,c_2,c_3 of C must be completed at time 18

(iii) For any optimal schedule of H_C, at least one task from C must be completed at time 15. Moreover, such a task can be chosen to be any of c_1, c_2, c_3. Such a task is said to be ahead of schedule in C, with a lead of 3 units of time.

The complete verification of these facts is left to the reader. However, for the sake of illustration, an optimal schedule is described in our figure by the completion time of each task of H_C, wherein we chose the middle task of C to be ahead of schedule. Similarly, we could have chosen any task of C to be ahead of schedule with a similar pattern of schedule.

Now, we claim that the so constructed instance I of $SC(G_T, K_{|T|}, 21)$ admits a schedule of a makespan ≤ 21 if and only if f is satisfiable.

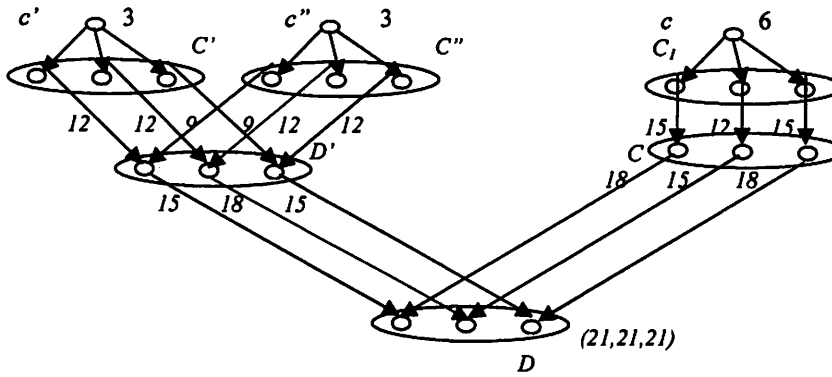

Fig. 6. The component H_C and an optimal schedule

Proof of our claim: Suppose first that f is satisfiable. Given a satisfying assignment for f, let φ be a function such that for all clause c, $\varphi(c)$ is a variable satisfying c in the given assignment. For all clause c, let us set $\varphi(c)=x_c$ for a shorthand and consider the task c_1 of $C \subset H_c$ to which x_c is joined in the above construction. Now, we have the following schedule of I:

1- For all x_c independently and optimally schedule the component X_c associated to x_c in such a way that task x_c is "ahead of schedule" with a lead of 3 units of time over the other tasks of X_c.

2- For all clause C independently and optimally schedule H_C in such a way that c_1 is ahead of schedule in C.

Observe that this schedule is feasible because it takes at most three units of time for x_c to communicate with the other tasks (since every variable appears at most in three clauses of f). This settles the "only if" part of the proof.

Now, the "if" part. Suppose that I can be scheduled within 21 units of time. For all x_i, set variable x_i to "true" if task x_i of X_i is completed at time t=9 (that is, if it is "ahead of schedule in X_i"). Set all other variable to "false".
Now, a schedule of makespan 21 yields an optimal schedule for every component H_C. From the property of H_C, every component H_C must contain a task c_i of C such that c_i is ahead of schedule. Clearly, the variable x_i joined to c_i in H_C is necessarily ahead of schedule in X_i too, which means that it is true in our assignment and consequently, every clause is satisfied.

From this, we conclude that *SC* admits no polynomial approximation ratio less than 22/21 unless P=NP; Hence the following corollary

Corollary: There is no polynomial approximation scheme for $SC(G_T,K_{|T|})$ unless P=NP.

In view of the preceding results, we think that, in fact, *SC* can not be polynomially approximated within a constant ratio but we are unable to prove it.

Conclusion

This work is about scheduling partially ordered tasks on a distributed MIMD architecture. We have proven that the problem is NP-hard even for special cases of interest to us. We also proved that the problem is hard for approximation in the sense that it admits no polynomial approximation scheme unless P=NP. Thus, we were led to adopt a more practical approach to the problem.

In our attempt to solve the problem heuristically, we have proposed two sequential algorithms and a parallel version of taboo search.

The first algorithm represents the Taboo method in its classical form, whereas the second integrates the intensification and diversification processes as special ingredients to guide the search. The third one is a parallelization of the second.
The implemented parallel algorithm is general. Indeed, it is independent of the architecture on which it executes. Moreover, no assumption is made as to the structure of the application. *Parallel_taboo* is a synchronous algorithm in which one of the processors is specified as a master dedicated to the task of providing the "slave" processors with initial solutions and strategies. The slaves apply Taboo_aspiration according to the strategies supplied by the master and send their results back to the master who is entrusted with the task of selecting the best solution.

From our experimental results, we noticed that whenever the communication costs were small, the tasks were fairly spread on the different processors. In all the other cases, the tasks tended to be executed on a limited number of processors.
In order to evaluate the performances of our algorithms, we have compared our results with those of [24]. On many instances, our results proved better than those of

simulated annealing and genetic algorithms. Our *diversif_intensif algorithm* turned out to be very efficient in terms of the quality of solution as compared with *Taboo_aspiration* and *parallel_taboo*. In view of the diversity of settings in which our tests were carried out, our results most convincingly (in statistical terms) indicate that our strategies and the way they were implemented are effective.
On the other hand, *parallel_taboo* has linear speed-up, which accounts for the fact that inter-processor communication costs are small with respect to the executions times of the tasks.
As a perspective to our work:

- a hybridization [13] of this method
- a statistical analysis of the different experimental results [19]
- a sensitivity analysis involving the different parameters of our algorithms
- Utilisation of dynamic programming-based techniques to explore large neighborhoods [4].

Bibliography

[1] N. Afrati and E. Bampis, A.V. Fishkin, K. Jansen, C. Kenyon, Scheduling to Minimize the Average Completion Time of Dedicated Tasks, Foundations of Software Technology and Theoretical Computer Science, pages 454-464, 2000.

[2] A. K. Amoura, E. Bampis, C. Kenyon, Y. Manoussakis, Scheduling Independent Multiprocessor Tasks, European Symposium on Algorithms, pages 1-12, 1997.

[3] B. Andersosn, S. Baruah, J. Jonsson, Static-priority scheduling on multiprocessors, Tech. Rep. UNC-CS TR0116, Department on Computer Science, University of North Carolina at Chapel Hill, May 2001.

[4] E. Angel, E. Bampis, L. Grouves, A dynasearch neighborhood for the bicriteria travelling salesman problem, In Multiple objective metaheuristics: MOMH, Nov. 4-5, Paris, 2002.

[5] P. Bouvry, Placement de taches sur ordinateurs parallèle à mémoire distribuée, PhD Thesis, University of Grenoble, 1994 .

[6] Chandra, M. Adler, P. Shenoy, Deadline fair scheduling : Bridging the theory and practice of proportion scheduling in multiprocessor servers, In Proc. of the 7th IEEE Real-Time Technology and applications Symposium, June 2001.

[7] J.Y. Colin, Problème d'ordonnancement avec délais de communication : complexité et algorithmes, Thèse de doctorat, Université Paris VI . MASI 91. 09 1991

[8] M. Cosnard, Y. Robert, Algorithmique parallèle : une étude de complexité, TSI . Volume 6, n° 2, 1987.

[9] A. De Werra, A. Hertz, Taboo search techniques: a tutorial and an application to neural networks, OR Spektrum, Vol. 11, pp 131, 1989.

[10] M.R. Garey, J.S. Johnson, Computers and Intractability: a guide to the theory of NP Completeness, Freeman and Co, 1979.

[11] F. Glover, Taboo Search - Part I, ORSA Journal of computing , 1(3):190-206, 1989.

[12] F. Glover, Taboo Search Fundamentals and Uses, Journal of heuristics, 1995.

[13] P. Greistorfer, Hybrid Genetic Taboo Search for a Cyclic Scheduling Problem, Meta-heuristics advances and trends in local search paradigms for optimization, Kluwer Academic Publishers, p. 213, 1999.

[14] A. Hertz, M. Widmer, La méthode TABOU appliquée aux problèmes d'ordonnancement, APPII, vol. 29 n° 45, pp 353-378, 1995.

[15] W. Jakob, M. Georges-Schleuter, C. Blume, Application of genetic algorithms to task planning and learning, Parallel problem solving from Nature PPSN'2, Amsterdam(North Holland), p.291-300, 1992.
[16] K. Jansen, L. Porkolab, Improved Approximation Schemes for Scheduling Unrelated Parallel Machines, ACM Symposium on Theory of Computing, 1999.
[17] K. Jansen, L. Porkolab, General Multiprocessor Task Scheduling: Approximate Solutions in Linear Time, Workshop on Algorithms and Data Structures, pp 110-121, 1999.
[18] Y.K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys, Vol 31, n°4, pp 406-471, 1999.
[19] Y.K. Kwok, I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, Journal of parallel and distributed computing, Volume 59, n°3, 381-422, 1999.
[20] A. Munier, Approximation algorithms for scheduling trees with general communication, Journal Parallel Computing, 25 n°1, 41-48, 1999.
[21] M. Norman, P. Thanish. Models of machines and computation for mapping in multicomputers. ACM Computing Surveys, Septembre 1993.
[22] C. Picouleau, Etude de problèmes d'optimisation dans les systèmes distribués, PhD Thesis, University Paris VII, 1992.
[23] A. Radulescu, C. Nicolescu, A.J.C. Van Gemund, P.P. Jonker, Mixed Task and data parallel scheduling for distributed system, In CDROM proceedings of the 15th International Parallel and distributed Symposium, San Francisco, California, 39-41, 2001
[24] M. Rahoual, J.C. Konig, Application de méta-heuristique au problème d'ordonnancement statique de processus sur architectures parallèles, Revue Calculateurs Parallèles, Vol 10 n° 6, ed HERMES 1998.
[25] Srinivasan, J. Anderson, Optimal rate-based scheduling on multiprocessors, 34th Annual ACM Symposium on Theory of computing, 2001.
[26] E.G. Talbi, T. Muntean, Méthodes de placement statique des processus sur architectures parallèles, TSI. Volume 10, n° 5, 1991.
[27] E-G. Talbi, Allocation de processus sur les architectures parallèles à mémoire distribuées, PhD Thesis, Institut National polytechnique de Grenoble, 1993.
[28] E. G. Talbi, Z. Hafidi, J-M. Geib, A parallel adaptive taboo search approach, Journal Parallel Computing, volume 24, number 14, pp 2003-2019, 1998.
[29] A-S. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[30] C-L.Wu, T-Y. Feug, Interconnection networks for parallel and distributed processing, IEEE Computer Society Press, Washington, 1984.