Tabu Search Type Algorithms for the Multiprocessor Scheduling Problem

Olivier BUFFET∗ Liliana CUCU† Lhassane IDOUMGHAR‡ René SCHOTT§

Abstract

This paper presents two Tabu Search type algorithms for solving the multiprocessor scheduling problem. This problem consists in finding a schedule for a general task graph to be executed on a multiprocessor system so that the schedule length can be minimized. The multiprocessor scheduling problem is known to be NP-hard, and to obtain optimal and suboptimal solutions, several heuristic based algorithms have been developed in [1, 2, 4, 6]. Our approaches are validated on 13 randomly generated instances. The numerical results show that our algorithms produce solutions closer to optimality and/or of better quality than the methods presented in [1].

1 Introduction

The problem of multiprocessor scheduling consists in finding a schedule for a general task graph (for example, figure 1) to be executed on a multiprocessor system so that the schedule length can be minimized. Each vertex is a task and edges are precedence relations between pairs of tasks. The sense of the arrow shows the direction of the precedence.

This scheduling problem is known to be NP-hard, and several approaches based on heuristic search have been developed to obtain optimal or (more often) suboptimal solutions. Efficient methods based on genetic algorithms, descent of gradient and Tabu Search [1, 2, 4, 5, 6] have been developed to solve the processor scheduling problem. [1] contains, to the best of our knowledge, the most efficient heuristics for the multiprocessor scheduling problem. This paper presents two Tabu Search type algorithms for solving more efficiently this problem than in [1].

The organization is as follows: Section 2 describes the model for multiprocessor schedulings. Section 3 recalls some hints on Tabu Search algorithms. In Section 4 we explain our contribution. Section 5 presents our experimental results and performance comparisons with the approach developed in [1]. Concluding remarks and further research aspects are contained in the last section.

∗Nancy-Université, INRIA and LORIA, 54506 Vandoeuvre-les-Nancy, France. Email: buffet@loria.fr
†Nancy-Université, INRIA and LORIA, 54506 Vandoeuvre-les-Nancy, France. Email: cucu@loria.fr
‡LMIA, Université de Haute-Alsace, 4 rue des Frères Lumière, 68093, Mulhouse, France. Email: lhassane.idoumghar@uha.fr
§Nancy-Université, Université Henri Poincaré, IECN and LORIA, 54506 Vandoeuvre-les-Nancy, France. Email: schott@loria.fr
2 Model Description

Consider a set of partially ordered computational tasks represented by a directed acyclic graph\(^1\) consisting of a finite nonempty set of vertices \(V\) and a set of finite directed edges \(E\) connecting the vertices.

The collection of vertices, \(V = \{T_1, T_2, ..., T_m\}\) represents the set of computational tasks to be executed. Let \(e_{ij}\) denote a directed edge from vertex \(T_i\) to \(T_j\). The set of directed edges, \(E = \{e_{ij}\}\) implies that a partial ordering (or precedence relation), \(\rightarrow\), exists between the tasks. \(T_i \rightarrow T_j\) means that task \(T_i\) must be completed before \(T_j\) can be initiated.

The problem of optimally scheduling a task graph onto a multiprocessor system with \(p\) processors consists of assigning the computational tasks to the processors, so that the precedence relations are maintained and all tasks are completed in shortest (possible) time. If task \(T_i\) is an ancestor of task \(T_j\) (i.e. if \(T_i\) must be executed before \(T_j\)), then we say that \(\text{height}(T_i) < \text{height}(T_j)\) where \(\text{height}(T_i) = 0\) if the set \(\text{PRECE}(T_i)\) of predecessors of \(T_i\) is empty and \(\text{height}(T_i) = 1 + \max_{T_j \in \text{PRECE}(T_i)} \text{height}(T_j)\) otherwise. This height function conveys the precedence relations between tasks. For example, in figure 1 we can see that:

- \(\text{height}(T_1) = \text{height}(T_2) = \text{height}(T_3) = 0\)
- \(\text{height}(T_4) = \text{height}(T_5) = 1\)
- \(\text{height}(T_6) = \text{height}(T_7) = 2\)

3 Tabu Search Algorithm

The basic idea of the Tabu Search metaheuristic [3] is to explore the search space of all feasible solutions by a sequence of moves. A move from one solution to another is generally the best available. However, in order to prevent oscillation and to provide a mechanism for escaping from locally optimal but not globally optimal solutions, some moves, at one particular iteration, are classified as forbidden or tabu. Moves are regarded as tabu by consideration of short-term and long-term history of the sequence of moves. A very simple use of this idea might be to classify a move as tabu if the reverse move has been made recently or frequently. There is also an aspiration criterion, which overrides the tabu moves in particular circumstances. These circumstances might include the cases where, by forgetting that a move is tabu, a solution which is the best so far is obtained.

Suppose it is required to minimize some cost function \(F\) on the search space \(S\). For combinatorially hard problems it may only be possible to obtain sub-optimal solutions, in which \(F\) is close to its minimum value. Sub-optimal problems may be obtained when a certain threshold for an acceptable solution has been achieved or when a certain number of iterations have been completed.

A characterisation of the search space \(S\) for which Tabu Search can be applied is when there exists a set of \(k\) moves \(D = \{d_1,...,d_k\}\) such that the application of the moves to a feasible solution \(s \in S\) leads to \(k\) (usually distinct) solutions \(D(s) = \{d_1(s),...,d_k(s)\}\). The subset \(N_{set}(s) \subseteq D(s)\) of feasible solutions is known as the neighbourhood of \(s\).

The method starts with a (possibly random) solution \(s_0 \in S\) and determines a sequence of solutions \(s_0, s_1, ..., s_n \in S\). At each iteration, \(s_{j+1}\) \((0 \leq j < n)\) is selected from the neighbourhood \(N_{set}(s_j)\). The selection process is first to determine the tabu set \(T_{set}(s_j) \subseteq N_{set}(s_j)\) and the aspiration set \(A_{set}(s_j) \subseteq N_{set}(s_j)\). Then \(s_{j+1}\) is the neighbour of \(s_j\) which is either an aspirant or not tabu and for which \(F(s_{j+1})\) is minimal; that is, \(F(s_{j+1}) \leq F(s_j)\) for all \(s_i \in (N_{set}(s_j) - T_{set}(s_j)) \cup A_{set}(s_j)\).

In the next section we will present different elements used by our algorithm based on a refined modeling of the input data and on a Tabu Search.

4 Our approach

4.0.1 Principle of our algorithm

The principle of our Tabu Search algorithm is described in Algorithm 1, where:

- Initial solution: For the multiprocessor scheduling problem, a legal schedule is one that satisfies the following conditions:
  1. The precedence relations among the tasks are satisfied.
  2. Every task is present and appears only once in the schedule (completeness and uniqueness).

---

\(^1\)We use the graph model proposed in [4].
The solution representation used in this paper is based on the schedule of the tasks on each individual processor [4]. This representation eliminates the need to consider the precedence relations between the tasks scheduled to different processors. The precedence relations within the processor, however, must still be maintained.

The representation of a schedule for Tabu Search Algorithms must accommodate the precedence relations between the computational tasks. This is resolved by representing the schedule as several lists of computational tasks. Each list corresponds to the computational tasks executed on a processor, and the order of the tasks in the list indicates the order of execution. Figure 2 illustrates the list representation of the schedule in Figure 1. This ordering allows us to maintain the precedence relations for the tasks executed on a processor (intraprocessor precedence relation) and ignores the precedence relations between tasks executed on different processors (interprocessor precedence relation). This is due to the fact that the interprocessor precedence relations do not come into play until we actually calculate the finishing time of the schedule. Each list can be further viewed as a specific permutation of the tasks in the list (allowing the last task to map to the first task). Figures 3 and 4 illustrate the permutation representation of the schedule in Figure 2. Thus, a schedule for n tasks and p processors is a permutation of n numbers with p cycles. The permutation representation of schedules is useful when we actually implement the function that generates the neighbour solutions.

Note that not every permutation of n numbers with p cycles corresponds to a legal schedule because of the precedence relations. This representation of schedules falls into the following category: The solution space is not in one-to-one correspondence with the search space. We must bear this in mind when we design the neighbouring solutions.

• Variable best_solution keeps the best solution found,

• current_iteration counts the iterations of the main loop and Notimprove remembers the last iteration in which the algorithm improved the best solution.

• Evaluate or objective function: For the multiprocessor scheduling problem, we can consider factors such as throughput, finishing time, and processor utilization for the fitness function.

The objective function used for our TS approaches (also defined in [4]) is based on the finishing time of the schedule. The finishing time (FT) of a current schedule, $S_c$, is defined as follows:

$$FT(S_c) = \max_{P_j} \text{ftp}(P_j)$$

where \( \text{ftp}(P_j) \) is the finishing time for the last task in processor \( P_j \).

Thus, the optimal schedule founded by the Tabu Search algorithms must have the smallest finishing time value than the other schedules.

```
1 current_iteration ← 0, Notimprove ← 0
2 Let current_solution be an initial solution and let best_solution be a best solution
3 best_solution ← current_solution
4 stopCriterion ← maximum iterations or optimal_solution is not attained or Notimprove = K
5 tabu_list ← K
6 Compute theorical min_cost
7 current_cost ← Evaluate (current_solution)
8 best_cost ← current_cost
9 while Not stopCriterion do
10     Neighbour ← Generate (current_solution)
11     Neighbour_cost ← Evaluate (Neighbour)
12     if Neighbour_cost < best_cost then
13         best_solution ← Neighbour
14         best_cost ← Neighbour_cost
15         Notimprove ← 0
16     else
17         Notimprove ← Notimprove + 1
18     end
19     if Neighbour_cost = current_cost then
20         max_repetition ← max_repetition + 1
21     end
22     current_solution ← Neighbour
23     currnet_cost ← Neighbour_cost
24     if max_repetition = 4 then
25         Diversification (current_solution)
26     end
27     current_iteration ← current_iteration + 1
28     Update (tabu_list)
29     Update (stopCriterion)
30 end
```

Algorithm 1: Our Tabu Search approach.

• Intensification: The role of the intensification is to drive the search towards interesting regions, by taking into account characteristics found in high quality solutions, in the selection of future solutions to visit. For each processor, our Tabu search algorithm checks if the possible permutation of the tasks would improve the quality of the current solution.

• Diversification: Tabu Search is based on Local Search principle. One of its main problems is that it spends most, if not all, of its time in a restricted region of the search space.
In order to avoid that a large region of the search space remains completely unexplored, it is important to diversify the search.

In our approach, the diversification technique used tries, randomly, either to penalize the current solution or to perform a random restart.

- The tabu list is updated in line 28.

- stop_criterion: this criterion indicates when the search process will be stopped. The update procedure verifies the following conditions:

  - maximum number of iterations is not attained
  - or Optimal_solution is not found (best_cost ≠ min_cost)
  - or Notimprove < (K = 60)

Our first implementation (called TS_version1) of Tabu Search Algorithm consists in adding solutions to the tabu list (with size = 7).

The second implementation (called TS_version2) forbids some characteristics of the solutions. The neighbour of each solution is defined by two types of permutation (see Figures 3 and 4). To each of them we are going to associate a specific characteristic:

1. for each processor attributed to task i, there exist i − 1 possible permutations between adjacent tasks, we identify them in a natural way by their position. When our Tabu Search algorithm escapes from a local minimum thanks to that modification, the corresponding permutation becomes tabu.

2. when our Tabu Search Algorithm escapes from a local minimum by changing the processor attributed to a task (for example, Figure 4), the previous attribution becomes tabu.

5 Experimental results

The Tabu Search algorithms have been implemented on a PC centrino, with a 1.8 Ghz processor and 512 Mbytes of memory running under Mandriva Linux 2007.

In order to compare the different scheduling approaches, we have generated randomly systems of 50 tasks. A total of 10 runs for each algorithm were conducted and the number of processors used to solve each problem is fixed to 10.

The best results obtained by all algorithms are given in Figures 5, 6 and 7.

Analysing these figures, we can see:

- In Figure 5, TS_version1 produces almost every time the optimal solution for almost all instances and improves DGV ([1]).

- In Figure 6, For 77% of the examples considered, TS_version1 provides rapidly a better solution than DGV ([1]). For the 23% remaining cases, TS_version1 takes a little bit more time than DGV but produces solutions of better quality than DGV.

- In Figure 7, TS_version2 provides more quickly a better solution than DGX ([1]). The improvements of computation times are due to the stopping criterion as well as to programming enhancements (for example, avoiding calculation of the cost at each iteration). DGX ([1]) and TS_version2 produce both the same good solution but TS_version2 needs less time.
6 Concluding remarks

Our Tabu Search type algorithms improve drastically the performances of these presented in [1]. We are currently implementing our algorithm on clusters of PC’s. Further research aspects include the design of a probabilist Tabu Search Algorithm for the multiprocessor problem as well as the use of Markov Decision Processes (MDP’s) and Particle Swarm Optimization (PSO). Designing hybrid algorithms for solving this problem is a real challenge.

References