Performance Evaluation of List Based Scheduling on Heterogeneous Systems

Hamid Arabnejad and Jorge G. Barbosa

Universidade do Porto, Faculdade de Engenharia, Dep. de Engenharia Informática,
Laboratório de Intelegência Artificial e Ciência dos Computadores,
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
{hamid.arabnejad,jbarbosa}@fe.up.pt

Abstract. This paper addresses the problem of evaluating the schedules produced by list based scheduling algorithms, with metaheuristic algorithms. Task scheduling in heterogeneous systems is a NP-problem, therefore several heuristic approaches were proposed to solve it. These heuristics are categorized into several classes, such as list based, clustering and task duplication scheduling. Here we consider the list scheduling approach. The objective of this study is to assess the solutions obtained by list based algorithms to verify the space of improvement that new heuristics can have considering the solutions obtained with metaheuritcs that are higher time complexity approaches. We concluded that for a low Communication to Computation Ratio (CCR) of 0.1, the schedules given by the list scheduling approach is in average close to metaheuristic solutions. And for CCRs up to 1 the solutions are below 11% worse than the metaheuristic solutions, showing that it may not be worth to use higher complexity approaches and that the space to improve is narrow.

1 Introduction

The problems of task matching and scheduling, in general, are to resolve a composite parallel program into several tasks and assign these tasks to a set of processor elements (PEs) to execute. These tasks have restriction of priority order to execute with each other due to its characteristic of data dependencies. The relationship among the tasks can be represented by a weighted Direct Acyclic Graph (DAG). Also, the processing elements are connected by a high speed communication network. Task matching is to assign a specific task to a suitable processing element to execute; and scheduling is to determine execution priority of each task among the composite parallel program. The general form of the problem has already been proved to be $NP-complete$ [2,11,14,15]. Although it is possible to formulate and search for the optimal solution, the feasible solution space quickly becomes intractable for larger problem instance. To overcome the exponential time complexity, heuristic based scheduling algorithms of been proposed that found a sub-optimal solution in polynomial time. These heuristics are categorized into several classes, mainly list based, clustering and task duplication scheduling. Among these, list scheduling algorithms are generally regarded as having a good cost performance trade-off because of their low

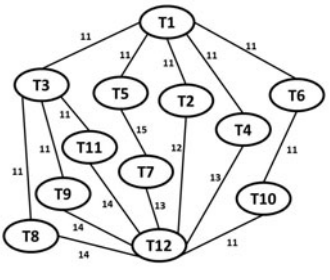
M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 440–449, 2012.

cost and acceptable results. In list scheduling, tasks are sorted by their priorities and scheduled accordingly [3,6,9,13,17,18]. Although these algorithms can find a feasible solution in polynomial time they are not able to guarantee to find a suitable solution when size of the problem becomes large. In this paper we evaluate the quality of the solutions obtained by two best list scheduling algorithms, namely HEFT and CPOP [18], for heterogeneous systems by comparing with the solutions obtained by metaheuristic algorithms. Once these last algorithms do not guarantee the optimal solution, we obtain for each scheduling the best solution and measure the distance to the list scheduling solution. The metaheuristic algorithms considered in this study are Ant Colony System (ACS), Simulated annealing (SA) and Tabu Search (TA).

At first, we introduce the DAG scheduling problem, then describe two static list scheduling algorithms, HEFT and CPOP. Followed by an introduction to the Ant Colony System, Simulated Annealing and Tabu Search. Further, we describe the design and the implementation on these algorithms with a discussion about the results achieved.

2 DAG Scheduling

A scheduling system model represented by a direct acyclic graph (DAG), $G = (V, E, P, W, data, rate)$, where V is set of v tasks, E is the set of e edges between tasks, and P is the set of processors available in the system. Each $edge(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started. A task with no predecessors is called an *entry* task, n_{entry}, and n_{exit} is one with no successors. W is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. Figure 1 shows an example of a DAG comprising 12 tasks to illustrate these definitions graphically. It can be seen that the immediate successors of t_3 are t_8, t_9 and t_{11}; the immediate predecessors of t_{10} is t_6. Furthermore, t_1 is an entry task and t_{12} represents a pseudo exit-task.



	P1	P2	P3	P4
T1	7	13	11	23
T2	17	23	10	9
T3	7	20	14	16
T4	24	8	21	13
T5	24	14	21	17
T6	21	11	8	13
T7	23	9	10	24
T8	7	18	19	24
T9	14	9	24	24
T10	19	10	19	8
T11	21	15	7	25
T12	6	13	20	10

Fig. 1. Example of a DAG and its computation costs matrix [CCR=0.8]

Each $w_{i,j}$ gives the estimated execution time to complete task n_i on processor p_j. The average of execution cost of a node n_i is defined $\overline{w_i} = (\sum_{j \in P} w_{i,j})/p$.

The *data* parameter is a $v \times v$ matrix of communication data, where $data(i,j)$ is the amount of data required to be transmitted from task n_i to task n_j. The *rate* parameter is a $p \times p$ matrix and represent the data transfer rate between processors. The communication cost of $edge(i,j)$, which is for data transfer from task n_i (scheduled on processor p_m) to task n_j (scheduled on processor p_n), is defined by $c_{i,j} = data(n_i, n_j)/rate(p_m, p_n)$. When both n_i and n_j are scheduled on the same processor ($p_m = p_n$), then $c_{i,j}$ becomes zero. The average communication cost of an edge is defined by $\overline{c_{i,j}} = data(n_i, n_j)/\overline{rate}$, where $\overline{rate}$ is the average transfer rate between the processors in the domain.

The $EST(n_i, p_j)$ and $EFT(n_i, p_j)$ are the *Earliest Execution Start time* and the *Earliest Execution Finish Time* of node n_i on processor p_j. For the entry task $EST(n_{entry}, p_j) = 0$. For other tasks, the EST and EFT values are computed recursively, starting from the entry task as shown by

$$EST(n_i, p_j) = \max\{T_{Available}(p_j), \max_{n_m \in pred(n_i)}\{AFT(n_m) + c_{m,i}\}\}$$

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j)$$

where $pred(n_i)$ is the set of immediate predecessor tasks of task n_i and $T_{Available}(p_j)$ is the earliest time at which processor p_j is available for task execution. The inner *max* block in the EST equation returns the *ready time*, i.e, the time when all data needed by n_i has arrived at the processor p_j.

The *objective function* of the scheduling problem is to determine the assignment of task of a given application to processors such that the schedule length or *makespan* is minimized. After a task n_i is scheduled on processor p_j, the Actual Start Time of node n_i ($AST(n_i)$) is equal to $EST(n_i)$ and the Actual Finish Time of node n_i ($AFT(n_i)$) is equal to $EFT(n_i)$. After all nodes in the DAG are scheduled, the schedule length will be $makespan = max[\ AFT(n_{exit})]$, i.e. the Actual Finish Time of exit task.

The *Critial Path* (CP) of a DAG is the longest path from the *entry* node to the *exit* node in the graph. The length of this path $|CP|$ is the sum of the computation cost of the nodes and inter-node communication costs along the path. The $|CP|$ value of a DAG is the lower bound of the schedule length.

3 List Scheduling Algorithms

The list scheduling technique [12] has the following steps: a) determine the available tasks to schedule, b) assign a priorities to them and c) until all tasks are scheduled, select the task with the highest priority and assign it to the processor that allows the earliest start time.

Two attributes frequently used to define the tasks priorities are the *upward* and the *downward* ranks. The *downward rank* of a node n_i ($rank_d$) is defined as the length of the longest path from an entry node to n_i (excluding n_i). The *upward rank* of a node n_i ($rank_u$) is the length of the longest path from n_i to an exit node. The nodes of the DAG with higher $rank_u$ values belong to the critical path.

3.1 HEFT Algorithm

The HEFT (Heterogeneous Earliest Finish Time) algorithm [18] is highly competitive in that it generates a comparable schedule length to other scheduling algorithms, with a low time complexity. The HEFT algorithm is an application scheduling algorithm for a bounded number of heterogeneous processors, which has two major phases: a *task prioritizing* phase for computing the priorities of all tasks and a *processor selection* phase for selecting the tasks in the order of their priorities and scheduling each selected task on its best processor, which minimizes the task's finish time. In HEFT algorithm, tasks are ordered by their scheduling priorities that are based on upward ranking ($rank_u$).

Algorithm 1. The HEFT algorithm

```
Compute rank_u(n_i) for all n_i ∈ V
ReadyTaskList ← Start Node
while ReadyTaskList ≠ Empty do
   n_i ← node with the maximum rank_u in ReadyTaskList
   for all p_j ∈ P do
      Compute EST(n_i, p_j)
      EFT(n_i, p_j) ← w_{i,j} + EST(n_i, p_j)
   end for
   Map node n_i on processor p_j which provides its least EFT
   Update T_Available(p_j) and ReadyTaskList
end while
```

3.2 CPOP Algorithm

The critical path (CP)is the longest path in a DAG. The Critical Path on Processor (CPOP) algorithm is a variant of the HEFT algorithm [18]. CPOP adopts a different mapping strategy for the critical path nodes and the non-critical path nodes. A CP processor is defined as the processor that minimizes the overall execution time of the critical path assuming all the critical path nodes are mapped onto it. If the selected node is a critical path node, it is mapped onto the CP processor. Otherwise, it is mapped onto a processor that minimizes its EFT (like in the HEFT algorithm).

Algorithm 2. The CPOP algorithm

```
Compute rank_u(n_i) and rank_d(n_i) for all n_i ∈ V
Identify the Critical Paths and mark the Critical Path Nodes
priority(n_i) ← rank_u(n_i) + rank_d(n_i)
ReadyTaskList ← Start Node
while ReadyTaskList ≠ Empty do
   n_i ← node with the maximum rank_u in ReadyTaskList
   if n_i ∈ Critical Path then
      Map n_i on the CP Processor
   else
      for all p_j in P do
         Compute EST(n_i, p_j)
         EFT(n_i, p_j) ← w_{i,j} + EST(n_i, p_j)
      end for
      Map node n_i on processor p_j which provides its least EFT
   end if
   Update T_Available(p_j)
   Update ReadyTaskList
end while
```

4 Metaheuristic Algorithms

4.1 Ant Colony System

Ant colony system (ACS) is a metaheuristic that was first proposed by Dorigo and Gambardella [4], it is one of the most popular swarm inspired methods in computational intelligence areas. And latter adapted to discrete optimization problems [5]. The basic idea is to imitate the cooperative behaviour of real ants, to solve optimization problems. At first, ants have no clue about which way belongs to the shortest path to nest, so they choose randomly. Once the ants discover a paths from nest to food, they changed pheromone on the path. So another ants can follow the trails to find the food source. The ants that found the shortest path will come back to nest sooner, than ants via longer paths, and that path will have higher traffic. As this process continuous, the shortest paths have a huge amount of pheromone and most of ants tend to choose these paths. ACS includes five steps: (1) ants initialization to positioning (2) for each ant applied a state transition rule to incrementally build a solution and a local pheromone updating rule (3) Global pheromone updating (4) ending test to evaluate the best solution that if it is not acceptable go to step 1.

To apply the ACS meta-heuristic to the task scheduling problem, we need to translate this problem into the structure of ACS so that ants can find solutions. For this purpose, we considered a Graph with two subgraphs G_1 and G_2, where the first represents the set of tasks to schedule and the second denotes the set of processors available. At each iteration, each ant selects a source node and a suitable processor based on a selection rule. Then we add tasks that are ready to schedule, i.e. tasks where their predecessors have been scheduled, and this procedure continues until all task are scheduled.

In ACS (Ant Colony System) the state transition rule provides a direct way to balance between exploration of new edges and exploitation of a priori and accumulated knowledge about the problem. It is defined as follows: an ant positioned on task i chooses the processor u to move to by applying the rule given by

$$Prob(i,p) = \begin{cases} \max\left[\tau(i,p) \times [\eta(i,p)]^{\beta}\right] & \text{if } q_0 < q(\text{exploitation}) \\ \dfrac{\tau(i,p) \times [\eta(i,p)]^{\beta}}{\sum\limits_{q \in P} (\tau(i,q) \times [\eta(i,q)]^{\beta})} & \text{otherwise (biased exploration)} \end{cases}$$

where q is a random number uniformly distributed in [0..1] and q_0 is a parameter ($0 \leq q_0 \leq 1$). Tuning the parameter q_0 allows modulation of the degree of exploration and the choice of whether to concentrate the search of the system around the best-so-far solution or to explore other tours, here $q_0 = 0.7$. And $\eta(n,p) = 1/AFT(n_{i,p})$ is the heuristic function and $\beta = 2$ is a parameter which determine the relative influence of the heuristic information.

The *Global Pheromone Update* Rule is performed only by the best ants that have the shortest path from source to sink. This rule besides the use of the pseudo-random-proportional rule, cause to encourage the ants in next iterations to search in a neighbourhood of the best path found up to current iteration. After all ants finished their tour, we can perform global updating for current iteration. The pheromone level is updated by applying the global updating rule $\tau(i,p) = (1-\rho)\cdot\tau(i,p) + \rho\cdot\Delta\tau(i,p)$ where $\Delta\tau(i,p)$ for global best tour is $\Delta\tau(i,p) = 1/\left[AFT_{best\ ant}(n_{exit})\right]$ and for other nodes is $\Delta\tau(i,p) = 0$. Also, $0 < \rho < 1$ is the pheromone decay parameter and here is $\rho = 0.1$. In addition to the global pheromone trail updating rule, in ACS the ants use a *Local Pheromone Update* rule in each iteration since each ant by choosing a processor p for task i, is applied by $\tau(i,p) = (1-\xi)\cdot\tau(i,p) + \xi\cdot\tau_0$ where $0 < \xi < 1$ denotes the pheromone decay parameter and $\tau_0 = \frac{1}{|V|}$ is the initial value of pheromone on all edges. Experimentally, a good value for ξ was found to be $\xi = 0.1$.

4.2 Simulated Annealing

Simulated Annealing (SA) is a generic probabilistic meta-algorithm proposed by Kirpatrick, Gelett and Vecchi [10] and Cerny [1] used to find an approximate solution to global optimization problems. It is inspired by annealing in metallurgy which is a technique of controlled cooling of material to reduce defects. In simulated annealing, a cost function to be minimized is defined in terms of the parameters of the problem at hand. The cost minimization process is governed by a cooling temperature which varies from a given high value to a low value slowly. At every temperature, we generate a fixed number of scheduling and calculate cost function(makespan) for each of them. If the cost function is less than the previous cost, the new configuration is accepted. If the cost is more than the previous one, the new configuration is chosen with a probability $r \leq exp(-\Delta C/T_k)$ where $r \in [0,1]$. Probabilistic acceptance of costlier solutions is behind the success of the simulated annealing process. Actually, when $\Delta C \leq 0$, we have a *downhill* step, that means a search for a new solution around a best solution. But if this condition is not satisfied, we can use the new solution instead of the best solution, with higher cost (*uphill* step) and helps the solution process overcome the possibility of getting trapped in a local minimum and move toward the global minimum. The three most important parts are: (1)*Cost Function* that is the schedule length of the solution; (2) *Generating mechanism* to randomly generate a scheduling of a set of tasks; and, (3) *Cooling mechanism* that initializes the temperature to a value T_0, and in each step, it decreases by $T_{k+1} = \alpha \times T_k$ and $\alpha = 0.1$ if we use a higher value for α we will move faster and we would have less exploration of the search space.

In our implementation the length of Markov chain is $|V|$, final temperature is 0.01, initial temperature is $\left[best_{makespan}(S_i) - worst_{makespan}(S_i)\right]/\log(0.9)$, where S_i is the initial solution, and the initial value of the cost function C is given by the *makespan*.

Algorithm 3. The Simulated Annealing algorithm

```
Create an initial(feasible) solution s;
Set an initial temperature T_0 (with k ← 0);
Set number of trials at each temperature level (level-length) α
while termination criterion not satisfied do
   for i = 0 → length_{Markov chain} do
      Create new neighbour s' by applying a random move to s;
      Calculate cost difference ΔC between s' and s : ΔC = C(s') − C(s);
      if ΔC ≤ 0 then
         Switch over to solution s' (current solution s is replaced by s');
      else
         Create random number r ∈ [0, 1];
         if r ≤ exp(−ΔC/T_k) then
            Switch over to solution s' (current solution s is replaced by s');
         end if
      end if
   end for
   Update best found solution (if necessary);
   Set k ← k + 1 and Set / Update temperature value T_k for next level k;
end while
return Best found solution
```

4.3 Tabu Search

Tabu search (TS) is one the a heuristic methods proposed by Glover [7] [8]. Unlike other meta-heuristics, in TS, we have an intelligent search to perform a systematic exploration of the solution space. The main idea in TS is to use the information about search history to guide local search approaches to overcome local optimality. In general we examine a path sequence of solutions and moves to the best neighbour of the current solution and, to avoid cycling, solutions that were recently examined are forbidden or tabu. Elements of Tabu Search: 1) *Tabu List* (short term memory): to record solutions to prevent revisiting a visited solution; 2) *Tabu tenure*: number of iterations a tabu move is considered to remain tabu; 3) *Aspiration criteria*: accepting an improved solution even if generated by a tabu move 4) *Long term memory*: to record attributes of elite solutions to be used in: a) Intensification (giving priority to attributes of a set of elite solutions); b) Diversification (Discouraging attributes of elite solutions in selection functions in order to diversify the search to other areas of solution space).

Algorithm 4. The Tabu Search algorithm

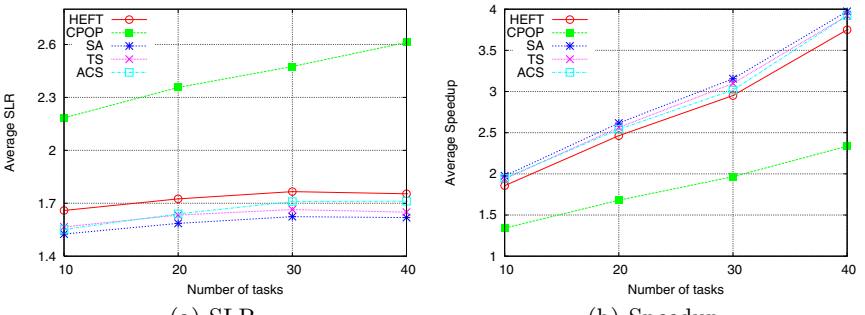
```
S ← random valuation of variables;
iter ← 0;
initialize randomly the tabu_list
while (eval(S)> 0) and (iter < Maxiter) do
   choose a move < V, v' > with the best performance among the non-tabu moves and the
   moves satisfying the aspiration criteria;
   introduce < V, v > in the tabu_list, where v is the current value of V
   remove the oldest move from the tabu_list
   assign v' to V;
   iter ← iter + 1;
end while
return S
```

5 Results and Conclusions

In this section, we evaluate and compare the solution performance of the HEFT and CPOP with metaheuristic algorithms for single DAG scheduling using an extensive simulation setup. The metrics used for comparison are the **SLR (schedule length ratio)** and the **Speedup** (used in [18]). In fact, the SLR metric make a normalization on the schedule length to a lower bound.

$$SLR = \frac{makespan(solution)}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in P} (w_{(i,j)})} \qquad Speedup = \frac{\min_{p_j \in P} \left[\sum_{n_i \in V} w_{(i,j)} \right]}{makespan(solution)}$$

The denominator in SLR is the minimum computation of tasks on critical path. With any algorithm, there is no makespan less than the denominator of SLR equation. Therefore, the algorithm with lower SLR is the best algorithm. Average SLR values over several task graphs are used in our results. In *Speedup*, the sequential time is obtained by the sum of the processing time on the processor that minimizes the total computation cost [18]. The DAGs used in this simulation setup were randomly generated using the program in [16] which considers the following parameters: *width* as the number of tasks on the largest level; *regularity* is the uniformity of the number of tasks in each level; *density* is the number of edges between two levels of the DAG. These parameters may vary between 0 and 1. An additional parameter, *jump*, indicates that an edge can go from level l to level $l + jump$. In this paper, we consider DAGs with 10, 20, 30 and 40 tasks; the number of processors equal to 4, 8, 16, and 32; CCR of 0.1, 0.5, 0.8 and 1; width equal to 0.1, 0.2, 0.8; density equal to 0.2, 0.8; and jumps of 1, 2, and 4. These combinations give 1152 different DAG types. Since 5 random DAGs were generate for each combination, the total number of DAGs used in our experiment was 5760. We do not considers CCR above 1 because for a high speed network it would not be a realistic value.



(a) SLR (b) Speedup

Fig. 2. The SLR and Speedup average values for each size graph and CCR=[0.1 0.5 0.8 1.0]

Figure 2 shows the results of SLR and Speedup for list scheduling algorithms (HEFT and CPOP) and metaheuristic algorithms (ACS, TS and SA). It can be observed that in average there is a consistent gap between the two types of algorithms, being the best solutions obtained by the Simulated Annealing metaheuristic. Also HEFT has always better performance than CPOP, as shown in [18]. Considering the results shown on table 1, it can be concluded that for low CCR (0.1) the HEFT gives near results comparing to metaheuristic approaches. This means that the effort of using a higher time complexity approach may not be worth. For higher CCRs up to 1.0 the improvement is always below 11%, which means that the improvement is not very high in order to compensate the usage of metaheuristic algorithms. For illustrative purpose, in figure 3 we can see an example of the makespan obtained by HEFT, AS, TS and ACS algorithms, for the DAG represented in Figure 1. Task 12 in HEFT is delayed due to communication costs from task 7.

Table 1. SLR improvement observed with metaheuristic algorithms compared to HEFT

	N=10			N=20			N=30			N=40		
CCR	SA	TS	ACS	SA	TS	ACS	SA	TS	ACS	SA	TS	ACS
0.1	0.80%	0.60%	0.53%	1.64%	1.38%	0.62%	3.16%	2.94%	1.35%	4.07%	3.90%	1.79%
0.5	7.03%	5.70%	5.74%	7.09%	5.66%	4.04%	7.97%	6.50%	2.84%	7.93%	6.74%	2.88%
0.8	9.96%	6.91%	8.27%	10.0%	6.80%	6.45%	9.93%	6.49%	4.09%	9.48%	6.61%	1.87%
1.0	10.0%	6.23%	7.74%	10.2%	5.65%	6.14%	9.45%	5.85%	2.98%	10.9%	6.98%	2.51%

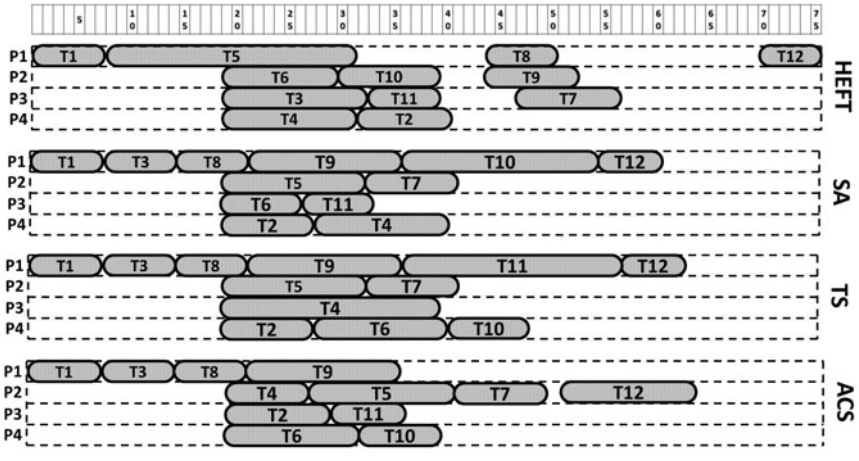


Fig. 3. Scheduling of task graph with HEFT, SA, TS, ACS

In conclusion, we can say that for low CCR (0.1) HEFT produces schedules competitive with metaheuristic approaches, with a lower time complexity. For higher CCRs up to 1, the improvement achieved with SA is below 11%, being

also competitive the schedules produced by HEFT. These results show also that new heuristic base algorithms have a narrow space of improvement over HEFT. Regarding the metaheuristic algorithms, SA showed to achieve consistently better scheduling solutions for DAG scheduling in heterogeneous systems.

References

1. Cerny, V.: Thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. Journal of Optimization Theory and Applications, 41–51 (1985)
2. Coffman, E.G.: Computer and job-shop scheduling theory. Wiley (1976)
3. Dhodhi, M.K., Ahmad, I., Yatama, A., et al.: An integrated technique for task matching and scheduling onto distributed heterogeneous computing system. Journal of Parallel and Distributed Computing 62(9), 1338–1361 (2002)
4. Dorigo, M., Gambardella, L.M.: Ant colony system: A cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation 1(1), 53–66 (1997)
5. Dorigo, M., Di Caro, G., Gambardella, L.M.: Ant algorithms for discrete optimization. Artifitial Life 5, 137–172 (1999)
6. El-Rewini, H., Lewis, T.G.: Scheduling parallel program tasks onto arbitrary target machines. Journal of Parallel and Distributed Computing 9(2), 138–153 (1990)
7. Glover, F.: Tabu search-part i. ORSA Journal on Computing 1(3), 190–206 (1989)
8. Glover, F.: Tabu search-part ii. ORSA Journal on Computing 2(1), 4–32 (1990)
9. Kim, D., Yi, B.-G.: A two-pass scheduling algorithm for parallel programs. Parallel Computing 20, 869–885 (1994)
10. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
11. Kohler, W.H., Steiglitz, K.: Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. Journal of ACM 2, 140–156 (1974)
12. Kwok, Y., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999)
13. Kwok, Y.-K., Ahmad, I.: Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Transactions on Parallel and Distributed Systems 7(5), 506–521 (1996)
14. Liou, J.-C., Palis, M.A.: A comparison of general approaches to multiprocessor scheduling. In: International Parallel Processing Symposium, pp. 152–156 (1997)
15. Papadimitriou, C., Yannakakis, M.: Scheduling interval ordered tasks. SIAM Journal of Computing 5, 73–82 (1976)
16. DAG Generation Program (2010), `http://www.loria.fr/~suter/dags.html`
17. Sinnen, O., Sousa, L.: List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Computing 30, 81–101 (2004)
18. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)