Abstract- Multiprocessor task scheduling is a well known NP-hard problem and numerous methods have been proposed to optimally solve it. The objective is makespan minimization, i.e. we want the last task to complete as early as possible. Simulated Annealing (SA) has been considered a very good tool for complex nonlinear optimization problem, such as multiprocessor task scheduling. However, a major disadvantage of the technique is that it is extremely slow. List-based scheduling algorithms are regarded as having acceptable results. In this paper we use a list scheduling based algorithm to find an initial solution and in the neighborhood generation phase of simulated annealing. We also parameterize SA and use a modified version of it. Simulation results show that our approach significantly improves the initial solution in considerably low time for different number of tasks; i.e. it efficiently outperforms the used list based scheduling approach.

Keywords: List based scheduling algorithms, modified simulated annealing, NP hard problems, task scheduling problem.

I. INTRODUCTION

Multiprocessor task scheduling is well known as one of the hardest combinatorial optimization problems[1]. The problem of scheduling of tasks in multiprocessor systems is to determine when and on which processor a given task executes[2].

It was Kirkpatrick et al[3] who first proposed Simulated Annealing, SA, as a method for solving combinatorial optimization problems. It is reported that SA is very useful for several types of such problems[4]. SA is a global optimization technique which traverses the search space by testing random mutations on an individual solution. A mutation that increases fitness is always accepted. A mutation which lowers fitness is accepted probabilistically[2]. SA, single or in hybrid methods, has been used in multiprocessor task scheduling in some approaches [2, 5-10].

The most remarkable disadvantages of simulated annealing is it needs a lot of time to find the optimum solution and it is very difficult to determine the proper cooling schedule.

There are two approaches to shorten the calculation time in SA. One is determining the cooling schedule properly. The other approach is to perform SA on parallel computers[11].

A comprehensive study of best practices of simulated annealing for task mapping is presented in[12]. In this book chapter, cooling schedule of SA for the task scheduling problem is analyzed. The authors show that SA is a well performing algorithm if used properly. We use some methods of parameter selections of it. In addition, we apply some more improvements, as come in the following, to optimally solve the problem.

List based scheduling algorithms are generally regarded as having a good cost performance trade-off because of their low cost and acceptable results[13-15]. They assign priority levels to the tasks and map the highest priority task to the best fitting processing element. In our new approach, we use a modified version of SA to further improve the result, obtained by list based scheduling.

We apply the list based scheduling mechanism to generate an initial solution, and in the neighborhood generation phase of SA to have a more informed search.

We are also inspired by the concepts in parallel simulated annealing[16], and use a concept, that we call agent. Simulation results show that this concept can also efficiently improve our proposed method, in terms of cost (makespan), and time, even while our approach is run on one single computer.

To sum up, we show that the combination of list scheduling based mechanism and a modified version of SA that we used can provide us a very good solution, in considerably low time; so can be used for scheduling of applications with large number of tasks.

The remainder of this paper is organized as follows. Section II states the problem. In section III, we introduce SA. Section IV describes our proposed task scheduling algorithms. Section V presents the experiment results. Then in Section VI, we conclude the paper.

II. PROBLEM STATEMENT

A homogeneous multiprocessor system is composed of a set $P=\{p_1…p_m\}$ of $M$ identical processors. They are connected by a fully connected communication network.
where all links are identical. Task preemption is not allowed.

The application program is modelled as directed acyclic graph (DAG), \( G(T, E) \) where \( T = \{ T_i : i = 1,\ldots,n \} \) is a set of \( N \) tasks and \( E \) is a set of directed edges among the tasks representing the precedence.

For any two tasks, \( i, j \in T, i < j \) means task \( j \) cannot be scheduled until \( i \) has been completed, \( i \) is a predecessor of \( j \) and \( j \) is a successor of \( i \). Weights associated with nodes represent the computation time and weights associated with edges represent the communication cost. The multiprocessor scheduling is to assign the set of tasks \( T \) onto the set of processors \( P \) in such a way that precedence constraints are maintained and to determine the start and finish time of each task with the objective to minimize the completion time. We assume that the communication system is content free and allows the overlap of computation with communication. Task execution is started only after all data have been received from its predecessor. The communication constraints are maintained.

The pseudo code of our method is shown in Fig. 1.

![Figure 1. Pseudo-code of the proposed method](image)

Before explaining the approach we describe three functions used in the pseudo code: \( cost, \) \( after \) and \( feasible \).

\( after(i) \) returns the task numbers which are scheduled on the same processor as \( i \) in the current scheduling, and whose start time is after or equal to the finish time of the task \( i \), in an ascending order.

\( feasible(S) \) gets the scheduling \( S \) and verifies if it is a feasible one, i.e., it conforms to the dependency relations between tasks in the DAG. \( Cost(S) \) returns the complementation time of the scheduling \( S \).

We produce an initial solution by applying the list scheduling technique, which consists in the following steps: a) determine the available tasks to schedule b) define a priority to them and c) until all tasks are scheduled, select the task with higher priority and assign it to the processor that allows the Earliest Start-time.

Examples of heuristics for assigning priority are critical path [20, 21], job length [22]. The Critical-Path-First, (CPF) method tries to select tasks on the critical path prior to other tasks. The flavor of this method can be seen in [23-25].
V. SIMULATION RESULTS

We implemented our approach in Borland C++ 5.02, using a system with 2 GBs of RAM and the CPU 2.2 GHz.

We defined a parameter $\alpha$ as equation (2):

$$\alpha = \frac{\sum_{i=1}^{N} \text{computationTime(Task } i)}{M}$$

Where $\text{computationTime(Task)}$ returns the time needed to execute this task, and $M$ is the number of processors.

We generated DAGs with random structures with the different number of nodes (tasks). We assigned weights to the nodes (the computation time) in such a way that $\alpha = 50$. We also made random numbers between 0 and 5 to assign weights to the edges (the communication cost). These constraints enabled us to have a more exact compare.

Since we applied CPFES, as the initial solution and applied the modified SA on it, we considerably improved the initial solution, which itself is regarded to be an acceptable one [13-15, 23-25].

There are three main parameters in our approach, $M$, $N$, $A$, that denote the number of processors, tasks and agents respectively.

In the first step we wanted to study how our approach works when the number of tasks increases. So we fixed $A$ and $M$ to 5 and 15 respectively. We produced three DAGs (with the specifications mentioned in the first paragraph of this section) with 100, 150 and 200 nodes (tasks), respectively. Fig. 2 and TABLE II show our approach produces very good improvements in considerably low time, for different number of tasks.

![Figure 2](image)

TABLE II summarizes the results shown in Fig. 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>$T_0 \times (0.95)^i \frac{C_{max}}{C_{opt}}$</td>
<td>Return temperature $T_i$ at iteration $i$</td>
</tr>
<tr>
<td>prob ($\Delta C$, $T$)</td>
<td>$\frac{1}{1 + \exp (-\frac{\Delta C}{C_{opt}})}$</td>
<td>The acceptance probability function</td>
</tr>
<tr>
<td>$T_0$</td>
<td>$\frac{\Delta C_{max}}{\ln \left( \frac{1}{0.45} \right)}$</td>
<td>The initial temperature</td>
</tr>
<tr>
<td>$R_{max}$</td>
<td>$M(N-1)(M)$ (denotes the number of processors)</td>
<td>Maximum Number of consecutive rejected moves</td>
</tr>
</tbody>
</table>
TABLE II. Simulation results, A=5, M=15.

<table>
<thead>
<tr>
<th>N</th>
<th>Relative improvement of initial solution (CPFES)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>39.7 %</td>
<td>4S</td>
</tr>
<tr>
<td>150</td>
<td>36.4%</td>
<td>6S</td>
</tr>
<tr>
<td>200</td>
<td>31.9%</td>
<td>8S</td>
</tr>
</tbody>
</table>

In the second step, we wanted to study the role of the new concept, agent, what we used. So we produced a DAG (with the specifications mentioned in the first paragraph of this section) with 150 nodes. We fixed M to 15. We applied our approach to the DAG three times and set the parameter A to 1, 5 and 10, respectively. When A is set to 1, it is like we did not use the concept. Fig. 3 shows when the number of agents increases our approach improves in terms of simulation time and cost.

TABLE III summarizes the results shown in Fig.3.

TABLE III. Simulation results, N=150, M=15.

<table>
<thead>
<tr>
<th>A</th>
<th>Relative improvement of initial solution (CPFES)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20 %</td>
<td>11S</td>
</tr>
<tr>
<td>5</td>
<td>36.4%</td>
<td>8S</td>
</tr>
<tr>
<td>10</td>
<td>39.8%</td>
<td>4S</td>
</tr>
</tbody>
</table>

In the third step, we wanted to study how our approach works when the number of processors varies. So, we used the DAG generated in the second step and fixed the parameter A to 5. We applied our approach to the DAG three times and set the parameter M to 10, 15 and 20, respectively. Simulation results show when the number of processors increases, the simulation time does not increase, and the cost decreases.

TABLE IV summarizes the results shown in Fig. 4.

TABLE IV. Simulation results, A=5, N=150.

<table>
<thead>
<tr>
<th>M</th>
<th>Relative improvement of initial solution (CPFES)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>34.1 %</td>
<td>8S</td>
</tr>
<tr>
<td>15</td>
<td>36.4%</td>
<td>6S</td>
</tr>
<tr>
<td>20</td>
<td>38%</td>
<td>6S</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

In this paper we used list based scheduling with the critical path priority that schedules each task on a processor in a way that minimizes the start time, as an initial solution of SA. This method produces an acceptable result, which we further improved by a modified version of SA. Based on a previously proposed method, we parameterized SA, to make algorithm scalable with respect to application and platform size. In the neighbor generation phase of SA, we again applied the list based scheduling algorithm to some tasks, to have a more informed search.

Being inspired by the concepts in parallel simulated annealing we used a concept that we called ‘agent’. This concept also improved our results, in the term of cost and time.

Simulation results showed that our proposed method produced very good solutions, in considerably low time, even while running on the DAGs with a large number of tasks.

REFERENCES


