An Improved Genetic Algorithm For Solving The Multiprocessor Scheduling Problem

1Imad Fakhri Al Shaikhli, 2Ismail Khalil

1Dept. of Computer Science Kict- Iium Kuala Lumpur-Malaysia.
2Institute of Telecooperation Johannes Kepler University Linz.

Abstract: Multiprocessor Scheduling Problem (MSP) is an NP-complete optimization problem. The applications of this problem are numerous, but are, as suggested by the name of the problem, most strongly associated with the scheduling of computational tasks in a multiprocessor environment. Many methods and algorithms were suggested to solve this problem due to its importance. Genetic algorithms were among the suggested methods. In this research, sound improvements were done on one of the known papers (Hou, E.S.H., N. Ansari and H. Ren, 1994). Results show very good improvements in increasing the percentage of getting the exact solution as well as decreasing the number of generations needed to converge.

Key words: multiprocessor scheduling problem, genetic algorithms, list scheduler, optimization.

INTRODUCTION

The general problem of multiprocessor scheduling can be stated as scheduling a set of partially ordered computational tasks onto a multiprocessor system so that a set of performance criteria will be optimized. The difficulty of the problem is characterized by:

a. The topology of the task graph representing the precedence relations among the computational tasks.
b. The topology of the multiprocessor system.
c. The number of parallel processors.
d. The uniformity of the task processing time.
e. The performance criteria chosen.

In general, the MSP is computationally intractable even under simplified assumptions (Gorey, M.R. and D.S. Johnson, 1979; Stone, H.S., 1977). Because of its computational complexity, heuristic algorithms have been proposed to obtain suboptimal solutions to various scheduling problems (Kasahara, P. and S. Narita, 1985; Chen, C.L., C.S.G. Lee and E.H. Hou, 1988; Hellstrom, B. and L. Kanal, 1992). Most of the existing techniques are based on list scheduling (Wang, Q. and K.H. Cheng, 1991; Tao Yang and Apostolos Gerasoulis, 1993). In list scheduling, each task is assigned a priority and whenever a processor becomes available, a task with the highest priority is selected from the list and assigned to the processor. The objective of the present work was to do analyze one the known genetic algorithms (Hou, E.S.H., N. Ansari and H. Ren, 1994) that was proposed for solving the multiprocessor scheduling problem by applying it on a common test bed so as to be able to study its performance, in order to improve as a result of the insights gained. In order to formalize the MSP, first we define a homogeneous multiprocessor system and a parallel program. A homogeneous multiprocessor system composed of a set P

P = p1, p2, pm of m identical processors, each processor can execute at most one task at a time and task preemption is not allowed.

The parallel program is described by an acyclic graph D = (T, A). The vertices represent the set T = (t1, t2, ..., tn) of tasks and each arc represents the precedence relation between two tasks. An arc (ti, tj) ∈ A represents the fact that tj can start execution only after the completion of the execution of ti. A path is a sequence of nodes, (t1, t2, ..., tk), 1 < k < n, such that ti is an immediate predecessor of tj, 1 < i < k.

To every task tj, there is an associated value representing its duration and we assume that these durations are known before the execution of the program.

Hence a schedule S is a vector:

S = {S1, S2, ..., Sn} where Sn = {tn1, tn2, ..., tnn}, i.e Sn is the set of the ni tasks scheduled to Pn.

For each task tn, 1 represents its execution rank in Pn under the schedule S. Further, for each task tn we denote P(tn, S) and r(tn, S) respectively, the processor and the rank in this processor of tn under the schedule S. The execution time yielded by a schedule is called make span.

List Scheduling:

In priority list scheduling, initially a list of tasks is ordered according to some given priority. At each scheduling step, the list scheduling heuristic schedules the highest priority task on the first idle processor. A generic procedure of list scheduling is described below:

a. priority -list = (n1, n2, ..., nn) sorted in a descending order of task priority;
b. clock = 0;
c. while(priority-list is not empty) do remove the leftmost free task (t j);
d. schedule t j to an available idle processor;
e. end[while] clock = the next earliest time when a processor becomes available;

What is interesting about list scheduling is that the performance is within 50% of the optimum independent of the priority list as shown by Graham et al. An important question is what choices of priority-lists will consistently give schedules that are "close" to the optimum schedule?. Experimentally, Adam, Charly and Dickson (1974) answered the above question by conducting an extensive empirical performance study of the Critical Path (CP) algorithm with other four priority list-scheduling algorithms. Their conclusion is that the CP heuristic is superior to other algorithms since it is near-optimal (i.e. within 5% of the optimal in 90% of random cases).

**Genetic Algorithm for solving the MSP problem (Hou, E.S.H., N. Ansari and H. Ren, 1994):**

There are many genetic algorithms for solving the MSP (Ahmad, I. and M.K. Dhodhi, 1996; Bauer, T., et al., 1995; Annie, S., et al., 2004; Shuang Zhou, E., Yong Liu, Di Jiang, 2006), one of them was (Hou, E.S.H., N. Ansari and H. Ren, 1994), which will be denoted in this rest of this article as (HAR). This algorithm will be analysed and discussed through the five main components of GA.

a. Representation: They represent the schedule as a list of computational tasks executed on number of processors, the order of the tasks in each list indicates the order of execution.

b. Initial Population: for generating the initial population the following steps were taken:

    Compute the height for each task in the task graph, where : height

    \[
    Ti = \begin{cases} 
    0 & \text{if } \text{Pred}(Ti) = \emptyset \\
    (1 + \max Tj \in \text{Pred}(Ti) \cdot \text{height}(Tj)) & \text{otherwise}
    \end{cases}
    \]

    Partition the tasks in the task graph into different sets \( G(h) \), \( G(h) \) is defined as the set of tasks with height \( h \), according to the value of the height.

    Each of the first \( p-1 \) processors, do the next step.

        Each set \( G(h) \):

        - Set \( NG(h) \) to the number of tasks in \( G(h) \).
        - Generate a number \( r \) (randomly) between 0 and \( NG(h) \).
        - Pick \( r \) tasks from \( G(h) \) and assign them to the current processor.
        - Assign the remaining tasks in the sets to the last processor.

c. Fitness Function (FF), \( \text{FF}(S) = C_{max} - \text{FT}(S) \), where \( C_{max} \) is the maximum finishing time observed so far(up to the previous generation),

    \( \text{FT}(S) = \max Pj \cdot \text{ftp}(Pj) \), where \( \text{ftp}(Pj) \) is the finishing time for the last task in processor \( Pj \).

d. Genetic Operators

    Crossover: They used the standard one point crossover but the important thing here is the selection of the crossover site(position where we can do the crossover), which have been chosen so that it satisfies the following conditions :

    - The height of the tasks next to the crossover site is different.
    - The height of the tasks immediately in front of the crossover site is the same.

    Mutation for a schedule \( S \) does the following: pick (randomly) a task \( Ti \). _ search \( S \) for a task \( Tj \) with the same height. Form a new schedule by exchanging the two tasks \( Ti, Tj \) in \( S \).

e. Parameters; they have used two sets of parameters : \( \text{POPSIZE} = 10, 20 \), \( \text{Crossover probability} = 1.0, 0.5 \), Mutation probability: = 0.05, 0.005, Number of generations = 1500, 2000

**The Test Bed:**

We have used DAGs (Directed Acyclic Graphs) of sizes (number of tasks) = 12, 16, 20, 24, 28, 32, 40 and 48, with number of processors = 2,3,4,5,6,8, 10 and 12 (higher number of processors for DAGs with bigger sizes), with number of edges in each DAG = same, double, triple and 4 \( * \) (the number of tasks). In all, we have 92 DAG’s.

We followed the method of (Yu-Kwong, Kwok and I. Ahmad) for generating these DAG’s: Suppose that the optimal schedule length of a graph and the number of processors used are specified as \( SL_{opt} \) and \( P \) respectively. Then for each processing element \( i \), we randomly generate a number \( x_i \) from a uniform distribution with mean \( v/p \) (\( v \) the number of tasks). The time interval between 0 and \( SL_{opt} \) of the processing element \( i \) is then randomly partitioned into \( xi \) sections.

Each section represents the execution time for one task. Thus, \( x_i \) tasks are scheduled to processor element \( i \) with no idle time slot. In this manner, \( v \) tasks are generated so that every processor has the same schedule length.
To generate an edge, two tasks \( n_a \) and \( n_b \) are randomly chosen such that 
\[ FT(n_a) < ST(n_b) \] 
(F\T stand for the finishing time and \( ST \) for the starting time). 
The edge is made to emerge from \( n_a \) to \( n_b \).

**Parametric Study:**
As part of the analysis of this algorithm, we will follow the method proposed in (Taha, Imad Fakhri, 2011). 
To study the performance of the algorithm, a parametric study was done to know the suitable set of parameters 
by running the algorithm on the same DAG and changing the probabilities of crossover and mutation as follows:
prob. of crossover from the set 0.4, 0.5, 0.6, 0.7, 0.8 and 0.9 prob. of mutation from the set 0.00121, 0.00156,
0.00175, 0.0026, 0.0034, 0.006, 0.01, 0.16, 0.2, 0.25. The result is shown in table 1:

<table>
<thead>
<tr>
<th>Prob. of crossover</th>
<th>Prob. of mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.73</td>
<td>0.3</td>
</tr>
</tbody>
</table>

**Experimental Results Before Improvements:**
After developing the algorithm and generating the DAGs with known optimal schedule length, experiment 
was done by executing the algorithm three times on each DAG and then by taking the best out of the three, the 
following parameters were fixed throughout the experiment: number of generation (NOG) = 100 
population size (POPSIZE) = number of tasks in the DAG 
probability of crossover for HAR = 0.73 
probability of mutation for HAR = 0.2 
The results are shown in table 2.

<table>
<thead>
<tr>
<th>No. of tasks</th>
<th>% of exact solution</th>
<th>No. of generations</th>
<th>% of Deviation</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>11</td>
<td>29.7</td>
<td>17</td>
<td>6.1</td>
</tr>
<tr>
<td>16</td>
<td>27</td>
<td>28</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
<td>45.18</td>
<td>22</td>
<td>44.5</td>
</tr>
<tr>
<td>24</td>
<td>27</td>
<td>47.27</td>
<td>25</td>
<td>69.6</td>
</tr>
<tr>
<td>28</td>
<td>9</td>
<td>53.81</td>
<td>16</td>
<td>135.5</td>
</tr>
<tr>
<td>32</td>
<td>18</td>
<td>59.54</td>
<td>14</td>
<td>163</td>
</tr>
<tr>
<td>40</td>
<td>30</td>
<td>55.5</td>
<td>4</td>
<td>307</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>70.6</td>
<td>3</td>
<td>361.8</td>
</tr>
<tr>
<td>Avg.</td>
<td>20</td>
<td>48.7</td>
<td>15</td>
<td>138.4</td>
</tr>
</tbody>
</table>

From the results in table 2 we can notice the following:
a. HAR failed to find the exact solution in 80% of the cases.
b. The deviation from the exact solution was 0.15. The most important question now is: why this 
unsatisfactory performance of HAR though it is not a blind search?
To answer this question we used the analysis method proposed in (Taha, Imad Fakhri, 2011) and found the 
following points:
- It cannot capture the whole search space (Bauer, T., et al., 1995) because it is possible to find legal 
schedules that violates the height condition but still maintain the interprocessor precedence relations and it is 
therefore, unfortunately, possible that a schedule that violates the height condition could be an optimal schedule.
- The initialization procedure is biased because the probability for a processor with higher index to have tasks 
assigned to it is lesser than the probability for a processor with lower index. This means that the distribution of 
the tasks among the processors is not uniform and that’s why when we have higher number of processors, the 
performance will decays as some of the processors might not get any task assigned to it. We have seen this when 
we tried to execute HAR on DAG’s of 32,40 and 48 tasks with number of processors 8,10 and 12. There were 
some problems because many processors will be empty because of the initialization.
- We have noticed that when the differences between the execution time of some tasks in a DAG is high, then 
HAR fails to get even a good solution because the difference between the execution time of the processors will 
be high while in the real case the execution time of the processors should be (usually) comparable.

**The Proposed Algorithm:**
After analyzing HAR using the method of (Taha, Imad Fakhri, 2011) and studying the drawbacks, the 
following improvements will be suggested:
a. New calculation of the height (Bauer, T., et al., 1995): To overcome the problem of not capturing some of 
the potential solutions which violate the height condition but still maintains the precedence relation, we used the 
following equation for calculating the height :
b. Using initialization of (Ahmad, I. and M.K. Dhodhi, 1996): Here they use a heuristic algorithm for the initialization which depends on the level of the tasks and that it gives a good initial population. So we tried to use this initialization after changing it to suit HAR as follows:

- Call the same initialize from (Ahmad, I. and M.K. Dhodhi, 1996) to generate the initial population.
- Call the list scheduler used in (Ahmad, I. and M.K. Dhodhi, 1996) for calculating the execution time for each schedule.
- Use the output (schedules) of the list scheduler above as an initial population for HAR.

c. Self-fixer operator: To overcome the problem mentioned above which happens when there were big gaps between the execution time of the processors, we have introduced a new operator which can adjust the schedule length by doing the following: Firstly the operator will work according to a probability Ps. after choosing a schedule (chromosome) we will do the following:

- choose the processor pi with the highest execution time and then
- choose the task Ti with the highest execution time within this processor.
- choose a task Tj randomly from the processor pi (other than Ti).
- Choose a processor pj randomly (other than pi).
- Assign Tj to pj (put it in a proper sequence).
- Remove Tj from pi

**Experimental Results After Improvement:**

After doing the improvements mentioned above, we did the same experiments again on the same DAG’s and results are shown in the table below:

<table>
<thead>
<tr>
<th>No. of tasks</th>
<th>% of exact solution</th>
<th>No. of generations</th>
<th>% of Deviation</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>77</td>
<td>1.6</td>
<td>0.5</td>
<td>3.3</td>
</tr>
<tr>
<td>16</td>
<td>54</td>
<td>23.45</td>
<td>1</td>
<td>16.8</td>
</tr>
<tr>
<td>20</td>
<td>45</td>
<td>19.18</td>
<td>0.6</td>
<td>39</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
<td>12.18</td>
<td>1</td>
<td>53.3</td>
</tr>
<tr>
<td>28</td>
<td>36</td>
<td>27.72</td>
<td>1</td>
<td>97.6</td>
</tr>
<tr>
<td>32</td>
<td>36</td>
<td>30.38</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>40</td>
<td>53</td>
<td>25</td>
<td>1</td>
<td>284.6</td>
</tr>
<tr>
<td>48</td>
<td>38</td>
<td>13.3</td>
<td>1</td>
<td>417.4</td>
</tr>
<tr>
<td>avg</td>
<td>45</td>
<td>19.1</td>
<td>1</td>
<td>150.25</td>
</tr>
</tbody>
</table>

From the above table we can see that the performance of the improved HAR is now much better than the old HAR.

**Conclusions:**

A careful analysis were done on the genetic algorithm of (Hou, E.S.H., N. Ansari and H. Ren, 1994) using the method of (Taha, Imad Fakhri, 2011) to have insight of the algorithm and to find the cases and circumstances where the algorithm fail. Then, major improvements were suggested base on the analysis done. Experimental results show that the improved algorithm outperforms the original one where the percentage of getting the exact solution increased from 20% to 45% and the number of generation needed to find the exact solution decreased by 40%.

**REFERENCES**


