List-Scheduling Techniques in Homogeneous Multiprocessor Environments: A Survey

Hamid Reza Boveiri
Sama Technical and Vocational Training College, Islamic Azad University, Shoushtar Branch, Shoushtar, Iran
boveiri@{shoushtar-samacollege.ir or ymail.com}

Abstract

Optimized task scheduling is one the most important factors to achieve high-performance in multiprocessor environments such as parallel and distributed systems. In such architectures, each program is decomposed into the smaller dependent segments so-called tasks. To formulate the problem, execution times of tasks, precedence constrains and communication costs among them are modeled using a directed acyclic graph (DAG) named task graph. The goal is to minimize the program completion-time (makespan) by means of mapping the tasks to the identical processors in such a way that precedence constrains are preserved. This problem is shown to be NP-hard in general form, and hence, a number of heuristic approaches to solve it have been introduced. A large number of proposed approaches in the literature use list-scheduling technique in which a list of tasks is created based on some priority measurements, and then in each step, the most priority task in the list is selected to schedule on the processor that allows the earliest start time (EST) until all tasks are scheduled. In this paper, we survey five different list-scheduling approaches namely HLFET, ISH, MCP, ETF and DLS from the different points of view, and describe the strategies and philosophies behind them. In addition, a comprehensive set of experiments and evaluations has been done, and different results and conclusions have been presented.

Keywords: Parallel and distributed systems, list-scheduling, multiprocessor task-graph scheduling, task priority measurements

1. Introduction

Nowadays, as a consequence of increase in time-complexity of programs and decrease in hardware costs, utilization of multiprocessor environments such as grid, cloud, parallel and distributed system has been extensively increased. In such architectures, each program is decomposed into the smaller segments named tasks. Tasks are not necessarily independent; some of them need data produced by the others as input to start execution, and hence, there are precedence constraints among tasks. After execution, each task stores the output data in the memory of the processor on which is executed. If a new task which needs these data is executed on this processor, it can access the data instantly without wasting any time, unless it has to wait for a while until the data can be ready in the memory of current processor. This transmission-delay is known as communication cost.

In static scheduling, task execution-times, precedence constraints and communication costs of a parallel program are determined during compile step. To formulate the problem each parallel-program is modeled using a directed acyclic graph (DAG) so-called task-graph. Tasks in the task-graph must be assigned to the identical processors in such a way that precedence constraints are preserved and the program completion-time (makespan) is minimized. This problem is shown to be NP-hard in general form and some restricted ones [1, 2] and [3], and hence, different heuristic approaches have been introduced to solve it with different strategies and mechanisms.
A number of proposed approaches so-called BNP (Bounded Number of Processors) methods like HLFE 
T \cite{4}, ISHT \cite{5}, MCP \cite{6}, ETF \cite{7} and DLS \cite{8} use list-scheduling technique. They make a list of ready-tasks at each step, and assign them some priorities. Then, repeatedly the most priority task in the ready-list is selected to schedule on the processor that allows the earliest start time (EST), and this process continues until all tasks are scheduled.

Since different methods utilize different priority measurements and strategies, this study has been done to answer the following fundamental questions: which philosophy is behind these methods, and what are their time complexities? How effective are they? How sensitive are they to various parameters and measures? What is their ranking from the performance point of view on a uniform basis?

The rest of the paper is organized as follows. Next section studies multiprocessor task scheduling problem. Section III describes list-scheduling technique and the related approaches. Implementation and experimental details are given in the section IV. Section V presents achieved results and comparisons, and finally the paper concludes in the last section.

2. Multiprocessor Task Scheduling

A directed acyclic graph $G = (N, E, W, C)$ named task-graph is used to formulate a parallel programs, where $N = \{n_1, n_2, ..., n_n\}$, $E = \{(n_i, n_j) | n_i, n_j \in N\}$ $W = \{w_1, w_2, ..., w_w\}$, $C = \{c(n_i, n_j) | (n_i, n_j) \in E\}$, and $n$ are set of nodes, set of edges, set of weight of nodes, set of weight of edges, and the number of nodes in the task-graph respectively.

Figure 1 shows task-graph of a real-world application composed of nine tasks. In this graph, nodes are tasks and edges specify precedence constraints among them. Edge $(n_i, n_j) \in E$ demonstrate that task $n_i$ must be finished before the starting of task $n_j$. In this case, $n_i$ is called parent and $n_j$ is called a chilled. Nodes without any parents and nodes without any children are called entry-nodes and exit-nodes respectively. Each node weight $w_i$ is the necessary execution time of task $n_i$, and each weight of edge like $c(n_i, n_j)$ is the required time for data transmission from task $n_i$ to task $n_j$ identified as communication cost. If both tasks are executed on the same processor, communication cost will be zero between them. Else, after finishing task $n_i$, task $n_j$ has to wait as $c(n_i, n_j)$ before starting in order to the input data become ready.

In static scheduling, tasks execution times, precedence constraints among tasks and communication costs are generated during the program compile stage. Tasks must be mapped into the given $m$ identical processor elements (homogeneous environment) with respect to their precedence so that the overall finish-time of the tasks (program completion-time or makespan) is minimized.

3. List-Scheduling Technique

Most scheduling algorithms are based on the so-called list-scheduling technique. The basic idea behind list-scheduling is to make a sequence of nodes as a list for scheduling by assigning them some priorities, and then repeatedly remove the highest priority node from the scheduling list, and allocate the node to which processor that allows the earliest-start-time (EST) until all nodes in the graph are scheduled. If all predecessors (parents) of task $n_i$ have been executed on the same processor like $p_i$, $EST(n_i, p_i)$ will be $Avail(p_i)$ that is the earliest time that $p_i$ is available to execute the next task, else the earliest-start-time of task $n_i$ on processor $p_j$ is computed by using

$$EST (n_i, p_j) = \max \left\{ Avail (p_j), \max_{n_m \in \text{Parents}(n_i)} (FT(n_m) + c(n_m, n_i)) \right\}$$

(1)

where $FT(n_m) = EST(n_m) + w_m$ is actual finish-time of task $n_m$, and $Parents(n_i)$ is the set of all parents of $n_i$. Finally, total finish-time of the program (makespan) is calculated using
\[ \text{makespan} = \max_{i=1}^{n} \{ FT(n_i) \} \]  

(2)

To open up the issue, list scheduling of the task-graph in fig. 1 is presented as follows. Topological number of tasks has been considered as priority measurement (e.g. \( n_1 \) is more priority task between \( n_1 \) and \( n_2 \)). In the first step, the ready-list has only one task to schedule, the only existing entry-node \( \{ n_1 \} \). After scheduling \( n_1 \), its children ready-to-execute-now are inserted to the ready-list \( \{ n_3, n_4, n_5 \} \). Among them, \( n_2 \) has the most priority based on the task topological number, so it is scheduled earlier and the ready-list becomes \( \{ n_3, n_4, n_5, n_6 \} \) (since \( n_6 \) is a child of \( n_2 \) which can be scheduled now, it was inserted to the ready-list). These operations continue until scheduling of all tasks in the task graph, which result in the following task order \( \{ n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9 \} \).

Some frequently used attributes to assign priority to the tasks are \( TLevel \) (Top-Level), \( BLevel \) (Bottom-Level), \( SLevel \) (Static-Level) and \( ALAP \) (As-Late-As-Possible). The \( TLevel \) or \( ASAP \) (As-Soon-As-Possible) of a node \( n_i \) is the length of the longest path from an entry-node to the \( n_i \) excluding \( n_i \) itself, where the length of a path is the sum of all nodes and edges weight along the path. The \( TLevel \) of each node in the task graph can be computed by traversing the graph in the topological order using

\[ TLevel(n_i) = \max_{j \in Parents(n_i)} \left( TLevel(n_j) + c(n_j, n_i) \right) \]  

(3)

The \( BLevel \) of a node \( n_i \) is the length of the longest path from \( n_i \) to an exit node. It can be computed for each task by traversing the graph in the reversed topological order as follows

\[ BLevel(n_i) = \max_{j \in Children(n_i)} \left( BLevel(n_j) + c(n_j, n_i) \right) + w_j \]  

(4)

where \( Children(n_i) \) is set of all children of \( n_i \).

If the edges weights are not considered in the computation of \( BLevel \), a new attribute called Static-Level or simply \( SLevel \) can be generated using (5).

---

**Figure 1. Task Graph of a Program with Nine Tasks**

[9]
Table I. \textit{HL, TL, BL, SL, ALAP, and NOO of Each Node in the Task Graph of Fig. 1}

<table>
<thead>
<tr>
<th>Node \text{(n_i)}</th>
<th>\text{HL}</th>
<th>\text{TL}</th>
<th>\text{BL}</th>
<th>\text{SL}</th>
<th>\text{ALAP}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{(n_1)}</td>
<td>0</td>
<td>0</td>
<td>37</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>\text{(n_2)}</td>
<td>1</td>
<td>6</td>
<td>23</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>\text{(n_3)}</td>
<td>1</td>
<td>3</td>
<td>23</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>\text{(n_4)}</td>
<td>1</td>
<td>3</td>
<td>20</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>\text{(n_5)}</td>
<td>1</td>
<td>3</td>
<td>30</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>\text{(n_6)}</td>
<td>2</td>
<td>10</td>
<td>15</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>\text{(n_7)}</td>
<td>2</td>
<td>22</td>
<td>15</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>\text{(n_8)}</td>
<td>2</td>
<td>18</td>
<td>15</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>\text{(n_9)}</td>
<td>3</td>
<td>36</td>
<td>1</td>
<td>1</td>
<td>36</td>
</tr>
</tbody>
</table>

\[ SLevel \ (n_i) = \max_{j \in \text{Children} \ (n_i)} \left( SLevel \ (n_j) + w_j \right) \]  \hspace{1cm} (5)

The \textit{ALAP} start-time of a node is a measure of how far the node's start-time can be delayed without increasing the overall schedule-length. It can be drawn for each node by traversing the graph in the reversed topological order using (6).

\[ ALAP \ (n_i) = \min_{j \in \text{Children} \ (n_i)} \left( CPL \cdot ALAP \ (n_j) - e(n_j, n_i) + w_i \right) \]  \hspace{1cm} (6)

where \textit{CPL} is the Critical-Path-Length, that is, the length of the longest path in the given task graph. Table I lists the above-mentioned measures for each node in the task graph of Figure 1.

To open up how these measures can be utilized in order to schedule the tasks of a task-graph, five well-known traditional list-scheduling algorithms namely HLFET, ISH, MCP, ETF and DLS are surveyed as follows.

3.1. The HLFET Algorithm

The HLFET (Highest Level First with Estimated Times) first calculates the \textit{SLevel} of each node in the task-graph. Then, make a ready-list in the descending order of \textit{SLevel}. At each instant, it schedules the first node in the ready-list to the processor that allows the earliest-execution-time (using the non-insertion approach) and then, updates the ready-list by inserting the new nodes ready now to execute, until all the nodes are scheduled. The time-complexity of the algorithm is \(O(n^2)\), where \(n\) is the number of tasks in the task-graph. Figure 2 (a) shows the scheduling Gantt chart of the graph of Figure 1 using HLFET algorithm on two processor elements.

3.2. The ISH Algorithm

The ISH (Insertion Scheduling Heuristic) algorithm uses the schedule-holes, the idle time slots, in the partial schedules. The algorithm tries to fill the holes by scheduling other nodes into them and use \textit{SLevel} as the priority measurement of a node. The time-complexity of the algorithm is \(O(n^2)\), and Figure 2 (b) shows scheduling Gantt chart of the graph of fig. 1 using ISH algorithm on two processor elements.

3.3. The MCP Algorithm

The MCP (Modified Critical Path) algorithm uses the \textit{ALAP} of the nodes as the priority. It first computes the \textit{ALAP} times of all the nodes, and then constructs a ready-list in the ascending order of \textit{ALAPs}. Ties are broken by considering the \textit{ALAP} times of the children of the nodes. The MCP algorithm then schedules the nodes in the list one by one to the
processor that allows the earliest-start-time using the insertion approach. The time-complexity of the algorithm is $O(n^3 \log n)$, and the scheduling Gantt chart of the graph of Figure 1 using MCP algorithm on two processor elements is shown by Figure 2 (c).

### 3.4. The ETF Algorithm

The ETF (Earliest Time First) algorithm computes the earliest-start-times for all the nodes in the ready-list by investigating the start-time of a node on all processors exhaustively. Then, it selects the node that has the smallest start-time for scheduling; ties are broken by selecting the node with the higher $SLevel$ priority. The time-complexity of the algorithm is $O(mn^2)$, where $m$ is the number of available processors. The scheduling Gantt chart of the graph of Figure 1 using EST algorithm on two processor elements is shown by Figure 2 (d).

### 3.5. The DLS Algorithm

The DLS (Dynamic Level Scheduling) algorithm uses an attribute called dynamic-level (or DL) that is the difference between the $SLevel$ of a node and its earliest-start-time on a processor. At each scheduling step, the DLS algorithm computes the DL for every node in the ready-list for all processors. The node-processor pair that gives the largest DL is selected to schedule, until all the nodes are scheduled. The algorithm tends to schedule nodes in a descending order of $SLevel$ at the beginning, but nodes in an ascending order of their $TLevel$ near the end of the scheduling process. The time-complexity of the algorithm is $O(mn^2)$, and Figure 2 (e) shows scheduling Gantt chart of the graph of Figure 1 using DLS algorithm on two processor elements.

---

**Figure 2. The Scheduling Gantt chart of the task graph of fig.1 using the four different traditional heuristics.** (a) The HLFET algorithm. (b) The ISH algorithm. (c) The MCP algorithm. (d) The ETF algorithm. (e) The DLS algorithm
4. Implementation and Experimental Details

The implementation was made on a Pentium IV (8-core 4.9GHz i7-3770k) desktop computer with Microsoft Windows XP (SP2) platform using Microsoft Visual Basic 6.0 as the programming language. For each experiment, mean of 10 times execution of the algorithm was considered to prevent the impertinence.

The utilized dataset for evaluating the algorithms was an enhanced version of dataset introduced in [10] composed of 125 random task graphs. The random task graphs have different shapes on three following parameters: Size ($n$): that is the number of tasks in the task graph. Five different values were considered {32, 64, 128, 256 and 512}.

- Communication-to-Computation Ratio (CCR): demonstrate how much a graph is communication or computation base. The weight of each node was randomly selected from uniform distribution with mean equal to the specified average computation cost that was 50 time-instance. The weight of each edge was also randomly selected from uniform distribution with mean equal to average-computation-cost $\times$ CCR. Five different values as CCR were selected {0.1, 0.5, 1.0, 5.0 and 10.0}. Selecting 0.1 makes computation intensive task-graphs. In contrast, selecting 5.0 makes the task-graph communication intensive.

- Parallelism: the parameter which determine the average number of children for each node in the task-graph. Increase in this parameter makes the graph more connected. Five different values of parallelism were chosen {3, 5, 10, 15 and 20}.

Whereas the archived makespan of these random graphs are in the wide range regarding their various parameters, $NSL$ (normalized schedule length), which is a normalized measure, is used. It can be calculated for every input task-graph by dividing the makespan to the lower bound defined as the sum of weights of the nodes on the original critical path using (7).

$$NSL = \frac{\text{Schedule} - \text{Length}}{\sum_{i \in CP} w_i}$$

where $CP$ is set of nodes on critical path (the longest path) of the given graph.

5. Results and Comparisons

The first set of experiments was done on entire random task-graphs. The number of processors was high-enough for each approach to produce its best scheduling. However, all the presented experiments were done again using only two processor elements for scheduling the task-graphs. The achieved results were about identical with the results presented in the paper, and certified the following conclusions. Figure 3 shows the average $NSL$ achieved by the introduced approaches regarding increase in the graph size from 32 to 512. Increase in the number of nodes monotonically results in higher $NSL$ for all approaches, nevertheless had no impact to change the individual ranking of each approaches. In average $NSL$, the performance ranking of the approaches is {ETF, MCP=DLS, ISH, HLFET}. It should be note that each ranking in this paper starts with the best approach and ends with the worst one with respect to the given comparison metric, that is, the ETF was the best, MCP and DLS were moderate and about identical (MCP was slightly better than DLS), and the ISH and HLFET were the worst ones.

The second set of experiments compares the related approaches from the increase in the communication-to-computation ratio (CCR) point of view. Figure 4 shows the results achieved by the each approach using entire random task-graphs with respect to the different graph CCR from 0.1 to 10. Graphs with $CCR = 0.1$ are highly computation intensive; In contrast, selecting 10.0 makes the graph communication intensive. With low CCRs, all the approaches had about identical performance, while growing the parameter made some approaches like HLFET and ISH inefficient; This phenomenon id as a
consequence of using $SLevel$ as the priority measurement of nodes, since $SLevel$ does not take into account the communications costs. DLS was slightly better, and MCP and ETF were about identical and the best to cope with such a this condition. Hence, the eventual $CCR$ ranking is as follows \{ETF$\approx$MCP, DLS, ISH, HLFET\}.

The next set of experiments compares the related approaches from the increase in the parallelism point of view (the average number of each node’s children). Fig. 4 shows the results achieved by each approach using entire random task-graphs with respect to the different graph parallelism from 3 to 20. Graphs with parallelism $= 3$ are highly sparse; In contrast, selecting 20 makes the graph highly connected. Surprisingly, increase in this parameter has no particular impact on the performance achieved by the approaches, and their rankings change in the random fashion. In average, ETF was the best and HLFET was the worst.

The last set of experiments was done on the entire random task-graphs and all different parameters aggregately. Figure 6 shows the results in average $NSL$. These results accompanied with the other ones obtained from the prior experiments eventually lead us to the final performance ranking that is \{ETF, MCP, DLS, ISH, HLFET\}. ETF is the best approach, MCP is slightly worse, DLS and ISH are moderate, and the HLFET is the worst.

**Figure 3.** The results in $NSL$ achieved by the different approaches using entire random task-graphs with respect to the different graph sizes from 32 to 512

**Figure 4.** The results in $NSL$ achieved by the different approaches using entire random task-graphs with respect to the different graph $CCR$ from 0.1 to 10
6. Conclusion

In this paper, we surveyed five different list-scheduling techniques named HLFET, ISH, MCP, ETF and DLS which are extensively used for task-graph scheduling in homogeneous multiprocessor environments. To evaluate the aforementioned approaches a dataset consisting of 125 random task-graphs with different shape parameters that are graph size, communication-to-computation ratio (CCR) and parallelism (the average number of children for each node in the task-graph) was utilized. The evaluation was made based on the performance that is the schedule length (makespan) achieved by each approach. Whereas the makespans archived from the random graphs are in the wide range regarding their various parameters, NSL (normalized schedule length), which is a normalized measure, was used in the evaluations and comparisons. Different sets of experiments were done using different shape parameters as the points of view, and the following conclusions were drawn: Although increase in the graph size (the average number of nodes in the task-graph) monotonically increases the achieved makespans, it did not change the overall performance ranking of the approaches. Next experiments showed sensibility of HLFET and ISH to the increase in the CCR parameter. It is rational because both use SLevel as the priority measurement; since SLevel do not take into account the communication costs among tasks. On the other hands, the ETF and MCP algorithms were the best to cope with such a condition. Last set of experiments showed that there is not any sensible relation between increase in the number of children for each node (parallelism) and the overall performance ranking of each approach. However, growing this parameter eventually leads to find slightly better solutions for each individual
algorithm. Considering all the experiments, we eventually leaded to the final performance ranking {ETF, MCP, DLS, ISH, HLFET}, that is, ETF was the best approach, MCP was slightly worse, DLS and ISH were moderate, and the HLFET was the worst.

References


Endnote

1 Highest Level First with Estimated Time
2 Insertion Scheduling Heuristic
3 Modified Critical Path
4 Earliest Time First
5 Dynamic Level Scheduling