Design and Analysis of Heuristic Functions for Static Task Distribution *

Behroz Shirmil
Ming-fang Wang

Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275

Abstract

Optimal task scheduling in a multiprocessor environment is known to be an NP-hard problem. Thus, the research efforts in this area have focused on heuristic methods for efficient distribution of tasks. This paper introduces two new static task scheduling algorithms. The first algorithm, called Heavy Node First (HNF), is very simple and based on a local analysis of program graph nodes at each level. The second method, called Weighted Length (WL), considers a more global view of the program graph, taking into account the relationship among the nodes at different levels. The two schemes are compared against the classical Critical Path Method (CPM). For a given Directed Acyclic Graph (DAG) of n nodes representing a program, it is shown that HNF requires \( O(n \log(n)) \) steps while WL and CPM require \( O(n^2) \) steps to accomplish the allocation. The non-trivial worst case performance of the three algorithms is analytically evaluated and their average case performance is evaluated through a simulation study. Considering the program execution time and the processor(s) idle time as our performance measures, it is shown that the performance of the three algorithms is almost the same. Therefore, taking into account the time complexity of the task distribution itself, we conclude that a simple and fast heuristic algorithm such as HNF may be sufficient to achieve reasonable performance.

1. Introduction

Recent real-time and 5th generation applications have created a computation gap between the performance capabilities of the existing machines and the performance required by these applications. The attempts to close this gap based on technology improvements alone, have failed due to the physical laws and limitations. The universally acceptable solution is through parallel processing. However, one of the most important issues in a parallel environment is that of the operating system and the management of the parallel tasks. The major question is how to distribute the tasks among several processing elements to achieve minimal execution time.

The task allocation can be performed either dynamically during the execution or statically at compile-time. Dynamic task allocation is based on the re-distribution of tasks among the processors during the execution, with the idea of balancing the load among the idle and busy processors. The approach is especially beneficial if the program behavior cannot be determined before the execution. However, the re-distribution process creates additional overheads which often cause a system performance degradation.

On the other hand, the static task allocation (scheduling) methods attempt to foresee the program execution behavior at compile-time and distribute program tasks among the processors accordingly. Although this eliminates the additional execution overhead, the approach can only be applied to a subset of problems whose run-time behavior is predictable. Examples of such applications include: matrix multiplication, sorting, Towers of Hanoi, Sieve of Eratosthenes, adaptive method of solving partial differential equations [2, 14, 15], etc.

In a static task scheduling approach, a program is represented by a Directed Acyclic Graph (DAG) [8, 11, 14]. A node of a DAG represents an operation or a task, while a directed edge represents the precedence among the nodes. Thus, the task scheduling problem consists of the distribution of the DAG nodes among the Processing Element (PE) so that the maximum parallel operations or minimal execution delay can be achieved. Of course, most programs contain constructs such as loops and conditionals. In general, these constructs do introduce uncertainty in the program behavior. However, in the case of the programs suitable for static task scheduling, these constructs introduce two exclusive solutions to a problem. These two solutions can be represented by two DAGs and both solutions can be distributed to the same set of PEs.

An optimal solution to the task scheduling problem is proven to be computationally intractable [6, 7, 10] and it is not likely to find an optimal algorithm to solve the problem in a polynomial-bound time. Therefore, some of the research efforts in this area have focused on heuristic methods and AI techniques to obtain near optimal solutions [8, 11, 13].

There are several static task scheduling methods including graph reduction, preemptive scheduling, max-flow min-cut, domain decomposition and priority list scheduling [4-6, 8-15]. In this paper we concentrate on the priority list scheduling methods [6, 8, 10]. Here, all the schedulable nodes of a DAG are put into a list according to some priority assigned to each node. The priority is determined based on the heuristic function(s) used. The node with the highest priority is then assigned
to a PE which is thought to be the most appropriate to carrying out the task.

This paper introduces two new static task scheduling algorithms and compares them against the classical Critical Path Method (CPM). The first algorithm, Heavy Node First (HNF), is based on a simple local analysis of the DAG nodes at each level. The second method, Weighted Length (WL), considers a global view of a DAG by taking into account the relationship among the nodes at different level. The three algorithms are analytically analyzed for their non-trivial worst case behavior and their average performances are evaluated through a simulation study.

Section 2 defines the framework of the research by covering our basic assumptions and definitions. Section 3 discusses the task scheduling algorithms and their analysis. Section 4 presents the simulation results. And finally, section 5 is the conclusion.

2. Definitions and Assumptions

In order to set up a framework for our discussions, we first introduce a set of assumptions and definitions. It is assumed that we have a multiprocessor environment in which each processor has its own local memory. The Processing Elements (PES) are connected through some interconnection network and the distributed tasks communicate to each other through a packet switching protocol. In addition, we assume that the number of processors is not enough to cope with the parallelism inherent in the programs (e.g. order of 10's of processors). Another assumption is that a program is represented by a DAG whose nodes are labeled by a weight which represents the length of time to execute the node.

It should be noted that our goal is to compare and evaluate a few heuristic functions for task allocation. We assume that the communication delay is uniform for all the PEs at the implementation level; e.g., a multistage network is used. Here, we combine the program message communication costs with the node execution delays. From now on, a node's weight represents both the execution delay of the node and the communication delay for receiving the message.

We now introduce a set of definitions which will be used in the description and analysis of the algorithms:

**Definition 1:** The accumulate time of PE, \( AT(PE_i) \), for \( 1 \leq i \leq m \) (where \( m \) is the number of PEs), is the total time needed for \( PE_i \) to finish all the tasks assigned to it up to current time.

**Definition 2:** A node is said to be a "mature" node if it is ready to be assigned to a PE. The maturity condition for a node at time \( t_p \) is satisfied if the node has no parent or all of its parent nodes are already assigned to some \( PE (PE_k) \) and \( AT(PE_k) \leq t_p \) (for all \( k \)). The last condition is required to ensure that a task is not completed before its predecessors.

**Definition 3:** The level of a DAG node is recursively defined as follows. All the mature nodes at time 0 are at level 0. All the nodes that mature when all the nodes at level \( i \) are distributed, are at level \( i+1 \).

**Definition 4:** \( LAST(PE_i) \) at time \( t \) is the level of the last node assigned to \( PE_i \) before \( t \). If no nodes have been assigned to \( PE_i \), then \( LAST(PE_i) = -1 \).

**Definition 5:** The "exit path" of a node \( P \) is a path from \( P \) to an exit point of the DAG, whose length is the longest path among all possible paths from \( P \) to any exit point. A node may have several different longest paths. Any one of these paths can be an exit path.

Figure 1 pictorially depicts some of the above definitions. In Figure 1, we use a capitalized character to identify a particular node. The number following the node ID is the weight of the node. Here, \( AT(PE_1) = 4 \) while \( AT(PE_2) = 7 \). The mature nodes at time \( t_2 \) are in the set (C, E, F). The level of node M is 2, \( LAST(PE_2) = 1 \), and exit path of node A is \{A, F, J, P\}. In the following sections we use \( WT(P) \) to indicate the weight of the node \( P \).

![Figure 1. An example of DAG and its partial assignment.](image)
3. The Heuristic Functions

In this section we first introduce a simple heuristic function for a fast task allocation scheme. The second heuristic function discussed here is based on the traditional critical path method. Finally, a more complex heuristic based on a global view of the tasks is proposed.

3.1. Heavy Node First (HNF) Heuristic

The general idea behind this heuristic is to use a simple and localized analysis of the tasks for their allocation to the PEs. For this purpose, we assign the tasks level by level and at each level assign the heaviest nodes first. The algorithm is as follows:

**Heavy Node First Algorithm (HNF)**

**Input:** A DAG with \( n \) nodes and with no redundant transitive arcs.

**Output:** A task schedule of nodes on the PEs.

1. Let \( j \) indicate the current level. Initially, \( j = 0 \). Two lists, CURRENT and NEXT, are used. Initially, both lists are empty.

2. Let \( S = \{ PE_1 | AT(PE_1) \text{ is the smallest at current time} \} \).

3. For each \( PE_i \), repeat steps 3.1-3.3.
   3.1 Let node \( P \) be the last node assigned to \( PE_i \), remove all the edges from \( P \) to its children.
   3.2 If \( LAST(PE_i) < j \), add all the nodes matured by \( P \) to CURRENT.
   3.3 If \( LAST(PE_i) = j \), add all the nodes matured by \( P \) to NEXT.

4. If CURRENT is empty, assign a dummy node to all the PEs in \( S \). The weight of the dummy node is equal to \( AT(PE_j) - AT(PE_i) \) where \( PE_j \) is the node with \( AT(PE_j) \) strictly larger than \( AT(PE_i) \). If no such \( PE_j \) can be found, stop the algorithm, otherwise goto step 2.

5. Assign a node with heaviest weight in CURRENT to a PE in \( S \). Remove this PE from \( S \). Remove the assigned node from CURRENT. If CURRENT is empty Goto step 2. If \( S \) is empty goto step 2.


In this algorithm, load balancing is achieved through i) assigning the mature nodes to the PE's with smallest accumulated time first and ii) assigning the heaviest nodes first. A similar version of this heuristic was introduced in [11].

The HNF algorithm requires \( n \) repetitions to complete, where \( n \) is the number of nodes. If we use a heap to store the nodes in the CURRENT list, one repetition of step 5 requires \( \log(n) \) time units. Therefore, the time complexity of this algorithm is \( O(n \log(n)) \).

Using the HNF algorithm, we may face the situation in which the last node assigned in a level can introduce \( m - 1 \) dummy (idle) nodes, one in each of \( m - 1 \) remaining PEs. The weight of each dummy node is equal to the weight of this node. Figure 2 shows an example of such a case. Assigning \( C \) to \( PE_1 \) causes two dummy nodes to be assigned to \( PE_2 \). If the dummy nodes occur at each level and the DAG nodes have the same weights, we have the worst case behavior of the HNF algorithm. Let \( n \) be the number of nodes in the DAG, \( m \) be the number of PEs, \( s \) be the number of DAG levels, and \( w \) be the weight of each node. The total execution delay under worst case condition is:

\[
T_1 = w \left( \frac{n}{m} + s \left( 1 - \frac{1}{m} \right) \right). \tag{1}
\]

If \( s \) is greater than \( n/m \), there cannot be enough parallelism to keep all the PEs busy. However, we assume that \( s \leq n/m \). Then \( T_1 \) becomes:

\[
T_1 = w \cdot \frac{n}{m} \left( 2 - \frac{1}{m} \right). \tag{2}
\]

Since \( s \leq n/m \), we can define a loose lower bound for the execution as:

\[
T_{low} = w \cdot \frac{n}{m}. \tag{3}
\]

This loose lower bound is valid only if there exists at least one arrangement which guarantees that at any moment there are always at least \( m \) mature nodes for the \( m \) PEs to execute. Comparing \( T_1 \) and \( T_{low} \) we have:

\[
\frac{T_1}{T_{low}} \approx (2 - \frac{1}{m}). \tag{4}
\]

Formula (4) gives the behavior of the HNF under the worst case situation. Under these conditions, the execution time of a schedule that HNF generates is almost twice as long as that of the optimal schedule. The additional delay \( \delta_{ig} \) introduced in the worst case becomes:
\[ T_{dy} = T_1 - T_{low} \leq \frac{n}{m} \left( 1 - \frac{1}{m} \right) \] (5)

If \( m = 1 \) then \( T_{dy} = 0 \) which is the case when the program is executed on a uniprocessor. However, if \( m \) increases (up to \( n \)), then \( T_{low} \) decreases (up to \( w \)), and thus, \( T_{dy} \) approaches \( T_{low} \).

In figure 3, another example of using HNF is shown. The sum of the weights of all the nodes is 18 time units. For a 3-PE system, the total execution time of any optimal schedule can be no less than 6 time units. Here HNF gives an optimal schedule.

In figure 4, another example of using IHNF is shown. The sum of the weights of all the nodes is 18 time units. For a 3-PE system, the total execution time of any optimal schedule can be no less than 6 time units. Herr HNF gives an optimal schedule.

The average case behavior of the algorithm is studied in our simulations and presented in a later section.

3.2. Critical Path Method (CPM)

Many of the earlier and classical task allocation schemes are based on critical path heuristic. The idea is that the tasks on the critical path determine the shortest possible execution delay for the whole program. Furthermore, the tasks on the critical path have to be executed in sequence. Therefore, one may identify the length of a critical path for each graph node, rank the nodes accordingly, and assign the nodes to the PEs based on the priority list scheduling method. In order to be able to complete this process we first need to be able to identify the length of the critical path for each node of the program graph. It should be noted that finding the length of the critical path is equivalent to finding an "exit path" for a node. The following algorithm finds the length of an exit path for each node of a graph.

**Exit Path Algorithm:**

- **Input:** A DAG.
- **Output:** The length of an exit path for each node.
- **Variables:** Let \( n \) be number of nodes in the DAG. For \( 1 \leq i \leq n \), let \( N_i \) be the outdegree (number of children) of node \( P_i \), and \( L_i \) be the length of exit path for node \( P_i \). Let CL be a list. Initially, \( L_i = 0 \), CL is empty, and \( N_i = 0 \) for all the exit nodes.

1. For all \( P_i \) being an exit node, assign the weight of \( P_i \) to \( L_i \). Add all the exit nodes to CL.
2. Let \( P_i \) be a node in CL. Repeat steps 2.1-2.3 until CL is empty:
   1. For each parent node \( P_j \) of node \( P_i \) do:
      1. If \( L_i < WT(P_j) + L_i \), then \( L_i = WT(P_j) + L_i \).
   2. If \( N_j = 0 \) and \( P_j \) is not an entry node, add \( P_j \) to CL.
   3. Remove \( P_i \) from CL.

In this algorithm, each node is processed only once in step 2. Since steps 2.1-2.3 can be repeated a maximum of \( n \) times, the time complexity of the algorithm is \( O(n^2) \).

We now present the Critical Path algorithm for task assignment based on using the length of an exit path for each node.

**Critical Path Method (CPM):**

- **Input:** A DAG with \( n \) nodes and with no redundant transitive arcs.
- **Output:** A distribution of DAG nodes among the PEs.

1. Call the Exit Path Algorithm to get the length of exit path for each node.
2. Repeat steps 2.1-2.4 until the accumulate time of all the PEs is the same and there are no mature nodes available:
   1. Choose \( PE_i \) such that \( AT(PE_i) \) is the smallest.
   2. Find a mature node with the largest exit path length.
   3. If found, assign the node to \( PE_i \).
   4. If there are no mature nodes available, assign a dummy node to \( PE_i \). The weight of the dummy node is equal to the difference between \( AT(PE_i) \) and \( AT(PE_j) \) where \( AT(PE_j) \) is strictly larger than \( (PE_i) \).

Assume that the information regarding the length of an exit path is kept as a heap. Step 1 requires \( O(n^2) \) iterations, while step 2 requires \( n \) times to complete. Step 2.2 requires \( O(\log n) \) times to complete. Therefore, the total time complexity of the CPM algorithm is about \( O(n^2 + n \log n) \) which is \( O(n^2) \).

Figure 4 depicts an example of application of the CPM algorithm to the same DAG used in Figure 3, given three PEs. In Figure 4, the length of exit path of a node is given under the node ID of a node. The principle behind this algorithm is to assign the nodes with the longest exit path length first. Thus, the critical paths of the graph, which represent the minimal execution time requirements, are assigned first. Load balancing is achieved by assigning the nodes to the PE with the smallest accumulation time at each step, and using the
non-critical nodes to fill the empty slots. Note that compared to HNF, the CPM method resulted in a non-optimal solution for this particular DAG.

For our analytical analysis of the CPM algorithm, we need to identify the worst case behavior of the algorithm. Let \( MT(t_i) \) be the set of mature nodes at time \( t_i \). \( P \) is said to be a control node if

\[
WT(P) = \min\{WT(P_i) | P_i \in MT(t_i)\}
\]

at time \( t_i \) and \( P \) is a parent of all the children of all \( P_i \in MT(t_i) \). \( P \) is called a control node since none of the children of the mature nodes in \( MT(t_i) \) can become mature until \( P \) is carried out.

Let \( PE_i \) be the PE to which the heaviest node in \( MT(t_i) \) is assigned. Also, let \( PE_1, PE_2, \ldots, PE_{m'} \) be the processing elements whose accumulated time is less than \( AT(PE_i) \) at time \( t_i \). Then, the total idle time due to \( P \) at time \( t_i \) is:

\[
T_{idle} = \sum_{j=1}^{m'} AT(PE_j) - AT(PE_i).
\]

The average delay caused by \( P \) is:

\[
T_{av,dp} = \frac{T_{idle}}{m}.
\]

This average delay is maximized if the weight of the nodes in \( MT(t_i) \) is the same (e.g. \( w \)) and \( m' = m - 1 \). In this case, we have:

\[
T_{av,dp} = \sum_{k=1}^{s-1} \left( \frac{1}{m} \right) w.
\]

If \( s \) is the number of the DAG's levels, then in the worst case, the total delay is:

\[
T_{dp} = \sum_{k=1}^{s-1} \left( \frac{1}{m} \right) w.
\]

For a control node \( P \), the idle set of \( P \) is defined as the set \( MT(t_i) - \{P\} \). Let \( D_1, D_2, \ldots, D_s \) be idle sets and let \( P_{c_i} \) be the control node for \( D_i \) (\( 1 \leq i \leq s \)). The CPM algorithm's worst case behavior occurs when \( P_{c_1} \) is an entry node, \( P_{c_2} \) is a child of \( P_{c_{i-1}} \) (for \( 2 \leq i \leq s \)), all the children nodes of \( P_{c_s} \) are exit nodes, and \( s = s' \).

Let \( |D_i| \) be the cardinality of the set \( D_i \) and let the weight of each node in the DAG be \( w \). Then, the worst case execution time is:

\[
T_2 = w(s + \sum_{i=1}^{s-1} \frac{1}{m}).
\]

Since \( \sum |D_i| = n \), we have:

\[
T_2 = w(s + \frac{n}{m} + s(1 - \frac{1}{m})),
\]

or equivalently:

\[
T_2 = w(\frac{n}{m} + s(1 - \frac{1}{m})�\).
\]

It is obvious that the worst case execution time ((1) and (10)) and the total additional delay time (8) and (7)) for the HNF and CPM algorithms are respectively equivalent.

### 3.3. Weighted Length (WL) Algorithm

The major problem with the CPM algorithm is that it does not consider the out degree of a node. In other words, the number of nodes (or subgraphs) dependent on a node are not taken into account. This is especially evident in the case of control nodes. For example, in CPM, if a heavy node is assigned to a \( PE \), then several other nodes must be assigned to other \( PE \)s in order to balance the load among the processing elements. But if this node is a control node, all other \( PE \)s must remain idle until the node is completed. In this case, it may be advantageous to assign the control node first (instead of last as in the CPM algorithm) and use the other nodes in the idle set to balance the load among the \( PE \)s.

In order to assign a control node earlier, it must be assigned a priority number which is greater than that of the other nodes. Therefore, we propose an extension to the CPM algorithm in which the heuristic consists of the length of an exit path, the branching factor for each node, the number of children nodes and the weights of the children nodes and their descendants. The heuristic assigns each node a number which we call weighted length. The weighted length singles out the control node more effectively and is expected to compensate the shortcomings of CPM. Let \( WL(P) \) be the weighted length of node \( P \), \( U(P) \) be the maximum weighted length of the children of \( P \), and \( V(P) \) be the summation of the weighted lengths of the children of \( P \). Then, the weighted length of node \( P \) is defined as:

\[
WL(P) = WT(P) + U(P) + \frac{1}{U(P)} \cdot \frac{1}{V(P)}.
\]

where, \( WT(P) \) is the weight of node \( P \). Note that the sum of the weighted length of the children of \( P \) are normalized over \( U(P) \). In order to complete the Weighted Length method, we first need an algorithm for calculating the weighted lengths of the nodes of a DAG:

**Weighted Length Calculation:**

**Input:** A DAG.
Output: The weighted length of each node.
1. Set $WL(P_i) = WT(P_i)$ for all $P_i$'s being an exit node.
2. Repeat steps 2.1-2.3 for every node $(P)$ whose $WL(P)$ is unknown, but the weighted lengths of its children are already calculated:
2.1 Find $U(P)$.
2.2 Calculate $V(P)$.
2.3 Set $WL(P) = WT(P) + U(P) + \frac{V(P)}{U(P)}$.

The time complexity of this algorithm is $O(n^2)$ since step 2 is repeated $n$ times and for each iteration, steps 2.1 and 2.2 may require up to $n$ operations. Here, $n$ is the number of nodes in the DAG.

The Weighted Length algorithm is exactly the same as the CPM algorithm with two exceptions. First, in step 1 we need to call the Weighted Length Calculation to find the weighted length of the DAG nodes. Second, in step 2.2 instead of finding a mature node with the largest exit path length, we need to find a mature node with the largest weighted length.

Figure 5 depicts an example of WL task allocation scheme. The DAG used in this example is the same as the one used in the HNF and CPM examples except that the length of exit path is replaced by the weighted length. It is noticeable that the total execution time is reduced compared to the CPM allocation and that no dummy nodes are introduced.

![Figure 5. An example of WL task allocation.](image)

Table 1. The execution time in different experiments.

<table>
<thead>
<tr>
<th>No. of PEs</th>
<th>HNF</th>
<th>CPM</th>
<th>WL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>125719</td>
<td>125719</td>
<td>125719</td>
</tr>
<tr>
<td>2</td>
<td>63702</td>
<td>63588</td>
<td>63588</td>
</tr>
<tr>
<td>3</td>
<td>43042</td>
<td>43310</td>
<td>43279</td>
</tr>
<tr>
<td>4</td>
<td>34125</td>
<td>33583</td>
<td>33517</td>
</tr>
<tr>
<td>5</td>
<td>28835</td>
<td>23206</td>
<td>28150</td>
</tr>
<tr>
<td>6</td>
<td>25818</td>
<td>25074</td>
<td>25025</td>
</tr>
<tr>
<td>7</td>
<td>23901</td>
<td>23288</td>
<td>23264</td>
</tr>
<tr>
<td>8</td>
<td>22710</td>
<td>22212</td>
<td>22210</td>
</tr>
<tr>
<td>9</td>
<td>22063</td>
<td>21991</td>
<td>21681</td>
</tr>
</tbody>
</table>

In the WL algorithm, the small exit path length of the important control nodes is compensated by considering the weight of its children. This gives the control nodes a better chance of being allocated first with good opportunities for using the other nodes at that same level for load balancing.

The time complexity of the WL algorithm is the same as the CPM algorithm, i.e. $O(n^2)$. The worst case occurs when at every moment all the mature nodes are also control nodes. In that case, the execution time of the tasks will be the same as the CPM schedule. However, the possibility of the occurrence of the worst case in the WL is very small.

4. Performance Evaluation

So far, we have presented an analysis of the worst case behavior of the three heuristic-based task allocation schemes, i.e. HNF, CPM, and WL. However, the analytical evaluation of these algorithms provides little information for their comparison. The three algorithms behave the same under their own worst case behavior. It must be noted that the difference of the execution times between a schedule generated by a heuristic algorithm and an optimal schedule is maximized when a DAG represents a worst case for the heuristic algorithm. The worst case analytical models that we presented here show the situation that a heuristic algorithm unnecessarily generates many idle tasks. For example, if a control node is executed earlier, the PEs will not be idle before the execution of all the nodes in one level is finished. Such unnecessary idle tasks eventually extend the total execution time.

In order to be able to evaluate the performance of the three methods based on the average cases, we conducted three simulation studies. The three algorithms were tested by their application to randomly generated DAGs using a varying number of PEs. In the first study, 110 different DAGs are randomly generated. Each DAG has 40 nodes and the weight of each node is randomly chosen between 1 and 50. The generated DAGs are then distributed between 1 to 9 processors using HNF, CPM, and WL algorithms. In each experiment, the total execution time and idle time are measured. Table 1 depicts the execution times in different cases, while Table 2 shows the total processor idle times in each experiment. From the simulation result we can see that the WL heuristic has resulted in a slightly better distribution compared to the CPM, and the CPM distributes the tasks a bit more efficiently than the HNF. However, the difference in the execution times and idle times for the three cases do not seem to be significant.
Table 2. The Processor idle times in different experiments.

<table>
<thead>
<tr>
<th>No. of PE</th>
<th>HNF</th>
<th>CPM</th>
<th>WL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1685</td>
<td>1477</td>
<td>1457</td>
</tr>
<tr>
<td>3</td>
<td>5207</td>
<td>4211</td>
<td>4118</td>
</tr>
<tr>
<td>4</td>
<td>10781</td>
<td>8613</td>
<td>8469</td>
</tr>
<tr>
<td>5</td>
<td>18456</td>
<td>16311</td>
<td>15970</td>
</tr>
<tr>
<td>6</td>
<td>29189</td>
<td>24725</td>
<td>24431</td>
</tr>
<tr>
<td>7</td>
<td>41588</td>
<td>37207</td>
<td>37127</td>
</tr>
<tr>
<td>8</td>
<td>56055</td>
<td>51810</td>
<td>51803</td>
</tr>
<tr>
<td>9</td>
<td>72607</td>
<td>60180</td>
<td>60155</td>
</tr>
</tbody>
</table>

In the second study, we experimented with the range of the weights for each node. In these experiments, 40 40-node random graphs were used and the number of processors ranged from 1 to 10. We first used a fixed weight of 10 units of time for each node. The speedup behavior of the three methods is presented in Figure 6(a). Then, we randomly chose the weight of the nodes in the range of 10 to 20 time units. Figure 6(b) depicts the speedup behavior of the three methods. In Figure 6 the speedup behavior of WL and HNF algorithms are compared to CPM. Here, EX(K) indicates the execution time resulting from application of heuristic K. As shown in Figure 6, when uniform weights are used, both HNF and WL methods out-perform CPM, although HNF is only slightly better. However, for a random distribution of weights in the range of 10 to 20 time units, WL is still performing better than CPM but CPM becomes better than HNF. Again, the differences of the execution time for different algorithms are not significant. In other words, the average speed up behavior of HNF and WL over CPM ranges from 0.98 to 1.03 which is essentially the same due to the assumed 10% margin of error.

In the third study, we tried to find out how close the results from the HNF algorithm are to the optimal results. Unfortunately, this was a very time consuming experiment. We could only afford to run some example DAGs with few nodes. We generated 100 10-node DAGs and 100 13-node DAGs. For each such DAG, there were two versions, one with unit weight nodes and the other with the node weight varying between 1 and 20. Each time a DAG was generated, the exhaustive search method was used to find out the optimal solution. Then HNF algorithm was applied and the results from the two methods were compared. Table 3 gives the result of the experiments. The schedules are based on a three PE system. We did not try DAGs with more nodes or use more PEs because the machine time increases drastically. It is interesting to note that the HNF always generated an optimal solution when a unit weight is used for the tasks and the number of nodes is limited to a small number (10).

Table 3. Optimality rate of the HNF algorithm.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>Optimality Rate of HNF</th>
<th>Optimality Rate of Random Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-node with varying weights</td>
<td>0.93</td>
<td>0.73</td>
</tr>
<tr>
<td>13-node with unit weight</td>
<td>1</td>
<td>0.94</td>
</tr>
<tr>
<td>13-node with varying weights</td>
<td>0.76</td>
<td>0.43</td>
</tr>
</tbody>
</table>

According to the above performance studies WL method is seemingly better than the other two. But the differences are very small. It is generally true that the more information is used by a heuristic, the better the results should be. But, it is also interesting to see how the efforts that are spent on collecting information is compared to the improvement of the result. In the task scheduling situation, the critical path algorithm and weighted length algorithm include more information about the DAG to make the scheduling decision. However, the improvements in their results do not match the efforts spent to collect the information. The HNF, even though very simple, gives results of the same quality as the other two heuristics.

5. Conclusion

This paper presented three different heuristic functions for static task allocation in a distributed multiprocessor environment. The first heuristic function (HNF) uses local, level by level information from a DAG to distribute the tasks. The second heuristic function (CPM)
takes advantage of the length of critical path for each node to achieve the allocation. Finally, the third heuristic (WL) extends CPM to include more global information for each node, such as the weight of the nodes in the subgraphs which are dependent on that node.

The three methods behaved similarly under worst case conditions as presented by analytical results. But intuitively we believe that the worst cases for different methods occurs with different probabilities. The worst case for WL occur with the least probability. This can be verified by the simulation results which indicate that WL is a slightly better than the other two. The simulation results show three methods in average cases. In our study, the time complexity of HNF allocation algorithm is $O(n \log n)$, while time complexity of the CPM and WL is $O(n^2)$. Judging the algorithms with their time complexity and the results they produce, HNF algorithm is obviously a very promising scheduling method.

References


