1982

Task scheduling on multiprocessors

Ravi Mehrotra
Carnegie Mellon University

Talukdar

Follow this and additional works at: http://repository.cmu.edu/ece
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
TASK SCHEDULING ON MULTIPROCESSORS

by

R. Mehrotra & S.M. Talukdar

December, 1932

DRC-18-55-82
ABSTRACT

The paper describes a technique for estimating the minimum execution time of an algorithm or a mix of algorithms on a multiprocessor. Bottlenecks that would have to be removed to further reduce the execution time are identified.

The main applications are for designing special purpose, dedicated multiprocessors. Today, a bewildering array of computer components - processors and devices with which to interconnect them - are available. The future will bring even more of these components. To intelligently choose mixes of them one needs systematic procedures.

In the procedure of this paper the multiprocessors are modelled by $P$, a set of processors and $R$, a set of resources that the processors can use. The algorithms are modelled by $T$, an ordered set of tasks. The problem of optimally assigning the processors to the tasks while meeting the resource constraints is NP-complete. A heuristic using maximum weighted matchings on graphs has been devised that is extremely fast and produces solutions that are reasonably close to the optimal solutions. The heuristic has been coded in Fortran and illustrations of its use included.

1. INTRODUCTION

1.1 The General Form of a Distributed Multiprocessor

Most existing codes have been written for Von-Neumann, general purpose computers with large virtual memories. However, it is now possible to assemble non-Von-Neumann architectures, often using just off-the-shelf components. The general form of one such class of architectures is shown in Fig. 1.1. The processors may have diverse processing, input and output characteristics and may be physically close or geographically separated. The communication network may use a transmission means (such as wires, satellites and optical fibers) and a variety of configurations (such as stars, trees and loops). We will elaborate on these alternatives in 1.4 and 1.5.

1.2 A Very Brief Review of Previous Work [1]-[5]

Research into the use of special computers for power system appreciations has concentrated on three limited possibilities:

1. The exclusive use of large vector machines like the CRAY-1

2. The combination of a host machine with an array processor like the AP120-B.

3. Homogeneous multiprocessors (large numbers of identical processors symmetrically interconnected)

These research efforts have met with some, but not spectacular, success. The reason is that power system algorithms contain a wide variety of tasks. Some work well on vector machines and array processors, others do not. Some work well on homogeneous multiprocessors, others do not. Therefore, the exclusive use any one limited hardware arrangement will inevitably lead to severe bottlenecks.

1.3 The Design Problem

To alleviate bottlenecks we need to ask the question: How can we assemble a computer system with the diverse skills needed to efficiently process all the tasks in a given power system algorithm or mix of algorithms? One way to go about answering this question is to take the following four steps:

1. Identify the computer components that are not available or will soon be available.

2. Categorize the tasks or, alternately, identify a set of primitives from which the tasks can be synthesized.

3. Determine how effective each component is for each primitive.

4. Devise a scheme for assembling mixes of components to best handle given mixes of primitives.

The result of taking these steps will be a computer of the type shown in Fig. 1.1 and dedicated to a mix of algorithms.

The emphasis of this paper is on step 4. We will discuss the other steps but to considerably lesser degrees.

1.4 Network Alternatives

Computer communication networks can be divided into three categories.
Whatever classification scheme is used, it is difficult to keep the categories from overlapping. One set of categories is:

- Special purpose, dedicated VLSI processors
- Minicomputers like DEC'S VAX/780 and PDP-11/70.
- Large, general purpose machines like IBM's 3030, Burroughs B5000, CDC's 7600, as well as large vector machines like the CRAY-1 and STAR-100.

1.6 Goals and Organization

In the past, computer alternatives were evaluated by tedious benchmarking or almost as tedious simulations, in view of the variety of alternatives now available, neither of these approaches is practical until the very last stages of the search for a computer system. For the early parts of the search one needs evaluation tools that are quick and easy to use. The rest of this paper will be devoted to the development of one such tool and to a simple example of its use. Specifically, the tool is an efficient, interactive program that estimates the minimum execution time and identifies the bottlenecks for a given mix of tasks on a given mix of computer components. The results enable the user to determine whether the mix of components is promising and how to incrementally change it to improve performance.

Section 2 gives a formal description of the problem we shall consider. Section 3 indicates a solution method after a brief review of methods that have been used to solve similar problems. The procedure described in section 3 has been coded as a user friendly interactive program in FORTRAN on DEC-20 system and is used to study a proposed multiprocessor architecture in section 4. Section 5 summarizes the results obtained so far and lists further work to be done.

2. MATHEMATICAL FORMULATION OF THE PROBLEM

This section establishes the basic vocabulary for the remainder of the paper and gives a precise mathematical formulation of the problem to be considered.

2.1 Algorithms

An algorithm A is described by $A^*\{T, c\}$ where T and c are sets defined as follows:

- $T$ is a set of tasks $\{T_1, T_2, \ldots, T_n\}$ and the set $c_{T_1}$ denotes the partial ordering relation on $T$ such that $T_a$ a $T_b$ implies that the execution of tasks $T_b$ called the successor of $T_a$ cannot begin until the execution of $T_b$ (called the predecessor of $T_a$) has been completed.
- When $a$ is empty the tasks are called independent. It is assumed that the tasks to describe $A$ are chosen from a finite set of tasks $T^c = \{T_1, T_2, \ldots, T_n\}$ of $n$ primitive tasks. Each task $T_a < T$ corresponds to some primitive task $T_a \in T^c$.

2.2 Multiprocessors

We may think of the multiprocessor architecture at a high level as having been assembled from processors that execute tasks in $T^c$ with the use of certain resources such as disk drives, tapes, memory and the interconnecting devices including the buses and data links that form the communication network of the system. The multiprocessor system $MP$ with $M$ processors and $L$ resources will be denoted by $MP\{P, R\}$ where $P = \{P_1, P_2, \ldots, P_M\}$ is a set of $M$ processors and $R = \{R_1, R_2, \ldots, R_L\}$ is a set of $L$ resources.
2.3 Algorithm: Multiprocessor Interaction

Each task \( T \leq T \) may be executed on any processor in \( P \). We define a function \( \lambda (T W_{11}.. T^{\omega}) \) so that the value at \( T \) represents the execution time it takes to execute task \( T \) on processor \( P \). Furthermore we define a function \( r(T)(r_1, r_2, ..., r_j) \) to represent the resource requirements of task \( T \) such that \( r_j \) is equal to the amount of discrete resource \( R \) needed while executing \( T \). \( R \) is the total units of \( R \) in the system.

2.4 Execution time of \( A \) on \( MP(P,R) \).

Let \( r(i) \) represent the starting time of the execution of task \( T_i \).

Define \( X(k)=1 \) if task \( T \) is executed on processor \( P \), at time \( k \) and zero otherwise. It is assumed that time is measured in terms of equal and indivisible units. Using the notation introduced in this section we define a feasible schedule to be a mapping \( \psi: T \rightarrow 1 \) such that the following 3 conditions are satisfied. (1 is a one dimensional space of integers representing time).

\[ \psi(T) \leq \psi(T') \quad \text{for} \quad i,j=1..K, \text{all} \quad k \geq 1 \]

\[ \psi(T) \leq \psi(T') \quad \text{then} \quad r(T) \leq r(T') \quad \text{for} \quad i,j=1..K, \text{all} \quad k \geq 1 \]

\[ \psi(T) \geq \psi(T') \quad \text{then} \quad r(T) \leq r(T') \quad \text{for} \quad i,j=1..K, \text{all} \quad k \geq 1 \]

C1 is needed to avoid the assignment of a task to more than one processor. C2 is a statement of the procedure constraints of \( A \). C3 is needed to ensure that the resources required by a job will be available while the job executes.

Corresponding to each feasible schedule \( \psi: T \rightarrow 1 \) we define the execution time of the algorithm \( A \) on \( MP \) as:

\[ L^P = \sum_{T} \psi(T) \quad \text{for} \quad i=1..K, j=\psi(U) \]

Thus the problem of finding the optimal assignment of tasks in the algorithm \( A \) on a multiprocessor \( MP \) so as to minimize the overall execution time may be stated as GSP

\[ \text{minimize} \quad L^P \quad \text{subject to} \quad \psi(T) \rightarrow 1 \]

We can find a lower bound \( L^P \) on \( L^P \) as follows:

For every node \( n_i \) of \( G(V,E) \) define the weight of \( n_i \) \( W(n_i) \) as:

\[ W < \min (t_{n_i}, t_{n_i},.., t_{n_j}) \]

Define the length of a directed path from node \( n \) to node \( n_i \) to be equal to the sum of weights of all the nodes in the directed path from \( n \) to \( n_i \). The longest directed path from a node with no predecessor to a node with no successor represents a lower bound on \( L^P \). This lower bound \( L^P \) is obtained by assuming that all the tasks in \( G(V,E) \) are assigned to the processor on which they take minimum execution time and there is a sufficient number of processors and resources in the system.

If the solution to GSP gives a value of \( L^P > L^P_{\infty} \), then the elements of the set \( P \) and \( R \) may be modified to reduce the difference between \( L^P_{\infty} \) and \( L^P\). This allows us to reconfigure a multiprocessor system that is most suited for executing the specific algorithm. The solution procedure (see section 3) used to solve (2.4) enables us to identify, what limits performance, thus suggesting a natural modification of the set \( P \) and/or \( R \).

2.5 A Cost Constraint

Let \( CP(P) \) represent the cost of processor \( P_i \), \( i=1..M \).

A cost constraint may be added to obtain GSPC as follows:

\[ \text{minimize} \quad L^P \quad \text{subject to} \quad \psi(T) \rightarrow 1 \]

All processors in the set \( P \) are not necessarily used. The solution procedure would select the particular mix of processors that minimizes the overall execution time with the total cost of processors in the mix being no more than \( C \).

3. SOLUTION PROCEDURE

GSP is a notoriously hard combinatorial analysis problem known as the General Scheduling Problem. This problem is MP-complete. Instead of seeking polynomial time optimal algorithms for NP-complete problems, one uses heuristic or approximate algorithms since, NP-complete solutions in polynomial time. Most work done in the area of scheduling has been devoted to the case when all processors in the system are identical [25]-[26]. Some enumerative and iterative techniques such as 'local search' and 'branch and bound' have been applied to subproblems of GSP [36]-[39]. But there are no heuristics for solving GSP itself and branch and bound techniques are computationally impractical for solving it.

We will proceed to develop a heuristic technique for solving GSP. The technique is based on finding maximum weighted matchings on graphs. It yields reasonably good solutions to GSP and GSPC.

The essential steps are:

1. Input Edge List Matrix of \( G(V,E) \). Execution Time Matrix \( [t] \) and Resource Requirement Matrix \( [r] \) (see Example 3.1).

2. Assign levels to the nodes. \( T_i \), of the task graph \( G(V,E) \). Intuitively, levels assigned to nodes are distances from a node with no successor and represent the precedence structure of \( G(V,E) \).

3. Making use of the levels of the nodes, assign corresponding tasks on the processors, disregarding the resource constraints. This step is carried out by finding maximum weighted matchings.

4. Schedule the tasks on the processors they have been assigned to, taking resource constraints into account. Make a list of resource shortages if any.

5. Repeat steps 3 and 4 until all tasks have been scheduled.

6. Output the schedule and the list of resource shortages (see Example XI).

The details of steps 2-5 are given in the appendix.

The heuristic to solve GSPC is similar to the above heuristic. The cost constraint is factored into the solution process by including it in the objective function of the maximum matching problem [40]-[42].
Figure 3.1. A task graph, G (V,E)

Table 3.1(a). Edge List Matrix.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1(b). Execution Times Matrix.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1(c). Resource Requirement Matrix.

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2. Resource Shortage Table.

<table>
<thead>
<tr>
<th>Task</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 3.2. Pictorial Representation of the Schedule.

Example 3.1. Consider a multiprocessor system MP(P,R) with P = {P1, P2} and R = {R1, R2}.

The inputs required for task graph of Fig 3.1 are given in Table 3.1(a-d). The values of [t] and [r] are specified in Table 3.1(b) and 3.1(c) respectively. (R1) = 2 units and (R2) = 3 units. L = 3.

Fig 3.2 is a pictorial representation of the output of the procedure. Note that L = 5. Table 3.2 is the output which indicates the bottlenecks which must be removed to reduce the overall execution time. Task T4 could not be started in parallel with T5 and T6 because of resource shortage. It needed 3 units of R1 and 2 units of R2 and none were available. If we let \( \sum R_i \leq 6 \) and \( \sum R \) then L would be equal to 5.

4. JJJSTRODUCTION OF THE DESIGN PROCESS

The solution procedure outlined in the previous section has been translated into a user friendly, interactive, FORTRAN program called SNONUET. It allows the user to modify the input parameters until either satisfactory execution time is obtained, or no further improvement is possible. SNONUET has been tested on a number of randomly generated examples and it produced near optimal schedules in most cases. The purpose of this section is to illustrate some of the uses of SNONUET. To do this we will use a simple example, chosen for explanatory purpose rather than realism.

4.1 Algorithmic Primitives

The first step in preparing the input data for SNONUET is to identify a set of primitive tasks, \( T_i \), in terms of which to describe the algorithm in question.

The primitives can be at various levels. Very high level primitives result in simple task graphs with a few nodes. For instance, an optimum power flow [43] can be described in terms of high level primitives by the task graph in Fig 4.1(a). Or, using the primitives given in Table 4.1, we could expand each node of the task graph. The task graph corresponding to FLF is shown in Fig 4.1(b).

A reasonable way to proceed is to use high level primitives for the initial design and then refine the design with lower level primitives.

4.2 Processor and Communication Network Alternatives

The first step in preparing the input data for SNONUET is to choose the processor and communication network alternatives to be considered.

We start with a unibus multiprocessor system shown in Fig 4.2, which is a special case of the general

Table 4.1

<table>
<thead>
<tr>
<th>Set of Primitive Tasks</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>No operation (used for synchronization)</td>
</tr>
<tr>
<td>T1</td>
<td>Vector Add (( C_i \cdot A_j + b_i, i=1..n ))</td>
</tr>
<tr>
<td>T2</td>
<td>Vector Inner Product (( C - I \times b_i ))</td>
</tr>
<tr>
<td>T3</td>
<td>Vector Scale (( b_i \cdot A_i, i=1..n ))</td>
</tr>
<tr>
<td>T4</td>
<td>Vector Divide (( b_i \cdot A_i, i=1..n ))</td>
</tr>
<tr>
<td>T5</td>
<td>Vector Multiply (( C_i \cdot A_i, i=1..n ))</td>
</tr>
<tr>
<td>T6</td>
<td>Vector Sort (arrange elements in an increasing order)</td>
</tr>
<tr>
<td>T7</td>
<td>LU Factorization</td>
</tr>
<tr>
<td>T8</td>
<td>Back Substitution (used to solve a set of linear equations)</td>
</tr>
</tbody>
</table>

Table 4.2

<table>
<thead>
<tr>
<th>Task</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLF</td>
<td>FULL LOAD FLOW</td>
</tr>
<tr>
<td>HU</td>
<td>HESSIAN UPDATE (BFGS FORMULA)</td>
</tr>
<tr>
<td>OPP</td>
<td>QUADRATIC PROGRAMMING PROBLEM SOLUTION</td>
</tr>
<tr>
<td>FE</td>
<td>FUNCTION EVALUATION</td>
</tr>
<tr>
<td>MVO</td>
<td>SOME MATRIX AND VECTOR OPERATIONS</td>
</tr>
<tr>
<td>SFE</td>
<td>FUNCTION EVALUATION AND SEARCH</td>
</tr>
</tbody>
</table>

Fig. 4.1. (a) Task Graph of \( mn \) Optimum Power Flow using high level primitives.

Fig 4.1 (a) Task Graph of \( mn \) Optimum Power Flow using high level primitives.

4.1 Algorithmic Primitives

The first step in preparing the input data for SNONUET is to identify a set of primitive tasks, \( T_i \), in terms of which to describe the algorithm in question.

The primitives can be at various levels. Very high level primitives result in simple task graphs with a few nodes. For instance, an optimum power flow [43] can be described in terms of high level primitives by the task graph in Fig 4.1(a). Or, using the primitives given in Table 4.1, we could expand each node of the task graph. The task graph corresponding to FLF is shown in Fig 4.1(b).

A reasonable way to proceed is to use high level primitives for the initial design and then refine the design with lower level primitives.

4.2 Processor and Communication Network Alternatives

The first step in preparing the input data for SNONUET is to choose the processor and communication network alternatives to be considered.

We start with a unibus multiprocessor system shown in Fig 4.2, which is a special case of the general
multiprocessor system of Fig 1.1 - the communication network is now the data channel of the host computer. The motivation for using the common data bus is the simplicity of the interconnection, also if the communication over the bus does not limit performance there would be no need to consider more sophisticated interconnection schemes. Each special purpose device is a processor/multiprocessor realized in VLSI, with its own private memory. The unibus of the system is considered to be a resource of the system. If the total time needed for all data transfers over the bus of all tasks in G(V,E) at any level is found to be more than 10H of L°° then the bus is considered to be a bottleneck. The details of the bus modelling procedure are described in [42]. SNONUET finds the latest finishing time of all tasks and identifies bottlenecks. If the unibus of the system is not a bottleneck, the number of special processors of a given type may be increased to check if further reduction in the overall execution time is possible. On the other hand if the unibus turns out to be a bottleneck, we introduce another bus amongst the processors sharing the congested bus, to relieve the congestion and improve speedup. The above steps are repeated until no further reduction in execution time results.

4.3 Cost and Time Data

The third and the major step in preparing the input data for SNONUET is to estimate the cost of the processors and the task running times.

We consider the execution of the G(V,E) of FLF shown in Fig 4.1(b) on the multiprocessor of Fig 4.2. Three different types of processors are considered, an array processor AP (such as AP 1208) and two special purpose VLSI peripheral processors SP1 and SP2. The estimates of the execution time of the host and the three types of processors considered are listed in Table 4.3.1. The per unit cost of the three types of processors is shown in Table 4.3.2.

4.4 Results

The output of SNONUET when no cost constraint was placed indicated that overall execution time could be reduced by increasing the number of APs to 5. The overall execution time obtained by SNONUET has been plotted vs the number of APs in Fig 4.3.1, after scaling it so that the stand alone host could sequentially execute FLF in 100 units of time.

The overall running time vs cost is plotted in Fig 4.3.2. It is important to note that SNONUET with a cost constraint generates only the points on the broken line. The other points correspond to mixes of processors that should not considered because they provide lesser speedups for the same cost.

<table>
<thead>
<tr>
<th>PRiNAlty</th>
<th>Task</th>
<th>AP</th>
<th>SP1</th>
<th>SP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T1</td>
<td>100</td>
<td>35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T2</td>
<td>1100</td>
<td>410</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T3</td>
<td>1000</td>
<td>340</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T4</td>
<td>1000</td>
<td>340</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*sort</td>
<td>4550</td>
<td>5000</td>
<td>700</td>
<td>0</td>
</tr>
<tr>
<td>UDF1</td>
<td>4000</td>
<td>3000</td>
<td>0</td>
<td>400</td>
</tr>
</tbody>
</table>

TABLE 4.3.1. Initial* of Execution Time

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>50</td>
</tr>
<tr>
<td>SP1</td>
<td>10</td>
</tr>
<tr>
<td>SP2</td>
<td>20</td>
</tr>
<tr>
<td>MGT</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE 4.3.2
5. CONCLUSIONS

This paper has described a systematic procedure that is useful in the selection and design of dedicated multiprocessors. The procedure has been coded into an interactive FORTRAN program called SNONUET.

Before SNONUET can be used one must break the algorithm into ordered tasks, select a set of processors for consideration and select a set of resources for the processors to use. One must also estimate the time and resource requirements for each primitive task. SNONUET will select a subset of processors and schedule them so as to minimize the execution time of the algorithm(s) and satisfy the cost constraint. This procedure enables us to plot the optimal speedup vs cost for a fixed communication network. It would be very useful when selecting a set of processors from those commercially available, when a local area network is used as the communication network of the system.

The example in Section 4 was chosen to illustrate the use of SNONUET, and not as a realistic design exercise. It could however be used for designing multiprocessors if they were to be dedicated to solving load flow problems.

Some further work needs to be done to obtain a better way to model the communication network when its architecture is specified and queueing delays are involved as a result of packet switching. In the present model, for each task requiring the communication network, expected queueing delays are added to the message transit times. The sum is treated as a deterministic time for which the resource corresponding to the communication network may not be used by another task. This is a major drawback of the procedure and we are working towards a remedy.

6. ACKNOWLEDGEMENT

This work was supported in part by the Electric Power Research Institute under project RP1764-3.

References
APPENDIX

The nodes, $T_x$, of the task graph $G_1(V,E)$ of an algorithm $A$ are assigned two levels $L^B(T_x)$ and $L^F(T_x)$ by the algorithm below.

**Algorithm Assign Levels:**

1. If $T_x$ has no successors, then $L^B(T_x) = 1$; otherwise, $L^B(T_x) = \max\{L^B(S) + 1\}$ for all tasks $T_S$.

2. Let $L^F(T_{max})$ represent the smallest integer such that $L^F(T_{max}) \geq L^B(T_x)$ for all tasks $T_x$.

3. If $T_x$ has no predecessor, then $L^F(T_x) = \max\{L^B(T_S) : T_S \in T^M\}$, otherwise, $L^F(T_x) = \min\{L^B(T_S) : T_S \in T^M\}$.

We present an intuitively obvious elementary theorem. Theorem. All tasks $T_x$ with $L^B(T_x) = L^F(T_x)$ may be started in parallel as independent tasks if all tasks $T_y$ with $L^F(T_y) = L^B(T_x)$ have completed execution, without violating the precedence constraints imposed by $G_1(V,E)$.

The tasks $T_x \in T$ are first assigned to processors $P \in P$ without regard to the resource constraints $R^* \in R$ and then scheduled on them taking into account the resource constraints. If resource constraints are violated, the starting time of the task is delayed until sufficient amount of resources are released by tasks which have already been scheduled. The tasks in $T$ are scheduled in the decreasing order of their levels $L^F(T_x)$.

In order to understand how the scheduling procedure works it is convenient to assume that all tasks $T_x$ with $L^F(T_x) = L^B(T_x)$ have already been assigned to processors and scheduled on them: Consider the set $\{T_x \in T : L^F(T_x) = L^B(T_x)\}$ of tasks $T_x$ such that $L^F(T_x) = 1$. Let $J_x = \{T_1, T_2, \ldots, T_y\}$ and define the set $J = \{1, 2, \ldots, |J_x|\}$ so that the elements of $J$ and $J_x$ are one to one correspondence with tasks in $J$. Let $P = \{1, 2, \ldots, |J|\}$ represent the set of processor indices to which tasks are to be assigned without regard to the resource requirements. This assignment problem may be formulated as an NP-complete integer linear program (ILP).

**ILP**

\[
\begin{align*}
\text{subject to } & \quad \sum_{i=1}^{J} z_{ij} \leq b_i^* \quad \text{for all } i \in I, \\
\end{align*}
\]

ILP is known as the Maximum Weighted Matching problem, $y^*$'s have the same interpretation as $z^*$. $c_{ij}$ and $b_i^*$ are defined by the algorithm Assign Tasks. Solution to (ILP) yields an upper bound, UB, for the solution to (ILP). The inequality constraints and the objective function of ILP are modified to improve UB and bring it closer to the solution of ILP.

**Algorithm Assign Tasks:**

1. Initialize: $b_i^* = \infty$ for all $i \in I$.

2. Solve (ILP').

3. For $i \in I$ if $t_{ij} = 1$ then $t_{ij} = 1$ for all $j \in J$.

4. Stop if $t_{ik} = 1$ for all $i \in I$.

5. Go to step 2.

6. Go to step 2.
Each time a task which has been assigned cannot be scheduled because of insufficient resource, an entry is made in a Resource Shortage Table indicating the particular task which could not be scheduled together with units of the particular resource/resources which were needed but were not available. Once all tasks $T_x$ with $L^B(T_x) \leq 1$ have been scheduled, tasks $T_x$ with $L^O(T_x) = 1$ are assigned to processors and then scheduled. The steps are repeated until all tasks $T_x$ with $L^B(T_x) \leq 1$ have been scheduled. The specific details and the rules for breaking ties while forming the different sets are described in [43].

The tasks on a processor are scheduled in the increasing order of their total resource requirements. Let $V^r_x \ldots r^s_x$ represent the total resource requirement of all tasks $T_x$ such that $L^B(T_x)-1$ or $L^O(T_x)-1$ and task $T_x$ has been scheduled. The remaining tasks are scheduled by the following algorithm:

**Algorithm Schedule Tasks**

1. While TP is not empty perform the step

   a. Let $s_0 = \max \{ \ell_{p_i} \mathbf{J} L^B(T_x) = 1 \}$

   b. For each $v, 1 \leq v \leq L$, let $r^v \mathbf{J} \ell_{p_i}$ where $x$ is such that $L^B(T_y) = 1$ or $L^O = 1$ and $T_x$ has been scheduled.

   c. Let $t_{p_i}$ be the first element in TP. While $Y$ is not empty perform the step

      i. Let $T_s$ be the first element in TP. While $Y$ is not empty perform the step

         a. If for each $v, 1 \leq v \leq L$, $r^v \mathbf{J} \ell_{p_i}$ then let $r(T_s) = s$, for each $v$, let $r^v \mathbf{J} \ell_{p_i}$ and remove $T_s$ from TP. Task $T_s$ is scheduled on processor $P$ and $X_{p_i}^t = 1$ for $k = s+1, \ldots, L$.

   d. Remove $t_{p_i}$ from TP.

2. Form a set of all tasks $T_x$ which have not been scheduled and have $L^B(T_x) \leq 1$. Repeat this step until $Y$ is empty. If for each $v, 1 \leq v \leq L$, $r^v \mathbf{J} \ell_{p_i}$ and if for some $i$, $t^i_p = t^i_p$ then schedule $T_s$ on $P$, and remove $T_s$ from $Y$. Else remove $T_s$ from $Y$.  
