[8] S. Feldman and C. Brown, "A system for program debugging via reversible execution," *ACM SIGPLAN Notices, Workshop on Parallel Distrib. Debugging*, vol. 24, pp. 112–123, Jan. 1989.
[9] R. Fitzgerald and R. F. Rashid, "The integration of virtual memory management and interprocess communication in accent," *ACM Trans. Comput. Syst.*, vol. 4, no. 2, pp. 147–177, May 1986.
[10] R. B. Hagmann, "A crash recovery scheme for a memory-resident database system," *IEEE Trans. Comput.*, vol. C-35, no. 9, pp. 839–843, Sept. 1986.
[11] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. Algorithms*, vol. 11, pp. 462–491, Sept. 1990.
[12] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum, "Transparent fault-tolerance in parallel Orca programs," Tech. Rep. IR-258, Vrije Univ., Amsterdam, the Netherlands, Oct. 1991.
[13] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, vol. 13, no. 1, pp. 23–31, Jan. 1987.
[14] T. H. Lai and T. H. Yang, "On distributed snapshots," *Inform. Processing Lett.*, vol. 25, pp. 153–158, May 1987.
[15] B. M. Lampson, "Atomic transactions," in B. M. Lampson, M. Paul, and H. J. Siegert, Eds., *Distributed Systems: Architecture and Implementation*. New York: Springer-Verlag, 1981, pp. 246–264.
[16] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.* vol. 7, pp. 321–359, Nov. 1989.
[17] K. Li and J. F. Naughton, "Multiprocessor main memory transaction processing," in *Proc. Int. Symp. Database in Parallel Distrib. Syst.*, 1988, pp. 177–187.
[18] K. Li, J. F. Naughton, and J. S. Plank, "An efficient checkpointing method for multicomputers with wormhole routing." *Int. J. Parallel Processing*, vol. 20, no. 3, June 1992.
[19] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," in *Conf. Proc., Usenix Winter 1992 Tech. Conf.*, 1992, pp. 283–290.
[20] P. R. McJones and G. F. Swart, "Evolving the UNIX system interface to support multithreaded programs," Tech. Rep. 21, DEC Syst. Res. Center, Sept. 1987.
[21] J. K. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The sprite network operating system," *IEEE Comput.*, vol. 21, pp. 23–36, Feb. 1988.
[22] C. Pu, "On-the-fly, incremental, consistent reading of entire databases," in *Proc. 11th Int. Conf. Very Large Databases*, 1985, pp. 369–375.
[23] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 220–232, 1975.
[24] P. Rovner, R. Levin, and J. Wick, "On extending modula-2 for building large, integrated systems," Res. Rep. 3, DEC Syst. Res. Center, 1985.
[25] K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases," Tech. Rep. CS-TR-126-87, Dept. of Comput. Sci., Princeton Univ., 1987.
[26] A. Silberschatz and J. L. Peterson, *Operating Systems Concepts*. Reading, MA: Addison-Wesley, 1988.
[27] M. E. Staknis, "Sheaved memory: Architectural support for state saving and restoration in paged systems," in *3rd Int. Conf. Architectural Support Programming Languages and Operating Syst.*, 1989, pp. 96–103.
[28] D.J. Taylor and M. L. Wright, "Backward error recovery in a UNIX environment," in *16th Ann. Int. Symp. Fault-Tolerant Computing Syst.*, 1986, pp. 118–123.
[29] C. P. Thacker and L. C. Stewart, "Firefly: A multiprocessor workstation," in *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Syst.*, 1987, pp. 164–172.
[30] M.M. Theimer, K. A. Lantz, and D.R. Cheriton, "Preemptable remote execution facilities for the v-system," in *Proc. 10th ACM Symp. Operating Syst. Principles*, 1985, pp. 2–11.

Lower and Upper Bounds on Time for Multiprocessor Optimal Schedules

Kamal Kumar Jain and V. Rajaraman

***Abstract*—The lower and upper bounds on the minimum time needed to process a given directed acyclic task graph for a given number of processors are derived. It is proved that the proposed lower bound on time is not only sharper than the previously known values but also easier to calculate. The upper bound on time, which is useful in determining the worst case behavior of a given task graph, is presented for the first time. The lower and upper bounds on the minimum number of processors required to process a given task graph in the minimum possible time are also derived. It is seen with a number of randomly generated dense task graphs that the lower and upper bounds we derive are equal, thus giving the optimal time for scheduling directed acyclic task graphs on a given set of processors.**

***Index Terms*—Bounds on number of processors, bounds on time, multiprocessing, optimal scheduling, parallel processing, performance evaluation, scheduling directed acyclic task graphs**

I. Introduction

Most parallel algorithms can be modeled as acyclic task graphs. It is well known [13] that scheduling a directed acyclic graph on a parallel computer to obtain minimum possible execution time is an NP-complete problem. Thus, heuristics are used to arrive at "good" schedules [6]–[12].

In this paper, we propose lower and upper bounds on the minimum time (LBMT and UBMT) required when a given task graph is scheduled on a machine with a fixed number of processors. These bounds provide a good measure of parallelism [14]–[16] in algorithms and are also useful in comparing heuristics for the scheduling problem [4], [5]. If the predicted LBMT is achieved by some schedule, then we know that this schedule is the best. Further, if an intelligent heuristic schedule that attempts to keep all processors busy gives an execution time close to the UBMT, then it is likely that no other such heuristic schedule will perform worse than this.

The problem of determining the LBMT has been given a lot of importance in the area of scheduling for the last three decades. Hu [1] gave an LBMT for the case when the task times are equal and the graph is a rooted tree. His results were extended by Ramamoorthy *et al.* [2] for general task graphs with equal task times. Fernandez *et al.* [3] presented a new formulation and proposed sharper bounds. However, calculating the bounds presented in [2], [3] is time-consuming.

Although work in this area still continues [17]–[22], the reported bounds are not better than the ones proposed in [3]. In this paper, we propose a new lower bound on the minimum time needed to process a given task graph on a machine with a fixed number of processors. It is proved that the proposed bound is not only sharper than the previously known values but also computationally less intensive.

We also propose an upper bound on the minimum time needed to process a given task graph on a machine with a fixed number of processors. The UBMT, which is useful in the determination of the worst case behavior of a given task graph, has not been reported earlier.

Another problem of interest is to find the upper and lower bounds on the minimum number of processors (UBMP and LBMP) needed to

Manuscript received December 28, 1992; revised June 3, 1993.
The authors are with the Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012 India.
IEEE Log Number 9401216.

process a given task graph in the minimum possible time. Although good lower bounds have been reported in the literature [2], [3], calculating these bounds is computationally quite complex. It turns out that our method is more efficient in calculating the LBMP proposed in [3].

Bounds reported in [2], [3] are valid for single-entry single-exit connected (SEC) types of graphs only. Therefore, dummy tasks need to be added before these bounds can be applied. The approach presented in this paper does not require any such conversion.

This paper is divided into the following sections. Some of the important terms used in this paper are defined in Section II. The proposed algorithms used for determining the E-precedence and L-precedence partitions are also given in the same section. The proposed LBMT is derived in Section III. This new LBMT is compared with earlier results in Section IV. In Section V, the LBMT of an example task graph is calculated using the existing and the proposed methods. The proposed UBMT is derived in Section VI. Bounds on the minimum number of processors required to process a given task graph in the minimum possible time are given in Section VII. In Section VIII, we discuss the computational aspects of the proposed bounds. Section IX gives the performance of the proposed LBMT and UBMT on a number of randomly generated task graphs. We conclude the paper in Section X.

II. Definitions and Assumptions

A. Definitions

Task Graph: Given a set of tasks, $T = \{T_1, T_2, \cdots, T_n\}$, the tasks are to be executed by a set of identical processors. If the precedence order on T is denoted by $<$, and if the execution time of ith task T_i is denoted by t_i, then the partially ordered set $(T, <)$ may be described by a finite, acyclic digraph $G = (V, A)$, called the task graph, where V is a finite set of vertices of cardinality n, and A is a set of arcs represented as vertex pairs. The arcs in A describe the precedences. The above definition was proposed by Graham [5].

The minimum possible time taken to execute all the tasks of a task graph, with unlimited number of processors, is called its *critical time* and is represented by $t_{\rm cp}$. For the task graphs where each task is of unit execution time, critical time becomes equal to the height of the task graph.

Given a task graph G in which all tasks require one unit of processing time, we can partition the set T into a set of HG nonempty disjoint subsets, $E = \{E_1, E_2, \cdots, E_{\rm HG}\}$. These are called the *earliest precedence partitions* [2], where:

$$E = \left(\bigcup_{i=1}^{HG} E_i\right) = T.$$

Two tasks belong to the same partition if they have the same earliest completion time. The meaning of the E-precedence partition is as follows. E_1 is the subset of tasks that can be initiated and executed in parallel at the very start. E_2 is the subset of tasks that can be initiated and executed in parallel after their corresponding predecessor tasks in E_1 have been executed, and so on. Thus, the elements of E_i represent those tasks that can be processed at the earliest time corresponding to an index i. We refer to these tasks as belonging to E-level$_i$.

Similarly, we can partition T into a set of HG nonempty disjoint subsets, $L = \{L_1, L_2, \cdots, L_{HG}\}$. These are called the *latest precedence partitions* [2], where:

$$L = \left(\bigcup_{i=1}^{\rm HG} L_i\right) = T.$$

Two tasks belong to the same partition if they have the same latest completion time. L_i represents the subset of tasks that must be executed latest by the end of i. We refer to these tasks as belonging to L-level$_i$.

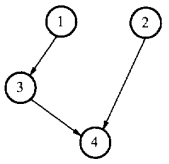

Fig. 1. Task graph.

Methods of finding these partitions presented in [2] require a number of matrix operations, and thus are complex to compute. We propose two algorithms to determine these partitions. These algorithms are simple to implement and do not require much computation.

Algorithm for E-Precedence Partitions:

```
for all tasks which do not have any
   predecessor
   E-level = 1;
for each task i which already has a E-level
assigned to it
 {
   for all its unassigned output tasks
     E-level =E-level of task i + 1;
   for all its assigned output tasks
     if E-level of output task is less than
     or equal to the E-level of task i
       E-level = E-level of task i + 1;
 }
```

The above algorithm gives subsets $E_1, E_2, \cdots, E_{HG}$ comprising the tasks at E-levels 1,2,$\cdots$,HG, respectively. For the task graph of Fig. 1, $E_1 = \{1, 2\}$, $E_2 = \{3\}$ and $E_3 = \{4\}$. Also, E-level (task 1) = E-level (task 2) = 1, E-level (task 3) = 2, and E-level (task 4) = 3. In this case, $HG = 3$.

Algorithm for L-Precedence Partitions:

```
for all tasks which do not have any
    successor
    L-level =HG;
for a task i which already has a L-level
    assigned to it
 {
    for all of its unassigned input tasks
      L-level = L-level of task i - 1;
    for all of its assigned input tasks
      if L-level of input task is greater
      than or equal to the L-level of task i
        L-level = L-level of task i - 1;
 }
```

The above algorithm gives subsets $L_1, L_2, \cdots, L_{HG}$ comprising the tasks at levels 1,2,$\cdots$,HG, respectively. For the task graph of Fig. 1, $L_1 = \{1\}$, $L_2 = \{2, 3\}$ and $L_3 = \{4\}$. Also, L-level (task 1) = 1, L-level (task 2) = L-level (task 3) = 2, and L-level (task 4) = 3.

Some More Definitions [3]: The *activity* of task T_j that finishes at time τ_j, $f(\tau_j, t)$ at time t is defined by the following expression:

$$f(\tau_j, t) = \begin{cases} 1, & \text{for } t \in [\tau_j - t_j, \tau_j], \\ 0, & \text{otherwise,} \end{cases}$$

where t_j is the execution time of task T_j.

The *load density function* of task graph G is the sum of all the task activities at time t and is defined by the following experssion:

$$F(\tau, t) = \sum_{j=1}^{n} f(\tau_j, t).$$

$f(\tau_j, t)$ indicates the activity of vertex j (or task T_j) along time, according to the restrictions imposed by the graph, and $F(\tau, t)$ indicates the total activity of the graph as a function of time. Note that τ_j can vary up to a certain value and depends on the starting time (or finishing time) of task T_j. A task T_j can be started at its earliest starting time or postponed, depending on the amount of time that it can be delayed. τ represents a particular combination of delays of all tasks. Of particular importance are the $F(\underline{\tau}, t)$, the *earliest load density function* for which all tasks are completed at their earliest times and $F(\overline{\tau}, t)$, the *latest load density function* for which all tasks are completed at their latest times. Also $\Phi(\tau, \theta_1, \theta_2)$ represents the load density function within time interval $[\theta_1, \theta_2] \subset [0, \omega]$ and is given by the following expression:

$$\Phi(\tau, \theta_1, \theta_2) = \int_{\theta_1}^{\theta_2} F(\tau, t)dt,$$

where ω is the least time in which all tasks of T have been completed. These definitions are useful in understanding the bounds given in [3].

B. Assumptions

The execution times for all of the tasks are assumed to be equal. However, this does not limit the scope of the proposed bounds. These bounds are valid for any general task graph in which the execution time of the tasks may not be equal, because such a task graph G can be converted into an equivalent task graph G' with equal execution times for all the tasks. Without loss of generality, integer execution times are considered in this paper.

We assume directed acyclic graphs with zero communication delays between tasks. Further, we also assume a nonpreemptive type of scheduling, wherein, once a processor begins executing a task, it cannot be interrupted until that task is completed.

Only the SEC type of task graphs have been considered in [3]. Such graphs have only one entry vertex, that is, a vertex with no predecessors and only one exit vertex, that is, a vertex with no successors, and the condition that every vertex is reachable from the entry vertex and reaches the exit vertex. For task graphs with multiple entry and/or exit tasks, some dummy vertices have to be added to convert them into SEC graphs. However, we found that this assumption is not at all necessary; therefore, bounds presented in this paper do not require any such conversion and are valid for all directed acyclic graphs.

III. Lower Bound on Time

Given a task graph, one would like to have a schedule that will execute the task graph on a parallel machine in the minimum possible time, called the *critical time* of that task graph. One can achieve this critical time when the machine has "enough" (roughly equal to the UBMP of the task graph) number of processors. We are interested in determining the minimum possible time (may not be the critical time) required to process a given task graph when the given number of processors is less than the UBMP of that task graph. This problem is NP-complete, because an optimal scheduling algorithm must be found to determine the time taken to execute the task graph. Instead of finding the minimum time, we propose a lower bound on the minimum time needed to execute a given task graph with the given number of processors in a parallel machine.

The best known LBMT is due to Fernandez *et al.* [3]. They consider a given integer interval $[\theta_1, \theta_2]$ within $[0, t_{cp}]$. Because tasks may be shifted within their completion intervals without making $\omega > t_{cp}$, we can move them so that they have a minimum possible overlap with $[\theta_1, \theta_2]$. $R(\theta_1, \theta_2, t)$ is called the load density function of the tasks or parts of tasks remaining within $[\theta_1, \theta_2]$ after all tasks have been shifted to form minimum overlap within the interval. The minimum increase in execution time over the critical path length when m identical processors are available is given by the following equation:

$$q_F = \max_{[\theta_1, \theta_2]} \left[-(\theta_2 - \theta_1) + \frac{1}{m} \int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2)dt \right]. \quad (1)$$

So, LBMT is given by the following equation:

$$t_L = t_{cp} + \lceil q_F \rceil. \quad (2)$$

$\lceil x \rceil$ represents the smallest integer not smaller than x. $\int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2)dt$ represents the activity of tasks that must be processed within this interval. $m \times (\theta_2 - \theta_1)$ represents the effective activity that can be executed with m processors. The excess area, divided by m, defines the increase in time over t_{cp}. The maximum over every interval is then the minimum time increase over t_{cp}.

Equation (1) can be rewritten as follows:

$$q_F = \max_{[\theta_1, \theta_2]} \left[-(\theta_2 - \theta_1) + \frac{1}{m} |\underline{\mathcal{F}} \cap \overline{\mathcal{F}}| \right], \quad (3)$$

where

$$\underline{\mathcal{F}} = f(\underline{\tau}, [\theta_1, \theta_2]),$$

and

$$\overline{\mathcal{F}} = f(\overline{\tau}, [\theta_1, \theta_2]).$$

The proof of (1) and equivalence of (1) and (2) are given in [3].

When all the task times are equal, (3) can be rewritten as follows:

$$q_F = \max_{0 \le t_k \le t_{cp}} \left[-t_k + \frac{1}{m} |\underline{\mathcal{F}} \cap \overline{\mathcal{F}}| \right]. \quad (4)$$

Equation (4) can be written in terms of E-precedence and L-precedence partitions as follows:

$$q_F = \max_{1 \le t_k \le t_{cp}} \left[-t_k + \frac{1}{m} \max_{1 \le j \le (t_{cp} - t_k + 1)} \left| \bigcup_{k=j}^{j+t_k-1} E_k \bigcap \bigcup_{k=j}^{j+t_k-1} L_k \right| \right]. \quad (5)$$

The equivalence of (4) and (5) will become clear in Section V, where an example is considered.

Proposed Bound: We propose the following three-step strategy to improve this bound.

1) Partition the given task graph G into independent task graphs.
2) Apply the bound represented by (2) and (5) to all of the smaller independent task graphs obtained from step (1).
3) Determine the new lower bound for the given task graph G from the bounds obtained in step (2).

It is necessary to introduce some more terms before we proceed. In what follows, tasks of a task graph may be assigned either E-level indices or L-level indices. Once this is done, we call the index a level.

Independent Levels: Two levels, i and $(i+1)$, of a task graph are said to be *independent levels* if $L^i \sim E^i = \phi$, where $L^i = \bigcup_{j=1}^{i} L_j$, $E^i = \bigcup_{j=1}^{i} E_j$, and $\sim$ is the difference operator. When two levels i and $(i+1)$ are independent, no task of the set L^i can be moved to a level greater than i.

Critical Level: Level i of a task graph is said to be a *critical level* if each task of the set L_i is connected to all the tasks of the set E_{i+1}. When a level i is a critical level then no task (or tasks) of the set E_{i+1} can be executed before all the tasks of the set L_i have been executed.

Independent Critical Level: A level i is said to be an *independent critical level* iff it is critical and is independent of level $(i+1)$.

Independent Task Graphs: Given a task graph G, it can be partitioned into a set of g *independent task graphs*, $G' = \{G_1, G_2, \cdots, G_g\}$, if we can identify $(g-1)$ independent critical levels $l_{k_1}, l_{k_2}, \cdots, l_{k_{g-1}}$ in G. G_1 is a task graph with the tasks of the set $\bigcup_{j=1}^{l_{k_1}} L_j$, G_i $(1 < i < g)$ is a task graph with the tasks of the set $\bigcup_{j=l_{k_{i-1}}+1}^{l_{k_i}} L_j$ and G_g is a task graph with the tasks of the set $\bigcup_{j=l_{k_{g-1}}+1}^{HG} L_j$. The precedence order and interconnection of these g task graphs are according to the precedence order of the given task graph G.

To partition a given task graph into a number of different independent task graphs, we first identify independent critical levels of the given task graph. After partitioning the given task graph G into g independent task graphs, we can apply the bounds given by (2) and (5) for each of the independent task graphs.

The new LBMT of a given task graph is given in Theorem 1.

Theorem 1: Given a task graph G partitioned into a set of g independent task graphs, $G' = \{G_1, G_2, \cdots, G_g\}$, the LBMT needed to process G using m identical processors is given by the following equation:

$$t_L = \sum_{i=1}^{g} t_{L_i}, \tag{6}$$

where

$$t_{L_i} = t_{\mathrm{cp}_i} + \lceil q_{F_i} \rceil.$$

t_{cp_i} is the critical time of the ith task graph G_i and q_{F_i} is given by (7) at the bottom of this page.

Proof: Each task graph in G' is an independent task graph. The precedence order of these task graphs is $G_1 < G_2 < \cdots < G_{g-1} < G_g$, which means the execution of G_2 cannot begin before the completion of G_1, and so on. So, for each of the task graphs in G', the LBMT can be determined according to (2) and (5) independently, and the LBMT for the original task graph G can thus be obtained by using (6). □

IV. Comparison of Lower Bounds on Time

Hu [1] gave a lower bound on the minimum time needed to process a task graph G, given a fixed number of processors, say, m. His bound, valid only for a rooted tree with equal task times is given by the following equation:

$$t_L = t_{\mathrm{cp}} + \lceil q_H \rceil,$$

where

$$q_H = \max_{\gamma} \left[-\gamma + \frac{1}{m} \sum_{j=1}^{\gamma} |L_j| \right].$$

γ denotes the level, and L_j stands for the latest precedence partition for level j.

Ramamoorthy *et al.* [2] presented the bound for a general task graph with equal task times, which is given by the following equation:

$$t_L = t_{\mathrm{cp}} + \lceil q_R \rceil,$$

where

$$q_R = \max_{\gamma} \left[-\gamma + \frac{1}{m} \sum_{j=1}^{\gamma} |L_j \cap E_j| \right].$$

E_j stands for the earliest precedence partition for level j; the meanings of γ and L_j have already been explained.

Fernandez *et al.* [3] improved the bound proposed by Ramamoorthy *et al.* This bound is given by (2) and (5). The new bound proposed in Theorem 1 can be rewritten as below:

$$t_L = t_{\mathrm{cp}} + q_{\mathrm{new}}, \tag{8}$$

where

$$q_{\mathrm{new}} = \sum_{i=1}^{g} \lceil q_{F_i} \rceil.$$

The above bound is better than all the previously known bounds. This is proved in Lemma 1.

Lemma 1: For a given task graph G, the lower bound on minimum execution time given by (7) and (8) is sharper than that given by (2) and (5), that is, $q_{new} \geq \lceil q_F \rceil$.

Proof: From (8), $q_{\mathrm{new}} = \sum_{i=1}^{g} \lceil q_{F_i} \rceil$. q_F and q_{F_i} are given by (5) and (7), respectively. From the partition procedure of a given task graph G into g independent task graphs, described earlier, in Section III, it is clear that $q_F = \sum_{i=1}^{g} q_{F_i}$. If some of the q_{F_i}'s are not integers, then we might get $\lceil q_F \rceil \leq \lceil q_{F_i} \rceil$. So, $q_{\mathrm{new}} \geq \lceil q_F \rceil$. This shows that the bound given by (7) and (8) is sharper than that given by (2) and (5). □

Lemma 1 shows that our lower bound on the minimum time is higher than earlier bounds; that is, our lower bound is closer to the optimal time than the earlier bounds. This result will help in comparing heuristic scheduling algorithms. The bounds proposed in this paper are better than the earlier bounds, mainly because of our partitioning of task graphs into a number of independent task graphs, as explained below.

Superiority Over q_F: The superiority of the new bound over the one proposed earlier in [3] can be explained when the worst and best cases of partitioning the given task graph are considered. In the worst case, the given task graph G cannot be partitioned at all, and we get $q_{\mathrm{new}} = q_F$. On the other hand, the best case occurs when the following four conditions are met.

1) The given graph G can be partitioned into a set of HG independent task graphs, $G' = \{G_1, G_2, \cdots, G_{HG}.\}$
2) All q_{F_i}'s are equal, $\forall i = 1, \cdots, HG$.
3) None of the $q_{F_i}s$ is an integer, $\forall i = 1, \cdots, HG$.
4) $\lceil q_F \rceil = \lceil q_{F_i} \rceil$, $\forall i = 1, \cdots, HG$.

Conditions (1)–(4) yield $q_{\mathrm{new}} = \lceil q_F \rceil + HG - 1$. Considering both the worst and the best cases, we arrive at the following important expression:

$$\lceil q_F \rceil \leq \lceil q_{new} \rceil \leq \lceil q_F \rceil + HG - 1. \tag{9}$$

Next we give a simple example to illustrate the sharpness of our bound in comparison with the previously known bounds.

V. Example

Consider the task graph of Fig. 2, to be scheduled on a parallel machine with two processors ($m = 2$). We compare our bound with the lower bounds given by Hu, Ramamoorthy *et al.* and Fernandez *et al.*

$$q_{F_i} = 1 \leq t_k \leq t_{cp_i} max \left[-t_k + \frac{1}{m} 1 \leq j \leq (t_{cp_i} - t_k + 1) max \left| \bigcup_{k=j}^{j+t_k-1} E_k \bigcap \bigcup_{k=j}^{j+t_k-1} L_k \right| \right] \tag{7}$$

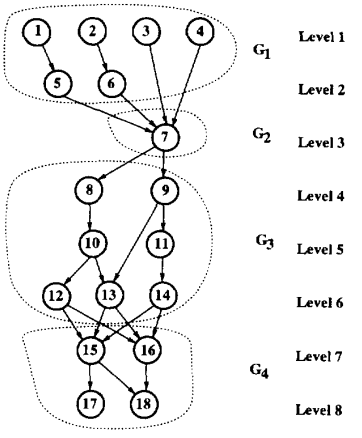

Fig. 2. Example task graph.

For the given task graph, we have the following:

$E_1 = \{1,2,3,4\}, L_1 = \{1,2\}, E_2 = \{5,6\}, L_2 = \{3,4,5,6\},$

$E_3 = L_3 = \{7\}, E_4 = L_4 = \{8,9\}, E_5 = L_5 = \{10,11\},$

$E_6 = L_6 = \{12,13,14\}, E_7 = L_7 = \{15,16\},$

$E_8 = L_8 = \{17,18\},$

and the critical time $t_{cp} = 8$ (assuming unit execution time).

Extension of Hu's Bound: The method of determining the Hu's bound for a general task graph is given in [3]. Here $t_L = t_{cp} + \lceil q_H \rceil$, where q_H is as given in Section IV. Accordingly, $q_H = \max(0,1,0.5,0.5,0.5,1,1,0) = 1$, which yields $t_L = 8 + 1 = 9$.

Ramamoorthy's Bound: Here $t_L = t_{cp} + \lceil q_R \rceil$, where q_R is as given in Section IV. Thus, $q_R = \max(0,1,0.5,0.5,0.5,1,1,0) = 1$, giving $t_L = 8 + 1 = 9$.

Fernandez's Bound: Here $t_L = t_{cp} + \lceil q_F \rceil$, where q_F is given by (5). In this case, $q_F = \max(0.5,1,0.5,0.5,0.5,1,1,0) = 1$, which again gives $t_L = 8 + 1 = 9$.

The New Bound: The new bound is given by Theorem 1. In this example task graph, levels 2, 3, 4, 5, 6, and 7 are independent, and levels 2, 3, and 6 are critical. Therefore, the example task graph can be partitioned into four independent task graphs, G_1, G_2, G_3, and G_4, as shown in Fig. 2. Also note that $t_{cp_1} = 2$, $t_{cp_2} = 1$, $t_{cp_3} = 3$, and $t_{cp_4} = 2$. In this case, the LBMT is determined by using (6) and (7).

Thus, we have the following equalities:

$$q_{F_1} = \max(0,1) = 1$$

$$q_{F_2} = 0$$

$$q_{F_3} = \max(0.5,0.5,0.5) = 0.5$$

$$q_{F_4} = 0.$$

Therefore, we have:

$$t_{L_1} = 2 + 1 = 3$$

$$t_{L_2} = 1 + 0 = 1$$

$$t_{L_3} = 3 + 1 = 4$$

$$t_{L_4} = 2 + 0 = 2,$$

and the lower bound on time is given by the following equation:

$$t_L = t_{L_1} + t_{L_2} + t_{L_3} + t_{L_4} = 3 + 1 + 4 + 2 = 10.$$

It is clear from this example that the new LBMT will perform even better for a larger task graph.

TABLE I
CHARACTERISTICS OF TASK GRAPHS (SET 1)

Task Graph	Nodes	Depth	Width	Critical Time
1	6	3	2	3
2	10	3	4	3
3	26	8	6	8
4	44	8	8	8
5	46	8	8	8
6	48	8	8	8
7	77	13	13	13
8	76	12	10	12
9	69	12	11	12
10	87	15	15	15
11	114	17	14	17
12	150	19	18	19
13	290	21	21	21
14	278	23	22	23
15	234	25	25	25
16	251	25	21	25
17	360	28	26	28
18	471	30	29	30
19	521	31	31	31
20	439	30	30	30

VI. UPPER BOUND ON TIME

In this section, we propose an upper bound on the minimum time needed to process a given task graph G with m identical processors. An upper bound on the minimum time is useful in comparing the performance of scheduling heuristics and in determining the worst case behavior of a task graph. Such a bound is not found in the literature. The analysis presented in this section assumes E-precedence type of task graphs.

First, a special case is considered, where the precedence order of tasks is such that the tasks need to be processed level-by-level only. It is obvious that for such a case, the UBMT with m processors is given by the following equation:

$$T_U = \sum_{i=1}^{HG} \left\lceil \frac{|E_i|}{m} \right\rceil. \tag{10}$$

However, this is a very loose bound if a more general case is considered. We improve upon this bound by identifying the minimally connected levels in the E-precedence type of task graphs which are defined as below.

Minimally Connected Levels: Two levels, i and $(i+1)$, of a given task graph G, are said to be *minimally connected levels* if the number of edges between them is equal to the number of tasks at level $(i+1)$, provided that the number of processors in the parallel machine is not greater than the number of tasks at level i.

If two levels i and $(i+1)$ are minimally connected, then these two levels can be merged to get a better UBMT, which is proved later in Theorem 2. In general, given a task graph G, we can find a two dimensional set, $\{R,S\} = \{(r_1,s_1),(r_2,s_2),\cdots,(r_p,s_p)\}$, where r_i is the level number of the task graph G and s_i is an integer that denotes the maximum number of levels, after the level r_i, which can be merged with level r_i and have not been merged with any other earlier level. Note that $s_i = 0$ indicates that the level r_i cannot be merged with the level $r_i + 1$ or $r_i - 1$.

Theorem 2: The upper bound on the minimum time required to process a given task graph G using a parallel machine with m processors is given by the following equation:

$$T_U = \sum_{j=1}^{p} \left\lceil \frac{1}{m} \sum_{i=r_j}^{r_j+s_j} |E_i| \right\rceil. \tag{11}$$

Proof: Let i and $(i+1)$ be the minimally connected levels. As explained earlier, these two levels can be merged. Accordingly,

TABLE II
PERFORMANCE OF LBMT AND UBMT (SET 1)

Task Graph	m = 2			m = 4			m = 8			m = 16			No. of Independent Graphs
	LBMT	UBMT	t_F	LBMT	UBMT	t_F	LBMT	UBMT	t_F	LBMT	UBMT	t_F	
1	3	3	3	3	3	3	3	3	3	3	3	3	2
2	5	5	5	3	3	3	3	3	3	3	3	3	1
3	13	14	13	8	9	8	8	8	8	8	8	8	2
4	22	23	22	12	13	12	8	8	8	8	8	8	3
5	23	24	23	12	15	12	8	8	8	8	8	8	1
6	24	26	24	12	14	12	8	8	8	8	8	8	1
7	39	43	39	20	24	20	14	15	14	13	13	13	1
8	38	40	38	19	22	19	13	15	13	12	12	12	1
9	35	36	35	18	20	18	12	13	12	12	12	12	1
10	44	48	44	23	27	**22**	16	18	16	15	15	15	2
11	58	60	**57**	31	33	**30**	18	21	18	17	17	17	3
12	75	79	75	38	44	38	22	26	22	19	21	19	1
13	145	149	145	73	79	73	37	44	37	22	29	22	1
14	139	142	139	70	75	70	35	44	35	24	32	24	1
15	117	121	117	59	66	59	30	39	30	26	29	26	1
16	126	132	126	63	72	63	32	40	32	25	28	25	1
17	180	187	180	90	99	90	46	55	46	29	37	29	1
18	236	242	236	118	128	118	59	70	59	33	45	33	1
19	261	267	261	131	141	131	66	77	66	35	47	35	1
20	220	226	220	110	121	110	55	66	55	31	39	31	1

t_F: LBMT due to Fernandez

the term $\left\lceil \frac{|E_i|}{m} \right\rceil + \left\lceil \frac{|E_{i+1}|}{m} \right\rceil$ in (10) gets replaced by $\left\lceil \frac{|E_i|+|E_{i+1}|}{m} \right\rceil$. In general, if we can find a 2-D set $\{R, S\}$, defined earlier, we get the following equation:

$$T_U = \sum_{j=1}^{p} \left\lceil \frac{1}{m} \sum_{i=r_j}^{r_j+s_j} |E_i| \right\rceil.$$

Clearly, (11) represents a better UBMT than (10). □

VII. BOUNDS ON THE NUMBER OF PROCESSORS

In this section, we present bounds on the minimum number of processors required to process a given task graph G in the minimum possible time. These bounds have also been reported earlier, in [3]. However, calculating the LBMP is computationally intensive. It is observed that the new approach of partitioning the given graph into a number of independent task graphs, introduced in this paper, reduces the time required to calculate this bound.

According to Fernandez *et al.*, the LBMP is given by the following equation:

$$m_L = \left\lceil \max_{[\theta_1,\theta_2]} \left[\frac{1}{\theta_2 - \theta_1} \int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2, t)dt \right] \right\rceil, \tag{12}$$

which can be written as follows:

$$m_L = \left\lceil \max_{[\theta_1,\theta_2]} \left[\frac{1}{\theta_2 - \theta_1} \left| \underline{\mathcal{F}} \cap \overline{\mathcal{F}} \right| \right] \right\rceil. \tag{13}$$

When all task times are equal, (13) can be rewritten as follows:

$$m_L = \left\lceil \max_{0 \le t_k \le t_{cp}} \left[\frac{1}{t_k} \left| \underline{\mathcal{F}} \cap \overline{\mathcal{F}} \right| \right] \right\rceil. \tag{14}$$

Equation (14) can also be written in the terms of E-precedence and L-precedence partitions as below:

$$m_L = \left\lceil \max_{1 \le t_k \le t_{cp}} \left[\frac{1}{t_k} \max_{1 \le j \le (t_{cp} - t_k + 1)} \left| \bigcup_{k=j}^{j+t_k-1} E_k \bigcap \bigcup_{k=j}^{j+t_k-1} L_k \right| \right] \right\rceil. \tag{15}$$

When the given task graph G is partitioned into a set of g independent task graphs, $G' = \{G_1, G_2, \cdots, G_g\}$, the LBMP needed to execute the given task graph G in the minimum possible time is given by the following:

$$m_L = \max \left(m_{L_1}, m_{L_2}, \cdots, m_{L_g} \right), \tag{16}$$

where

$$m_{L_i} = \left\lceil \max_{1 \le t_k \le t_{cp_i}} \left[\frac{1}{t_k} \max_{1 \le j \le (t_{cp_i} - t_k + 1)} \left| \bigcup_{k=j}^{j+t_k-1} E_k \bigcap \bigcup_{k=j}^{j+t_k-1} L_k \right| \right] \right\rceil. \tag{17}$$

It must be pointed out that although the value of m_L obtained from (16) and (17) is the same as the one obtained from (15), this method is computationally more economical than the earlier one. This is explained in Section VIII.

For the sake of completeness, the upper bound on the minimum number of processors required to process a given task graph in the minimum possible time is given by the following:

$$m_U = \min \{ F_{\max}(\underline{\tau}, t), F_{\max}(\overline{\tau}, t) \}. \tag{18}$$

More details can be found in [3]. Equation (18) can be rewritten as

TABLE III
CHARACTERISTICS OF TASK GRAPHS (SET 2)

Task Graph	Nodes	Depth	Width	Critical Time
1	4	2	2	2
2	8	2	4	2
3	13	4	4	4
4	18	3	6	3
5	60	8	16	8
6	64	2	32	2
7	72	4	32	4
8	78	13	16	13
9	78	10	16	10
10	87	8	32	8
11	105	12	32	12
12	111	10	32	10
13	173	15	32	15
14	212	17	32	17
15	249	20	32	20
16	302	22	32	22
17	362	24	32	24
18	407	26	32	26
19	412	28	32	28
20	491	30	32	30

TABLE IV
PERFORMANCE OF LBMT AND UBMT (SET 2)

Task Graph	m = 2			m = 4			m = 8			m = 16			No. of Independent Graphs
	LBMT	UBMT	t_F	LBMT	UBMT	t_F	LBMT	UBMT	t_F	LBMT	UBMT	t_F	
1	2	2	2	2	2	2	2	2	2	2	2	2	2
2	4	4	4	2	2	2	2	2	2	2	2	2	2
3	7	7	7	4	4	4	4	4	4	4	4	4	3
4	9	9	9	5	5	5	3	3	3	3	3	3	2
5	32	32	30	18	18	15	10	10	9	8	8	8	5
6	32	32	32	16	16	16	8	8	8	4	4	4	2
7	36	36	36	18	18	18	10	10	9	6	6	5	2
8	43	43	39	24	24	20	17	17	15	13	13	13	12
9	41	41	39	22	22	20	13	13	11	10	10	10	5
10	45	45	44	25	25	22	14	14	11	10	10	9	6
11	55	55	53	31	31	27	20	20	16	14	14	13	9
12	58	58	56	32	32	28	19	19	14	12	12	11	9
13	89	89	87	48	48	44	26	26	22	17	17	16	8
14	108	108	106	57	57	53	32	32	27	20	20	18	13
15	127	127	125	67	67	63	36	36	32	26	26	21	10
16	155	155	151	81	81	76	45	45	38	28	28	24	12
17	186	186	181	98	98	91	54	54	46	33	33	26	13
18	207	207	204	108	108	102	59	59	51	36	36	28	13
19	208	208	206	108	108	103	61	61	52	39	39	30	15
20	249	249	246	129	129	123	72	72	62	43	43	32	15

t_F: LBMT due to Fernandez

shown below:

$$m_U = \min\{\max(E_1, E_2, \cdots, E_{HG}), \max(L_1, L_2, \cdots, L_{HG})\}. \tag{19}$$

VIII. COMPUTATIONAL ASPECTS

The LBMT and LBMP proposed by Fernandez *et al.* require the processing of $\frac{t_{\rm cp}(t_{\rm cp}+1)}{2}$ intervals. Therefore, their computational complexity is proportional to $t_{\rm cp}^2$. This limits their practical usefulness. Furthermore, set theoretic operations are slow in comparison with arithmetic operations. Therefore, it is useful to reduce the processing time of these methods. Our approach not only reduces the computing time but also gives a better value of the LBMT. Although the latter part has already been proved in Lemma 1, the former is proved next in Lemma 2.

Lemma 2: The processing requirement for (7) and (8) is less than those of (2) and (5).

Proof: If a given a task graph G can be partitioned into a set of g independent task graphs, $G' = \{G_1, G_2, \cdots, G_g\}$, with critical times $t_{cp_1}, t_{cp_2}, \cdots, t_{cp_g}$, respectively, then the processing requirement is roughly proportional to $\sum_{i=1}^{g} \frac{t_{cp_i}(t_{cp_i}+1)}{2}$. Let T_p be the processing time required to partition the task graph. Then the total processing requirement for (6) is given by the following equation:

$$T_p + \sum_{i=1}^{g} \frac{t_{cp_i}(t_{cp_i}+1)}{2}.$$

The information required to partition a given task graph can be acquired at almost negligible computational cost if it is acquired at the time of storing the task graph as a data structure and finding E-precedence and L-precedence partitions. Therefore, it is reasonable to neglect $O(T_p)$ from this analysis. This reduces to proving the following equation:

$$O\left(\sum_{i=1}^{g} \frac{t_{cp_i}(t_{cp_i}+1)}{2}\right) \le O\left(\frac{t_{\rm cp}(t_{\rm cp}+1)}{2}\right).$$

In the worst case, $g = 1$; that is, we cannot partition the given task graph at all. Therefore, $\frac{t_{cp_1}(t_{cp_1}+1)}{2} = \frac{t_{\rm cp}(t_{\rm cp}+1)}{2}$, because $t_{cp_1} = t_{\rm cp}$.

For the best case, on the other hand, $g = HG$, and $t_{cp_i} = 1$ $\forall i = 1, \cdots, HG$. This gives $\sum_{i=1}^{g} \frac{t_{cp_i}(t_{cp_i}+1)}{2} = HG$, which shows that computational requirement roughly grows with HG or $t_{\rm cp}$ (and not with $t_{\rm cp}^2$). □

IX. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed LBMT and UBMT with the help of 20 randomly generated task graphs, whose characteristics are given in Table I. A program was developed to generate these random graphs. The values of LBMT and UBMT for different number of processors for these task graphs are given in Table II. Another set of 20 dense task graphs, whose characteristics are given in Table III, is also considered for comparing the results. Values of the proposed LBMT and UBMT, along with the LBMT due to Fernandez for this set of task graphs, are given in Table IV.

It can be seen from Table II that our lower bound is better than the Fernandez bound in only a few cases. The reason is that the task graphs of set 1 are not dense enough to be partitioned. It can also be observed from Table II that the values of LBMT and UBMT are quite close to each other for a fixed number of processors. It is desirable to have LBMT and UBMT as close to each other as possible, because as these two values converge to a common value, the exact time needed to schedule a task graph is obtained. Table IV shows that both LBMT and UBMT have the same values. Thus, an exact time needed to schedule the given task graph is found.

X. CONCLUSION

Earlier lower bounds to find the minimum time required to process a given acyclic task graph on a parallel computer with m processors did not examine the possibility of partitioning a task graph into a set of independent task graphs. Such a partitioning is possible, and, using this partitioning, a new lower bound can be obtained which is better. The time to calculate the bound is also reduced. Calculation of upper bound was not attempted earlier. It is seen that for many large, densely connected task graphs, the upper and lower bounds are equal. This gives the exact time required to execute an acyclic task graph on a m processor machine using an intelligent heuristic schedule that attempts to keep all the processors in a parallel computer busy.

REFERENCES

[1] T. C. Hu, "Parallel sequencing and assembly line problems," *Oper. Res.*, vol. 9, pp. 841–848, Nov. 1961.

[2] C. V. Ramamoorthy, K. M. Chandy and M. J. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, no. 2, pp. 137–146, Feb. 1972.
[3] E. B. Fernandez and B. Bussell, "Bounds on the number of processors and time for multiprocessors optimal schedules," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 745–751, Aug. 1973.
[4] E. G. Coffman *et al.*, Eds., *Computer and Job-Shop Scheduling Theory.* New York: Wiley, 1976.
[5] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Maths.*, vol. 17, pp. 416–429, 1969.
[6] D. Helmbold and R. Mayr, "Two processor scheduling is in NC," *SIAM J. Computing*, vol. 16, no. 4, pp. 747–759, Aug. 1987.
[7] E. Coffman and R. Graham, "Optimal scheduling for two processor systems," *Acta Informatica*, vol. 1, pp. 200–213, 1972.
[8] T. Kawaguchi and S. Kyan, "Worst case bound on an LRF schedule for the mean weighted flow-time problem," *SIAM J. Computing*, vol. 15, no. 4, pp. 1119–1129, Nov. 1986.
[9] D. Dolev and M. Warmuth, "Scheduling flat graphs," *SIAM J. Computing*, vol. 14, no. 3, pp. 638–657, Aug. 1985.
[10] D. Dolev and M. Warmuth, "Scheduling precedence graphs of bounded height," *J. Algorithms*, vol. 5, pp. 48–59, 1984.
[11] H. N. Gabow, "An almost linear algorithm for two processor scheduling," *J. ACM*, vol. 29, no. 3, pp. 766–780, 1982.
[12] C. S. R. Murthy and V. Rajaraman, "Task assignment in a multiprocessor system," *Microprocessing and Microprogramming: Euromicro J.*, vol. 26, pp. 63–71, 1989.
[13] J. D. Ullman, "NP-complete scheduling problems," *J. Comput. Sys. Sci.*, vol. 10, pp. 384–393, 1975.
[14] D. J. Kuck *et al.*, "Measurements of parallelism in ordinary Fortran program," *Comput.*, vol. 7, no. 1, pp. 37–46, 1974.
[15] B. Jereb and L. Pipan, "Measuring parallelism in algorithms," *Microprocessing and Microprogramming: Euromicro J.*, vol. 34, pp. 49–52, 1992.
[16] K. K. Jain and V. Rajaraman, "Parallelism measures of task graphs on multiprocessors," *Microprocessing and Microprogramming*, to appear.
[17] K. So *et al.*, "A speedup analyzer for parallel programs," in *Proc. ICPP*, 1987, pp. 653–661.
[18] B. P. Lester, "A system for computing the speedup of parallel programs," in *Proc. ICPP*, 1986, pp. 145–152.
[19] C. Polychronopoulos and U. Banerjee, "Speedup bounds and processors allocation for parallel programs on multiprocessors," in *Proc. ICPP*, 1986, pp. 961–968.
[20] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proc. 3rd Supercomputing Conf.*, 1990, pp. 324–333.
[21] N. L. Soong, "Performance bounds for a certain class of parallel processing," in *Proc. ICPP*, 1979, p. 115.
[22] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. Comput.*, vol. 38, pp. 408–422, March 1989.

An Equivalence Theorem for Labeled Marked Graphs

Yaron Wolfsthal and Michael Yoeli

***Abstract*—Petri nets and their languages are a useful model of systems exhibiting concurrent behavior. The *sequential language* associated with a given Petri net S consists of all possible firing sequences of S, where each element of a firing sequence is a single transition. The *concurrent language* associated with S consists of all possible concurrent firing sequences of S, where each element of a concurrent firing sequence is a set of transitions. The sequential language and the concurrent language associated with S are denoted by $(L)(S)$ and $(\pi)(S)$, respectively. In this paper, we consider an important special case of Petri nets, called *labeled marked graphs*. The main result derived in this paper states that if Γ_1 and Γ_2 are two structurally deterministic labeled marked graphs, then $(L)(\Gamma_1) = L(\Gamma_2) \Leftrightarrow \pi(\Gamma_1) = \pi(\Gamma_2)$.**

***Index Terms*— Concurrency, marked graphs, Petri nets, structural determinism**

I. Introduction

Petri nets and their languages are a useful model of systems exhibiting concurrent behavior (cf. [1]–[4]). The occurrence of an event in the system represented by a Petri net S is modeled by the *firing* of the corresponding *transition* in S. A sequence of events in the system is modeled by a *firing sequence* in S. As suggested in [5], a (single) step of a Petri net can be naturally viewed as the concurrent firing of a set of transitions, rather than the firing of a single transition. This idea naturally extends to defining a *concurrent firing sequence* to be a string of sets of transitions, each of which consists of some concurrently firable transitions. The *sequential language* associated with a Petri net S consists of all possible firing sequences of S. The *concurrent language* associated with S consists of all possible concurrent firing sequences of S. The sequential language and the concurrent language associated with S are denoted by $L(S)$ and $\pi(S)$, respectively.

In this short note, we consider an important special case of Petri nets, called *labeled marked graphs* [6]. Labeled marked graphs generalize the useful concept of marked graphs [7]–[10] by allowing different transitions to have identical labels. In particular, we consider *structurally deterministic* (s.d.) labeled marked graphs [11]. Informally, a labeled marked graph is *s.d.* if, at any given time, no two transitions with identical labels are firable. The main result derived in this paper states that if Γ_1 and Γ_2 are two s.d. labeled marked graphs, then $L(\Gamma_1) = L(\Gamma_2) \Leftrightarrow \pi(\Gamma_1) = \pi(\Gamma_2)$. Put another way, two s.d. labeled marked graphs have identical concurrent languages if and only if they have identical sequential languages. We also show, on the other hand, that the result does not hold if the structural determinism requirement is omitted.

II. Definitions

In Definitions 1–10 below, we formally define marked graphs and the sequential and concurrent languages associated with them.

Definition 1: A *Petri net* is a 4-tuple $S = (P, T, V, M)$, where

1) P and T are finite sets of *places* and *transitions*, respectively.
2) $P \cap T = \emptyset$, $P \cup T \neq \emptyset$.
3) V is a function $V: (P \times T) \cup (T \times P) \rightarrow \{0, 1\}$.
4) M is a *marking function* (abbreviated *marking*) $M: P \rightarrow \omega$, where ω denotes the set of non-negative integers.

Definition 2: A Petri net $S = (P, T, V, M)$ is a *marked graph* [7]–[10] if for each $p \in P$, $\sum_{t \in T} V(p, t) = \sum_{t \in T} V(t, p) = 1$. For a marked graph $S = (P, T, V, M)$ and a place $p \in P$, the transitions t, t' that satisfy $V(t, p) = 1$ and $V(p, t') = 1$ are called the *input transition* and the *output transition*, respectively, of p. If $V(p, t) = 1$ $[V(t, p) = 1]$ for some $p \in P$ and $t \in T$, we say that p is an *input place* [*output place*] of t.

It is useful to depict marked graphs according to the common graphical representation of Petri nets (cf. [1]), where

1) places are represented by circles;
2) transitions are represented by bars;
3) the function V is represented by directed arcs in the obvious way; and

Manuscript received May 1, 1992; revised September 3, 1993.

Y. Wolfstahl is with MATAM — Advanced Technology Centers, IBM Haifa Research Group, Haifa, Israel 31905; e-mail: yaron@vnet.IBM.COM.
M. Yoeli is with MATAM — Advanced Technology Centers, IBM Haifa Research Group, Haifa, Israel 31905, and the Department of Computer Science, Technion — Israel Institute of Technology, Haifa, Israel.

IEEE Log Number 9401381.