Vol. 37 No. 4 Transactions of Information Processing Society of Japan Apr. 1996

Regular Paper

An Improved Increase over the Minimum Execution Time of a Parallel Program

DINGCHAO LI,† YUJI IWAHORI† and NAOHIRO ISHII††

In this paper we are concerned with lower bounds on the minimum time required to execute a parallel program on a homogeneous multiprocessor. We present an increase analysis method to tighten the existing lower bounds, using the results we have presented elsewhere. The new method is based on the propagation analysis of time delays along the critical paths of a given program. An illustrative example and theoretical analysis are provided to demonstrate the effectiveness of the proposed technique.

1. Introduction

The goal of multiprocessor scheduling is to spread the load to all processors as evenly as possible in order to make processors work efficiently, which will lead to shorter overall processing time. Depending on when the scheduling is done, scheduling strategies can be divided into two major classes: static and dynamic. Since the dynamic scheduling needs the hardware to check for dependences among instructions and build sophisticated heuristics in the hardware to select independent instructions, some current machines are designed with the requirement that instructions are scheduled at compile time using only static information. The VLIW (Very Long Instruction Word) architecture[1)] is one of the most successful examples.

However, although a static scheme is beneficial, finding the best schedule at compiler time is extremely complex and is known to be NP-hard in the strong sense[2)]. As a result, considerable research in the area has to concentrate on the development of efficient heuristic algorithms. These heuristics yield good solutions within polynomial time for restricted classes of systems and applications[3)−9)], but do not guarantee optimality. This naturally brings up the problem of how to evaluate their performance. To solve this problem, Adam et al. quantified the differences in five different list scheduling algorithms by running them on a large number of sample problems[3)], and Khan et al. compared the relative performance of several promising scheduling techniques in the same way[4)]. However, it is in general very difficult for this approach to indicate the goodness of the schedules in terms of how close they come to an optimum solution. This is because the exhaustive search for optimum solutions is usually impractical and prohibitively expensive[3),11),14)], especially when programs are large and complex. An alternative theoretical approach for overcoming the difficulty is to evaluate them using lower bounds on the minimum time (the length of optimal schedules) as absolute performance measures [3),10)−15)]. Therefore, for this reason the development of tight lower time bounds has become an important research direction in multiprocessor scheduling theory.

Another very good reason for the development is that the bounds are particularly useful in the branch-and-bound algorithms, where they are used by one of the elimination rules to prune the search tree. For example, Kasahara and Narita developed a depth first/implicit heuristic search algorithm (DF/IHS) based on the branch-and-bound method and the critical path method[7)], and Chen et al. proposed a state-space search algorithm (A^*) coupled with a heuristic derived from the Fernandez and Bussell bound to solve the multiprocessor scheduling problem[8)]. The efficiency of such algorithms clearly depends on the "sharpness" of the bounds incorporated in them, and hence tight lower time bounds is essential in order to effectively guide the search for a near-optimal solution[6)].

This paper addresses the problem of finding lower time bounds for homogeneous multiprocessor optimal schedules. The best bound known today is due to Fernandez and Bussell[10)]. Using the basic concepts described

† Educational Center for Information Processing, Nagoya Institute of Technology
†† Department of Intelligence and Computer Science, Nagoya Institute of Technology

in this bound, Al-Mouhamed presented a lower time bound for the case where communication time between tasks is not negligible compared to the task execution time[11], and Li et al. presented lower time bounds for precedence graphs with different types of tasks on the assumption that communication time between tasks is negligible or not[12],[13]. In this paper, we attempt to provide a new theoretical framework for tightening the existing bounds. More specifically, we will present improvements over the increase computation method suggested by Fernandez and Bussell, using some results of the earlier studies[12],[14]. The key idea behind our method is to analyze the propagation delays associated with tasks on the critical paths of a given program. Delaying such tasks is equivalent to lengthening the total program's execution time. Therefore, the analysis result naturally leads to an increase over the minimum execution time of the program, which is always sharper than the known value.

The rest of this paper is organized as follows. The next section starts with some preliminary definitions and assumptions about the underlying model. Section 3 briefly describes the existing increase analysis methods, and then shows how to improve them to obtain a sharper increase over the minimum execution time. Section 4 analyzes the theoretical performance of the method proposed in this paper. It also provides an example to illustrate how the method is applied to compute the increase. And finally, Section 5 contains our conclusions.

2. Definitions and Assumptions

The basis of our program model is the traditional task graph, as shown in Ref. 10). To establish a starting place for more realistic lower bound computations, in this paper we neglect message communication costs among tasks. A task graph without communication costs is generally defined to be a finite directed acyclic graph $G(\Gamma, \mathcal{A}, \mu)$ with a finite nonempty set Γ of vertices, a finite set $\mathcal{A}$ of directed arcs and a time function μ. The vertex set $\Gamma = \{T_1, T_2, \cdots, T_n\}$ consists of n tasks, each of which is an indivisible unit of computation and is executed on a machine P with m identical processors. The arc $(T_i, T_j) \in \mathcal{A}$ defines the dependence constraint (partial ordering) between tasks T_i and T_j; i.e., it implies that the execution of T_i must be completed before the execution of T_j can be started. Associated with each task T_i is a nonnegative number $\mu(T_i)$ which represents the required execution time of the task on any of the m processors.

If there exists an arc (or a path) from T_i to T_j in a graph G, we will say that T_i is a predecessor of T_j, and T_j is a successor of T_i. For convenience, throughout this paper we use $Pred(T_i)$ to represent the set of predecessors of T_i, and $Succ(T_i)$ to represent the set of successors of T_i. Let an entry task be one with no predecessors, and an exit task be one with no successors. We assume without loss of generality that G has exactly one entry task T_1 and exactly one exit task T_n. Thus, a critical path in the graph can be defined to be the longest path from T_1 to T_n. Denote by t_{cp} the length of a critical path, which is just the sum of all the task execution times along the path. Obviously, t_{cp} represents the minimum time required for the execution of the graph; i.e., the execution time below t_{cp} for the graph is not possible, even with unlimited numbers of processors.

A task graph as described above provides a convenient abstraction of a parallel program. To accommodate the deterministic scheduling approach, we further consider like most previous work that it is a deterministic model; i.e., the task execution times and the dependence relationships in G are known with certainty in advance and remain unchanged during execution. In addition, we make the following common assumptions concerning the execution behavior of any single task in the graph: (1) a task requires only one processor, and (2) once a task starts its execution, it can run to completion without interruption.

Based on the above definitions and assumptions, now we can proceed to show the general form of the lower time bounds, which is defined to be a function of the target machine architecture and program characteristics.

Definition 1: Let $\Delta(P, G)$ represent an increase over the minimum execution time of a graph G when the number of processors in a machine P is not enough to cope with the parallelism inherent in the graph. Then, a lower bound, denoted as $LB_{time}(P, G)$, on the minimum time required to execute G on P is given by

$$LB_{time}(P, G) = t_{cp} + \Delta(P, G).$$

It is easy to see from the definition that the problem we need to solve in this paper is how to sharpen the increase $\Delta(P, G)$ such that the bound $LB_{time}(P, G)$ is maximized. Next, we

introduce two definitions which are used in the increase computation.

Definition 2: Let $\tau_{es}(T_i)$ be the earliest starting time of T_i, which indicates the least time in which T_i can be started due to the dependence constraints of G. Then, $\tau_{es}(T_1)$ is defined to be 0, and $\tau_{es}(T_i)$ for all the tasks T_i $(i = 2, 3, \cdots, n)$ is recursively defined as follows.

$$\tau_{es}(T_i) = \max_{(T_j, T_i) \in \mathcal{A}} \{\tau_{es}(T_j) + \mu(T_j)\}.$$

Definition 3: Let $\tau_{ls}(T_i)$ be the latest starting time of T_i, which indicates how long the starting of T_i can be delayed without increasing t_{cp}. Then, $\tau_{ls}(T_n)$ is defined to be $t_{cp} - \mu(T_n)$, and $\tau_{ls}(T_i)$ for all the tasks T_i $(i = n-1, n-2 \cdots, 1)$ is recursively defined as follows.

$$\tau_{ls}(T_i) = \min_{(T_i, T_j) \in \mathcal{A}} \{\tau_{ls}(T_j) - \mu(T_i)\}.$$

3. Increase Computation

This section discusses several techniques for estimating the increase over the minimum execution time of a given task graph. We start with a brief review of basic concepts used to calculate an increase $\Delta(P, G)$, and follow with a discussion of the implementation variations on the increase computation. The end goal is to present an improvement over the existing methods so as to obtain an increase sharper than the known value.

3.1 Fundamental Approach

Assume that task T_i $(\in \Gamma)$ starts for its execution at time $\tau(T_i)$ and ends at time $\tau(T_i) + \mu(T_i)$. The activity of T_i along time, according to the constraints imposed by G, can be defined as

$$f(T_i, t) = \begin{cases} 1 & t \in [\tau(T_i), \tau(T_i) + \mu(T_i)] \\ 0 & \text{otherwise.} \end{cases}$$

Thus the total activity of G at time t is simply the sum of all the task activities of Γ at this time, and it will be represented as a load density function:

$$F(t) = \sum_{T_i \in \Gamma} f(T_i, t).$$

Now let us move tasks without making the execution time of G exceed t_{cp}, so that they have a minimum possible overlap within a given interval $[\theta_1, \theta_2] \subset [0, t_{cp}]$. Also let $F(\theta_1, \theta_2, t)$ denote the load density function of the tasks or parts of tasks remaining within this interval after all the tasks have been shifted. The minimum load of the graph within $[\theta_1, \theta_2]$ is then the function which is defined as

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

Consider the minimum load and the effective activity $m(\theta_2 - \theta_1)$ that can be executed with m processors. Obviously, their difference, divided by m, indicates an increase in time over t_{cp}. Denote by $\delta(\theta_1, \theta_2)$ the increase. Mathematically,

$$\delta(\theta_1, \theta_2) = \lceil \frac{1}{m} \Phi(\theta_1, \theta_2) - (\theta_2 - \theta_1) \rceil \quad (1)$$

where $\lceil\ \rceil$ represents the minimum integer greater than the value of this expression. By evaluating the increase for every interval in $[0, t_{cp}]$, we can thus have the following equation.

$$\Delta(P, G) = \max_{0 \le \theta_1 < \theta_2 \le t_{cp}} \delta(\theta_1, \theta_2). \quad (2)$$

3.2 Bag Intersections

Fernandez and Bussell gave a method to evaluate the minimum load $\Phi(\theta_1, \theta_2)$, based on the theory of "bags". A bag is defined to be a set where repeated elements are accepted, and it is constructed in such a way that it contains as its elements the names of tasks in $[\theta_1, \theta_2]$, repeated as many times as there are time units of the respective tasks in this interval.

Let $\underline{\mathcal{F}}$ be the bag of the earliest task activities that contains all the tasks (or parts of tasks) executed at their earliest starting times, and $\overline{\mathcal{F}}$ be the bag of the latest task activities that contains all the tasks (or parts of tasks) executed at their latest starting times. The bag intersection $\underline{\mathcal{F}} \cap \overline{\mathcal{F}}$ of $\underline{\mathcal{F}}$ and $\overline{\mathcal{F}}$ is then defined as the bag of those elements common to the two operand bags. Since $\underline{\mathcal{F}} \cap \overline{\mathcal{F}}$ shows the task portions that cannot be processed earlier than the interval $[\theta_1, \theta_2]$ or later than this interval, the number $|\underline{\mathcal{F}} \cap \overline{\mathcal{F}}|$ of the task names in the bag intersection, including the repeated ones, is equivalent to the minimum load $\Phi(\theta_1, \theta_2)$ if we assume integer intervals. This leads to

$$\Phi(\theta_1, \theta_2) = |\underline{\mathcal{F}} \cap \overline{\mathcal{F}}|.$$

Consequently,

$$\delta(\theta_1, \theta_2) = \lceil \frac{1}{m} |\underline{\mathcal{F}} \cap \overline{\mathcal{F}}| - (\theta_2 - \theta_1) \rceil. \quad (3)$$

To demonstrate how a bag intersection can be obtained, let us now consider the task graph of **Fig. 1**, where the task name T_i is shown in the upper portion of each vertex, the execution time $\mu(T_i)$ (time units) is shown in the lower portion of the vertex, and the earliest starting time $\tau_{es}(T_i)$ and the latest starting time $\tau_{ls}(T_i)$ are shown between the square bracket

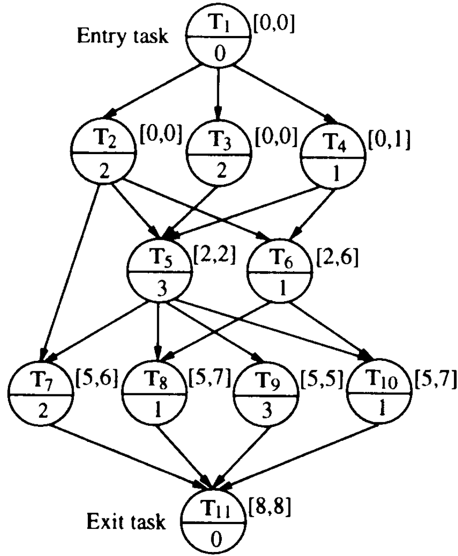

Fig. 1 Example of a task graph.

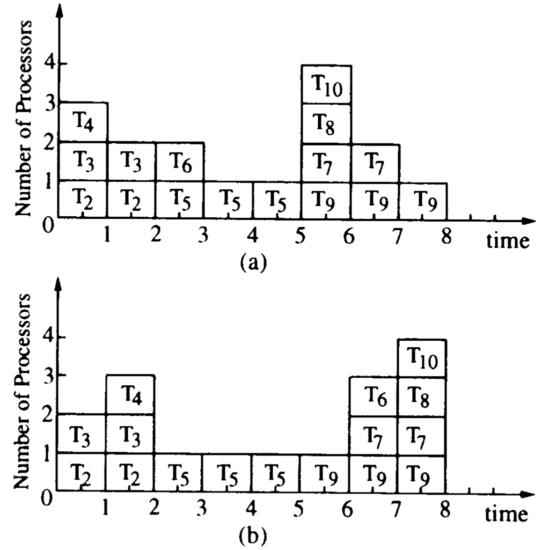

Fig. 2 Bags within the critical path length. (a)The bags of the earliest task activities. (b)The bags of the latest task activities.

near to the vertex. Based on the earliest and latest times of every task in the graph, we first describe the bags of the earliest task activities and the bags of the latest task activities as shown in **Fig. 2**. Then by applying the theory of "bags" to an integer time segment, for example, the interval [5, 8], we can obtain the bag $\underline{\mathcal{F}} = \{T_7, T_7, T_8, T_9, T_9, T_9, T_{10}\}$ and the bag $\overline{\mathcal{F}} = \{T_6, T_7, T_7, T_8, T_9, T_9, T_9, T_{10}\}$ within this interval. Thus, we have the bag intersection $\underline{\mathcal{F}} \cap \overline{\mathcal{F}} = \{T_7, T_7, T_8, T_9, T_9, T_9, T_{10}\}$ and then the number $|\underline{\mathcal{F}} \cap \overline{\mathcal{F}}| = 7$. Assume that only $m = 2$ identical processors are provided for the execution of the graph. By Eq. (3), the increase in the interval is then $\delta(5, 8) = 1$ (time unit).

3.3 Interval Overlaps

The computation described in the preceding section provides insight into the shortcomings of the Fernandez-Bussell method. First, the method is time-consuming because set theoretic operations are used, especially in the case when the mean execution time of tasks is larger. Second, it is inconvenient to deal with task graphs with real task execution times because it has to assume that $[\theta_1, \theta_2]$ is an integer interval. In this section, we show a very simple formulation for the increase computation, which improves the computation efficiency without sacrificing the quality of the increases, and needs not assume integer intervals.

We first denote by $t(\theta_1, \theta_2, T_i)$ the minimum execution time of task T_i within $[\theta_1, \theta_2]$ ($\subset [0, t_{cp}]$), which represents the time by which T_i (the part of T_i) must be executed in this interval. Then, the minimum load $\Phi(\theta_1, \theta_2)$ can be defined as the sum of all the minimum execution times of the tasks executed in $[\theta_1, \theta_2]$; mathematically, it is given by

$$\Phi(\theta_1, \theta_2) = \sum_{T_i \in \Gamma} t(\theta_1, \theta_2, T_i).$$

Next, we consider how to calculate $t(\theta_1, \theta_2, T_i)$. In an earlier paper[12], we presented a unified formulation for the computation, based on the evaluation of the minimum overlap between the interval $[\theta_1, \theta_2]$ and the interval $[\tau, \tau + \mu(T_i)]$ ($\tau_{es}(T_i) \leq \tau \leq \tau_{ls}(T_i)$) in which T_i is executed. By rewriting it to be consistent with our considerations, we can obtain $t(\theta_1, \theta_2, T_i)$ as follows.

$$t(\theta_1, \theta_2, T_i) = \begin{cases} \min[\tau_{ec}(T_i) - \theta_1, \theta_2 - \tau_{ls}(T_i), \\ \quad \theta_2 - \theta_1, \mu(T_i)] \\ \quad \theta_1 < \tau_{ec}(T_i) \text{ and } \theta_2 > \tau_{ls}(T_i) \\ 0 \quad \text{otherwise.} \end{cases} \tag{4}$$

where $\tau_{ec}(T_i)$ is defined as $\tau_{es}(T_i) + \mu(T_i)$, representing the earliest completion time of T_i. Although the equation was originally proposed for task graphs suitable for heterogeneous multiprocessors, it is directly applicable to programs composed of identical tasks because the latter is a special case of the former. We will not prove it here, due to the limitation of space. The details can be found in Ref. 12).

Based on the equations described above, we finally derive the following increase.

$$\delta(\theta_1, \theta_2) = \lceil \tfrac{1}{m} \textstyle\sum_{T_i \in \Gamma} t(\theta_1, \theta_2, T_i) \\ -(\theta_2 - \theta_1) \rceil \tag{5}$$

which is equivalent to Eq. (3) when we assume

Table 1 The increase over the critical path length within every interval.

$[\theta_1,\theta_2]$	δ	$[\theta_1,\theta_2]$	δ	$[\theta_1,\theta_2]$	δ	$[\theta_1,\theta_2]$	δ	$[\theta_1,\theta_2]$	δ
0, 1	0	0, 2	1	0, 3	0	0, 4	0	0, 5	0
1, 2	0	1, 3	0	1, 4	0	1, 5	0	1, 6	0
2, 3	0	2, 4	0	2, 5	0	2, 6	0	2, 7	0
3, 4	0	3, 5	0	3, 6	0	3, 7	0	3, 8	0
4, 5	0	4, 6	0	4, 7	0	4, 8	0		
5, 6	0	5, 7	0	5, 8	1	0, 6	0		
6, 7	0	6, 8	0	0, 7	0	1, 7	0		
7, 8	0	0, 8	0	1, 8	0	2, 8	0		

integer intervals.

The new formulation is obviously more practical and convenient in the sense that it makes the increase computation faster and easier. Using it, in the following we demonstrate the detail of computing the increase $\Delta(P,G)$, which will serve to motivate our improvement on its sharpness. Consider the task graph shown in Fig. 1 once again as an illustrative example, and assume that 2 identical processors are available. **Table 1** shows the evaluation of the increase $\delta(\theta_1,\theta_2)$ for every integer interval $[\theta_1,\theta_2]$ ($\subset [0,8]$). From the table, we have $\delta(0,2) = \delta(5,8) = 1$ (time unit). Then by Eq. (2), the maximum increase over every interval is that $\Delta(P,G) = \max\{\delta(0,2),\delta(5,8)\} = 1$ (time unit).

Now we proceed to make an interesting observation about the two increases $\delta(0,2)$ and $\delta(5,8)$. First, it is not hard to see from Fig. 2 that within the intervals $[0,2]$ and $[5,8]$, the task sets $\{T_2,T_3,T_4\}$ and $\{T_7,T_8,T_9,T_{10}\}$ require more processors than the machine provides and consequently, they cause the increases $\delta(0,2)$ and $\delta(5,8)$, respectively. Furthermore, by referring to Fig. 1 we know that T_5 is delayed by $\delta(0,2)$ and T_{11} is delayed by $\delta(5,8)$ so as to preserve the dependence structure of the graph. The two postponements must propagate down to the critical paths $\{T_1,T_2,T_5,T_9,T_{11}\}$ and $\{T_1,T_3,T_5,T_9,T_{11}\}$, since tasks on critical paths have to be executed in sequence. As a result, the increase over the minimum execution time of this graph is at least $\delta(0,2)+\delta(5,8) = 2$ (time units), rather than $\max\{\delta(0,2),\delta(5,8)\} = 1$ (time unit). The question arises: can we find an effective scheme that obtains an increase sharper than $\Delta(P,G)$? Obviously, some improvement to the existing methods is possible.

3.4 Propagation Delays

A study of the example in the preceding section yields an obvious idea about how to tighten the increase $\Delta(P,G)$ further. More specifically, it tells us that one way of trying to find a better increase is to analyze the propagation of the delays incurred due to non-zero increases $\delta(\theta_1,\theta_2)$, based on the dependence structure of a graph G. In this section, we describe in detail an efficient strategy that helps to implement the idea at the expense of a slight increase in the computational complexity.

Before discussing the strategy, we need to introduce a few more terms. For convenience, throughout the rest of this paper a task that lies on critical paths will be called a *critical task*; a task with a delay at that point will be termed *infected*. We refer to a delay as an *infection*, and use these two terms interchangeably. We also refer to the set of tasks generating an infection as an *infection source*. For example, consider the task graph shown in Fig. 1. We say that tasks T_1, T_2, T_3, T_5, T_9 and T_{11} in the graph are critical tasks. In addition, if $m = 2$ in the target machine, we say that tasks T_5, T_6 are infected by the infection source $\{T_2,T_3,T_4\}$, and task T_{11} is infected by the infection source $\{T_7,T_8,T_9,T_{10}\}$.

A task may be *directly* or *indirectly* infected by an infection source. The important point to note here is that the propagation of a delay occurs only when at least one critical task is infected either directly or indirectly. For example, as can be seen from Fig. 1, the delay from the infection source $\{T_2,T_3,T_4\}$ (if it exists) is transmitted to the task set $\{T_7,T_8,T_9,T_{10}\}$ because the critical task T_5 is directly infected, and it is then transmitted to task T_{11} because the critical task T_9 is indirectly infected. In practice, once a critical task in a task graph is directly infected by an infection source, the delay incurred due to the source must propagate down to critical paths and then lengthen the total execution time of the graph. Therefore, for our purpose here, we need only focus our attention on the delays associated with critical tasks. In the following, we first show how to determine an infection source and then discuss

how to analyze the propagation of the delay incurred due to the source along a critical path.

The determination of an infection source can be done very simply, based on whether or not the minimum execution time of every task is zero during the intervals within which the increases occur. Mathematically speaking, if $\delta(\theta_1,\theta_2) > 0$, then an infection source is the set $\{T_{i_j}\}_{j=1}^{g}$, where T_{i_j} satisfies the condition $t(\theta_1,\theta_2,T_{i_j}) \neq 0$. Let π_i denote the source and $d(\pi_i)$ $(= \delta(\theta_1,\theta_2))$ denote the delay from π_i. Also, assume without loss of generality that all the infection sources are sorted in the nonincreasing order of the increases associated with them. We then obtain a sequence of infection sources, $\Pi = \langle\pi_1,\pi_2,\cdots,\pi_h\rangle$, where h is the number of the sources and the following condition holds $d(\pi_1) \geq d(\pi_2) \geq \cdots \geq d(\pi_h)$.

Once all the infection sources have been manifested as above, what we need to do next is to find out the critical tasks that are directly infected by these sources. Consider an infection source $\pi_i = \{T_{i_j}\}_{j=1}^{g}$. Our approach for doing this has two conceptual steps. The first step is to obtain the common successors of tasks in the source, by taking the intersection $\bigcap_{j=1}^{g} Succ(T_{i_j})$. The second step is to determine the critical tasks infected directly via π_i according to dependence constrains among these tasks. Assume that T_κ is such a task. We introduce here a binary operator $\overset{d}{\rightarrow}$ to represent the relationship between π_i and T_κ; i.e., $\pi_i \overset{d}{\rightarrow} T_\kappa$ means that π_i directly postpones the execution of T_κ. Furthermore, let $d(T_\kappa)$ denote a direct-delay of T_κ, which indicates how long the execution of T_κ is postponed because of the direct infection from an infection source. We then have $d(T_\kappa) = d(\pi_i)$ if $\pi_i \overset{d}{\rightarrow} T_\kappa$.

In a similar way, we can also obtain a critical task that directly postpones the execution of all the tasks in π_i. By making use of the same operator $\overset{d}{\rightarrow}$ to represent the relationship between them, we formally have that if $T_\kappa \overset{d}{\rightarrow} \pi_i$ then the critical task $T_\kappa \in \bigcap_{j=1}^{g} Pred(T_{i_j})$ and there is no critical task $T'_\kappa \in Succ(T_\kappa)$ with $T'_\kappa \in \bigcap_{j=1}^{g} Pred(T_{i_j})$.

Now, it remains to show how to analyze the propagation of the delays incurred due to infection sources. Let us consider a critical path $\rho_\kappa = \langle T_{\kappa_1},\cdots,T_{\kappa_i},\cdots,T_{\kappa_l}\rangle$, where $T_{\kappa_1} = T_1$, $T_{\kappa_l} = T_n$ and $T_{\kappa_i} \in \Gamma$ such that for all i $(1 \leq i < l)$, $(T_{\kappa_i},T_{\kappa_{i+1}}) \in \mathcal{A}$. Assume that two tasks T_{κ_i} and T_{κ_j} $(i < j)$ on the path

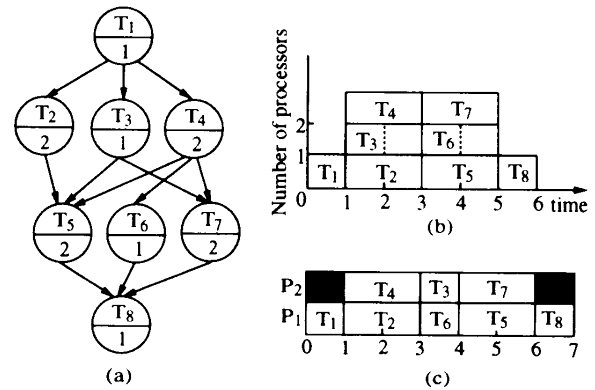

Fig. 3 Example of a propagation analysis. (a) A task graph. (b) Execution timing diagram of tasks. (c) A optimal schedule with 2 identical processors.

are directly infected by π_i and π_j, respectively; i.e., $\pi_i \overset{d}{\rightarrow} T_{\kappa_i}$ and $\pi_j \overset{d}{\rightarrow} T_{\kappa_j}$. Clearly, it is not always able to accumulate $d(T_{\kappa_i})$ and $d(T_{\kappa_j})$ simply. This is because $d(T_{\kappa_j})$ may be not independent of $d(T_{\kappa_i})$. **Figure 3** shows such an example, where $\{T_2,T_3,T_4\} \overset{d}{\rightarrow} T_5$ and $\{T_5,T_6,T_7\} \overset{d}{\rightarrow} T_8$ when $m = 2$ in the target machine. As can be seen from this figure, the delay of T_5 lengthens the longest path from T_4 to T_8 by $d(T_5)$ and consequently, the delay incurred due to $\{T_5,T_6,T_7\}$ is zero because $T_5 \in \{T_5,T_6,T_7\}$. This shows that the sum, $d(T_5)+d(T_8)$, is incorrect in this case (see Fig. 3 (c)). Therefore, to avoid the problem, we must place some restrictions on the propagation analysis. Here, instead of estimating all the direct-delays associated with critical tasks on ρ_κ, we start with $d(T_{\kappa_i}) = 0$ for all i $(1 \leq i \leq l)$ and proceed as follows.

(1) Select an infection source π_i with the minimum subscript i from Π,

(2) Determine tasks T_{κ_p} and T_{κ_q} on ρ_κ such that $T_{\kappa_p} \overset{d}{\rightarrow} \pi_i \overset{d}{\rightarrow} T_{\kappa_q}$. If they exist, then let $d(T_{\kappa_q}) = d(\pi_i)$ and subdivide ρ_κ into three subpaths using them as dividing-points,

(3) Repeat the computation until no more infection sources or no more subpaths remain unanalyzed.

All the three parts mentioned above make the direct-delays of critical tasks of ρ_κ independent. Therefore, the sum of these delays represents the shortest possible execution delay for ρ_κ. Then by taking the maximum over the sum for every critical path, we derive an increase in the execution time over t_{cp} as follows.

$$\Delta'(P,G) = \max_{\kappa} \sum_{T_{\kappa_i} \in \rho_\kappa} d(T_{\kappa_i}) \qquad (6)$$

where ρ_κ denotes the κth critical path in G.

A proof of correctness of the increase outlined above is straightforward, and is given in the following lemma.

Lemma 1: The execution of a graph G with a machine $\mathcal{P}$ must be delayed by a time no less than $\Delta'(P,G)$.

Proof: The execution time of G is no smaller than the completion time of any its tasks. The minimum completion time of exit task T_n is therefore the minimum execution time necessary to run G on P. Thus, to prove this lemma, we need only show that the execution of T_n is delayed at least by $\Delta'(P,G)$.

Assume without loss of generality that $\Delta'(P,G) = d(T_{\kappa_i}) + d(T_{\kappa_j})$, where T_{κ_i} and T_{κ_j} lie on a critical path ρ_κ. Also assume that $T_{\kappa_j} \in Succ(T_{\kappa_i})$, $\pi_i \overset{d}{\rightarrow} T_{\kappa_i}$ and $\pi_j \overset{d}{\rightarrow} T_{\kappa_j}$. Then by the method described above, we know that $\pi_j \subset Succ(T_{\kappa_i})$ because there must exist a task T_{κ_k}, $i \le k < j$, such that $T_{\kappa_k} \overset{d}{\rightarrow} \pi_j$. This shows that π_i has to transmit the delay $d(\pi_i)$ to all the tasks in π_j through T_{κ_i} and then indirectly infect T_{κ_j}. T_{κ_j} is therefore postponed at least by $d(T_{\kappa_i}) + d(T_{\kappa_j})$. This delay propagates down to ρ_κ and then arrives at T_n because critical tasks have to be executed in sequence. Consequently, T_n is also postponed at least by $d(T_{\kappa_i}) + d(T_{\kappa_j})$. This completes this proof. □

4. Discussion

This section evaluates the quality of the proposed propagation analysis method against that of Fernandez and Bussell. Our emphasis is mainly on the theoretical proof, without much concern for the experimental evaluation.

Lemma 2: For any task graph model $G(\Gamma, \mathcal{A}, \mu)$ to be scheduled on a machine P, the increase $\Delta'(P,G)$ always satisfies $\Delta'(P,G) \ge \Delta(P,G)$.

Proof: If $\Delta(P,G) = 0$, then the statement is obvious since $\Delta'(P,G) = 0$. Let us assume here that $\Delta(P,G) = \delta(\theta_1, \theta_2) > 0$. By Eq. (2), we know that $\delta(\theta_1, \theta_2)$ is the maximum over every interval within $[0, t_{cp}]$. Thus, $\delta(\theta_1, \theta_2) = d(\pi_1)$ since infection sources are sorted in the non-increasing order of the increases associated with them. As a result, we have $\Delta(P,G) = d(\pi_1)$ in this case.

On the other hand, it is true that every task is reachable from entry task T_1 and reaches exit task T_n. Therefore, we can always find two tasks T_{κ_p} and T_{κ_q} on a critical path ρ_κ $(= \langle T_{\kappa_1}, \cdots, T_{\kappa_i}, \cdots, T_{\kappa_l} \rangle)$, such that $T_{\kappa_p} \overset{d}{\rightarrow} \pi_1 \overset{d}{\rightarrow} T_{\kappa_q}$. This shows that there must exist at least one critical task T_{κ_q} whose direct-delay is equal to $d(\pi_1)$; i.e., $d(T_{\kappa_q}) = d(\pi_1)$. The following inequality then becomes obvious.

$$\begin{aligned} \Delta'(P,G) \ge & \textstyle\sum_{i=1}^{q-1} d(T_{\kappa_i}) + d(T_{\kappa_q}) \\ & + \textstyle\sum_{i=q+1}^{l} d(T_{\kappa_i}) \\ \ge & \; d(\pi_1). \end{aligned}$$

This completes the proof. That is, we have $\Delta'(P,G) \ge \Delta(P,G)$. □

The above lemma shows that the increases obtained by our method can never give lower values than the Fernandez-Bussell increases. So we can say that $\Delta'(P,G)$ is sharper than $\Delta(P,G)$. Of course, there may be many cases where $\Delta(P,G)$ is acceptable, but in complex graphs the advantages of having a sharp increase are clear if we think in the search for a near-optimal schedule.

As an illustrative example, let us come back to the task graph shown in Fig. 1 and assume that 2 identical processors are available. In this case, $\Delta(P,G) = 1$ (time unit) and there exist two infection sources: $\pi_1 = \{T_2, T_3, T_4\}$ with $d(\pi_1) = 1$ (time unit) and $\pi_2 = \{T_7, T_8, T_9, T_{10}\}$ with $d(\pi_2) = 1$ (time unit). In the following, we demonstrate how the proposed method can be applied to obtain $\Delta'(P,G)$. Consider the first critical path $\rho_1 = \langle T_1, T_2, T_5, T_9, T_{11} \rangle$ in the graph. First, our method chooses the infection source π_1 to analyze the propagation of the delay $d(\pi_1)$ along the path. Then, since there exist tasks T_1 and T_5 such that $T_1 \overset{d}{\rightarrow} \pi_1 \overset{d}{\rightarrow} T_5$, it computes $d(T_5) = 1$ (time unit) and partitions the path into the three subpaths $\langle T_1, T_1 \rangle$, $\langle T_1, T_2, T_5 \rangle$ and $\langle T_5, T_9, T_{11} \rangle$. Note that $\langle T_1, T_1 \rangle$ and $\langle T_1, T_2, T_5 \rangle$ need no propagation analysis further. Thus, it chooses another infection source π_2 for the analysis along the path $\langle T_5, T_9, T_{11} \rangle$. Since $T_5 \overset{d}{\rightarrow} \pi_2 \overset{d}{\rightarrow} T_{11}$, it computes $d(T_{11}) = 1$ (time unit) and then partitions the path into $\langle T_5, T_5 \rangle$, $\langle T_5, T_9, T_{11} \rangle$ and $\langle T_{11}, T_{11} \rangle$. Now, no more infection sources remain unanalyzed and hence it finishes this process. Thus, as the result of the analysis, we obtain that the shortest possible execution delay for the path is the sum of $d(T_5)$ and $d(T_{11})$. In the same way, the method analyzes the propa-

gation of $d(\pi_1)$ and $d(\pi_2)$ along the second critical path $\rho_2 = \langle T_1, T_3, T_5, T_9, T_{11}\rangle$, and obtains $d(T_5) = d(T_{11}) = 1$ (time unit), too. By Eq. (6), we finally have $\Delta'(P, G) = d(T_5) + d(T_{11}) = 2$ (time units), which is sharper than $\Delta(P, G) = 1$ (time unit).

The theoretical analysis and example described above have demonstrated that the proposed method has superior performance over the technique suggested by Fernandez and Bussell. This performance is clearly attained at the price of added complexity. However, it may be not very expensive from the point of computation. We now proceed to analyze its computational complexity. Let n, p and h represent the number of tasks, the number of critical paths and the number of infection sources, respectively. First, we consider the time needed to calculate the increases $\delta(\theta_1, \theta_2)$ for all the possible intervals $[\theta_1, \theta_2]$ within $[0, t_{cp}]$. It is easy to see that the computation requires the processing of $t_{cp}(t_{cp}+1)/2$ intervals $[\theta_1, \theta_2]$ and hence the computational complexity is $O(n * t_{cp}^2)$ as in the Fernandez-Bussell method. Second, we estimate the time spent at the propagation analysis of the delays associated with critical tasks. Our propagation analysis algorithm consists of a main loop containing an inner loop. It is obvious that the main loop repeats p times for processing all the critical paths. The running time depends mainly on the number of repetitions of the inner loop. The inner loop consists of three steps and repeats h times in the worst case. The most time-consuming step is clearly step 2, which takes at most $O(n^3)$ to determine T_{κ_p} and T_{κ_q} for π_i. However, we can limit the worst case computation time spent on the step to $O(n)$, by determining both the critical tasks which directly postpone the execution of π_i and the critical tasks whose executions are directly postponed by π_i before the propagation analysis begins. Thus, the computational complexity of the algorithm is $O(p * h * n)$ in the worst case. We feel that it is acceptable since the number of intervals with non-zero increases, that is, h, is much less than $t_{cp}(t_{cp}+1)/2$ in practice.

5. Concluding Remarks

This paper discussed a new technique to find an increase over the minimum execution time of a given task graph. Performance evaluation showed that the increase obtained by it is of higher quality than that of other known methods. Therefore, the technique can be used to provide better performance measures of static scheduling heuristics and design more efficient optimal branch-and-bound schemes of solving a scheduling problem. In addition, although the traditional task graph model was assumed here, the technique can be directly incorporated into the methods shown in Refs. 11), 12) and 13) to improve the sharpness of the existing bounds, since the basic principles are similar except that they consider inter-processor communication costs and the processor heterogeneity. To evaluate the performance of the technique under different task graph model assumptions, experimental work is currently being carried out by using a set of sample programs and will be reported in the future.

Acknowledgments The work reported in this paper was supported in part by the Ministry of Education, Science and Culture under Grant No.07780250, and the Hori Information Science Promotion Foundation.

The authors are grateful to the anonymous referees for their valuable comments on the presentation of this paper.

References

1) Fisher, J.A.: The VLIW Machine: a Multiprocessor for Compiling Scientific Code, *IEEE Trans. Comput.*, Vol.17, No.7, pp.45–53 (1984).
2) Lenatra, K.K. and Kan, A.H.G.R.: Complexity of Scheduling under Precedence Constraints, *Oper. Res.*, Vol.26, No.1, pp.22–35 (1978).
3) Adam, T.L., Candy, K.M. and Dickson, J.R.: A Comparison of List Schedules for Parallel Processing Systems, *Commun. ACM*, Vol.17, No.12, pp.685–690 (1974).
4) Khan, A.A., McCreary, C.L. and Jones, M.S.: A Comparison of Multiprocessor Scheduling Heuristics, *Proc. the 1994 International Conference on Parallel Processing*, Vol.II, pp.243–250 (1994).
5) Coffman, E.G.: *Computer and Job-shop Scheduling Theory*, John Wiley & Sons (1976).
6) Kohler, W.H.: A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems, *IEEE Trans. Comput.*, No.12, pp.1235–1238 (1975).
7) Kasahara, H. and Narita, S.: Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Trans. Comput.*, Vol.C-33, No.11, pp.1023–1029 (1984).
8) Chen, C.L., Lee, C.S.G. and Hou, E.S.H.: Efficient Scheduling Algorithms for Robot Inverse Dynamics Computation on a Multiprocessor

System, *IEEE Trans. Syst., Man, Cybernetics*, Vol.18, No.12, pp.729–743 (1988).
9) Hou, E.S.H., Ansari, N. and Ren, H.: A Genetic Algorithm for Multiprocessor Scheduling, *IEEE Trans. Parallel Distributed Syst.*, Vol.5, No.2, pp.113–120 (1993).
10) Fernandez, E.B. and Bussell, B.: Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules, *IEEE Trans. Comput.*, Vol.C-22, No.8, pp.745–751 (1973).
11) Al-Mouhamed, M.A.: Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs, *IEEE Trans. Softw. Eng.*, Vol.16, No.12, pp.1390–1401 (1990).
12) Li, D., Arita, T., Ishii, N. and Sowa, M.: A Lower Bound on the Time of Programs for Heterogeneous Parallel Processor, *Trans. IPS Japan*, Vol.34, No.11, pp.2378–2385 (1993).
13) Li, D., Ishii, N. and Sowa, M.: A Performance Measure for the Scheduling of Typed Task Systems with Communication Costs, *Trans. IPS Japan*, Vol.35, No.8, pp.1624–1633 (1994).
14) Li, D., Takagi, H. and Ishii, N.: Increase Analysis in the Total Execution Time of a Parallel Program, *Proc. the International Symposium on Parallel Architectures, Algorithms and Networks*, pp.390–397 (1994).
15) Garey, M.R., Graham, R.L. and Johnson, D.S.: Performance Guarantees for Scheduling Algorithms, *Oper. Res.*, Vol.26, No.1, pp.3–21 (1978). (distributed to authors).

(Received May 15, 1995)
(Accepted January 10, 1996)

Dingchao Li received the B.S. degree in Computer Science from Beijing University of Aeronautics and Astronautics, Beijing, China, in 1983, the M.S. degree in Computer Engineering from Fukui University, Fukui, in 1991, and the Doctor of Engineering degree in Electrical and Computer Engineering from Nagoya Institute of Technology, Nagoya, in 1994. He is currently an Assistant at the Educational Center for Information Processing at Nagoya Institute of Technology. His research interests include parallel and distributed processing, parallel computer architectures and their compilers. He is a member of the IPS Japan and the IEICE Japan.

Yuji Iwahori was born in Nagoya, Japan on September 5, 1959. He received the B.S. degree from Dept. of Computer Science, Faculty of Engineering, Nagoya Institute of Technology in 1983, the M.S. degree and the Ph.D. degree from Dept. of Electrical and Electronics Engineering, Tokyo Institute of Technology, in 1985 and 1988. He joined the Educational Center for Information Processing, Nagoya Institute of Technology as a research associate in 1988, and has been an associate professor of the same institute since 1992. In the meantime, he joined the Laboratory for Computational Intelligence, Dept. of Computer Science, The University of British Columbia as a visiting scientist in 1991 and 1994. His interests include Computational Vision, Neural Network and Intelligent Software. He is a member of IPSJ, IEICE, and IEEE Computer Society.

Naohiro Ishii received the B.E. and M.E. degrees and the Doctor of Engineering degree in Electrical and Communication Engineering from Tohoku University, Sendai, in 1963, 1965 and 1968, respectively. From 1968 to 1974 he was at the Tohoku University School of Medicine, where he worked on information processing in the central nervous system. Since 1975, he has been with the Nagoya Institute of Technology, where he is a Professor in the Department of Intelligence and Computer Science. His current research interests include nonlinear analysis of neural network, computer processing and artificial intelligence.