JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **10**, 1–17 (1990)

Optimal Parallel Evaluation of AND Trees

BENJAMIN W. WAH*

Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana–Champaign, 1101 West Springfield Avenue, Urbana, Illinois 61801

AND

GUO-JIE LI

Institute of Computing Technology, Academia Sinica, P.O. Box 2704-1, Beijing, People's Republic of China

AND-tree evaluation is an important technique in artificial intelligence and operations research. An example is the divide-and-conquer algorithm, which can be considered the evaluation of a precedence graph consisting of two opposing AND trees. In this paper, the optimal degree of parallelism for evaluating any given AND tree is quantitatively analyzed. The efficiency analysis is based on both preemptive and nonpreemptive critical-path scheduling algorithms. It is found that the optimal degree of parallelism depends on the complexity of the problem to be solved, the shape of the precedence graph, and the task-time distribution along each path. The major results consist of showing the optimality of the preemptive critical-path scheduling algorithm for evaluating any given AND tree on a fixed number of processors, and tight bounds on the number of processors within which the number of processors leading to the optimal processor-time efficiency can be sought efficiently. © 1990 Academic Press, Inc.

1. INTRODUCTION

A wide class of problems arising in artificial intelligence, operations research, decision making, and various scientific and engineering fields involves finding a solution of a problem which is made up of a large number of subproblems. Solving such a problem can be represented as an AND-tree computation. Examples include evaluating arithmetic expressions, searching possible solution trees of logic programs, evaluating functional programs, scheduling operations in assembly lines, finding the extremum, merge-sorting, and quick-sorting.

There are two kinds of AND trees, *intrees* and *outtrees*, which specify the precedence relationships among nodes. In an intree (resp. outtree), each node has at most one immediate successor (resp. predecessor), and the root is an exit node (resp. entry node). Every node is reachable from the entry node for an outtree or can reach the exit node for an intree. In recursive computations, such as divide-and-conquer algorithms [2, 4, 5, 26], a problem is partitioned into smaller and distinct subproblems, and the solutions found for the subproblems are combined into a solution for the original problem. The procedure is applied recursively until the subproblems are so small that they can be solved directly. In this way, the evaluation can be viewed as a process with two phases, the decomposition of subproblems based on an outtree and the composition of results based on an intree. Hence, the precedence graph is composed of an intree and an outtree. We call this particular precedence graph an *outin tree.* Deterministic programs can be represented by outin trees. Functional programming deals exclusively with AND graphs [1], and data-flow graphs are AND graphs [11]. In this paper, AND trees and outin trees are used synonymously.

Outin trees have the following characteristics. First, they are single-entry single-exit precedence graphs without cycles. Second, in contrast to general forests, an outin tree has one outtree and one intree, and a one-to-one correspondence of all the leaves in the two trees. We call these leaves the *leaves of the outin tree.* Third, the subtasks generated by a task in an outtree are usually less complex; likewise, the parent subtasks of a task in an intree are generally less complex. In general, outin trees can be considered special cases of single-entry single-exit acyclic precedence or data-flow graphs. We focus our studies on outin trees in this paper, although some of our results can be generalized to general acyclic single-entry, single-exit precedence graphs.

Figure 1 illustrates an outin tree, which reflects the precedence relationships among tasks in a merge-sort algorithm to sort six numbers. The nonterminal nodes in the outtree represent decompositions, each of which splits a (sub)list into two smaller sublists, whereas the nonterminal nodes in the intree part represent composition, each of which generates a sorted list based on two smaller sorted sublists.

Evaluation of outin trees naturally suggests implementation on parallel computers due to the independence of sub-

* Research supported by National Aeronautics and Space Administration Contract NCC 2-481 and National Science Foundation Grant MIP 89-10584.

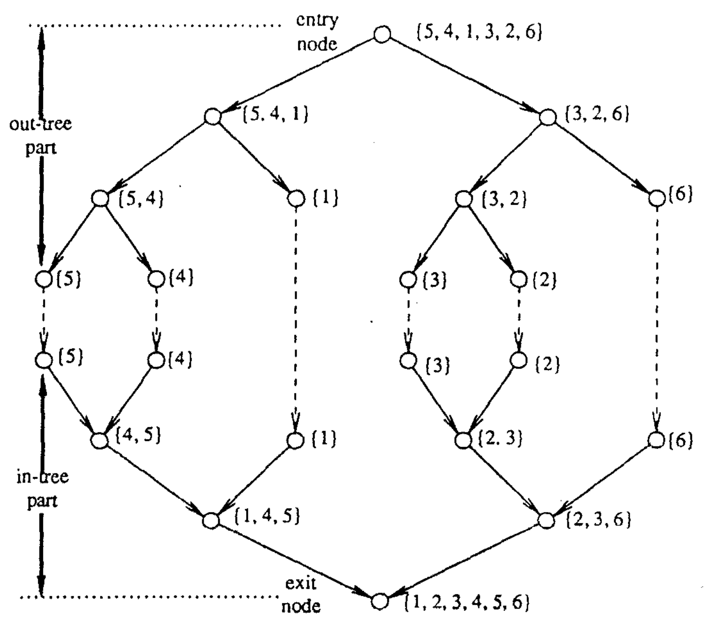

FIG. 1. Outin-tree representation of the sort-merge algorithm.

problems. AND-tree evaluations are important in the parallel evaluation of logic programs [9]. Studies conducted on parallel computers for executing divide-and-conquer algorithms can be classified into three types. First, multiprocessors that are connected in the form of a tree, especially a binary tree, can be used to exploit the potential parallelism of divide-and-conquer algorithms [3, 17, 28]. A second approach is the virtual-tree machine [6], which consists of a number of processors that are connected by an interconnection network, such as a binary n-cube network, and a suitable algorithm to decide when and where each subproblem should be solved. The third approach is a variation of the above approaches using a common memory. All processors are connected to the memory by a common bus [18].

To evaluate an AND tree in parallel, it is necessary to schedule the subproblems to achieve high throughput and processor utilization. An important problem is to determine the proper degree of parallelism that optimizes the processor-time efficiency. If the degree is too small, then the granularity of execution may be large, and the processors may be underutilized. In contrast, if the degree is too large, then the granularity of execution will be small, which results in tight coupling and prohibitive communication overhead. The degree of parallelism must be properly chosen to obtain a proper balance between processor utilization and communication overhead.

In previous studies, one can find different arguments on the issue of granularity. Some researchers advocate a fine grain, while others suggest a coarse grain. For example, in the FFP machine [25], a small grain is chosen for supporting ease and generality of parallel computations. In contrast, in Rediflow [21], large-grain parallelism is used to minimize communication overheads. Tang and Lee have derived the speedup ratios (ratio of order-of-magnitude time complexity of a sequential divide-and-conquer algorithm to the order-of-magnitude time complexity of the same algorithm in a parallel SIMD computer model) for various node complexities of any given outin tree [30]. They have also derived the order-of-magnitude number of processors that can achieve the optimal speedup for various node complexities.

In this paper, we quantitatively analyze the optimal degree of parallelism for evaluating any given outin tree using both preemptive and nonpreemptive scheduling policies. We identify the factors that influence the optimal degree of parallelism, in particular, the relationship between degree of parallelism and problem complexity. In Section 2, we prove that the preemptive critical-path scheduling algorithm is optimal in minimizing the completion time for scheduling any given outin tree. We also develop tight bounds on the number of processors within which the processor-time efficiency can be optimized. The case using preemptive critical-path scheduling is discussed in Section 3, while the case using nonpreemptive critical-path scheduling is discussed in Section 4.

In the analysis shown in this paper, it is assumed (a) that the precedence graph is known and is in the form of an outin tree; (b) that the computation time of each task is deterministic and known; (c) that the computational overheads of tasks are monotonically decreasing in the intree part and are monotonically increasing in the outtree part when nonpreemptive scheduling algorithms are used; (d) that the parallel processors are identical; (e) that an asynchronous model of parallel computation is assumed: a new task in the ready queue is scheduled whenever a task completes; (f) that all idle processors, if any, must be assigned to compute an available executable task; and (g) that the overhead involved in scheduling a task is negligible.

2. SCHEDULING PARALLEL OUTIN-TREE EVALUATIONS

2.1. *Terminologies and Background*

The precedence graph of an outin tree is oriented such that the entry node is at the top of the figure and the exit node is at the bottom. An arc is assumed to be always directed toward the bottom of the graph. The *length* of a node is the sum of the task execution times for nodes in the longest path from this node to the exit node. Figure 2a is an example of an outin tree. In this figure, the number inside a node is its task execution time, and the number next to a node is its length.

The execution time of a task can be interpreted as either its maximum processing time or its expected processing time. In the former case, the worst-case time to complete the schedule is considered, while in the latter case the length of the schedule represents a rough estimate on the average time of computation. In some outin-tree problems, the execution time of each task can be predicted quite accurately. For example, in evaluating arithmetic expressions, the time to execute a primitive operation, such as a multiplication, is known. In other cases, the average execution times may have to be estimated from statistics or from previous experience. In all cases, the communication overhead is nontrivial when preemptions are allowed, and the task time should also include the overhead of preemptions.

Our goal is to choose an algorithm that minimizes $c_{\max}$, the maximum completion time, for scheduling an outin tree on a set of P identical processors, and to find the number of processors that optimizes the processor-time efficiency based on this scheduling algorithm. Our scheduling problem is similar to the $P/\text{tree}/C_{\max}$ scheduling problem in which tree precedence graphs are considered [8, 10, 16]. Note that one must start with the optimal scheduling algorithm in order to find the optimal degree of parallelism.

If preemption is allowed, $P/\text{preemption, intree}/C_{\max}$ can be solved optimally either by Muntz and Coffman's Critical-Path Scheduling (CPS) algorithm in $O(N)$ time [27], or by other polynomial-time algorithms [13]. Besides being efficient and optimal, the CPS algorithm is easy to implement and, consequently, is one of the most common scheduling algorithms [22, 24]. In the CPS algorithm, the next job chosen is the one with the longest length of unexecuted jobs. This longest path is called the *critical path.* If preemption is not allowed, then optimal scheduling algorithms have been obtained only for two cases: (a) all tasks have equal execution times and the precedence relationships are in the form of an intree (Hu's algorithm [19]) and (b) when two processors are used [7]. Hu's optimal scheduling algorithm is indeed a CPS algorithm. Many other cases have been proved to be NP-hard [23, 31].

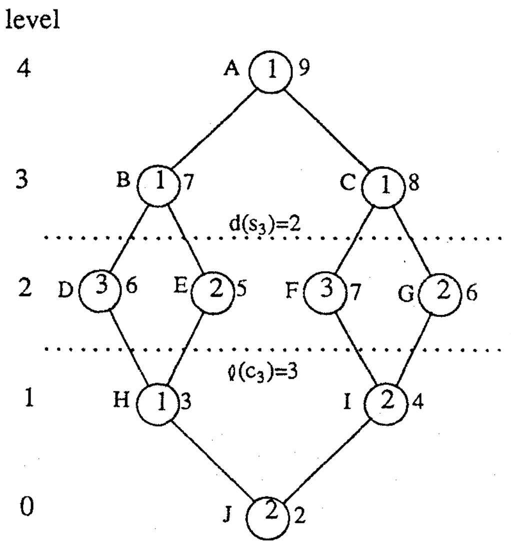

FIG. 2a. Task precedence graph as outin tree.

In the case that the precedence graph is a tree, that all processors are identical, and that each task requires t_i, $0 < t_i \leqslant t_{\max}$, units of time to complete, the nonpreemptive CPS algorithm turns out to be almost-optimal in the sense that

$$T_p(k) \leqslant T_{np}(k) \leqslant T_p(k) + t_{\max}, \qquad (2.1)$$

where $T_{np}(k)$ and $T_p(k)$ are, respectively, the completion times required by the nonpreemptive and preemptive CPS algorithms using k processors [20]. Some researchers have strived for nonpreemptive scheduling algorithms in order to solve scheduling problems with tree precedence [12, 16, 22]. Garey, Dolev, *et al.* have studied the scheduling of forests consisting of intrees and outtrees [12, 14]. Given a fixed number of processors, polynomial-time algorithms with high complexities for finding an optimal schedule of these forests have been developed.

In this research, we are interested in a special, but widely used, form of precedence trees, namely, the outin trees. With respect to a fixed number of processors, evaluating an outin tree can be divided into the *splitting, all-busy,* and *combining* phases. In the splitting phase, the problem is decomposed, and the number of busy processors is increased from one up to at most $k - 1$, where k is the maximum number of processors available. In the combining phase, the subproblems are composed, and the number of busy processors is decreased from at most $k - 1$ to one. During these two phases, some processors are idle. In contrast, in the all-busy phase, all the k processors are busy. Schindler has proved that the schedule of a precedence graph is optimal if either the computations can be completed in only the all-busy or combining phase or the precedence graph can be partitioned into the all-busy and combining phases by a "height-line" [29]. We show later that this result can be extented to scheduling outin trees, and that the CPS algorithm guarantees the optimal preemptive scheduling and near-optimal nonpreemptive scheduling of outin-tree evaluations.

To analyze the properties of *preemptive CPS,* in short, *PCPS,* algorithms, it is more convenient to represent a task (a node of the outin tree) of execution time t_i by a chain task, which is t_i *element tasks* (or *element nodes,* or in short, *e-tasks* or *e-nodes*), each of which has one unit of execution time (see Fig. 2b). We use a subscript i in the task identifier to indicate the ith e-task in the chain task. Hence, F_2 is the second e-task of task F. The new outin tree is called the *element-outin tree* (or *e-outin tree*). For each chain task, the e-task farthest from the exit e-node of the e-outin tree is called a *task-head* e-node. It is easy to verify that the length of the task-head e-node is the same as the length of the original multiunit task. Two e-nodes are said to be in the *same e-level* of the e-outin tree if their lengths are identical; that is, the e-level number of an e-node is equal to its length assuming that the exit e-node is in Level 0. To distinguish

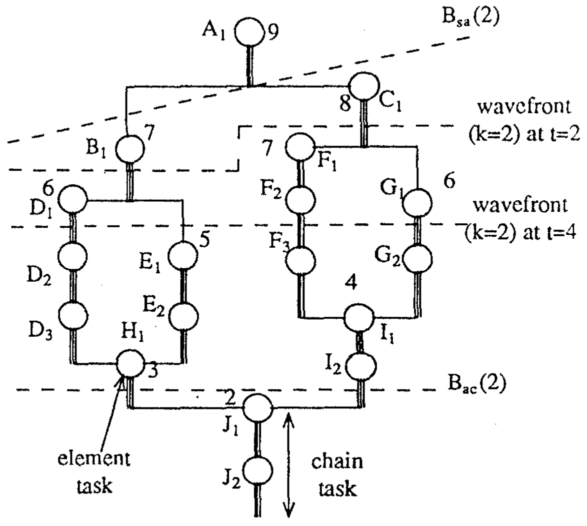

FIG. 2b. Task precedence graph as e-outin tree using chain tasks (all tasks require unit times).

between nodes and levels in the original outin tree (as exemplified by Fig. 2a) and in the e-outin tree (as exemplified by Fig. 2b), we use *tasks* and *levels* with respect to the original outin tree and *e-task* and *e-level* with respect to the e-outin tree in subsequent discussions.

There is another variation of preemptive scheduling algorithms called General Scheduling (GS) discipline [27], which is strongly related to preemptive critical-path scheduling. In the GS algorithm, each processor in the system is considered to have a certain amount of computing capacity rather than as a discrete unit, and this computing capacity can be assigned to tasks in any amount between zero and the equivalence of one processor. For example, if we assign half of a processor to task P_i with execution time t_i, then it will take $2 \cdot t_i$ units of time to complete P_i.

In scheduling a given outin tree using the GS discipline, one processor is assigned to each of the k e-nodes farthest from the exit e-node. If there is a tie in the lengths among u e-nodes for the last v, $u > v$, processors, then v/u of a processor is assigned to each of these u e-nodes. Each time when either (a) a chain task of the e-outin tree is completed or (b) a point is reached where, if we continue with the present assignment, some e-nodes will be computed at a faster rate than other e-nodes that are farther from the exit e-node, then the processors are reassigned to the remaining tree according to the CPS principle. Situation (b) occurs when an e-node that is being computed has the same length as that of some unexecuted task-head e-node(s). In this case, one (or part of a) processor must be assigned to the unexecuted task-head e-node. The GS discipline is illustrated in Fig. 2c.

Muntz and Coffman have proved the equivalence between the GS and PCPS algorithms [27]. That is, if preemptions were permitted, then the "processor-sharing" capability is not needed for optimal scheduling. To illustrate this equivalence, Fig. 2d shows the preemptive schedule for the corresponding e-outin tree in Fig. 2b.

In practice, preemptions are usually restricted to the beginning of a time unit, so the overhead of a practical PCPS algorithm is equal to that of Hu's algorithm, which assumes that tasks have unit execution times. From Eq. (2.1), we have [27]

$$\begin{aligned} T_p(k) = T_{gs}(k) \leqslant T_h(k) \leqslant \lceil T_{gs}(k) + 1 \rceil \\ = \lceil T_p(k) + 1 \rceil, \end{aligned} \tag{2.2}$$

where $T_h(k)$ and $T_{gs}(k)$ are the completion times required by Hu's and GS algorithms, respectively. Equation (2.2) shows that the behavior of the GS algorithm is very close to that of any PCPS algorithm which only allows preemptions at the beginning of a time unit. In subsequent discussions, the results are derived without any restriction on the allowable times for preemptions. Moreover, we use the GS discipline as a means for analyzing the properties of PCPS algorithms. The results derived are the same as those when the PCPS algorithm is used.

At time t, an e-node is said to be *active* if either a processor or part of a processor is assigned to it. The total number of active e-nodes may be greater than the number of processors since some e-nodes may share processors. All active e-nodes form a wavefront in the e-outin-tree evaluation. Two particular times of the wavefront are of special interest: $t_{sa}(k)$ and $t_{ac}(k)$. The computation enters the all-busy phase at $t_{sa}(k)$ and enters the combining phase at $t_{ac}(k)$. In both times, the wavefronts serve as phase boundaries. We call the former phase boundary $B_{sa}(k)$ and the latter $B_{ac}(k)$.

For the task graph in Fig. 2a, if the PCPS algorithm is employed, then $t_{sa}(2) = 1$ and $t_{ac}(2) = 8.5$ (see Figs. 2c and 2d). The corresponding phase boundaries $B_{sa}(2)$ and $B_{ac}(2)$ are indicated in Fig. 2b.

If a preemptive (resp. nonpreemptive) CPS algorithm is applied, then the computational times required by k processors to complete the splitting, all-busy, and combining

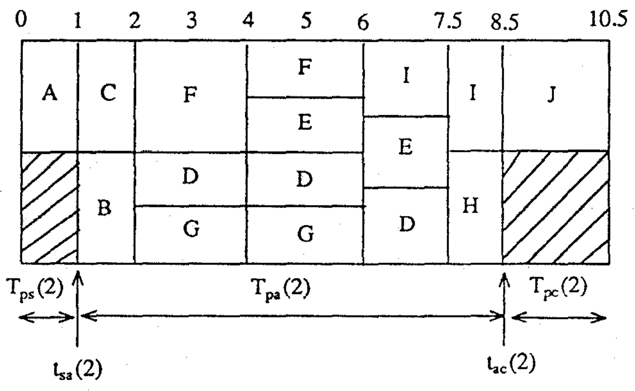

FIG. 2c. Timing diagram for general scheduling (processor sharing) using two processors for the e-outin tree in Fig. 2b.

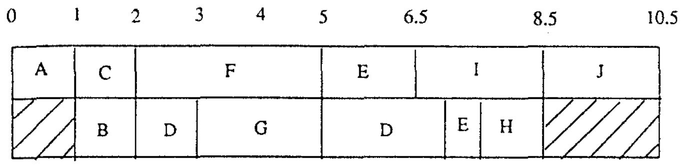

FIG. 2d. Timing diagram for preemptive CPS scheduling using two processors for the e-outin tree in Fig. 2b.

phases are denoted by $T_{ps}(k)$, $T_{pa}(k)$, and $T_{pc}(k)$ (resp. $T_{nps}(k)$, $T_{npa}(k)$, and $T_{npc}(k)$).

In the intree part of an e-outin tree, each e-node is associated with a path to the exit e-node, while each e-node in the outtree part may be associated with more than one path to the exit e-node. The longest path from any given e-node to the exit e-node is selected as the *execution path* through this e-node. For an e-node, if more than one such longest path exists, then a left-to-right tie-breaking rule may be used to break the tie. In the outtree part, if an e-node has q immediate successors, one of which (called the *immediate execution successor*) is in its execution path, then the other $q-1$ immediate successors (called *path heads*) serve as heads of new execution paths. As a result, each active e-node corresponds to a unique execution path to the exit e-node.

For example, in Fig. 2b, the execution path from A_1 is $(A_1, C_1, F_1, F_2, F_3, I_1, I_2, J_1, J_2)$ and B_1 is the path head of the execution path $(B_1, D_1, D_2, D_3, H_1, J_1, J_2)$. Note that when k processors are used, at most $k-1$ path heads are active in the splitting phase.

Let $A(t, k)$ be the set of active e-nodes at time t when k processors are used. This set can be divided into two classes in terms of the lengths of the corresponding execution paths. At time t, the active e-nodes whose execution paths are the shortest among all active e-nodes belong to a subset $A_s(t, k)$ and lie in a single e-level, called the *minimal active e-level.* The other active e-nodes belong to another subset $A_h(t, k)$. (If t and k are obvious in the context, they will be omitted for brevity.) For example, in Fig. 2b, when t is 2, e-nodes D_1 and G_1 belong to $A_s(2, 2)$, and e-node F_1 belongs to $A_h(2, 2)$.

2.2. *Optimality of the PCPS Algorithm*

In the following four propositions, we show the properties of the PCPS algorithm. We then prove a new result in Theorem 2.1 that the PCPS algorithm is optimal in minimizing the completion time for evaluating any given outin tree using a fixed number of processors.

During the evaluation of an outin tree, the active e-nodes are executed at different rates depending on whether the assigned processor is shared or not. Let $r_i(t)$ be the processing rate in e-node per unit time for active e-node i at time t. The following proposition distinguishes the processing rates under various conditions.

PROPOSITION 2.1. *During a parallel evaluation of an outin tree with k processors, $r_i(t)$, the processing rate of active e-node i at time t, satisfies the conditions*

$$r_i(t) = \begin{cases} =1 & \text{if e-node } i \in A_h(t, k) \quad \text{or} \quad |A(t, k)| \leq k \\ <1 & \text{otherwise,} \end{cases}$$

where $|A(t, k)|$ is the cardinality of $A(t, k)$, the set of active e-nodes.

Proof. This follows from the GS strategy immediately. ∎

Proposition 2.1 reflects the following facts. First, in the splitting and combining phases, the processing rate for any e-node is one; that is, an e-node is processed in each time unit. Second, if an active e-node is not in the minimal active e-level, then one processor (rather than part of a processor) has to be assigned to it; that is, its corresponding processing rate is one. Third, in the all-busy phase, if the number of active e-nodes is equal to the number of processors, then the processing rates for e-nodes in the minimal active e-level is also one. It is only when the number of active e-nodes is larger than the number of processors used that the processing rates of e-nodes in the minimal active e-level are less than one.

Let $h_{\max}$ be the length of the critical path in the outin tree to be evaluated. Since at least one time unit is needed to complete each e-node, it is evident for any scheduling algorithm that

$$T(k) \geq h_{\max}, \tag{2.3}$$

where $T(k)$ is the completion time using k processors under a preemptive or nonpreemptive scheduling discipline. The following proposition shows the relationship between $T_p(k)$ and the shape of the phase boundary.

PROPOSITION 2.2. (*a*) *$T_p(k) > h_{\max}$ implies that all active e-nodes on the phase boundary B_{ac} are located in the same e-level.* (*b*) *If an active e-node on the phase-boundary B_{ac} belongs to A_h, then $T_p(k) = h_{\max}$.*

Proof. (a) By Proposition 2.1, $T_p(k) > h_{\max}$ implies that at least one node in the critical path must share a processor with another node(s) in the all-busy phase. This means that

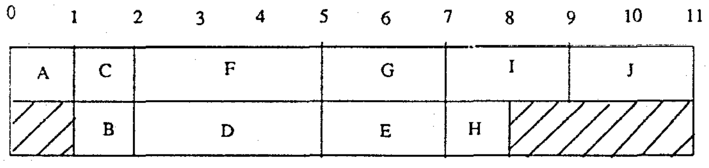

FIG. 2e. Timing diagram for nonpreemptive CPS scheduling using two processors for the e-outin tree in Fig. 2a.

at time t in the all-busy phase, some active node in the critical path must belong to $A_s(t, k)$. According to situation (b) of the GS strategy, when an active node in the critical path has the same length as that of some unexecuted task-head node(s), one (or part of) processor must be assigned to the unexecuted task-head node. That is, during the evaluation of an AND tree, the active node in the original critical path still has the longest execution path among all nodes in the remaining unexecuted AND tree. As a result, at time t in the all-busy phase, all active nodes must be in the same level and hereafter all their successors in the same level are evaluated simultaneously and at the same rate. It implies that all active nodes on the phase boundary B_{ac} are located in the same level. (b) In this case, at any time during the computation, the active nodes in the critical path belong to A_h. From Proposition 2.1, all nodes in the critical path are executed at the rate of one element task per unit time; hence, $T_p(k) = h_{max}$. ■

The example in Figs. 2b and 2c illustrates Property (a) above. At time $t = 4$, e-node G_1 in the critical path completes and "enters" the set A_s, and hereafter all active e-nodes are in the same e-level. Proposition 2.2 reflects the fact that preemptive scheduling algorithms distribute work uniformly among the available processors, thereby reducing the computational time required in the combining phase. This is the reason for a preemptive schedule to be shorter than or equal to the nonpreemptive counterpart.

To investigate the optimal scheduling of outin trees, we need to examine the phase boundaries when a different number of processors are used. The following proposition compares two boundaries with respect to k and $k + 1$ processors. In subsequent discussions, the phase boundary in a *single e-level* means that all active e-nodes on this boundary have execution paths with the same length.

PROPOSITION 2.3. *If phase boundary $B_{ac}(k + 1)$ is in a single e-level, then phase boundary $B_{ac}(k)$ must be in a single e-level.*

Proof. This proposition can be proved by contradiction. Let P_e be the entry node in the outin tree to be considered. By Proposition 2.2, the number of e-nodes from P_e to the phase boundary $B_{ac}(k + 1)$ along the critical path is less than $(h_{max} - T_{pc}(k + 1))$ (see Fig. 3).

$$T_p(k + 1) = T_{ps}(k + 1) + T_{pa}(k + 1) + T_{pc}(k + 1) \geqslant h_{max}$$

or

$$(h_{max} - T_{pc}(k + 1)) \leqslant (T_{ps}(k + 1) + T_{pa}(k + 1)). \quad (2.4)$$

Suppose when k processors are used and after $(T_{ps}(k + 1) + T_{pa}(k + 1))$ time units, there is an active e-node P_k be-

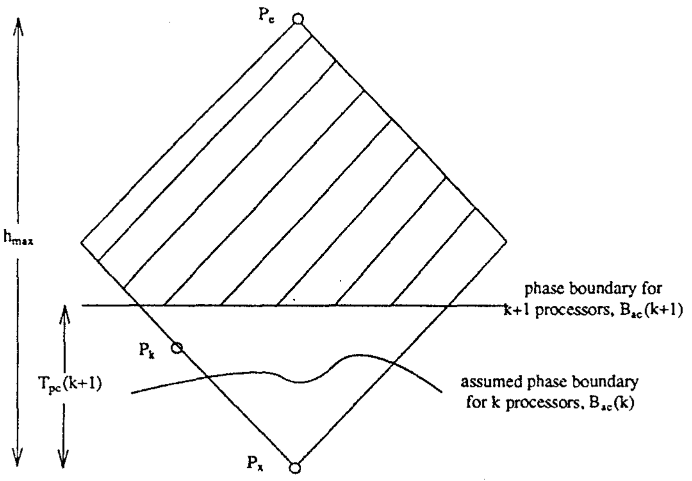
P_e
h_{max}
phase boundary for k+1 processors, $B_{ac}(k+1)$
P_k
$T_{pc}(k+1)$
assumed phase boundary for k processors, $B_{ac}(k)$
P_x

FIG. 3. Proof of Proposition 2.3.

longing to $A_h(k)$ in the critical path. According to Proposition 2.1, during this time interval, the processing rate in the critical path is always one, so the number of e-nodes from P_e to P_k is equal to $(T_{ps}(k + 1) + T_{pa}(k + 1))$, which is larger than $(h_{nax} - T_{pc}(k + 1))$. By our assumption, when $k + 1$ processors are used, the phase boundary $B_{ac}(k + 1)$ is in a single level. This means that when k processors are used, the wavefront after $(T_{ps}(k + 1) + T_{pa}(k + 1))$ time units must be below the phase boundary $B_{ac}(k + 1)$. The above argument implies that the number of e-nodes evaluated by k processors after $(T_{ps}(k + 1) + T_{pa}(k + 1))$ time units would be larger than that evaluated by $k + 1$ processors in the same time interval. The latter is indicated by the shaded area in Fig. 3. This is impossible since, in the first $T_{ps}(k + 1)$ time units, the number of e-nodes evaluated by k processors is equal to that evaluated by $k + 1$ processors and, in the subsequent $T_{pa}(k + 1)$ time units, the number of e-nodes evaluated by k processors is less than $(k + 1)T_{pa}(k + 1)$. Hence, if the phase boundary $B_{ac}(k + 1)$ is in a single level, then when k processors are used and after $(T_{ps}(k + 1) + T_{pa}(k + 1))$ time units, the wavefront must be in a single level and above $B_{ac}(k + 1)$; that is, there are at least $k + 1$ active nodes in the wavefront at that time. Hereafter, the wavefront must move down level by level according to the PCPS algorithm, so the phase boundary $B_{ac}(k)$ must be in a single level. ■

If the phase boundary $B_{ac}(k + 1)$ is not in a single e-level, then there are k', $0 < k' < k + 1$, active e-nodes belonging to $A_h(k + 1)$ on this phase boundary. During the splitting and all-busy phases, we can partition the $(k + 1)$ processors into two groups. The first group consists of k' processors which only evaluate e-nodes in the k' execution paths from the topmost k' path heads to the k' e-nodes. The other $(k + 1 - k')$ processors evaluate all e-nodes other than those in the aforementioned k' execution paths. Accordingly, we can prove the following proposition.

PROPOSITION 2.4. *If an outin tree is evaluated by the PCPS algorithm, then $T_{ps}(k+1) \geqslant T_{ps}(k)$, and $T_{pc}(k+1) \geqslant T_{pc}(k)$.*

Proof. First, it is easy to see that $T_{ps}(k+1) \geqslant T_{ps}(k)$, since when $k+1$ processors are used, the computation cannot enter the all-busy phase during the first $T_{ps}(k)$ time units. Second, the proof of Proposition 2.3 shows that if the phase boundary $B_{ac}(k+1)$ is in a single level, then $T_{pc}(k+1) \geqslant T_{pc}(k)$. Therefore, we only need to show that when $B_{ac}(k+1)$ is not in a single level, this inequality still holds.

Suppose P_{k+1} is a node on the phase boundary $B_{ac} \times (k+1)$ and also on the critical path; i.e., $T_{pc}(k+1)$ is equal to the length from P_{k+1} to the exit node P_x (see Fig. 4). When k processors are used, if P_{k+1} becomes active at time t, we claim that at this time either nodes on $B_{ac}(k+1)$ or their predecessors (rather than successors) should be active. That is, the wavefront should be totally above or in $B_{ac}(k+1)$. The claim is certainly true if the wavefront is in a single level. Otherwise, P_{k+1} belongs to $A_h(t, k)$. According to Proposition 2.1, along the critical path from P_e to P_{k+1}, the processing rate is always one, so $t = T_{ps}(k+1) + T_{pa}(k+1)$.

Suppose that P_a is one of the k_1 nodes on the phase boundary $B_{ac}(k+1)$, which belongs to $A_h(t, k+1)$ (see Fig. 4). We call these k_1 nodes *high-boundary nodes* because all their predecessors are evaluated at unit rate when $k+1$ processors are used.

Assume that P_h, one of the topmost k_1 path heads which is active in the splitting phase, is a predecessor of P_a. When k processors are used, P_h cannot be evaluated earlier than the time it was evaluated when $k+1$ processors were used because the processing rate cannot be greater than one. Likewise, when k processors are used, P_a and the other $k_1 - 1$ high-boundary nodes cannot be evaluated earlier than time t.

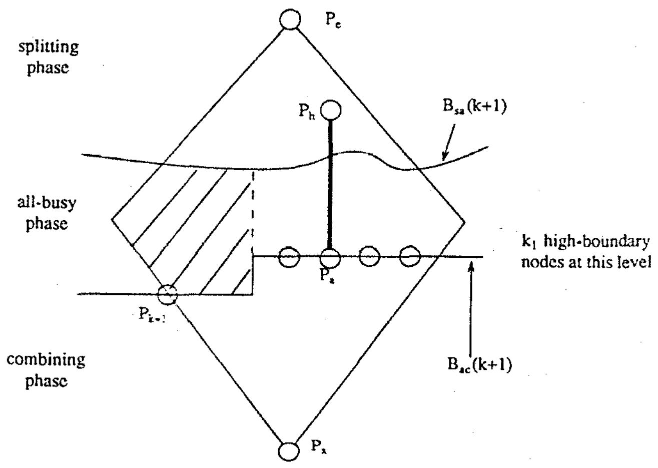

FIG. 4. Proof of Proposition 2.4.

If at time t and when k processors are used some predecessors of the high-boundary nodes share processors with other tasks, then they must be in the minimal active level, $A_s(t, k)$; that is, all wavefronts when k processors are used are above $B_{ac}(k+1)$ at this time. When k processors are used, if the k_1 execution paths from the topmost k_1 path heads to the k_1 high-boundary nodes are evaluated without processor sharing, then all nodes except those in the k_1 execution paths, as indicated by the shaded area in Fig. 4, are evaluated by $k - k_1$ processors, and all nodes in the same level within the shaded area should share processors. Since less processors are available, the processing rate when k processors are used should be smaller than that when $k+1$ processors are used. At time t, the minimal active level when k processors are used must also be above $B_{ac}(k+1)$.

Having proved our claim, we found that at time t there are more than k active nodes in the wavefront when k processors are used. This implies that $T_{pc}(k) < T_{pc}(k+1)$. ■

The following theorem shows that the PCPS algorithm can be used to find the optimal preemptive schedule for outin trees.

THEOREM 2.1. *PCPS is a minimum-completion-time scheduling algorithm for any given outin tree and a fixed number of identical parallel processors.*

Proof. Let $\Phi_{ps}(k)$ and $\Phi_{pc}(k)$ be the total amount of idle times in the splitting and combining phases when the PCPS algorithm is applied and k processors are used. Clearly,

$$T_p(k) = \frac{\Phi_{ps}(k) + T(1) + \Phi_{pc}(k)}{k}. \tag{2.5}$$

Minimizing $T_p(k)$ implies minimizing $(\Phi_{ps} + \Phi_{pc})$. In the PCPS algorithm, once an e-node is available, that is, its predecessor node has been finished, a processor is assigned to it immediately. The time spent in the splitting phase for any schedule cannot be shorter than that in the PCPS algorithm. This means that Φ_{ps} for the PCPS algorithm is the minimum.

We now consider Φ_{pc}. If the phase boundary B_{ac} is not in a single e-level, then $T_p(k) = h_{max}$ according to Proposition 2.2. That is, the PCPS algorithm achieves the minimum computational time $T(k)$ according to Eq. (2.3). Hence, we only need to consider the case when the phase boundary B_{ac} is in a single e-level. This boundary is indicated by line B in Fig. 5.

Suppose that an arbitrary scheduling algorithm is used; the corresponding phase boundary is denoted by B'. Note that it is impossible for all e-nodes on boundary B' to be beneath boundary B; otherwise, at least one processor is idle before the wavefront achieves B', which implies that B' is not a phase boundary. In other words, at least one e-node

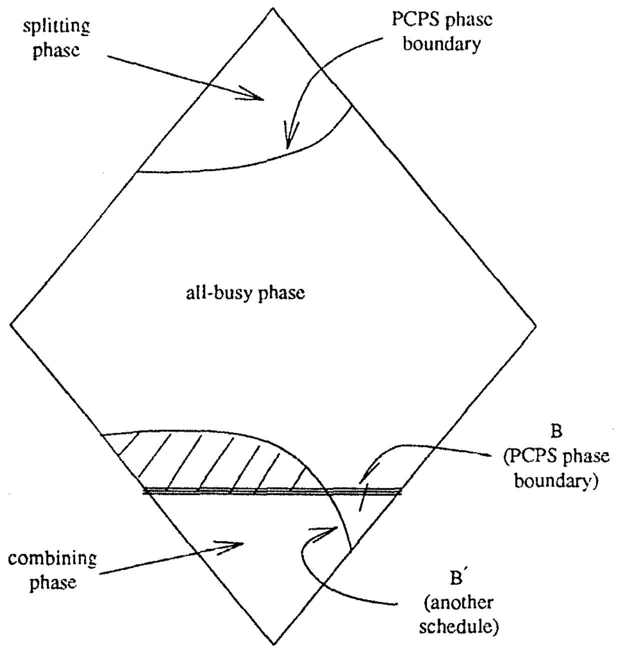

FIG. 5. Proof of Theorem 2.1.

on boundary B' is above or on boundary B_{ac}. Similarly, it is impossible for all e-nodes of boundary B' to be above boundary B.

Let N_{pc} and N'_c be the amount of task times in the combining phase of the PCPS and any other scheduling algorithm. Let $T'_c(k)$ and $\Phi'_c(k)$ be the computational time and the total idle time in the combining phase when an arbitrary scheduling algorithm using k processors is adopted. If $T_{pc}(k)$ equals $T'_c(k)$, then $N_{pc}(k) \geqslant N'_c(k)$; hence, $(\Phi'_c - \Phi_{pc}) = (N_{pc} - N'_c) \geqslant 0$. If $T_{pc}(k)$ is less than $T'_c(k)$, since at least one e-node on boundary B' is beneath or on boundary B_{ac}, $(N'_c - N_{pc})$ cannot be larger than the amount of task times for e-tasks beneath B' and above B (the shaded area in Fig. 5). Since, from Proposition 2.1, the processing rate for any path in the combining phase is one, then after $(T'_c - T_{pc})$ time units, all e-nodes in the shaded area in Fig. 5 must be completed. As less than k e-nodes can be completed during a time unit in the combining phase, the amount of e-nodes in the shaded area must be less than $k(T'_c - T_{pc})$. Therefore,

$$(\Phi'_c - \Phi_{pc}) = [k(T'_c - T_{pc}) - (N'_c - N_{pc})] > 0.$$

This means that Φ_{pc}, the total idle time in the combining phase, is also minimum for the PCPS algorithm. In summary, Φ_{ps} and Φ_{pc} are minimum for the PCPS algorithm, which implies that the PCPS algorithm is the minimum-completion-time algorithm. ■

The above proof does not imply that the PCPS algorithm is the unique optimal algorithm. However, it proves that the amount of idle times introduced by the PCPS algorithm is the minimum. In the following section, we find, with respect to this algorithm, the range on the number of processors that can optimize the processor-time efficiency.

3. OPTIMAL DEGREE OF PARALLELISM IN PREEMPTIVE SCHEDULING

The criteria that can be used to define the optimal degree of parallelism are U, speedup, $k \cdot T^2$, and $A \cdot T^2$, where U is the processor utilization, k is the number of processors, T is the computational time, and A is the area of a VLSI implementation.

The complexity of divide-and-conquer algorithms in a SIMD model and the conditions to assure the optimal processor utilization have been studied [18]. It is found that processor utilization increases monotonically with decreasing number of processors, which means that processor utilization is the maximum when one processor is used. Hence, processor utilization is not an adequate measure for the effects of parallel processing.

Tang and Lee use the speedup ratio as a measure of effectiveness of scheduling a divide-and-conquer algorithm and develop the optimal order-of-magnitude number of processors to achieve the optimal speedup [30]. However, their results do not lead to the optimal number of processors for evaluating any given outin tree because they are only true asymptotically.

The measure used in this paper for deriving the optimal degree of parallelism for evaluating a given outin tree is $k \cdot T^2$. This measure considers both processor utilization and computational time, since

$$k \cdot T^2(k) = \frac{T(1)T(k)}{U} = \frac{k \cdot T^2(1)}{\text{speedup}^2},$$

$$\text{where} \quad U = \frac{\text{speedup}}{k} \quad \text{and} \quad \text{speedup} = \frac{T(1)}{T(k)}.$$

To minimize $k \cdot T^2$ means to reduce the computational time and to maximize the processor utilization. $k \cdot T^2$ is linearly related to $A \cdot T^2$ if the area of connection wires is proportional to the area of processing elements. As both computational time and processor utilization are important in many applications, $k \cdot T^2$ is a good criterion to use. In other applications, such as real-time processing, the completion time may be more critical and processor utilization is a secondary consideration. In this case, a different optimization criterion may have to be used.

Note that finding the number of processors to minimize $k \cdot T^2$ is equivalent to finding the number of processors to maximum the speedup. Hence, the order-of-magnitude results derived by Tang and Lee [30] on optimizing with respect to the speedup ratio also apply to our case on optimizing with respect to the $k \cdot T^2$ measure. An advantage on using the $k \cdot T^2$ measure rather than the speedup is that it is a product rather than a ratio, which simplifies the derivation of results in this paper.

Our goal in this paper is to find the optimal degree of parallelism for evaluating any given outin tree with respect to the $k \cdot T^2$ measure; that is, given an outin tree, we need to choose k to minimize $k \cdot T^2$, or given a fixed k, we need to determine the type of outin trees (their shapes, complexities, etc.) and its proper size that can be solved most efficiently by this system. We derive our result with respect to scheduling any given outin tree and not with respect to the order-of-magnitude behavior of all outin trees.

It is difficult to find directly the optimal degree of parallelism with respect to $k \cdot T^2$ because the optimal degree depends on the execution time of each task and the shape of the outin tree. We attempt to find the optimal degree of parallelism via an intermediate variable, the total idle time. Let $\Phi_p(k)$ (resp. $\Phi_{np}(k)$) be the total amount of idle times when a preemptive (resp. nonpreemptive) scheduling algorithm with k processors is used. $\Phi_p(k)$ takes into account the idle times in both the splitting and the combining phases. Clearly, $\Phi_p(k) = [\Phi_{ps}(k) + \Phi_{pc}(k)]$ and

$$k \cdot T(k) = T(1) + \Phi(k). \tag{3.1}$$

Equation (3.1) holds for both preemptive and nonpreemptive scheduling algorithms.

The total idle time $\Phi_p(k)$ is related to both k and $k \cdot T^2$. The following two lemmas show the difference between the total idle times when different numbers of processors are used.

LEMMA 3.1. *When an outin tree is evaluated by the PCPS algorithm,*

$$[\Phi_p(k+1) - \Phi_p(k)] \leqslant h_{\max}, \tag{3.2}$$

where $h_{\max}$ is the length of the critical path.

Proof. Two cases are possible.

(a) $T_p(k+1) = h_{\max}$. The computational time for evaluating any AND tree cannot be less than $h_{\max}$ regardless of k, i.e., $T_p(k) \geqslant h_{\max}$. Hence, we can derive Eq. (3.2) using Eq. (3.1).

(b) $T_p(k+1) > h_{\max}$. In this case, all active nodes at time $t_{ac}(k+1)$ are in a single level from Proposition 2.2. According to Proposition 2.3 and its proof, when k processors are used, $B_{ac}(k)$, the phase boundary, is in a single level that is beneath $B_{ac}(k+1)$. Hence, each processor has identical amounts of idle times in both cases of using k and $k+1$ processors except that the additional $(k+1)$th processor becomes idle first in the case of using $(k+1)$ processors. That is,

$$\Phi_{pc}(k+1) - \Phi_{pc}(k) = T_{pc}(k+1). \tag{3.3}$$

In the splitting phase, the idle time of the $(k+1)$th processor is equal to $T_{ps}(k+1)$, which is not included in the case of k processors. It is certain that

$$\Phi_{ps}(k+1) - \Phi_{ps}(k) = T_{ps}(k+1). \tag{3.4}$$

Since at least one e-node in the critical path has been evaluated in the all-busy phase in this case, and the processing rate is one in the splitting and all-busy phases (according to Proposition 2.1), we have

$$\begin{aligned}\Phi_p(k+1) &- \Phi_p(k)\\ &= [T_{ps}(k+1) + T_{pc}(k+1)] < h_{\max}. \quad \blacksquare\end{aligned}$$

LEMMA 3.2. *Suppose that an outin tree is evaluated by the PCPS algorithm, then*

$$[\Phi_p(k+1) - \Phi_p(k)] \geqslant [T_{ps}(k) + T_{pc}(k)] > 0. \tag{3.5}$$

Proof. Adding Eqs. (3.3) and (3.4) results in the condition

$$\begin{aligned}T_{ps}(k+1) + T_{pc}(k+1) &= [\Phi_{ps}(k+1) + \Phi_{pc}(k+1)]\\ &\quad - [\Phi_{ps}(k) + \Phi_{pc}(k)]\\ &= \Phi_p(k+1) - \Phi_p(k).\end{aligned}$$

From Proposition 2.4, we know that

$$T_{ps}(k+1) + T_{pc}(k+1) \geqslant T_{ps}(k) + T_{pc}(k) > 0.$$

The lemma is, therefore, proved. ■

LEMMA 3.3. *Suppose that an outin tree is evaluated by the PCPS algorithm, then*

$$T_p(k+1) \leqslant T_p(k).$$

Proof. A sketch of the proof is given here. Suppose that the given outin tree is optimally scheduled for k processors. Two cases are possible. First, if $T_p(k) = h_{\max}$, then $T_p(k+1) = h_{\max}$, and $T_p(k+1) = T_p(k)$. Second, if $T_p(k) > h_{\max}$, then at least one node in the critical path must share a processor with another node(s) in the all-busy phase according to Proposition 2.1. This means that some of these nodes can be scheduled in the $(k+1)$th processor, resulting in a shorter or equivalent all-busy phase when $k+1$ processors are used. This implies that $T_p(k+1) \leqslant T_p(k)$. ■

The above lemmas reveal that when the number of processors used are increased, the total idle times must increase, and the difference of the total idle times with respect to k_1 and k_2 processors is bounded by $(k_2 - k_1)[T_{ps}(k) + T_{pc}(k)]$ and $(k_2 - k_1) \cdot h_{\max}$. From these facts, we can determine the conditions under which $k \cdot T^2$ is either monotonically increasing or decreasing with respect to k. The following theorem shows the relation between $\Phi_p(k)$ and $k \cdot T^2$.

THEOREM 3.1. *Suppose that an outin tree is evaluated by the PCPS algorithm. $k \cdot T_p^2(k)$ is monotonically increasing with k if $[\Phi_p(k+1) - \Phi_p(k)] > T_p(k)/2$. $k \cdot T_p^2(k)$ is monotonically decreasing with k if $[\Phi_p(k+1) - \Phi_p(k)] < T_p(k)/(2 + 1/k)$.*

Proof. By Eq. (3.1), we get

$$
\begin{aligned}
&(k+1)T_p^2(k+1) - k \cdot T_p^2(k) \\
&\quad = [T_p(1) + \Phi_p(k+1)]^2/(k+1) \\
&\qquad - [T_p(1) + \Phi_p(k)]^2/k \\
&\quad = (k[T_p(1) + \Phi_p(k+1)]^2 - k[T_p(1) + \Phi_p(k)]^2 \\
&\qquad - [T_p(1) + \Phi_p(k)]^2)/k(k+1) \qquad (3.6) \\
&\quad = ([\Phi_p(k+1) - \Phi_p(k)][2T_p(1) + \Phi_p(k+1) \\
&\qquad + \Phi_p(k)] - k \cdot T_p^2(k))/(k+1) \\
&\quad = ([\Phi_p(k+1) - \Phi_p(k)][(k+1)T_p(k+1) \\
&\qquad\qquad + k \cdot T_p(k)] - k \cdot T_p^2(k))/(k+1).
\end{aligned}
$$

Since from Lemma 3.2, $\Phi_p(k+1) > \Phi_p(k)$, hence

$$
\begin{aligned}
(k+1)T_p(k+1) &= [T_p(1) + \Phi_p(k+1)] \\
&> [T_p(1) + \Phi_p(k)] = kT_p(k).
\end{aligned} \tag{3.7}
$$

From Eqs. (3.6) and (3.7), we conclude that if $[\Phi_p(k+1) - \Phi_p(k)] > T_p(k)/2$, then $(k+1)T_p^2(k+1) > kT_p^2(k)$. From Lemma 3.1 and Eqs. (3.6) and (2.3), we obtain the following condition.

If

$$[\Phi_p(k+1) - \Phi_p(k)] < \frac{T_p(k)}{2 + 1/k},$$

then

$$
\begin{aligned}
&[(k+1)T_p^2(k+1) - k \cdot T_p^2(k)] \\
&< \left[\frac{k \cdot T_p(k)}{(2k+1)} \frac{[(k+1)T_p(k+1) + k \cdot T_p(k)]}{(k+1)} - \frac{k \cdot T_p^2(k)}{(k+1)}\right] \\
&\quad < \left[\frac{k \cdot T_p(k)}{2k+1} \frac{(2k+1)T_p(k)}{(k+1)} - \frac{k \cdot T_p^2(k)}{(k+1)}\right] = 0. \qquad \blacksquare
\end{aligned}
$$

Theorem 3.1 restricts the region within which we need to find a value k that minimizes $k \cdot T_p^2(k)$. In other words, the approximate condition that adding a processor will not degrade the processor-time efficiency is that all processors should be busy at least half of the time.

In the example shown in Fig. 2, $T_p(1) = 18$, $\Phi_p(1) = 0$, $T_p(2) = 10.5$, $\Phi_p(2) = 3$, $T_p(3) = 9$, and $\Phi_p(3) = 9$. (Readers are suggested to schedule this outin tree with three processors.) Since $(\Phi_p(2) - \Phi_p(1)) = 3 < T_p(1)/(2 + 1/1) = 6$, and $[\Phi_p(3) - \Phi_p(2)] = 6 > T_p(2)/2 = 5.25$, according to Theorem 3.1, we conclude that the use of two processors minimizes $k \cdot T^2$ for this outin tree.

A question about the monotonicity of $[k \cdot T_p^2(k+1) - k \cdot T_p^2(k)]$ now arises naturally. If $[k \cdot T_p^2(k+1) - k \cdot T_p^2(k)]$ is increasing monotonically with k, then $k \cdot T_p^2(k)$ is a unimodal function of k, and the optimal value of k can be found easily. This monotonicity is proved in the following theorem.

THEOREM 3.2. *Suppose that an outin tree is evaluated by the PCPS algorithm, then $k \cdot T_p^2(k)$ is a concave function of k, that is, $k \cdot T^2(k)$ achieves the minimum when $k = k'$, and $k \cdot T^2(k)$ is monotonically decreasing (resp. increasing) with k when $k < k'$ (resp. $k > k'$).*

Proof. To show $k \cdot T_p^2(k)$ is a concave function of k, we need to prove that its second-order difference is positive, namely, $[(k+2)T_p^2(k+2) - (k+1)T_p^2(k+1)] > [(k+1)T_p^2(k+1) - k \cdot T_p^2(k)]$. Let $\Delta(k \cdot T_p^2(k))$ denote $[(k+1)T_p^2(k+1) - k \cdot T_p^2(k)]$. Then

$$
\begin{aligned}
&\Delta(k \cdot T_p^2(k)) \\
&\qquad = k[T_p^2(k+1) - T_p^2(k)] + T_p^2(k+1)
\end{aligned} \tag{3.8}
$$

$$
\begin{aligned}
&\Delta((k+1)T_p^2(k+1)) \\
&\quad = (k+1)[T_p^2(k+2) - T_p^2(k+1)] + T_p^2(k+2).
\end{aligned} \tag{3.9}
$$

Subtracting Eq. (3.8) from Eq. (3.9) and applying Eq. (3.1) yield

$$
\begin{aligned}
&\Delta((k+1)T_p^2(k+1)) - \Delta(k \cdot T_p^2(k)) \\
&\quad = (k+2)[T_p^2(k+2) - T_p^2(k+1)] \\
&\qquad - k[T_p^2(k+1) - T_p^2(k)] \\
&\quad = [T_p(k+2) + T_p(k+1)] \qquad (3.10) \\
&\qquad \times [\Phi_p(k+2) - \Phi_p(k+1) - T_p(k+1)] \\
&\qquad - [T_p(k+1) + T_p(k)] \\
&\qquad\qquad \times [\Phi_p(k+1) - \Phi_p(k) - T_p(k+1)].
\end{aligned}
$$

From Eqs. (2.3) and (3.10) and Lemmas 3.1, 3.2, and 3.3, we conclude that $\{\Delta((k+1)T_p^2(k+1)) - \Delta(k \cdot T_p^2(k))\} > 0$. $\blacksquare$

We have found the condition under which $k \cdot T_p^2$ is increased or decreased based on the intermediate variable, $\Phi_p(k)$, and that $k \cdot T_p^2$ is a concave function. Next, we determine the number of processors such that $k \cdot T_p^2$ is minimum for any given outin tree.

Note that in the original outin tree, each node is a multiunit task, and tasks in a level may have different lengths. If there are $m(i)$ tasks in level i, then there are $m(i)$ paths from level i to the exit node. Among these paths, the minimum length is denoted by $l(i)$. Similarly, we can define the *depth* of a node as the sum of task times along a path from the entry node to and including this node, and denote the shortest depth from the entry node to level i by $d(i)$. For example, in Fig. 2a, node A is in level 4, and node J is in level 0. Nodes B and C are in level 3. The depths of nodes B and C are both 2; hence, $d(3) = 2$.

Given k processors, we can find c_k, a particular level in the intree of the original outin tree, such that $m(c_k)$, the number of tasks in this level, is less than k, but $m(c_k + 1) \geqslant k$. This particular level is called the *minimum-all-busy level.* Note that the entry node has the maximum level number and the exit node has the minimum level number. Likewise, in the outree, there is a level called the *maximum-all-busy level* and denoted by s_k, such that $m(s_k) < k$ and $m(s_k - 1) \geqslant k$. For the outin tree in Fig. 2a, $s_3 = 3$, $c_3 = 1$, $d(s_3) = 2$, and $l(c_3) = 3$.

Note that the minimum-all-busy level (resp. maximum-all-busy level) does not correspond to the phase boundary B_{ac} (resp. B_{sa}). However, by recognizing the minimum-all-busy and maximum-all-busy levels, we can roughly estimate the locations of the phase boundaries. Recall from Proposition 2.2 that $T_p(k) = h_{max}$ if the phase boundary B_{ac} is not in a single e-level. In this case, $(k+1)T_p^2(k+1) > k \cdot T_p^2(k)$. To achieve the minimum $k \cdot T_p^2(k)$, the number of processors should be reduced until the phase boundary appears in a single e-level. (When $B_{ac}(k)$ is not in a single e-level but $B_{ac}(k-1)$ is, $k \cdot T_p^2(k)$ may be minimum.) This observation shows that the use of the minimum-all-busy level to estimate T_{pc} is accurate in most cases. The same argument applies to T_{ps}. The following lemma shows that the shortest length from the minimum-all-busy (resp. maximum-all-busy) level to the exit (resp. entry) node gives the lower-bound computational time in the combining (resp. splitting) phase.

LEMMA 3.4. *Suppose that an outin tree is evaluated by k processors. If s_k and c_k are the maximum-all-busy and minimum-all-busy levels, then (a) $T_{ps}(k) \geqslant d(s_k)$, and (b) $T_{pc}(k) \geqslant l(c_k)$. Further, if the phase boundary $B_{ac}(k)$ lies in a single e-level, then $T_{pc} = l(c_k)$.*

Proof. In part (a), assume that there are m tasks in the maximum-all-busy level, s_k, where $m < k$. Hence, the computational process can enter the all-busy phase only if some task in the minimum-all-busy level has finished. That is, the computational time required in the splitting phase cannot be less than $d(s_k)$, the depth of the shortest path from the entry node to the minimum-all-busy level s_k. Likewise, we can prove part (b). Lastly, if the phase boundary $B_{ac}(k)$ lies in a single level, then all nodes in $B_{ac}(k)$ complete simultaneously. From then on, each e-node in the combining phase is allocated an entire processor, and no tasks got delayed due to processor sharing. As a result, the all-busy phase ends when it enters the minimum-all-busy level, and $T_{pc}(k) = l(c_k)$. ∎

This lemma is illustrated by the example in Fig. 2a. Suppose that three processors are used, the maximum-all-busy level contains tasks B and C, and the minimum-all-busy level contains tasks H and I. $T_{ps}(3)$ (resp. $T_{pc}(3)$) cannot be less than 2 (resp. 3) because if either task B or C, which are associated with the shortest depth from the entry node to this level, is not finished, then the computation cannot enter the all-busy phase. Likewise, if task H has been assigned to a processor, then the computation must have entered the combining phase. As another example, the minimum-all-busy level when two processors are used contains J. Since $B_{ac}(2)$ lies in a single e-level, $T_{pc}(2) = l(c_2) = 2$.

For an arbitrary outin tree and $k > 1$,

$$T_{ps} \geqslant t_{en}, \quad T_{pc} \geqslant t_{cx}, \quad \text{and} \quad \Phi_p \geqslant (t_{en} + t_{ex}), \qquad (3.11)$$

where t_{en} and t_{ex} are the task times of the entry and exit nodes, respectively. When more than one processor is used, some processors must be idle when the entry and exit nodes are evaluated. If the times spent in evaluating the entry and exit nodes dominate all other computations, then parallel processing is definitely inefficient. The following corollary identifies the condition under which sequential computation is better than parallel processing.

COROLLARY 3.1. *Suppose that an outin tree is scheduled by the PCPS algorithm and $(t_{en} + t_{ex}) > T(1)/2$, then sequential processing, i.e., $k = 1$, achieves the minimum $k \cdot T^2$.*

Proof. This follows from Theorem 3.1 immediately. ∎

Having proved a series of propositions, lemmas, and theorems, the main theorem for deriving the optimal degree of parallelism under the PCPS scheduling algorithm can be obtained now. In the following theorem, the region on k in which we can find the optimal degree of parallelism for evaluating any given outin tree is given.

THEOREM 3.3. *Suppose that an AND tree is evaluated by the PCPS algorithm and $k > 1$, then*

$$(k+1)T_p^2(k+1) > k \cdot T_p^2(k) \quad \textit{if} \quad k > \frac{2T_p(1)}{h_{max}} \qquad (3.12)$$

and

$$(k+1)T_p^2(k+1) < k\cdot T_p^2(k) \quad \text{if} \quad k < \left\{\frac{T_p(1)+t_{en}+t_{ex}}{2h_{max}} - \frac{1}{2}\right\}. \tag{3.13}$$

Proof. Let $\Delta\Phi_p(k) = [\Phi_p(k+1) - \Phi_p(k)]$. From Lemma 3.2,

$$\Delta\Phi_p(k) \geqslant T_{ps}(k) + T_{pc}(k). \tag{3.14}$$

Since the idle time of each processor cannot be larger than $(T_{ps} + T_{pc})$, we have

$$\Phi_p(k) \leqslant (k-1)[T_{ps}(k) + T_{pc}(k)]. \tag{3.15}$$

By Theorem 3.1 and Eqs. (3.1), (3.14), and (3.15), the condition that guarantees the monotonic increase of $k\cdot T_p^2(k)$ with k is

$$(k+1)T_p^2(k+1) > k\cdot T_p^2(k) \quad \text{if}$$
$$[T_{ps}(k) + T_{pc}(k)] > \frac{T_p(1) + (k-1)[T_{ps}(k) + T_{pc}(k)]}{2k},$$

or equivalently,

$$(k+1)T_p^2(k+1) > k\cdot T_p^2(k) \quad \text{if} \quad k > \left(\frac{T_p(1)}{T_{ps}(k)+T_{pc}(k)} - 1\right). \tag{3.16}$$

On the other hand, when all tasks but those in the critical path can be completed by $(k-1)$ processors during $h_{max} - [T_{ps}(k) + T_{pc}(k)]$ time units, i.e.,

$$(k-1) > \frac{T_p(1) - h_{max}}{h_{max} - [T_{ps}(k) + T_{pc}(k)]}, \tag{3.17}$$

the phase boundary B_{ac} must not be in a single e-level. As discussed before, the number of processors to achieve the optimal processor-time efficiency cannot be larger than the RHS of Eq. (3.17) plus one. By Lemma 3.4 and Eqs. (3.16) and (3.17),

$$(k+1)T_p^2(k+1) > k\cdot T_p^2(k) \quad \text{if}$$
$$k > \min\left\{\frac{T_p(1)}{d(s_k)+l(c_k)} - 1, \frac{T_p(1)}{h_{max} - [d(s_k)+l(c_k)]}\right\}. \tag{3.18}$$

The condition described in Eq. (3.12) is obtained from Eq. (3.18).

Note that $\Delta\Phi_p(k) \leqslant h_{max}$, and that $k\cdot T_p(k) > [T_p(1) + t_{en} + t_{ex}]$ according to Lemma 3.1 and Eqs. (3.1) and (3.11). By Theorem 3.1, the following result can be derived.

$$(k+1)T_p^2(k+1) < k\cdot T_p^2(k) \quad \text{if} \quad h_{max} < \frac{T_p(1)+t_{en}+t_{ex}}{2k+1}$$

which is equivalent to Eq. (3.13). ∎

To find the number of processors that achieves the optimal processor-time efficiency, we need to search the small region of k defined by Theorem 3.3. The lower and upper bounds of this region are, respectively, $\{T_p(1)/(2h_{max}) - 1\}$ and $\{2T_p(1)/(h_{max}) + 1\}$. Note that we have not made any assumption about the distribution of task times in deriving these bounds. Since $k\cdot T_p^2$ is a concave function of k (Theorem 3.2), the desirable number of processors can be found efficiently by a binary search. The binary search can be completed within about $\log_2(T_p(1)/h_{max})$ steps. Each step in the binary search tests whether $\Delta(k\cdot T_p^2(k))$ is positive. If it is, then a smaller value of k will be checked in the next step, otherwise, a larger k will be tested.

For any k inside the search region, the phase boundary $B_{ac}(k)$ is in a single e-level; hence the location of $B_{ac}(k)$ can be uniquely determined without knowing the detailed schedule. Accordingly,

$$\Phi_p(k) = k[T_{ps}(k) + T_{pc}(k)] - [N_{ps}(k) + N_{pc}(k)],$$

where $N_{ps}(k)$ and $N_{pc}(k)$ are the amount of task times in the splitting and combining phases. From Lemma 3.4, $T_{pc}(k) = l(c_k)$, and $T_{ps}(k)$ can be found directly from the e-outin tree. As a result,

$$k\cdot T_p^2(k) = \frac{[T_p(1) + \Phi_p(k)]^2}{k}.$$

For instance, suppose that N items need to be sorted. It is well known that $T(1) = N\cdot\log_2 N$ if a merge-sort algorithm is used. In this case, the overhead in the intree part dominates that of the outtree part. For the intree part, $h_{max} = N + N/2 + \cdots + 1 = 2N - 1$, so the lower and upper bounds of the search region can be determined from Theorem 3.3, which are close to $(\log_2 N)/4$ and $\log_2 N$, respectively. Since there are only $(3\cdot\log_2 N)/4$ candidate values in this search region, $\log_2\log_2 N$ steps of a binary search can guarantee to find the optimal number of processors for parallel merge-sorting. For problems such as evaluating numerical or logic expressions and finding the maximum (or minimum) value, all task times are identical. Theorem 3.3 predicts that the optimal degree of parallelism is between $N/(2 \cdot \log_2 N)$ and $2N/\log_2 N$. Figure 6 shows the simulation results of applying a nonpreemptive CPS algorithm to a binary intree of 4096 terminal nodes and $t_i = 1$ for all i. Since all tasks have unit execution times, the performance of the nonpreemptive CPS algorithm is very close to that of GS algorithm (see Eq. (2.2)). In this example, $k \cdot T^2$ is minimum when 431 processors are used, which is between $N/(2 \cdot \log_2 N)$ (=170) and $2N/\log_2 N$ (=683).

The above analysis reveals that the number of processors for optimizing processor-time efficiency in evaluating any given outin tree is related to the following parameters, all of which can be obtained easily from the original outin tree:

(a) $T(1)$, the time required by a sequential evaluation, which is the sum of all task times in the outin tree;

(b) $h_{\max}$, the length of the critical path;

(c) $d(s_k)$, the depth from the entry node to s_k, the maximum-all-busy level; and

(d) $l(c_k)$, the length from c_k, the minimum-all-busy level, to the exit node (recall that c_k and s_k depend on k).

$T(1)/h_{\max}$ reflects the shape of the outin tree, while $T(1)/d(s_k)$ and $T(1)/l(s_k)$ reflect the distribution of the task times. If the outin tree is "wide" and nearly balanced, i.e., $T_p(1)/h_{\max}$ is large, then a large number of processors are more appropriate. Further, if tasks in levels closer to the entry and exit nodes have long execution times, i.e., $T_p(1)/[d(s_k) + l(c_k)]$ is small, then the degree of parallelism should be small in order to optimize the processor-time efficiency. Both $T(1)/h_{\max}$ and $T(1)/[d(s_k) + l(c_k)]$ are related to the problem complexity.

The influence of problem complexity on the optimal degree of parallelism is again shown in the next section with respect to nonpreemptive scheduling algorithms.

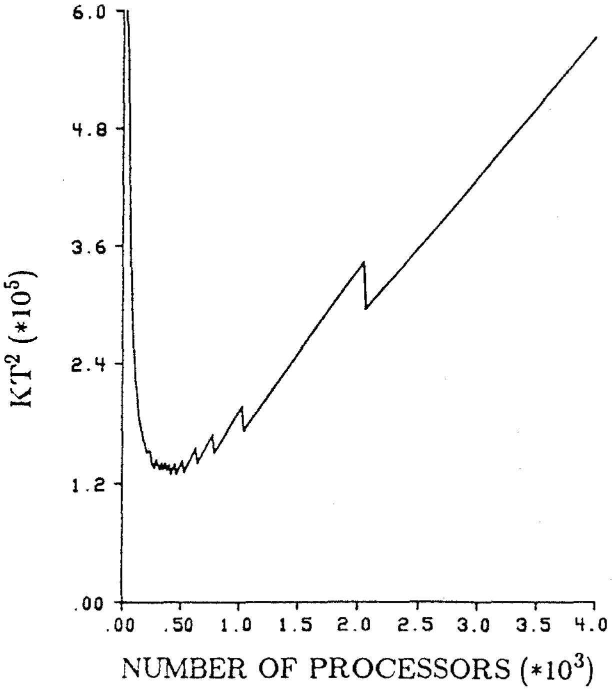

FIG. 6. Simulation results to find the number of processors to achieve the optimal processor-time efficiency in evaluating an intree which has 4096 leaves and unit execution times for all nodes using a non-preemptive CpS algorithm.

4. OPTIMAL DEGREE OF PARALLELISM IN NONPREEMPTIVE SCHEDULING

Nonpreemptive CPS algorithms are similar to the PCPS algorithm except that preemption is not allowed. In the nonpreemptive CPS algorithm, one processor is assigned to each of the k nodes farthest from the exit node. If there is a tie in lengths among more than one node, then a left-to-right tie-breaking rule is used to assign a processor to one of these nodes. When a task of the outin tree is completed, the free processor is assigned to the node farthest from the root in the remaining outin tree to be evaluated. Figure 2e illustrates a schedule obtained by a nonpreemptive CPS algorithm.

In general, nonpreemptive scheduling is more practical due to the smaller task-switching overheads; however, it is more difficult to predict its performance and determine the optimal degree of parallelism. The difficulty lies in the anomalous behavior of nonpreemptive CPS algorithms. Graham has proved that if an AND tree is evaluated twice by using k_1 and k_2 processors, respectively [15], then

$$\frac{T_{\mathrm{np}}(k_1)}{T_{\mathrm{np}}(k_2)} \leqslant \left\{1 + \frac{k_2 - 1}{k_1}\right\}.$$

The above inequality implies that the anomaly $T_{\mathrm{np}}(k+1)/T_{\mathrm{np}}(k) < k/(k+1)$ is possible. In other words, $k \cdot T_{\mathrm{np}}^2(k)$ is generally not a concave function of k and cannot be searched by a binary search or other efficient search methods. This phenomenon is illustrated in Fig. 6.

In a special case, if the execution times of tasks of an outtree are monotonically decreasing as the tree is decomposed, then it is shown in Lemma 4.1 below that $\Phi_{\mathrm{np}}(k_2) > \Phi_{\mathrm{np}}(k_1)$ holds for $k_2 > 2k_1$. Likewise, the same relation holds for the case when the execution times of tasks of an intree are monotonically increasing as the tree is composed. Using the conditions developed for $\Phi_{\mathrm{np}}(k_2)$ and $\Phi_{\mathrm{np}}(k_1)$, we show in Theorem 4.1 the conditions under which $k \cdot T^2$ is monotonically increasing or decreasing with k. These results demonstrate that the optimal degree of parallelism of outintree computations based on a nonpreemptive CPS algorithm can be bounded in a relatively small region. The assumption on monotonic distribution of task times is valid in many divide-and-conquer algorithms.

LEMMA 4.1. *Suppose that an outin tree is scheduled by a nonpreemptive CPS algorithm and that $t_i > t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then*

$$[\Phi_{np}(k_2) - \Phi_{np}(k_1)] \geqslant \{(k_2 - k_1)[T_{ps}(k_1) + T_{pc}(k_1)] - k_1 t_{npa}(k_1)\} > 0 \quad \text{if} \quad k_2 > 2k_1 \tag{4.1}$$

$$[\Phi_{np}(k_2) - \Phi_{np}(k_1)] \leqslant \{(k_2 - k_1)[T_{ps}(k_2) + T_{pc}(k_2)] + k_2 t_{npa}(k_2)\} \quad \text{if} \quad k_2 > k_1, \tag{4.2}$$

where $t_{npa}(k)$ is the longest task time among all tasks in the all-busy phase when k processors are used.

Proof. Since the same behavior appears in the splitting phase under both preemptive and nonpreemptive scheduling, and only tasks in the all-busy phase are involved in the proof of Eq. (2.1) [20], it is easy to refine Kaufman's results as follows.

$$T_p(k) \leqslant T_{np}(k) \leqslant T_p(k) + t_{npa}(k). \tag{4.3}$$

By Eq. (4.3), we have

$$k_2 T_{np}(k_2) \geqslant k_2 T_p(k_2) \tag{4.4}$$

and

$$k_1 T_{np}(k_1) \leqslant k_1 T_p(k_1) + k_1 t_{npa}(k_1). \tag{4.5}$$

In Lemma 3.2, we have shown that $\Phi_p(k + 1) - \Phi_p(k) \geqslant [T_{pc}(k) + T_{ps}(k)]$. From Proposition 2.4 and Eq. 3.1, we get, for $k_2 > k_1$,

$$[k_2 T_p(k_2) - k_1 T_p(k_1)] = [\Phi_p(k_2) - \Phi_p(k_1)] \geqslant [(k_2 - k_1)(T_{ps}(k_1) + T_{pc}(k_1))]. \tag{4.6}$$

By Eqs. (4.4)–(4.6) and the hypothesis of monotonicity of the task times, which guarantees that $(T_{ps}(k_1) + T_{pc}(k_1)) > t_{npa}(k_1)$, Eq. (4.1) is proved. The proof for Eq. (4.2) is analogous. ■

We should point out that the above lemma holds for the case in which a part of the phase boundary is in the intree and another part is in the outtree. The above lemma is true because the task time of a node in the all-busy phase is less than either T_{ps} or T_{pc} from the monotonicity assumption on task times; that is, $[T_{ps}(k_1) + T_{pc}(k_1)] > t_{npa}(k_1)$ is always true regardless of the location of the phase boundary.

Similar to Theorem 3.1, we first study the relationship between $k \cdot T^2$ and the idle times. The following theorem gives the conditions under which $k \cdot T^2$ is monotonically increasing or decreasing based on the intermediate variable $\Phi_{np}(k)$.

THEOREM 4.1. *Suppose that an outin tree is scheduled by a nonpreemptive CPS algorithm and that $t_i > t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then*

$$k_2 T^2_{np}(k_2) > k_1 T^2_{np}(k_1) \quad \text{if} \quad [\Phi_{np}(k_2) - \Phi_{np}(k_1)] > \left[\frac{k_2 - k_1}{2} T_{np}(k_1)\right] \quad \text{and} \quad k_2 > 2k_1; \tag{4.7}$$

$$k_2 T^2_{np}(k_2) < k_1 T^2_{np}(k_1) \quad \text{if} \quad [\Phi_{np}(k_2) - \Phi_{np}(k_1)] < \left[\frac{2(k_2 - k_1)k_1}{2k_1 + 3k_2} T_{np}(k_1)\right] \quad \text{and} \quad k_2 > k_1. \tag{4.8}$$

Proof. From Eq. (3.1), we have

$$\begin{aligned}[k_2 T^2_{np}(k_2) - k_1 T^2_{np}(k_1)] &= \left\{\frac{(T_{np}(1) + \Phi_{np}(k_2))^2}{k_2} - \frac{(T_{np}(1) + \Phi_{np}(k_1))^2}{k_1}\right\}. \\ &= (k_1\{[\Phi_{np}(k_2) - \Phi_{np}(k_1)]\cdot[2T_{np}(1) + \Phi_{np}(k_2) \\ &\quad + \Phi_{np}(k_1)]\} - (k_2 - k_1)[k_1 T_{np}(k_1)]^2)/k_1 \cdot k_2.\end{aligned} \tag{4.9}$$

From Lemma 4.1, if $k_2 > 2k_1$, then $\Phi_{np}(k_2) > \Phi_{np}(k_1)$, which implies that $k_2 T_{np}(k_2) > k_1 T_{np}(k_1)$ and that $(2T_{np}(1) + \Phi_{np}(k_2) + \Phi_{np}(k_1)) > 2k_1 T_{np}(k_1)$. Hence, we obtain Eq. (4.7) from Eq. (4.9).

On the other hand, from Lemma 4.1,

$$\Phi_{np}(k_2) \leqslant [\Phi_{np}(k_1) + (k_2 - k_1) \times (T_{ps}(k_2) + T_{pc}(k_2)) + k_2 t_{npa}(k_2)].$$

Note that in intrees and outin trees, t_{ex}, the execution time of the exit node, must be included in h_{max}, and $t_{ex} > t_{npa}$. hence,[1]

$$t_{npa}(k_2) < h_{max}/2. \tag{4.10}$$

Moreover, $[T_{ps}(k_2) + T_{pc}(k_2)] \leqslant h_{max} \leqslant T_{np}(k_1)$. Thus,

$$(2T_{np}(1) + \Phi_{np}(k_2) + \Phi_{np}(k_1)) \leqslant \left(2k_1 T_{np}(k_1) + (k_2 - k_1)T_{np}(k_1) + \frac{k_2 T_{np}(k_1)}{2}\right)$$

[1] For outin trees, $t_{npa}(k) < h_{max}/3$ because t_{en} and t_{ex}, the execution times of entry and exit nodes, must be included in h_{max}, and $[t_{en} + t_{ex}] > 2t_{npa}$.

$$= \left(k_1 + \frac{3k_2}{2}\right) T_{\mathrm{np}}(k_1),$$

and Eq. (4.8) follows from Eq. (4.9). ■

The main theorem to find the optimal degree of parallelism can be derived from Theorem 4.1. Before this theorem is proved, the following lemma is needed.

LEMMA 4.2. *For any given outin tree, suppose that both the PCPS and the nonpreemptive CPS algorithms are applied, then* $[T_{\mathrm{nps}}(k) + T_{\mathrm{npc}}(k)] \leq [T_{\mathrm{ps}}(k) + T_{\mathrm{pc}}(k) + t_{\mathrm{npa}}(k)]$.

Proof. By observing the PCPS and nonpreemptive CPS algorithms, it is easy to see that $T_{\mathrm{nps}}(k) = T_{\mathrm{ps}}(k)$. From Kaufman's proof of Eq. (2.1) [20], we have

$$T_{\mathrm{npc}}(k) < [T_{\mathrm{pc}}(k) + t_{\mathrm{npa}}(k)]. \quad ■ \tag{4.11}$$

The example in Fig. 2 illustrates this lemma. Here, $[(T_{\mathrm{nps}} + T_{\mathrm{npc}}) - (T_{\mathrm{ps}} - T_{\mathrm{pc}})] = 1$, which is less than t_{npa} (=3).

THEOREM 4.2. *Suppose that an outin tree is scheduled by a nonpreemptive CPS algorithm and that $t_i > t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then k, the number of processors that minimizes $k \cdot T^2_{\mathrm{np}}(k)$, is bounded between $[T_{\mathrm{np}}(1) + t_{\mathrm{en}} + t_{\mathrm{ex}}]/(8h_{\max})$ and $3T_{\mathrm{np}}(1)/[d(s_k) + l(c_k) - 2t_{\mathrm{npa}}(k)]$.*

Proof. From Lemma 4.1 and Theorem 4.1, we obtain, for $k_2 > 2k_1$, that

$$k_2T^2_{\mathrm{np}}(k_2) > k_1T^2_{\mathrm{np}}(k_1) \quad \text{if} \quad \{(k_2 - k_1)[T_{\mathrm{ps}}(k_1) + T_{\mathrm{pc}}(k_1)] - k_1t_{\mathrm{npa}}(k_1)\} > \left\{\frac{k_2 - k_1}{2} T_{\mathrm{np}}(k_1)\right\}. \tag{4.12}$$

From Eq. (3.1), Lemma 4.2, and the fact that $\Phi_{\mathrm{np}}(k_1) < \{k_1[T_{\mathrm{nps}}(k_1) + T_{\mathrm{npc}}(k_1)]\}$, the condition in Eq. (4.12) can be rewritten as

$$\{(k_2 - k_1)[T_{\mathrm{ps}}(k_1) + T_{\mathrm{pc}}(k_1)] - k_1t_{\mathrm{npa}}(k_1)\} > \left\{\frac{k_2 - k_1}{2k_1}[T_{\mathrm{np}}(1) + k_1(T_{\mathrm{ps}}(k_1) + T_{\mathrm{pc}}(k_1) + t_{\mathrm{npa}}(k_1))]\right\} \quad \text{for} \quad k_2 > 2k_1. \tag{4.13}$$

A simple algebraical manipulation of Eq. (4.13) yields the condition

$$k_1 > \frac{T_{\mathrm{np}}(1)}{T_{\mathrm{ps}}(k_1) + T_{\mathrm{pc}}(k_1) - \dfrac{k_2 + k_1}{k_2 - k_1} t_{\mathrm{npa}}(k_1)} \quad \text{for} \quad k_2 > 2k_1. \tag{4.14}$$

When $k_2 = 3k_1$, Eq. (4.14) becomes

$$k_1 > \frac{T_{\mathrm{np}}(1)}{T_{\mathrm{ps}}(k_1) + T_{\mathrm{pc}}(k_1) - 2t_{\mathrm{npa}}(k_1)}. \tag{4.15}$$

Since $[(k_2 + k_1)/(k_2 - k_1)]$ is monotonically decreasing with increasing k_2, Eq. (4.15) and the condition $k_2 > 3k_1$ imply Eq. (4.14). Note that Eq. (4.15) assures that for any k larger than $\max[3, 3T_{\mathrm{np}}(1)/(T_{\mathrm{ps}}(k) + T_{\mathrm{pc}}(k) - 2t_{\mathrm{npa}}(k)]$, we can find $k' < k/3$ such that $k'T^2_{\mathrm{np}}(k') < kT^2_{\mathrm{np}}(k)$. As a result, all numbers larger than $[3T_{\mathrm{np}}(1)/(T_{\mathrm{ps}}(k) + T_{\mathrm{pc}}(k) - 2t_{\mathrm{npa}})]$ can be excluded from consideration. The upper bound of the region containing the optimal number of processors is obtained by applying Lemma 3.4.

From Lemma 4.1, Theorem 4.1, and the fact that $[T_{\mathrm{ps}}(k) + T_{\mathrm{pc}}(k)] < h_{\max}$, we get

$$k_2T^2_{\mathrm{np}}(k_2) < k_1T^2_{\mathrm{np}}(k_1) \quad \text{if} \quad \left\{h_{\max} + \frac{k_2t_{\mathrm{npa}}(k_2)}{k_2 - k_1}\right\} < \left\{\frac{2k_1T_{\mathrm{np}}(k_1)}{2k_1 + 3k_2}\right\} \quad \text{and} \quad k_2 > k_1. \tag{4.16}$$

When $k_2 = 2k_1$, by Eqs. (3.1), (3.11), and (4.10), the condition in Eq. (4.16) can be rewritten as

$$k_1 < \frac{T_{\mathrm{np}}(1) + t_{\mathrm{en}} + t_{\mathrm{ex}}}{8h_{\max}}. \tag{4.17}$$

This gives the lower bound of the optimal number of processors; that is, for any k less than $(T_{\mathrm{np}}(1) + t_{\mathrm{en}} + t_{\mathrm{ex}})/(8h_{\max})$, we can find $k' = 2k$ such that $k'T^2_{\mathrm{np}}(k') < kT^2_{\mathrm{np}}(k)$. ■

As an example, we can determine the area within which the number of processors that optimizes processor-time efficiency can be found for the parallel merge-sort of N elements. In this problem, the computational overhead in the intree is dominant, so only the part of the intree has to be considered in the scheduling. From Theorem 4.2, the lower bound of the search region is $(\log_2 N)/16$, since $T_{\mathrm{np}}(1) = N \cdot \log_2 N$ and $h_{\max} < 2N$. If N is large enough, then $[d(s_k) + l(c_k) - 2t_{\mathrm{npa}}(k)]$ will be larger than $1.5N$; hence, the upper bound of the search region is $2 \cdot \log_2 N$. In contrast, recall that the search region to this problem is bounded by $(\log_2 N)/4$ and $\log_2 N$ when preemptive scheduling is used.

Comparing these bounds with those in Theorem 3.3, we see that the range within which the optimal number of processors using nonpreemptive scheduling can be found is larger than that of preemptive scheduling. Moreover, $k \cdot T^2$ is not monotonically decreasing or increasing with k for nonpreemptive scheduling; i.e., $k \cdot T^2$ is not a unimodal function of k. Hence, an exhaustive search is required to find the optimal degree of parallelism in the region bounded by Theorem 4.2.

5. CONCLUSIONS

In this paper, we have (a) proved that the critical-path scheduling algorithm is optimal in minimizing the completion time of evaluating any given outin tree on a fixed number of identical parallel processors when preemptions are allowed, and (b) derived tight bounds on the number of processors within which the processor-time efficiency of the parallel evaluation of any given outin tree under preemptive and nonpreemptive scheduling can be optimized. According to our efficiency analysis, we found that the optimal degree of parallelism depends on the problem complexity, the shape of the precedence graph (balanced or skewed), and the task-time distribution along each path (random or monotonic).

The complexity of each node in the outin tree is an important factor that influences the optimal degree of parallelism. If the overhead of each node is high with respect to the size of its input (such as $\Theta(2^n)$ for inputs of size n) and a large number of processors are used, then the processor-time efficiency must be poor regardless of the capacity of the interconnection network. In this case, the time needed to evaluate a subproblem is increased rapidly during the decomposition phase in the outtree and the composition phase in the intree. Hence, the root and exit nodes of the tree are obvious bottlenecks. In contrast, if the complexity of each node is a constant, then the root and exit nodes are not bottlenecks. Examples of this kind of problems include finding the maximum and evaluating an arithmetic expression. Here, a computing system with a large number of processors is appropriate, and a large speedup is expected. Tree-structured computer architectures [17, 25] and virtual-tree computers [6] are good candidates in these applications. In cases in which the overhead of each node is small with respect to the size of its inputs (such as $\Theta(n)$ or $\Theta(\log n)$ for inputs of size n), then the time needed to evaluate a subproblem is increased slowly during the decomposition phase in the outtree and the composition phase in the intree. A computing system with a moderate number of processors is more cost-effective. For example, to sort 4000 elements by a parallel merge-sort algorithm, using 10 to 12 processors is a good choice.

The shape of the outin tree is another important factor to be considered. Let $T_p(1)/h_{\max}$ be "average width" of an outin tree. The optimal degree of parallelism is found to depend strongly on the average width. If the outin tree is "wide," then the degree of parallelism is high and the granularity can be small. On the other hand, if the outin tree is "narrow," then the degree of parallelism is low and the granularity is necessarily large. In order to achieve a high processor-time efficiency, a narrow tree may have to be restructured to arrive at a different representation.

Lastly, the distribution of task times is an important factor that influences the processor-time efficiency. For many practical problems, especially when divide-and-conquer algorithms are used, the precedence graph is nearly balanced and the task times of all nodes in each level are approximately equal. In this case, well-balanced workloads with overlapped process communications can be assigned to processors connected in a SIMD model. The optimal degree of parallelism is, therefore, close to the theoretical one predicted in this paper. On the other hand, for problems represented as irregular outin trees and for problems in which the overhead of each task may be data dependent, it is important that the underlying computer architecture support the dynamic distribution of workload. For a computer system with a high degree of parallelism, an efficient interconnection network is needed. In a computing system with a low degree of parallelism, an effective load balancing mechanism is necessary. The correct analysis for the latter system should, therefore, include the communication overhead, as process communications may not overlap with computations. Consequently, the optimal degree of parallelism may be less than the theoretical value predicted in this paper.

REFERENCES

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and algebra of programs. *Comm. ACM* **21,** 8 (1978), 613–641.
2. Bentley, J. L., and Shamos, M. I. Divide and conquer in linear expected time. *Inform. Processing Lett.* **7,** 2 (1978), 87–91.
3. Bentley, J. L., and Kung, H. T. A tree machine for searching problems. *Proc. Int. Conf. on Parallel Processing.* IEEE, 1979, pp. 257–266.
4. Bentley, J. L. Multidimensional divide-and-conquer. *Comm. ACM* **23** (1980), 51–59.
5. Bentley, J. L., Haken, D., and Saxe, J. B. A general method for solving divide-and-conquer recurrences. *SIGACT News* (Fall 1980), 36–44.
6. Burton, F. M., and Huntbach, M. M. Virtual tree machines. *IEEE Trans. Comput.* **C-33,** 3 (1984), 278–280.
7. Coffman, E. G., Jr., and Graham, R. H. Optimal scheduling for two processors systems. *Acta Inform.* **1,** 3 (1972), 200–213.
8. Coffman, E. G., Jr. (Ed.) *Computer and Job-Shop Scheduling Theory.* Wiley, New York, 1976.
9. Conery, J. S., and Kibler, D. F. AND parallelism and nondeterminism in logic programs. *New Generation Comput.* **3,** 1 (1985), 43–70.
10. Conway, R. W., Maxwell, W. L., and Miller, L. W. *Theory of Scheduling.* Addison-Wesley, Reading, MA, 1967.
11. Dennis, J. B. Data flow supercomputers. *Computer* **13,** 11 (Nov. 1980), 48–56.
12. Dolev, D., and Warmuth, M. Profile scheduling of opposing forests and level orders. *SIAM J. Algorithm Discrete Math.* **6,** 4 (Oct. 1985), 665–687.
13. Garey, M. R., and Johnson, D. S. Scheduling tasks with nonuniform deadlines on two processors. *J. ACM* **23,** 3 (1976), 461–467.
14. Garey, M. R., Johnson, D. S., Tarjan, R. E., and Yannakakis, M. Scheduling opposing forests. *SIAM J. Algorithm Discrete Math.* **4,** 1 (Mar. 1983), 72–93.
15. Graham, R. L. Bounds for certain multiprocessing anomalies. *Bell Sys. Tech. J.* **45,** 9 (Nov. 1966), 1563–1581.

16. Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. N. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Math.* **5** (1979), 287–326.
17. Harris, J. A., and Smith, D. R. Simulation experiments of a tree organized multicomputer. *Proc. 6th Annual Symp. on Computer Architecture.* IEEE/ACM, Apr. 1979, pp. 83–89.
18. Horowitz, E., and Zorat, A. Divide-and-conquer for parallel processing. *Trans. Comput.* **C-32,** 6 (June 1983), 582–585.
19. Hu, T. C. Parallel sequencing and assembly line problems. *Oper. Res.* **9,** 6 (1961), 841–848.
20. Kaufman, M. An almost-optimal algorithm for the assembly line scheduling problem. *Trans. Comput.* **C-23,** 11 (1974), 1169–1174.
21. Keller, R. M., Lin, F. C. H., and Tanaka, J. Rediflow multiprocessing. *Proc. COMPCON Spring.* IEEE, 1984, pp. 410–417.
22. Kunde, M. Nonpreemptive LP-scheduling on homogeneous multiprocessor systems. *SIAM J. Comput.* **10,** 1 (Feb. 1981), 151–173.
23. Lenstra, J. K., Rinnooy Kan, A., and Brucker, P. Complexity of machine scheduling problems. *Proc. Discrete Mathematics.* North-Holland, Amsterdam, 1977, pp. 343–362.
24. Lloyd, E. L. Critical path scheduling with resource and processor constrains. *J. ACM* **29,** 3 (1982), 781–811.
25. Mago, G. Making parallel computation simple: The FFP machine. *Proc. COMPCON Spring.* IEEE, 1985, pp. 424–428.
26. Monier, L. Combinatorial solutions of multidimensional divide-and-conquer recurrences. *J. Algorithms* **1,** (1980), 60–74.
27. Muntz, R., and Coffman, E., Jr. Preemptive scheduling of real-time task on multiprocessor systems. *J. ACM* **17,** 2 (Apr. 1970), 324–338.
28. Peters, F. J. Tree machine and divide-and-conquer algorithms. *Lecture Notes CS 111 (CONPAR81).* Springer-Verlag, New York/Berlin, 1981, pp. 25–35.
29. Schindler, S. On optimal scheduling for multiprocessor systems. *Proc. Princeton Conf. on Information Science and Systems,* 1972, pp. 219–223.
30. Tang, C. Y., and Lee, R. C. T. Optimal speeding up of parallel algorithm based on the divide-and-conquer strategy. *Inform. Sci.* Elsevier, Amsterdam/New York, 1984, pp. 173–186.
31. Ullman, J. D. NP-complete scheduling problems. *J. Comput. Syst. Sci.* **10** (1975), 384–393.

Received March 15, 1989; accepted August 24, 1989

BENJAMIN W. WAH received the Ph.D. degree in computer science from the University of California, Berkeley, California, in 1979. He has been on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, Indiana, between 1979 and 1985. He is now a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana–Champaign, Urbana, Illinois. Between 1988 and 1989, he served as a program director of the Microelectronic Systems Architecture Program, National Science Foundation. He has published extensively in the areas of computer architecture, parallel processing, artificial intelligence, distributed databases, and computer networks. For his contributions to research, he has been selected as a University Scholar of the University of Illinois in 1989. Dr. Wah is the Associate Editor-in-Chief of the *IEEE Transactions on Knowledge and Data Engineering,* an area editor of the *Journal of Parallel and Distributed Computing,* and an editor of *Information Sciences.* He serves as a member of the Governing Board of the IEEE Computer Society, Program Cochair of the 1990 International Conference on Parallel Processing, Program Chair of the 1991 Distributed Computing Systems Conference, and a program evaluator for ABET (computer Engineering) and CSAC (computer science). Previously, he served on the editorial board of the *IEEE Transactions on Software Engineering* and was a Distinguished Visitor of the IEEE Computer Society.

GUO-JIE LI is a professor at the Institute of Computing Technology, Academia Sinica, and the Director of the newly formed National Research Center for Intelligent Computing Systems, Beijing, China. His research interests include parallel processing, artificial intelligence, theory of algorithms, and computer architecture. Li graduated from Peking University in 1968 and received the M.S. degree in computer science and engineering from the University of Science and Technology of China and the Institute of Computing Technology, Academia Sinica, in 1981. He received the Ph.D. degree in electrical engineering in 1985 from Purdue University. Between 1985 and 1986, he was a postdoctoral research associate in the Coordinated Science Laboratory of the University of Illinois at Urbana–Champaign. Li has published over 30 papers and has co-edited *Tutorial: Computers for Artificial Intelligence Applications* (IEEE Computer Society Press, 1986). He has also written several chapters in several published books. He is an editor of the *Journal of Computer Science and Technology* and the *Journal of Pattern Recognition and Artificial Intelligence.*