View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Calhoun, Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Faculty and Researcher Publications

Faculty and Researcher Publications

1995-12

Task Scheduling in Multiprocessing Systems

El-Rewini, Hesham

http://hdl.handle.net/10945/41234

http://www.nps.edu/library

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

Task Scheduling in Multiprocessing Systems

Hesham El-Rewini and Hesham H. Ali
University of Nebraska at Omaha

Ted Lewis
Naval Postgraduate School

The complex problem of assigning tasks to processing elements in order to optimize a performance measure has resulted in numerous heuristics aimed at approximating an optimal solution.

Jobs needing to be processed in a manufacturing plant, bank customers waiting to be served by tellers, aircraft waiting for landing clearances, and program tasks to be run on a parallel or distributed computer: What do these situations have in common? They all encounter the scheduling problem that emerges whenever there is a choice concerning the order in which tasks can be performed and the assignment of tasks to servers for processing. In general, the scheduling problem assumes a set of resources and a set of consumers serviced by those resources according to a certain policy. The nature of the consumers and resources as well as the constraints on them affect the search for an efficient policy for managing the way consumers access and use the resources to optimize some desired performance measure. Thus, a scheduling system comprises a set of consumers, a set of resources, and a scheduling policy.

A task in a program, a job in a factory, and a customer in a bank are examples of consumers. A processing element in a computer system, a machine in a factory, and a teller in a bank are examples of resources. First come, first served is an example of a scheduling policy. Scheduling policy performance varies with circumstances. While the equitable first-come, first-served policy is appropriate in a bank, it may not be the best policy for jobs on a factory floor or tasks in a computer system. This article addresses the task scheduling problem in many of its variations and surveys the major solutions.

The scheduling techniques we discuss might be used by a compiler writer to optimize the code that comes out of a parallelizing compiler. The compiler would produce grains of sequential code, and the optimizer would schedule these grains such that the program runs in the shortest time. Another use of these techniques is in the design of high-performance systems. A designer might want to construct a parallel application that runs in the shortest time possible on some arbitrary system. We believe the methods presented here can also be adapted to other (related) resource allocation problems of computing.

Suppose a team of programmers wants to divide a programming task into subsystems. Let's say each programmer depends on some other programmer(s) for interface specifications; otherwise, the programmers work in parallel. Thus, the team can be modeled as a parallel processor, and the program to be written can be modeled as parallel tasks. The scheduler now tries to find the assignment of subsystems to programmers that will let the team finish the program in the shortest time.

This article concerns scheduling program tasks on parallel and distributed systems. The tasks are the consumers, and they will be represented through the use of directed graphs called task graphs. The processing elements are the resources, and their interconnection networks can be represented through the use of undirected graphs. The scheduler generates a schedule in the form of a timing diagram called the Gantt chart, which is

0018-9162/95/$4.00 © 1995 IEEE

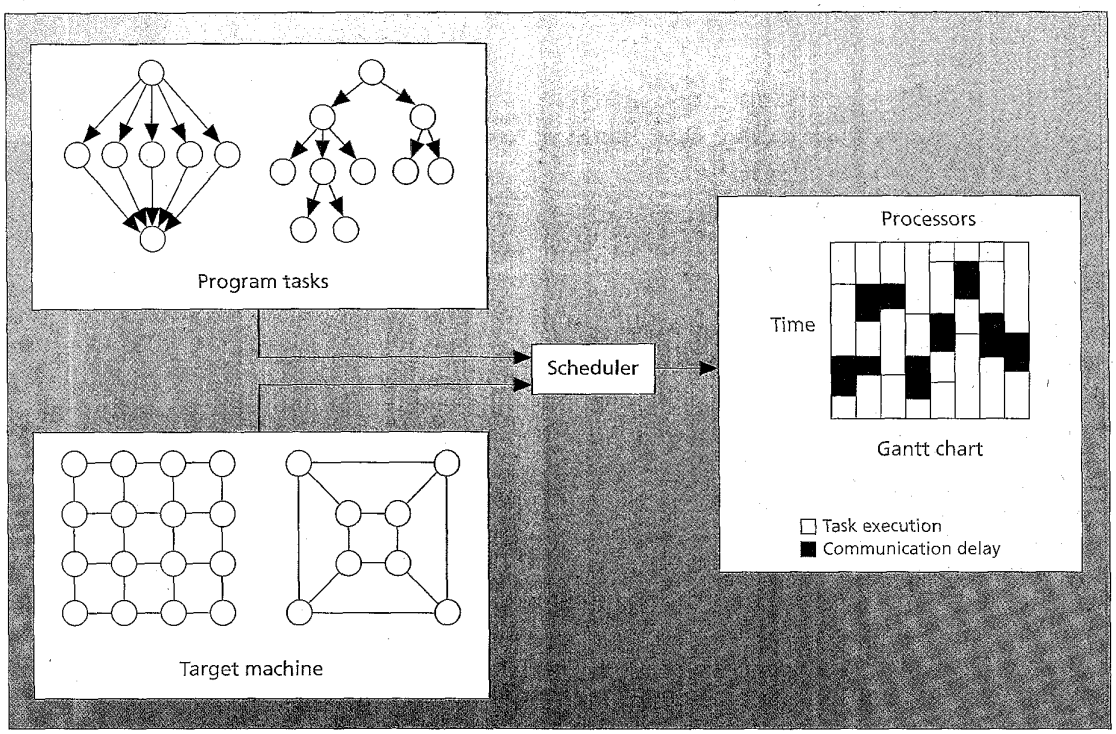


Figure 1. The components of a scheduling system.

used to illustrate the allocation of the parallel program tasks onto the target machine processors and the execution order of the tasks. Figure 1 shows the components of a scheduling system. Task graphs, target machines, and Gantt charts will be described in a later section.

We will provide models for representing parallel programs, parallel systems, and communication cost. Then, since task scheduling has been shown to be computationally intractable in many cases, we will summarize the NP-complete results. (See the sidebar "NP-completeness.") Solutions to this problem can be categorized into optimal algorithms for some restricted cases and heuristic algorithms for the more general cases. We present several optimal algorithms for solving the scheduling problem in some special cases. We then cover scheduling heuristics and summarize three scheduling tools.

SCHEDULING MODEL

A parallel program is modeled as a partially ordered set (poset) $P = (V, <)$, where V is a set of tasks. The relation $v_i < v_j$ in P implies that the computation of task v_j depends on the results of the computation of task v_i. In other words, task v_i must be computed before task v_j, and the result of the computation of task v_i must be known by the processor computing task v_j. Associated with each task v_i is its computation cost A_i. Associated with each arc (i, j) connecting tasks v_i and v_j is the communication cost D_{ij}. The partial order $<$ is conveniently represented as a directed acyclic graph called a task graph $G = (V, E)$. A directed edge (i, j) between two tasks v_i and v_j specifies that v_i must be completed before v_j can begin.

The target machine is assumed to be made up of an arbitrary number of processing elements. Each processing element can run one task at a time, and all tasks can be processed by any processing element. The processing elements are connected via an arbitrary interconnection topology that can be represented by an undirected graph. Three parameters, S_i, B_i, and I_i, are associated with each processing element p_i, where S_i is the speed, B_i is the time to initiate a task on p_i, and I_i is the time to initiate a message on p_i. Associated with each edge connecting two processing elements is the transfer rate over the link between the two processing elements.

Figure 2 shows an example of a task graph and a target machine. The task graph consists of nine nodes, with each node representing a task. The number shown in the upper portion of each node is the task number. The number in the lower portion of a node i represents the parameter A_i, and the number next to an edge (i, j) represents the parameter D_{ij}; for example, $A_1 = 6$, and $D_{12} = 25$. The target machine is an eight-node hypercube.

A schedule of the task graph $G = (V, E)$ on m processors is a function f that maps each task to a processor and a start-

NP-completeness

NP-complete problems are those that are strongly suspected to be computationally intractable. A problem belongs to the class P if a solution of the problem can be obtained by a polynomial-time algorithm. A problem belongs to the class NP if the correctness of a solution for the problem can be verified by a polynomial-time algorithm. A problem is in the class NP-hard if it is as hard as any problem in NP. Hence, every problem in NP is polynomial-time reducible to an NP-hard problem. Finally, NP-complete problems are the NP-hard problems that are also in NP.

ing time. Formally, $f: V \rightarrow \{1, 2, \ldots, m\} \times [0, \infty)$. If $f(v) = (i, t)$ for some $v \in V$, we say that task v is scheduled to be processed by processor p_i starting at time t. There exists no $u, v \in V$ such that $f(u) = f(v)$. Note that if $u < v$, $f(u) = (i, t_1)$, and $f(v) = (j, t_2)$, then $t_1 < t_2$. A schedule can be represented informally using Gantt charts. A Gantt chart consists of a list of all processors in the target machine, and for each processor a list of all tasks allocated to that processor ordered by their execution time, including task start and finish times. The scheduling goal is to minimize the total completion time of a parallel program. This performance measure is known as the schedule length or maximum finishing time. Figure 3 shows examples of Gantt charts.

COMMUNICATION COST MODELS

Two key components contribute to the total completion cost: execution time and communication delay. Several different models can be used to compute the communication delay. Here, we present three models that can be used to compute the cost of executing a parallel program on a set of processing elements. Communication delay is the key element differentiating these models. Given a task graph $G = (V, E)$ and its schedule f on m processors, we define $proc(v)$ to be the processor assigned to process task v in f for every $v \in V$. The three models, A, B, and C, are as follows:

Model A

In this model, the Gantt chart that represents the schedule f does not reflect the communication delay, but it shows that the precedence relations between tasks are preserved. Program completion cost can be computed as total cost = communication cost + execution cost, where execution cost = schedule length and communication cost = number of messages * communication delay per message. Here, the number of messages = the number of node pairs (u, v) such that $(u, v) \in E$ and $proc(u) \neq proc(v)$.

Model B

This model is similar to Model A, but it uses a more practical method in evaluating the number of messages. By counting each arc (u, v) such that $proc(u) \neq proc(v)$ as communication using Model A, we may be passing the same information between a given pair of processors several times. For instance, if tasks u, v, and $w \in V$ such that (u, v), $(u, w) \in E$ and $proc(v) = proc(w) \neq proc(u)$, then the result of computing u must be communicated to the processor computing v and w. This communication is counted twice in Model A and only once in Model B, as follows: Total cost = communication cost + execution cost, where execution cost = schedule length and communication cost = number of messages * communication delay per message. Here, number of messages = number of processor-task pairs (P, v) such that processor P does not compute task v but computes at least one direct successor of v.

Model C

In this model, we assume the existence of an I/O processor that is associated with every processor in the system. The term *processing element* is used to imply the existence

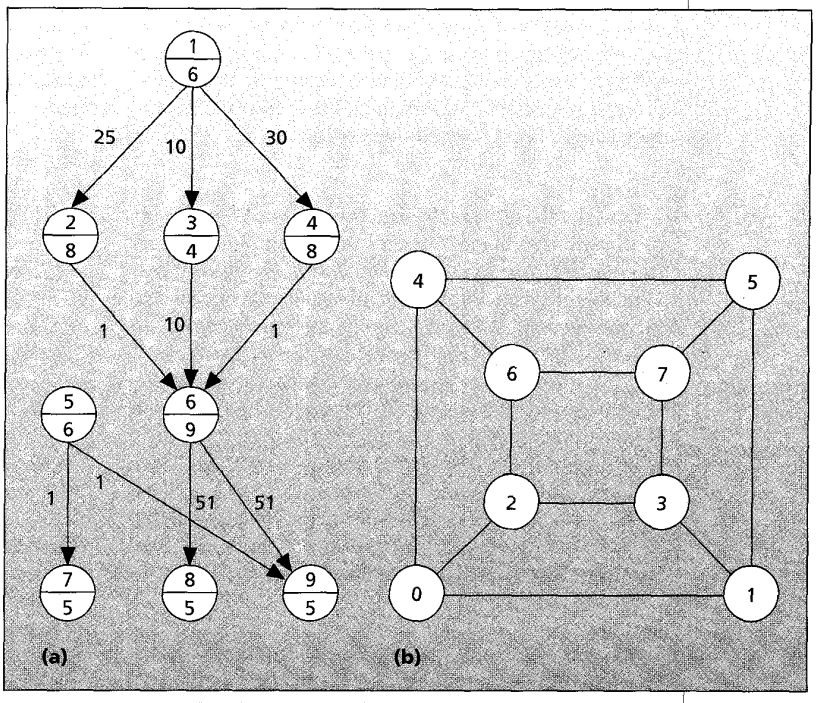


Figure 2. An example of a task graph (a) and a target machine (b).

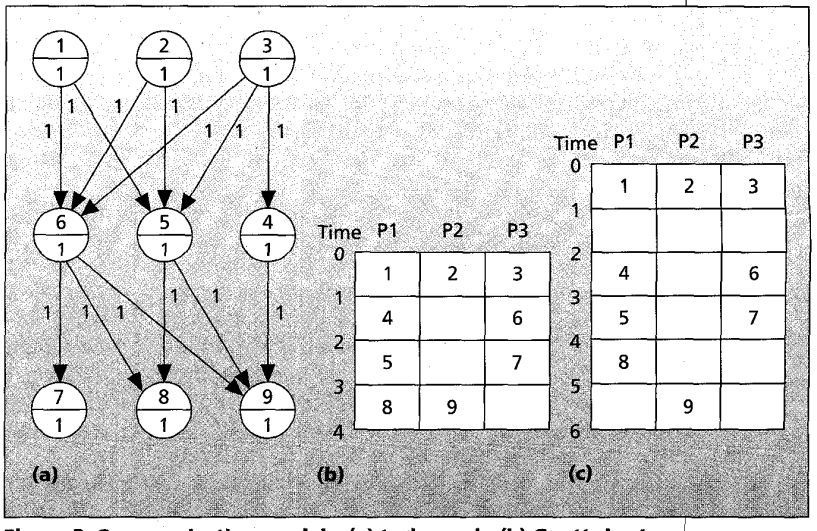


Figure 3. Communication models: (a) task graph, (b) Gantt chart (Models A and B), (c) Gantt chart (Model C).

of an I/O processor. A processing element can execute a task and communicate with another processing element at the same time. Communication time between two tasks allocated to the same processing element is assumed to be zero. For any two tasks $v_i, v_j \in V$, if $v_i < v_j$ and $f(v_i) = (k, t)$, then v_j should be scheduled on either

- processor p_k on time t_1 and $t_1 \geq t + A_i$ or
- processor p_l, $l \neq k$ on time t_2 and $t_2 \geq t + A_i + D_{ij}$.

When this model is used, the communication delay can be easily shown on the Gantt chart representing the schedule. Also note that a task can be scheduled in the communication holes in a Gantt chart. In other words, a task can be assigned to a processing element for execution while this processing element is communicating with another processing element. The program completion time is computed as total cost = schedule length.

Example 1

Consider the task graph given in Figure 3a. We compute the program completion cost of this task graph on three processing elements using the three models given above. Suppose that tasks are assigned arbitrarily to processors as follows: Tasks 1, 4, 5, and 8 are assigned to processor P1, tasks 2 and 9 are assigned to processor P2, and tasks 3, 6, and 7 are assigned to processor P3. In this example, we assume (for simplicity) that the communication delay per message equals one unit of time. The Gantt charts shown in Figure 3 reflect the assignment given above. The Gantt chart in Figure 3b shows only the precedence constraints on the tasks; it does not show the communication delay. This Gantt chart can be used to compute the completion cost using Models A and B as follows:

Model A

Execution cost = schedule length = 4 units of time
Number of messages = | (1, 6), (2, 5), (2, 6), (3, 4), (3, 5), (4, 9), (5, 9), (6, 8), (6, 9) | = 9
Communication cost = 9 * 1 units of time
Total cost = 4 + 9 = 13 units of time

Model B

Execution cost = schedule length = 4 units of time
Number of messages = | (P3, 1), (P1, 2), (P3, 2), (P1, 3), (P2, 4), (P2, 5), (P1, 6), (P2, 6) | = 8
Communication cost = 8 * 1 units of time
Total cost = 4 + 8 = 12 units of time

Model C

The Gantt chart of Figure 3c reflects both the precedence relation and the communication cost on the basis of Model C as follows:

Total cost = schedule length = 6 units of time

THE NP-COMPLETENESS OF THE SCHEDULING PROBLEM

In this section we list some of the NP-complete results in the scheduling problem. It has been proven that the problem of finding an optimal schedule for a set of tasks is NP-complete in the general case and in several restricted cases.[1] For a formal definition of the notion of NP-completeness and other related issues, refer to the literature.[1,2]

When communication cost is ignored

Below, we give the formal definition of some versions of the scheduling problem that were proven to be NP-complete.

- (Problem 1): General scheduling problem. Given a set T of n tasks, a partial order $<$ on T, weight A_i, $1 \leq i \leq n$, and m processors, and a time limit k, does there exist a total function h from T to $\{0, 1, \ldots, k-1\}$ such that

  (1) if $i < j$, then $h(i) + A_i \leq h(j)$
  (2) for each i in T $h(i) + A_i \leq k$
  (3) for each t, $0 \leq t < k$, there are at most m values of i for which $h(i) \leq t < h(i) + A_i$?

The following problems are special cases of problem 1:

- (Problem 2): Single-execution-time scheduling. We restrict problem 1 by requiring $A_i = 1$, $1 \leq i \leq n$. (All tasks require one time unit.)
- (Problem 3): Two-processor, one- or two-time-units scheduling. We restrict problem 1 by requiring $m = 2$, and A_i in $\{1, 2\}$, $1 \leq i \leq n$. (All tasks require one or two time units, and there are only two processors.)
- (Problem 4): Two-processor, interval-order scheduling. We restrict problem 1 by requiring the partial order $<$ to be an interval order and $m = 2$.
- (Problem 5): Single execution time, opposing forests. We restrict problem 1 by requiring $A_i = 1$, $1 \leq i \leq n$, and the partial order $<$ to be an opposing forest.

(See the "Definitions" sidebar for an explanation of "interval orders" and "forests.")

In 1972 Karp proved problem 1 to be NP-complete. In 1975 Ullman proved problems 2 and 3 to be NP-complete. Problem 4 was proven to be NP-complete by Papadimitriou and Yannakakis in 1979. In 1983 Garey, Johnson, Tarjan, and Yannakakis proved that problem 5 is also NP-complete. References to the proofs are listed in the literature.[2]

Considering communication cost

The complexity of the scheduling problem changes on the basis of which cost model is used to compute communication. The following is a summary of the NP-complete results

Definitions

Forests—A task graph is an *in-forest* if each task has at most one immediate successor. A task graph is an *out-forest* if each task has at most one immediate predecessor. A union of in-forests (at least one) and out-forests (at least one) is called an opposing forest.

Interval orders—A partially ordered set $(V, <)$ is an interval order if its elements can be mapped into intervals on the real line $(R, <)$ such that for all $x, y \in V$, $x < y$ if the interval assigned to x completely precedes the interval assigned to y. In the context of scheduling, the term *interval-ordered tasks* is used to indicate that the task graph, which describes the precedence relations among the system tasks, is an interval order.

using these models. Using Model A, Afrati et al. showed that scheduling a tree with communication on an arbitrary number of processors is an NP-complete problem.[2]

Using Model B, Prastein proved that by taking communication into consideration, even when the execution time for all tasks is identical and equal to the communication cost between any pair of processors, the problem of scheduling an arbitrary precedence program graph on two processors is NP-complete and scheduling a tree-structured program on arbitrarily many processors is also NP-complete. Prastein also indicated that scheduling a tree-structured task graph on two processors, using Model B, is an open problem in general.[2]

Using Model C, Papadimitriou and Yannakakis proved that the problem of optimally scheduling unit-time task graphs with communication on an unlimited number of processors is NP-complete when the communication between any pair of processors is the same and greater than or equal to one. In addition, they introduced an algorithm to approximate the optimal schedule length within a factor of two.[2]

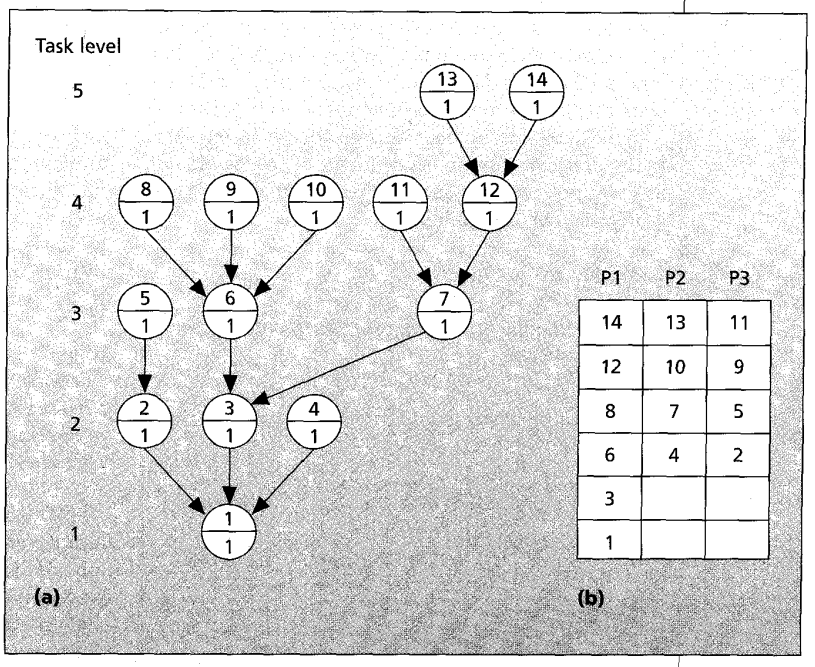


Figure 4. Scheduling an in-forest on three identical processors: (a) task graph, (b) Gantt chart.

OPTIMAL SCHEDULING ALGORITHMS

Scheduling task graphs is known to be polynomial in only a few cases. In this section we discuss the solutions when communication cost is ignored and those when communication cost is considered.

When communication cost is ignored

There are only three cases for which polynomial-time algorithms can be obtained: tree-structured task graphs on an arbitrary number of processors, interval orders on an arbitrary number of processors, and arbitrary task graphs on two processors.

SCHEDULING TREE-STRUCTURED TASK GRAPHS. In a classic paper,[3] Hu presented an algorithm called the level algorithm, which can be used to solve the scheduling problem in linear time when the task graph is either an *in-forest* (each task has at most one immediate successor) or an *out-forest* (each task has at most one immediate predecessor) and all tasks have the same execution time. In this section, we assume that the task graph is an in-forest of n tasks. There is no loss of generality if we assume that all the tasks have unit execution times. The algorithm given in this section is linear in the number of tasks. We first define the following:

Task level

Let the level of a node x in a task graph be the maximum number of nodes (including x) on any path from x to a terminal task (leaf). In a tree, there is exactly one such path. A terminal task is at level 1.

Ready tasks

Let a task be *ready* when it has no predecessors or when all its predecessors have already been executed.

Algorithm 1

(1) The level of each node in the task graph is calculated as given above and used as each node's priority.
(2) Whenever a processor becomes available, assign it the unexecuted ready task with the highest priority.

Algorithm 1 can be used in the out-forest case with simple modification.

Example 2

Consider the problem of scheduling the task graph given in Figure 4a on a fully connected target machine of three identical processors. Applying Algorithm 1, we first compute the level at each node. At the beginning, among all the ready tasks (4, 5, 8, 9, 10, 11, 13, and 14), tasks 13 and 14 at level 5 are assigned first, as shown in the resulting schedule given in Figure 4b. Following the path from node 13 (or node 14) to the terminal node 1, we can see that regardless of the number of available processors, at least five units of time will be required to execute all the tasks in the system. With only three processors, the optimal schedule length is six. Note that the schedule shown in Figure 4b is not the only optimal schedule that the algorithm can generate.

SCHEDULING INTERVAL-ORDERED TASKS. The algorithm we present here is for scheduling interval-ordered tasks on an arbitrary number of processors. An interval order is a task graph in which the nodes can be mapped into intervals on the real line and two nodes are related if the corresponding intervals do not overlap.[2,4,5] The properties of

interval orders make it possible to apply a simple greedy algorithm to find an optimal schedule when the execution time of all tasks is the same. At any given time, if there is more than one task ready for execution, picking the task with the maximum number of successors will always lead to the optimal solution, since its set of successors will include the set of successors of other ready tasks. This idea is presented in Algorithm 2.[5]

Algorithm 2

(1) The number of successors of each node is used as its priority.

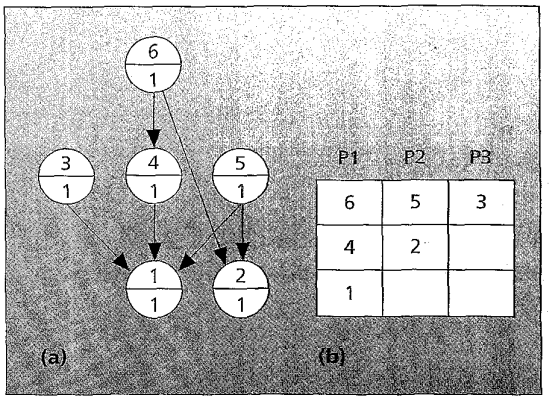


Figure 5. Scheduling an interval order on three identical processors: (a) task graph, (b) Gantt chart.

(2) Whenever a processor becomes available, assign it the unexecuted ready task with the highest priority.

This algorithm solves the unit-execution-time scheduling problem for interval order (V, E) in $O(|E|+|V|)$ time, since the number of successors of all tasks can be computed in $O(|E|)$ time and sorting the tasks according to the number of successors can be done in $O(|V|)$ time using the bucket sort technique. Step 2 can be implemented in $O(|V|)$ time.

Example 3

Consider the problem of scheduling the interval order given in Figure 5a on a fully connected target machine of three identical processors. Figure 5b shows the resulting schedule after Algorithm 2 is applied.

ARBITRARY TASK GRAPHS ON TWO PROCESSORS. When all tasks have the same execution time, there are no known polynomial algorithms for scheduling task graphs on a fixed number of processors m if $m > 2$. The first polynomial-time algorithm for $m = 2$, based on matching techniques, was presented by Fujii et al.[6] The time complexity of their algorithm is $O(n^{2.5})$. Improved algorithms have been obtained by Coffman and Graham, Sethi, and Gabow.[7,8] The time complexity of these three algorithms is $O(n^2)$, $O(\min(en, n^{2.61}))$, and $O(e + n\alpha(n))$, respectively, where n is the number of nodes and e is the number of arcs in the task graph.

Below, we present the algorithm given by Coffman and Graham. The approach is similar to that used for scheduling trees: Labels giving priority are assigned to tasks, and a list for scheduling the task graph is constructed from the labels. Labels from the set $\{1, 2, \ldots, n\}$ are assigned to each task in the task graph by the function $L(*)$, as we explain in Algorithm 3.

Algorithm 3

(1) Assign 1 to one of the terminal tasks.
(2) Let labels $1, 2, \ldots, j-1$ be already assigned. Let S be the set of unassigned tasks with no unlabeled successors. We next select an element of S to be assigned label j. For each node x in S define $l(x)$ as follows: Let $y_1, y_2, \ldots, y_k$ be the immediate successors of x. Then $l(x)$ is the decreasing sequence of integers formed by ordering the set $\{L(y_1), L(y_2), \ldots, L(y_k)\}$. Let x be an element of S such that for all x' in S, $l(x) \leq l(x')$ (lexicographically). Define $L(x) = j$.
(3) When all tasks have been labeled, use the list $(T_n, T_{n-1}, \ldots, T_1)$, where for all i, $1 \leq i \leq n$, $L(T_i) = i$, to schedule the tasks.

Since each task executes for one unit of time, processors 1 and 2 both become available at the same time. We assume that processor 1 is scheduled before processor 2.

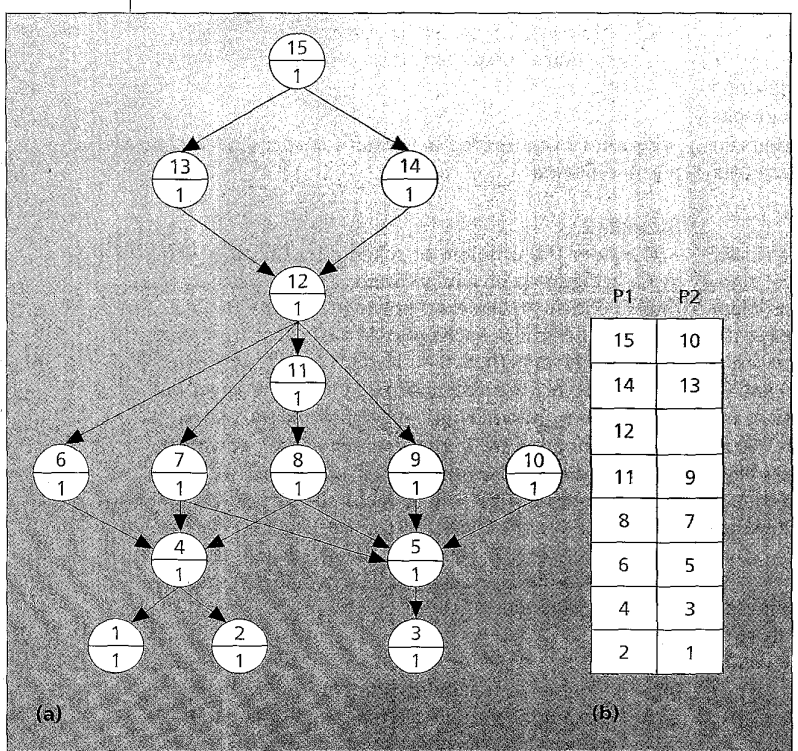


Figure 6. A task graph and its optimal schedule on two processors: (a) task graph, (b) Gantt chart.

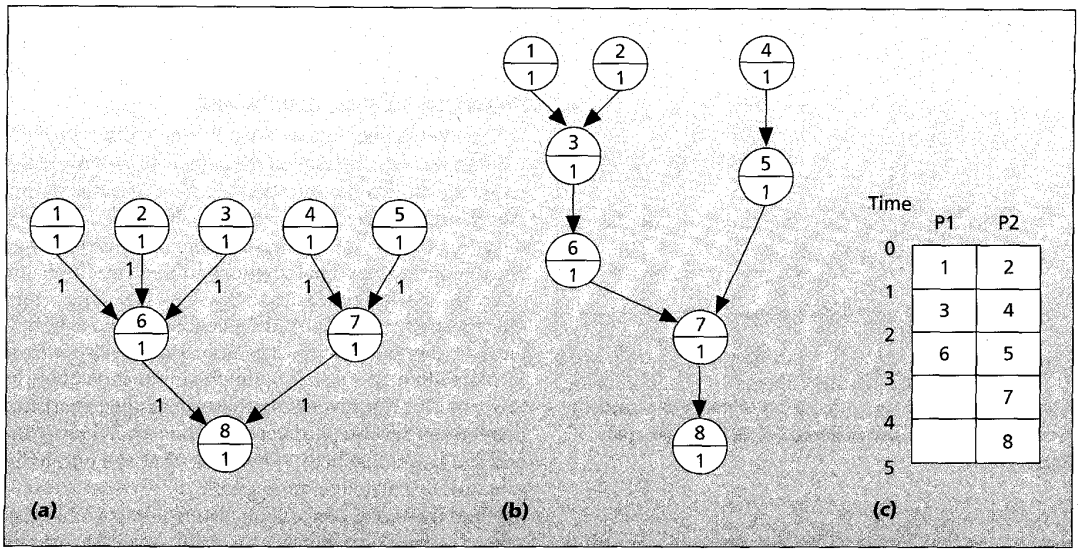


Figure 7. (a) Tree-structured task graph, (b) augmented task graph, and (c) an optimal schedule of the tree in (a) with communication.

Example 4

To understand the algorithm, let's examine the task graph given in Figure 6a. The three terminal tasks are assigned the labels 1, 2, and 3. At this point, the set S of unassigned tasks with no unlabeled successors becomes $\{4, 5\}$. Also, note that $l(4) = \{2, 1\}$, and $l(5) = \{3\}$. Since $\{3\} > \{2, 1\}$ (lexicographically), we assign labels 4, 5 to the tasks as given in the figure. The algorithm continues until all tasks are labeled. The number within each node in Figure 6 indicates its label. Task 15, with the highest label, is scheduled first on processor 1, then task 10 is scheduled on processor 2. After tasks 15 and 10 complete, the only ready tasks are 13 and 14. Recall that a task is called ready when all its predecessors have been executed. Try the labeling algorithm on the rest of the nodes in the given task graph. Figure 6b shows the output schedule that results after applying Algorithm 3.

Considering communication cost

Here we present two cases for which scheduling task graphs with communication is known to be polynomial: scheduling trees on two processors and scheduling interval orders on an arbitrary number of processors.

SCHEDULING TREES ON TWO PROCESSORS. The main idea of the algorithm presented below is to augment the task graph with new precedence relations to compensate for communication. Scheduling the augmented task graph without considering communication is equivalent to scheduling the original task graph *with* communication.[2] Here we assume that the tree is an in-forest. However, the algorithm can be easily modified to handle the out-forest case. The time complexity of the algorithm is $O(n^2)$. We first define the following:

Node depth

A node with no predecessors has a depth of zero. The depth of any other node is defined as the length of the longest path between the node and any node with depth zero.

Operation swapall

Given a schedule f, we define the operation *swapall*(f, x, y), where x and y are two tasks in f scheduled to start at time t on processors i and j, respectively. The effect of this operation is to swap all the task pairs scheduled on processors i and j in the schedule f at time t_1, $\forall\, t_1, t_1 \geq t$.

Algorithm 4

(1) Given an in-forest $G = (V, E)$, identify the sets of siblings $S_1, S_2, \ldots, S_k$, where S_i is the set of all nodes in V with a common child, $child(S_i)$.
(2) $E1 \leftarrow E$
(3) For every set S_i

- Pick node $u \in S_i$ with the maximum depth
- $E1 \leftarrow E1 - (v, child(S_i))\ \forall\, v \in S_i$ and $v \neq u$
- $E1 \leftarrow E1 \cup (v, u)\ \forall\, v \in S_i$ and $v \neq u$

(4) Obtain the schedule f by applying Algorithm 1 on the augmented in-forest $F = (V, E1)$.
(5) For every set S_i in the original in-forest G, if node u (with the maximum depth) is scheduled in f in the time slot immediately before $child(S_i)$, but on a different processor, then apply the operation *swapall*($child(S_i), x, f$), where x is the task scheduled in the time slot immediately after u on the same processor.

Example 5

Consider the in-forest shown in Figure 7a. The in-forest, after the new arcs are added, is shown in Figure 7b. Algorithm 1 can be applied to this augmented in-forest to obtain a schedule wherein communication delays are considered. The operation swapall is applied when commu-

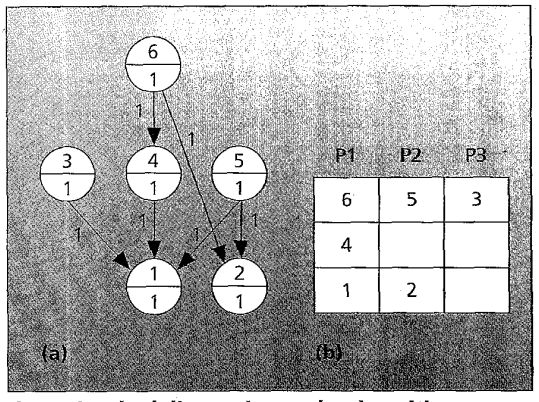


Figure 8. Scheduling an interval order with communication on three processors: (a) task graph, (b) Gantt chart.

nication restrictions are violated in the output schedule. The final schedule is shown in Figure 7c.

SCHEDULING INTERVAL ORDERS WITH COMMUNICATION. We present an algorithm that we (Ali and El-Rewini) introduced in 1993 that finds a solution when execution time is the same for all tasks and is identical to communication delay.[4] We first define the following:

- $n_2(v)$: the number of successors of task v.
- *start-time*(v, i, f): the earliest time at which task v can start execution on processor p_i in schedule f.
- *task*(i, t, f): the task scheduled on processor p_i at time t in schedule f. If there is no task scheduled on processor p_i at time t in schedule f, then *task*(i, t, f) returns the empty task ϕ. Note that $n_2(\phi) < n_2(v)$ for any task $v \in V$.

Algorithm 5

(1) Use $n_2(v)$ as the priority of task v, and ties are broken arbitrarily.
(2) Nodes with highest priority are scheduled first.
(3) Each task v is assigned to processor p_i with the minimum start time.
(4) If *start-time*(v, i, f) = *start-time*(v, j, f), $1 \le i, j \le m$, task v is assigned to processor p_i if *task*$(i,$ *start-time*$(v, i, f) - 1, f)$ has the minimum n_2.

The time complexity of the algorithm is $O(ne)$, where n is the number of tasks and e is the number of arcs in the interval order.

Example 6

In this example, we schedule the interval order of Figure 8a. We assume that the communication cost on all arcs equals one. As shown in Figure 8b, scheduling tasks 6, 5, 3, and 4 is straightforward. At this point, tasks 1 and 2 have the same out degree ($n_2(1) = n_2(2) = 0$), but we will assume that task 2 is considered for scheduling before task 1. Task 2 can start execution on processors P1, P2, and P3 at the same time (third time slot). Algorithm 5 will not schedule it on processor P1, so task 1 can start execution in the third time slot. Otherwise, the schedule length would have been four instead of three.

HEURISTIC ALGORITHMS

To provide solutions to real-world scheduling problems, we must relax restrictions on the parallel program and the target machine representations. Recent research in this area has emphasized heuristic approaches. A heuristic produces an answer in less than exponential time but does not guarantee an optimal solution. Intuition usually helps us come up with heuristics that use special parameters affecting the system indirectly. A heuristic is said to be better than another heuristic if solutions approach optimality more often, or if a near-optimal solution is obtained in less time. The effectiveness of these scheduling heuristics depends on several parameters of the parallel program and the target machine. A heuristic that can optimally schedule a particular task graph on a certain target machine may not produce optimal schedules for other task graphs on other machines. As a result, several heuristics have been proposed, each of which may work under different circumstances.

One class of scheduling heuristics that includes many schedulers is list scheduling. In list scheduling, each task is assigned a priority, then a list of tasks is constructed in a decreasing priority order. A ready task with the highest priority is scheduled on the task's "best" available processor. The schedulers in this class differ in the way they assign priorities to tasks and in the criteria used to select the "best" processor to run the task. Priority assignment results in different schedules because nodes are selected in a different order. Algorithm 6 shows the general list-scheduling algorithm.

Algorithm 6

(1) Each node in the task graph is assigned a priority. A priority queue is initialized for ready tasks by inserting every task that has no immediate predecessors. Tasks are sorted in decreasing order of task priority.
(2) As long as the priority queue is not empty, do the following:
    - Obtain a task from the front of the queue.
    - Select an idle processor to run the task.
    - When all the immediate predecessors of a particular task are executed, that successor becomes ready and can be inserted into the priority queue.

A different type of scheduling heuristics tries to partition the scheduling process into two phases: processor assignment (allocating tasks to the system processors) and task ordering (scheduling the tasks allocated on each processor). Task graphs can be clustered as an intermediate phase to solve the allocation problem of the scheduling process. Algorithm 7 shows the general idea of this method.

Algorithm 7

(1) Cluster the tasks assuming an unlimited number of fully connected processors. Two tasks in the same

cluster are scheduled in the same processor.

(2) Map the clusters and their tasks onto the given number of processors (m). In this step, the following optimizations are performed:

- *Cluster merging.* If the number of clusters is greater than the number of available processors, the clusters are merged into m clusters.
- *Physical mapping.* The actual architecture is not fully connected. A mapping must be determined such that overall communication between clusters is minimized.
- *Task execution ordering.* After the processor assignment of tasks is fixed, execution ordering is determined to ensure the correct dependence order between tasks.

More details on scheduling heuristic algorithms can be found in the literature.[2,9-11]

SCHEDULING TOOLS

Software development is intrinsically difficult and time consuming for both sequential and parallel computing applications. However, designing and writing software for parallel computers is even more difficult because of the increased complexity incurred in dealing with task scheduling, synchronization, and performance. The use of software tools is one way to make software development for parallel computers easier.

The software development process for parallel computers starts with identifying parallelism in an application, continues through task partitioning, scheduling, and performance tuning, and ends with code generation. Experience has shown that it is very difficult to automate this entire process. However, it is possible to automate some phases of the development life cycle to increase programming productivity and take advantage of the computer's ability to perform tedious chores better than humans. Task scheduling appears to be one such chore that would benefit by being automated through software tools. In developing a parallel application, a programmer needs help in answering a number of scheduling-related questions. What is the best grain size for the program tasks? What is the best scheduling heuristic? How can performance be improved? How many processors should be used? Where should synchronization primitives be inserted in the code? Answers to these questions and many others can be determined through cooperation between a tool and a human program developer. Software tools have been used at two different phases of the software development life cycle: design and code generation. Three scheduling tools in particular incorporate many of the heuristics in the literature: Parallax, Hypertool, and Pyrros. We describe them only briefly; more details can be found in the literature.[10-12]

Parallax

Parallax is a software tool that aids in parallel program design by automating a number of scheduling heuristics and performance analysis tools.[12] As Figure 9 shows, Parallax produces (1) schedules in the form of Gantt charts for several scheduling algorithms, (2) performance charts in the form of line and bar graphs, and (3) critical-path analysis. With Parallax, a user can

- model a parallel program as a task graph,
- choose a method of optimization from several scheduling heuristics that will automatically produce a schedule,
- choose the topology of the desired target machine (or design an arbitrary topology for the parallel processor of interest), and
- observe anticipated scheduling and performance estimates obtained from scheduling the task graph onto the target machine.

Parallax, which supports most of the scheduling heuristics introduced in the literature, is a tool for investigating scheduling heuristics before actually executing a parallel program. That is, Parallax is a design tool as opposed to a programming tool. It can also be used to refine an existing parallel program after performance data has been collected from one or more runs. The basic idea behind Parallax is to help a human user create a parallel program design as a task graph, enter the target machine as a graph, and then perform a number of "what if" analyses.

Pyrros

Pyrros is a compile-time scheduling and code generation tool.[11] Its input is a task graph and the associated sequential C code. The output is a static schedule and a parallel C code for a given architecture. Pyrros has the following components: a task graph language with an interface to C, a scheduling system, a graphic displayer, and a code generator. The task graph language lets users define partitioned programs and data. The scheduling system is used for clustering the graph, load balancing and physical mapping, and computation/communication ordering. The graphic displayer is used for displaying task graphs and scheduling results. The code generator inserts synchronization primitives and performs code optimization for various parallel machines. A user first edits the program with the task graph language to specify the dependence information between partitioned program segments, the associated C code, weights, and the maximum number of processors available. Pyrros can display

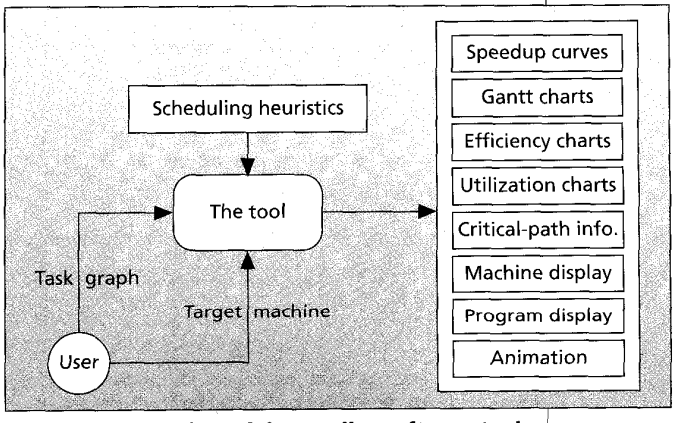


Figure 9. An overview of the Parallax software tool.

the input dependence graph to help the user verify the correctness of the flow dependence between tasks, and it can also generate the schedule and the code. The user can check the scheduling result by letting Pyrros display the schedule Gantt chart in the graph window and the statistics information in a separate text window. Figure 10 shows an overview of Pyrros.

Hypertool

Hypertool takes a user-partitioned program as its input, automatically allocates the partitions to processors, and inserts proper synchronization primitives where needed.[10] First, a designer develops a proper algorithm, performs partitioning, and writes a program as a set of procedures.

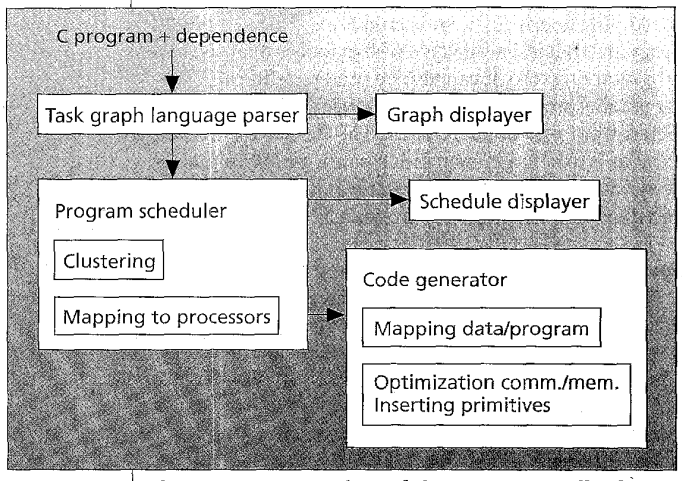


Figure 10. An overview of the Pyrros compile-time scheduling and code generation tool.

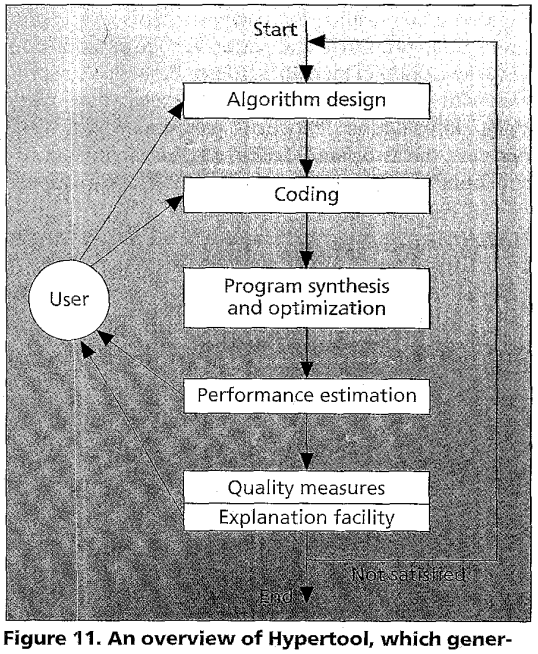


Figure 11. An overview of Hypertool, which generates performance estimates.

The program looks like a sequential program, and it can be debugged on a sequential machine. Through parallel code synthesis and optimization, this program is automatically converted into a parallel program for a distributed-memory target machine. Hypertool can then generate performance estimates, including execution time, communication time, suspension time for each processor, and network delay for each communication channel. If the result is not satisfactory, the programmer can use the information provided by Hypertool's performance estimator and explanation facility to try to redefine the partitioning strategy and the size of the partitions. Figure 11 illustrates the Hypertool components.

THE GOAL OF SCHEDULING IS TO DETERMINE an assignment of tasks to processors and an order in which tasks are executed to optimize some performance measure. Scheduling is a computationally intensive problem and known to be NP-complete. Because of the problem's intractability, recent research has emphasized heuristic approaches. Optimal algorithms can be obtained only in some restricted cases. Even though several instances of the problem have been proven to be NP-complete, several open problems remain. For example, scheduling task graphs when communication is not considered and when all tasks take the same amount of time on fixed $m \geq 3$ processors is still an open problem.

More research is needed to obtain optimal algorithms when certain restrictions are relaxed in the cases that have already been solved. For example, communication delay is a major parameter in parallel and distributed systems and should be considered. Since optimal schedules can be obtained in restricted cases that may not represent real-world situations, a simplified suboptimal approach to the general form of the problem is needed. Recent research in this area has emphasized heuristic construction, evaluation, and application. The challenge is to incorporate real-world parameters into our problem solutions. Another research direction is the development of scheduling software tools to help design parallel programs, automatically generate parallel code, and estimate performance. ■

References

1. J. Ullman, "NP-Complete Scheduling Problems," *J. Computer and System Sciences*, Vol. 10, 1975, pp. 384-393.
2. H. El-Rewini, T. Lewis, and H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, Englewood Cliffs, N.J., 1994.
3. T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, No. 6, 1961, pp. 841-848.
4. H. Ali and H. El-Rewini, "An Optimal Algorithm for Scheduling Interval Ordered Tasks with Communication on *N* Processors," *J. Computer and System Sciences*, to appear in 1995.
5. C.H. Papadimitriou and M. Yannakakis, "Scheduling Interval-Ordered Tasks," *SIAM J. Computing*, Vol. 8, 1979, pp. 405-409.
6. M. Fujii, T. Kasami, and K. Ninomiya, "Optimal Sequencing of Two Equivalent Processors," *SIAM J. Appl. Math.*, July 1969.
7. E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, Somerset, N.J., 1976.
8. H. Gabow, "An Almost Linear Algorithm for Two-Processor Scheduling," *J. ACM*, Vol. 29, No. 3, July 1982, pp. 766-780.
9. A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *J. Par-*

allel and Distributed Computing, Dec. 1992, pp. 276-291.

10. M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 101-119.
11. T. Yang and A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors," *Proc. Sixth ACM Int'l Conf. Supercomputing*, ACM, New York, 1992, pp. 428-443.
12. T. Lewis and H. El-Rewini, "Parallax: A Tool for Parallel Program Scheduling," *IEEE Parallel and Distributed Technology*, Vol. 1, No. 2, May 1993, pp. 62-72.

Hesham El-Rewini *is an associate professor of computer science at the University of Nebraska at Omaha. His research interests include parallel programming environments and task scheduling, and he has coauthored two books and published numerous articles on these topics. He was a guest editor for* IEEE Parallel and Distributed Technology*'s August 1993 issue and is guest editor for the magazine's Fall 1996 issue on the engineering of complex distributed systems, for which he is currently accepting papers. He is on several international conference program committees and continues to chair the software track of the Hawaii International Conference on System Sciences. El-Rewini received BS and MS degrees from the University of Alexandria, Egypt, in 1982 and 1985, respectively, and a PhD from Oregon State University in 1990, all in computer science. He is a member of the ACM and the IEEE Computer Society.*

Hesham H. Ali *is an associate professor of computer science at the University of Nebraska at Omaha. He is the coauthor of two recent books,* Task Scheduling in Parallel and Distributed Systems *and* Introduction to Graph Algorithms, *and the author of numerous articles in his research fields, which include the development of graph analysis and generation tools and the application of graph theory in parallel computing and VLSI design. Ali received BS and MS degrees from the University of Alexandria, Egypt, in 1982 and 1985, respectively, and a PhD from the University of Nebraska–Lincoln in 1988, all in computer science. He is a member of the ACM.*

Ted Lewis *is professor and chair of computer science at the Naval Postgraduate School in Monterey, California. A past editor-in-chief of both* Computer *and* IEEE Software, *he has published extensively in the areas of parallel computing and real-time software engineering. He received a BS in mathematics from Oregon State University in 1966 and MS and PhD degrees from Washington State University in 1970 and 1971, respectively. He is a member of the IEEE Computer Society.*

Readers can contact El-Rewini and Ali at the Department of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182-0243, e-mail {rewini, hesham}@cs.unomaha.edu and Lewis at the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943-5100, e-mail lewis@cs.nps.navy.mil.

Judith Schlesinger, a former parallel computing area editor for Computer, *coordinated the review of this article and recommended it for publication. Her e-mail address is judith@super.org.*

Book and Software Authors

- Programming and Development Tools for Communications Professionals, Engineers, and Scientists
- Software Engineering
- VLSI Design and Testing
- Graphics Applications
- Object-Oriented Programming
- Communications and Networking
- Computer Systems and Architectures

Theron Shreve, *Acquisitions Editor*
685 Canton Street, Norwood, MA 02062
(800) 225-9977 aqartech@world.std.com

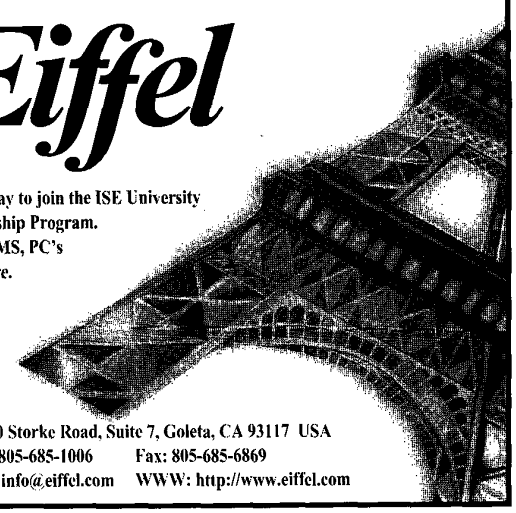
Artech House Publishers BOSTON • LONDON

Reader Service Number 2

Give your students the best...

Teach programming the object-oriented way with

Eiffel



Call today to join the ISE University Partnership Program.
Unix, VMS, PC's *and more.*

ISE, 270 Storke Road, Suite 7, Goleta, CA 93117 USA
Phone: 805-685-1006 Fax: 805-685-6869
E-mail: info@eiffel.com WWW: http://www.eiffel.com

Reader Service Number 3