Scheduling Conditional Branching using Representative Task Graphs

Hesham El-Rewini and Hesham H. Ali

Department of Math and Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0243

Abstract. We study the problem of scheduling parallel programs with conditional branching on parallel processors. The major problem in having conditional branching is the non-determinism since the direction of a branch may be unknown until the program is midway in execution. In this paper, we overcome the problem of non-determinism by proposing a probabilistic model that distinguishes between branch and precedence relations in parallel programs. We approach the problem of scheduling task graphs that contain branches by representing all possible execution instances of the program by a single deterministic task graph, called the representative task graph. The representative task graph can be scheduled using one of the scheduling techniques used for deterministic cases. We also show that a schedule for the representative task graph can be used to obtain schedules for all execution instances of the program.

1. Introduction

A recent survey by Casavant [3] and related studies [6,15] have shown that the problem of optimally scheduling parallel program tasks on parallel computers is NP-hard in its most general form as well as many special cases. The complexity of the problem rises even further when some important factors such as communication and conditional branching are considered. Several researchers have studied restricted forms of the problem by constraining the task graph representing the parallel program or restricting the parallel system model [8,9,10,11,12].

An optimal schedule is an allocation and ordering of parallel program tasks on multiprocessor system such that the tasks complete in the shortest time. We distinguish between allocation and scheduling. An allocation is an assignment of the program tasks to the processors [1,2,5,13]. An allocation may result in a non-optimal schedule, because the order of execution of each task plays an important role in determining the completion time of the program. Scheduling determines both the allocation and the execution order of each task in the parallel program.

Few researchers have investigated the problem of scheduling task graphs that contain conditional branches. This is mainly due to the difficulty in modeling the problem in its general form as well as the associated non-determinism since the direction of a conditional branch is unknown before execution. In [14], Towsley introduced a computational model of a distributed program containing probabilistic branches and an allocation algorithm based on the shortest path method. Chou and Abraham [4] considered a computational model that includes conditional branching and concurrent execution of modules. The models given in [4,14] are not general enough to accommodate the most general form of the problem and lack the distinction between branch relations and dependency relations among tasks. Both methods can be classified as allocation techniques rather than scheduling techniques.

In this paper, we propose a new model to represent parallel programs containing conditional branching in the most general form. The model consists of two graphs; the *Branch Graph* and the *Precedence Graph*. We also introduce a new approach to deal with the non-determinism in scheduling conditional branching. Our approach is based on representing all execution instances of the parallel program by one representative task graph. A deterministic scheduling technique can then be applied to schedule the representative task graph. The obtained schedule is used to generate schedules for all possible execution instances of the program.

The paper is organized as follows. In Section 2, the new parallel model is described. Section 3 introduces the new scheduling approach. We give an illustrative example in section 4 and the concluding remarks are given in Section 5.

2. The Parallel Program Model

In this model, the behavior of a parallel program is represented by two acyclic directed graphs called the *Branch Graph*, $G = (T, E_b)$ and the *Precedence Graph*, $H = (T, E_p)$, where T is a set of M vertices representing the program tasks, E_b is a set of branch edges, and E_p is a set of precedence edges. An execution instance of a program is defined as the set of tasks that are selected for execution at one time for some input.

A branch edge $(T_i, T_j) \in E_b$ implies that there exists some execution instance of the program that includes the tasks T_i and T_j. It also implies that there is a conditional test associated with T_i that decides whether task T_j is going to be part of the same execution instance. Associated with each branch edge (T_i, T_j) is $P(T_i, T_j)$, the probability of having T_j in the same execution instance with T_i. Let $IMS = \{v_1, v_2, \dots, v_m\}$ be the set of all immediate successors of vertex u in the Branch Graph, then $\sum_{i=1}^{m} P(u, v_i) = 1$.

A Branch Graph consists of a collection of connected components. Each component is a *Fan Graph*, which can be defined recursively as follows. A directed acyclic graph with $n + 2, n > 1$ nodes is called a Fan Graph iff it has n independent nodes with one common parent (s) and one common child (t) as shown in Figure 1. If G is a Fan Graph, then a graph obtained from G by replacing any node by a Fan Graph is also a Fan Graph. It can be noticed that Fan Graphs form a special class of Series-Parallel graphs. Fan Graphs are typical in expressing selections (branching) in structured programming. Figure 2-a shows an example of a Branch Graph, consisting of 21 nodes, that represent branching in a parallel program. The number shown inside each node is the node title and the number next to an edge (i, j) represent the parameter $P(i, j)$. For example $P(1,3) = 0.2$.

A precedence edge (T_i, T_j) implies that Task T_j cannot start execution until after the completion of task T_i. The precedence edge (T_i, T_j) might also represent data flow between tasks T_i and T_j. Associated with the edge (T_i, T_j) is the data size $D(T_i, T_j)$, which might have the value zero if there is no dataflow between the two tasks. Associated with each task T_i is the number of instructions to be executed $INS(T_i)$. Figure 2-b shows an example of a Precedence Graph, consisting of 21 nodes, that represent precedence relations as well as dataflow in the parallel program. The number shown in the upper portion of each node is the task title, the number in the lower portion of node i represents the parameter $INS(i)$, and the number next to an edge (i, j) represents the parameter $D(i, j)$. For example $INS(4) = 11$ and $D(4, 5) = 10$ while $D(18, 19) = 0$ indicating that task 18 must precede task 19 and there is no dataflow between them.

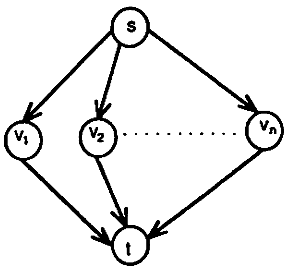


Figure 1 Fan Graph

3. Scheduling Task Graphs with Branches

In this section we introduce a new approach to handle conditional branching in the scheduling problem. In this approach, the probabilistic task graph is approximated by exactly one deterministic task graph, called the *Representative Task Graph*, that represents all possible execution instances. Each node in the representative task graph represents one or more tasks of the original set of tasks. Similarly, each edge represents one or more precedence relations in the original Precedence Graph. The representative task graph is selected such that any one of its feasible schedules provide feasible schedules for all possible execution instances of the program. The conventional techniques used for scheduling deterministic task graphs without branching can be used to schedule the obtained task graph [7,8,9,12].

In order to achieve that, we add one restriction to the Precedence Graph. We assume that if two nodes u and v have a common parent and a common child in the Branch Graph, then there exists no node x such that (x, u) and (v, x) are edges in the Precedence Graph. This is typical in several applications in which the alternative modules of a selection are related in a similar way with other modules

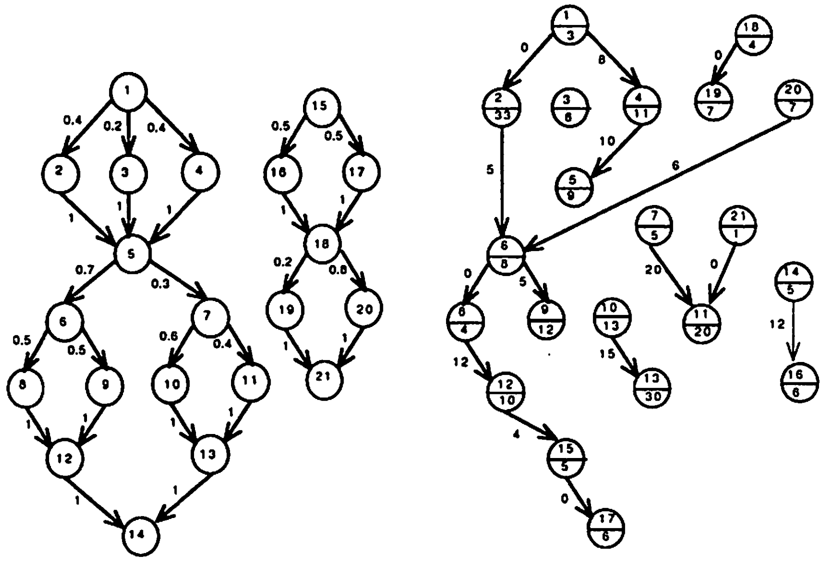

(a) The Branch Graph (b) The Precedence Graph

Figure 2 A Parallel Program Model

in the program. We also assume that the branching in the program is free of nested conditional selections. This restriction implies that the branch graph consists of a collection of components. Each component is a Restricted Fan Graph, which is a Fan Graph with the restriction that if a node has only one parent, it cannot have more than one child. At the end of this section this restriction will be relaxed and a method will be suggested to accommodate nested branches.

The basic idea in this approach is to find some tasks with common properties in order to be able to represent several tasks by one task. These properties express the way each task is related to other tasks in the program. A task can be related to other tasks in several ways including precedence relation, data dependency and occurrence in some execution instances. The following definitions will be used to describe these properties.

Similar nodes. Two nodes u and v are said to be similar if they have the same sets of successors and predecessors in the Branch Graph. In other words, similar nodes are those that are related in the same way to all other nodes in the Branch Graph. Notice that the concept of similarity is transitive.

Twin nodes. Two nodes u and v are said to be twins if they satisfy the following conditions:

1 u and v are similar
2 u and v have the same sets of successors and predecessors in the Precedence

Graph

3 $D(x,u) = D(x,v), \forall x$ such that $(x,v) \in E_p$

4 $D(u,y) = D(v,y), \forall y$ such that $(v,y) \in E_p$

In other words, twin nodes are those that are related in the same way to all other nodes in both the Branch Graph and the Precedence Graph.

The scheduling technique used in this section is based on the following Lemmas. The proofs of Lemmas 1 and 2 follow immediately from the above definitions.

Lemma 1. *Given a Branch Graph, $G = (T, E_b)$ and $u, v \in T$, if u and v are similar then there exists no execution instance that contains both of them.*

Lemma 2. *Given a Branch Graph, $G = (T, E_b)$ and $u, v \in T$, if u and v are twins and they require the same amount of computation, then an optimal schedule for an execution instance, EI_u containing u can be obtained by replacing v by u in an optimal schedule for an execution instance, EI_v containing v, where $(EI_u \cup EI_v) - (EI_u \cap EI_v) = \{u, v\}$.*

Corollary 1. *Given a parallel program model represented by Branch Graph, $G = (T, E_b)$ and Precedence Graph, $H = (T, E_p)$, if every pair of similar nodes are twins with the same amount of computation, then the parallel program model can be reduced to exactly one deterministic task graph called the Representative Task Graph, $F = (V, E)$, where $V \subseteq T$ and $E \subseteq E_p$. An optimal schedule for the tasks contained in any execution instance can be obtained from scheduling the tasks contained in the representative task graph.*

Proof: The Representative Task Graph, $F = (V, E)$, can be constructed as follows. The set of tasks V can be obtained from the Branch Graph by removing all nodes except one from every set of twins. The obtained set of nodes, V forms an execution instance that is isomorphic to all possible execution instances of the program. Consequently, using the same scheduling technique, the schedules of all possible execution instances are also isomorphic. The set of edges E can be obtained from the Precedence Graph, $H = (T, E_p)$ as follows. If two nodes $u, v \in V$ and the edge $(u, v) \in E_p$ then we add the edge (u, v) to the set E.

Given the optimal schedule of the tasks of the representative task graph, we can construct an optimal schedule for other tasks as follows: if task $v \in V$ is scheduled on processor P, then any twin of v will be assigned to the same processor P with the same execution order. Recall that two twins can not belong to the same execution instance (Lemma 1).

Lemma 3. *Given a parallel program model represented by Branch Graph, $G = (T, E_b)$ and Precedence Graph, $H = (T, E_p)$. If every pair of similar nodes are twins, then then the parallel program model can be reduced to exactly one deterministic task graph called the Representative Task Graph, $F = (V, E)$, where*

$V \subseteq T$ and $E \subseteq E_p$. An optimal schedule for the tasks contained in any execution instance can be obtained from scheduling the tasks contained in the representative task graph.

Proof: The Representative Task Graph, $F = (V, E)$, can be constructed as follows. The set of tasks V can be obtained from the Branch Graph by removing all nodes except the one with maximum computation time from every set of twins. The execution instance corresponding to the set V is isomorphic to all possible execution instances of the program in all aspects except the amount of computation needed at each task. Using the same scheduling technique, feasible schedules of all possible execution instances can be obtained from a schedule for the representative task graph. The task with maximum amount of computation is selected from every set of twins to guarantee feasible schedules for all other tasks. Selecting a different one might generate schedules that are not feasible for other tasks. Given a schedule of the tasks of V, we can construct a feasible schedules for other tasks as follows: if task $v \in V$ is scheduled on processor P, then any twin of v will be assigned to the same processor P with the same starting time of execution.

In the general case, the similar nodes might not be twins. Extra edges might be added and the parameter $D(*, *)$ associated with some edges might be modified in the Precedence Graph in order to guarantee that every pair of similar nodes are also twins. Adding the new edges should not create conflicts with the original set of edges. We define the *Augmented Precedence Graph* to be the original Precedence Graph along with the new added edges and modified parameters. The following theorem defines the Augmented Precedence Graph formally and describes how to handle the general case of scheduling parallel programs with branches.

Theorem 1. *Given a parallel program model represented by Branch Graph and Precedence Graph, the parallel program model can be reduced to exactly one deterministic task graph, called the Representative Task Graph after augmenting the Precedence Graph. The schedule obtained from scheduling the representative task graph can be used to schedule the tasks contained in other execution instances.*

Proof: The proof is divided into two steps; constructing the Augmented Precedence Graph and using the previous Lemmas to generate the Representative Task Graph.
Step 1: Given the Branch Graph, $G = (T, E_b)$ and the Precedence Graph, $H = (T, E_p)$ the Augmented Precedence Graph $AG = (T, E_{ap})$ is constructed as follows:

- $E_{ap} \leftarrow E_p$
- If two nodes u and v are similar but not twins in G, then for any node x in T, we do the following to force the twin relationship:
    * If (x, u) and $(x, v) \in E_{ap}$ and $D(x, u) \leq D(x, v)$ then $D(x, u) \leftarrow D(x, v)$

* If (u, x) and $(v, x) \in E_{ap}$ and $D(u, x) \leq D(v, x)$ then $D(u, x) \leftarrow D(v, x)$
* If $(x, u) \in E_{ap}$ and $(x, v) \notin E_{ap}$, then $E_{ap} \leftarrow E_{ap}U\{(x, v)\}$
* If $(u, x) \in E_{ap}$ and $(v, x) \notin E_{ap}$, then $E_{ap} \leftarrow E_{ap}U\{(v, x)\}$

Step 2: Having the Branch Graph and the Augmented Precedence Graph, the conditions in Lemma 3 are satisfied and the Representative Task Graph can be used to find a schedule for all possible execution instances.

In the above approach, we assumed that the parallel program model did not include nested branching. An example of nested conditional branching is shown in Figure 3. We suggest a method to accommodate nested branching in parallel programs. In this method, the twin relation is applied to the similar nodes of the inside branch (nodes x and y in the Figure). These nodes are replaced by just one node as discussed earlier and then this node along with the common parent and the common child are composed into a unified node. This unified node will be similar to the nodes of the outside branch (node z in the Figure). The composition process is repeated until the branch graph is free of nested branching and the approach mentioned earlier can be applied. Having done the composition in the Branch Graph, the Precedence Graph needs to be modified accordingly. If nodes $u_1, u_2, \ldots, u_n$ are combined in node u then $INS(u) \leftarrow INS(u_1) + INS(u_2) + \cdots + INS(u_n)$, and the union of predecessors (successors) of nodes $u_1, u_2, \ldots, u_n$ will form the predecessors (successors) of node u.

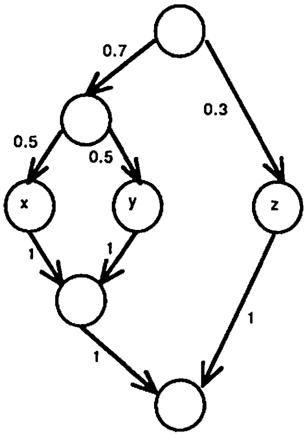
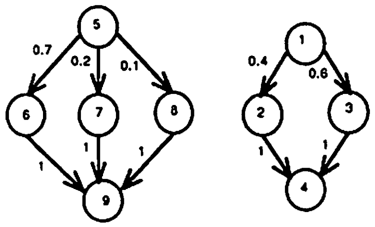


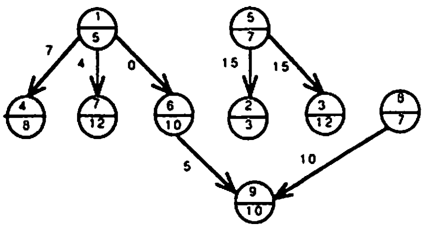
Figure 3 Nested Conditional Branching

4. Example

In this example, we construct a unified schedule for the parallel program represented by the Branch Graph and the Precedence Graph, shown in Figure 4, on a target machine consisting of two identical processing elements. We assume that the processing elements have unit speed and the communication links have unit transfer rate. Thus, we can deal with the parameters $INS(*)$ and $D(*,*)$ as units of time. Notice that tasks 2 and 3 are twins while tasks 7,8 and 9 are similar but not twins.

To construct the Augmented Graph, the two edges $(1,8)$ and $(7,9)$ are added with the parameters $D(1,8) = 4$ and $D(7,9) = 10$. Also some parameters are modified; $D(1,6) = 4$ and $D(6,9) = 10$. The Augmented Graph is given in Figure 5 where the added edges are given in bold and the modified parameters are underlined. Notice that tasks 7, 8 and 9 are twins in the Augmented Precedence Graph. Using Lemma 3, the Representative Task Graph consists of the set of tasks $\{1,3,4,5,7,9\}$. Figure 6 shows the obtained Representative Task Graph and the corresponding schedule generated by the deterministic scheduling method given in [7]. The nodes 2, 6 and 8 are scheduled by replacing nodes 3, 7 and 7 in the schedule, respectively.





(a) The Branch Graph (b) The Precedence Graph

Figure 4 A Program Model

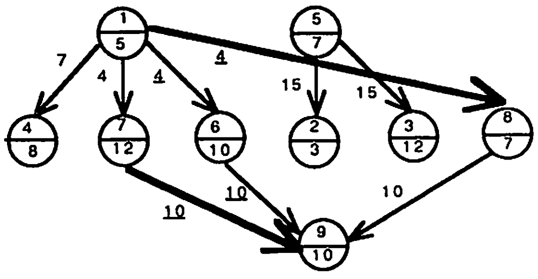


Figure 5 The Augmented Precedence Graph for Precedence Graph of Figure 4

5. Conclusions

In this paper, we introduced a new model for representing parallel programs that contain conditional branching. This model captures all the information needed to deal with the most general form of the scheduling problem. The model consists of two acyclic directed graphs called Branch Graph and Precedence Graph. We also introduced an approach for scheduling task graphs with branches on parallel

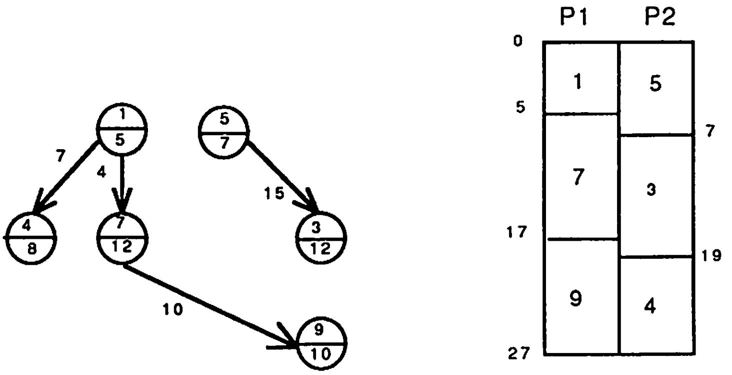


Figure 6 The Representative Task Graph and the corresponding Schedule

machines. In this approach, all execution instances are represented by a single task graph, called the representative task graph, and a schedule for this graph is used to obtain schedules for all execution instances.

References

1. S. Bokhari, *A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in Distributed Processor System*, IEEE Transaction on Software Engineering, vol. SE-7, no. 6 (1981).
2. S. Bokhari, *On the Mapping Problem*, IEEE Transactions on Computers, C-30 3 (1981), 207–214.
3. T. Casavant and J. Kuhl, *A Taxonomy of Scheduling in General Purpose Distributed Computing Systems*, IEEE Transaction on Software Engineering, vol. SE-14, no. 2 (1988).
4. T. Chou and J. Abraham, *Load Balancing in Distributed Systems*, IEEE Transaction on Software Engineering, vol. SE-8, no. 4 (1981).
5. W. Chu, L. Holloway, M. Lan and K. Efe, *Task Allocation in Distributed Data Processing*, IEEE Computer (1980), 57–69.
6. E. Coffman, "Computer and Job-Shop Scheduling Theory", John Wiley & Sons, 1976.

7. H. El-Rewini and T.G. Lewis, *Scheduling Parallel Program Tasks onto Arbitrary Target Machines*, Journal of Parallel and Distributed Computing (1990).
8. H. El-Rewini, *Task Partitioning and Scheduling on Arbitrary Parallel Processing systems*, Ph.D. thesis, Department of Computer Science, Oregon State University (1989).
9. T. Hu, *Parallel Sequencing and Assembly Line Problems*, Operation Research **9** (1961), 841–848.
10. B. Kruatrachue, *Static Task Scheduling and Grain Packing in Parallel Processing Systems*, Ph.D. thesis, Department of Computer Science, Oregon State University (1987).
11. C.Y. Lee, J.J. Hwang, Y.C. Chow, and F.D. Anger, *Multiprocessor Scheduling with Interprocessor Communication Delays*, Operations Research Letters, 7, 3 (1988), 141–147.
12. V. Linnemann, "Deterministic Processor Scheduling with Communication Cost", Fachedaling Informatik Universitat, Frankfurt, 1985.
13. V. Lo, *Heuristic Algorithms for Task Assignment in Distributed Systems*, Proc. 4th Int. Conf Distr. Comput. Syst. (1984), 30–39.
14. D. Towsley, *Allocating Programs Containing Branches and Loops Within a Multiple Processor System*, IEEE Transaction on Software Engineering, vol. SE-12, no. 10 (1986).
15. J. Ullman, *NP-Complete Scheduling Problems*, Journal of Computer and System Sciences **10** (1975), 384–393.