Scheduling Interval Ordered Tasks in Parallel

S. Sunder and Xin He

Department of Computer Science,
State University of New York at Buffalo,
Buffalo, NY 14260, USA

Abstract. We present the first NC algorithm for scheduling n unit length tasks on m identical processors for the case where the precedence constraint is an interval order. Our algorithm runs on a priority CRCW PRAM in $O(\log^2 n)$ time with $O(n^5)$ processors, or in $O(\log^3 n)$ time with $O(n^4)$ processors. The algorithm constructs the same schedule as the one produced by the sequential algorithm (list scheduling). On the other hand, we show that when the precedence constraints are allowed to be arbitrary, the construction of the list schedule is P-complete.

1 Introduction

The problem of scheduling unit execution time tasks on m identical processors under arbitrary precedence constraints has been studied extensively in the past. The problem is known to be NP-hard when m, the number of processors, is part of the input [1]. Polynomial time algorithms are known when m = 2 [2, 3, 4]. The problem with m = 3 is an outstanding open problem in scheduling theory and has motivated considerable research [5]. Polynomial time algorithms for solving this problem are known when m is part of the input and the precedence constraints are trees [6] or interval orders [7]. (see [8] for a survey of results on other special cases of the problem)

The main algorithmic tool employed in obtaining polynomial time sequential algorithms for solving this problem is known as list scheduling. Briefly the method works as follows: Form a priority list of tasks and construct a schedule iteratively by choosing a maximal set of $r \leq m$ independent tasks (tasks with no precedence constraints within them) of highest priority in each iteration. The sequential algorithms for 2-processor scheduling [2], for scheduling interval orders [7], and for scheduling trees [6] are based on the list scheduling method. Helmbold and Mayr [9] showed that the construction of the list schedule (with $m = 2$, arbitrary execution times for tasks and empty precedence constraints) is P-complete, and hence unlikely to be parallelizable. However, NC algorithms based on completely different ideas are known for 2-processor scheduling [10, 11] and for scheduling trees [9, 12].

In this paper, we present the first NC algorithm for solving the m-processor scheduling problem for interval orders. This problem was posed as an open problem in [9]. Our algorithm makes use of structural properties of interval orders and some techniques developed by Bartusch et al [5]. We also strengthen the P-complete result.

* Research supported in part by NSF grant CCR-9011214
in [9] by showing that the construction of the list schedule for general precedence constraint graphs is P-complete, even when all tasks have unit execution time. Surprisingly, our parallel algorithm for scheduling interval orders constructs the same list schedule produced by the sequential algorithm in [7].

The parallel computation model we use is a parallel random access machine (PRAM). The model consists of a number of identical processors and a common memory. In each time unit, a processor can read a memory cell, perform an arithmetic or logic operation and write into a memory cell. Both concurrent read and concurrent write are allowed. In case of a write conflict, the processor with the highest priority succeeds. The present paper is organized as follows. In section 2, we introduce basic definitions and prove our P-complete result. Section 3 presents an algorithm for computing the optimal schedule length for interval orders. Section 4 presents our NC algorithm for scheduling interval orders.

2 Basic Definitions and List Scheduling

Let \( G = (V, A) \) be a partial order (or equivalently a transitive acyclic directed graph) consisting of \( n = |V| \) nodes. We sometimes refer to \( G \) as a precedence constraint graph and the elements of \( V \) as tasks. A node \( u \) is a successor of a node \( v \) if there is a directed path from \( v \) to \( u \) in \( G \). The set of successors of \( v \) is denoted by \( N_G(v) \) (or simply \( N(v) \) if the context is clear). \( v \) is a maximal node if \( N(v) = \emptyset \).

A schedule of length \( t \) for \( G \) on \( m \) processors is an \( m \times t \) matrix \( S \), where the columns are indexed by \( 1, \ldots, t \) and the rows are indexed by \( 1, \ldots, m \). Each task \( x \) is assigned to an unique entry \( (p(x), t(x)) \) in \( S \) such that \( (x, y) \in A \) implies \( t(x) < t(y) \). For any task \( x \in V \), the entry \( (p(x), t(x)) \) denotes that \( x \) is scheduled on processor \( p(x) \) at time instant \( t(x) \). No two tasks are assigned to the same entry in \( S \). The length of \( S \) is denoted by \( ||S|| \). An entry in \( S \) is also called a slot. An entry of \( S \) to which no task is assigned is said to be empty. Two schedules \( S_1 \) and \( S_2 \) for \( G \) are considered to be the same if for every task \( x \) in \( G \), the column assigned to \( x \) in \( S_1 \) is the same as the column assigned to \( x \) in \( S_2 \). (Since the processors are identical, it is irrelevant which processor is assigned to the task.) We denote the sub-schedule of \( S \) consisting of columns \( i, i+1, \ldots, j \) (\( 1 \leq i \leq j \leq t \)) by \( S[i,j] \). Let \( S', S'' \) be two schedules of size \( m \times t_1 \) and \( m \times t_2 \) for two partial orders \( G_1 \) and \( G_2 \). The concatenation of \( S' \) and \( S'' \), denoted by \( S' \circ S'' \), is the schedule of size \( m \times (t_1 + t_2) \) obtained by concatenating the two matrices \( S' \) and \( S'' \).

The following list scheduling algorithm is frequently used in sequential scheduling algorithms. The inputs to the algorithm are: an arbitrary precedence constraint graph \( G = (V, A) \), a precedence-preserving list \( L \) of the tasks in \( V \) (namely if \( u \) is a successor of \( v \) in \( G \) then \( v \) precedes \( u \) in \( L \)), and the number of processors \( m \).

Algorithm List-Schedule \((G = (V, A), L, m)\)

1. \( t = 1 \).
2. While \( L \neq \emptyset \) Do
   (a) Initialize an empty set \( L' \).
   (b) Put the first node \( v \) of \( L \) into \( L' \). Scan \( L \) from left to right. When a node \( w \) is scanned, if \( w \) is independent with every node in \( L' \) (namely, for any \( u \in L' \),