A Parallel Algorithm for Generating Maximal Interval Groups in Interval Databases Based on Schedule of Event Points

Nabil Arman
Palestine Polytechnic University, Palestine

Abstract: In this study, a parallel algorithm to generate all maximal interval groups form a given interval set is presented. The algorithm makes use of intraoperation parallelism to speed up the generation of the maximal groups. The development of efficient algorithms to enumerate all intervals that have certain properties has attracted a large amount of research efforts due to the important role of interval-based reasoning in different areas like rule-based systems, including Expert Systems (ESs), Information Distribution Systems (IDSs) and database systems to name a few. These algorithms are very crucial to answer certain queries about these intervals.

Key words: Maximal interval groups, interval database, parallel databases

INTRODUCTION

Interval-based reasoning has an important role in many areas like rule-based systems, including Expert Systems (ESs), Information Distribution Systems (IDSs) and database systems. Intervals are appropriate and convenient for representing events that span continuous period of time. One may query an interval database to determine what events occur during a given interval. Algorithms to enumerate all intervals that have certain properties have attracted a large amount of research efforts due to the important role of interval-based reasoning in different areas, including rule-based systems and database systems (Aiken et al., 1995; Chamberlain, 1994; Chinn and Madey, 1997; Cormen et al., 2001; Harrison, 1993). These algorithms have an important role in all these systems. An algorithm that finds an interval in an interval tree, represented as a red-black tree, which overlaps a given interval is presented by Cormen et al. (2001). However, the algorithm has the overhead of building and maintaining the interval tree and it can only determine pairs of intervals that overlap. Our algorithm, on the other hand, determines all interval groups that overlap and makes use of intraoperation parallelism to speed up processing.

Many queries in interval databases, including the generation of maximal interval groups, have data requirements that may run into terabytes. Handling such large volumes of data at an acceptable rate is difficult, if not impossible, using single-processor systems. In fact, a set of commercial parallel database systems, such as Teradata DEC series of computers have demonstrated the feasibility of parallel database queries. As a matter of fact, the set-oriented nature of database queries naturally lends itself to parallelization (Silberschatz et al., 2005). In a database of n intervals, there is a need to find all maximal groups, where each group has the intervals that overlap. In a temporal database that stores all courses classes and their times, a query may be asked to generate all groups of classes that meet at a certain time point. In an IDS, it is always needed to check the time validity of rules to determine if they can be chained. This has an important role in controlling the operation of an IDS which is a cornerstone of Command, Control, Communication, Computer and Intelligence (C4I) systems. This study presents a parallel algorithm to generate all maximal interval groups form a given interval set.

Interval grouping parallel algorithm: The generation of the maximal interval groups in interval databases can be parallelized using intraoperation parallelism. The processing of this query can be speeded up by parallelizing the execution of many individual operations involved in the generation of the maximal interval groups. To simplify the explanation and presentation of the algorithm, it is assumed that there are n processors, \( P_1, \ldots, P_n \) and \( n \) disks \( D_1, \ldots, D_n \) where disk \( D_i \) is associated with processor \( P_i \).

There are many approaches to the design of parallel algorithms (Berman and Paul, 2003). One approach is to modify an existing efficient sequential algorithm focusing on those parts of the algorithm that can be parallelized. Another approach is to design a completely new parallel algorithm that may have no natural sequential analog.

The approach used in this algorithm is the first one. A sequential algorithm that generates the maximal interval
groups was presented by Arman (2004). The algorithm doesn't make use of the benefits of parallel architectures which are becoming more popular for query processing in large databases.

An important issue in the design of parallel database algorithms is how data is partitioned among processors and their disks. There are many partitioning techniques, including round-robin, hash partitioning and range partitioning. However, these partitioning techniques have their pros and cons and using any of them in our parallel algorithm is not very useful. For example, partitioning the intervals, in the interval database, in a round-robin technique places intervals, that are related to each other in different disks resulting in many extra passes/merges to perform the grouping. Therefore, our algorithm partitions the interval database based on a schedule of event points to be explained.

Before explaining the parallel grouping algorithm, some concepts that will be used in the algorithm are explained. The algorithm uses the concept of event points and event point schedule (Cormen et al., 2001). An event point is a point on the spatial dimension, where some intervals are leaving a certain interval group and other intervals are entering another interval group. The set of these event points constitutes a schedule of event points. In our algorithm, the real schedule is determined dynamically as the algorithm progresses. The algorithm uses intervals where an interval \( i = [t_i, t_j] \) is represented as an object with fields low \([i] = t_i\) (the low endpoint) and high \([i] = t_j\) (the high endpoint). Two intervals overlap if their intersection is not empty. The algorithm also sorts the intervals in Lexicographic ordering. This can be performed using a parallel sort algorithm like range-partitioning sort or parallel external sort-merge (Silberschatz et al., 2005). An interval set is sorted in lexicographic ordering if whenever \([i, j] < [h, k]\) then either \(i < h\) or \(i = h\) and \(j < k\). Let IS denote an interval set and let \(t_i, t_j, t_k\) denote all potential event points. Let \(t_{\text{last}}\), be high[\text{last interval}], which is an event point representing a guard condition for the algorithm. Let LIG \((t_s)\), denote the Low Interval Group of \(t_s\), which is the set of intervals \( I \) whose \(\text{high}[i] \geq t_s\) and \(\text{low}[i] < t_s\). Let UIG \((t_s)\), denote the Upper Interval Group of \(t_s\), which is the set of intervals \( I \) whose \(\text{low}[i] < t_s\) and \(\text{high}[i] > t_s\), where \(t_{s+1}\) is the next event point. Then for every event point \(t_s\) and its next event point \(t_{s+1}\), \(\text{IG}(t_s) = \text{LIG}(t_s) \cup \text{UIG}(t_s)\). Thus, \(\text{IG}(t_s)\) for event point \(t_s\) consists of the set of intervals whose \(\text{high}[i] \geq t_s\) and \(\text{low}[i] < t_s\) and the set of intervals whose \(\text{low}[i] < t_s\) and \(\text{high}[i] > t_s\), where \(t_{s+1}\) is the next event point of \(t_s\).

The grouping algorithm is implemented by the procedure \(\text{Parallel\_Determine\_Interval\_Groups}\) as given in Fig. 1, which can be invoked with any interval set IS to be grouped into maximal groups, such that each interval group IG has the maximum number of intervals such that for any interval \(I_i\), \(I_j\) in IG, \(I_i \cap I_j \neq \phi\).

\(\text{Parallel\_Determine\_Interval\_Groups}\) algorithm determines all potential event points that represent the set of all distinct low endpoints in the interval set. In doing that, the interval set IS is partitioned and allocated to the \(n\) processors using range partitioning on the intervals' low end points. Each processor \(P_i\) determines all potential event points in its partition locally. The results from processors \(P_1, \ldots, P_n\) are merged together to form event_points.

The interval set IS and event_points are replicated on all \(n\) processors to be used in computing the maximal interval groups. The event points are distributed on the \(n\) processors in a round-robin scheme, where each processor \(P_i\) determines maximal interval group \(G_i\) based on its assigned event point locally. If the number of event points \(m\) is less than the number of processors \(n\), then \(m\) processors are used. The maximal interval groups from processors \(P_1, \ldots, P_n\) are merged to produce the final result.

A draft version of the algorithm appeared by Arman (2006).

**Example:** Consider the interval set: \([0, 1], [0, 3], [0, 5], [0, 7], [0, 9], [0, 11], [2, 13], [4, 13]\) and assume there are 4 processors \(P_1, P_2, P_3, \text{and } P_4\).
The algorithm sorts the interval set if it is not sorted using a parallel sort algorithm. The algorithm then partitions the interval set using range partitioning. Assume the partition vector is $<1, 2, 3>$. Based on this vector, intervals whose low end point is less than 1 are placed on $D_1$. Intervals whose low end points are greater than or equal to 1 and less than 2 are placed on $D_2$. Intervals whose low end points are greater than or equal to 2 and less than 3 are placed on $D_3$. Finally, intervals whose low end points are greater than 3 are placed on $D_4$. Thus, IS is distributed as follows:

- $D_1$ contains $[0,1], [0,3], [0,5], [0,7], [0,9]$ and $[0,11]$
- $D_2$ contains no intervals
- $D_3$ contains $[2,13]$
- $D_4$ contains $[4,13]$

Therefore, the processors determine the event points in parallel as follows:

- $P_1$ determines event point: 0
- $P_2$ determines event point: 1
- $P_3$ determines event point: 2
- $P_4$ determines event point: 4

The event points from $P_1, \ldots, P_n$ are merged to produce the event points 0, 2, 4.

After replicating IS and event points on all $n$ processors, the algorithm distributes the event points on the $n$ processors in a round robin scheme. Therefore,

- $P_1$ is assigned event point 0
- $P_2$ is assigned event point 2
- $P_3$ is assigned event point 4
- $P_4$ is not assigned any event point and is free to be used if needed

The processors determine maximal interval groups based on the event points as follows:

- Event point $t_1 = 0$.
  Next event point $t_2 = 2$ determined from the event points
  $P_1$ determines the maximal group $IG(t_1 = 0) = \{[0,1], [0,3], [0,5], [0,7], [0,9], [0,11]\}$
- Event point $t_2 = 2$.
  Next event point $t_3 = 4$ determined from the event points
  $P_2$ determines the maximal group $IG(t_2 = 2) = \{[0,5], [0,7], [0,9], [0,11], [2,13]\}$

Fig. 2: Comparative performance for the parallel vs. sequential algorithm

- Event point $t_1 = 4$.
  Next event point $t_2 = \text{Null}$ since 4 is the last event point
  $P_4$ determines the maximal group $IG(t_1 = 4) = \{[0,5], [0,7], [0,9], [0,11], [2,13], [4,13]\}$

The maximal groups from the processors are merged to produce the final result.

**Performance evaluation of the parallel algorithm:** To perform the simulation study of the parallel algorithm, a cluster of 4 PCS running a Linux operating system is used as a parallel machine. The implementation language is C with an MPI package. For the sequential algorithm one of these PCS is used. The implementation language is C. To determine the performance of the new parallel algorithms, simulations of the algorithms were performed for random interval databases with 10000, 20000, 30000, 40000 and 50000 intervals. The sequential algorithm was used to generate the maximal interval groups and the time taken for each case was recorded. The same random interval databases were used by the new parallel algorithm and the time taken for each case was also recorded. These numbers were plotted for different interval databases as shown in Fig. 2.

It is important to study the speedup obtained from parallelizing the algorithm of generation of maximal interval groups in interval databases, since it is an important metric for measuring efficiency of parallel database algorithms and the benefit of parallelizing algorithms in general. The ideal situation is obtaining linear speedup. However, a number of factors work against this ideal situation like startup costs, interference and skew. For example, when the number of intervals in the interval database is 50000, the time for the sequential algorithm version is about 320 sec and the time for the parallel algorithm version is about 90 sec. The speedup is
320/90 = 3.55 (sublinear) and that is close to the number of processors/machines used in the simulations which is 4.

CONCLUSIONS

A parallel algorithm for generating the maximal interval groups has been presented. The algorithm is very crucial to answer certain queries about the intervals in an interval set. The algorithm can be used to generate the maximal interval groups needed in many systems, including IDS, expert systems and temporal database systems. The algorithm makes us of intraoperation parallelism to speed up the generation of the maximal interval groups in an interval database. A simulation study demonstrates that the speedup obtained is sublinear.

REFERENCES


266