Arithmetic for Parallel Linear Recursive Query Evaluation in Deductive Databases

Jerome Robinson and Shutian Lin
Department of Computer Science, University of Essex, Colchester, CO4 3SQ, U.K.

Abstract: An arithmetic approach, such as the Level-Finding method described in this paper for evaluating linear recursive queries in deductive database systems provides great potential for parallel processing. It has advantages over other approaches using state-of-the-art parallel processing technology to improve processing speed. In the arithmetic approach, we identify two kind of parallelism, namely bridge node parallelism and formula parallelism. In this paper, the arithmetic foundations and algorithm to exploit formula parallelism were given. The algorithm is fully parallel.

1 Introduction

Deductive databases extend traditional database systems, using logic rules to deduce new facts from stored facts. Rules that use recursion are particularly useful because they specify a succession of repeated operations on data, providing a deductive program rather than a mere relational algebra expression. Users of traditional databases must embed relational queries into an imperative host language to obtain the iterative capabilities of recursive rules. Efficient evaluation of recursive rules is therefore essential in deductive database implementation. The problem is to devise a method that can use query constants to restrict data traffic during evaluation but at the same time ensure termination and completeness.

The Henschen-Naqvi method of evaluation [Henschen 84] like other Wavefront methods [Yu 87, Han 88, Lavington 88, Robinson 90] is efficient because it uses query constants to directly restrict the number of tuples involved in the iterative evaluation. A problem arising from this efficiency is that if cycles exist the derived data has insufficient information to decide termination. Naive and Semi-Naive are alternative evaluation strategies that find the Least Fixed Point of a relational algebraic function. They terminate correctly when no new tuples appear during an iteration. This simple termination condition works regardless of cycles in the EDB because in a fixpoint computation, if none of the arguments change then the result cannot change. The same does not apply to wavefront methods if the underlying data is cyclic. In that case a succession of iterations may produce nothing new, followed by an iteration providing new results. Wavefront methods are graph traversals and a path leading to new nodes may only become accessible once in each circuit of a data cycle. An example illustrates the problem:

The query ?-path(a,y). on recursively defined relation ‘path’:
    path(x,y) :- Link(x,y).
    path(x,y) :- Left(x,v), path(v,w), Right(w,y).

This is the ‘same generation’ rule cluster. The query asks for all nodes in graph ‘Right’ that are the same distance as node ‘a’ in graph ‘Left’ from a common Link tuple. Distance is measured by counting the number of arcs in a path between two nodes.
Using EDB relations: \( \text{Left} = \{ \text{ab, bf, fg, ga} \} \); \( \text{Link} = \{ \text{bc} \} \); \( \text{Right} = \{ \text{cd, de, ec} \} \); whose graph is:

Results can only appear at iteration numbers 1, 5, 9, 13, 17, 21, ... because only those allow access to the Link path. New results appear at iterations 1, 5, and 9. Continuing after iteration 9 will produce no more new results. Steps 1 to 5 produce a new tuple from at least one of the relations. Steps 6, 7, 8 produce no tuples that have not been seen before, but it would be wrong to stop after step 6 (using the LFP termination condition) because new results arise at step 9. It may be necessary to use cycles more than once to obtain all answers, but how do you know when to stop?

Section 2 answers that question. Section 3 discusses aspects of parallelism. Section 4 provides fundamental theorems for the existence of formula parallelism, optimisation, and duplication result identification. Section 5 presents an algorithm for exploiting formula parallelism. Section 6 concludes the paper.

2.1 The Solution

If wavefront methods are used with cyclic data then auxiliary information about the structure of the underlying data graph is needed, to identify accessible levels in \( \text{Left} \) and \( \text{Right} \). The levels are the integer values whose arithmetic forms the basis of the new query evaluation strategy. The process of finding \( \text{Level} \) information can be used as the query answering mechanism itself, rather than simply a control for relational database operations: A node \( n \) in graph \( \text{Right} \) is an answer to the query if and only if there is at least one level the same in sets \( \text{RS}_a(b) \) and \( \text{RS}_p(n) \). \( \text{RS}_p(q) \) denotes the Recurrence Set of node \( q \) with respect to node \( p \). This is the set of distances at which node \( q \) will be encountered if graph traversal starts at node \( p \).

Node \( b \) is a ‘bridge node’, identifiable from the node set in graph \( \text{Left} \) by two criteria: i) it is reachable from node ‘a’, and ii) it occurs as a value of the first attribute in relation \( \text{Link} \). It appears during graph traversal at a number of levels (distances) from node \( a \). Its set of levels is infinite because there is a cycle on the path from \( a \) to \( b \). In general, for any graph structure in \( \text{Left} \), for any simple path from query node ‘a’ to bridge node ‘b’ the Recurrence Set will be either a single level or else an infinite number of levels. (A ‘simple path’ is one in which no node appears more than once). In the graph above, \( b \) occurs at levels \( \{ 1, 5, 9, 13, ... \} \). It recurs at multiples of the cycle length, 4, above a start level, 1. The infinite set of integers can be represented compactly by formula. This allows infinite sets to be transferred rapidly between processors by sending the formula instead of the values. The formula for node \( b \) with respect to \( a \) above is \( 1(4) \), using a notation where \( v(w) \) means \( v + k \times w \) \( \{ k = 0, 1, 2, ... \} \).

[Wu 88] shows that any collection of cycles can be merged to a single virtual cycle, formula \( v(w) \), plus a finite set of levels with values less than \( v-1 \). He also shows an efficient way to obtain level formulae for complex structures of multiple cycles.

A formula is evaluated for each node pair \( <a,b> \) where ‘a’ is a query constant and ‘b’ a bridge node. Parallel evaluation on separate processors is possible since different
node pair calculations need no interaction.

A formula is evaluated for each node pair \(<a,b>\) where ‘a’ is a query constant and ‘b’ a bridge node. Parallel evaluation on separate processors is possible since different node pair calculations need no interaction.

For each bridge node, b, there is a set of RightOrigin nodes, c, obtained by selection using value b on relation Link. Each c node is the origin of a graph, i.e. the root of a (perhaps infinitely deep) tree. The set of nodes in Right is finite, however, and the reachable node set is even smaller. Each of these reachable nodes, n, will be output as an answer to the query if sets RS\(_a\)(b) and RS\(_c\)(n) have a level in common. Calculation of RS\(_c\)(n) for each <c,n> pair can be done in parallel, and answer n can be output if the processor has the formula denoting b’s level set.

2.2 The Algorithm
Algorithm-of-Level-Finding-method for query node ‘a’
Begin 
\[ \text{Determine the set, B, of bridge nodes; i.e. (reachable node) } \cap \Pi_1(\text{Link}). \]
For each bridge node, b, b \(\in\) B :
\[ \text{Compute its Recurrence Sequence set, RS\(_a\)(b), in Left.} \]
\[ \text{Obtain its corresponding set, H, of RightOrigin nodes, by selection from} \]
\[ \text{the Link relation, } \Pi_2(\sigma_{1=b} \text{ Link).} \]
For each h, h \(\in\) H :
\[ \text{(Obtain its set, R, of reachable nodes (by transitive closure in} \]
\[ \text{‘graph Right’ from origin node h).} \]
For each node, r, r \(\in\) R :
\[ \text{Compute RS\(_b\)(r).} \]
If RS\(_a\)(b) \(\cap\) RS\(_b\)(r) \(\neq\) \(\emptyset\) 
\[ \text{Then } r \text{ is an answer.} \]
end.

3.1 Managing Parallelism
The purpose of our research is to develop a parallel implementation of the Level-Finding evaluation strategy. The Level-Finding method has advantages when compared with other parallel systems: (i) the initial workload to be distributed is fairly well defined (by number of nodes and their simple path lengths), and (ii) redistribution of workload at runtime entails little overhead (no movement of code and minimal data transfer). Moreover, we expect the array of parallel processors to operate in conjunction with a database system whose work can overlaps that of the processor array. In a closely-coupled configuration an extended RDB, such as the IFS/2 [Lavington Parle92] able to perform iterative operations such as transitive closure, performs a preliminary operations on the EDB relations, to download only query-relevant tuples. It extracts and downloads the query subgraph from the base relations. This database creates download data as waves of tuples during a breadth-first graph traversal. A wave downloads as the next is created.

After deriving the Query Subgraph the database can identify Bridge Nodes, by Joining the derived relation and the Link EDB relation. This can alternatively be done during the iterative evaluation, by Joining each wave. Bridge nodes discovered during graph traversal are automatically classified by simple path length. This is useful when allocating work to processors, since each task is the recurrence sequence evaluation for a specified simple path. Each processor has a set of these tasks and a set of short paths is likely to finish before a set of long paths.

The database system could download the bridging subset of Link tuples. This allows processors to start path evaluation in Right when their work in Left is finished. However, better load balancing is possible, using information the database can produce: i) the set of <bridge node, RightOrigin node> tuples for each bridge node,
and ii) the set of reachable nodes for each RightOrigin node.

Paths from RightOrigin nodes can be evaluated by a new set of processors (i.e., different from those working on the paths through Left to bridge nodes). However, at some stage the results of path evaluation in Left and Right for each bridge node must be brought together. Full interconnection between processors is not needed. Pairs of processors only need to be connected if they share a bridge node. Communication between them is simply the unidirectional transfer of a Recurrence Sequence formula.

Dynamic redistribution of workload is simple: Shifting a task from one processor to another requires only the transfer of a pair of node identifiers. The processor already contains the necessary graph data and program to identify a path between the two nodes and evaluate its Recurrence Sequence.

### 3.2 Comparison with other Methods

The popular Naive and Semi-Naive methods evaluate the Least Fixed Point (LFP) of the relational algebraic function defined by the recursive rule cluster. They avoid termination problems from cyclic data but are less efficient than wavefront methods. They must materialise the whole virtual relation rather than the subset relevant to the query. The query is finally answered by selecting from the relation tuples with attributes matching the query constants. Loss of results by earlier Selection is discussed in [Agrawal, 88]. Rule rewriting methods, such as Magic Sets, magic counting, and its variants, attempt to emulate the efficiency of the wavefront methods. They are a processing phase before Naive or SemiNaive evaluation. Their purpose is to avoid irrelevant tuples generation during Naive or SemiNaive evaluation. However, they require the materialisation of an auxiliary relation. Since two virtual relations are now materialised instead of one, the benefits of rule-rewriting are not always realised. Sometimes the result is less efficient than the original.

Furthermore, LFP evaluation is inherently non-parallel since it requires every new tuple to be compared with all tuples derived so far. This is the mechanism to detect termination. Using multiple processors results in large communication overheads. Semi-Naive evaluation (more efficient than Naive when implemented on a single processor) suffers even more from this inter-processor communication problem than Naive, because it must do a Set Difference operation at every iteration, to derive the delta for the next iteration. This causes a synchronisation delay at every iteration, whereas Naive only uses Difference as an asynchronous termination detection scheme [Cacace 90]. In contrast, the cycle merging and path length calculating approach [Wu 88] is inherently parallel. If each processor has access to the base relations it has enough information to pursue one or more paths between query constant nodes and result nodes. Separate paths can be pursued in separate processors without interaction.

Although the discussion above referred to the two sided (same generation) rule form the arguments clearly apply in general to n-sided recursions. Further study will extend existing classifications of rule forms [eg Youn 88, Han 89] and show the exact application domain of the new method. The standard evaluation strategy for successful DDB languages is to select the most efficient method of evaluation for each particular query form/rule cluster combination. This approach is not yet available for systems with access to general-purpose multiprocessors, because there are not yet enough general multiprocessor methods to allow it. [Hulin 89, Van Gelder 86, Wolfson, 88] has included other work on this problem. A goal in devising methods for multiprocessor front-ends should be to produce a scheme compatible with the multiprocessor architecture on which other methods, including the most general, will work. Moreover, in order to be accessible to a wide range of users, the multiprocessor
should be a readily available configuration. This criterion applies to the parallel Level-Finding method.

3.3 Bridge Node and Formula Parallelism

From the algorithm of the Level-Finding method above it is evident that processing one bridge node is independent of the others. There is no interaction between different bridge node processes. The performance of bridge node parallelism and the easy load balancing it allows, is currently being assessed experimentally and will be reported in a later paper. At present we provide the theoretical basis for another aspect of parallelism inherent in the Level-Finding approach, namely formula parallelism. Recurrence sequence formulae can be processed in parallel using the algorithm specified in section 5.4. We now provide the arithmetic foundations for that algorithm.

4 Arithmetic Foundations of Formula Parallelism

4.1 Definitions and Notation

Definition 1: (Divisibility) An integer b is divisible by an integer a, not zero, if there is an integer x such that \( b = ax \), and we write \( a \mid b \). When b is not divisible by a, we write \( a \nmid b \).

Definition 2: (Congruence) If an integer m, not zero, divides the difference \( a - b \), we say that \( a \) is congruent to \( b \) modulo \( m \) and write \( a \equiv b \) (mod m). If \( a - b \) is not divisible by \( m \), we say that \( a \) is not congruent to \( b \) modulo \( m \), and in this case we write \( a \not\equiv b \) (mod m).

Definition 3: (Residue) If \( x \equiv y \) (mod m) then y is called a residue of x modulo m. A set \( x_1, x_2, \ldots, x_m \) is called a complete residue system modulo m if for every integer \( y \) there is one and only one \( x_j \) such that \( y \equiv x_j \) (mod m).

Definition 4: (Greatest Common Divisor) The integer \( a \) is a common divisor of \( b \) and \( c \) in case \( a \mid b \) and \( a \mid c \). Since there is only a finite number of any nonzero integer, there is only a finite number of common divisors of \( b \) and \( c \), except in the case \( b = c = 0 \). If at least one of \( b \) and \( c \) is not zero, the greatest among their common divisors is called the greatest common divisor of \( b \) and \( c \) and is denoted by \( \gcd(b, c) \). Similarly, we denote the greatest common divisor \( g \) of the integers \( b_1, b_2, \ldots, b_n \), not all zero, by \( \gcd(b_1, b_2, \ldots, b_n) \).

4.2 Formula of Recurrence Sequence (RS) of a Bridge Node

For a given path from query constant node to a bridge node, the cycles on the simple path can be merged into a virtual cycle [Wu 88]. The virtual cycle length is the greatest common divisor of the cycles. The RS of a bridge node can be represented as a union of two sets. Set 1 is a finite number of integers, denoted as \( F_{\text{set}} \). Set 2 is an infinite integer set, which is expressed as a formula \( d + ke \) denoted by \( d(e) \), where \( d \geq 0 \) and \( e > 0 \). \( d \) is the recurrence start point, \( e \) is the virtual cycle length.

In summary, the RS of a bridge node with respect to the query constant node derived from a simple path can be represented as \( F_{\text{set}} \cup d(e) \), where the sets of integers \( F_{\text{set}} \) and \( d(e) \) are disjoint with all values in \( F_{\text{set}} < (d - 1) \).

Suppose there are several simple paths, \( p_1, p_2, \ldots, p_m \), from query constant node to a given bridge node, then the recurrence sequence of the bridge node is the union of all recurrence sequences derived from all of the simple paths.

In this section, theorems to support the existence of formula parallelism are presented. The main point here is that formulae for bridge node recurrence sequences can not always be merged into one formula.
Take the graph shown by Figure 1 as "graph Left", for example, the simple paths from a to b are $p_1$: $<a, c, b>$ and $p_2$: $<a, d, e, b>$. The length of $p_1$, $|p_1|$, is 2. $|p_2| = 3$. The RS of b w.r.t. a generated from $p_1$ and $p_2$ are represented as $\{\phi\} \cup 2(4)$ and $\{\phi\} \cup 3(5)$ respectively.

![Graph Left](image)

Fig. 1: "graph Left", a and b are query constant and bridge node respectively

The whole set of RS of b w.r.t. a is the union of the RS generated from two simple paths $p_1$ and $p_2$. That's, $RS_a(b) = \{\phi\} \cup 2(4) \cup \{\phi\} \cup 3(5)$. This is the union of two formulae, $2(4)$ and $3(5)$, which denote two sets of integers. The two sets may have values in common, since the two formulae may both generate the same integer. Can a single formula $F_set \cup d(e)$ be found which denotes the same set as $2(4) \cup 3(5)$?

**Theorem 1: [Niven 91]**

1. $a = b \pmod{m}$, $b = a \pmod{m}$, and $a - b \equiv 0 \pmod{m}$ are equivalent statements.
2. if $a = b \pmod{m}$ and $b = c \pmod{m}$, then $a = c \pmod{m}$.
3. if $a = b \pmod{m}$ and $c = d \pmod{m}$, then $a + c = c + d \pmod{m}$.
4. if $a = b \pmod{m}$ and $c = d \pmod{m}$, then $ac = bc \pmod{m}$ for $c > 0$.

**Theorem 2 [Wu 88]:** let $m, n > 0$ and $gcd(m, n) = r$, then for any $t \geq 0$, $r*[(p-1)(q-1)] + r = k_1 \times m + k_2 \times n$, where $p = m/r$ and $q = n/r$, for some integers $k_1, k_2 \geq 0$.

**Lemma 1:** If $a = b \pmod{m}$ and $a = b \pmod{d}$, then $d | m$.

**Proof:** Suppose $a = b \pmod{m}$ and $a = b \pmod{d}$, $d > 0$, then $d \not| m$. According to definition 3 (congruence), the following equations are held:

$$\begin{align*}
    a &= m * k_1 + b \\
    a &= d * k_2 + b \\
\end{align*}$$

That's $m * k_1 + b = d * k_2 + b$. $\Rightarrow m * k_1 = d * k_2$. Take case of $k_1 = 1$, then $m = d * k_2$. Since $m, d > 0, d \not| m$ is held, which is contradictory with the initial assumption. Therefore, Lemma 1 is proved. $\blacklozenge$

**Lemma 2:** If integers generated by formula $d_1(e_1)$ can be produced by an other formula $d(e)$, where $d_1, d \geq 0, e_1, e > 0$, then $e \mid e_1$, and $d_1 = d \pmod{e}$.

**Lemma 2** is easy to be proved by using theorem 1 and lemma 1. $\blacklozenge$

**Theorem 3:** There is no formula $d(e)$ where $d \geq 0, e > 0$ that generates the identical sequence as the union of every two distinct formulae $d_1(e_1)$ and $d_2(e_2)$ where $d_1, d_2 \geq 0, e_1, e_2 > 1$.

**Proof:** Assumption - A formula $d(e)$ can be found to generate identical recurrence sequence as the union of every two distinct formulae $d_1(e_1)$ and $d_2(e_2)$, where $d, d_1, d_2 \geq 0$ and $e > 0, e_1, e_2 > 1$.

According to Lemma 2, $e \mid e_1$ and $e \mid e_2$, therefore, $e$ is one of common divisors of $e_1$ and $e_2$. Moreover, $d_1 = d \pmod{e}$ and $d_2 = d \pmod{e}$.

**Theorem 1.2:**

Let $e$ be the greatest common divisor of $e_1$ and $e_2$, i.e. $e = gcd(e_1, e_2)$. According to the Theorem 2, there exists an integer $k, k = \left\lfloor \frac{e_1}{e} \right\rfloor \cdot \left\lfloor \frac{e_2}{e} \right\rfloor + t$, $t \geq 0$, such that $e \cdot k = e_1 \cdot k_1 + e_2 \cdot k_2$. Take the case of $k_1, k_2 = 1$, integer, $e \cdot k$, generated by the new formula can be represented as: $e \cdot k + d = (e_1 + d_1) + (e_2 + d_2)$. It holds the following
properties: \( e \cdot k + d \equiv e_1 + d_1 + d_2 \) (mod \( e_2 \)) and \( e \cdot k + d \equiv e_2 + d_2 + d_1 \) (mod \( e_1 \)).

Now, let us check if \( e \cdot k + d \equiv d_1(e_1) \) or \( d_2(e_2) \). Suppose \( e \cdot k + d \equiv d_2(e_2) \), then \( e \cdot k + d \equiv d_2 \) (mod \( e_2 \)). Theorem 1. (1), (2), \( d_2 \equiv e_1 + d_1 + d_2 \) (mod \( e_2 \)). Theorem 1. (1).

That’s, \( e_1 + d_1 = 0 \) (mod \( e_2 \)). \hspace{1cm} ---- (eq - x)

Since \( e_1, e_2 > 1 \), (eq - x) is not held naturally. The condition for (eq - x) to be true is as follows:

\[ e_2 \mid (e_1 + d_1). \hspace{1cm} ---- (eq - x1) \]

If \( e \cdot k + d \equiv d_1(e_1) \), for the case of \( k_1, k_2 = 1 \) the following condition should be held.

\[ e_1 \mid (e_2 + d_2). \hspace{1cm} ---- (eq - x2) \]

The truth of equation either (eq - x1) or (eq - x2) is not held to every \( d_1, d_2 \geq 0 \) and \( e_1, e_2 > 0 \), such as, \( d_1 = 2, d_2 = 3, e_1 = 4, e_2 = 5 \). Therefore, some integers generated by the formula \( d(e) \) are not in \( d_1(e_1) \) nor \( d_2(e_2) \). This is contradictory to the assumption. Therefore, theorem 3 is proved.

4.3 Theorems for Merge of Formulae

Arithmetic theories to reduce number of formulæ for the purpose of optimisation are presented in this section. Given a set of formulæ, the merge of formulæ here is the activity of reducing the number of formulæ in the given formulæ set.

Theorem 4: Given any two formulæ \( d_1(e_1) \) and \( d_2(e_2) \), where \( d_1, d_2 \geq 0, e_1, e_2 > 0 \), if \( e_2 \mid e_1 \) and \( d_1 \equiv d_2 \) (mod \( e_2 \)), then the union of these two formulæ can be merged into one formula \( \max(d_1, d_2)(e_2) \) U F_set, where F_set is a finite set of integers less than \( \max(d_1, d_2) \).

Proof: Since \( e_2 \mid e_1 \), \( e_2 > 0 \), applies to any integer, \( l \), if \( l = d_1 \) (mod \( e_1 \)), according to Theorem 1. (5), then \( l = d_1 \) (mod \( e_2 \)).

Since \( d_1 \equiv d_2 \) (mod \( e_2 \)) and \( l = d_1 \) (mod \( e_2 \)), according to Theorem 1. (2), \( l = d_2 \) (mod \( e_2 \)). Therefore, the integers generated by formula \( d_1(e_1) \) and \( d_2(e_2) \) are congruent to \( d_2 \) (mod \( e_2 \)). Since the start point of integers in the union of \( d_1(e_1) \) and \( d_2(e_2) \) is \( \max(d_1, d_2) \), the formula which generates the identical sequence as the union of \( d_1(e_1) \) and \( d_2(e_2) \) after the recurrence start point is \( \max(d_1, d_2)(e_2) \). The recurrence sequences less than the start point form the F_set.

Theorem 5: Given any two formulæ \( d_1(e_1) \) and \( d_2(e_2) \), where \( d_1, d_2 \geq 0, e_1, e_2 > 0 \), if one of \( e_1 \) and \( e_2 \) is 1, then the sequence in the union of formulæ \( d_1(e_1) \) and \( d_2(e_2) \) can be represented by \( F_set \) U d(1), where \( d = d_1 \) if \( e_i = 1 \) (i = 1, 2).

Proof: Suppose \( e_1 = 1 \), then \( e_1 \mid e_2 \). According to Theorem 1. (5), any integer \( l_2 \) generated by formula \( d_2(e_2) \) holds \( l_2 = d_1(e_1) \), i.e., for any integer \( l_2 \geq d_1, l_2 = d_1 + k \cdot e_1, k = 0, 1, \ldots \). If the integers less than \( d_1 \) exist, they are in \( F_set \), the theorem 5 is proven. To apply theorem 5 to a given set of \( n \) formulæ, \( \{d_i(e_i)\} \), we have the conclusion that the union of the \( n \) formulæ can be expressed by \( F_set \) U \( \min(d_1, d_2, \ldots, d_n) \)(1), where \( F_set = \{i \mid i < \min(d_1, d_2, \ldots, d_n)\} \) and \( i \in U \{d_i(e_i)\} \), \( d_i \) (i = 1 to \( n \)) are those for which \( e_i = 1 \).

Theorem 6: Given a set of formulæ \( SF, \{d_i(e_i) \mid e_i \neq 1, i = 1, 2, \ldots\} \), divide \( SF \) into subsets, SFSs. Each subset is defined as: \( SFS_i = \{d_j(e_j) \mid e_j = i, i \) is an integer\}. If \( |SFS_i| = i, \) and \( d_j \equiv d_k \) (mod \( i \)) for \( j \neq k \), then the given set of formulæ can be merged into \( F_set \) U \( \max\{d_j \mid d_j(e_j) \in SFS_i\}(1) \).

Proof: Suppose a subset \( SFS_w \) is such a set of formulæ that meets the conditions of theorem 6. Let \( x_1, x_2, \ldots, x_w \) represent the integers generated by formula \( d_{w1}(e_{w1}), \ldots, d_{wn}(e_{wn}) \).
\[d_{w^2}(e_{w^2}), \ldots, d_{ww}(e_{ww})\] respectively.

\[x_1 = d_{w^j}(\text{mod } w)\]
\[x_2 = d_{w^2}(\text{mod } w)\]

\[\ldots\]
\[x_w = d_{ww}(\text{mod } w)\]

Since \(d_{w_i} \neq d_{w_k} (\text{mod } w)\), where \(i \neq k\), for every integer \(x\) there is one and only one \(d_{w_j}( w \geq j \geq 1)\) such that \(x \equiv d_{w_j}(\text{mod } w)\). According to definition of residue system, the set of \(d_{w_1}, d_{w_2}, \ldots, d_{ww}\) is a complete residue system modulo \(w\). Therefore, for every integer \(x\) if \(x > \max\{d_{w_1}, d_{w_2}, \ldots, d_{ww}\}\), \(x\) can be generated by formula \(\max\{d_{w_1}, d_{w_2}, \ldots, d_{ww}\}(1)\). The integers, which are less than \(\max\{d_{w_1}, d_{w_2}, \ldots, d_{ww}\}\) and in the union of formula form \(F_{\text{set}}\). Based on theorem 5, theorem 6 is correct. The proof is complete.

4.4 Theory for Identifying Duplicate Results

Duplication might occur when recurrence sequence formulae of a bridge node are processed in parallel. In this section, theorems needed to identify the duplicates are given.

Theorem 7: (Cycle Intersection Theorem) [Wu 88] Given \(a, c \geq 0\) and \(b, d > 0\). Then
1) \(a(b) \cap c(d) \neq \emptyset\) if and only if \(|a(b) \cap c(d)| = \infty\) and
2) \(a(b) \cap c(d) \neq \emptyset\) if and only if \(\gcd(b, d)\) is a divisor of the absolute difference of \(a\) and \(c\).

Theorem 7 provides a means to test whether two formulae generate common integers. If two formulae produce common integers, then the identity of those integers can be obtained using theorem 8 (The Chinese Remainder Theorem) below.

Now let us consider the situation of more than two formulae. Suppose we have \(r\) formulae. The simplest case of those \(r\) formulae generating the same integer \(x\) can be represented as the following congruences:

\[x \equiv a_1 (\text{mod } m_1)\]
\[x \equiv a_2 (\text{mod } m_2)\]

\[\ldots\]
\[x \equiv a_r (\text{mod } m_r)\]

Theorem 8: (The Chinese Remainder Theorem). Let \(m_1, m_2, \ldots, m_r\) denote \(r\) positive integers that are relatively prime in pairs, and let \(a_1, a_2, \ldots, a_r\) denote any \(r\) integers. Then the congruences (cong - eq) have common solutions. If \(x_0\) is one such solution, then an integer \(x\) satisfies the congruences (cong - eq) if and only if \(x\) is in the form \(x = x_0 + k* m\) for some integer \(k\). Here \(m = m_1/m_2 \ldots m_r\).

\(x_0\) can be found in the following way [Niven 91]: Writing \(m = m_1 m_2 \ldots m_r, m/m_j\) is an integer. \(x_0 = \sum (m_j/m_j) b_j a_j\), where \(b_j (m_j/m_j) = 1 (\text{mod } m)\). Thus, find out all \(b_j (j = 1, 2, \ldots, r)\), \(x_0\) will be obtained.

Theorem 7 gives necessary and sufficient condition for any two formulae to generate common integers. Theorem 8 identifies integers generated by more than \(1\) formula based on the hypothesis that the moduli \(m_i\) should be relatively prime. When \(m_i\) is not relative prime, the solution is: if \(x_0\) is one of the solutions, then an integer \(x\) is a solution if and only if \(x = x_0 + k* \text{lcm}(m_1, m_2, \ldots, m_r)\) for some integer \(k\), where \(\text{lcm}(m_1, m_2, \ldots, m_r)\) is the least common multiple of the \(m_i\).

5 Exploitation of Formula Parallelism

As described in sections above, the recurrence sequences of a bridge node may be a union of several formulae which may not be reducible to a single formula \(F_{\text{set}} \cup d(e)\). The formulae associated with a bridge node can be processed in parallel -
different formulae on different processors.

The operation of processing formulae in parallel, in fact, is to traverse "graph Right" in parallel to retrieve the answers to the given query. If the "graph Right" for processing the given query is available to any processor then any formula can be processed by any processor, since the necessary information for retrieving answers is the levels generated by a formula and the graph from which the answers are retrieved. There is no communication between processors during formula processing.

The more formulae associated with a bridge node, the more parallelism we would be able to exploit. However, sometimes it might not be necessary to process all of them since some formulae may be implied by others. Therefore, before processing the formulae associated with a bridge node, it is useful to filter out formulae which do not contribute to the RS. This filtering work is supported by the theorems mentioned in section 4.3.

5.1 Algorithm of Merging Formula

Reviewing the theorems in section 4, if one of the given formulae is in the form \( d(1) \), then all the formulae can be merged into \( F_{set} \cup d_{x}(1) \) according to theorem 5. Therefore, the first step of the algorithm is to identify any formula in the form \( d(1) \). Also, in any step of processing, identification of formula \( d(1) \) is important in order to save processing time, since during formula merging, a formula in form of \( d(1) \) may be generated. If there is no formula \( d(1) \) then the formulae implied by others will be checked.

It is a simple arithmetic calculation to check whether a formula \( d_{1}(e_{1}) \) is implied by another formula \( d(e) \). According to theorem 4, if two formulae \( d_{1}(e_{1}) \) and \( d_{2}(e_{2}) \) meet two conditions: (1) \( e_{1} \) is divisible by \( e_{2} \), and (2) \( d_{2} - d_{1} \) is divisible by \( e_{2} \), then formula \( d_{1}(e_{1}) \) is contained in formula \( d_{2}(e_{2}) \). Thus, formula \( d_{1}(e_{1}) \) can be removed from the formula set.

The complete residue system modulo \( m \) where \( m \) is an integer greater than 1, is an important concept in merging formulae. Once a complete residue system modulo \( m \) is found, the whole set of formulae can be merged into a formula in the form \( d(1) \).

Even though there is no formula containing another formula in the given formula set, it might be the case that a subset of formulae makes up a complete residue system modulo \( m \). A simple example is when the given formula set is \{ 0(3), 1(3), 2(3) \}. The formulae in the given set don’t contain each other. However they form the complete residue system of modulo 3. Thus, the whole set of formulae can be merged into the formula \( 0(1) \).

It is also an arithmetic operation to see if a set of formulae makes up a complete residue system. A necessary condition for this is that the number of formulae be equal to the modulus, i.e. the \( e \) in formula \( d(e) \). If the necessary condition is met, then according to theorem 6 each pair of \( d_{i}, d_{j} \) where \( i \neq j \) should be not congruent to each other under modulus \( e \). If this second condition is satisfied, a complete residue system to modulus \( e \) has been found.

The formula merging algorithm can be described as the following procedure:

Algorithm of merging formulae:

Begin

Search the set of formulae \( d_{i}(e_{i}) \) SF, to obtain a set SF\(_{1} \), with \( e_{i} = 1 \).

If \( SF_{1} \neq \phi \)

Then merge the formulae into one formula;

Else

Filter out unnecessary formulae;

Check if there exists a complete residue system;

If there is a complete residue system

Then merge the formulae in SF into one formula.

End
5.2 Duplicate Removal

Duplication of result values could occur when processing formulae on different processors, since a node in "graph Right" could satisfy more than one formula. If the result from parallel processing formulae involves further operations, duplicates would cause an endless-loop. Therefore, removal of duplicates is essential.

The reason duplication occurs is that an integer is generated by more than one formula. There are two solutions to removing duplicates: 1) filter out the duplicates in the answer sets; 2) identify in advance the Duplicate Integer Set (DIS) and retrieve the nodes at those levels from the "graph Right" only once from the nodes in "graph Right" at levels specified by the integers in set DIS.

Solution 1) is simple but will produce increased communication between processors and so reduce processing speed. With increasing size of individual answer sets, the load of message passing between processing elements will become severe.

Solution 2) has to identify the set of integers generated by more than one formula. It does not need communication between processing elements. Thus, solution 2) is preferable.

If two (or more) formulae create common integers then the number of integers is infinite. However, the infinite sequence of integers is in a regular pattern according to theorem 8 (The Chinese Remainder Theorem). In other words, the common integers generated can be represented by a formula in the same form as \( x_0(m) \). This is an advantage since the newly generated formula can be processed in the same way as the original formulae.

It is simple to test whether two formulae will generate common integers by using theorem 7. If two formulae generate common integers, then the constraining-formula representing the common integers can be found using theorem 8. The constraining-formula is used to ensure that any common integer of two formulae is processed only once.

5.3 Constrained-Relationship Matrix (CRM)

The constrained-relationship Matrix is an \( n \times n \) matrix, where \( n \) is number of formulae. Elements of a CRM take values 0 or 1. The value 0 for element \( CRM_{ij} \) indicates that the \( i \)th and \( j \)th formulae don’t generate common integers. On the other hand, a 1 indicates the \( i \)th and \( j \)th formulae produce common integers. For example, if a given formula set is \( \{1(3), 2(6), 1(8)\} \), then the constrained-relationship matrix of the formula set is as follows:

\[
\begin{array}{ccc}
1(3) & 2(6) & 1(8) \\
1(3) & x & 0 & 1 \\
2(6) & 0 & x & 0 \\
1(8) & 1 & 0 & x \\
\end{array}
\]

It shows, for example, that formulae 1(3) and 1(8) will produce duplicate results whereas 1(3) and 2(6) will not. In a CRM matrix, element \( CRM_{ij} \) takes value x for obvious reasons.

5.4 Algorithm for Parallel Processing of Formulae

The algorithm for parallel processing of formulae consists of distribution and manipulation strategies. The distribution part assigns formulae to different processors for processing. The manipulation part is what each processor does to retrieve duplicate-free answers to the given query. In order to describe the algorithm clearly, let us make the following assumption:

A bridge edge is \(<b, g>\); the recurrence sequence formulae of bridge node \( b \) with
respect to the query-constant constitute set \( SF \);

**Distribution-part** /* Algorithm-of-parallel-processing-formulae*/

begin {
    1. Merge formulae in set \( SF \) to produce set \( SF' \);
    2. If \( |SF'| = 1 \) /* only one formula in set \( SF' \) */
        Then Process formula;
        Else
            Create constrained-relationship matrix of \( SF' \);
            for \( i = 0; i < n; i++ \)
              for \( j = 0; j < i - 1; j++ \)
                if \( CRM_{ij} == 1 \)
                    Put constraining-formula of \( ith \) and \( jth \) formulae in set \( CF_i \);
                    Send \( ith \) formula with its constraining-formula set \( CF_i \) to a processor
                    for processing;
            }
}

**Manipulation-part** (formula\(_i\), \( CF_i \))

begin {
    1. Retrieve reachable node set \( RNS_g \) from node \( g \) in "graph Right"
    2. For each node \( n \), \( n \in RNS_g \):
        
        Calculate \( RS_g(n) \);
        
        If the formula of \( RS_g(n) \) generates common integer as formula \( i \) and
        NOT as any formula in its constrained formula set \( CF_i \)
        Then \( n \) is answer.
    }
}

The manipulation part of the algorithm is modification of the algorithm we used in the Level-Finding approach. This algorithm is fully decomposable. During formula processing, no communication is required between the parallel processing elements.

6 Conclusion

The arithmetic approach used in the Level-Finding method for evaluating recursive queries in deductive database systems provides great potential of parallel processing. In this paper, we identified two kinds of parallelism, i.e. *bridge node parallelism* and *formula parallelism*. The arithmetic foundations and algorithm to exploit *formula parallelism* were given. The algorithm is parallel. No communication is necessary during processing each formula and termination is easily controlled.

The usual approach in efficient DDB systems is to select the most efficient evaluation strategy that will work with the current query. However, the existing repertoire of standard evaluation procedures lacks methods able to utilise a standard multiprocessor platform. The current paper makes a start at remedying that deficiency.

Although many operations on sets of tuples are inherently parallel, existing algorithms to implement those operations include features which prevent parallel processing. Features such as inter-processor data traffic and synchronisation delays degrade performance badly.

We have introduced a parallel strategy for evaluating an important (widely used) class of recursive queries in Deductive Databases, and provided new theoretical results to allow the efficient use of formula-level parallelism.

The magic sets method and its variants for cyclic data, magic counting [Sacca 87, Greco 92] are rule rewriting methods for Naive and Semi-naive evaluation. They are designed to improve the efficiency of Naive or Semi-naive evaluation by reducing the number of irrelevant tuples involved. But since those methods are not easily parallelised it is beneficial to seek alternative evaluation strategies. An advantage of graph traversal methods, including the Level Finding method, is their ability to use query constants directly, without the need for rule rewriting. A disadvantage is their
lack of generality. However, chaining recursions include many of the most popular queries, so an efficient parallel evaluation strategy for them is useful in a DDB’s repertoire of query processing methods, since overall performance depends on the most frequent forms of query.

Acknowledgements

It is a pleasure to acknowledge the stimulating research environment provided by the IFS research group at Essex University. Investigation of the Parallel Level Finding Method for recursive query evaluation is supported by SERC research grant GR/H/17701.

References


Cacace, Filippo; Ceri, Stefano; and Houtsma, Maurice A.W., “An Overview of Parallel Strategies for Transitive Closure on Algebraic Machines”, Proc PRISMA 90, pp 48-66.


Han, Jiawei; Qadah, Ghassen; and Chau, Chinying, “The Processing and Evaluation of Transitive Closure Queries”, Intl Conf on Extending Database Technology - EDBT 88, Venice, 1988, pp 49-75.

Han, Jiawei, “Compiling General Linear Recursions by Variable Connection Graph Analysis”, Comput. Intell. 5, pp12-31, 1989.


Hulin, Guy, “Parallel Processing of Recursive Queries in Distributed Architectures”, VLDB Conf 1989, pp 87-96.


Wolfson, Ouri; and Silberschatz, Avi, “Distributed Processing of Logic Programs”, Proc ACM SIGMOD 88, pp 329-336.

Wu, Ching-Shyan; and Henschen, Lawrence, “Answering Linear Recursive Queries in Cyclic Databases”, FGCS 88, Vol 2, pp 727-734.


Youn, Cheong; Henschen, Lawrence J.; and Han, Jiawei, “Classification of Recursive Formulas in Deductive Databases”, Proc ACM SIGMOD 88, pp 320-328.