PROCESSING THE TRANSITIVE-CLOSURE LOGIC RULES ON
SHARED-NOTHING MULTIPROCESSOR SYSTEMS

Ghassan Z. Qadah
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois 60208

Abstract
This paper presents and evaluates, using rigorous analytical models, a number of parallel algorithms suitable for processing an important class of recursive queries, the instantiated transitive closure (TC) queries. These algorithms are variants of the sequential 6-wavefront algorithm, designed to run on a shared-nothing (message-passing) type of multiprocessor systems. The results obtained in this paper indicate that the relative performance of these algorithms is a strong function of not only the parameters which characterize the system's hardware, but also those parameters which characterize the database and the processed query. Two parallel algorithms have been identified to be the best performing ones, one for systems with slow interconnection networks and the other for systems with fast networks.

1. Introduction
Efficient processing of transitive-closure (TC) rules and queries has received much attention lately because of their role in improving the intelligence of database systems (an excellent review of the work done in this area can be found in [8]). In general, a logical rule is linearly recursive if the rule's head predicate appears only once in its body. A linear recursive rule is a transitive closure (TC) rule if its body contains only one-sided join with the recursive predicate. For example, the rule
\[ A(X, Z), R(Z, Y) \]

is a TC rule, where \( A(X, Z) \) is an extensional (base) predicate. Within the context of deductive databases [1], \( A(X, Z) \) is defined by a two-attribute normalized database relation with very many tuples. To generate solutions from a TC rule, another non-recursive rule, the exit rule must exist. For example, the rule
\[ A(X, Y) \]

is an exit rule. A TC query is a headless rule which is defined by a transitive closure rule and its associated exit rule. For example, \( A(X, Y) \) is a TC query. In general, a 2-place unit query, such as \( A(X, Y) \), may take on different forms. Here we are concerned with the "partially instantiated" type of queries. More specifically, we instantiate a variable in a query, say \( X \), to a set of constants; then as a result of processing the query, the other variable \( Y \), will be assigned a (possibly empty) set of constants. More formally then, the solutions to the query \( A(c_1, c_2, \ldots, c_n, y) \) is the set \( \{ y | R(c_i, y) \text{ is true and } i \in \{1, 2, \ldots, n\} \} \).

The generation of solutions for a logical query can be done using a compilation-based method [4, 7], in which the set of rules which defines the query is transformed into an equivalent set of relational algebra expressions each of which contains only extensional (base) relations. The union of the solutions to these expressions generates the solutions to the original query. A well-known sequential algorithm which uses the compilation-based technique to generate solutions to an instantiated TC query is the 6-wavefront algorithm [2, 3]. This paper develops and evaluates, using rigorous analytical models, the performance of several parallel variants to the 6-wavefront algorithm. These variants assume the shared-nothing type of multiprocessor systems [6]. A machine of this type consists of a set of processing nodes, each of which has a processor, a local high-speed main-memory and a secondary storage device. A suitable message-passing network is used to interconnect these processing nodes. In this study, we assume that the interconnection network is a token-bus capable of performing both point-to-point and broadcasting types of communication.

The rest of this paper is organized as follows. Section 2 presents the sequential 6-wavefront algorithm and a number of its parallel variants. Several comparative performance studies to the parallel algorithms are presented in section 3. Finally, section 4 presents some concluding remarks.

2. The 6-wavefront Algorithm and its Parallel Variants
From the 6-wavefront algorithm (and its parallel variants) point of view, the binary base relation of the transitive closure rules, \( A(X, Y) \) for example, is viewed as a graph (refer to Figure 1): nodes in the graph correspond to the values in the attributes, \( X \) and \( Y \), and directed edges correspond to tuples. In general, the base relation, as shown in Figure 1, forms a directed cyclic graph. The generation of solutions for the instantiated TC query can be viewed as the collection of those nodes in the graph reachable from the nodes representing the query constant symbols. The 6-wavefront and its various parallel variants are presented next.

The 6-wavefront Algorithm
The 6-wavefront algorithm is an iterative (breadth-first search) algorithm described in [2, 3], however, for the completeness of this paper, we present it here again. It starts out from some initial nodes in the base graph (the ones which correspond to the constants in the processed query) and iterates to find all of the nodes reachable from these initial nodes (the values associated with these nodes constitute the solutions to the processed query). The algorithm, as shown in Figure 2, employs two tem-

Figure 1: A Sample Binary Relation A. (a) In Table Form and (b) In Graph form.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Procedure δ-wavefront (input: query-constants, A(base relation); output: closure)

begin
wave and closure are unary relations.
A is a binary relation.

wave = \{query-constants\};
closure = \ø;

While (wave ≠ \ø) Do

wave = \{wave(X) \in A(X,Y); + + +\}
wave = wave \cap closure;

closure = wave \cup closure;
endwhile

Figure 2: The δ-wavefront Algorithm.

The parallel/distributed variants of the δ-wavefront algorithm can be thought of as points in a two dimensional space with the "scheme type adopted for storing the base relation and closure" as these dimensions (The storage scheme type for the wave is dependent on the scheme types used to store the base and the closure relations and therefore, it can not be used as a third dimension). Since each one of these relations, as discussed above, can be replicated, randomly-distributed or cluster-distributed, potentially nine different parallel variants to the δ-wavefront algorithm can be generated. The name of each one of these algorithms can be constructed from the word PDWF (to indicate parallel δ-wavefront) subscripted by a 2 digits string, the first/second digit indicates the scheme used by the algorithm to store the base/closure (with the value 1/2/3 to denote replicated/random/cluster distribution), respectively. Therefore, PDWF12 refers to the parallel δ-wavefront algorithm which replicates its base relation and randomly distributes its cluster one. Studying nine different algorithms is a tedious task, fortunately we do not need to do so for two reasons. First, some of the variants generated as indicated above are meaningless. That is, they do not constitute valid parallel δ-wavefront algorithms. Second, some variants are obviously performing worse than others and therefore can be discarded.

For the two reasons presented above, we will present and study only a selected subset of the possible δ-wavefront variants. These are: PDWF21 (randomly-distributed base relation, replicated closure), PDWF31 (cluster-distributed base relation, replicated closure), PDWF23 (randomly-distributed base relation, cluster-distributed closure) and PDWF33 (cluster-distributed base relation, cluster-distributed closure). To facilitate the formal presentation of these variants, we have defined two network communication operations, send and receive. These operations assume that each one of the processing nodes is assigned a unique ID between 1 and n, where n is the number of nodes attached to the network. The operation send (data, Sender= sender-ID, Receiver= receiver-ID) is used by a PN whose ID is sender-ID to send data to another PN, receiver-ID. data can be modified with a selection expression. The instance send(wave,Cont.) means only those elements of wave which return the value j when a hash function is applied to them. Hence, send(wave,Cont.) means "transmit only the j-th portion of wave". receiver-ID can be either an ID of a PN, e.g., j or Broadcast. If Receiver=j is used, only PNj can receive data. If the other hand Receiver=Broadcast is used by a sender, every PN can potentially receive data. Send=Broadcast seems rather meaningless.

The operation receive (location, Sender= sender-ID) is used by a PN to receive data into location sent by sender-ID. At the risk of stretching slightly the semantics of location, we adorn it with a selection expression: receive(location,Cont.) means "store only the j-th portion of the incoming data into location". Sender=j enables only data sent by PNj to be received. Another operating mode would be Sender=Broadcast, in which case the sender is not distinguished. However, this mode can in general lead to an error when there are multiple senders, therefore, it is not used. Next, the different parallel variants are presented.

A. The PDWF21: The PDWF21 algorithm, as shown in Figure 3, randomly distributes the base relation A among the various PNs. However, it maintains throughout its execution a replicate copy of closure and wave in each one of these nodes. It starts out by sending the set of query constants to every PN to be stored as wave. Each PN then executes one iteration of a sequential δ-wavefront algorithm (the first three statements in the "While" loop of Figure 3), thus generating a new local wave and updating its local closure according to such a relation. At this
PDWF$_{31}$

$\mathcal{A}(X,Y)_{\text{local}}$ is the portion of $\mathcal{A}(X,Y)$ stored in a PN.
Each PN is assigned a unique id between 0 and $n-1$.
Each PN simultaneously executes an identical copy of this program.
$Q_c$ is the set of query constants.

if ($ID = Master$) /* broadcasting query constants */
    
    wave $= Q_c$.
    send($Q_c, Sender = Master, Receiver = Broadcast$);
else
    receive(wave, Sender = Master);

while ($\text{wave} \neq \emptyset$)
    
    /* obtain local wave */
    wave $= \pi_Y \{ \text{wave}(X) \mid \text{local} \}$;
    /* obtain local wave */
    wave $= \text{wave} - \text{closure}$;
    closure $= \text{closure} \cup \text{wave}$;
    sendwave $= \text{wave}$;

    for ($i = 0$ to $n-1$) /* broadcast sendwave */
        if ($i = ID$)
            send(sendwave, Sender = ID, Receiver = Broadcast$);
        else
            receive(tempwave, Sender = i);
    sendwave $= \text{sendwave} - \text{tempwave}$;
    closure $= \text{closure} \cup \text{tempwave}$; /* update closure */
    wave $= \text{wave} \cup \text{tempwave}$; /* construct global wave */

Figure 3: The PDWF$_{21}$ Algorithm.

After the global wave and closure are constructed in each one of the processing nodes, PDWF$_{21}$ proceeds through another iteration of the "While" loop (sequential 6-wavefront algorithm). This repetition continues until the globally constructed wave becomes empty.

B. The PDWF$_{31}$: The algorithm PDWF$_{31}$, shown in Figure 4, assumes that the base relation $\mathcal{A}$ is partitioned using a suitable hashing function into $n$ (roughly) equal segments, each of which is stored on a different processing node. Furthermore, PDWF$_{31}$ partitions the set of initial query constants into $n$ segments using the same hash function as above and distributes these segments to the processing nodes to be stored as local wave's. Therefore, PDWF$_{31}$ starts with each one of the local wave's containing only those values which may appear in the $X$ attribute of the base relation segment stored in the corresponding node. It also maintains a replicate copy of the closure in each one of the processing nodes.

PDWF$_{31}$

$\mathcal{A}(X,Y)_{\text{hash=ID}}$ is the portion of $\mathcal{A}(X,Y)$ assigned to PN$_{ID}$.
Each node simultaneously executes an identical copy of this program.
At end of execution, each PN has a conv of the solutions in closure.

if ($ID = Master$)
    
    wave $= Q_{c, hash=ID}$; /* query constants distribution */
    send($Q_c, Sender = Master, Receiver = Broadcast$);
else
    receive(wave$_{hash=ID}$, Sender = Master);

while ($\text{all wave's} \neq \emptyset$)
    
    /* obtain local wave */
    wave $= \pi_Y \{ \text{wave}(X) \mid \text{local} \}$;
    wave $= \text{wave} - \text{closure}$;
    closure $= \text{closure} \cup \text{wave}$; /* local update to closure */
    sendwave $= \text{wave}$; /* for broadcasting */
    /* start building the new local wave */
    wave $= \text{wave}_{hash=ID}$;

    for ($i = 0$ to $n-1$) /* broadcast sendwave */
        if ($i = ID$)
            send(sendwave, Sender = ID, Receiver = Broadcast$);
        else
            receive(tempwave, Sender = i);
    sendwave $= \text{sendwave} - \text{tempwave}$;
    closure $= \text{closure} \cup \text{tempwave}$; /* global update to closure */
    /* continue building the new local wave */
    wave $= \text{wave} \cup \text{tempwave}_{hash=ID}$;

Figure 4: The PDWF$_{31}$ Algorithm.

PDWF$_{31}$, as shown in Figure 4, proceeds with executing (in all of the nodes simultaneously) a step of the sequential 6-wavefront algorithm as in PDWF$_{21}$ with one difference: each PN keeps all newly, locally generated values in its sendwave for broadcasting and retains only a small portion (those values satisfying "hash=ID") for its next wave. When the sendwave of another PN is received (into tempwave), a PN updates its closure with the entire tempwave (to maintain a replicated closure on each node) but uses only a portion of tempwave (those values satisfying "hash=ID") to update the local wave (to maintain a cluster-distributed wave).

After one round of local wave broadcasting by all PNs is finished, PDWF$_{31}$ repeats another iteration of the "While" loop of Figure 4. The iterations continue until the wave becomes empty. However, if a PN terminates the execution simply because its local wave is empty, an error might occur. What if another PN sends later values to this now-terminated PN? As a solution to this problem we assume the existence of an intelligent controller on the network which monitors the status of the local waves. When all waves are empty, the controller generates "Everyone May Now Terminate" signal, and only upon the arrival of such a signal the PNs can safely stop execution.

C. The PDWF$_{32}$: The algorithm PDWF$_{32}$, shown in Figure 5, assumes that the base relation $\mathcal{A}$ is randomly partitioned into $n$ segments ($\mathcal{A}_{\text{local}}$'s), each of which is stored on one of the processing nodes. Moreover, the elements of the closure generated as a result of processing this algorithm is to be stored among the processing nodes in a clustered fashion (using some hashing function). PDWF$_{32}$ starts by sending all of the initial query constants to each one of the processing nodes where it is stored as a (local) wave. wave is then used by every processing node...
to probe its local segment of the base relation to generate a new local wave. Unlike the algorithms presented earlier, PDWFs delays updating local closure (after each JOIN and PROJECTION step) until all of the newly generated waves are hashed to the processing nodes and unioned (first for-loop in Figure 5) to form in each PN the portion of the generated (global) wave which must be used to update its local closure. Notice that a PN, whose id is ID, does not have to receive the entire content of sendwave from another PN, but only its portion which satisfy "hash=ID". Here we assume that a sender node still broadcasts its entire sendwave to all of the nodes in the system, however, a receiver node accepts only a selected subset of the broadcasted wave values.

D. The PDWFm: The algorithm PDWFm assumes that the base relation A is partitioned using a hashing function into n equal segments (A\_{hash=ip}'s), each of which is stored on one of the processing nodes. Furthermore, the closure generated as a result of processing this algorithm is to be stored likewise. PDWFm proceeds exactly as PDWFs (refer to Figure 5) except that the quantity A(X,Y)\_{local} in PDWFs must be replaced by A(X,Y)\_{hash=ID} to denote the fact that the base relation is clusterly distributed and not randomly distributed. Furthermore, the last part of the algorithm PDWFs, the global wave construction part, must also be omitted since the algorithm PDWFm does not need to construct such a wave.

3. A Comparative Performance Study for the Parallel Algorithms

The performance of the various parallel algorithms relative to the \(6\)-wavefront algorithm and relative to each other is investigated in this section using the well known measure speedup (S). This measure is defined for a given parallel algorithm \(X\) as the ratio "total-execution-time (\(6\)-wavefront)/total-execution-time (X)". The Derivation of formulas to compute the total-execution-time for the \(6\)-wavefront and the different parallel variants will not be presented in this paper due to the limits imposed on the number of its pages, however, the interested reader can refer to [5] for these formulas and their derivations. It is interesting to note here that these derivations make a number of important assumptions concerning the base relation and the processed queries, namely:

1. The base relation \(A\) is binary with attributes \(X\) and \(Y\) and with cardinality \(N_t\). The values in the attributes \(X\) and \(Y\) are randomly drawn with replacement from the same domain. This domain is assumed to be finite with a size \(N_d\) and the probability to draw its values is uniform. By fixing the value of \(N_t\) and changing the value of \(N_t/N_d\) (\(=N_r\), the out-degree of nodes in the graph representing the base relation), one can generate directed cyclic graphs with fixed number of arcs (i.e. base relations of the same cardinality) but variable out-degree nodes.

2. Relation \(A\) is stored in pages using a hash-based scheme. A hash function uses the \(x\) attribute value of a given tuple from \(A\) to compute the address of the page in which that tuple is stored. The pages storing relation \(A\) are assumed to be distributed between the system's main-memory(ies) and disk(s).

3. The generated solutions for a processed query is stored using a scheme similar to the one used by the base relation.

4. The different query constants \((Qr*\times N_t)\) of them, where \(Q_r\) is the ratio "number of query constants/\(N_t\)" are randomly drawn without replacement using a uniform distribution, from the domain underling the two attributes of the base relation.

In this section, two performance studies are carried out. The initially master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure. Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.

Initially, the master PN, Master, broadcast \(Q_r\) to all PNs. At end of execution, each PN has a portion of the final answer in closure.
base relations with low $\text{OutDegree}$ and $T_{\text{net}} \approx 8$ msec (where $T_{\text{net}}$ is the time needed to broadcast a page across the token-bus). This figure shows that the four studied algorithms share almost the same speedup behavior with respect to $n$: rapid (exponential) increase in speedup when $n < 10$, and rapid decrease in speedup when $n > 10$. The drastic change in the performance at $n = 10$ is easy to explain, since $10^2$ is the value of $n$ at which the base relation is completely stored in the main-memory of the processing nodes (at $n = 10$, the closure is memory-resident too).

At $n < 10$ the base relation (and probably the closure) is not main-memory resident which implies that the processing of the TC queries incurs disk I/O's. The rapid increase in speedup is, therefore, due mainly to decrease in the system's IO. At $n > 10$, the system relations are main memory residents and therefore, no disk I/O advantages can be obtained by increasing $n$. The drop in speedup is due to the fact that the decrease in the CPU time due to the increase in $n$ is much less than the extra communication time introduced by nodes on the network, hence a net decrease in the system's speedup.

The speedup of the parallel algorithms has been found to be a simple function of $N$, (the base relation's $\text{OutDegree}$), increasing as $N$ is increased (compare Figures 7 and 8 to Figure 6). The slope of speedup curves for $n > 10$ has also been found to be a strong function of $N$, decreasing as $N$ increases in value (compare Figures 6 and 7). At large enough values of $N$, as shown in Figure 8, such a slope becomes positive over the whole range of the parameter $n$. That is, the speedup of the parallel algorithms (especially PDWF$_{31}$ and PDWF$_{33}$) increases (or relatively as $n$ is increased).

It is interesting to observe in Figures 6 through 8 that the algorithm PDWF$_{33}$ is faster than PDWF$_{31}$ and both of these algorithms are faster than PDWF$_{21}$ and PDWF$_{23}$. That is, as cluster distribution of base relation is better than the random one, and the cluster distribution of the closure is better than replicating it.

The speedup of each one of the parallel algorithms has been found to increase as the parameter $Q$, (the query constant ratio's) is increased (compare Figures 6 and 9). $T_{\text{net}}$ has also been found to affect the performance of the parallel algorithms. In general, increasing $T_{\text{net}}$ brings a decrease in the speedup of the parallel algorithms (compare Figures 8 and 10). Furthermore, when the network is slow enough, the algorithms PDWF$_{31}$ and PDWF$_{21}$ perform better than PDWF$_{33}$ and PDWF$_{23}$ as shown in Figure 10. That is, replicating the closure is better than clustering it. An interesting finding.

From the study presented above, the following conclusions can be reached:

- Although, the relative performance of the parallel algorithms is a strong function of many parameters, nevertheless, the algorithms PDWF$_{33}$ and PDWF$_{31}$ are the best performing ones. The former loads the network in a distributed system more than the local disk and therefore it is suitable for multiprocessor systems having fast interconnection networks, whereas, the latter algorithm loads the local disk more than the network and therefore it is suitable for machines with slow networks.

- The optimal number of nodes needed to process a given best performing algorithm is strongly dependent on the amount of main-memory available in a node (physical resources) and on $N$, (the base relation).

- When processing a TC query on a distributed system, it is very important to use the system's parallelism wisely since a wrong decision might result in a severe increase in the query processing time as a result of over deploying too many nodes. A case in point is the processing of a query against a base relation with $N = 1$ on a 10 node system versus a 100 node system (refer to Figure 6). Making the wrong decision, that is choosing the system with the 100 nodes, will substantial increase the execution time of the query (about 20 times).

- Although, the amount of information transmitted over the network as a result of executing a TC algorithm is a function of very many parameters, nevertheless it is relatively large. Suitable data compression schemes must be found to help reduce their volumes especially when $n$ is large.

Case 2: The Constant Main-memory System

This performance study assumes a constant main-memory system. That is, for a given total amount of main-memory in the system, $M$, the amount of main-memory per processing node, $M_{\text{per node}}$, is $M/n$. Figure 11 through 13 plots the performance measure speedup of the parallel algorithms versus $n$ for $N_*=1,1,3$, and 5, respectively. The parameters $T_{\text{net}}, Q$, and $M$ (divided equally between the base relation and the closure) are assigned the values .8 msec, .001 and $2^N$, respectively. The value assigned to $M$ implies that the base relation and the closure is completely contained in the main-memory of the uniprocessor system. Furthermore, it implies that the base relation (as well as the closure of some parallel algorithms) are completely stored in the main-memories of the distributed system. Such an environment represents the worse case performance for the distributed system (relative to the uniprocessor one) since we deprive such a system from performing parallel disk I/O's.

Figure 11 through 13 show that it is possible to gain some speed advantages (over a uniprocessor system with similar amount of main-memory) by processing the TC query using the PDWF$_{33}$ and only when the base relation $\text{OutDegree}$ is relatively large (refer to Figures 12 and 13). However, for base relations with small $\text{OutDegree}$, distributing the processing of the TC queries will not bring any speedup advantage, on the contrary it will slow down the processing of the query (refer to Figure 11).

Comparing Figure 14 with Figure 11 or Figure 15 with Figure 11 shows that an increase in $Q$, or a decrease in $M/N$, brings, in general, an improvement in the performance of some of the parallel algorithms (distributed system) relative to the sequential one.

An important conclusion to be drawn form the performance study presented above, concerning the question "who is better, to process the TC queries on a uniprocessor having a large amount of main-memory or on a number of processors each of which has smaller amount of memory." The answer is complicated and in general depends on the different system's parameters (characteristic of the base relation against which the TC query is running, the query characteristic itself and the hardware characteristics). However, the worst case study presented above shows that with the proper choice of the parallel algorithm and the number of processing nodes, the distributed system will outperform the centralized one over a wide range of the system's parameters.

4. Conclusions

This paper presents and evaluates, using rigorous analytical models, the performance of a number of parallel algorithms suitable for processing an important class of recursive queries, the instantiated TC queries. These algorithms are variants of the sequential $\delta$-wavefront algorithm, designed to run on shared-nothing (message-passing) type of multiprocessor systems. The results obtained form the studies presented in this paper suggest that the relative performance of these algorithms is a strong function of not only the parameters which characterize the system's hardware, but also those parameters which characterize the base relation and the processed query. We have also found that, in general, a TC query can be speeded up by distributing its execution provided that the proper parallel algorithm and the proper number of processing nodes are chosen. Two parallel algorithms have been identified to be the best performing ones, one for systems with slow interconnection networks and the other for systems with fast networks. We have also found
that the available parallelism in a distributed system must be used wisely. Allocating too many nodes to a query which does not need them results in a substantial loss of potential speedup in the processing of the query.

References


Figure 11: PDWF Speedup Comparisons (Const. Memory; \(T_{\text{eff}}=0.8\) msec; \(Q_r=0.001\) and \(N_r=0.1\)).

Figure 12: PDWF Speedup Comparisons (Const. Memory; \(T_{\text{eff}}=0.8\) msec; \(Q_r=0.01\) and \(N_r=1.3\)).

Figure 13: PDWF Speedup Comparisons (Const. Memory; \(T_{\text{eff}}=0.8\) msec; \(Q_r=0.001\) and \(N_r=5\)).

Figure 14: PDWF Speedup Comparisons (Const. Memory; \(T_{\text{eff}}=0.8\) msec; \(Q_r=0.01\) and \(N_r=0.1\)).

Figure 15: PDWF Speedup Comparisons (Const. Memory; \(T_{\text{eff}}=0.8\) msec; \(M_R=1\); \(M_R=0.15\); \(Q_r=0.001\) and \(N_r=0.1\)).