Implementation and performance evaluation of a parallel transitive closure algorithm on PRISMA/DB

Maurice A.W. Houtsma* Annita N. Wilschut Jan Flokstra

University of Twente, Department of Computer Science, P.O. Box 217
7500 AE Enschede, the Netherlands
{houtsma, annita, flokstra@cs.utwente.nl}

Abstract

This paper is one of the first to discuss actual implementation of and experimentation with parallel transitive closure operations on a full-fledged relational database system. It brings two research efforts together; the development of an efficient execution strategy for parallel computation of path problems, called **Disconnection Set Approach,** *and the development and implementation of a parallel, main-memory DBMS, called PRISMA/DB. First, we report on the implementation of the disconnection set approach on PRISMA/DB, showing how the latter's design allowed us to easily extend the functionality of the system. Second, we investigate the disconnection set approach's parallel behavior and performance by means of extensive experimentation.*

It is shown that the parallel implementation of the disconnection set approach yields very good performance characteristics, and that (super)linear speedup w.r.t. a special implementation of semi-naive is achieved for regular, so-called linear fragmentations. We also present a number of experiments that show to what extent data fragmentation issues influence the performance. Finally, we discuss the speedup and benefits to be achieved for arbitrary fragmentations.

*The research of Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

1 Introduction

For years now, attention has been paid to the extension of query languages with (types of) recursion, and the development of effective query optimization strategies [4, 7, 12, 18]. An overview of this area can be found, for instance, in [8]. As the actual computation of recursive queries usually amounts to an iteration over join sequences, recursive queries are computationally very expensive. Therefore, the need for parallel computation is obvious. Parallel computation is a way to reduce the response time to recursive queries, and a lot of research in this area has recently been done, e.g., [10, 27]. In particular, the transitive closure operation has been studied, as an example of a recursive query with great practical value [1, 9, 13, 16, 17, 24]. An extensive survey of parallel execution strategies for transitive closure and logic programs can be found in [5, 6]; [6] also presents an analysis of the effect of initial data distribution on performance.

Although many parallel transitive closure algorithms have been proposed, very few have actually been implemented in a real database management system, and performance analysis via implementation is largely missing. Performance analysis of parallel transitive closures is mainly done using (analytical) simulation [13, 17, 24]. [24] presents a number of hash-based algorithms for the transitive closure operation, and evaluates them using analytical simulation. [17] uses simulation to evaluate a parallel algorithm that combines ideas on traversal recursion [21] and disconnection sets [13] (it is, however, not yet suitable for graphs containing cycles). In all these studies, many simplifying assumptions, such as regularity in the number of tuples produced in each iteration and the fact that data represents a tree structure are made. Consequently, the results of these studies will not reflect actual behavior of systems implementing these strategies. In [1, 23] some actual experiments on transitive closure algorithms are reported. [1] reports on experiments with 8 processors connected by a standard VMEbus; only DAGs are considered in their experiments. [23] reports on a so-called semi-simulation on a single transputer, and some assumptions are made to give an indication of its parallel behavior. In both papers, the graphs that were used in the experiments were small due to limited memory.

In [13], we proposed a parallel transitive closure algorithm, called the disconnection set approach, that is based on semantic data fragmentation. The algorithm was evaluated using simulation, and the results of this study were encouraging. However, for problems as complex as transitive closure, actual implementation is *essential* to really evaluate and understand the behavior of an algorithm. This paper describes the implementation of the disconnection set approach on PRISMA/DB, and the experiments we did (with large relations) to evaluate its parallel behavior. PRISMA/DB is a parallel, main-memory, relational DBMS that was developed in the Netherlands [2, 20]. One of the goals of the PRISMA project was to provide a parallel DBMS that is flexible in its architecture and query execution strategy, so that new functionality can easily be added. This paper shows that implementation of a new transitive closure algorithm and the disconnection set approach were straightforward.

The experiments we describe in this paper first of all intend to show the parallel behavior of the disconnection set approach on PRISMA/DB. We explain this behavior, and show what influences it. It turns out that the results of the parallel approach consistently outperform those of a fast semi-naive algorithm, that the disconnection set approach shows good parallel behavior, and that (super)linear speedup is achieved, even for simple cases. We then investigate what specific characteristics of the fragmentation influence the results. In parallel to the experimentation reported here, we have been working on the design of fragmentation algorithms for the disconnection set approach; this is reported in [15].

The paper is structured as follows. In Sec. 2 we give an overview of the disconnection set approach, followed by an introduction to PRISMA/DB in Sec. 3. In Sec. 4 we describe the implementation of the disconnection set approach on PRISMA/DB. Sec. 5 describes the performance analysis of the disconnection set approach, discusses the parallel behavior obtained, reports on experiments studying several characteristics of the data and its fragmentation that influence the performance, and explains the superlinear speedup obtained. Finally, Sec. 6 summarizes and concludes the paper.

2 Disconnection set approach

In [13] we introduced a parallel strategy for solving all sorts of transitive closure problems (shortest path, bill of materials, etc.) called the *disconnection set approach*. In [14] a formal description of this strategy was given and it was proven correct and complete. We will now give a short introduction to the disconnection set approach, enough for the reader to get an impression of the approach and an idea of its main characteristics. For a complete description and more detailed information, including formal proofs and how to deal with updates, we refer to [13, 14].

The disconnection set approach is a so-called 'semantic approach' to parallel transitive closure computations, just like parallel hierarchical evaluation [16] which was inspired by it. Noticing that communication between processors and computing the same tuples on different processors are the main bottlenecks to efficient parallel computation, and that database problems require coarse-grain parallelism, the disconnection set approach was developed in the following way.

The disconnection set mirrors a very natural way of dealing with transitive closure problems by humans. When one needs to travel by rail from one part of Europe to another, instead of dealing with the extensive European railroad network as a whole, we deal with it country by country.[1] For instance, when seeking a connection from Amsterdam to Stockholm we first find a connection from Amsterdam in Holland to the border with the Germany, then we find a connection from there to the border of Germany and Denmark, then we find a connection from there to the border of Denmark and Sweden, and finally we find a connection from there to Stockholm. Ideally, there is only one chain of countries leading from one city to another; in reality there may be more and the process described above has to be repeated for each such chain.

More formally, given a directed graph $G = \langle V, E\rangle$, we assume an edge-disjunct fragmentation in n fragments $G_1 = \langle V_1, E_1\rangle, \ldots, G_n = \langle V_n, E_n\rangle$ such that $E_i \cap E_j = \emptyset$ $(i \neq j)$. The disconnection sets are formed by the node intersection of the fragments $ds_{ij} = V_i \cap V_j$ $(i \neq j)$. It is required that the disconnection sets

[1]In practice, this has the nice implication that queries about the shortest path of two cities in Holland can be answered by the Dutch railway computer system alone.

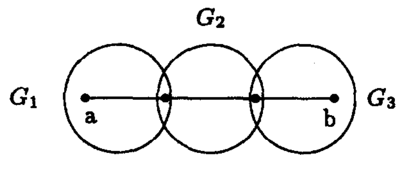


Figure 1: Illustration of disconnection set approach

be pairwise disjunct. Furthermore, a small amount of "complementary information" ($cids_{ij}$) is stored for each disconnection set; it is required to guarantee a correct and complete answer (in case of paths that zigzag in between two fragments) [14]. For instance, for shortest path queries, this complementary information stores the cost of the shortest path for each pair of nodes in the same disconnection set.

For a description of what happens when a path query is processed, consider Fig. 1 and a path query concerning the connections from node a to node b. After determining the fragments containing nodes a and b and the chain of fragments to be followed, the following XRA-commands (XRA is the internal relational interface language of PRISMA/DB) are executed in parallel:

$$\begin{array}{l} \text{tcl}(\{a\},\ G_1 \cup cids_{12}) \ltimes ds_{12} \\ \text{tcl}(ds_{12},\ G_2 \cup cids_{23}) \ltimes ds_{23} \\ \text{tcl}(ds_{23},\ G_3) \ltimes \{b\} \end{array}$$

In this program, $tcl(x, y)$ denotes the transitive closure over relation y, starting from the set of nodes x. The first query finds all paths in the transitive closure of G_1 and its complementary information that start from a, and semijoins the result with the nodes in the disconnection set. The second and third query do likewise. To obtain the final result, the results of the three intermediate queries have to be joined. Note, that we have assumed a so-called linear fragmentation here; as mentioned before, for non-linear fragmentation the strategy works too, but now has to be executed for each chain of fragments connecting the fragments that contain the start and end nodes.[2]

Two obvious characteristics that influence the performance of the disconnection set approach are: size of the disconnection sets and size of the fragments. Small disconnection sets are preferable: they function as additional selections in the computation of the global transitive closure query; from each disconnection set a kind of 'magic cone' is built that restricts the amount of data to be considered. The size of a fragment is an indication of the amount of work to be done for computation of the local transitive closure query. It should be sufficiently large, so a processor has enough work to process it, and the fragments should preferably be balanced in size, so all processors may finish at approximately the same time with the local transitive closure queries.

[2]In case of fragmentation graphs that contain many cycles, the technique described in [16] can be used to reduce possible overhead.

3 PRISMA/DB

PRISMA/DB is a full-fledged parallel, main-memory relational DBMS, designed and implemented from 1986 to 1991 by several Dutch Universities and Philips Research Laboratories. A goal of the PRISMA project was to provide flexibility in architecture and query execution strategy, to enable experiments with the functionality and performance of the system. This flexibility is used here to implement a parallel transitive closure algorithm, and to evaluate its performance.

PRISMA/DB is used for research in various directions, such as analysis of its performance and parallel behavior [26], interoperator pipelining for parallel implementation of multi-join queries [25], and the use of parallelism for integrity constraint enforcement [11]. Here, we explore a fourth direction: parallel recursive query processing. A full description of design, architecture, and implementation of PRISMA/DB can be found in [2], here we give a brief introduction into the hardware and architecture of PRISMA/DB. An overview of the results of the entire project can be found in [3].

3.1 The POOMA machine

PRISMA/DB is implemented on a shared-nothing parallel multi-processor machine called POOMA. A 100-node prototype, located at Philips Research Laboratories, and an 8-node prototype, located in our own laboratory, exist. The work reported in this paper was done on the 8-node prototype. Each node of POOMA consists of a 68020 data processor with 16 Mbytes of memory, a disk, and a communication processor that links it to 4 other nodes using bidirectional links. Some nodes have an ethernet card that links the system to a Unix host.

3.2 The architecture of PRISMA/DB

Figure 2 presents an overview of the architecture of PRISMA/DB. The architecture consists of components that are implemented as communicating processes. Some components are instantiated several

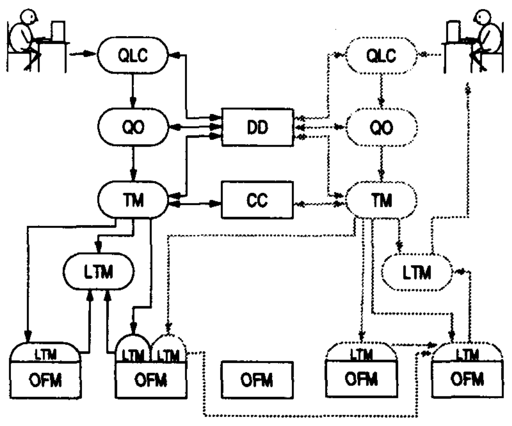


Figure 2: Global architecture of PRISMA/DB

times in the system, others have a single instantiation that serves the entire DBMS. The architecture is flexible; components can be created and deleted dynamically, according to the use of the system. The rectangles in Figure 2 represent permanent components, the ovals represent transient components belonging to a single user session. The dotted ovals show transient components belonging to a second, concurrent user session.

The data dictionary (DD) and concurrency controller (CC) are central components, with their usual function. Standard two-phase locking with shared and exclusive locks is used for concurrency control. Figure 2 shows that these components are used by both user sessions.

The query preprocessing layer of the system is formed by the query language compiler (QLC) and query optimizer (QO). The QLC provides an interactive interface to the user and translates queries from a user language into the internal relational language of the system, called XRA. Translated queries are sent to the QO, which optimizes them into parallel execution plans. The transaction manager (TM) forms the execution control layer of the system, it enforces the ACID transaction properties.

The data storage and query execution layer consists of one-fragment managers (OFMs) and local transactions managers (LTMs). OFMs are permanent; they store and manage a single fragment of a relation.[3] OFMs serve as storage units of the database and can be accessed by all user sessions. LTMs are transient and private to a transaction; they are the relational engines of the system, supporting the relational operators. An LTM can be attached to an OFM, in which case it is allocated to the processor hosting the OFM and can directly access its data. LTMs that are not attached to an OFM, process intermediate results and can be allocated to any processor. Parallelism is achieved by having several LTMs process parts of the data concurrently

[3]Only horizontal fragmentation is currently supported.

An eXtended Relational Algebra (XRA) is used as internal representation of queries. This language consists of the normal relational operations extended with some primitives for grouping and for recursive query processing. Also, the language allows the expression of a wide range of parallel execution plans for a query. Each relational operation can be executed by an arbitrary number of processors, and the result of an operation can be distributed efficiently over an arbitrary number of destinations.

The modification to the system that was done for the research in this paper, was the implementation of a transitive closure algorithm in the LTM, and some minor adjustments in the QLC and QO, to have these components correctly handle the new algorithm. These changes took less than two weeks.

3.3 The execution monitor

PRISMA/DB has a built-in execution monitor that allows detailed analysis of the execution characteristics of a query. The execution monitor enables the writing of cheap log messages during the execution of a query; collection of log data is postponed until after query execution. The execution monitor consists of a data structure on each processor, which is shared by all processes that run on that processor. Processes write simple atomic log messages consisting of local time (the local clocks are synchronized), process identification, and an indication of what the process is doing, such as "start" or "ready".

In the current version of PRISMA/DB, all LTMs log the time at which they initialize and the time at which they are ready. This requires two log messages per LTM, and therefore the execution of a query is hardly influenced by monitoring. After the execution of a query, log data are collected into a file for analysis. The costs of initializing an operation process can be retrieved using initialization time of subsequent LTMs, and costs of local processing can be found from the difference between the "init" and "ready" mark of an LTM.

4 Implementation of the disconnection set approach

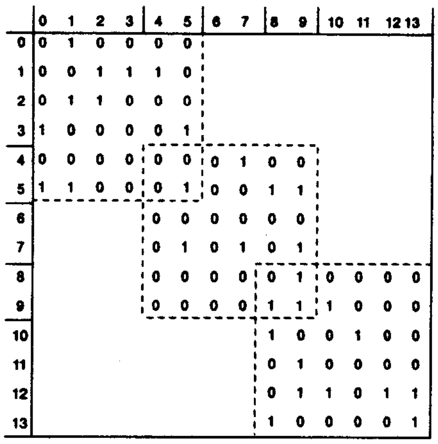
This section describes the implementation of the disconnection set approach on PRISMA/DB. To implement this parallel transitive closure strategy, a suitable central transitive closure algorithm is needed (to compute locally the transitive closure of each fragment), and an execution plan—that uses this transitive closure algorithm together with other primitives that are provided by PRISMA/DB—to generate the answer to a path problem over a fragmented graph. Furthermore, test data is needed for experiments with the implementation of the disconnection set approach. This section describes these issues: first the generation of test data, then the central transitive closure algorithm that we implemented, and finally the execution of path queries over a fragmented graph (using an example query execution).

4.1 Generation of test data

Synthetic test data was used for the experiments. They were provided by a generator that produces a directed graph over a given set of nodes. This graph is already fragmented according to the requirements of the disconnection set approach.[4] Note, that the generator produces fragmented data, instead of generating a graph which is subsequently partitioned over fragments. Fragmentation design of arbitrary graphs is a separate problem; first results of our research in this area are given in [15]. The results of the experiments reported in this paper will be used in further study of fragmentation design algorithms.

As explained in Sec. 2, this paper considers fragmentations with a linear fragmentation graph, so the generator only produces directed graphs with a linear fragmentation. Fig. 3 shows example output of the generator; a directed graph over a set of 14 nodes is represented by an adjacency matrix, with $[i, j]$ set to 1 if node i is connected to node j. For each entry in the matrix that is set to 1, a binary tuple is generated. The tuples are assigned to fragments; the node intersections of the fragments are the disconnection sets. The graph in Fig. 3 consists of three fragments of 6 nodes, with an overlap of 2 nodes between subsequent fragments; each fragment contains 10 connections. The fragment boundaries are indicated in the figure by lines between the node numbers; fragment 1,

[4] As one of the reasons for experimentation is to find out the influence of several characteristics of data fragmentation on performance, we chose to generate test data instead of using an arbitrary graph, so we can control relevant characteristics.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	0	0	0	0								
1	0	0	1	1	1	0								
2	0	1	1	0	0	0								
3	1	0	0	0	0	1								
4	0	0	0	0	0	0	0	1	0	0				
5	1	1	0	0	0	1	0	0	1	1				
6					0	0	0	0	0	0				
7					0	1	0	1	0	1				
8					0	0	0	0	0	1	0	0	0	0
9					0	0	0	0	1	1	1	0	0	0
10									1	0	0	1	0	0
11									0	1	0	0	0	0
12									0	1	1	0	1	1
13									1	0	0	0	0	1

Figure 3: Sample output of the data generator

2, and 3 contain connections between nodes 0–5, 4–9, and 8–13. The disconnection set between the first two fragments contains nodes 4 and 5, the other disconnection set contains nodes 8 and 9. As the disconnection set approach requires an edge-disjunct fragmentation (see Sec. 2), edges in the overlap between two fragments are assigned to an arbitrary fragment. In this case, they are assigned to the left fragment; hence, tuple $(8, 9)$ is assigned to fragment 2.

The generator takes as input parameters: the number of nodes per fragment, the number of connections in each fragment, and the number of nodes per disconnection set. The connections in each fragment are generated randomly. In this way, graphs can be generated that allow investigation of the influence of each of these parameters on the performance of the parallel transitive closure computations. A fragmentation degree of 6 was used throughout this paper; it gives proper insight in the behavior of the parallel algorithm. Moreover, the graphs still fit in main memory; central computation of the queries is thus still feasible.

4.2 Central transitive closure algorithm

Path problems usually concern connections between a set of start nodes and a set of destination nodes, and in the disconnection set approach the computations on the fragments concern connections between two disconnection sets. Therefore, we implemented a central semi-naive transitive closure algorithm that builds spanning trees from a given start set of nodes. We will refer to this algorithm as *selective semi-naive* (SSN). Given a binary relation R, representing the connections in a graph, and a start set S, representing a set of nodes to start from, the algorithm to compute the transitive closure of R from S is:

#nodes in start set	response time
1	2.7
2	5.4
4	8.6
6	15.8
8	21.7
10	27.8
20	52.0
360	3273.6

Table 1: Response time in seconds of SSN for different start sets; base table containing 900 tuples.

```
$tcl := R \ltimes S$
/* to start with the paths that start in S */
$new := tcl$
while $new \neq \emptyset$ do
        $new := (new \bowtie R) - tcl$;
        $tcl := tcl \cup new$
od
return tcl
```

A hash-based implementation was used for this algorithm. The joins are executed as hash-joins, and the result tuples of each *new* ⋈ *R* are inserted into a *tcl* hash-table that checks whether the tuple was already there. If not, the tuple is also inserted into *new.*

The performance gain on a single processor is tremendous; Table 1 relates the execution times to the size of the start set, where the base relation contains 900 connections (represented by tuples) over 360 nodes. The last entry in this table uses the entire relation as start set; it corresponds to the response time of the full transitive closure operation. Not only does SSN reduce the response time by orders of magnitude compared to semi-naive transitive closure computation followed by a selection, it also reduces memory usage significantly. The majority of the experiments reported in this paper would be impossible without this reduction in memory usage.

Recently, other algorithms have also been developed for the problem of transitive closure computations with a start set [19, 22], but these are not very well suited for a relational environment such as PRISMA/DB.

4.3 Implementation of path queries on PRISMA/DB

Once a local transitive closure algorithm was implemented, support of the disconnection set approach was relatively straightforward; disconnection set queries could now be formulated directly in XRA. Fig. 4 shows the query tree that is used for a path query on 6 fragments, and the allocation of operations to processors. Initially, each participating processor hosts 3 fragments: a fragment of the graph ($frag_i$), the disconnection set between the local fragment and the previous fragment ($ds_{i-1,i}$), and the precomputed complementary information over the disconnection set ($cids_{i,i+1}$).[5] The dotted boxes in the figure represent the processor boundaries. Each processor starts computing the transitive closure over its local data; these computations are done independently in parallel. As described in Sec. 2, the complementary information is added to the fragments before computing the local transitive closures. Each transitive closure is computed from a local start set (either the *start* set of the query or a disconnection set). The result of each transitive closure is semi-joined to the local end set (either a disconnection set or the *end* set). During our experimentation it became clear that it is indeed profitable to do these semi-joins, to reduce the size of the operands of the final joins. The five join operations synthesize the final result.

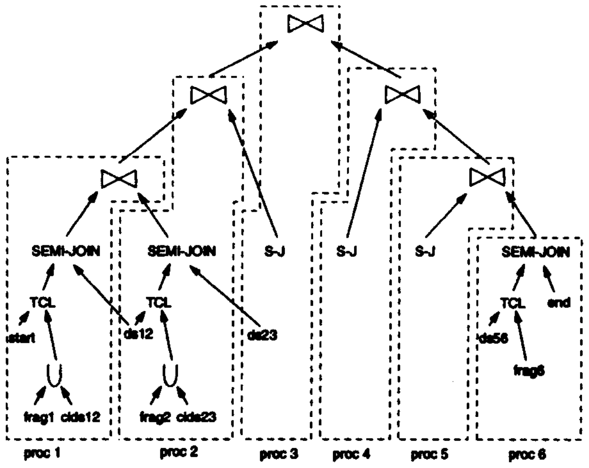


Figure 4: Example query tree for path query (processors 3–5 equivalent to processor 2)

Fig. 5 shows the parallel behavior of an example query; it is generated using the PRISMA/DB execution monitor. Each line in this figure represents a processor, a line starts at the time at which the transitive closure process on the processor is created. A mark on each line indicates when the transitive closure op-

[5]The first and last fragment are treated slightly different; they host a *start* respectively *end* set instead of a disconnection set respectively complementary information

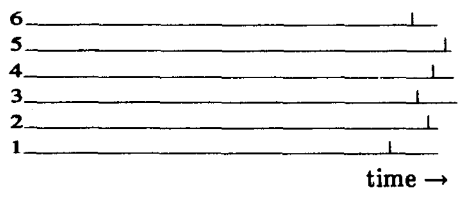


Figure 5: Parallel execution of DSA on PRISMA/DB

eration is finished; the line ends when the processor is ready executing its final joins. The figure shows good parallel behavior, resulting in linear speedup with respect to SSN, as will be shown shortly. It also shows that each processor spends most of its time on the local transitive closure. Finally, the structure of the query tree in Fig. 4 is reflected in the execution monitor diagram. For example, processor 1 joins the results of the transitive closures that are executed on processors 1 and 2; this join operation has to wait for the result of the transitive closure on processor 2 to be entirely available. Processor 6 does not have to wait for any remote operand for a join, it can finish after completing its transitive closure. Processor 3, finally, executes the last join, and therefore this processor is last to finish.

The next section investigates the influence of various parameters on the parallel behavior, and the speedup achieved by using the disconnection set approach.

5 Performance analysis of the disconnection set approach

We now discuss the performance of the disconnection set approach on PRISMA/DB. We study the influence of various parameters (number of nodes, connectivity, size of the disconnection sets) on both response time and speedup with respect to SSN. Recall from Sec. 4.2 that the latter is already much faster than an ordinary semi-naive implementation of transitive closure followed by a selection. The central transitive closure computations were executed on a single node of PRISMA/DB.

5.1 Number of nodes and connectivity of the graph

We first discuss the influence of the number of nodes and connectivity of a graph on the response time and speedup of path queries. The number of nodes in a

	# nodes		
#connections	300	600	1000
low	500	1000	3000
middle	1000	2000	5000
high	1500	3000	10000

Table 2: Number of connections used in the experiments

#nodes	300		600		1000	
algorithm	DSA	SSN	DSA	SSN	DSA	SSN
low	5.8	19.1	16.1	78.4	57.8	333.8
middle	21.6	96.3	39.6	255.5	102.6	708.8
high	32.5	177.6	59.3	395.2	207.2	1550.5

Table 3: Response time in seconds of parallel and central execution of a path query

fragment is varied over 300, 600, and 1000, and a low, middle, and high number of connections is chosen for each number of nodes used. Table 2 relates numbers of connections (i.e. tuples) to numbers of nodes. Data is partitioned over 6 fragments in all experiments, so the entire graph is defined on 1800, 3600, and 6000 nodes. (Our largest experiment thus ran over 6 fragments each containing 10000 tuples.) The disconnection set between two subsequent fragments consists of 6 nodes; the number of nodes in the start and destination sets is chosen equal to the number of nodes in the disconnection set. The path query that is executed here, as everywhere else in this paper, asks for the connections between a given set of start nodes in the first fragment, and a given set of destination nodes in the last fragment.

Table 3 lists the response times of this query, both for the parallel execution of the disconnection set approach (DSA), and for selective semi-naive (SSN). Figure 6 presents a plot of the same results. In the diagram, the response time is plotted against the number of nodes for both algorithms, and for low, middle, and high connectivities.

Response time increases, for both implementations, with an increasing number of nodes, and with increasing connectivity. This, because the total amount of work to be done increases for both approaches. Note that the parallel implementation outperforms the central one in all cases. The diagrams in Figure 6 even show that the response times of the queries on graphs with a low connectivity using SSN, is higher than the response times of the queries on graphs with a high connectivity using the parallel algorithm.

A more detailed comparison between both implementations is shown in Fig. 7. For each query, the

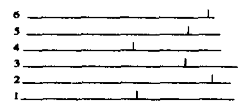
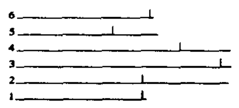
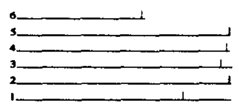
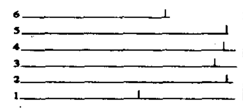
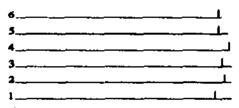
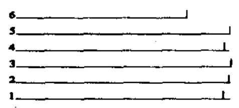
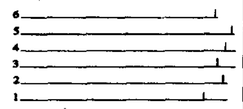
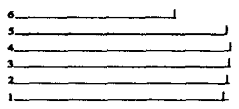
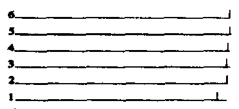
#connections per fragment	#nodes		
	300	600	1000
low	 3.3	 4.9	 5.8
middle	 4.5	 6.4	 6.9
high	 5.5	 6.7	 7.5

Figure 7: Execution characteristics and speedup of the parallel execution of the disconnection set approach.

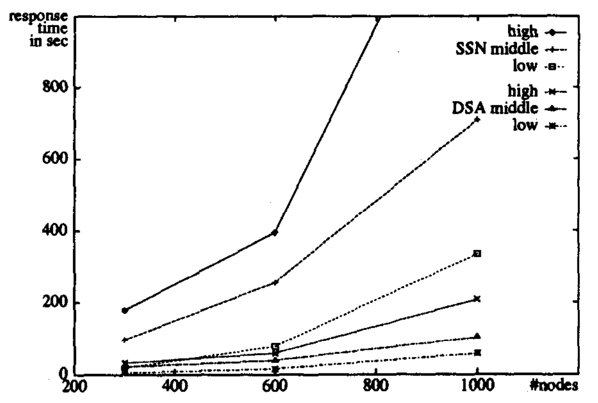


Figure 6: Response time in seconds of parallel and central execution of a path query

characteristics of the parallel execution and speedup with respect to SSN are given. Speedup figures increase from upper left to lower right in the figure. This means that large graphs with a relatively high connectivity yield more speedup than smaller ones and/or a low connectivity. This phenomenon can be explained by the execution characteristics. Computation times of the fragment closures differ considerably for a small number of nodes and for low connectivities. Hence, some processors are idle, waiting for the result of other processors, while others are still busy computing their transitive closure. Therefore, speedup is limited in these cases. For larger problems, however, not only the computation time per fragment increases, but also the variation between the local transitive closure computation times decreases (to be discussed shortly). This causes all processors to be busy all of the time, so that good speedup is achieved. The lower right of Figure 7 reports superlinear speedup.

As noticed, we encountered large variations in the computation times for local transitive closures. This variation is larger when graphs get smaller and connectivity lower. Table 4 shows the sizes (in number of tuples) of the transitive closure from a start set of 6 nodes of 300-node fragments. (The same graphs were used for the experiments in Fig. 7) Each fragment in an experiment is generated at random, following the same procedure. The data in Table 4 shows a large variation in the size of the transitive closures for low connectivities. This variation in the size of the transitive closures is responsible for the less than linear speedup in the top row of Fig. 7. As an aside, we

	#connections		
	500	1000	1500
fragment 1	408	1722	1782
fragment 2	593	1740	1764
fragment 3	818	1695	1776
fragment 4	787	1746	1788
fragment 5	590	1160	1782
fragment 6	815	1492	1794

Table 4: Size of the local transitive closure, using a start set of 6 nodes and 300 node fragments.

	#connections per fragment		
ds size	1000	2000	3000
4	8.9	25.7	38.5
6	16.1	39.7	59.6
9	22.4	62.3	100.5

Table 5: Execution time in seconds, compared for varying disconnection set sizes

may conclude that the number of nodes and the connectivity of a graph are not sufficient to give a good estimation of the size of the transitive closure of a graph (and its computation time).

The experiments reported in this section were also done using disconnection sets of 4 and 9 nodes; the results of those experiments are similar. In the next section, the effect of the size of the disconnection set is studied in isolation.

5.2 Effect of Disconnection Set size

We now discuss the impact of the size of the disconnection sets on the response time of the disconnection set approach. In each experiment, the graph was fragmented over 6 fragments, with 600 nodes per fragment. Three connectivities were used (1000, 2000, and 3000 connections per fragment), and for each connectivity the disconnection set size was varied over 4, 6, and 9 nodes. The size of the start and the end sets was set to 4 tuples for each experiment. Table 5 shows the response time of the parallel implementation of the disconnection set approach for various disconnection set sizes. Fig. 8 presents a plot of the same results.

The results show that the size of the disconnection sets is of major importance for the performance of the disconnection set approach. A small increase in the disconnection set size causes a significantly larger response time. The reason for this increase in response time is twofold: Firstly, the computation time of a

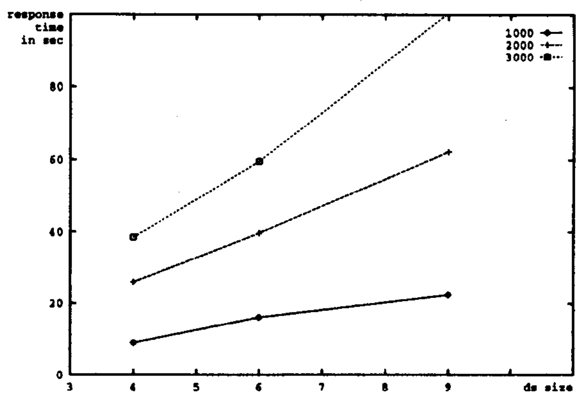


Figure 8: Execution time in seconds, compared for varying disconnection set sizes

local transitive closure increases with increasing disconnection set size, as the disconnection set is used as start set in the transitive closure computation. As shown in Sec. 4.2, the computation time of a transitive closure algorithm is very sensitive to the size of the start set. Secondly, the operands of the final join operations are larger for larger disconnection sets, and therefore the final joining phase takes longer. A study of the log data (equivalent to the ones shown in Fig. 7) revealed that the increase in computation time for the transitive closure is the main reason for the increase in response time.

Another conclusion that can be made concerns the fragmentation strategy that should be used for the disconnection set approach. A fragmentation strategy may have to make the choice between generating many small fragments with large disconnection sets, or fewer large fragments with small disconnection sets. Our results show that neither choice is always the right one. For instance, in Table 5 the queries on the 1000 connections per fragment graphs outperform the other queries, but the query on the 3000 connections graph with small disconnection sets performs better than the query on the 2000 connections graph with larger disconnection sets. From these observations it can be concluded that both the fragment size and the size of the disconnection sets is of major importance to the performance of the disconnection set approach.

5.3 Speedup

Most results show an almost linear to superlinear speedup, we will now discuss the main reasons for this. In Sect. 5.1, we already explained why for small fragments speedup may be slightly smaller than linear. We will also argue that in many cases the disconnection set approach will achieve a linear to superlinear speedup with fewer processors than in our experiments.

Superlinear speedup in the experiments with large relations occurs because costs of the central transitive closure algorithm increase more than linear with the size of its operand. Both in the central and in the parallel case, each transitive closure is locally computed using SSN; the former with the entire relation as operand and the latter with fragments as operands. The cost of SSN increases more than linear with the size of the relation it computes the transitive closure of, because of the combined effect of *i)* a larger operand used in successive join operations, which already yields a linear growth in the costs, and *ii)* a bigger number of iterations, because longer paths in the graph have to be calculated. The last argument, of course, only holds when the initial adjacency matrix is not too sparse or too dense, so that long paths actually exist in the transitive closure. This was the case in our experiments. The superlinear increase in the cost of SSN, together with the unskewed parallel execution for the larger experiments as shown in Fig. 7, explains the superlinear speedup achieved.

Our experiments, with a linear chain of fragments, did not take full advantage of the disconnection set approach, which allows to ignore non-relevant fragments. Suppose that we move the start set from fragment 1 to fragment 4. The central algorithm still has to consider the entire relation (i.e. fragments 1–6), whereas the disconnection set approach only considers fragments 4–6 and still gives a correct and complete answer [14]. This means that we will need only half of the processors and consequently the same speedup is realized using a smaller number of processors! The ability to ignore non-relevant fragments is a very important aspect of the disconnection set approach, and it will occur a lot in practical applications of the disconnection set approach. Hence, the disconnection set approach will in many cases be able to achieve good linear speedup using fewer processors than in our experiments.

Of course, there is some overhead in the disconnection set approach to find all applicable routes through the fragmentation graph. However, the fragmentation graph is orders of magnitude smaller than the graph itself (our largest graph contained 6000 nodes and 60000 tuples, with the fragmentation graph containing 6 nodes and 5 tuples). Special algorithms for graph traversal in main-memory can be used for finding the routes in the fragmentation graph, so any noticeable overhead seems unlikely. If the fragmentation graph contains cycles, overlapping paths (common subexpressions) can be detected so they are computed only once. If the fragmentation graph happens to be complex and contains many cycles, a generalization of the disconnection set approach called *Parallel Hierarchical Evaluation* [16] can be used for efficient parallel computation.

6 Conclusions

In this paper, we combined two of our research efforts: the disconnection set approach and PRISMA/DB. The implementation of the disconnection set approach for transitive closure computations on PRISMA/DB allowed a detailed performance evaluation of the parallel transitive closure algorithm. It was shown how we could benefit from the flexibility provided by PRISMA/DB, allowing changes to the system to be made relatively easy. Also, the execution monitoring facilities of PRISMA/DB were useful to analyze the results of our experiments.

The results show that the disconnection set approach provides a good parallel behavior of the computation. Indeed, the philosophy behind this strategy—independent computation on only part of the data by having some minimal complementary information—is clearly paying off. The results also show that, compared to selective semi-naive (already orders of magnitude faster than semi-naive transitive closure computation followed by a selection), the disconnection set approach performs very well, yielding (super)linear speedup. The results of the performance evaluation show that large fragments yield better speedup than small ones, due to the large variation in computation time for the local transitive closures in case of small fragments. Also, it was shown that the disconnection set performs best with small disconnections sets. It may even pay to choose for a smaller number of larger fragments, if the size of the disconnection sets is reduced in return.

We explained the superlinear speedup obtained, and argued that such speedup may be reached with considerably fewer processors in many practical applications.

An interesting side result of the experiments is that we showed the large variation in size of the transitive closure, and its relationship to the degree of connectivity of a graph. This issue deserves further research.

We are currently building a tool to automate the process of experimentation (computing complementary information, fragment graph, allocating fragments, etc.). Moreover, in parallel to the experimentation reported here, we have studied the issue of fragmentation design [15]. As a next step, we will now combine these efforts, and are going to undertake large-scale experiments (on the 100-node machine) with the fragmentation design algorithms and the experimentation tool. As an aside, we are also considering experiments to investigate if it would be possible to come up with a cost estimation of transitive closure, given the connectivity of a graph.

Acknowledgements

We thank Peter Apers for encouraging our research and commenting on earlier drafts of this manuscript.

References

[1] AGRAWAL, R. AND JAGADISH, H.V. "Multiprocessor transitive closure algorithms," in *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, Texas, Dec. 5–7 1988, pp. 56–66.

[2] APERS, P.M.G., VAN DEN BERG, C.A., FLOKSTRA, J, GREFEN, P.W.P.J, KERSTEN, M.L. AND WILSCHUT, A.N. "PRISMA/DB: A Parallel Main-Memory Relational DBMS," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, Dec. 1992, pp. 541–554.

[3] AMERICA, P. (Ed.), *Parallel Database Systems, Proc. of the PRISMA Workshop*, LNCS 503, Springer-Verlag, 1991.

[4] BANCILHON F., D. MAIER, Y. SAGIV AND J.D. ULLMAN "Magic sets and other strange ways to implement logic programs", *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, Cambridge (MA), March 1986.

[5] CACACE, F., CERI, S., AND HOUTSMA, M.A.W. "An overview of parallel strategies for transitive closure on algebraic machines," in [3].

[6] CACACE, F., CERI, S., AND HOUTSMA, M.A.W. "A survey of parallel execution strategies for transitive closure and logic programs," in *Distributed and Parallel Databases*, Vol. 1, No. 4, Oct. 1993.

[7] CERI S., G. GOTTLOB AND L. LAVAZZA "Translation and optimization of logic queries: the algebraic approach", *in Proc. of the 12th Int. Conf. on Very Large Data Bases*, Kyoto, pp. 395-403, August 1986.

[8] CERI S., G. GOTTLOB AND L. TANCA *Logic Programming and Databases*, Springer-Verlag, 1990.

[9] CHEINEY J.P. AND C. DE MAINDREVILLE "A parallel strategy for transitive closure using double hash-based clustering," *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Aug. 1990.

[10] GANGULY S., A. SILBERSCHATZ AND S. TSUR "A framework for the parallel processing of Datalog queries," *Proc. SIGMOD 90.*

[11] GREFEN, P.W.P.J. AND APERS, P.M.G. "Parallel Handling of Integrity Constraints on Fragmented Relations," *Proc. of the Second Int. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 2-4, 1990.

[12] HOUTSMA, M.A.W. AND APERS, P.M.G. "Algebraic optimization of recursive queries," *Data and Knowledge Engineering*, Vol. 7, No. 4, March 1992.

[13] HOUTSMA, M.A.W., APERS, P.M.G., CERI, S. "Distributed computation of transitive closure queries: The disconnection set approach," *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990.

[14] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Complex transitive closure queries on a fragmented graph," *Proc. 3rd Int. Conf. on Database Theory*, December 1990, Paris, France, Lecture Notes in Computer Science No. 470, Springer.

[15] HOUTSMA, M.A.W., APERS, P.M.G., AND SCHIPPER. G.L.V. "Data fragmentation for parallel transitive closure strategies," in *Proc. 9th Int. Conf. on Data Engineering*, Vienna, April 1993, pp. 447–456.

[16] HOUTSMA, M.A.W., CACACE, F., AND CERI, S. "Parallel hierarchical computation of transitive closure queries," in *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, Dec. 1991, Miami Beach, Fl.

[17] Hua, K.A. and Hannenhali, S.S. "Parallel transitive closure computations using topological sort," in *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, Dec. 1991, Miami Beach, Fl.

[18] Ioannidis Y.E. "On the computation of the transitive closure of relational operators," *Proc. Int. Conf. Very Large Data Bases*, Kyoto 1986, pp. 403–411.

[19] B. Jiang, "A suitable algorithm for computing partial transitive closures in databases, in *Proc. 6th Int. Conf. on Data Engineering*, Los Angeles, 1990, pp. 264-271.

[20] Kersten, M.L., Apers, P.M.G., Houtsma, M.A.W., van Kuijk, H.J.A., and van de Weg, R.L.W. "A distributed, main-memory database machine," *Proc. of the 5th Int. Workshop on Database Machines*, Karuizawa, Japan, October, 1987; also appeared in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka (eds.), Kluwer Academic Publishers, 1988.

[21] Larson, P.-A. and Deshpande, V. "A file structure supporting traversal recursion," in *Proc. ACM-SIGMOD*, 1989, pp. 243-252.

[22] Toroslu, T. and Qadah, G.Z. "The efficient computation of strong partial transitive closure," in *Proc. 9th Int. Conf. on Data Engineering*, Vienna, April 1993, pp. 530–537.

[23] Shao, J., Bell, D.A., and Hull, M.E.C. "An experimental performance study of a pipelined recursive query processing strategy," in *Proc. of the Second Int. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 2-4, 1990, pp. 30–43.

[24] Valduriez P. and Khoshafian, S. "Parallel Evaluation of the Transitive Closure of a Database Relation," Int. Journal of Parallel Programming, 17:1, Feb. 1988.

[25] Wilschut, A.N. and Apers, P.M.G., "Dataflow query execution in a parallel main-memory environment,"*Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, Dec. 1991, Miami Beach, Fl.

[26] Wilschut, A.N., Flokstra, J., and Apers, P.M.G. "Parallelism in a main-memory DBMS: The performance of PRISMA/DB," in *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver, Canada, Aug. 1992.

[27] Wolfson O. and Ozeri, A., "A new Paradigm for Parallel and Distributed Rule-processing", in *Proc. ACM-SIGMOD 1990*, pp. 133-142.