Recursive Query Processing on a Connection Machine

Th. Zurek
Department of Computer Science
Edinburgh University
Email: tz@dcs.ed.ac.uk

E. M. Minty
Edinburgh Parallel Computing Centre
Edinburgh University
Email: elspeth@epcc.ed.ac.uk

P. Thanisch
Department of Computer Science
Edinburgh University
Email: pt@dcs.ed.ac.uk

Abstract

In this paper we give an example of recursive query processing on an SIMD hardware architecture. On one hand the class of linear recursive queries is more general than the class of queries covered by traditional query languages and on the other hand parallelism provides several new possibilities for efficient query processing. This fits well into the arising needs caused by massively growing amounts of data and more expressive queries to be processed in reasonable amounts of time.

The paper presents a model that uncovers a variety of parallel strategies for query processing. We concentrate on two significantly different strategies: the bottom-up and the transitive closure strategy. The join and the transitive closure work out to be the most important operations according to efficiently processing linear recursive queries. Therefore data parallel algorithms for these operations are presented and their respective implementations are analysed. Finally performance results for the two processing strategies are given for two graph structures. The performance results show that neither of the strategies proves to be the best in every case but that a query optimizer can choose the best one in each single case by considering data and query parameters.

1 Introduction

1.1 Parallel Databases

Megapundits in high-performance database technology have been amazingly wrong in their predictions. Ten years ago, the paper *Database Machines: An Idea Whose Time Has Passed ?* [BD83] wrote the obituary for parallel database systems. Today, of course, huge profits are made by companies producing database machines and these machines are used to good effect in many branches of industry, commerce and government. If we look back just four years, the megapundits made another prediction, namely that the only hardware architecture suitable for database machines is *shared-nothing* architecture, i.e. an architecture in which each processor has its own memory and secondary storage access and in which the processors communicate via message passing. There is also a presumption that the processors execute asynchronously. We shall refer to a parallel computer that conforms to this architectural description as a *multicomputer*. In this chapter, we would like to dispute the more recent position taken by the megapundits.

Let us focus our attention on the relational data model which allows us to think of the data as if it were organized in tables and which provides a set of simple operations to be performed on these tables. The most computationally expensive of these operations is the *join*, which combines information from two tables to create a new table, which may, in turn, be joined with other another table.

DeWitt and Gray [DG90] view the problem as being the efficient, parallel implementation of the join operator. For them, the key computational problem is to implement the join operator so that it executes efficiently once.

We are interested in a rather different class of queries, namely the *linear recursive* queries, which implies that we need to *repeat* an operation, such as a join, some *data-dependent* number of times. For this class of queries, a richer repertoire of operators, such as transitive closure, are potentially useful. We represent parallel query processing strategies at a level of abstraction from the hardware and software implementation that facilitates efficient implementation on a variety of parallel platforms, not just shared-nothing architectures.

In order to prove our point, we use a target architecture that is radically different from that envisaged by DeWitt and Gray, namely Thinking Machines Corporation's Connection Machine CM-200. In the CM-200, operations are carried out on all the data items synchronously. This results in a single control flow on highly parallelized data. This approach fits well with the concepts of data parallelism, in particular because most of the relational algebra operators allow the data to be split into a large number of portions which can be operated on independently.

The communication effort to be made by the programmer is to describe (mostly regular) patterns of communication to distribute the data amongst the large number of processors. This allows communication calls to be optimised by the compiler and/or performed by highly optimised communication functions provided through software libraries, the hardware itself or the operating system. This, combined with the ability of the CM-200 to perform array operations on large amounts of data at high speed, makes it a platform that is worth considering for problems of database processing.

We concentrate on a particular problem in query processing, namely finding a technique that can produce answers efficiently for the class of *linear recursive* queries. This class of queries is not as general as the class of recursive queries studied by the deductive database community, but it is more general than the class of queries covered by the traditional query languages as SQL, QBE or Quel.

Relational databases only became a practical proposition when researchers developed sophisticated query optimisation techniques. These analyse a user's query, expressed in a high-level declarative query language such as SQL, and, using statistical knowledge about the data in the database, choose a sequence of operations that can produce an answer to the query 'efficiently', i.e. using a reasonable amount of processor time and disk operations.

With the added computational difficulty of recursive queries, the need for good query optimization techniques is even greater. In this paper, we describe the work that must be done in order to build an optimizer that can exploit the techniques that we have developed for query processing.

1.2 Overview

In section 1.3 we will give a short introduction to the Connection Machine architecture and the underlying paradigma of data parallelism.

In section 2 a query processing model is presented. It is based on the notion of tuple generations as produced by an ordinary bottom-up evaluation of a linear recursive query. These generations will be organized in a matrix. This matrix model provides a kind of gameboard for query processing: several 'moves' (i.e. database operations) are allowed to move over the board (i.e. to process the query) and we can think of several sequential and parallel strategies of combining the moves to 'run' over the board.

Section 3 discusses the optimizing aspects of these strategies and identifies several data and query characteristics that influence the performance of query processing. One result of this analysis is that performance heavily depends on the performance of the join and transitive closure operations that are applied for processing the query. Therefore section 4 presents a parallel join and a parallel transitive closure algorithm and discusses the respective performance issues.

Section 5 gives some performance results for processing linear recursive queries on DAG- and tree-structured database relations. The results are summarized in section 6.

1.3 The Connection Machine CM-200

The CM we used is an SIMD computer with 16384 processors connected by a 10-dimensional hypercube network. Each processor has a local memory of 256 kBits so the CM has 512 MBytes total memory. In one step each active processor executes the same instruction but on different data (i.e. SIMD) whereas inactive processors do not participate. The successful use of the CM requires an algorithm that can exploit this data parallelism. The large number of processors, the underlying communication network and the parallel mass storage system – the Data Vault – make it suitable for processing large amounts of data.

The Data Vault has 10 GBytes of memory and a possible data transferring rate of 25 MBytes per second. It employs an array of 42 disk drives. Of these 42 drives, 32 hold data. Reading from and writing to these 32 drives means reading and writing on 32 drives in parallel. Having a fast secondary storage system is really important as the sizes of databases are growing with increasingly ambitious database modelling.

Data parallel computing associates one processor with each data element. Therefore parallel data structures are spread across the processors, with a single element stored in each processor's memory. When parallel data structures contain more elements than the system has processors, the system operates in virtual processor mode, presenting the user with a large number of processors with a correspondingly smaller memory and performance. Each physical processor is made to simulate the appropriate number of virtual processors [Cor91].

2 Query Processing

We will now describe a model for processing linear recursive queries. This model is based on the notion of tuple *generations* as they are produced by a naive bottom-up evaluation of a linear recursive query. This is described in section 2.1. Section 2.2 introduces the *generation matrix model* – a query processing model based on tuple generations. In the general query optimizer architecture given by [IW91][1] this model can be related to the ordering stage although it also covers also aspects of the planning stage.

2.1 Tuple Generations

In this paper we will deal with linear recursive queries that have the following form[2]:

$$\begin{aligned} P(X_1, X_2, Z) &= E(X_1, X_2, Z). & (1) \\ P(X_1, X_2, Z) &= P(X_1, X_2, Y) \bowtie \\ & \quad Q(X_1, Y, Z). & (2) \end{aligned}$$

(1) is called the *exit-rule* or *exit-equation* and (2) the *linear recursive rule / equation* of the query. The term *linear* refers to the fact that the predicate symbol P of the left side of the equation appears only once on the right side. [JAN87] and [ZT93] show that every linear recursive query containing only one linear recursive rule can be transformed to an equivalent query of the form shown above so we can assume this general form.

The naive way of evaluating a linear recursive query is bottom-up, i.e. starting with the initial relation given by the exit equation further tuples are added by applying the linear recursive equation until no more new tuples are added. We will demonstrate this by an example.

Example

Let us consider a query with (1) as the exit and (2) as the linear recursive rule.

For the purpose of a bottom-up evaluation the rules (1) and (2) are used in the following way:

1. The initial tuples are given by the exit rule (1). These tuples are said to be of *generation* P_0 so

$$P_0(X_1, X_2, Z) = E(X_1, X_2, Z).$$

2. Further tuples are added by sequential applications of equation (2) to produce the generations P_1, P_2, ... :

$$\begin{aligned} P_{i+1}(X_1, X_2, Z) &= P_i(X_1, X_2, Y) \bowtie \\ & \quad Q(X_1, Y, Z). \end{aligned}$$

[1] This architecture consists mainly of three stages: *rewriting* (the query) – *ordering* (the operations to evaluate the query) – *planning* (the execution from a structural and physical point of view).

[2] These are *Datalog equations* [Ull88]. Throughout this paper we rather prefer describing the queries by using relational algebra operators to make the reader aware of the operations that are involved in query processing although the resulting expressions are in 'Datalog style', i.e. we use Datalog-like notations like $P(X, Y, Z)$ and refer to attributes in the relation P by variables X, Y, Z. This also simplifies the notation of a join as the join conditions can be easily derived by matching the variables. Therefore we just notate a join by $\bowtie$ without giving the conditions explicitly. This notation is a hybrid of relational algebra and logic programming notation.
Using relational algebra expressions suggests that the techniques proposed in this paper can be applied to a wide range of query languages and not only to Datalog.

$$P_3(X_1, X_2, Z) = P_2(X_1, X_2, Z) \bowtie Q(X_1, Y, Z)$$

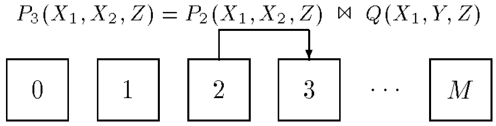

Figure 1: Graphical representation of a bottom-up evaluation producing a chain of tuple-generations.

Therefore a bottom-up evaluation produces the generations P_i of tuples in a sequential order starting with the initial relation E as P_0 and joining Q in each step.

Finally the resulting relation P of the query is obtained by

$$P = \bigcup_{i \geq 0} P_i = \bigcup_{i=0}^{M} P_i$$

such that

$$P_j \subseteq \bigcup_{i=0}^{M} P_i \quad \text{for all } j > M \tag{3}$$

which means that all P_j with $j > M$ do not contribute any new tuple to P. The existence of such an M is guaranteed [Ull88] [Ull89].

Normally we choose M to be the smallest number such that (3) holds.

The notion of representing the resulting relation as a set of generations of tuples is quite useful. Note that a generation is a set of tuples that were generated in the same way and generations do not have to be disjoint as some tuples might be derived in several ways. The union of all generations represents the result of the query. In figures in this paper, the set of tuples in a generation shall be represented graphically as a square.

A bottom-up evaluation can then be considered as producing a chain of generations, up to generation M, as in figure 1 where we get generation P_{i+1} by applying the linear recursive rule (or equation) to generation P_i.

We note that a bottom-up evaluation requires M join operations where M depends on the characteristics of the underlying data (relations).

2.2 Generation Matrix

[JAN87] presented an alternative technique to process linear recursive queries consisting of an exit rule like (1) and a linear recursive rule (2). This technique computes the transitive closure of relation Q or a relation derived from Q. An important numerical parameter, d, for this technique is obtained from a graph theoretic analysis of the query. We do not want to go into the details of this technique. The reader might refer to [JAN87] or [ZT93].

The important thing about this technique is that it provides a possibility of computing tuple generations other than applying the linear recursive rule (2) to generation P_i to compute generation P_{i+1}. For comparing the bottom-up and the transitive-closure technique we will reorganize the generations in form of a *generation matrix*.

In the last section we introduced tuple generations as subsets of the final result. In figure 1 they are represented as a square in the order in which they are produced by a naive bottom-up evaluation. For our purposes they are reordered and organized in a matrix with generation P_0 as the top left element. This matrix has d columns and M/d rows where

- d is a data-independent parameter given by the query and
- M is the number defined in (3); M depends on the characteristics of the actual database relations involved in the query and is therefore a data-dependent parameter.

Figure 2 shows the generations P_i that form the result P of a linear recursive query. Basically computing the result of a linear recursive query means computing the generations of its underlying generation matrix. We will now show graphically in which way these generations can be computed: we start in the top left corner as the generation P_0 (square 0) is given by the exit rule (1). Our goal is to proceed to every square (i.e. to compute every generation) in the matrix. Various operations are allowed that correspond to moves from one square to another:

(R) moving to the next generation to the right (i.e. computing P_{i+1} from P_i) means applying the linear recursive rule r to P_i (see figure 3),

(R2) moving two generations to the right (i.e. computing P_{i+2} from P_i) means applying the linear recursive rule r twice or applying the 2-step rule[3], r^2, once to P_i (see figure 4),

[3] An x-step rule, r^x, is a rule that has the same effect as applying a linear recursive rule, r, x times. The 2-step and the d-step rules of (2) are shown in table 1.

move	corresponding operation
(R)	$P_{i+1} = P_i \bowtie Q$
(R2)	$P_{i+2} = P_i \bowtie Q \bowtie Q$
(Rd)	$P_{i+d} = P_i \bowtie \underbrace{Q \bowtie \cdots \bowtie Q}_{Q^d}$
(TC)	$Q^+_{a_j}$ for $j = 1, \ldots, T$

$Q_{a_j} = \sigma_{X=a_j}(Q(X,Y,Z))$ and $\{a_1, \ldots, a_T\}$ is the set of values of the X-attribute in $Q(X,Y,Z)$.

Table 1: Correspondence to operations

(Rd) moving to the generation below (i.e. computing P_{i+d} from P_i) means applying the linear recursive d-step rule to P_i (see figure 5),

(TC) covering, or moving to, all generations of a column means applying the transitive closure technique using the d-step rule r^d. This move can only be executed if we have already reached or passed P_g where g is – like d – another characteristic of the linear recursive rule. (see figure 6).

Finally we can apply (R), (R2) and (Rd) simultaneously to a group of generations (squares) to reach their respective neighbours (see figure 7), i.e. applying a join to several generations, e.g.

$$P_{k+1} \cup P_{l+1} = (P_k \cup P_l) \bowtie Q$$

A summary of the moves is given in table 1; the costs of the moves are shown in table 2.

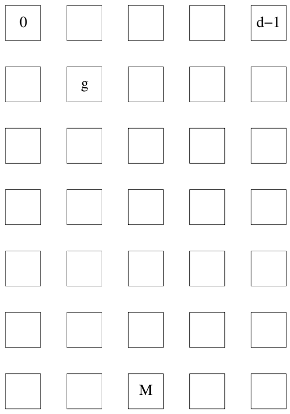

Figure 2: Generation matrix

Obviously we could also generate moves (R3) or (R4) to move three or four generations to the right by deriving the 3-step rule and the 4-step rule.

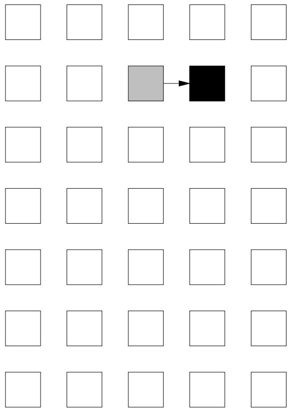

Figure 3: Moving one square to the right by applying the linear recursive rule r

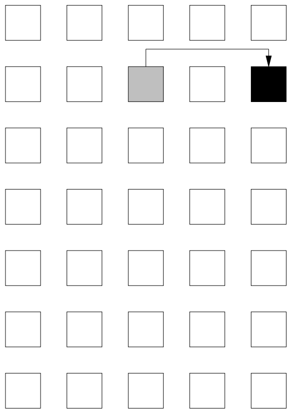

Figure 4: Moving two squares to the right by applying the 2-step rule r^2

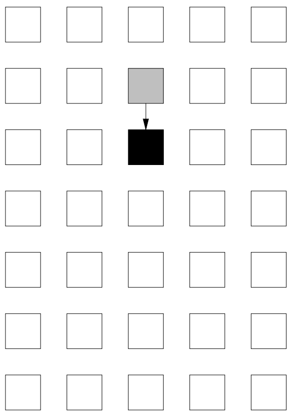

Figure 5: Moving one square down by applying the d-step rule r^d

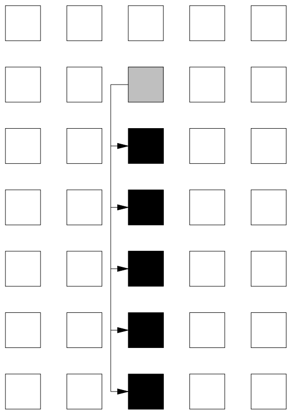

Figure 6: Covering the rest of a column by applying the transitive closure technique

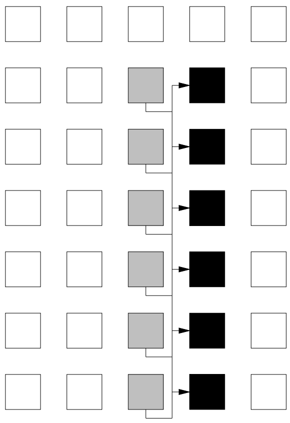

Figure 7: Simultaneous right move

move	costs
(R)	1 join for each move
(R2)	1 join for each move 1 join for building $Q \bowtie Q$ (once)
(Rd)	1 join for each move $(d-1)$ joins for building Q^d (once)
(TC)	T selection operations to build the Q_{a_j}'s T transitive closures

Table 2: Costs of the moves

These (Rx) moves do not appear to be of any practical use as they 'jump' over one or more squares that have to be visited anyway. But we should note the general possibility of creating and applying such moves. We will mainly apply the (R) and the (TC) move.

The generation matrix model is one amongst various formalisms for representing parallel computations. The formalism adopted depends, in part, on the way in which the parallel algorithm was designed. For example a parallel algorithm could be represented by an acyclic directed graph (DAG) in which the nodes correspond to sequential tasks and the arcs correspond to a precedence relation. This approach is mainly used for task-farming parallelism. Alternatively, the computation could be represented by a dataflow diagram.

For our present purposes, we use a special-purpose graphical representation where the 'nodes' correspond to intermediate results that are all subrelations of the result relation. There is a partial order between the nodes in the sense that

- one subrelation may only be generated using our repertoire of simple operations once certain other subrelations have been generated, and
- if two subrelations are incomparable under the partial order, then they may be generated by concurrent, independent operations.

Several strategies/routes of 'running through the matrix' can be considered. The strategy described by [JAN87] is the following (see also figure 8):

(a) Starting at square 0 perform right-moves (R) until you reach square g.

(b) Move down the column of square g – this corresponds to a (TC) move.

(c) Perform $(d-1)$ concurrent right moves (R) from all squares of the column in which square g is located to get to the remaining squares. – this corresponds to a simultaneous right-move as shown in figure 7.

Alternatively the naive bottom-up strategy is shown in figure 9.

Table 2 shows that the costs of a move are not constant and depend on query-dependent parameters like d and also on data-dependent parameters like T[4] or on the size of the relations involved in

[4] T is the number of values in the X_1 attribute of relation Q of the linear recursive rule (2).

a join operation. Therefore predicting the costs of a strategy is not straightforward although it would be nice if the costs of a strategy could be predicted so that an optimizer could choose the most effective strategy for each individual query.

The performance of a strategy to move through the matrix depends on several issues:

- on the value of d, i.e. the number of columns of the matrix,
- on the value of g as we are forced to perform only right-moves (R) up to square g,
- on the efficiency of a right-move (R), i.e. essentially the efficiency of the underlying join operations as one right-move is in fact executing one join operation,
- on the efficiency of the (TC) move that is dictated by
  - the number T of transitive closures for computing the generations of the column in which square g is located,
  - the efficiency of the underlying transitive closure operation.

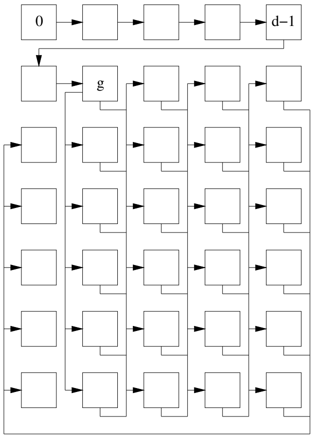

Figure 8: Transitive closure strategy of moving through the generation matrix

Basically the strategy of figure 8 implies

- $d-1$ joins to compute Q^d for the d-step rule r^d,
- g joins to move to generation g,
- T transitive closures to move down the column of g and
- $d-1$ joins move simultaneously to the remaining generations.

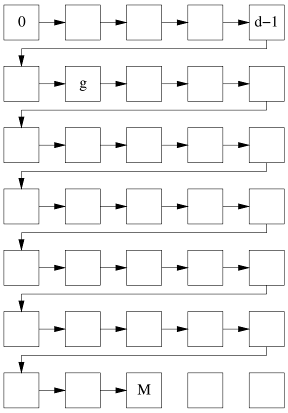

Figure 9: Bottom-up evaluation of a query

The $g + 2\,(d-1)$ joins and T transitive closures have to be compared with the M joins that are necessary to perform a bottom-up evaluation (see section 2.1). Comparison must also be made on the size of the relations that are involved in the joins and the transitive closures. Apart from that the efficiency of the transitive closure technique depends also on the length L of the longest path of the underlying relation. One can expect that the transitive closure technique cannot improve the performance to a large extent or may be even slower if L is small which suggests that M might be also small so the number of joins to do a bottom-up evaluation would be also small.

2.3 Parallel Aspects

The generation matrix model of section 2.2 provides a framework for analysing several strategies to process a linear recursive query. The moves (R) and (TC) may be combined in several ways for query evaluation. Considerations of sequential strategies dwell upon whether to apply the (TC) move or to go rather for an ordinary bottom-up evaluation, i.e. applying (R) moves. This is a crucial point and we can open the ground for a wide range of new processing strategies by involving parallelism. Parallel issues in the evaluation process are:

Data parallel moves: Figure 7 gives already an example in which an (R) move can be applied to several portions of data (e.g. several generations) in parallel. This is called data parallelism as the same instructions are executed on different items of data. In this case data parallelism is possible because there are no constraints in between the portions of data. E.g. generations do not have to be disjoint and can be built independently from one another. This means that (R) moves – and the same applies to (TC) moves – can be performed simultaneously on several generations, e.g. as shown by figure 7.

Different moves in parallel: The (R) and (TC) moves do not interfere with each other so (R) and (TC) moves can be executed in parallel even on the same portion of data, i.e. (R) or (TC) can be applied whenever possible and regardless of whether there is already another activity going on.

Parallelising the (R) move: Clearly the (R) move can be parallelised internally. As we have already seen it consists mainly of a join operation[5]. Therefore efforts should be mainly concentrated on implementing one or more appropriate[6] parallel join algorithms.

Parallelising the (TC) move: The same ideas that have applied to the (R) move can be used for the (TC) move. The dominant operation in this case is clearly the transitive closure. There are several well investigated parallel transitive closure algorithms based on boolean matrix multiplications [Akl89], the Warshall constraints [ADJ90] or hash-join techniques [ZZO93].

3 Query Optimizing Aspects

Usually a database management system (DBMS) incorporates a query optimizing module that decides on the best way to process the query. Given the generation matrix model and the moves and the parallel issues of sections 2.2 and 2.3, a variety of strategies to move through the generation matrix, i.e. of processing a query, can be considered. Naturally it would be nice to find out a way to determine the optimal, i.e. the fastest or cheapest strategy.

We will now analyze the costs $c(S)$ of a strategy S. Therefore the following notations will be used:

- A strategy S is a list of stages s_i

$$S = [s_1, s_2, \ldots, s_n]$$

- A stage s_i is either
  - sequential, i.e. only one single move is executed in the stage, or
  - parallel, i.e. several moves are executed in parallel.

  We will denote a stage as a sequence of moves

$$s_i = (m_1, m_2, \ldots, m_{k_i})$$

  where $k_i = 1$ for a sequential s_i. For $k_i > 1$ all the m_j may be executed simultaneously, i.e. in parallel.
  We will refer to a move m_j of a stage s_i by

$$s_i.m_j$$

- We can consider the m_j to be elements of a set of move types e.g.

$$m_j \in \{\text{(R)},\text{(R2)},\text{(Rd)},\text{(TC)}\}$$

  although this does not link the moves with the data on which the move is executed and which is also an important performance issue. But this keeps the notation as simple as possible for the moment.

We can now describe the costs $c(S)$ by

$$c(S) = \sum_{i=1}^{n} c(s_i) = \sum_{i=1}^{n} c(s_i.m_1) \qquad (4)$$

[5] Projection and selection operations may also be involved but can be neglected regarding the performance issues that are dominated by the join operation

[6] E.g. regarding the underlying machine's architecture.

if S is sequential, i.e. all s_i in S are sequential and

$$c(S) = \sum_{i=1}^{n} c(s_i) = \sum_{i=1}^{n} \max_{j=1}^{k}(c(s_i.m_j)) \qquad (5)$$

if S is parallel, i.e. there is a parallel s_i in S. We note that the sequential case (4) is also covered by the (more general) parallel case (5).

Formula (5) gives two factors by which the costs of a strategy are influenced

- the number n of stages in the strategy and
- the costs $c(s_i.m_j)$ of the moves involved.

We will now track down what basic characteristics and parameters of the data relations, the query and the strategy influence these two factors.

The number n of stages that are necessary to move over the generation matrix is dominated by the size or dimensions of the matrix and the grade of parallelism achieved by the strategy. The dimensions of the matrix are determined by

- the number d of columns where d is given by the query and
- the number of rows which depends on the length of the longest path in the directed graph that is defined by the underlying database relation.

The size of the matrix is the same for each of the strategies we want to compare, so this will not cause any difference and can be considered as a given fact to all strategies. Evidently the number of stages can be reduced by packing as many moves into a stage as possible, i.e. executing as many moves in parallel as possible.

The costs $c(s_i.m_j)$ of a single move are not constant and difficult to calculate as can be seen in table 2: they are mainly dominated by the performance of the join and the transitive closure algorithms that are used and the parameters d (given by the query) and T (given by the data) so $c(s_i.m_j)$ can be tracked down by calculating the costs of the operations involved in the move and the number of times these operations have to be executed.

The costs of a join operation are determined by

- the size of the databases relations involved in the join and
- the data skew, i.e. whether the data values are distributed uniformly or not[7]

[7] There is a variety of papers covering the problems data skew causes on parallel joins, e.g. [WD91] [WDY93] [KO90].

whereas the costs of the transitive closure operation is dominated by

- the size of the underlying graph and
- the depth of the transitive closure.

In summary, there is a variety of parameters that influences the performance of the processing strategies. The parameters can be divided into two groups:

- *query-dependent* parameters like d and g and
- *data-dependent* parameters like the sizes of the relations involved in the query, T (the number of values for a particular attribute of a relation), the length of the longest path of the underlying graph of a relation, some data skew statistics about the distribution of attribute values and other relevant characteristics.

The first group of parameters can be derived just before the query is processed whereas the data-dependent parameters are usually accessible from the database catalog that holds several meta-information about the data to provide it to the optimizer. So we can assume that all or most of the parameters mentioned so far are available before the query is processed.

The following sections will discuss the performance issues of an actual implementation of a join and a transitive closure algorithm and the bottom-up and transitive closure strategies of figures 9 and 8 on a Connection Machine CM-200. Several parameters regarding implementation, operating system and hardware characteristics are also in the group of data and query parameters that influence the performance of query processing. This leads to a wide range of optimizing decisions that have to be taken to identify the best way of how to process a particular query. Discussing all these aspects exceeds the purpose of this paper so we will concentrate on giving some ideas and some examples in sections 4 and 5.

4 Parallel Join and Transitive Closure Algorithms

Considerations of sections 2 and 3 worked out that the join and the transitive closure are the two crucial operations in query evaluation according to the generation matrix model. In this section we will give a short description of the two algorithms that were implemented in the experimental database system we used for performance testing. We will concentrate on the issues regarding the performance behaviour of the algorithms. The reader might refer to the bibliography to get some more details about the algorithms themselves.

4.1 Parallel Hash-Join

The parallel hash-join algorithm was proposed in [KM91]. We incorporated the actual implementation described in [Min93].

The notion of a parallel hash-join is quite simple. Consider joining two relations R and S on a common attribute X as in

$$R(\ldots, X, \ldots) \bowtie S(\ldots, X, \ldots) \qquad (6)$$

The tuples of R and S holding the same value on the X attribute are sent to two separate cluster spaces on the same processor[8]. A hash function having the possible values of X as its argument gives the actual address of the processor to which the tuples are to be sent. So each processor holds the tuples that have to be joined in its two cluster spaces. Having distributed the data in this way each processor computes the cartesian product of the tuples in its cluster spaces. Finally the local results have to be put into one relation to form the global result of the join. So the essential steps of the parallel hash-join algorithm are:

1. a parallel clustering phase in which the tuples are spread over the processors depending on their value in the join attribute,
2. a local join phase in which each processor performs a join operation on its local data and
3. a joined relation generation phase in which the global result is built out of the results of the local join.

As we have already mentioned step 2 computes the local cartesian product of the tuples held on one processor. This is done by performing two nested loops over the number r_i and s_i of items in the two cluster spaces of processor i.

In the case of a one-attribute-pair-join (6) a local cartesian product is computed. In (7) there is a two-attribute-pairs-join

$$R(\ldots, X, Y, \ldots) \bowtie S(\ldots, X, Y, \ldots) \qquad (7)$$

In this case step 1 remains the same: tuples are sent to the processors according to their values on the X attribute (alternatively on the Y attribute). Step 2 works out to be a real equi-join as not every pair of tuples is joined (as in the case of a cartesian product) and the values of Y attribute (the X attribute respectively) of the tuples are matched before joining them. So a local join operation is performed on each processor.

The performance of the algorithm is mainly dictated by step 2 which has a time complexity of $O(\max_{i=1}^{n}(r_i * s_i))$ if r_i and s_i are the numbers of tuples of relations R and S on processor i using a total of n processors.

An ideal, uniform distribution of the work load amongst the processors is obtained if the the data is uniformly distributed over the cluster spaces as shown by figure 10. The function $\max(r_i * s_i)$ and therefore the performance of the algorithm is sensitive to three types of data skew which we will characterize by using the terms given by [WDJ91]:

Tuple Placement Skew: Some values of the join attribute occur in a large number of tuples whereas other values occur only in few tuples (i.e. some r_i and s_i are large whereas the others are small). This implies that some processors will hold a large number of tuples whereas some hold only a small number (see figure 11).

Redistribution Skew: The join attribute has only a small number of different values which means that a lot of processors will hold no tuples ($r_i = 0$ or $s_i = 0$) whereas a small group of processors has to cope with a large number of tuples, i.e. there is a mismatch between the distribution of the join key values in the relations and the redistribution to the n processors (see figure 12).

Join Product Skew: Although there might be a uniform distribution as shown in figure 10 there can be a difference in the size of the local join results due to selectivity[9]: some processors have to build a large result whereas others produce only a small one (see figure 13).

In fact all types of data skew cause a load imbalance amongst the processors. Nevertheless it is important to differentiate between these three types

[8] Actually only the addresses of the tuples are sent to reduce the costs of communication.

[9] In the case of a singe attribute pair join the algorithm we used computes a local cartesian product so no selectivity effects can arise when a uniform distribution is given. Nevertheless the case of two attribute pairs for the join is sensitive to join product skew.

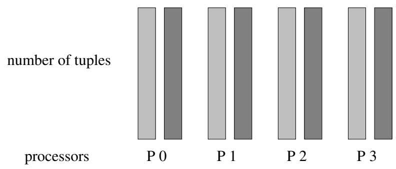

Figure 10: A uniform distribution of the data amongst the processors

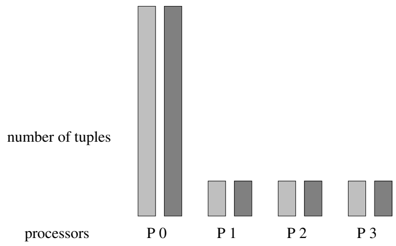

Figure 11: Effect of tuple placement skew

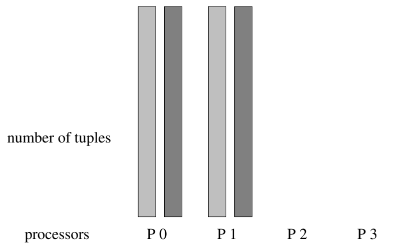

Figure 12: Effect of redistribution skew

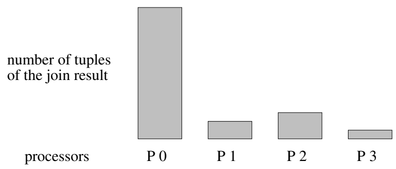

Figure 13: Effect of join product skew

because they are caused by different facts:
The data skew of a relation can be described by a function $f(v)$ where v is a value an attribute – in our case we concentrate on the attribute that is used for the join. $f(v)$ gives the number of tuples in the relation that hold value v in the respective attribute. If the set or range of values of v is denoted by $range(v)$ and $|range(v)|$ refers to the cardinality the range then[10]

- tuple placement skew is characterized by large values of

$$tps = \max(f(v)) - \min(f(v))$$

- redistribution skew is characterized by low values of

$$rs = |range(v)|$$

- join product skew is characterized by large values of

$$jps = \max(\text{sizes of local join results})$$

Figures 14, 15 and 16 show the performance behaviour of the parallel hash-join algorithm implemented on a Connection Machine CM-200. In each case the join

$$R(X,Y) \bowtie S(Y,Z)$$

was computed where R and S held 100 tuples each in the experiments of figure 14 and 1000 tuples each in all the other cases. In the experiments of figure 16

$$jps = \max(r_i * s_i)$$

because a single attribute pair join was computed. So figure 16 confirms the time complexity of $O(\max(r_i * s_i))$ of the parallel hash-join.

4.2 Boolean Matrix Multiplication Transitive Closure

This algorithm [Akl89] uses the *adjacency* matrix A of a graph and computes the powers $A^2, A^4, A^8, \ldots$ up to some A^{2k} such that

$$A^k = A^{2k} \tag{8}$$

which is guaranteed for $k \geq n$ where n is the number of nodes in the graph. A is an $n \times n$ boolean matrix. The powers of A are computed by performing several boolean matrix multiplications.

The matrices have the following properties

[10] Please note that the following definitions of tps, rs and jps were made for the purposes of this paper. In general skew is described slightly different.

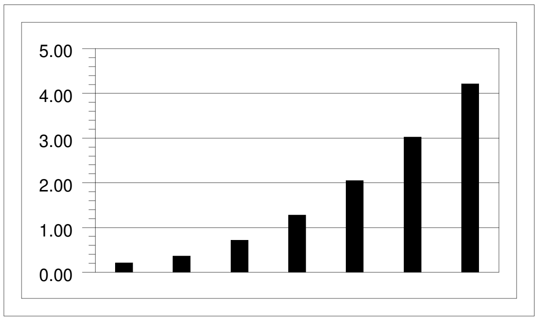

Figure 14: Tuple placement skew effect on the performance behaviour of the parallel hash-join on a CM-200 using 16384 processors. Results are shown for $tps = 5, 10, 15, 20, 25, 30$. Times are CM busy times in seconds.

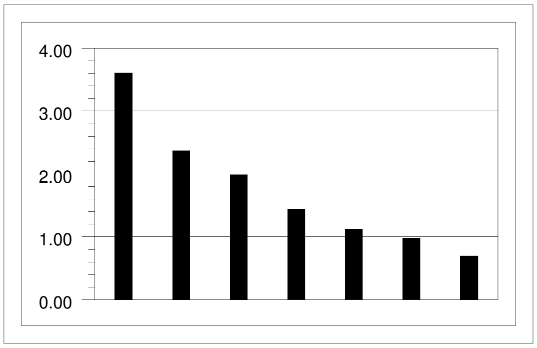

Figure 15: Redistribution skew effect on the performance behaviour of the parallel hash-join on a CM-200 using 16384 processors. Results are shown for $rs = 40, 50, 60, 70, 80, 90, 100$. Times are CM busy times in seconds.

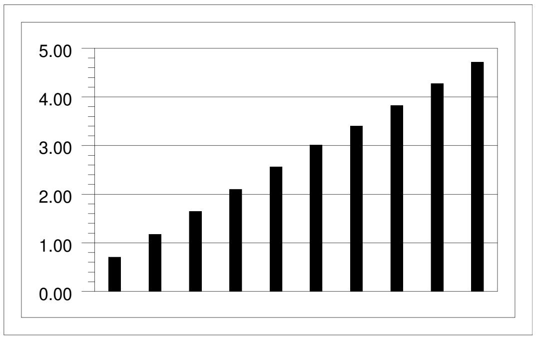

Figure 16: Join product skew effect on the performance behaviour of the parallel hash-join on a CM-200 using 16384 processors. Results are shown for jps = 100,200,300,400,500,600,700,800,900,1000. Times are CM busy times in seconds.

- A holds a 1 in position (i, j) iff there is an edge from node i to node j or $i = j$ and holds a 0 in any other cases,

- A^k holds a 1 in position (i, j) iff there is a path of length k or less from node i to node j and holds a 0 otherwise.

Actually A^k with property (8) is called the *connectivity* matrix and holds the reflexive and transitive closure of the graph represented by a boolean matrix.

The algorithm needs $\lceil \log_2 L \rceil$ matrix multiplications if L is the length of the longest path in the graph. The worst case is $L = n$, i.e. the longest path is involving every node of the graph. So in the worst case $\lceil \log_2 n \rceil$ matrix multiplications are performed.

There are parallel matrix multiplication algorithms of time complexity $O(\log n)$ but from a practical point of view it is unreasonable to use one of these as they require n^3 processors [Akl89]. It is reasonable to expect graphs with 1000 or more nodes which would require 1 000 000 000 processors or more. This results in an incredible overload of virtual processors per physical processors which reduces or cancels the benefit of achieving a time complexity of $O(\log n)$ for matrix multiplication.

Therefore a matrix multiplication algorithm of time complexity $O(n)$ for n^2 processors is used which results in a total time complexity of $O(n \log n)$ requiring n^2 processors for computing the transitive closure. The actual algorithm that was implemented on the CM-200 was proposed by [Can69]. Its suitability for hypercube architectures was established in [Joh87] [JH89].

The implementation of the boolean matrix multiplication transitive closure algorithm proved to be better than several other implementations of parallel Warshall-derived transitive closure algorithms that are also based on boolean matrices: see figure 17 and [ZT93] for details.

5 Some Performance Results

In sections 2.2 and 3 we identified several parameters that influence the performance of query processing. In this section we will give performance results obtained by some sequences of experiments on a Connection Machine CM-200. In these experiments we concentrated on investigating the performance dependency with respect to one single parameter, namely the length L of the longest path in the underlying graph of a database relation. For

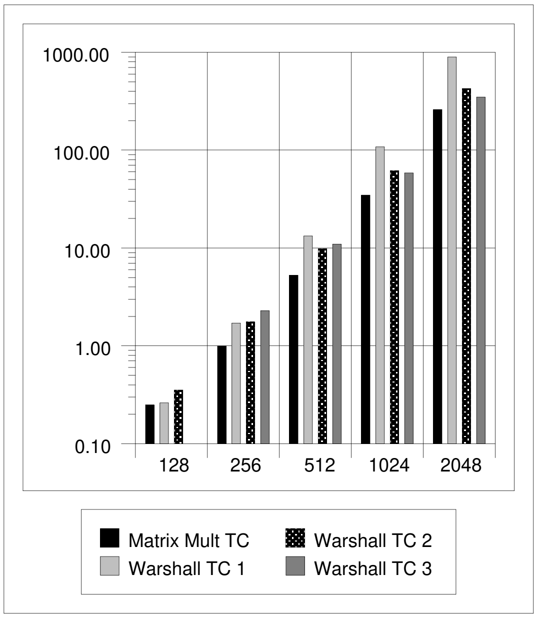

Figure 17: Performance results of transitive closure algorithms on a CM-200 with 16384 processors for graphes with $n = 128, 256, 512, 1024, 2048$ nodes. Times are seconds of CM busy time.

the experiments we used the most simple linear recursive query that computes the transitive closure of a relation Q:

$$P(X,Z) = Q(X,Z). \tag{9}$$
$$P(X,Z) = P(X,Y) \bowtie Q(Y,Z). \tag{10}$$

In each experiment Q holds 1000 tuples.
We give results for two different types of structures for the graph[11] that is defined by relation Q:

- acyclic directed graphs (DAGs) and
- trees

Such graphs can be found e.g. in relations holding data about company hierarchies or part-subpart relationships.

For each type of graph a sequence of experiments was set up. In each sequence the length L of the longest path in Q (respectively in the underlying graph of Q) was varied over $\{1, 2, \ldots, 10\}$.

The times given in the experiments are CM elapsed times, i.e. they include also the time spent on the non-parallel front-end computer and not only the time when the CM was busy. This is important because we want to consider also the parts of the code that do e.g. data structure transformations (before calling the actual database operation) or interaction with secondary storage. Using only the CM busy times would give misleading results in these cases.

5.1 Performance Results for DAGs

Figure 18 shows the type of DAG that was used in the experiments of figure 19. Each such DAG has $4 \cdot L$ edges and represents therefore $4 \cdot L$ tuples of Q. The relation Q was built up by several such DAGs[12] each with a longest path not longer than L.

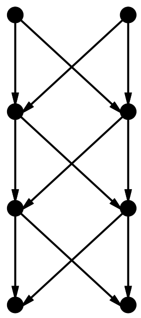

Figure 18: A DAG structure with $L = 3$ used for the experiments.

Figure 19 shows the performance of the bottom-up and the transitive closure strategies on a CM-200. The bottom-up strategy performs better for $L \leq 5$ whereas the transitive closure strategy is better on larger values of L. Clearly this value should no be considered as 'naturally given' as it may vary depending on the implementation but we can conclude that there is such a threshold value for L that marks the crossing point in which a query optimizer should switch from one strategy to the other.

Another important aspect is that the evaluation time of the bottom-up strategy is increasing linearly with the longest path as the number of joins is equal to the length of the longest path.

The times for the transitive closure strategy increase according to the number of matrix multiplications that have to be performed. This number is given by $\lceil \log_2 L \rceil$ which causes the 'stair'-effect for the transitive closure times in figure 19.

[11] A relation $Q(X,Z)$ can be considered as the set of edges of the graph, i.e. a tuple $(a,b) \in Q$ denotes that there is an edge from node a to node b. The values for attributes X and Y define the set of nodes of the graph.

[12] The exact number D of DAGs is computed by

$$D = 999/(4 \cdot L) + 1$$

Figure 19: Performance of the two strategies on DAG-structured database relations with the longest path L varying over $1, 2, \ldots, 10$ using 16384 processors. Times are CM elapsed times in seconds.

5.2 Performance Results for Trees

Figure 20 shows the type of binary tree graphs that was used in the experiments of figure 21. Q was built of several such trees each one having a longest path not longer than L.

Naturally the number of trees in Q is higher for small values of L because small L values imply that each tree has only a small number of edges which represent the tuples of Q. Therefore a larger number of trees is required in these cases. But a higher number of trees implies a higher number n of nodes, i.e. attribute values in Q. This is an important fact as the size of the adjacency matrix of the graph is determined by n. In the experiments n varied from 1500 for $L = 1$ to 1000 for $L = 10$. An important switch is between $n = 1034$ for $L = 4$ and $n = 1017$ for $L = 5$ because the CM operating system requires matrix sizes to be powers of 2 as the matrix represents a set of virtual processors to be mapped to the set of physical processors. For $1 \leq L \leq 4$ matrices of size 2048×2048 were used which represents a ratio of 256 virtual processors per physical processor whereas for $L \geq 5$ matrices of size 1024×1024 were used with a ratio of 64 virtual processors per physical processor. This marginal effect given by the hardware and the operating system software can be seen in figure 21 which shows the performance results of the experiments.

Figure 21 shows that the transitive closure strategy performs better for $L \geq 6$. As in the DAG experiments the existence of such a threshold value should be noted.

To make the contrast between the two strategies more evident we give also performance results for $L = 11, 12, \ldots, 20$ in figure 22. The bottom-up strategy increases linearly as the number of joins – which is the dominant operation of the strategy – is equal to L. The transitive closure strategy performs – apart from the effects implied by the virtual/physical processor ratio that we have discussed above – according to the number of matrix multiplications which is $\lceil \log_2 L \rceil$ (see section 5.1).

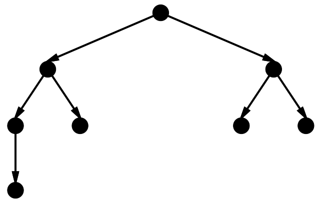

Figure 20: A binary tree with $L = 3$ and 8 nodes.

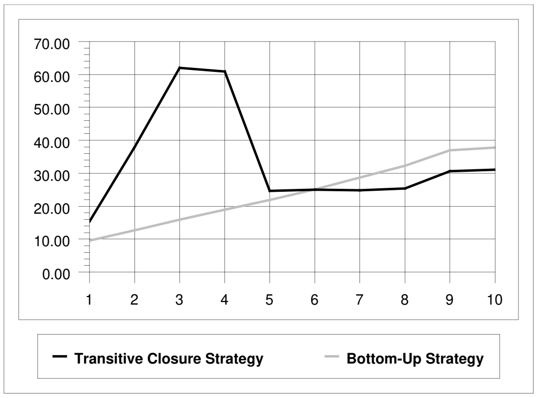

Figure 21: Performance of the two strategies on tree-structured database relations with the longest path L varying over $1, 2, \ldots, 10$ using 16384 processors. Times are CM elapsed times in seconds.

6 Summary and Conclusions

In section 2 we have presented the generation matrix model which offers several possibilities of processing a query. Two possibilities are the bottom-up and the transitive closure strategies.

Section 3 gave several parameters that influence the performance of these strategies and the join and transitive closure operations have been identified to be the most crucial. Therefore section 4 analyzed two implementations of parallel join and transitive closure algorithms and showed their performance behaviour.

Finally section 5 presented some performance results for two types of graph structures: DAGs and

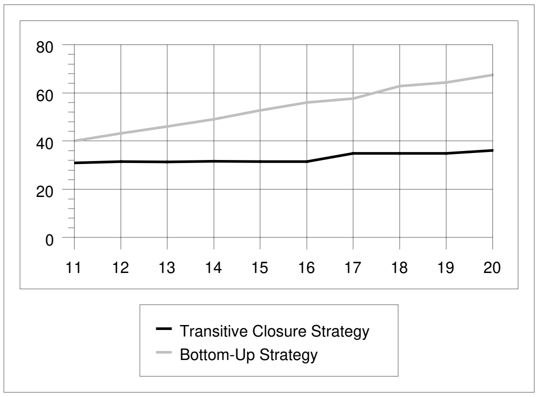

Figure 22: Performance of the two strategies on tree-structured database relations with the longest path L varying over $11, 12, \ldots, 20$ using 16384 processors. Times are CM elapsed times in seconds.

trees. In the experiments we concentrated on varying only one of several parameters that influence the performance, namely the length of the longest path of the relational graph structure. The analysis and the performance results show that the behaviour of linear recursive query processing strategies is quite sensitive to several factors. An optimal strategy can only be found by analyzing the characteristics of the query, the data and the algorithms for the database operations involving factors of the hard- and software systems as the virtual/physical processor ratio.

The transitive closure strategy has proved to be better if the length of the longest path passes a certain threshold value whereas the bottom-up strategy performed better on small values. An optimizer can therefore choose the appropriate strategy if the length of the longest path or at least an upper bound is known in advance or provided by the statistical information in the database catalogue.

The Connection Machine system architecture has also proved to be worth being considered as a platform for database query processing. We have shown that there are several data-parallel aspects than can be exploited naturally by spreading the data over a large number of processors and processing each portion of data independently by using the same database operations. This holds for the several database operations as the join but also for the overlying query processing model as the generation matrix where several 'moves' can be executed simultaneously on different portions of data.

Acknowledgements

The work presented in this paper was done at the Department of Computer Science of Edinburgh University. The programs were implemented and run on the Connection Machine CM-200 of the Edinburgh Parallel Computing Centre (EPCC). Parts of the work were funded by TRACS, an EC-funded project on high performance computing.

References

[ADJ90] R. Agrawal, S. Dar, and H. V. Jagadish. Direct Transitive Closure Algorithms: Design and Performance Evaluation. *ACM Transactions on Database Systems*, pages 427–458, September 1990.

[Akl89] S. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[BD83] H. Boral and D. DeWitt. Database Machines: An Idea Whose Time Has Passed? A critique of the future of database machines. In H.-O. Leilich and M. Missikoff, editors, *Proceedings of the Workshop on Database Machines*. Springer Verlag, 1983.

[Can69] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[Cor91] Thinking Machines Corporation. Connection Machine CM-200 Series Technical Summary, June 1991.

[DG90] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad. *ACM SIGMOD RECORD*, 19(4):104–112, December 1990.

[IW91] Y. E. Ioannidis and E. Wong. Towards an Algebraic Theory of Recursion. *Journal of the ACM*, 38(2):329–381, April 1991.

[JAN87] H. V. Jagadish, R. Agrawal, and L. Ness. A Study of Transitive Closure As a Recursion Mechanism. In *Proceedings ACM SIGMOD 1987 Conference on Management of Data*, pages 331–344, 1987.

[JH89] S. L. Johnsson and Ching-Tien Ho. Multiplication of Arbitrarily Shaped Matrices on Boolean Cubes Using the Full Communications Bandwith. Technical Report TR 721, Yale University, Department of Computer Science, 1989.

[Joh87] S. L. Johnsson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, pages 133–172, April 1987.

[KM91] M. Kitsuregawa and K. Matsumoto. Massively Parallel Relational Database Processing on the Connection Machine CM-2. In *Annual Report of Kitsuregawa Lab.* University of Tokyo, 1991.

[KO90] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). In D. McLeoad, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 210–221. Morgan Kaufmann, August 1990.

[LVZZ94] R. S. G. Lanzelotte, P. Valduriez, M. Zaït, and M. Ziane. Industrial-Strength Parallel Query Optimization: Issues and Lessons. *Information Systems*, 19(4):311–330, 1994.

[Min93] E. M. Minty. An Algorithm for a Data Parallel Hash Join on the Connection Machine. Technical Note EPCC-TN93-02, Edinburgh Parallel Computing Centre (EPCC), March 1993.

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ull89] J. D. Ullman. *Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[WD91] C. B. Walton and A. G. Dale. Data Skew and the Scalability of Parallel Joins. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 44–51. IEEE Comput. Soc. Press, December 1991.

[WDJ91] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 537–548. Morgan Kaufman San Mateo, September 1991.

[WDY93] J. L. Wolf, D. M. Dias, and P. S. Yu. A Parallel Sort Merge Join Algorithm for Managing Data Skew. *IEEE Transactions on Parallel and Distributed Processing*, 4(1):70–86, January 1993.

[ZT93] Th. Zurek and P. Thanisch. Processing Linear Recursive Database Queries on the Connection Machine. Technical Report EPCC-TR93-05, Edinburgh Parallel Computing Centre (EPCC), 1993.

[ZZO93] X. Zhou, Y. Zhang, and M. E. Orlowska. A Parallel Transitive Closure Algorithm for SIMD Meshes. *Australian Computer Science Communication*, 15(1):143–151, 1993.