PB-168

Staunstrup et al.: A Multiprocessor Database Machine

TRYK: DAIMI/RECAU

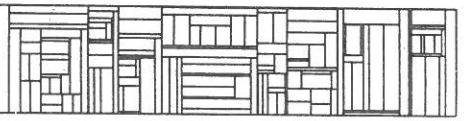
ISSN 0105-8517

PHYSICAL DATAREPRESENTATION IN A MULTIPROCESSOR DATABASE MACHINE

Jørgen Staunstrup
Jens Ove Jespersen
Ole V. Johansen

DAIMI PB-168
November 1983

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



PHYSICAL DATAREPRESENTATION IN A MULTIPROCESSOR DATABASE MACHINE

Jørgen Staunstrup, Jens Ove Jespersen, and Ole V. Johansen

Computer Science Department
Aarhus University
DK-8000 Aarhus C
Denmark

By using a multiprocessor to implement the lowest level of a relational database we want to achieve fast execution of database operations such as join, find, and update. But the potential speed improvements provided by a multiprocessor can only be achieved if one can construct algorithms and corresponding physical data representations that can utilize the potential. By choosing a particular representation, the grid file, and analyzing its behaviour, we want to point out the difficulties encountered in trying to achieve speed improvements from a multiprocessor.

1. INTRODUCTION

This is a summary of some preliminary investigations on how to utilize multiprocessor architectures to achieve fast execution of the operations on a relational database. We want to emphasize the importance of finding algorithms and data representations that utilize the potentials of a multiprocessor. One such representation based on the grid file [Nievergelt et al. 81] is suggested and analyzed. This representation is appealing because it is simple and independent of the distribution of data and operations; but no claim is made that the chosen representation is optimal or even superior to other representations. At the moment it is not even clear what should form the basis for a comparison with other representations.

The preliminary status of this work means that it is not a complete description of a multiprocessor database machine. Rather, we point out what we consider a major difficulty: programming a multiprocessor.

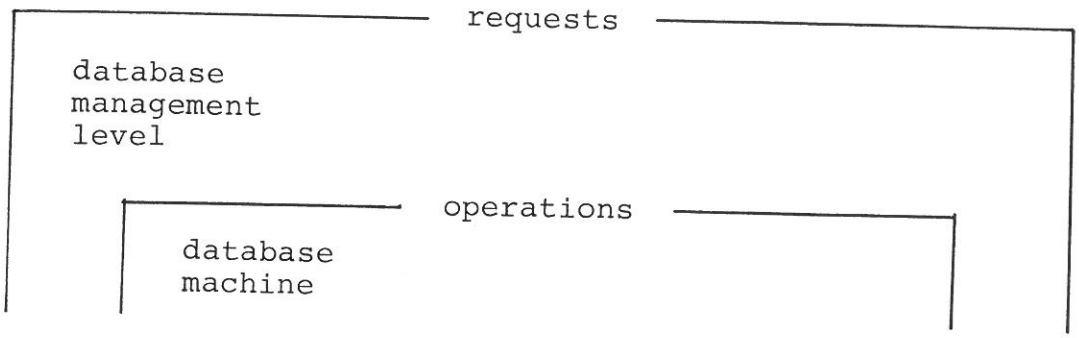
It is assumed that the reader is familiar with fundamental database concepts, in particular with the relational model.

2. WHAT IS A MULTIPROCESSOR DATABASE MACHINE?

First we describe what we understand by a multiprocessor database machine mainly to stress the difference from apparently similar notions e.g. a distributed database.

PAC 067 1 70

A database may be viewed as follows:



The operations which users perform on the database are called requests. These requests are transformed into operations. We distinguish between the database management level and a database machine. The most important tasks of the database management level are:

- translation of requests into operations
- privacy and consistency checking
- logging and backup.

This leaves the database machine with the responsibility of performing the operations. It uses some physical representation to store the data on the available storage media. When performing its operations the physical representation is manipulated. In a traditional database the database management level provides one sequence of operations to be performed by the database machine using a single processor. We want to study a multiprocessor database machine where a number of processors cooperate on performing the operations provided by one database manager. This should be distinguished from a distributed database where each processor has its own database manager.

There are two reasons for using a multiprocessor database machine:

- to improve the speed i.e. perform the requests faster
- to improve the reliability i.e. make the database less sensitive to failures in isolated parts of the system.

Our work concentrates on the first of these two aspects although we recognize that the second is at least as important.

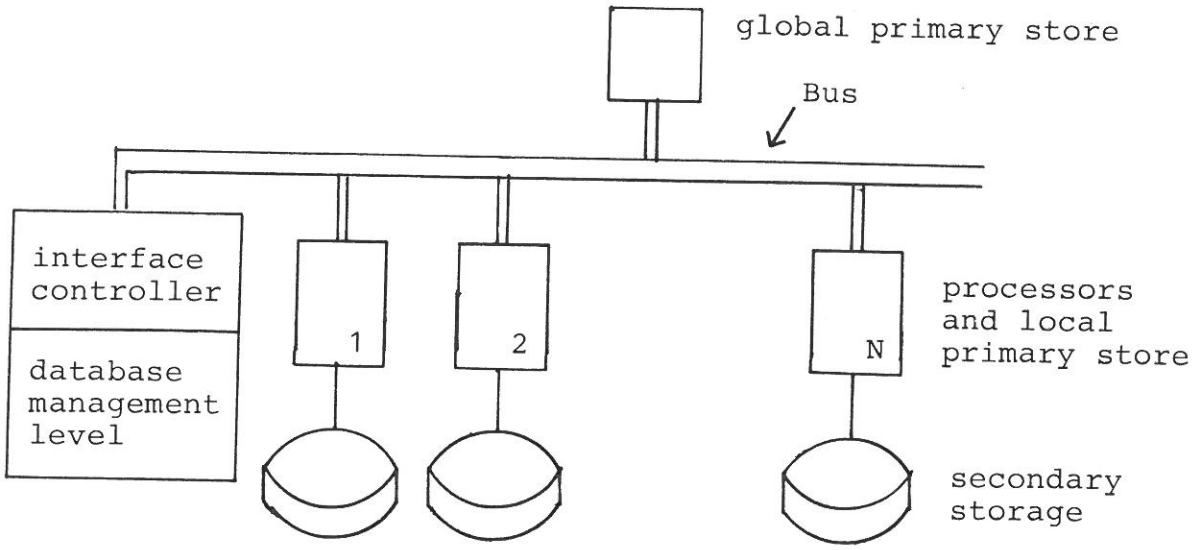
Speed improvements can be achieved by performing several requests simultaneously, called inter request concurrency or by performing several parts of one request simultaneously, intra request concurrency. Both of these should be used in order to perform the operations as fast as possible, but the inter request concurrency is administered by the database management level to maintain consistency of the database whereas a multiprocessor database machine must utilize intra request concurrency. Hence we concentrate on the latter in this paper.

3. MACHINE ARCHITECTURE

This section describes the machine architecture of the database machine that forms the basis of our investigations, it is based on the Multi-Maren multiprocessor [Møller-Nielsen and Staunstrup 82]. This machine was built in 1981 and has been in daily use since whereas

the database machine discussed here is so far a paper design only.

The database machine consists of a number of processors each with a local primary and secondary store:



PAC 067 2

70/

The emphasis in this paper is on software aspects such as data representation, the advantages and disadvantages of the architecture sketched above is not discussed in further detail, but a more comprehensive discussion may be found in [Hsiao and Menon 81].

The database is divided into a number of parts, each administered by a separate processor. Some operations may be performed locally with only a small amount of communication between the processors, but other operations e.g. the join operation considered in the next section may require substantial amounts of data to be communicated among the processors.

We imagine that the database management level is completely separated from the multiprocessor e.g. by running it on a dedicated processor which supply the database machine with operations to be performed through the interface controller.

4. DATA REPRESENTATION AND ALGORITHMS

This section describes a physical data representation and corresponding algorithms based on the machine architecture outlined above. The discussion is centered around the join operation which is the most complicated and time consuming of the relational operations.

A relation is a set of tuples: $\{(a_1,a_2,\ldots,a_m) \mid a_i \in D_i\}$ where D_i is the domain of the i'th attribute. A join of two relations P and S over a common attribute P_k and S_j respectively is a new relation J. This new relation can be computed by the following straightforward algorithm:

Nested loop algorithm:

```
for all (p1,p2,...,pm) in P do
  for all (s1,s2,...,sn) in S do
    if pk = si
      then J := J ∪ {(p1,p2,...,pm,s1,s2,...,si-1,si+1,...,sn)}
```

The running time of this algorithm is proportional to $|P \times S|$ i.e. the number of elements in the cartesian product. This running time can be reduced in at least the following two ways:

- Localizing data: the number of iterations in the inner loop of the above algorithm can be reduced if for each tuple in P a small part of S can be isolated where all the potential matches ($P_k = S_i$) occur. There have been proposed many algorithms and physical data representations which aim at localization. The most obvious thing to do is to sort the two relations with respect to a particular attribute. Another possibility is using the grid file representation [Nievergelt et. al 1981], this is investigated below.

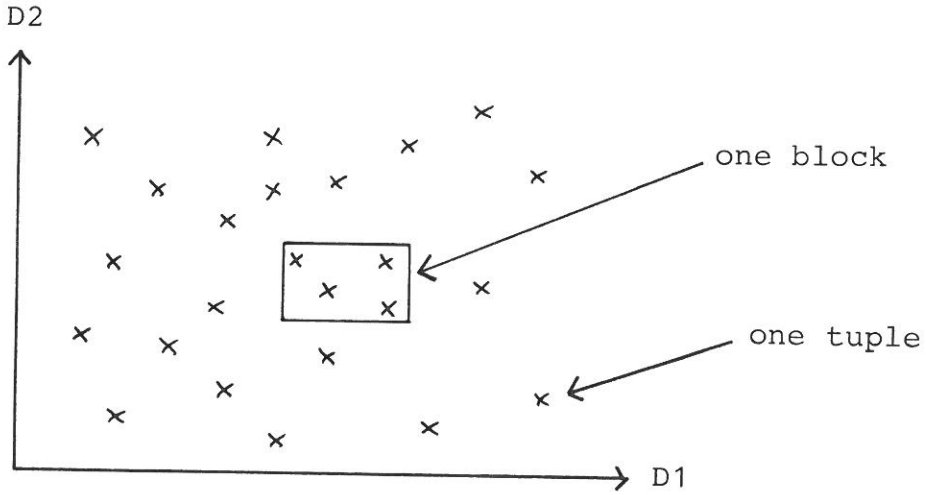
- Concurrency: the join does not prescribe any order in which the tuples should be inspected and included in the new relation. This may therefore be done on several sub-relations simultaneously.

This paper focuses on the last alternative. At first sight it seems obvious how to do this, namely letting each processor handle an equal fraction of the iterations in the nested loop algorithm. It is, however, not as simple as this. Most existing databases achieve significant speed improvements by localizing data. To be a real improvement a multiprocessor database machine must therefore utilize both locality and concurrency. In the following such an attempt is made to illustrate some of the difficulties involved in doing this.

4.1. Physical Data Representation

The tuples of all relations must be stored in some data structure on secondary storage e.g. discs. A common feature of such secondary stores is that data are grouped into so-called blocks where entire blocks only can be read or written.

In this study the grid file representation is chosen. This representation aims at obtaining good locality by storing tuples with nearly the same attribute values close to each other, usually in the same block. Another way of achieving good locality is by sorting the relation with respect to one of the attributes. If almost all join operations are over the same attribute, this is to be preferred. The disadvantage is, of course, that joins over all other attributes gain no locality. With the grid file representation all attributes are treated symmetrically. Consider a relation with two attributes over the domains D_1 and D_2. The following diagram shows the tuples in this relation as points in a two-dimensional space:

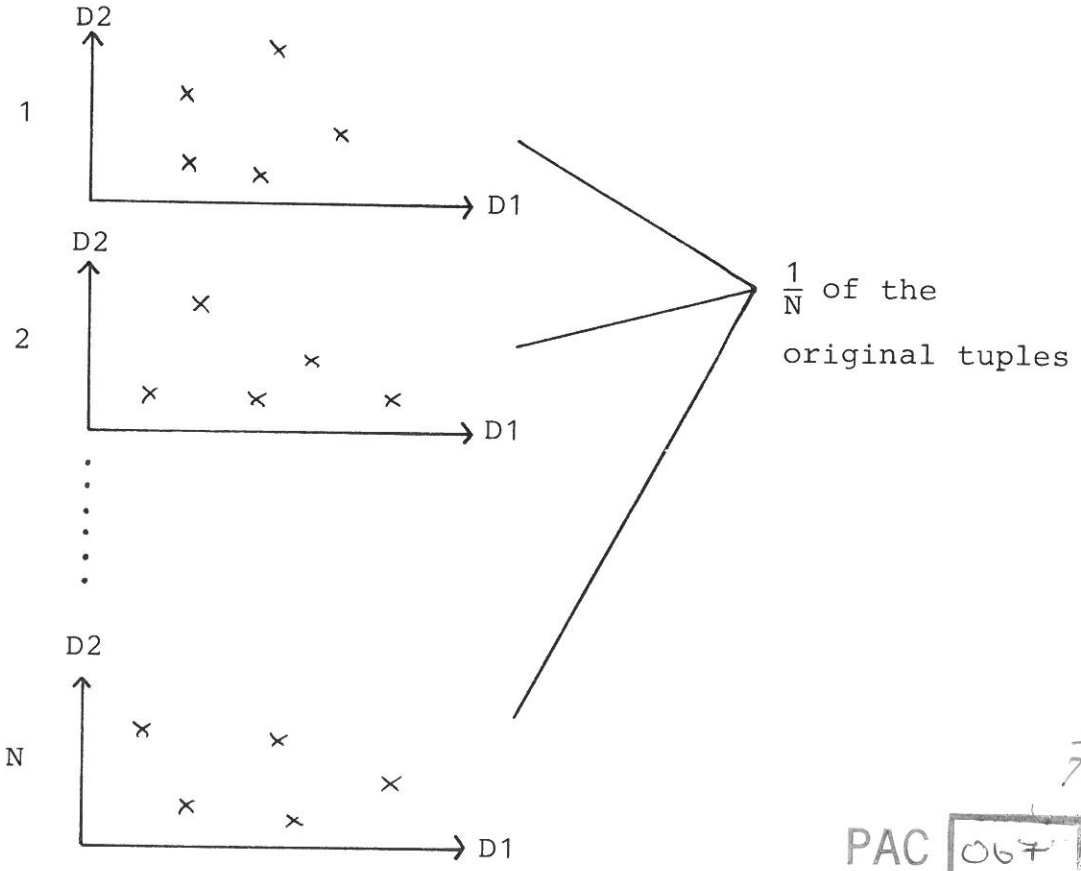


PAC 067 3

701

For any particular tuple t the goal is that all tuples in the neighbourhood of t are placed in the same block as p. Tuples are inserted and deleted dynamically, this leads to a splitting and joining of blocks. A directory keeps track of where to find a particular tuple. The technical details involved in realizing this are omitted here, they may be found in [Nievergelt et al. 1981, Jespersen and Johansen 1983].

The next step is to decide how the tuples/relations are distributed on the secondary stores of the processors. To be consistent with the choice of the grid file representation all attributes should be treated symmetrically. Consider again the diagram showing all the tuples of a relation with two attributes as points in a two-dimensional space. These tuples are now distributed at random among the N processors so that they all end up with approximately 1/N of the tuples.



701

PAC 067 4

The random distribution is simple to maintain dymamically. A tuple can be deleted immediately and new tuples are inserted at the processor with the currently lowest load. Each of the processors organize its tuples as a distinct grid file with a separate catalog. Hence, there is no global information telling which processors have which tuples. Let us repeat that we make no claim that the architecture and data representation described above is optimal, but its simplicity and symmetry is appealing.

4.2. The Concurrent Join Algorithm

When using the randomized representation in a multiprocessor with N processors each handles approximately 1/N of the tuples as separate sub-relations P_i and S_i:

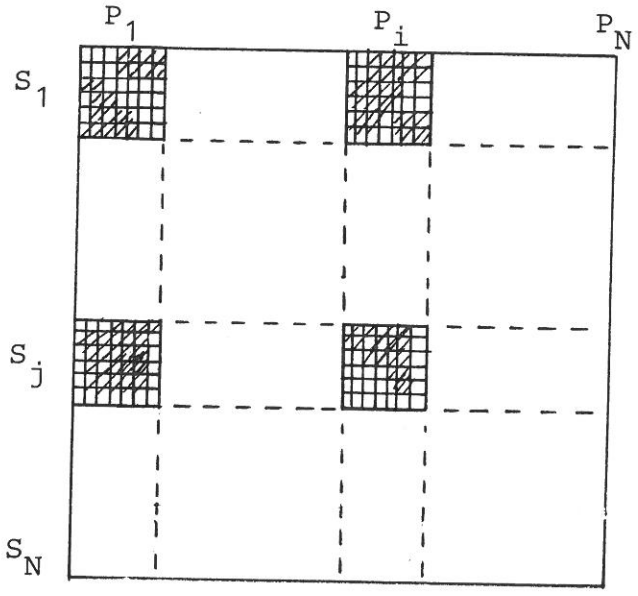
$$P = P_1 \cup P_2 \cup \ldots \cup P_N$$
$$S = S_1 \cup S_2 \cup \ldots \cup S_N$$

Now consider a join of the two relations P and S, $S \bowtie P$:

$$\begin{aligned} S \bowtie P = {} & S_1 \bowtie P_1 \cup S_1 \bowtie P_2 \cup \ldots \cup S_1 \bowtie P_N \\ & \cup S_2 \bowtie P_1 \cup S_2 \bowtie P_2 \cup \ldots \cup S_2 \bowtie P_N \\ & \vdots \\ & \cup S_N \bowtie P_1 \cup S_N \bowtie P_2 \cup \ldots \cup S_N \bowtie P_N \end{aligned}$$

Hence the join is split into N^2 joins of sub-relations. The joins in the diagonal are local joins within one processor, whereas all the remaining joins require communication of sub-relations between the processors.

The locality of the grid file reduces the amount of blocks to be considered in each of the N^2 joins but it does not eliminate any of the joins, which would of course be preferable. We have, however, not been able to find symmetric representations with this property, this is discussed in further detail in [Jespersen and Johansen 83]. The diagram shown below gives an example of what can be achieved by the grid file locality. Each square represents a join of two blocks, the shaded squares are the joins that are avoided:



70/
PAC 067 S

Processor i ($1 \leq i \leq N$) handles one row of the matrix:

$$S_i \bowtie P_1 \cup S_i \bowtie P_2 \cup \ldots \cup S_i \bowtie P_n$$

As mentioned above $S_i \bowtie P_i$ can be performed directly by processor i but the remaining join operations require communication of the tuples in $P_1, P_2, \ldots$ which must be sent to processor i. Similarly processor i must send the tuples in P_i to all the remaining N-1 processors. So the gain obtained by being able to perform N block transfers from the secondary store simultaneously is reduced by the necessity to communicate some of the blocks between the processors.

5. ANALYSIS OF THE RANDOMIZED REPRESENTATION

This section contains an analysis of the randomized representation described in section 4. Since the join operation is regarded as the most demanding of the database operations the analysis is based on estimates of the performance of this operation. It is, however, not clear how this performance should be measured. The following list contains some of the possible measures:

- the number of tuples communicated between the processors
- the number of storage references made by each processor
- the number of block transfers to and from secondary store
- the running time for executing a join operation on a particular machine.

We have chosen the number of block transfers as a measure, because it is reasonably machine and implementation independent. Furthermore it is the dominant factor in the running time. This choice has several disadvantages e.g. it ignores the cost of communication between the processors. It would be nice to find a better measure which is machine independent yet capture all aspects of the algorithm. The choice of such a measure is also under debate for monoprocessor representations [Batory and Gotlieb 82].

5.1. Upper and Lower Limits

We analyze the number of block transfers (from secondary storage) needed for joining P and S which are both two-dimensional relations. This analysis reveals some of the fundamental properties of the randomized representation.

With the chosen measure a perfect join algorithm would be one where each block of both relations is transferred from the secondary store exactly once. This is obviously a lower limit. In practice this lower limit cannot be achieved due to the limited capacity of the main store. But the better the locality of the representation the closer one gets to the lower limit.

Consider a block P_b of the relation P, in general there may be several blocks of S which intersect with P_b. (Two blocks intersect if there is a tuple in each of them with the same attribute value). In the algorithm described here the blocks of the relation P are transferred

once only whereas the blocks of S which intersect with more than one block of P are transferred several times. In the worst case all blocks of S are transferred for each block in P. Let #P and #S be the number of blocks in P and S respectively. If T denotes the number of transfers from secondary store, the above can be summarized as:

$$\#P + \#S \leq T \leq \#P + \#P * \#S$$

5.2. Estimating the Average

First we estimate the average number of block transfers needed when a single processor is used: T_1. This is compared with the number of block transfers needed by each processor in a N-processor database machine: T_N. Based on these we give the speed-up i.e.

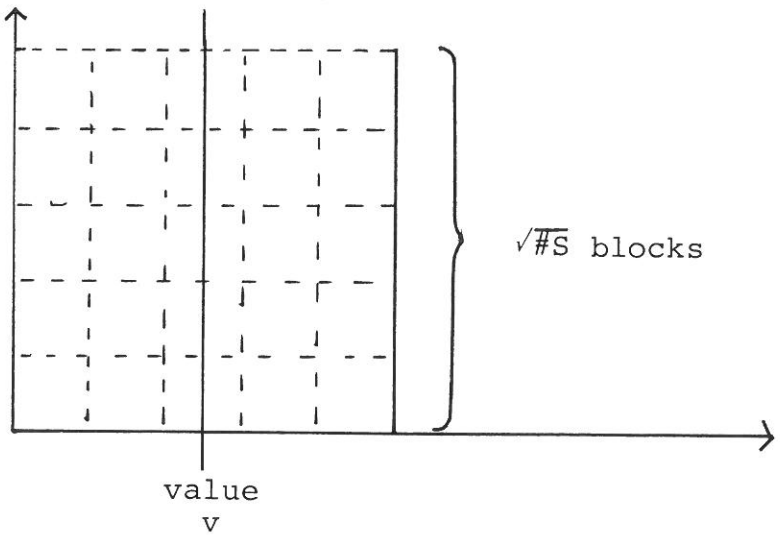
$$S_N = \frac{T_1}{T_N}$$

5.2.1. Estimating T_1

Consider a block P_b of the relation P, if it intersects with many blocks of S many blocks must be transferred from secondary storage to perform the join, whereas if it intersects with a few only these few are transferred. Let R denote the average number of blocks which intersects with each block of P, intuitively this is the ratios of the average "length" of the blocks in the two relations. We postulate that the average number of block transfers from S for each block in P, called TS, is:

$$TS = G(R) * \sqrt{\#S}$$

G is a function of R for which we have no analytic expression. It expresses how much locality the grid file provides. In our simulations G varies between 0.1 and 0.2 which can be interpreted as the locality achieved from the grid file avoiding 80-90% of the potential block transfers (80-90% of the squares in the diagram from section 4.2 are shaded). The factor $\sqrt{\#S}$ has a simple geometrical interpretation. In our implementation of the grid file the blocks tend to be squares i.e. on the average there are $\sqrt{\#S}$ blocks which may hold a particular value of an attribute:



PAC 06 6

It follows that:

$$T_1 = \#P * TS + \#P = \#P * G(R) * \sqrt{\#S} + \#P$$

The second addend "#P" stems from the one transfer of P (outer loop of the algorithm in section 4).

5.2.2. Estimating $\#T_N$

Now consider the multiprocessor algorithm described in section 4. Each processor has approximately 1/N of the blocks of both relations i.e. #P/N blocks of P and #S/N blocks of S. It was claimed above that R (the ratio of the average "length" of a block) determined how much the locality provided by the grid file representation helped. When the tuples are distributed randomly into N sub-relations, it means that the sub-relations become sparser (see section 4.1) i.e. the "length" of blocks increases. But since this happens to both relations R is not changed, hence G(R) is the same. The average number of blocks transferred from S for each block of P by processor j, TS_j, is therefore

$$TS_j = G(R) * \sqrt{\#S/N}$$

Since processor j must join its sub-relation of S with the whole of P we get

$$\begin{aligned} T_N &= \#P * TS_j + \#P/N \\ &= \#P * G(R) * \sqrt{\#S/N} + \#P/N \end{aligned}$$

5.2.3. Speed-up

Since all processors do their joins simultaneously we get:

$$\begin{aligned} S_N &= \frac{T_1}{T_N} \\ &= \frac{\#P * G(R) * \sqrt{\#S} + \#P}{\#P * G(R) * \sqrt{\#S/N} + \#P/N} \\ &\approx \frac{\#P * G(R) * \sqrt{\#S}}{\#P * G(R) * \sqrt{\#S/N}} \\ &= \sqrt{N} \end{aligned}$$

This estimate has been verified by simulation. Two small artificial relations with #P = #S = 200 were constructed and the number of block transfers were counted when 2, 4, 6 and 8 processors participated in the join. The results are summarized in the table given below:

N	T_1	T_N	S(N)	$\sqrt{N}$
2	3396	2382	1.43	1.41
4	3391	1688	2.01	2
6	3372	1323	2.55	2.45
8	3450	1132	3.05	2.83

Although the simulation is very small it confirms the above analysis of the number of block transfers. In [Jespersen and Johansen 83] the analysis is carried further to cover relations with more than two attributes.

Conclusion

The notion of a multiprocessor database machine has been presented. This is a special purpose machine built as a multiprocessor to speed up database operations. The key points are choice of physical data representation and a corresponding hardware architecture.

One such pair of representation and architecture has been analyzed. This has been done by studying the performance of the join operation. The join was chosen as a representative for the most complicated and time consuming operations. It appears that the proposal leaves a lot of room for improvements. Along with such an improvement it would be nice to find measures by which different representations/algorithms/architectures could be compared. The number of transfers between secondary storage and primary storage used here seems like a very crude measure.

In summary this stresses the importance of finding efficient multiprocessor algorithms and datastructures, unless this is done even the most sophisticated hardware cannot be utilized.

References

Batory and Gotlieb 82: A Unifying Model of Physical Databases, ACM Tr. on Databases 7, Dec. 1982.

Jespersen and Johansen 83: En Database-maskine baseret på en multiprocesenhed, Thesis (in Danish), Computer Science Department, Aarhus, May 1983.

Møller-Nielsen and Staunstrup 82: Early Experience from a Multiprocessor Project, DAIMI PB-142, Computer Science Department, Aarhus, Jan. 1982.

Nievergelt et al. 81: The Grid File: an adaptable, symmetric, multikey file structure, ETH-46, Zurich, Dec. 1981.

Hsiao and Menon 83: Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth, TR-81-7, Computer Science Research Center, Ohio State University, Ohio.