AN INVESTIGATION OF MULTIPROCESSOR STRUCTURES AND ALGORITHMS FOR DATA BASE MANAGEMENT

by

James Richard Goodman

Memorandum No. UCB/ERL M81/33

18 May 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the U.S. Army Research Office Grant DAAG29-18-G-0167.

AN INVESTIGATION OF MULTIPROCESSOR STRUCTURES AND ALGORITHMS FOR DATA BASE MANAGEMENT

by

J. R. Goodman

Memorandum No. UCB/ERL M81/33

18 May 1981

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

Rapid advances in semiconductor and disk technologies over the last few years have dramatically altered the role of computers in society. In particular, the use of computers for the storage, control, and dissemination of data has become a major factor in our modern economy. Traditionally, computers have not been design with data base applications in mind, and only in recent years has a serious effort been made to design computing hardware specifically for data base applications. Though some data base applications are performed well with conventional machines, a large class of problems are still prohibitively expensive. Through the use of parallelism, it is argued that greater performance improvements can be obtained for this class of problems than for the more traditional class.

The use of multiple processors closely cooperating to solve a single problem in the data base environment is studied. A survey is made of other designs, including the Intel study of the Relational Associative Processor (RAP), originally proposed at the University of Toronto.

A structured approach is taken to develop a set of principles by which a data base machine can be specified and designed. First the technological capabilities and constraints for the 1980's are examined. Then a study is made of data base operations and the the most critical ones are identified. Next one of these operations is examined to determine how it can be implemented on a variety of processor topologies and the best topology is determined for that operation. Then algorithms are developed for implementing the most critical relational data base operation: join.

A number of new algorithms are developed for implementing the join operation, utilizing hashing methods, and a number of well-known, conventional algorithms are extended to the multiple processor environment. The cost of these methods is determined in terms of disk accesses and in terms of communication among the processors, and the algorithms are compared under a variety of circumstances. The hashing algorithms are shown to be superior to the more conventional algorithms, and are shown to be limited only by the bandwidth of the disk.

The implications on the processor architecture are examined, and features are identified to support the hashing methods well. No exotic technologies are proposed.

The approach is shown to be viable for the specification of the optimal data base machine using the technology of the 1980's.

Acknowledgements

Prof. Alvin Despain was of invaluable assistance in every phase of the preparation of this work. Without his assistance, it never would have happened. I also wish to thank Prof. Eugene Wong and Prof. Austin Hoggett for their helpful comments on the contents of this work. Professors Carlo Sequin, David Patterson, and Mike Stonebraker provided many useful suggestions, and participated in many of the discussions which culminated in this document. Mr. Scott Fehr helped formulate many of the early ideas on which this work was based, and was extremely helpful in getting the text of this document into hard copy. Ms. Carole Veiss also assisted greatly in the final steps of assembling this work.

Finally, I want to thank my wife Dorothy, who didn't think I would ever finish, for sacrificing so much so that this could be completed.

J. R. G.

November 12, 1980

1.1.2. The Thesis

Given the importance of the DBMS to the modern world, and the rapid technological advances occurring in the computer field, it is important to take a structured approach to the problem of designing a data base management computing system. This involves the study of DBMS operations and algorithms to determine their requirements, the consideration of technology trends to determine the potentials and constraints of the future, and the use of a set of design principles for analyzing the performance of those tasks on the postulated system. One of the main thrusts of computer architecture currently is the search for methods which allow greater parallelism of operations, because the current execution speeds for sequential methods have been pushed to the point where further increases are extremely difficult. Thus the problem is how to exploit parallelism of all kinds to provide high performance data base management systems.

Certain types of data base operations are not widely performed today because of their high cost. The performance of these operations contains a great deal of inherent parallelism, however, and the potential for exploiting it has yet to be realized.

In the specification of an architecture, it is important to consider the algorithms to be implemented on the system. Different topologies of processors may be optimal for different problems — or even different algorithms for solving the same problem — so that in attempting to determine the best architecture, both the algorithm and the structure must be considered. Thus it is an extremely important, but complex, problem to design a large, high-performance, highly parallel system for support of a large DBMS.

CHAPTER 1

Introduction and Background

1.1. Introduction

1.1.1. The Importance of Data Base Management Systems

The growth of data base applications for computer systems has been extraordinary in the last few years. The rapid technological advances of recent decades have made possible the use of computers for the storage and retrieval of information over an incredibly broad range of applications. Dolotta [Dolotta 76], quoted by Sibley [Sibley 77], predicted that in 1985 $233 billion, based on 1970 US dollars, would be spent in this country alone for data processing. In 1975, Martin [Martin 75] cited evidence [Frost 73] that 20% of the Gross National Product of the United States was devoted to the collection and dissemination of data, with a rapidly increasing proportion of it being accessible by computers. He also showed that computer storage is growing exponentially, faster than any other area of computers, and predicted that the exponential growth will likely continue for one to two decades.

Modern computer systems allow the rapid access of huge amounts of data. Martin stated [Martin 75] that the first trillion bit on-line storage system was available in 1975, and emphasized the major role that data banks will play in the running of industry in the future. Among the most significant problems in computer science today is the issue of implementing Data Base Management System (DBMS) to provide high performance for an ever-widening array of applications.

assumptions and constraints. This will include only a sub-set of all possible DBMS machines, of course, and in particular will apply to those systems dominated by non-traditional queries of the type assumed.

1.1.4. Research Plan

The initial task must be to determine the constraints and define the assumptions required. This is fairly straightforward for the technology, since there is much agreement about the potentials for VLSI and magnetic disk technology. However, defining user requirements of DBMS machines, quantitatively or even qualitatively, will prove to be much more difficult. To accomplish this, models of DBMS operations will be developed to isolate the important functions and algorithms.

As an aid in developing many alternative hypotheses for structures and algorithms, the past work in the design of special-purpose DBMS systems will be studied in detail. Critical performance issues can then be studied by applying the models, given the assumed constraints, to the possible hypotheses. In particular, a few important, basic operations that control the performance will be identified. The relative cost and performance of these critical operations can then be studied to choose among the alternative hypotheses. This can be done by analysis, simulation, or experiment, or some combination of these.

Once the best structure has been determined for the critical operations, complete algorithms for the major DBMS functions can be determined. This can then be used to specify the architectural requirements for the components. The principles developed to guide the choice of the multiprocessor topology, the design of the DBMS algorithms, and the components will then constitute the desired principles of generic DBMS machines.

1.1.3. Goals of the Research

The primary goal of this work is to develop principles by which a DBMS machine can be designed rather than to design a particular machine. The technique employed to achieve this is inductive inference, in which alternative hypotheses are devised and critical analysis or experiments are conducted to exclude one or more of the hypothesized principles. This process can then be applied repeatedly to select and refine the principles.

This technique is to be pursued under the constraints of current and projected near-term technology and user requirements. The objectives are to maximize the system performance/cost ratio, particularly for operations currently recognized as desirable, but prohibitively expensive.

For the purposes of this investigation, the traditional hierarchal view of a computer system is assumed in which the system is viewed as a collection of components, interconnected to form a particular topological structure. The system is expected to manipulate information according to algorithms selected for it. This view allows the posturing of hypotheses regarding components, structure, algorithms, and their interaction. In particular, for this research, emphasis will be on the tight interdependence of algorithms and structure.

The first major goal is to determine the best topology for a multiprocessor data base machine, given a set of constraints and assumptions. This must include, of course, consideration of the algorithms and component capabilities. The next major goal is to determine the critical functions and the best algorithms to perform these functions for the defined structures. Finally, the functional and performance capabilities of the components can be determined.

The collection of principles governing each of these areas then serve to define the generic architecture of DBMS machines that fall under the original

1.1.5.1.2. The Network Model

This is a generalization of the hierarchal model in that more general graphs than a tree are allowed. It is more complex because of the fact that it can no longer be laid out in linear space in a straightforward way, but must rely on explicit pointers to define all the edges of the graph. The proposal known as CODASYL [CODASYL 71] provides a detailed definition of this model.

1.1.5.1.3. The Relational Model

This model, due to Codd [Codd 70], stores a set of records of a given type in an unordered array known as a *relation*. Each record, known as a *tuple*, contains fields, often called *domains*. More precisely, a domain is the set of values that the field can take on. An element in a field is known as a *column value*. Each record has a unique value assigned to it, commonly known as the "tuple identifier," or *TID*. The relational model makes no use of physical positions of records to define the relationships among records, requiring explicit declaration of all such information.

The relational model, being the newest of the data base models, as yet has been implemented in very few commercial products. However, it has been the subject of intense study, both in academia and in industry, in the last decade and numerous experimental versions have been developed [Astrahan 76], [Stonebraker 76].

The difference in the organization of the data base means that certain operations on the data base may be done in rather different ways. In particular, the hierarchal and network models are characteristically accessed through a so-called *procedural* language, where at any point in the

1.1.5. Terminology

A large number of terms have come into use with respect to data base systems. In many cases the distinction between terms is largely historical with no widely agreed upon distinction between the terms. Many corporations have added to the confusion by introducing their own terminology for concepts widely known by other names. Thus it is necessary to define some terms to be used, since their meaning is not universally recognized.

It is assumed here that a data base consists of one or more *files*, each containing one or more types of *records*. A record of a given type contains the same set of (possibly empty) *fields*. Each field contains one element from a set defined for that field.

1.1.5.1. Data Base Organizations

The organization of data bases can be broadly classified in three types [Date 75], though many implementations, particularly recently, have been hybrids, created in an attempt to distill the desirable qualities of each.

1.1.5.1.1. The Hierarchical Model

The oldest model for organizing and accessing the data is the heirarchical model. In this method, the records are stored in linear memory as a tree structure, usually in post-fix form. The relative positions of the records are significant in that they convey implicit information about the structure of the data. The IBM DBMS known as IMS is the most widely recognized hierarchal system, though it has now been enhanced to support the network model, discussed next, to some extent [Date 75].

relational model will be studied in more depth, some of the specific operations allowed will be described.

1.1.5.2.1. Restriction

The operation known as restriction, or sometimes as selection, is the specification which eliminates some of the records (tuples) of the relation. That specification might be, for example, that the entry in a certain field have a particular value. Referring to the telephone book data base model, a restriction might be

List all of the entries for people named Nixon.

In many cases a single record may survive a restriction. For example, if all values in a given field are unique, then at most one record will survive. On the other hand, a large number of records may survive if, for example, the query is:

List all of the entries for people who live on Telegraph.

1.1.5.2.2. Projection

Projection is the elimination of certain fields of all records. In many cases, a number of the fields contain information which is irrelevant for the query, and these fields are deleted. This may have the effect, however, of producing multiple, identical records, since many records may be identical in a number of fields. The relational data base model assumes that all records are unique. Thus in theory, duplicates must be eliminated. In practice, because of the expense of eliminating duplicates, this process is frequently either delayed or not checked.

execution of the program a "current" position in the data base exists. Thus it is possible to select a string of records, each being accessed on the basis of its relation to the last. This approach is known as *navigation.*

In contrast, the relational model has no defined order for the records, so no such navigation is possible, and all accesses must be made by the selection of records meeting a set of criteria. This qualification is known as *selection.* A language supporting such set selection is known as *non-procedural.*

An important observation about these two approaches is appropriate here. The procedural methodology encourages the use of language structures which obscure the parallelism inherent in many data base operations. Although a look at the program may reveal that a large number of records are to be accessed, and the same sequence of operations is to be performed on them, the low level accesses to the data base itself may exhibit none of that potential parallelism, appearing simply as a stream of requests for various records. While this may also be true for the relational model at the lowest levels, the statement of the query as a set of qualifications on the relation makes that inherent parallelism easier to pass on to the access method responsible for retrieving the physical records.

1.1.5.2. Operations on the Data Base

The accessing of selected portions of the data in a DBMS is controlled through the specification of a *query.* Those records which meet the specification are said to *qualify.* Frequently the qualifying records are returned as the result desired by the user. An alternative operation, however, is to change some parts of those records which qualify. Because the

once, two records occur for each possible pair of entries under that phone number. Fig. 1.1 gives a simple example of such a join.

1.1.5.3. Support Structures

A wide range of methods have been developed for the rapid access of records in a DBMS. Such support structures, called *access methods*, perform the function of retrieving specified records efficiently. The most common access methods involve the use of *indices*, which allow the rapid determination of the address of a record based on the value of one or more of its fields. The value used to access a particular record is known as the *key*, and a field on which an index exists in known as a *key field*. Typically, an index consists of a set of pairs whose first component is a particular value of the key field, and whose second component is a pointer to a component with such a value in that field. When multiple records have the same value in a key field, however, the index entry may be a list, rather than a pair, where all but the first entry are pointers to the appropriate records.

A number of techniques exist for gaining rapid access to the relevant portion of the index. There may be an index on the index, for example, or the index itself may be stored as some kind of a tree structure [Held 78]. The values in the index may be hashed to allow rapid access to data which is non-random [Martin 75, pp. 291-311]. If the right index exists, the simplest queries generally can be answered by retrieving the relevant portion of the index, which may involve one or more disk accesses, followed by a direct access to the record requested, which may also require a disk access.

Frequently, restriction and projection are used together to fetch only certain desired fields of a number of records. For example, in the telephone directory model, a query requiring both a restriction and a selection would be:

List all of the street names present in the directory.

If the list is to be ordered and non-redundant, then a sort and elimination of duplicate entries will be required.

1.1.5.2.3. Join

The *join* operation, sometimes called *equi-join,* is the means by which the information in different relations is combined. A join is defined for two relations each of which has a field with a common *domain,* i.e., a common set of possible values. The relation formed by the join contains all the columns of each relation. The join relation consists of some subset of the cartesian product of the two relations. By far the most common subset is all those pairs of records for which the value in the join fields of the two records is the same. Although this is technically known as "equi-join," it is so common that it is frequently termed simply "join." That will be the usage here.

The two relations participating in the join can in fact be the same relation. Using the telephone directory model once again, consider the following query:

List all of the phone numbers listed more than once.

A join of the relation with itself using the *phone_number* field results in a record containing two copies of an entry for each entry in the directory. In addition, for every phone number present more than

In the relational model, the tuples are not ordered explicitly. The order in which they are stored, therefore, is of no consequence except in terms of performance. In fact, the order in which they are stored has a great bearing on the difficulty of responding to a particular query. If, for example, all the records satisfying the query are on the same block on the disk, then only one disk access need be made after they have been identified. This property has been defined by [Blasgen 76] as *clustering*. A relation is said to be clustered with respect to a particular index if a scan of the relation itself reaches the records in the same order that they are listed in the index when sorted by key value.

1.1.5.4. Hardware

Among the hard disks available today, two basic types are identifiable: the fixed-head disk (FHD) and the moving-head disk (MHD). The FHD has a set of read/write *heads* which are permanently mounted and remain stationary while the magnetic surface moves beneath it and which read and store the data. The MHD, on the other hand, has an additional capability to move the head in and out on the surface to access different sets of data. The arrangement is similar to a phonograph record where there recording is a series of circular *tracks* rather than a single long spiral. The FHD can only have a small number of such tracks, since a separate head must be supplied for each. The MHD, on the other hand, can be moved from one track to another to access different sets of data. In addition, a series of such platters may be stacked up, with space in between for a rod containing two heads to be inserted. Often the FHD is also constructed of multiple platters with only one or two heads associated with a surface of each platter. The *arm*, which con-

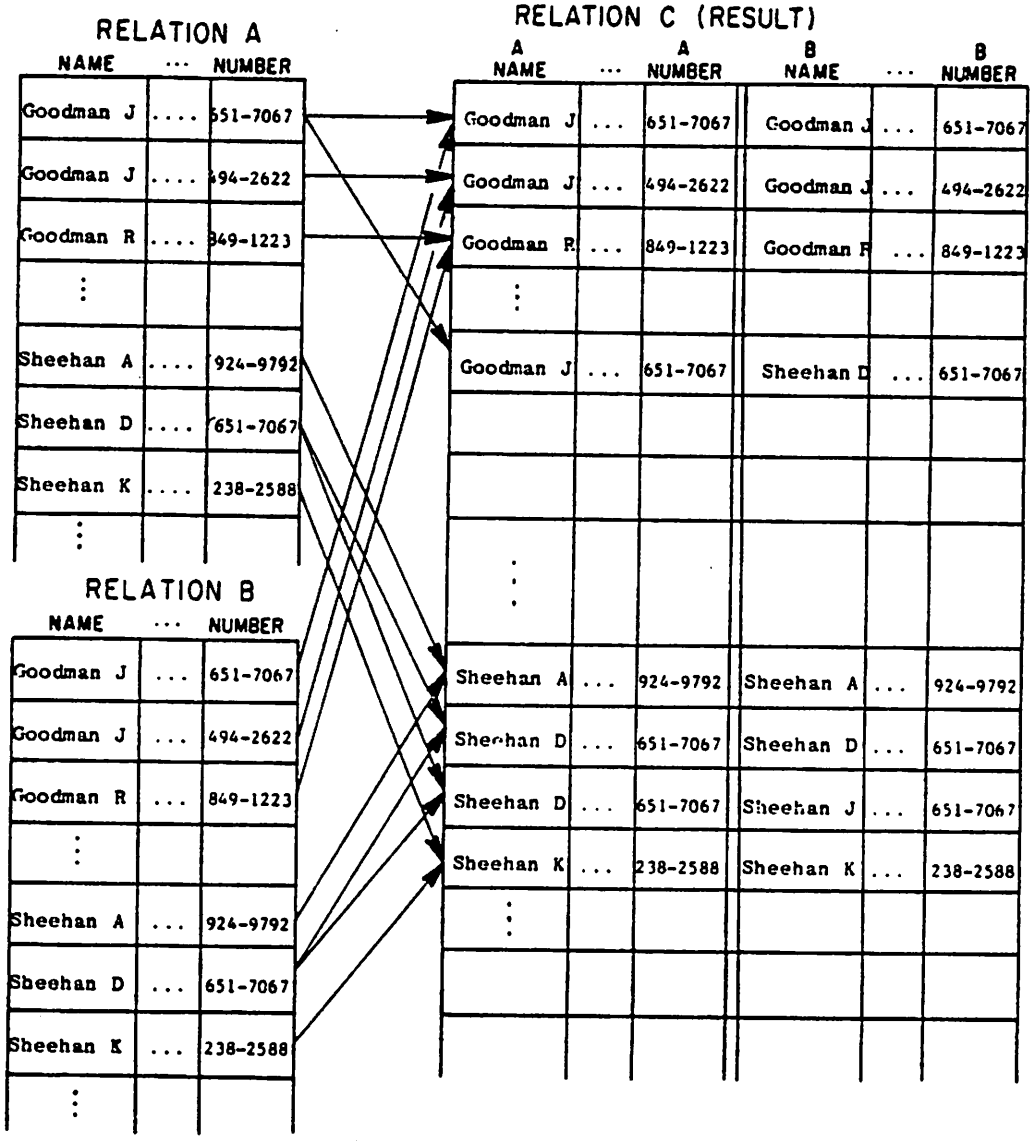

Figure 1.1. The Join Operation in the Relational Data Base Model.

It has been noted that the density of devices produced on a single semiconductor circuit has been increasing exponentially [Noyce 77], though a moderate slowing in that trend has been perceived in the last few years [Patterson 79]. Assuming a continuation of the recent trends, it has been predicted that integrated circuits containing one million devices will be fabricated during the middle 80's or even earlier [Patterson 79]. The problem of utilizing this quantity of devices to build complex functions within the economic and technological constraints of the semiconductor industry poses a serious challenge to the industry.

With the increasing complexity of functions being developed for a single integrated circuit through the use of Very Large Scale Integration (VLSI) technology, one approach proposed to handle large computer application problems is the use of large numbers of "single-chip computers" [Despain 78]. Though the potential for high performance by exploiting such hardware techniques appears to be enormous, the problems of multiprocessor architecture are numerous and difficult, particularly for designs using more than a handful of processors. Many of the problems of using large numbers of processors together are not well understood.

To satisfy semiconductor economic constraints, it is necessary to define a small number of components which can be used as basic building blocks for even very large computing systems. Suppose that the basic building block is taken as one such component (or possibly several) which is, in many ways, a computer in its own right. Given such a component, the important questions then are:

(1) How should a large number of these components be connected together if the computing environment envisioned is a data base application?

sists of all rods, can be moved in and out, and for each position, every head can access a different track. The set of tracks accessible through all the heads at any one time is known as a *cylinder*. The set of all tracks accessible to one head, which are all on the same side of a single platter, is known as a *surface*.

When a piece of data is to be fetched, it is frequently necessary to wait until the platter turns so that the data is under the read head. In the case of the FHD, and for the MHD when the head is already situated in the correct position, this time is known as the *latency*, and is equal to the time for the platter to make half a revolution, on average, and a full revolution, worst case. In addition, for the MHD, if the arm is not properly positioned, so that it must be moved, an additional delay is encountered, and this time is known as *seek time*. Seek time is generally significantly smaller for adjacent tracks than for those further apart, but the function is not linear. Latency and seek time do not overlap. Thus the MHD provides access to much more data than the FHD, but the time to access it is generally slower, unless the arm happens to be in the correct position.

1.1.6. Technology Projections and Constraints

The rapid advances in computer systems in the recent past have been due primarily to the technological advances in semiconductor technology, disk technology, and other related areas. The architecture of these systems has been driven by these advances, and they continue to be the predominant influence in new designs. The resulting rapid reduction in the cost of computing relative to alternative possibilities has resulted in a wide range of applications which are influencing the architecture.

Thus a design which takes full advantage of the technology available must find ways to exploit the capabilities of VLSI and magnetic disk. The technology advances in these two areas will surely have a profound impact on the architecture of computers over the next few decades, since so much of the computer system is made up of these components.

1.1.7. DBMS User Requirements and Needs

One of the prime uses of computers today is for the control and storage of textual information [Date 76]. The recognition of a need for controlling access, maintaining integrity, and allowing sharing of this information has resulted in very large systems programs, collectively known as a data base management system (DBMS). The data base itself, composed of a wide variety of possible kinds of data, may be as small as a few hundred entries, or it may number in the millions. In addition, a wide variety of support structures are frequently maintained to allow rapid access to the data under certain conditions, hopefully those most often occurring. These structures, which have a wide variety of names and implementations, are frequently quite large themselves, sometimes as large as the data they support. [Martin 75], [Maller 78].

The programs that control and manipulate the data may be extremely large, rivaling, or even exceeding in size the operating system itself [Gray 78], [Kerr 79]. (The INGRES system, for example, which is a university research project, not a commercial product, and for which many features have not been implemented, nevertheless comprises more than half a million bytes of source code [Stonebraker80].) While in many applications the resulting performance is satisfactory, in many other cases it is not. However, the recent explosion in the use of data base management systems testifies to the fact that the market

(2) What sort of algorithms can be applied in the answering of a single large query presented to the system, and how well would such a system perform?

(3) What sort of functional capabilities are required of each component to provide high performance execution of the algorithms.

Disk technology has shown rapid advances over the past several decades, though its progress has recently been largely overlooked due to the even more rapid advances in semiconductor technology. Nevertheless, disks are the dominant technology for on-line mass storage today, and there is no indication that any other technology threatens their position [Hoagland 76]. The density of storage on disks has increased at an exponential rate since the 50's — an improvement of more than three orders of magnitude — and indications are that this growth rate will continue for some time [Hoagland 76]. Hoagland states that nearly two more orders of magnitude improvement appear to be feasible.

Though some improvement in seek time has occurred, Hoagland claims that changes of an order of magnitude in the near future are neither feasible nor worthwhile. Latency times are not expected to drop dramatically because spinning disks are largely limited by the speed of sound in their rotational velocity. Transfer rates will improve only in proportion to the square root of density improvements, *i.e.*, to linear density improvements, since they are restricted by the velocity of the disk.

The error rate of data read from disks today is quite low because detection and correction of errors is standard. However, Hoagland predicts that medium flaws in the magnetic material will cause an increase in the error rates and require the capability for marking sections of the disk as defective.

query made. Hawthorn [Hawthorn 79a] has characterized queries as "overhead intensive," for those queries for which determining how to access the data and answer the query dominates the work, and "data intensive," for those queries for which manipulations of the data itself limit the response. When a DBMS is properly designed to handle the class of queries it receives, the actual interaction with the data may be quite small — one or two access to data possibly already residing in main memory. If, on the other hand, the query received is not of a type anticipated in the design of the data base, the response may involve arbitrarily complex interactions with the data base. Though such operations are frequently not performed today because of the expense, a substantial reduction in their cost will surely increase their use.

Any architectural innovations which can dramatically improve the performance/cost ratio will have great appeal. Therefore, the past proposals for innovative computer architectures will be analyzed with respect to their ability to enhance the performance and lower the cost of data base management systems.

1.2. Background

1.2.1. A Survey of DBMS Hardware Designs

For some time now the special requirements of a DBMS machine have been recognized [Slotnick 70],[Parker 71],[Parhami 72],[Coulouris 72]. The functions consuming the bulk of the resources in a DBMS machine are very different from the classical needs of numerical computation. While large demands are placed on the system I/O (input/output), the CPU (central processing unit) is heavily taxed with chores such as sorting, merging lists, and comparing strings, with only limited numerical computation required. [Hsiao

for such systems is quite elastic [Martin 75].

In today's environment, many systems are unable to answer certain questions, despite the fact that the desired information is present. The data is not organized in a manner such that the answer can be found at an acceptable cost. To demonstrate why this is true, consider the following problem: Given the telephone directory as a data base, determine the name of the occupant at a known address. Although the information is clearly present in the directory it cannot be readily retrieved. If this question were to be asked frequently, the problem could be solved by producing a secondary index of addresses so that the data could then be fetched directly. It is not feasible to build such so-called secondary indices for all possible queries, however, since those queries can become arbitrarily complicated. For example, a much more difficult question regarding the telephone directory model would be to determine the names of all people whose telephone number is listed in the book more than once. The only general solution is to be prepared to search the entire data base — possibly many times.

As the size of the data base grows, these problems become worse. This is due to a number of causes. For example, a sort becomes rapidly worse once the file to be sorted no longer fits in main storage [Knuth 75]. Likewise support structures to aid in efficient retrieval of records may not fit in main storage if the data base is too large, resulting in a rapid increase in disk activity. Yet the trend today, and the expectation of tomorrow, is that data bases will continue to grow at a substantial pace [Martin 75].

The users of data base systems are difficult to define, in large part because they are so incredibly diverse. It is possible, however, to characterize the use of data base systems in broad categories according to the types of

1.2.1.1. Head Per Track Devices

Although interest in special hardware has been demonstrated long ago, [Goldberg 62] [Yau 66] [Dugan 66], much of the recent interest has developed from the idea of using a head-per-track disk with peripheral logic to create a quasi-associative memory. The idea is that if the data coming under each head can be read and processed in real-time, then the entire disk can be searched in one complete revolution of the disk. In order to process the data at such a rate, special hardware is included for each head to read the data and perform some filtering of the data. Thus the disk may be thought of as an associative memory with a cycle time equivalent to one revolution of the disk. Fuller *et al* [Fuller 65], first proposed the idea that a disk could be associatively searched. [Slotnick 70], however, was the first to suggest that the logic-per-track idea might be a generally useful concept in a range of applications. Interestingly enough, he was motivated by the claim that "the cost of random access memory ... has not been going down as rapidly as most buyers would wish." Early designs using this idea were proposed in the early seventies by [Parker 71], [Parhami 72], [Coulouris 72], and [Minsky 72].

An interesting problem, first pointed out by [Minsky 72], is the inherent serialization of the data on a disk track. If control information about a record is to be stored with it, it may be necessary to access it before the record itself is manipulated. On the other hand, it may not be possible to determine the information to be entered until the whole record has been read. For example, if the first field in the record was to be modified based on a comparison of a string against the value in the fourth field, it is necessary to read the fourth field, then modify the first.

78]. As computers are increasingly being recognized as information management tools, increasing attention is being paid to these textual manipulation functions. In particular, during the past decade, a number of researchers have proposed new and very special hardware to effectively perform these functions [Slotnick 70], [Parker 71], [Coulouris 72], [Healy 72], [Minsky 72], [Parhami 72], [Copeland 73], [Moulder 73], [Canaday 74], [Madnick 75], [Ozkarahan 75], [Banerjee 76], [Lin 76], [McGregor 76], [DeWitt 78], [Babb 79].

Several different approaches are possible to support DBMS applications with special hardware. One is to build a special purpose computer to handle the functions of the DBMS exclusively. It is usually attached to a more conventional system which provides user interface support and other computer functions. This is known as a *back-end* system. Another approach is to build special hardware which allows the general purpose computer to perform its data base functions more efficiently. The line between these two approaches is not distinct, however, since most proposals of the latter type include the use of a sophisticated controller for the special hardware which, along with that hardware, may be considered a special purpose computer.

A third approach is to build a computer system which is designed to perform all the normal computing tasks demanded of it, but which is organized in a way which allows it to support the DBMS requirements especially well. In particular, a multiprocessor system might be organized so that many processors could cooperate efficiently on a single data base query and so that the data required from the disk could frequently be read directly by that processor which requires it.

The past proposals considered can be grouped in three general classes according to the unifying concept which characterizes them.

1.2.1.1.1. CASSM

In 1972, Healy, Doty, and Lipovski [Healy 72] proposed a system which they called CASSM. A prototype of CASSM using a fixed head floppy disk [Copeland 73] [Su 75] was proposed. Although the claim has been made that the system was implemented [Lipovski 78] it is not clear from a conversation with one of the principles [Copeland 77] whether any hardware was actually assembled. The design, Fig. 1.2, called for a number of cells, each cell containing one track of a fixed head disk and some special logic. Each cell could directly communicate with its two immediate neighbors and with a single common controller through a bus. The controller alone communicated with the central processor, which was responsible for communications with the users and general I/O. The design provided direct support for a variety of data structures, such as trees, sets, graphs, and relations. Garbage collection — the clearing out of portions of the data base that have been deleted and the resulting compaction possible — was performed by the hardware automatically.

CASSM provided support for variable length records and the capability to do operations on very large records — even larger than one track. Thus an entire tree could be dealt with as an entity. As a result, CASSM provided nice capabilities for dealing with a heirarchical or network model data base. It supported the relational data base model for restriction and projection, and had several solutions for the join problem. If the join field was quite narrow, the bit

Minsky solved this problem by introducing two heads per track, one for reading and one for writing. He also suggested the alternate approach of storing the control information in a separate, randomly accessible memory. This allows the record and the control information to be read, modified, then written. In addition to the considerable additional expense that this solution entails, [Hoagland 76] has pointed out that shrinking mechanical tolerances associated with increasing disk densities make this solution increasingly difficult and expensive. He predicts that the same transducer will be used for both reading and writing data on standard disks once the track density becomes sufficiently high. [Watson 74] proposed that the problem could be solved better by using two separate tracks and transferring the data of the "logical track" alternately between them with a buffer introduced for the needed delay time. [Copeland 74] later showed that the same problem could be solved using only three physical tracks for two logical tracks.

[Bush 76] proposed a novel solution by introducing a small random access memory for each track to store a control bit to be modified on a given revolution. On the following revolution the bit is stored at the beginning of each record and a new one can be generated. Thus all of the control bits are available at the beginning of the reading of the record. Only one control bit could be modified, however. The example above, for example, could be solved by setting to '1' the control bit for every record which qualified in field four. On the following revolution, the first field could be modified by reading the control bit, often called a *mark bit*, and modifying field one if the bit is set.

1.2.1.1.2. RAP

Ozkarahan, Schuster, and Smith [Ozkarahan 75], proposed a somewhat different head-per-track processor which they called the Relational Associative Processor (RAP). It was intended to perform efficiently operations only on a relational data base, and as such, limited all records to fixed length fields. Individual field lengths could be specified, but were constant for that field for all records. Capability was provided for inter-record processing to be performed efficiently through the introduction of a number of comparators for each disk track. A prototype was built and demonstrated using CCD's.

RAP implemented the join operation by the use of mark bits for each record and a number k of comparators for each cell. A set of records from relation A was selected and their join values broadcast to all the cells to be compared against the join value in each record of relation B. If any of the comparators registered a match, that record was sent to the controller to be joined with the corresponding record from relation A. Each record in A so selected had a mark bit set. This procedure was repeated, each time selecting records from A without the mark bit set, until no records remained. The number of operations was proportional the square of the number of records divided by k, so the speed could be increased greatly by the use of a large value of k.

A serious problem of the RAP design was the restriction that only one relation could be present in a given cell. This necessarily limited the parallelism, since many of the cells could not be used on

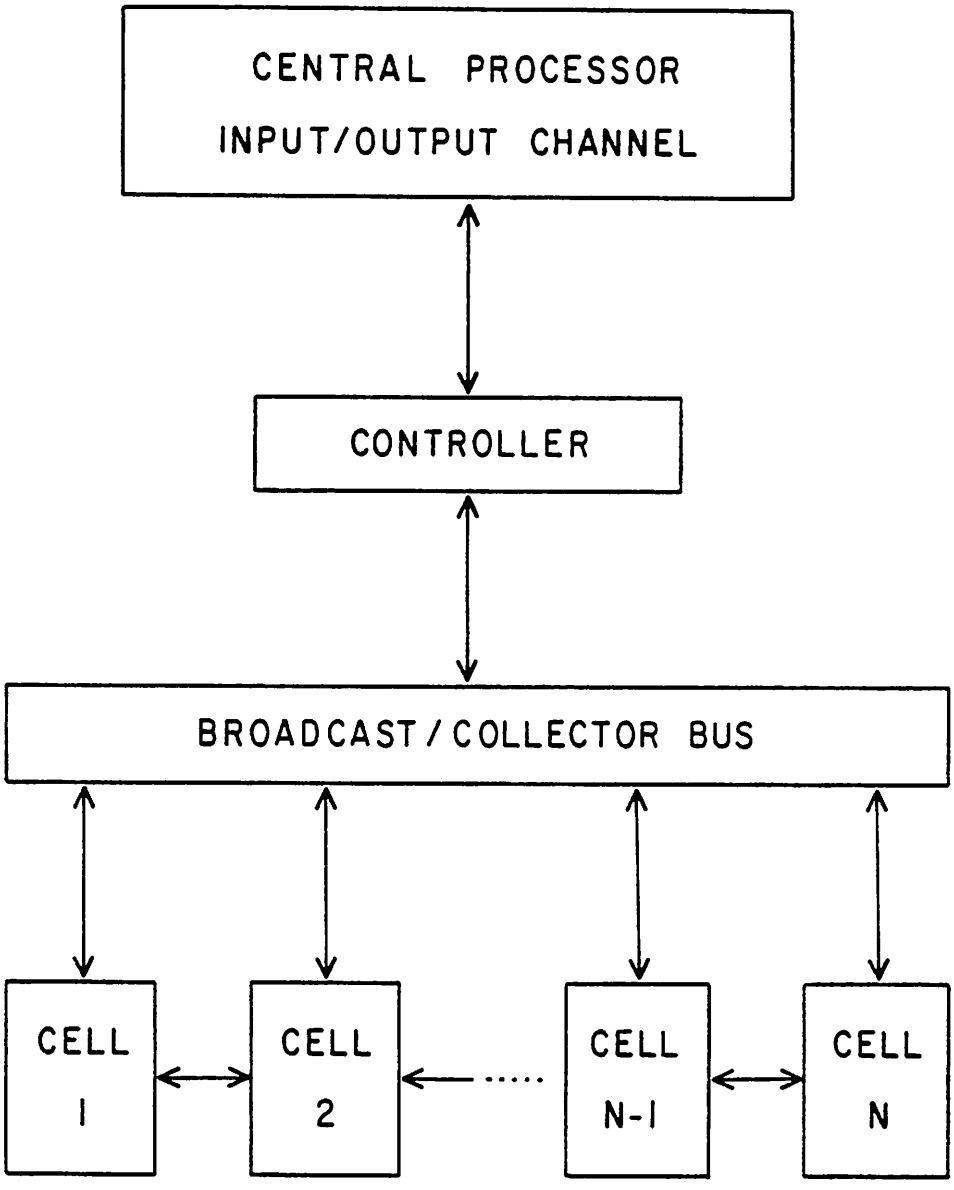

Figure 1.2. The Organization of CASSM. From [Healy 72].

sort order, something RAP and CASSM did not provide.

Although sort order is sometimes important, this organization otherwise presents an inferior solution. It requires close communication among all the cells, since all must participate in the extracting of a single record. In addition, it doesn't distribute the work load evenly. For a query where the qualification is on a single field, only one cell is busy, the others simply waiting for a command to transmit their portion of the record. The alignment problems associated with disk advances mentioned earlier also makes doubtful the viability of this approach.

CASSM, RAP and RARES all gain a significant advantage over the conventional DBMS because of the elimination of the need to support indices. As mentioned earlier, indices allow the rapid response to a set of queries. However, the modification of the data base, in many cases requires the modification of the indices as well. This is expensive, since it may require the shuffling of entries within one or more index. Thus a trade-off exists between rapid response to queries and rapid response to modifications to the data base [Moulder 73]. In a system where frequent modifications occur, but rapid access is still required, these designs have their maximum advantage.

All the above proposals suffered from a severe cost constraint. All were proposed using fixed head disk initially, and thus had limited capability to handle data bases larger than that which would fit on a fixed head disk. [Schuster 76] investigated the introduction of swapping techniques to handle data bases too large to fit on the RAP disk, and concluded that this was feasible under limited conditions, primarily that the queries exhi-

a particular query. Also, because of the limited buffering allowed in each cell, collisions could occur on the bus if many processors attempted to send results to the controller simultaneously. [Ozkarahan 77], however, showed that this would not be a problem unless the number of records in the result relation was excessive.

The potential collision problem on the buss might have been alleviated by performing projections in the individual cells, so that only the needed fields were sent across the buss to the controller. The cells did have the capability to perform the restriction, and did only send across the part of the record that was needed. Unfortunately, the design was such, however, that the transmission of a partial record still required the entire time slot, so the potential was not realized.

RAP was implemented with very primitive logic, resulting in considerable loss of performance. The cell logic consisted primarily of a shift register which had associated with it a comparator. This meant that all the time that the data was being shifted into the proper position, the comparator was idle. Then once the data was shifted to the correct position, the comparator had to be very fast to complete its operation before the data was shifted once again. Thus much of the hardware was not used very effectively.

1.2.1.1.3. RARES

Lin, Smith, and Smith [Lin 76] introduced a design which grouped tracks into blocks so that the data could be stored orthogonally across the tracks instead of sequentially along a single track. This design provided for the output of multiple qualifying records in

usefulness of such systems for the data base environment has been limited as a result of the high cost and the small memory capacity inherent in such designs. Because of this, a mass storage device with phenomenal bandwidth has been proposed to broaden their applicability. Linde *et al* [Linde 73] compared a hypothetical "Associative Processor Computer System" to a conventional IBM 370/145 system for search, update, and retrieval. In their proposal, they assumed the system to include a half-million-byte random access memory (which they considered large) with the capability of swapping data to and from the associative memory units at a rate of 1.6 billion bytes per second. To put this number into perspective, this is more than 1200 times the maximum bandwidth available through a conventional selector IBM channel at that time [Katzan 71]. Their conclusions were that they could provide substantial improvement if the mass storage was large enough to keep the system busy though they made no attempt to determine if their assumed memory size was large enough.

Edelberg and Schissler [Edelberg 76] investigated a slightly different idea called "intelligent memory". The system consisted of a string of processing elements (PE's). Each PE consists of a circulating loop of storage cells and some processing logic. Each PE could communicate with its two neighbors. In addition there was limited broadcast/response capability. They studied the structure for performance of sort, search, load/unload, retrieval, and update. Although substantial improvement was claimed over conventional structures, an unfortunate result was that loading the full memory took twice as long as the worst case sort time. This resulted from the necessity of loading the string linearly through one of only two ports.

bit significant locality of access. Under less ideal conditions, they concluded that the time required to swap the data between the RAP device and a secondary storage would result in poor performance. Thus the above proposals appear to have limited application. Although they may provide improved performance over a conventional system for a small system with frequent updates and rapid response requirements, they are not the general solution to the data base problem, since large data bases cannot be handled well in general.

1.2.1.2. Associative Processors and Associative Memories

Concurrently with the developments described above, others have pursued the use of an associative processor, such as the STARAN [Moulder 73], to provide data base management support. Like many of the tasks tried on STARAN, apparently the data base application was I/O bound, spending nearly all its time loading and unloading the associative processor. In a cryptic paper Love [Love 73] described a hybrid associative processor using MOS shift-register memories. Each associative processor, however, contained only 64 bits of circulating, quasi-associative memory. Furthermore, a crossbar switch was proposed to connect the individual associative memories to a proposed MOS static shift-register bulk memory. Since a large data base requires a large amount of memory and this system included a large amount of additional hardware, the cost per bit of such a DBMS would have been exorbitant.

Berra [Berra 74] has concluded that the enormous arithmetic capabilities of machines such as STARAN are not needed for data base applications and that simpler associative memories, such as the AM [Moulder 73], developed by Goodyear, can support a DBMS adequately. The

demonstrate superiority to overcome the natural reluctance to change. Let us examine some of the common differences between the proposals described above and the conventional DBMS to see where that superiority might appear.

1.2.2.1. The Role of Auxiliary Structures in DBMS

Conventional data base management systems usually provide numerous access methods for the data in the data base to allow most or all of the anticipated queries to be answered rapidly. Because of the redundancy in the information, these structures often have large memory requirements of their own, occasionally even greater than the data base itself, but they pay off handsomely when conditions are right. If indexing capability has been provided for the appropriate fields, it is often possible to find the desired information in a large data base with a very small number of disk accesses. Under such conditions, the DBMS performance is usually quite satisfactory. However, there is a cost. Each time an update to the data base occurs, the index structures must be updated as well. The more auxiliary structures existing, the more expensive (and slower) is an update. The result is a trade-off: Quick response on retrievals means slow response on updates and *vice versa.*

When quick response is not a prime criterion for retrievals, or when updates are relatively infrequent, conventional DBMS's often perform satisfactorily. If the query being handled is of a class that has been anticipated, so that the proper auxiliary structures have been generated, the response frequently is quite good. If any of the above assumptions are false, however, performance will likely be unsatisfactory. It must be remembered, too, that queries can always be formulated for which the needed auxiliary structures are not available. Under such circumstances,

1.2.1.3. The Back-end Machine

A third approach to improving DBMS performance has been the introduction of the so-called "back-end" computer. [Canaday 74]. The concept is that the DBMS functions is removed from the host processor to a separate, possibly general purpose, machine which has no other function. The earliest reported work in this area was done at Bell Laboratories in an experimental system called XDMS [Canaday 74]. Numerous advantages were postulated, in particular the following:

(1) Economy through specialization — smaller, simpler programs and the use of a processor particularly suited (or modified to suit) the data base environment.
(2) Shared data — multiple hosts can share the data because of the well-defined interface.
(3) Protection and security — the physical isolation of the data base system makes it susceptible to fewer failures. Also, the host has an opportunity to detect and act quickly upon failures in the back-end and *vice versa*.
(4) Transportability — once developed, the system could be attached to other hosts with a relatively small investment.

The Bell Laboratories study concluded that for simple commands, costs were more or less equivalent to that of a conventional system being developed concurrently, but for complex commands the cost savings were one to two orders of magnitude. In addition, Canaday *et. al.* reached the conclusion that the other benefits mentioned were realizable, and in particular, that back-end DBMS was not only feasible, but that it would substantially improve the cost/performance even if the back-end processor were an off-the-shelf minicomputer.

1.2.2. Comparison to Conventional Hardware

The traditional DBMS system has benefited from a large amount of commercial investment, and a novel approach to the problem must be able to

1.2.2.2. Set Processing vs. Record-at-a-time

Many authors [Healy 72] [Moulder 73] [Ozkarahan 75] have noted that quasi-associative devices offer a greater improvement over conventional models in the environment where the answer to a query is a large set of records, not just a single one. This results from the fact that the associative device takes very little longer to retrieve a set of qualifying records than to find just one, since each record must be examined to see if it qualifies regardless. This is in marked contrast to conventional systems where, unless the qualifying records happen to cluster properly, the retrieval time may be directly proportional to the number of qualifying records. This concept, known as *set processing*, is popular particularly with the proponents of the relational model, where all the information is explicit, and therefore the concept of navigating through the data base is not so useful. Thus the query languages developed for relational data base systems [Chamberlin 74][Held 75] generally encourage the types of queries for which the quasi-associative devices perform best.

1.2.2.3. Storage Requirements

Any solution utilizing a modified head-per-track disk to store the entire data base will be more expensive than either a moving head disk or an unmodified head-per-track disk. This is clear because of the significantly higher cost per bit of the fixed head disk, and the increased cost of the controller to allow the simultaneous reading of all heads. Thus the cost for a given size of memory, except in exceptional circumstances, will be higher for the special purpose system than for a conventional one, even taking into account the reduction in storage requirements due to the elimination of the auxiliary storage structures.

the only solution for a conventional, sequential computer is to search sequentially through the entire collection of data records. For many queries it is necessary to make multiple passes through the file. For the conventional organization under such circumstances, the file must be brought in serially, possibly several times, and one firm limit on the response to the query is the time required to read the file in. On the other hand, the head-per-track architectures are capable of a great deal of parallelism because all the heads are being accessed in parallel, potentially increasing the bandwidth in proportion to the number of read heads.

The head-per-track proposals and the associative processor proposals all have a common trait: they have eliminated the auxiliary structures. Indeed, much of the improvement claimed for these proposals is the elimination of the need to maintain these structures [DeFiore 74], [Ozkarahan 77], [Maller 78]. While these solutions are appealing, there is not a great deal of hope for dramatically improving the performance where conventional systems already perform well. On the contrary, examples have been constructed where the head-per-track solution would be much slower [Ozkarahan 77][1], [Hawthorn 79a]. Thus it would seem that a design which claims to improve the performance of DBMS's in general will be required to support some auxiliary structures. Anything less would start with a giant step backward unless it can somehow improve dramatically on the performance of conventional models at the tasks they perform well.

[1]See, for example, Fig. 7, page 190.

1.2.3. Some Recent Proposals

In the last few years the categories described above have become less distinct. In particular, although little serious consideration has been given to implementation of a head-per-track "associative disk" as discussed above, some research has appeared using moving-head-disk [Banerjee 78], [Babb 79] and much has been made of the possibility of performing a similar function using one of the emerging technologies, such as CCD's [Schuster 78], [DeWitt 78], bubbles [Edelberg 76], [Chang 78], [Liu 79], and EBAM's [Hsiao 77]. Until the recent consensus that CCD's are not a viable technology as demonstrated by the fact that no new devices have been announced for several years, the most considered was the use of CCD's. The idea, which also can be applied to bubbles, is to replace the disk with some form of circulating memory, either a standard component or a special device designed for the purpose. Such a system, though conceptually similar to a head-per-track disk, in implementation is much more like an associative memory, with logic distributed throughout the memory. For either bubbles or CCD's the logic might actually be placed on the same component with the memory. Such designs are often thought of in the context of a back-end processor, or even as a back-end to a back-end. In addition, other authors [Stonebraker 78] [Madnick 75] [Ames 77] have proposed the use of multiple processors to perform the various functions of the DBMS. These may be as sophisticated as small mini-computers [Stonebraker 78], or as small as early microprocessors [Ames 77].

Because they have the most in common with the ideas presented here, some of these designs will be examined in more detail. Schuster, one of the originators of the RAP design, has helped Intel Corporation in the develop-

Furthermore, the cost per bit of memory will inevitably be higher, maybe an order of magnitude or more, and the only hope for developing an effective system is through superior performance. On the other hand, cost considerations dictate that head-per-track systems are out of the question for moderate to large data bases except for those of pecuniary indifference.

As mentioned earlier, [Schuster 76] proposed that a head-per-track device such as RAP could handle a data base much too large to fit on the head-per-track disk. The technique is to swap into the head-per-track disk only those portions of the data base which are being accessed. The study concluded that such was feasible if enough locality exists within the data base queries. That question remains unanswered. They have suggested further that RAP might be useful for holding the catalogues and indexing information of a data base too large to fit on the RAP device. This auxiliary information may be seen as a small separate data base, having many of the requirements under which RAP's performance is projected to be particularly good. The potential places, then, for such special-purpose machines as described would seem to be the following:

(1) Applications where performance of conventional systems is inadequate because of the need for quick retrieval response in an environment of many updates.
(2) Applications where the queries are so varied and unpredictable or so complicated that the performance of conventional systems is inadequate.

Both types of applications are more attractive for quasi-associative designs if the data base is not excessively large. Quite possibly the latter application might prove attractive if substantial improvement over current capabilities could be demonstrated.

1.2.3.2. DBC

Probably the most ambitious attempt at a back-end computer specifically designed for database applications is the database computer (DBC) under development at Ohio State University [Banerjee 76], [Kannan 76], [Hsiao 76]. This design proposes the use of modified, moving-head disks for the mass memory (MM) of the database. In addition, a "structure memory" (SM) is proposed for the storage of auxiliary information about the database. Hsiao and Banerjee predict, however, that it should be much smaller than typical auxiliary memory requirements with current conventional designs [Banerjee 79]. They propose that this memory be built from one of the emerging technologies — bubbles, CCD's, or electron beam addressable memories (EBAM's).

The modification to the moving-head disks is to enable the use of all heads on the disk simultaneously. It is claimed, not entirely convincingly, that this is "feasible and relatively low in cost" [Banerjee 79], and the models presented assume that storage on such devices is no more expensive than conventional drives. Such a device is inevitably more expensive, however, since a significant amount of electronics must be duplicated which otherwise could be shared.

One might further ask if such modifications are really necessary, since only one disk is ever accessed at a time. It would seem that equal parallelism might be obtained without the modifications by appropriately placing the data so that an entire bank of disks could be read concurrently. While this makes the disk controller somewhat more complicated, it can be argued that disk controllers are becoming cheaper both absolutely and relative to disk drives because of progress in VLSI technol-

ment of a proposed data base computer using CCD's. Since this work has been presented only in the most limited way, and since the present author has had the privilege of participating in the design of the Intel version of RAP, chapter two will deal with some of the design decisions made and the trade-offs considered. Copeland, one of the participants of the CASSM study, has continued his work at Tektronix Corporation [Copeland 77] and has also proposed a CCD system.

1.2.3.1. ECAM

Anderson and Kain [Anderson 76] have described a "content-addressed memory designed for data base applications". Although it is claimed to be independent of storage technology, the design assumes CCD's. It is intended as a back-end machine for one or more host computers. It consists of a master computer which is in fact a standard minicomputer, a custom-designed controller, and a large number of modules. Each consists of a 4K bit serial memory and enough logic to perform associative functions on the memory. The system is intended to be expandable up to 10^9 bits.

Directories are maintained on the contents, apparently primarily to keep track of the data format, but no other auxiliary structures are mentioned. Though the design would be quite expensive because of the large requirement for semiconductor memory, the claim is made, without substantiation, that the system will operate "roughly 200 times as fast as a conventional database system on a large commercial mainframe." [Anderson 76, p. 194]

Functional decomposition is claimed to be an effective way to convert high level concepts to requests to the physical information structures. He proposes the use of a separate set of processors for each functional level, which he claims simplifies implementation, enhances modularity, and makes possible the introduction of specialized processor functionality and parallel execution of lower level primitives. He proposes the implementation of a *pipeline* through the use of queueing facilities and internal multiprogramming, and claims that inter-processor communication is made relatively simple and efficient by the hierarchal structure of the function decomposition.

Madnick states that "most practical applications result in clustered references such that during any interval of time only a subset of the information is actually used, especially when you consider the use of indexes and other control information." To take advantage of this *locality of reference*, he proposes physical decomposition to create the effect of a large, high-speed storage by means of a hierarchy of memory blocks with a range of speeds and costs.

He stresses the asynchronous nature of both the functional and physical decomposition for maximum parallelism. While he is quite vague about any implementation, he does propose a microprocessor complex to implement the hierarchal decomposition.

1.2.3.4. DIRECT

[DeWitt 78] has proposed implementing INGRES [Held 75], on a unique back-end computer attached to a small host whose responsibility is the user interface. See Fig. 1.3. The back-end consists of the following:

ogy.

In the DBC system, while all the heads on one disk are used simultaneously, the only parallelism planned for the multiple moving-head disks is the expectation that, while one is dumping its data to the track information processors (TIP), other disks will be positioning themselves for future operations. The question arises, how much of the time does a disk spend preparing to retrieve data as opposed to the time needed to transmit it? In the case of the DBC, one revolution, typically 17 msec, is the time to transmit the data sought. Since a seek is typically in the same range, and including average latency of 8.5 msec, it would appear that the best possible level of parallelism within the moving-head disks is quite low, certainly less than 5. Dividing the disks into banks, however, each bank having as many disks as there are tracks per disk, and reading all the disks in a bank in parallel, would provide the same data rate with unmodified disks. If there are enough drives to create a reasonable number of banks, the performance should be as good or better than that obtained with the expensive modified drives. Thus it appears to be possible to build a large system with the same performance as DBC but using conventional disks, deriving the parallelism from the simultaneous access of many disks.

1.2.3.3. INFOPLEX

[Madnick 75] [Hsiao 77]. has proposed a system called INFOPLEX which addresses what he calls "hierarchical decomposition." He claims that two major types of parallelism to be exploited in an *information system:* functional and physical.

(1) a number (n) of query processors, each actually a PDP 11/03 with 28K words of memory.
(2) a number (m) of CCD memory modules, each containing a portion of the data base currently being accessed.
(3) a mass storage device to store the database.
(4) an interconnection matrix to connect the query processors and the mass storage device to the CCD memory modules, and
(5) a back-end controller to control information flow, assign processors, page data between the CCD modules and the mass storage device, and resolve concurrency conflicts.

The proposed design for the interconnection matrix is bit-serial, with each memory module broadcasting its data serially and continuously, in synchronization with all other modules, to all query processors. The processor wishing to access a particular memory module simply selects that module and reads the data. DeWitt argued that frequently the query processor could immediately begin processing a page from the memory module. Since it only holds one page, the processor could start at whatever point on the page it found the module broadcasting, rather than waiting for the beginning of the page, then reading the entire page in. If the processor has to process the entire page anyway, it may start almost anywhere. This argument ignores the point, of course, that many queries will only be seeking a single record, with its address known to the processor. When this occurs, a large wait time is encountered for the processor, on the average, half the time it takes a module to broadcast its entire memory, which is .012 seconds. A further problem results from the fact that page sizes will be forced to grow as semiconductor technology increases storage density on a single device.

The cross-bar switch proposed to implement the interconnection matrix has some serious drawbacks. The DIRECT proposal concedes that other data base techniques are superior for small values of m and n, but

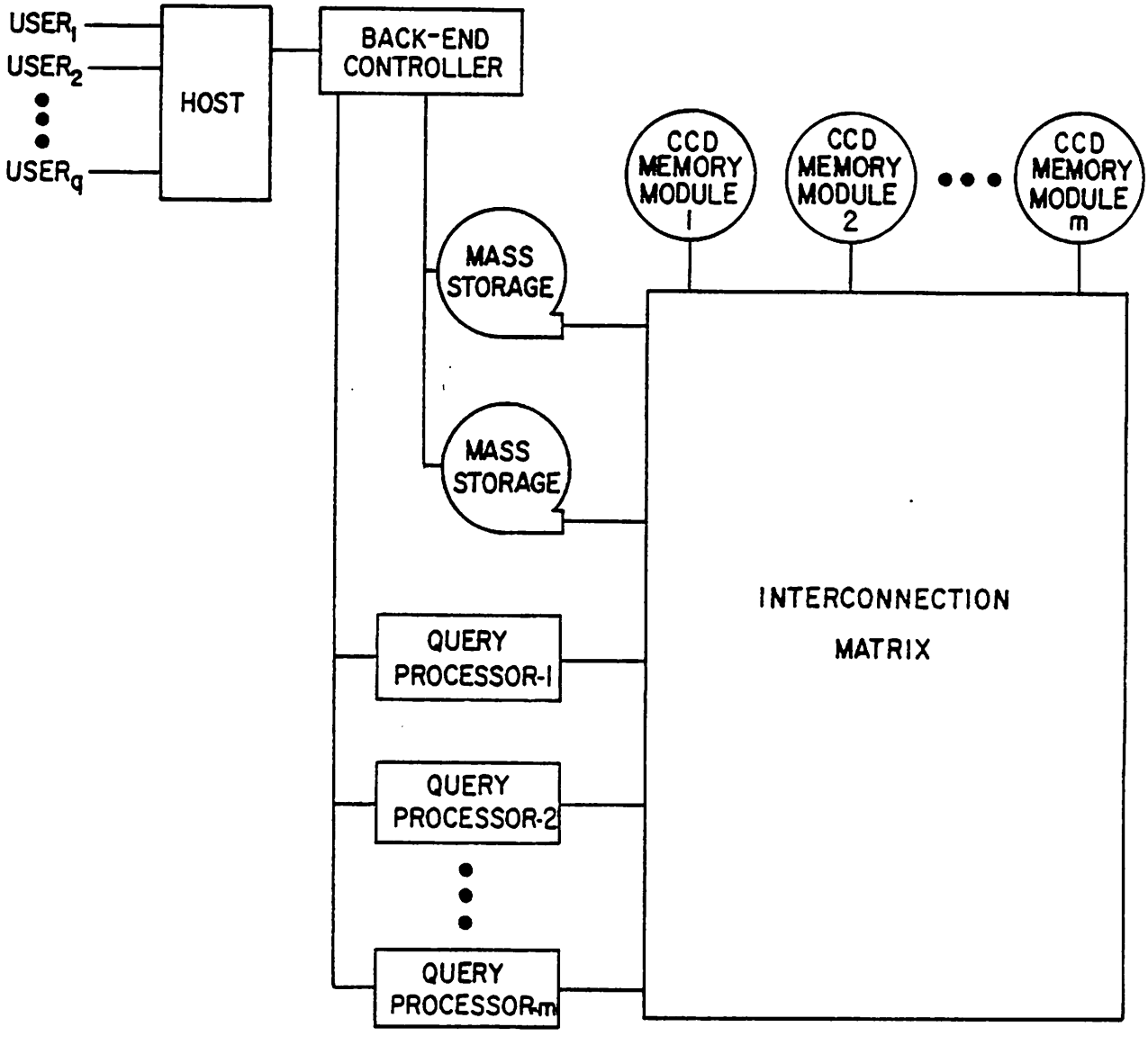

Figure 1.3. The Organization of DIRECT. From [DeWitt 78].

enough to keep the rest of the system busy.

1.2.3.5. MUFFIN

Stonebraker [Stonebraker 79], one of the originators of INGRES [Held 75], proposes a distributed data base machine to implement INGRES. He proposes a network of "processing nodes," called *A-cells* (application program cells) and *D-cells* (data base nodes). Both kinds of cells consist of "ordinary processors," though he claims they should be different because of a different optimization of the instruction set.

The D-cells are totally dedicated to the execution of run-time data base system code and there is no general purpose operating system present. The cells may or may not have a disk system attached, and their main storage is used for a "cache" for the disk, if it exists. The message protocol observed by the D-cell is carefully restricted to minimize the overhead associated with message traffic. As a result, the D-cells never originate messages, but only respond to them.

The A-cells will run a conventional operating system and include some support peripherals. They will handle the user interface and the application program, and are responsible for parsing the query, modification for support of views, and enforcing integrity constraints and protection [Stonebraker 75]. They will coordinate the execution of the INGRES transactions by making calls to the D-cells.

A sophisticated page replacement algorithm for the "cache" of the D-cell is proposed which can take advantage of the special knowledge available about future references to the same page.

claims that the cross-bar switch is the best interconnection for larger values. The network complexity grows more rapidly than either m or n, however. As a new query processor is added, not only does the switch become one unit wider, the number of loads placed upon the CCD transmitter is increased by one. More seriously, the number of places from which write data can be received increases by one, necessitating more gating logic at each memory module. The addition of a new memory module, on the other hand, requires more inputs to each selector for each query processor, in addition to the general increase in the switch size. To make matters worse, expandability is clearly limited beyond the initial design.

In justification of the interconnection matrix, DeWitt argued that greater parallelism can be obtained by allowing both inter- and intra-query concurrency. This argument assumes that all the processors cannot be kept busy with only intra-query concurrency. It will be shown in chapter two that this assumption is false. Thus little is gained by the generality of the interconnection matrix and a seemingly superior implementation would allow each query processor to access only a small subset of the memory modules. While the claimed superiority over the original RAP design due to inter-query concurrency may be valid, the implementation of the relational join operation on DIRECT is clearly inferior to RAP. One of RAP's major contributions is its capability to handle this operation well.

The query processors are so slow in the initial implementation that the interconnection matrix has been reduced to a single, multiplexed bus [DeWitt 78a]. This solution creates a bottleneck if the processors are fast

functions efficiently. The concept, which may have broad application, uses a hashing technique to perform the join operation. This idea, which is very similar to the approach taken on CAFS, to be described next, is a powerful concept which may offer significant advantages over any other reported method.

1.2.3.7. CAFS

A data base machine actually being implemented in England is the Content Addressed File Store (CAFS) [Coulouris 72], [Maller 78], [Babb 79]. This is the first reported commercial product and appears to have had a great deal of influence on the design of LEECH. It is an integrated design which has attempted to address the issues of security, privacy, ease of access, data integrity, and consistency in a system with extremely high performance. [Maller 78] argues that to overcome the inherent problems posed for a large DBMS,

(1) an effective method must be provided for associative access to files of data, and

(2) there must be a method for correlating the data retrieved from different files without overwhelming the facility with massive sorting and merging.

In the terms of the relational model, he says this implies that the operations of selection, projection and equi-join must be implemented directly.

CAFS proposes the use of moving head disk, with logic per head for quasi-associative searching capability.

CAFS comprises five principal sub-units (Fig. 1.4):

A cluster of A-cells and D-cells, called a *pod*, are connected together by means of a bus or local network. The pods may be connected by means of a *gateway*, through an A-cell, utilizing a lower speed communication link.

Stonebraker stresses the importance of the bus structure within the pod because of the importance of the broadcast capability — the ability to send the same message to a number of nodes. This is the only reason offered, however, and is heavily dependent on the assumption that "the cost to send data to all sites is equal to the cost to send it to any site" [Epstein 80]. He also rejects the use of identical hardware nodes connected in some structure because of a large cost savings claimed by only providing needed functions in the A- and D-cells. He further states that the two kinds of cells will probably have different instruction sets. Since both are to be built from available conventional, general-purpose systems, it is not clear how much different the instruction sets will be, but the two kinds of nodes clearly will contain very different software.

Stonebraker makes an argument against the use of a hierarchy of processors because he says such an organization "implies that concurrency control and crash recovery are at least partly centralized." He further points out the danger of a bottleneck at the top of a hierarchal structure if a straightforward approach is used to implement the join operation.

1.2.3.6. LEECH

McGregor, Thomson, and Dawson [McGregor 76] described a back-end system being developed in Scotland intended to support all data models but especially the relational view. Although the hardware description is very sketchy, a novel concept is proposed for performing certain

(1) Control Processor
(2) Direct Access Unit
(3) Associative Searching Unit
(4) Record Retrieval Unit
(5) File Correlation Unit

The Control Processor is a conventional minicomputer whose purpose is to do resource management and task scheduling. The Direct Access Unit performs the standard functions required for the reading of a disk, but can read multiple heads in parallel.

The Associative Searching Unit executes concurrent search tasks on a stream of data coming from a number of disk channels. It performs the qualification for the query, determining if each record meets the criteria stated. It is able to perform its function at the speed it is delivered from the disks, so no block buffering of the input data is required.

The Record Retrieval Unit collects the records for the result. It performs the projection by saving only those fields which are wanted. In addition, it performs the restriction by writing over those records for which the Associative Search Unit find no match.

The file Correlation Unit provides an efficient mechanism for the evaluation of joins and the elimination of duplicates. The technique used employs a set of 1 bit wide memories of size 256K bits in the experimental version. The memories are used to store in compact form the set of values present in some join field of a relation. To demonstrate the power of the File Correlation Concept, consider the join operation described earlier using the telephone directory. If the field has a width of 18 bits or fewer, the field itself can be used as an address for one of the memories. Suppose that the phone numbers are sufficiently short that they can be

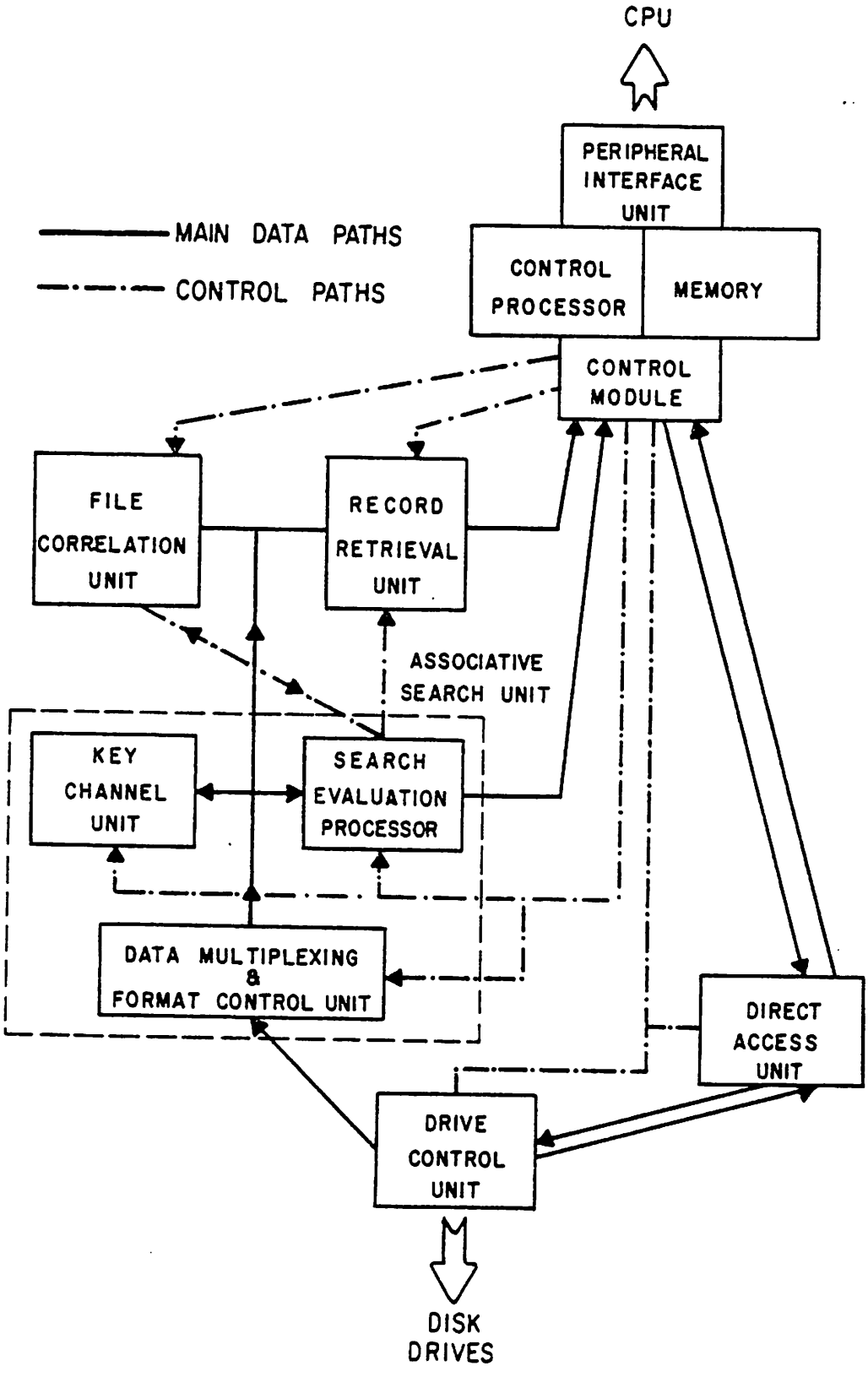

Figure 1.4: The Organization of CAFS. From [Maller 78].

this value is added to a new list saved in the separate storage area. After the second pass, all those entries in the separate storage area not occurring more than once are discarded, and the remaining values are all those occurring more than once.

Frequently at least one relation must be scanned twice in order to perform the join operation. This results from the request to provide other fields of records participating in the join. For example, if the telephone query were to list all the names of people who had phone numbers listed more than once, it would be necessary to scan the relation a second time to find all the occurrences of a given listing.

In the general case of the join algorithm, the relations are scanned a second time, and the records qualifying are retrieved along with some which appear to qualify as a result of a collision in the hash table. In order to eliminate these extra records, called *phantoms*, the join must now be performed again. This is usually a simple task, since the number of records involved is usually quite small. Thus an acceptable level of phantoms occurring is quite high, say 20 percent. The important point is that no records are lost. Only false records occur and they are purged in the subsequent operation.

The same physical hardware is capable of eliminating duplicates after a restriction by the use of similar techniques, but in that operation there is no way to guarantee that unique records will not be thrown away as a result of collisions. Again, this probability can be made arbitrarily small, but not zero.

encoded in 18 bits. Then the following method can be employed to find all phone numbers listed more than once:

> The 1-bit memory is cleared. The telephone directory is scanned sequentially, and for each entry a '1' is stored in the location with the address specified by the phone number. Before it is stored, however, the memory location is checked to see if it is already a '1'. If it is, then it may be assumed that this number has already occurred in the directory and this phone number is returned.

Thus, in a single scan of the file, the join operation is performed.

If the numbers are sufficiently large that they cannot be encoded in 18 bits, then a more elaborate approach is required. The phone number field is hashed to produce an 18-bit field which can be used to address the memory. Now if a '1' is found in a memory location, it is not safe to assume that the phone number has been seen before, since more than one phone number will hash to the same location. This is overcome by using several of the 1-bit memories, each with a different hashing function. Unless all the 1-bit memories show a value of '1' it is safe to assume that the number has not been encountered before. If all do show a '1', it is still possible that this is the result of a number of collisions, but Babb has shown [Babb 79] that this probability can be made arbitrarily small by the use of a sufficient number of hash tables (1-bit stores). However, for those who are unable to accept a non-zero probability of error, the result can be guaranteed with a second pass. On the first pass, each phone number thought to occur twice is saved in a separate storage area. After the pass, the 1-bit memories are cleared, and the numbers saved are hashed and entered into the memories. The separate storage area can now be cleared. A second pass is made, and each value is hashed. If a value occurs for which all hash table entries are '1', then

in a single integrated circuit, an idea which may soon be realizable. This idea will be pursued further in a later chapter.

1.3. Lessons to be Learned

Several issues have been raised in the cited works which should at least be considered in any future design. These issues are discussed next.

1.3.1. The Importance of Auxiliary Structures

Although auxiliary structures are cumbersome and inelegant, they do provide some powerful capabilities at relatively low cost. For many common DBMS applications there is evidence [Ozkarahan 77], [Hawthorn 79a] that none of the above cited proposals will offer dramatic performance improvement. For those applications where the data base shows little dynamic fluctuation, where few updates affecting the auxiliary information are required, and where all the queries handled are of a class anticipated when the data base was generated, the need to exhaustively search the data base never occurs. For many applications it would appear, then, that the continued use of the auxiliary structures is warranted. A design which claims any generality must therefore either demonstrate that these structures can be eliminated, or provide for their support. With a few exceptions, (e.g. [Berra 76], [Maller 78], and [Banerjee 78]), in the past these structures have been ignored.

1.3.2. Set Processing

As mentioned earlier procedural languages encourage navigating the data base, with the result that the inherent parallelism within a query is hidden. Therefore, set-oriented, non-procedural queries will inevitably become relatively more attractive. Thus any future data base machine should be designed to encourage the use of non-procedural languages.

1.2.4. List Merging Processors

Some interesting research has been performed regarding the design of special hardware for a related problem, namely document retrieval systems such as EUREKA [Hollaar 77]. Because these systems generally have complex inverted files to assist in the retrieval of documents by keywords, the results are significant also for more structured data base systems, many of which also maintain inverted files.

Stellhorn [Stellhorn 74] argued that the most time consuming task in many document retrieval systems is the standard set operations on the elements composing the inverted lists. This is done to facilitate Boolean expressions qualifying the request. He designed a special unit to merge two lists rapidly using the Batcher even-odd sort [Batcher 68]. Although the method required[2] $N \cdot lgN$ merge elements to process two lists of length N, a number that grows unacceptably for hardware, a technique was demonstrated for partitioning the lists and merging by parts. While this would indeed be a very powerful tool, it is doubtful that a backing memory could keep such a device busy.

Hollaar [Hollaar 75] designed a merge unit capable of merging two lists rapidly by the use of a special list merging processor which used a single, high-speed comparator to merge two lists rapidly. He demonstrated that, by combining these in a binary tree structure, very complex Boolean expressions could be rapidly analyzed.

Both of these designs included considerable discussion of implementations using currently available integrated circuits. The latter appears to be a reasonable idea if the list-merging module could, for example, be incorporated

[2]Throughout this document, lg means $\log_2$.

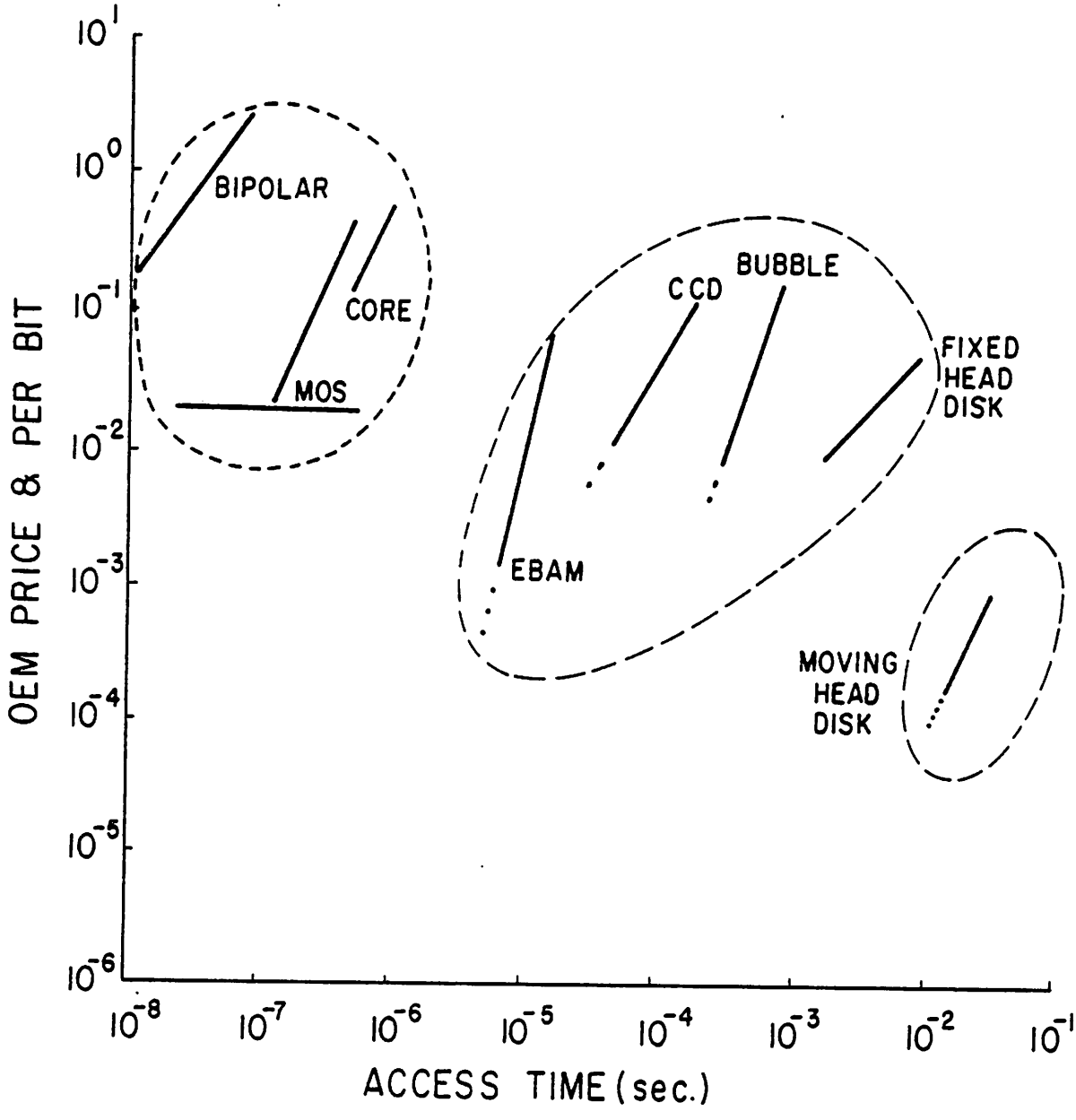

Figure 1.5. The Speed/Cost Curve for Computer Memory. The projection of price/performance for various memory technologies for the period 1977-1987. From [Hsiao 77].

1.3.3. The Relationship Between Processing and Storage

Assuming that there are multiple units capable of performing processor functions and multiple storage units, a fundamental issue is how these units should be interconnected. At one extreme is the associative memory, which has Boolean logic associated with exactly one memory unit, possibly as small as one bit. At the other extreme is the general network, such as DIRECT [DeWitt 78], which allows any processor to access any memory unit. While the former requires much logic, which is poorly utilized, the latter required an expensive, complex switching network to connect logic dynamically to any desired memory cell. So the choice exists between the need for extra processing power which is not utilized and the need for complex switching circuits. In this sense, memory is the only intractable requirement, and of course it may be a hierarchy. It is suggested here that an intermediate scheme, minimizing the processing logic requirements but using it well, while also minimizing the interconnection logic will offer better cost/performance than either extreme.

1.3.4. The Cost of Storage

The dominant technology for data base storage and retrieval is moving head disk [Kerr 79]. While this medium has a low storage cost, it also has long access times and slow data transfer rates relative to main memory speeds. Most of the proposals suggest the use of higher speed technology or modifications to the moving head disk, either of which, needless to say, is also more expensive than standard moving head disks. The assumption, then, is that one major performance limitation is the inadequacy of the storage medium. While moving disk does have serious limitations, no emerging technology appears to be challenging its position on the speed/cost curve, Fig. 1.5.

CHAPTER 2

The Intel/RAP Machine

In 1976, Schuster, one of the designers of the Relational Associative Processor (RAP) at the University of Toronto, approached Intel Corporation about the possibility of implementing RAP using charge-coupled devices (CCD's). Intel subsequently began an investigation into the feasibility of such a system. The present author was fortunate to become involved in this research.

2.1. The Relational Associative Processor

As mentioned previously, RAP was a design for a relational data base management system to adapt a fixed head disk to appear as an "associative disk". RAP, Fig. 2.1, consisted of a controller, a set function unit to perform statistical functions, and an array of *cells*. Each cell consisted of one track of a fixed-head disk and special logic to perform simple operations on the data coming off the disk. The cells could send or receive data from the controller through a common bus. The controller in turn communicated with the host to which it was attached, using a relatively high level language interface, such as SEQUEL [Chamberlin 74]. Thus the entire data base could be searched quickly with complex selection criteria, but without the use of indices.

There were several reasons for rethinking the design of RAP. Intel's interest in the RAP project came from the possibilities for using semiconductors memories, not magnetic disks. Also, as is typical of university projects, the original RAP did not use state-of-the-art technology. The comparison logic was fairly complex, and no LSI parts were used.

Thus all the proposals mentioned here are proposing more expensive memory, hopefully resulting in better performance. This question is at the heart of the issue, however, and must be investigated further. Must all memory be higher speed and higher cost? Or can the hierarchal memory concepts which have emerged be used, requiring only a small amount of more expensive memory? This dissertation will investigate these questions further.

1.3.5. The Nature of the Query

Many data base management systems respond to queries almost exclusively through the use of the auxiliary structures constructed for that purpose [Gray 78]. They may have little or no reason ever to scan the entire data base to answer queries for which the best access methods do not exist. Under such circumstances, there is little hope that the ideas presented in this chapter will offer significant improvement in cost/performance. However, for those applications where the queries are not anticipated, or for any of several reasons, the appropriate access methods are not available, these proposals offer the hope of significant performance enhancement. At least one manufacturer believes that such an approach is feasible [Babb 79]. If such applications prove to be enhanced by the ideas presented here, it is reasonable to expect a dramatic growth in the use of DBMS's in a plethora of new applications.

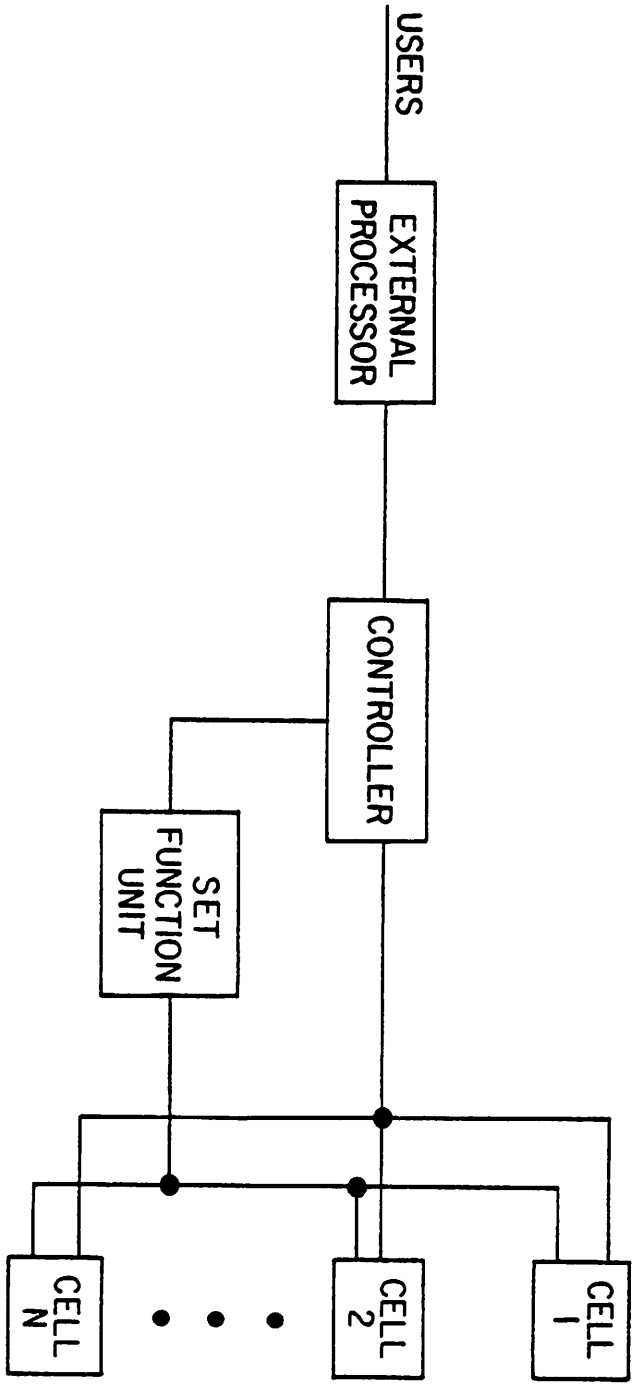

Fig 2.1. The Organization of RAP.

2.2. Problems with RAP

A fixed-head disk has a fixed rotation period. A given location is readable only once during that period. Furthermore, nothing else can be done when the data under the head is not wanted. Thus the original RAP hardware, made in Toronto, had to be very fast to be able to do the comparisons quickly when the data had arrived, yet wasted much of its time waiting for the data to get in the right position. On the other hand, when the data did not fill the disk, the time waiting for the data to come under the head was wasted. It is not desirable to have the tracks very full, on average, because the data may grow. This results in a considerable loss in potential processing efficiency.

Another problem of the Toronto design was the problem of collisions when two or more cells found records simultaneously and both needed to be transmitted to the controller. The original design was such that when two cells tried to transmit a record to the controller, one succeeded and marked the record transmitted. The other cell, however, failed, and had to record that fact and try again on the following revolution. Thus in the case where all the cells had a record to transmit at the same time, RAP required as many revolutions as there were cells. Nothing can be done if all the records in the relation need be transmitted to the controller, since the bus was busy all the time. However, in many cases a collision occurred, forcing an extra disk revolution, even when a small number of records was being retrieved. Although a small amount of buffering might help considerably in this case, the bookkeeping would be non-trivial, and it was argued that collisions would be rare, anyway.

A more serious problem was the fact that the minimum time for even a trivial search was one revolution. This limited speed, and made it pointless to put more than one relation to a cell. Thus only a small number of the cells would be

available, even those not yet announced, were not nearly able to use this kind of bandwidth. In fact the speed match was off by several orders of magnitude and considerable effort was made to design a cell which could utilize this bandwidth.

A basic difference from the Toronto design resulted from the fact that the CCD's intended for this project could be halted for a time. It was therefore possible to stop at one page, process the data as long as needed, then skip to the next. Since a page could be very quickly passed over, little time was lost if much of the data was unneeded. Thus the relations could be partitioned across the cells evenly, i.e., for each relation, each cell was to contain the same number of pages, within one page. Now all the processors could be kept busy nearly all the time, producing a high degree of parallelism.

Unlike the original RAP design, in which all the cells were working in lockstep because of the minimal buffering from the disk, the Intel design assured that the processors would not be in lockstep. Each cell had its own set of CCD's and advanced the page when it was ready. The asynchronous nature of the cells guaranteed that collisions could be handled more efficiently. The worst that could happen would be that one cell simply stopped working while waiting for the bus to send retrieved records to the controller. A further advantage of the greater intelligence given to the cells was that the operations could be specifically tailored for the current query. That is, before each operation, the program to be performed by each cell could be broadcast to all the cells over the common bus. A complex program could be broadcast in a time insignificant in comparison to that for the resulting query.

2.4. Fixed v. Variable Length Records

A major consideration in the design was the trade-off between storing records of fixed-length only, or allowing the length to vary. The argument for variable

busy at any one time, namely those containing the relation being addressed.

Finally, no provision was made for recovery from failures. Magnetic disks have a relatively high error rate, and their use in modern computers is only possible because of error detection and either correction or retry. No provision was made for errors in RAP, an issue which would almost certainly make such a product commercially infeasible.

2.3. A New Technology

The opportunity to employ a new technology in RAP allowed a fresh look at the RAP design. In particular, it was felt that some of the unique characteristics of the Intel CCD device could best be exploited by certain changes in the RAP physical structure. The Intel 64K CCD was a unique device in that it was composed of a large number (256) of loops, each being relatively short (256 bits). Thus at any position, 256 separate bits were randomly addressable at a very high data rate (2.5 Mhz). Furthermore, the device could be halted at one loop position for an extended period of time, possibly 1 ms. or longer.

A prime consideration in the new design was reliability. Although semiconductor devices are in general no less reliable than magnetic media, they nevertheless do exhibit random loss of data on occasion. Intel wanted to be able to handle not only random single-bit failures, but also the case where an entire device failed. Clearly this meant that the error correction code word had to be spread over many devices. Thus a cell was defined to be a multiple of 72 devices (or possibly even 143), each of the 72 devices having one bit of each code word. This dictated a minimum cell size of 524k bytes. As a result of this decision, at any one time no less than 2048 bytes was randomly addressable, a capability which proved useful. This unit was referred to as a *page*. Also, an incredible data rate was available from such a cell — 20 megabytes per second. It was clear that the microprocessors

cally require less than 30 msec. to execute [Schuster77].

length records was that many of the characters would be null and much space could be saved by not reserving places for such data. However, some overhead is involved, regardless, whether a field delimiter is included or an extra, regular bit is added to delimit the field, and it becomes substantial if the fields are short. Some compression is possible given fixed length records, so that the wasted space due to null entries might not be as serious as at first thought. In addition to this well-known problem of fragmentation [Knuth 73a], the cell design introduced an additional capability if fixed-length records only were used. Because of the random addressability within the page, if the record length is known, then the fields not required for a given query can be skipped over completely. This was a significant difference with respect to earlier proposed logic-per-track devices. In many cases, this savings would be quite substantial.

The greatest disadvantage of the fixed length records was the subsequent requirement that all relations must be normalized, i.e., no field could actually be a list or tree. This has serious implications if a network or hierarchal data base model is envisaged, and very little otherwise. It was decided that the savings accruing from being able to skip over fields made the fixed record length the more attractive. The standard relational model was thus supported, i.e., different domains, or columns, could be of arbitrary width as long as every row had the same format.

Thus the following architecture emerged. Each cell now contained at least half a million bytes of memory, including part of all, or nearly all of the relations in the system. The processing was done in parallel by all the cells, and was proportional to the complexity of the query and the size of the relation. High utilization would result from the fact that the work load was generally spread evenly over all the cells. Calculations suggested that the type of queries anticipated would typi-

hardware plummeting and the cost of search operations being reduced relative to indexing, exhaustive search will become much more commonplace in the future.

3.1. Models

It is possible to model a data base system at several different levels. At one level the logical operations that are performed on the database can be modeled. For example, in the relational framework, this might include restriction (selection), projection, equi-join, non-equi-join, update, create, destroy, etc. At a lower level, the primitives used to implement these operations can be modeled. This might include, among other things, searching for pointers or elements, sorting, merging lists or other equivalent operations, and the coordination of operations necessary to insure that concurrent operations do not leave the data base in an unacceptable state. At a still lower level, the actual instruction sequences being used to implement these operations can be studied.

Virtually all data base management systems perform sorting and list merging and use indices to facilitate the access of data. Although the logical operations performed vary greatly, depending on the data base model, it is claimed that the underlying operations will be much the same in two systems performing the same high level functions, whether the data base models are similar or not. So that some insight into the lower level operations performed may be gained, some operations in a relational model and the algorithms by which they might be performed on a distributed architecture will be examined. In particular, a query analyzed by Blasgen and Eswaran [Blasgen 77] involving restriction, projection and join will be considered. How the operation might be performed in the multiprocessor environment will be studied.

CHAPTER 3

Models of a Data Base Management System

In this chapter and the following ones a number of basic models of a data base management system will be considered to judge potential performance of a multiple processor computing system. In order to evaluate the performance of a postulated system, one could take a number of different approaches. These are the possibilities:

(1) *Analyze Models.* This is the most desirable method if feasible. The ideal is to develop a model which is simple enough to be studied analytically, yet complex enough so that the results are not compromised.

(2) *Simulation.* If the model developed cannot be analyzed satisfactorily, an alternative is to simulate the model. This technique is generally much more work, and provides less satisfactory results than analyzing the model directly, but can be used on much more complex models.

(3) *Build and Measure Actual System.* If the above two methods provide unsatisfactory or insufficient information about the behavior of the system, the last possibility is to build the system, possibly simplified somewhat, and take measurements on it directly. This approach provides the most detailed information but is generally much more work and should be avoided or delayed until the potentials of the first two approaches have been exhausted.

The workings of a data base management system are extremely complicated, and simulation of such a system is a major undertaking. In order to provide some insight into the behaviour of a DBMS in the environment proposed here, an attempt has been made to analyze the nature of the operations that are performed in a real DBMS, and to reduce these operations to a series of models which can be analyzed.

The intent of this work is to investigate the potential of a DBMS using multiple processors, so the modeling of currently available systems is not particularly appropriate. For example, the potential for parallelism appears to be greater for queries that cannot be handled by indices, i.e., where exhaustive search is required. Such queries are rarely performed today on significant data bases, because they are too expensive. With the cost of

ble. Here it is assumed that this overhead can be reduced to a level where it is not a major limitation on the system.

The individual processors may be handling different queries in parallel, which is a problem already widely studied, or they may be working jointly on a single large query. The focus of this study is the latter problem.

Hawthorn [Hawthorn 79a] has characterized data base queries in two general categories which she calls "overhead intensive" and "data intensive" queries. In the case of data intensive queries, a major constraint on the speed of execution is the I/O activity generated. Additional hardware on the disk which allows the reading of all the tracks of a cylinder simultaneously is assumed in order to provide for a high potential I/O bandwidth. Though this may increase the cost of the disk somewhat, it is claimed that the majority of the additional logic required can be implemented by the use of the same single chip device proposed for the processors.

In order to take advantage of the potential parallelism available in such an architecture, the data base must be organized in a careful way. Here it is proposed that relations be clustered on cylinders, using no more cylinders than necessary. For a relation which can be placed on a single cylinder, this means that when the arm is in the right position, the entire relation can be accessed in a single revolution as with many of the data base machines described in chapter 1. It also allows for the parallel processing of this data, since each processor receives a portion of the relation in parallel if it is distributed across the cylinder. Indexes should be placed on the same cylinder if they fit, and on a nearby one if they don't. If an index is the clustering index then the index should be partitioned over the surfaces in the same way as the relation itself.

3.2. The System Model

The opportunities provided by VLSI technology will soon allow us to construct a powerful processor, including a significant amount of memory, on a single piece of silicon. The economics of the semiconductor industry dictate that a small number of types of devices can be fabricated, though the volume of each of these may be quite large and the complexity of each may be great [Despain 78]. The most serious limitations imposed by the technology are power dissipation and I/O pins [Despain 78]. These constraints make desirable the construction of a large system from a small number of types of components in a carefully designed manner to take maximum advantage of the pins available.

For the data base computer model, an architecture of the following type is proposed. A large number of single chip computers are connected together in a yet-to-be-determined way and cooperate to perform the tasks required. These processors each have their own memory and may or may not have I/O devices attached to them. A typical I/O device would be the head attached to a single platter of a disk. Thus an entire cylinder can be read in parallel into a set of these processors in a single revolution of the disk. The array of processors has many of the properties of a network and in many regards they may be thought of as a network of homogeneous processors. However, a significant difference is that the communication among the processors is extremely high — in the range of 10 million bytes per second.

Another difference is that a highly optimized protocol is assumed to allow extremely efficient message passing among the processors. The overhead of message traffic among processors has been recognized to be a serious constraint in the data base machine [Hawthorn 79a], [Epstein 80], [McCreery 79]. This problem must be handled well or the multiprocessor approach to DBMS is probably infeasi-

have them apply that criteria as the data are fetched from the disk. Assuming that the selection criteria are not complex so that the processor is able to keep up with the data streaming off the disk, then the disk activity will limit the speed of the response.

If the relation is ordered in a particular way, it may be possible to limit the search operation to certain nodes, i.e., processors, by recognizing that the selection criteria eliminate all the tuples at certain nodes. This may be particularly useful in the case where the criteria are intended to specify exactly one record. In general, this is probably not very helpful, however. The disk presents the primary limitation on response time, and in the general case, two queries will require access to different cylinders. Thus the processors idled will not be able to do anything else, since the disk arm is set to a particular position handling the current query and cannot be moved so that they may process a different one. They may be able to do housekeeping chores or other kinds of operations, however. On the other hand, there is a cost associated with this, since a simple index is required to be able to identify which processors should be used.

If an index does exist, it may be quicker to search the index for the criteria than to search the entire relation. This will not be true, however, if the relation is small enough that it fits on a single cylinder, since one revolution is required to search either the index or the relation in that case, and searching the index then usually requires further I/O accesses to fetch the data. (It is possible that only pointers to the data are required and that these can be adequately supplied by the index).

Thus a necessary condition for indices to be useful for restriction is that the relation be so large that it will not fit on a single cylinder. Under this cir-

For such an organization, the following operations on a relational data base shall be considered: restriction, projection, and join (equi-join). As mentioned in chapter 1, these are the functions which [Maller 78] thinks need to be implemented directly for a high performance DBMS. Epstein [Epstein 80] pointed out that updates are often initially processed as a retrieval, followed by some lower level processing. Particularly from the standpoint of inter-processor traffic, the extension of a retrieval to include updates is, in most cases, trivial.

The concurrent access to a data base by multiple users gives rise to a plethora of problems related to maintaining consistency and recovering from systems failures [Eswaran 76]. Though little has been said here about updating the data base, the problem is well-known and exists even in the case of a single processor [Gray 78]. Clearly, efficient implementation of locking primitives is important. However, the issue studied here is the effect of the DBMS on the architecture of the computer system. There is no reason to think that conventional locking techniques [Ries 77] cannot be implemented in the proposed system with little impact.

Therefore, this study will consider only retrieval.

3.2.1. Restriction

Restriction is the selection of a subset of the records of the relation as a result of some qualification. Frequently, a restriction results in only a single result tuple. If the qualification includes the specification of the value of a field for which an index exists, it is usually faster to search the index instead of the relation itself to determine the result tuples, then accessing the relation only to fetch the actual qualifying records.

Consider first the case where no index exists. The complete relation must be searched. The algorithm for this operation is simple: Broadcast the selection criteria to all the processors connected directly to a disk surface and

3.2.2. Projection

Projection is the operation to eliminate certain fields of a relation. In many cases the elimination of certain fields means that the resulting relation may have duplicate, identical records. This happens whenever two records differ only in the fields which are eliminated in the projection.

The projection can be performed easily in the postulated environment, since each processor can scan the part of the relation associated with it, producing in place of each record a new record consisting of the subset of fields surviving the projection. These records can be stored internally if they are sufficiently small. Otherwise, they may be stored in a temporary file, either on the same track as the original relation, if there is room, or on another track of the same surface. Thus each processor can handle its portion of the file independently as it comes of the disk except for the final step — elimination of duplicates. In the general case, these duplicates may be scattered over all the participating processors, and these pieces must now be compared to remove the duplicates.

3.3. Topology

Initially the question occurs whether the topology of the network is of any importance. In particular, if the network is so efficient that it can handle the data as rapidly as it can be retrieved from secondary storage, then the issue of its topology is irrelevant. This may in fact be the case in certain circumstances, depending particularly on the nature of the queries, the organization of the I/O, and the organization of the data base. Clearly a tremendous bandwidth is necessary from the I/O devices to perform exhaustive searches rapidly. This might be facilitated by attaching one processor within the structure to each head of a moving head disk, allowing access to all heads in parallel. A file can then be retrieved quickly if

cumstance, the access of an index may be considerably faster than a scan of the entire relation, which entails seeks. If the results of the index are randomly scattered over the relation, however, the resulting fetching of them may take as long as scanning the relation in the first place (or even longer if the fetching is not done in an intelligent way).

In general, then, an index is useful if the number of cylinders accessed in searching the index is less than the number of cylinders of the relation which can be elided as a result of the index scan. Of course, this may be impossible to predict before the scanning of the index is performed, but in the case where only a single record ultimately is to be fetched, the advantage of the index will be predictable.

For a relation which is frequently accessed, it may be useful to keep some information about its partitioning in the processor. This allows restrictions which depend on the clustering field to be handled quickly by eliminating entire cylinders of the relation which are known not to meet the qualification criteria. This sort of index is small enough that it can be kept in the processor main storage if it is frequently used, thus requiring no I/O activity.

The communication among the processors required for restriction is fairly small. The selection criteria must be broadcast to all the I/O nodes, and presumably the results of the query must be assembled somewhere. In addition, if an index is used, and it is not the clustering index, i.e., if it is not ordered and partitioned in the same way as the relation itself, then the identifier of qualifying records must be sent from the node identifying the record to the node into which that record can be accessed.

CHAPTER 4

The Elimination of Duplicates

The projection operation performed on a relational data base has been shown to be a simple problem for the individual processors to handle except for the elimination of duplicates among the various nodes. In the handling of a complex query this reduction in data is often crucial and may be performed multiple times. The elimination of duplicates is so expensive that the user is often given the opportunity to tell the system when it need not be done. This operation will be studied at great length in order to determine its communication requirements.

In terms of the telephone directory model consider the following operation:

List all of the street names present in the directory.

The stripping away of the names, street numbers, and telephone numbers will leave the desired information, but in a highly redundant form, since many people live on the same street. The result may be thought of as a list of numbers and the problem is to eliminate multiple occurrences of a number.

Here it is assumed that the number of elements N making up the list is large — too large to be reasonably supported by a single processor. Duplicates can be eliminated by exhaustive comparison or by sorting, or by some combination of the two. The advantage of the sorting approach is this: direct comparison of all pairs of elements in a list of length N requires $O(N^2)$ comparisons to eliminate all duplicates. Sorting algorithms, on the other hand, may require only $O(N \lg N)$ comparisons worst case and may, depending on the order in the list, require a substantially smaller number than that. Algorithms exist, in fact, for sorting in $O(N)$ operations [Knuth 73, pp. 99-102]. However, sorting implies the moving of a large amount of data, more or less randomly. This is awkward, particularly if the

it resides only on a few cylinders. Ideally, it should reside only on a single cylinder, being spread over multiple disks if necessary to accommodate it. In general, individual relations may be too large to fit on a single cylinder and therefore reside in a cluster of cylinders. Algorithms which will handle such files will be examined.

3.4. Summary of Critical DBMS Operations

The important relational DBMS operations have been examined and the two which present the most challenge to the computing system are the join operation and the elimination of duplicates resulting from the projection operation. These two operations must now be studied in more detail to determine how they can best be supported. In chapter 4 the elimination of duplicates problem will be considered to contrast various interconnection alternatives among the processors.

maximum number of comparisons performed in any one node, and MN_{max}, the maximum number of messages transmitted over any one link.

Identifying the duplicates in two ordered lists is equivalent in complexity to merging the lists since a trivial addition to the merging procedure is to test for equality and eliminate duplicates. For all methods presented, it is assumed that each processor first sorts and eliminates its own duplicates. Since this requirement is the same for all methods, it has been ignored. Thus, to merge two ordered lists of lengths L_1 and L_2 requires $L_1 + L_2$ comparisons[1]. How can the duplicates be eliminated?

Consider how it might be done by first eliminating duplicates within a list, then by comparing every pair of lists. Somehow the lists must be transmitted in a regular way so that all lists are compared against each other. However, a method must avoid the problem of mutual destruction of all duplicates. The following method does this by assigning priorities to the processors:

4.1.1. Method 1: Sequential Broadcast/Common Bus Organization

The P processors are ordered and connected to a common bus. Each processor eliminates the duplicates within its own list. The first processor broadcasts its condensed list in sorted order to the remaining processors and quits. Each processor which receives the list compares it against its own elements and eliminates all elements that match an element of the broadcast list. The remaining processors sequentially broadcast their condensed lists and quit. When all processors but the last have broadcast their lists, the duplicates have been eliminated.

This algorithm has the desirable property that the message size shrinks as the duplicates are eliminated. Thus the total length of the message units sent is the length of the list of all elements with duplicates eliminated less the

[1]Knuth [Knuth 75, pp. 198-200] shows that the minimum possible for the worst case is actually $L_1 + L_2 - 1$ if L_1 and L_2 are approximately equal. Here the constant term is ignored and it is assumed that, in general, merging y lists, each of length L, can be performed in $y \cdot L \cdot lg\, y$ comparisons, recognizing that this simplification results in the assumption that merging two lists of length one requires two compares.

elements to be sorted are in different processors. Thus a tradeoff exists between the movement of data and the number of comparisons, depending on the approach chosen.

The methods to follow will be compared in two ways: the cost of computation C and the cost of communication M. The computation will be measured crudely by estimating the number of comparisons required. The interprocessor movement of data is measured by the sum of the transmission of every element across every interprocessor link. If an element must traverse three such links, then three message element links must be counted. All elements are assumed to be the same length. Computation costs will be determined both for total system and for the busiest processor. Communication costs will be determined both for the total system and for the busiest link. This allows an analysis based on either through-put requirements or response-time requirements, i.e., bandwidth or latency requirements.

4.1. Parallel Methods for Eliminating Duplicates

Assume that a number of identical computers (P) are connected so that they can communicate in a fairly intimate way among themselves via messages. (P is assumed to be a power of 2, except where noted). Suppose that a list of numbers of total length N, is segmented into P lists of equal length $L = \frac{N}{P}$ and distributed over the P processors. In the general case, a wide range of possible outcomes could result from the elimination of duplicate elements, depending on the degree of redundancy in the data base. In order to establish bounds on the size of the task, consider the two extreme cases:

(1) *All elements are identical.* For this case let $C1$ be the total number of comparisons done in all processors and $M1$ be the total number of message element links. Also, define $C1_{max}$, the maximum number of comparisons performed in any one node, and $M1_{max}$, the maximum number of messages transmitted over any one link.

(2) *All elements are unique,* i.e., *there are no duplicates.* In this case, let CN be the total number of comparisons done in all processors and let MN be the total number of message element links. Also, define CN_{max}, the

Another sort of priority can be introduced by allowing an additional processor to do the comparison of the two or more lists and produce the result:

4.1.2. Method 2: Tree Organization

> Each processor, after eliminating its own duplicates, sends its list to its parent in sorted order, which merges the lists it receives, eliminating the duplicates, and sends the result on to its parent. This continues until the final list is formed at the root of the tree.

In this structure there are actually $\frac{yP - 1}{y - 1}$ processors connected as a tree, where y is the branching factor of the tree and P is a power of y. P processors are leaves, $\frac{P - 1}{y - 1}$ are non-leaves. The P processors at the leaves have direct access to the data. This method requires more processors, nearly twice as many as in the previous case. It is very effective if the number of duplicates is large. However, if few duplicates exist, the length of the list will increase, by nearly a factor of y at each stage, increasing both the computation and the worst case message traffic with each level up the tree. Thus each succeeding step uses only $1 / y$ as many processors, each of which must do y times as much computation. For this case CN is calculated as follows:
There are $lg_y P$ levels (counting the root or the leaves, but not both). There are $\frac{y(P - 1)}{y - 1}$ links, one above every node except the root. Numbering the levels in ascending order starting with the leaves as 0,

level 1 contains $\frac{P}{y}$ processors, each merging y lists of length L,

level 2 contains $\frac{P}{y^2}$ processors, each merging y lists of length yL,

. . .

level j contains $\frac{P}{y^j}$ processors, each merging y lists of length $y^{j-1}L$.

number of unique elements in the last processor. Obviously, this is the least possible communication cost.

It solves the problem of saving exactly one copy of each element by serializing the broadcasts. Note that these broadcasts cannot be done in parallel, even if multiple busses are available. As a result, the parallelism is limited. On average, no more than half of the processors are busy. Since each list is sorted before communication begins, the removal of the duplicates can be done in one pass for each broadcast. If there are no duplicates actually present, then the total number of comparisons CN is

$$CN = 2L(P-1) + 2L(P-2) + \cdots + 2L(1) = 2L\frac{P(P-1)}{2} = N(P-1).$$

$$CN_{\max} = 2L(P-1) = 2(N-L).$$

Also

$$MN = (P-1)L = N-L,$$

$$MN_{\max} = MN = N-L.$$

Here the broadcast of a message to many other processors is counted as only one message sent. If there is only one unique element, only that one element is sent, and each of the other $P-1$ processors compare it and eliminate their copy:

$$C1 = 2(P-1), \quad C1_{\max} = 2,$$

and

$$M1 = 1, \quad M1_{\max} = 1.$$

Of course, there will be some messages necessary to notify other processors that no more elements are to be sent, but this is considered overhead which, in general, is small enough to ignore.

$$Ml_{max} = 1.$$

Consider the binary tree case ($y = 2$). Since the lists were sorted before being sent to a parent all that is required of the parent is a merge of ordered lists, a procedure that increases only linearly with the length of the list. Now suppose that instead of sending the list to a common parent, the two processors divide their elements into two lists in a commonly agreed way and exchange one of them. This leads to the first algorithm utilizing a global sorting:

4.1.3. Method 3: Binary Merge/N-cube

> P processors are numbered in binary from left to right, starting with 0. Each processor eliminates its duplicates, leaving them in sorted order. The range of values of the sort field is partitioned in a universally agreed-upon way, (the obvious way, for example, is to use the most significant bit of each element), and each processor breaks its list into two parts. It then sends one of the two lists to the processor having the same address except for the most significant bit as follows: If the most significant bit of the address of the sending processor is a 1, it sends the first list. Otherwise, it sends the second list.
>
> After merging the received list with the retained one and eliminating duplicates, each processor repeats the process, but with the following modification:
>
> Each partition of the range of the sort field is further sub-divided into two parts. If the straight-forward way is used, then on step j, the jth most significant bit of the address is used to determine which list to send and to whom it will be sent.
>
> This process is repeated n times, after which the range is partitioned into P parts, and one processor contains all the values for exactly one partition. If the obvious partition was used, all numbers are sorted into the proper list according to their n most significant bits.
>
> The links required between the processors form the n-dimensional structure known as the n-cube [Siegel 79] and sometimes called *hypercube.*

This procedure requires only $n = lg\ P$ steps and does not get more complex on subsequent steps — in fact it gets shorter with the elimination of

---

when y is a power of 2.

Assuming that y lists of length L require $yL \lg y$ comparisons[2] gives

$$CN = \frac{P}{y} yL \lg y + \frac{P}{y^2} y^2 L \lg y + \cdots + \frac{P}{y^{\lg_y P}} y^{\lg_y P} \cdot L \lg y$$

$$= P \cdot L \cdot (\lg_2 y)(\lg_y P) = N \lg P.$$

$CN_{\max}$ is computed for the top node, merging y lists of length $\frac{N}{y}$. Again assuming that y lists of length L require $y \cdot L \cdot \lg y$ comparisons,

$$CN_{\max} = y \frac{N}{y} \lg y = N \lg y.$$

The calculation of MN follows from the observation that each element goes from a leaf node to the root, i.e., through $\lg_y P$ links. Therefore,

$$MN = N \lg_y P = \frac{N}{\lg y} \lg P.$$

Each of the top level links carries $\frac{N}{y}$ elements, i.e.,

$$MN_{\max} = \frac{N}{y}.$$

This difficulty suggests that the upper nodes might require greater power and larger memory. If all elements are identical, no such congestion occurs. Each non-leaf node receives y lists of length 1. Assuming that merging y lists of length 1 requires $y \lg y$ operations, then

$$C1 = \frac{P - 1}{y - 1} y \lg y.$$

Since each non-leaf node does the same number of operations,

$$C1_{\max} = \frac{C1}{\textit{number of non-leaf nodes}} = y \lg y.$$

Since exactly one element passes through each link,

$$M1 = \frac{y}{y - 1}(P - 1),$$

and

---

[2]Strictly speaking, this is an equality only if y is a power of 2. Comparison is inherently a binary operation, and a y-way merge can be accomplished with only about $\lg y$ comparisons per element using a selection tree

the redundant elements. After each of the $lg\ P$ exchanges, all P processors merge two lists of length $\frac{L}{2}$. Therefore,

$$CN = (lg\ P)P \cdot 2\left\lceil\frac{L}{2}\right\rceil lg\ 2 = N\ lg\ P.$$

Symmetry arguments guarantee that CN_{max} and MN_{max} are just $\frac{CN}{P}$ and $\frac{MN}{P}$ respectively. Assuming that on each move, half the elements are moved[3],

$$MN = lg\ P\ \frac{N}{2} = \frac{N\ lg\ P}{2.}$$

For the unique case, after exchange j, $\frac{P}{2^j}$ nodes have a list of length 1 while the remainder have the empty list.

$$C1 = 2(1)\ lg\ 2\sum_{j=1}^{lg\ P}\frac{P}{2^j} = 2(P - 1),$$

$$C1_{max} = 2(1)\ lg\ 2\ (lg\ P) = 2lg\ P.$$

During exchange j, $\frac{P}{2^j}$ elements are sent:

$$M1 = \sum_{j=1}^{lg\ p}\frac{P}{2^j} = P - 1$$

and $M1_{max} = 1$.

There is nothing magic about the binary process, however. It is possible to use a y-way sort and divide the elements into y lists, sending $y - 1$ off at each step. This would require fewer steps, since

$$lg_y\ P < lg_2\ P$$

for all values of $y > 2, P > 1$. Carrying this idea to its extreme, this could be done in base P, in which case only one swap would occur. This results in the following method:

[3] In the worst case, when all elements are moved each time, $MN = N\ lg\ P$.

4.1.4. Method 4: P-Merge

> Each processor orders its own list and, after eliminating its own duplicates, partitions the list into P separate lists in a consistent way for all processors. Numbering these sublists from lowest segment to highest, the jth segment is sent to the jth processor. Each processor retains only that sublist which it would send to itself, and merges it and the P - 1 incoming sublists as they arrive.

This method again is near optimal in terms of the transmission of information, at least for the case where there are few duplicates. With no duplicates, each processor merges P lists, each of length $\frac{L}{P}$:

$$CN = P\left[P\,\frac{L}{P}\,lg\,P\right] = N\,lg\,P.$$

$$CN_{max} = \frac{CN}{P} = lg\,P.$$

Each processor sends $L - \frac{L}{P}$ elements:[4]

$$MN = P\left[L - \frac{L}{P}\right] = N - L.$$

$$MN_{max} = \frac{MN}{number\ of\ links} = \frac{N - L}{P(P-1)/2} = 2\frac{L}{P}.$$

For the single unique element case, only one processor receives anything: P - 1 lists of length 1.

$$C1 = P \cdot 1 \cdot lg\,P = P\,lg\,P, \quad C1_{max} = C1 = P\,lg\,P,$$

$$M1 = P - 1, \quad M1_{max} = 1.$$

Another extension of Method 2 is two build a network which contains the interconnections for one dimension of the n-cube and has the capability to move the data among processors so that each exchange can be accomplished with immediate neighbors. It has been shown [Stone 71] that both the shuffling of the data and the exchange can be effected in paths through only one link each for the network known as the perfect shuffle. This leads to our

[4] $MN = N$, worst case.

last method.

4.1.5. Method 5: Binary Merge/Perfect Shuffle Connection

> P processors are numbered in binary from left to right, starting with 0. Each processor has a link to one neighbor whose address is the same except for the least significant bit. This link is used to implement the exchange. In addition, each processor has two other links to the two processors having the same address but shifted (end-around) one position. Each of these links is used once for each shuffle. Each processor eliminates its duplicates, leaving them in sorted order. Using the agreed test, (again perhaps the most significant bit of each element), each processor partitions its list into two smaller ones. It then exchanges one list with its neighbor.
>
> After merging the received list with the retained one and eliminating duplicates, a shuffle is performed, i.e., each processor sends its entire list over the link to the processor with the same address shifted one position, say, left.
>
> This process is repeated $lg\ P$ - 1 times, after which the range is partitioned into P parts and each processor has all the values for exactly one partition.

The computation involved here is the same as for the n-cube structure, so CN and CN_{max} are precisely the same as that case. Assuming again that on each move, half the elements are moved, the communication involved in Method 3 is again required, but additional communication is incurred because of the shuffles. Each shuffle involves sending all surviving elements through one link, and since there are $lg\ P$ - 1 shuffles,

$$MN = \frac{N\ lg\ P}{2} + N(lg\ P - 1) = 3\frac{N\ lg\ P}{2} - N.$$

Since more traffic goes over the shuffle links, the traffic on the busiest link is

$$MN_{max} = \frac{N(lg\ P - 1)}{P} = L(lg\ P - 1).$$

For the unique case, again $C1$ and $C1_{max}$ are the same as for Method 3. Again an additional communication cost is incurred because of the shuffle. During shuffle j, $j = 1,2,3,\cdots,(lg\ P - 1)$, $\frac{P}{2^j}$ nodes transmit a list of length 1, the remainder transmitting the empty list. Thus the additional

communication cost for the shuffles is

$$\sum_{j=1}^{(\lg P - 1)} \frac{P}{2^j} = P - 2,$$

so the total communication cost is

$$M1 = P - 1 + P - 2 = 2P - 3.$$

The busiest link is the exchange link of one particular processor which carries the unique element on every exchange. Thus,

$$M1_{\max} = \lg P.$$

4.2. Comparison of the Methods

Table 4.1 compares the five methods under the assumption that no duplicate data exists. Table 4.2 compares the five methods for the model where all elements are identical. The parameters have been normalized for the case of all unique elements by dividing by L, the length of the list in each processor. For purposes of comparison, the following assumptions have been made:

(1) Order is initially totally random, but the elements are evenly distributed among the processors.
(2) The numbers are scattered randomly, i.e., evenly, over their possible values.

The first assumption seems reasonable, though presumably it corresponds to some sort of worst case. The second assumption, however, requires some justification. Normally one would expect to find severe clustering of the numbers resulting from the fact that they are normally derived from natural language or other organized sets of data. They can be randomized, however, by hashing the sort field. The sort order is changed, making the end result of little use as a sorted list. This is not terribly important in many cases, however, since the elimination of duplicates is so often an intermediate result and ordering is not useful anyway.

	Method				
	1	2	3	4	5
No. of Processors	P	$\frac{yP - 1}{y - 1}$	P	P	P
Number of links	1	$\frac{y(P - 1)}{y - 1}$	$\frac{P \lg P}{2}$	$\frac{P(P-1)}{2}$	$\frac{3P}{2}$
CN / L	$P(P - 1)$	$P \lg P$	$P \lg P$	$P \lg P$	$P \lg P$
MN / L	$P - 1$	$\frac{P}{\log y} \lg P$	$\frac{P \lg P}{2}$	$P - 1$	$\frac{3P \lg P}{2} - P$
CN_{max} / L	$2(P - 1)$	$P \log y$	$\lg P$	$\lg P$	$\lg P$
MN_{max} / L	$P - 1$	$\frac{P}{y}$	1	$\frac{2}{P}$	$\lg P - 1$
A	$P - 1$	$y + 1$	$\lg P$	$P - 1$	3
$\overline{MN} / L$	$(P - 1)^2$	$\frac{(y + 1)}{\log y} P \lg P$	$\frac{P(\lg P)^2}{2}$	$(P - 1)^2$	$\frac{9P \lg P}{2} - 3P$
$\overline{MN_{max}} / L$	$(P - 1)^2$	$\frac{y + 1}{y} P$	$\lg P$	$2\frac{(P - 1)}{P}$	$3(\lg P - 1)$

Table 4.1. Comparison of five methods for eliminating duplicates assuming that no duplicates exist. Method 1: Sequential broadcast. Method 2: y-branch tree. Method 3: Binary merge. Method 4: P merge. Method 5: Perfect shuffle. CN is the total number of comparisons done in all processors. MN is the total number of message element links. CN_{max} is the maximum number of comparisons done in one processor. MN_{max} is the maximum number of message elements passing through any one node. $\overline{MN} = MN \cdot A$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{MN_{max}} = MN_{max} \cdot A$ is the normalized measure of busiest link traffic. A is the normalization factor for the variable number of ports required.

	Method				
	1	2	3	4	5
No. of Processors	P	$\frac{yP - 1}{y - 1}$	P	P	P
Number of links	1	$\frac{y(P - 1)}{y - 1}$	$\frac{P \lg P}{2}$	$\frac{P(P-1)}{2}$	$\frac{3}{2}P$
$C1$	$2(P - 1)$	$\frac{y \log y}{y - 1}(P-1)$	$2(P - 1)$	$P \lg P$	$2(P - 1)$
$M1$	1	$\frac{y}{y - 1}(P - 1)$	$P - 1$	$P - 1$	$2P - 3$
$C1_{max}$	2	$y \log y$	$2 \lg P$	$P \lg P$	$2 \lg P$
$M1_{max}$	1	1	1	1	$\lg P$
A	$P - 1$	$y + 1$	$\lg P$	$P - 1$	3
$\overline{M1}$	$P - 1$	$\frac{y(y + 1)}{y - 1}(P - 1)$	$(P - 1) \lg P$	$(P - 1)^2$	$6P - 9$
$\overline{M1_{max}}$	$P - 1$	$y + 1$	$\lg P$	$P - 1$	$3 \lg P$

Table 4.2. Comparison of five methods for eliminating duplicates assuming that all elements are identical. Method 1: Sequential broadcast. Method 2: Tree. Method 3: Binary merge. Method 4: P merge. Method 5: Perfect Shuffle. $C1$ is the total number of comparisons done in all processors. $M1$ is the total number of message element links. A is the normalization factor for the variable number of ports required. $\overline{M1} = M1 \cdot A$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{M1_{max}} = M1_{max} \cdot A$ is the normalized measure of worst case link traffic.

A more serious problem is that if the hashing function fails to randomize the data sufficiently, some nodes may receive very large lists. This would imply that each processor must have enough memory to hold the entire list, violating our assumptions made. This problem can be resolved, however, by aborting an operation as soon as an overflow occurs, and substituting a more appropriate hashing function.

The comparison of message traffic among processors is not straightforward if the processors in the different cases have different kinds or numbers of ports. One might reasonably expect that processors with more ports or faster ports would be more expensive, so that it is also only fair to expect more performance from them. In the above cases the processors vary widely in their I/O capability, from a binary tree or the perfect shuffle, which need only three ports per processor, to the complete interconnection, which requires as many ports on each processor as there are processors, less one. The bus structure is even harder to compare, since although only one port is specified, it nevertheless is obviously much different than the port required by the other cases.

Despain [Despain 79] has shown for a single chip computer that power considerations limit the total I/O bandwidth of the processors. Thus if the total bandwidth available is B bits/second, it can be assumed that each of K identical ports can transmit a maximum of $\frac{B}{K}$ bits per second. He has also shown for the case where Q processors share a bus that a processor using the bus can achieve a bandwidth of only[5] $\frac{B}{Q-1}$ bits/second. A further result is that reduced bandwidth is equivalent to an increase in the average path length for a message, i.e., for a given set of message interchanges there exists an average path length A, such that

[5]In the special case where the processors broadcast sequentially. In the general case where all processors are vying for the bus, it is much worse, i.e., $B / (Q - 1)^2$.

$$A \cdot B_E = B,$$

where B_E is the effective bandwidth through a port and B is the total bandwidth available to a processor.

If it is assumed that a structure with equivalent performance can be obtained by reducing the number of ports and increasing the number of intermediate nodes traversed, then equivalence among the various processors is defined by multiplying the message traffic by the average path length. Thus define

$$\overline{MN} = A \cdot MN, \quad \overline{MN_{\max}} = A \cdot MN_{\max},$$

and

$$\overline{M1} = A \cdot M1A, \quad \overline{M1_{\max}} = A \cdot M1_{\max}.$$

Tables 4.1 and 4.2 show values for the effective path length A and the compensated values for message traffic. Under these assumptions, the tree (method 2) and the perfect shuffle (method 5) have the least total message traffic, regardless of the duplication factor, though the binary merge with the n-cube (method 3) has less message traffic if P is quite small. Also, the optimal value for y is 4, although the differences are small for values of 2 to 8. On the other hand, when the duplication is low, the tree exhibits congestion near the root, and all methods but the bus are superior to the tree with respect to the busiest link, increasingly so with larger values of P. When the duplication is high, however, the binary tree is exceedingly effective, with the busiest link not affected even with increases in P. Clearly none of these structures is best over the range of consideration.

Some of the methods are asynchronous. The P - 1 messages that each processor sends in the P-merge (method 4) need not be sent simultaneously. Each processor can begin processing the second phase of the P-merge as soon as one message has arrived. Thus communication and processing can be overlapped.

The binary merge (method 3) likewise can proceed asynchronously, with each node having a list of other processors with which it must communicate sequentially. Thus, either of these methods can be implemented on a general computer network where all nodes can communicate with all others. Both require many messages to many different nodes, so it should be noted that efficient communications are vital in the elimination of duplicates using a sorting scheme.

It is interesting to observe in the tree (method 2) that if the initial elimination of duplicates results in the elements being sorted, then the processors above the bottom level can proceed asynchronously in a pipeline fashion. Each processor may begin processing as soon as it has received one element from each of its children. After selecting the lowest value of those received, it can immediately send this element on to its parent, and remove it from its own list. Thus it is not at any time required to store the complete lists, which may be growing quite large. Of course the node at the top must do something with the resulting list, and it might turn it around and send it down the tree, where it can be sorted on the way down, thus preserving a useful sort order. Thus all the non-leaf nodes can be working simultaneously, resulting in a higher degree of parallelism than might otherwise be expected.

The tree model (method 2) uses up to twice as many processors as the other models. The difference in performance, however, is much greater than a factor of two. The amount of computation is no more than for any other method, so the processors on the average do only half as much work. The total message traffic, on the other hand, is much less than for any other method, for large values of P. The important consideration here is how rapidly the requirements grow as the number of processors grows, and in this respect, a mere factor of two is quite unimportant.

Methods 3 and 4 require substantially more data paths than any of the other methods. Clearly the P-merge is not feasible for large P if $P(P-1)$ dedicated links are required. Even the $P \lg \frac{P}{2}$ links required by the binary merge are hard to justify for large values of P. This would mean $\lg P$ links per node if dedicated links were used.

A more serious problem is the lack of expandability imposed by these structures. A processor may have only a fixed number of ports, particularly if it is a single integrated circuit. Methods 3 and 4 require an increase in the number of ports as the number of processors grows, so that if room is left for expansion, then some of the available ports are unused, wasting available resources.

The perfect shuffle seems to have many of the properties needed here, though it is markedly inferior to the binary tree in the case of high duplication. It also has a large enough linear coefficient for MN that its superiority occurs only for large values of P. But it poses some unfortunate problems as well. It certainly cannot be gracefully expanded, since it requires a power of two processors. Furthermore, the routing of messages in such a structure is difficulty because of its lack of symmetry.

On the other hand, the sequential broadcast method takes substantially longer to execute than the other methods. Also, if the duplication factor is low, it requires far more comparisons than any other method and much more communication bandwidth than any method except the complete interconnection.

It is clear that the bus is inferior to the other methods. However, the others all have shortcomings which are extremely serious. The question then arises — is it possible to construct a network on which several of the methods can be implemented so that the best method may be employed in a given situation?

Assuming that each processor in a structure has the same number of ports, a significant variation among these structures is the portion of ports actually used. The complete interconnection and the n-cube algorithms, for example, use every port. But the binary tree uses less than two-thirds of the ports it has, since each leaf node has two unused ports. It has been suggested [Despain 78] that this is desirable to allow a convenient placement of I/O devices, a point that all structures must address somehow. Thus a fairer comparison might require that each structure have as many unused ports as it has processors. For methods 3, 4, and 5 this would be approximately equivalent to increasing the value of A by 1.

An alternative approach, and one taken here, is to connect the unused ports of such a structure in some regular way. One possibility for the binary tree is to connect the leaves to form the perfect shuffle interconnection, Fig. 4.1. The exchange can now be accomplished by messages exchanged through the common parent. Unfortunately, this doubles the traffic during the shuffle, which is already the dominant traffic for method 5. A better possibility exists.

4.3. X-Tree

A topology recently proposed [Goodman 79] in connection with X-Tree [Despain 78], can implement any of the algorithms. The structure, called *Hyper-tree*, is the binary tree topology, but with each node having one extra link connecting it in a regular way to another node at the same level (Fig. 4.2.). The structure is particularly well-suited for communications among leaf nodes which are nearest neighbors in the n-cube. Since the structure is a binary tree, obviously method 2 (binary tree) can be implemented directly on the structure. In addition, method 3 (n-cube) can be implemented by using the leaf processors, passing messages through intermediate nodes where necessary. Furthermore, the structure has been shown to be well-suited for communication among all leaf nodes, so that method 4

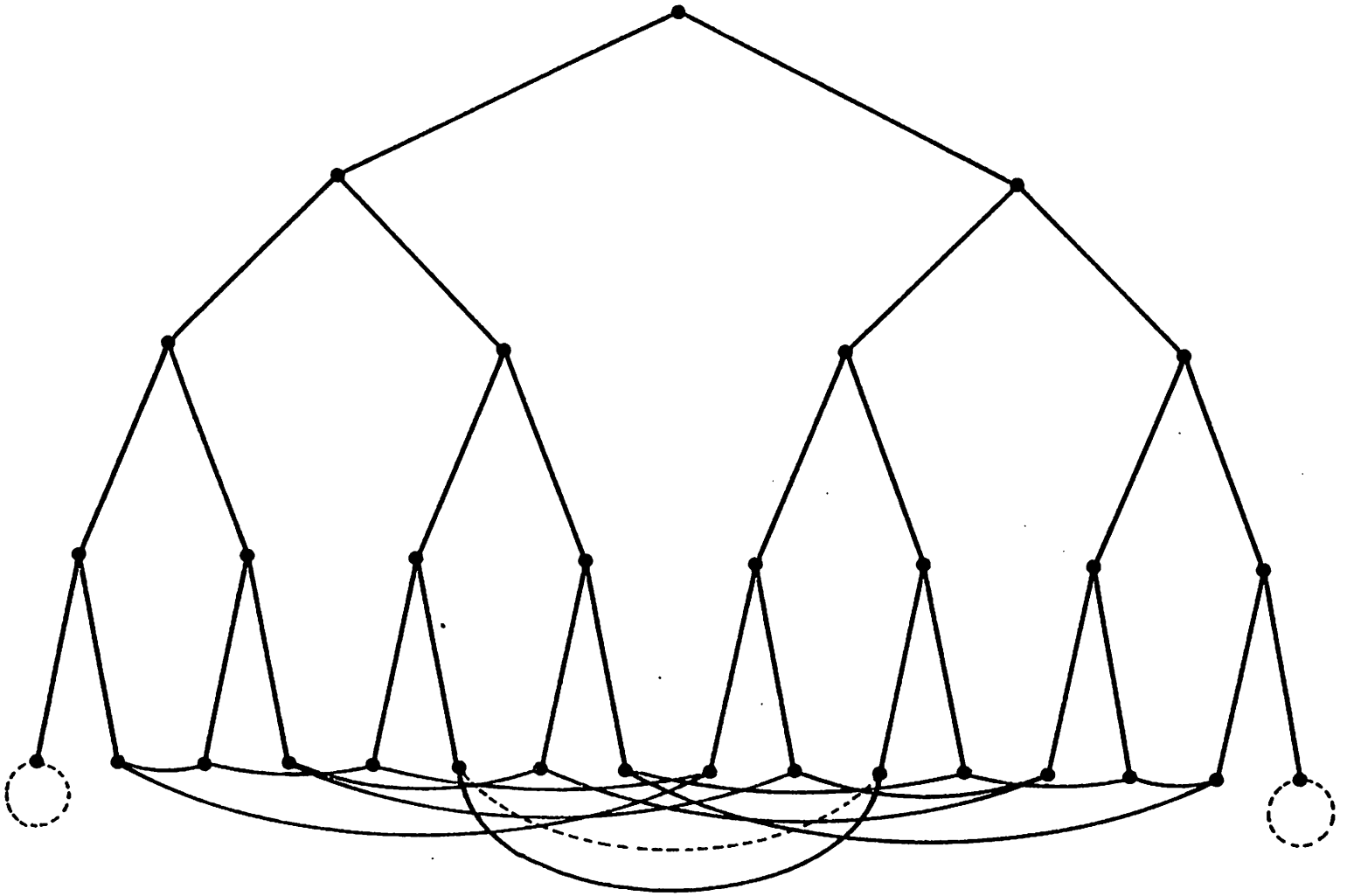

Figure 4.1. Perfect Shuffle Interconnection Superimposed on the Leaves of the Binary Tree.

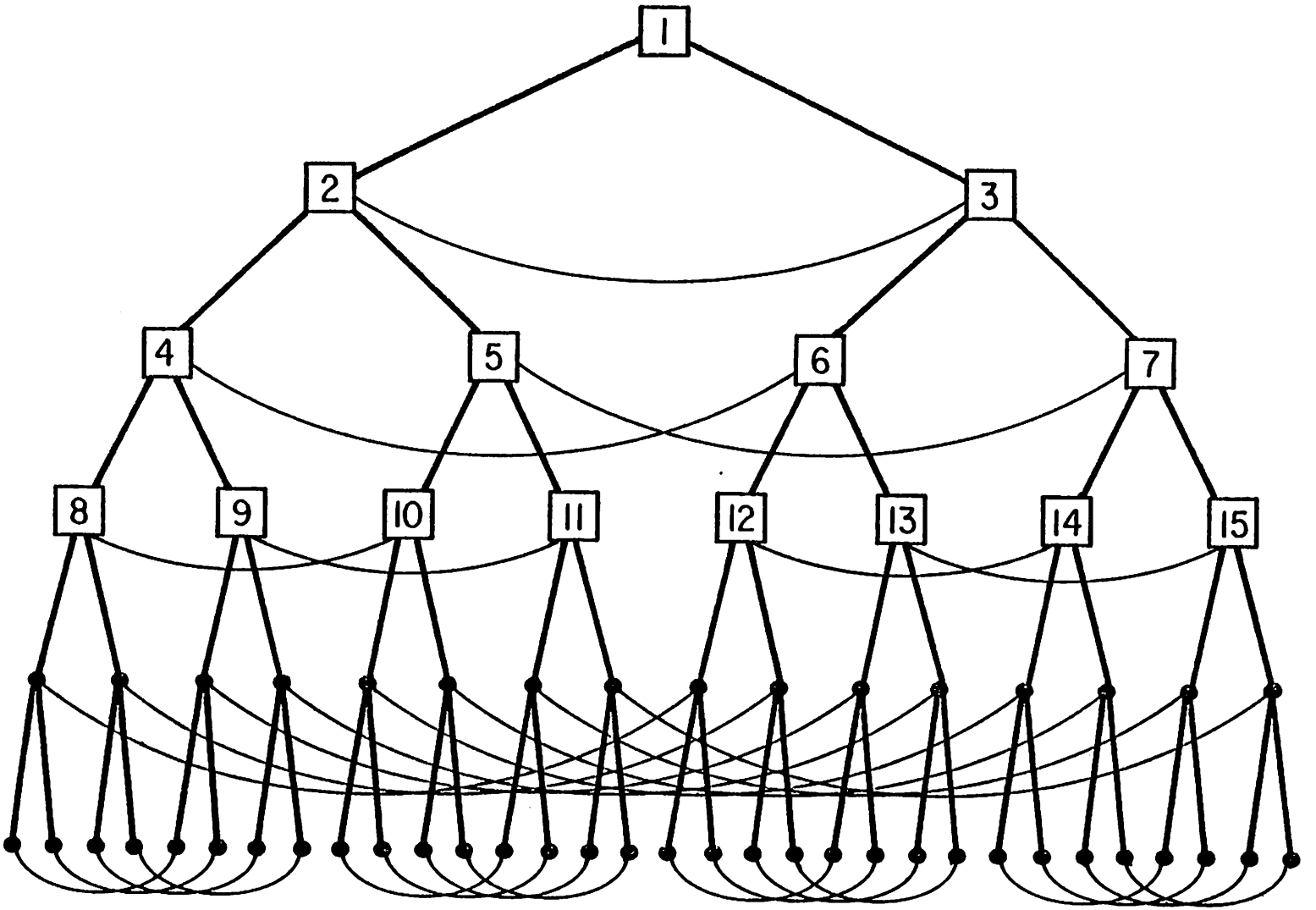

Figure 4.2. Interconnection of Hypertree I. From [Goodman 79].

could also be implemented conveniently. Method 5, the perfect shuffle algorithm, could also be implemented, though nothing is gained by the shuffle, so it is essentially the same as method 3. However, the extra ports of the leaf nodes can be connected in a perfect shuffle so that the horizontal links can be used for the exchange (Fig. 4.3.). With this addition, the structure can perform the binary merge as well as the perfect shuffle network except that the value of A is 4 instead of 3.

Tables 4.3 and 4.4 show the values for this model assuming the structure is used to implement methods 2, 3, 4, and 5. The computations, of course, do not change, being determined by the method and the corresponding logical structure. Degradation occurs for the binary tree structure because the multiplication factor A, is increased by one to accommodate the additional link required for the Hypertree connection.

Method 3 is implemented using the extra links. It has been shown [Goodman 79] that for communication between any pair of leaf nodes, an optimal path exists which goes no more than half way up the tree. This guarantees that the bottleneck which would occur in the simple binary tree if few duplicates are present, will not occur, or at least will be much less serious, since a factor of $\sqrt{P}$ more links are available to handle the traffic over the most heavily used path.

The best method to use varies greatly, depending on the amount of duplication in the list, the number of processors, and the relative importance of total traffic versus busiest link traffic.

For the high duplication case, the simple binary tree algorithm is always the best for the worst case link traffic, though it is slightly inferior to the binary merge (n-cube) algorithm in total traffic. Note also that these two methods have the lowest computational requirements as well, under these conditions.

	Method			
	2	3	4	5
$CN\ /\ L$	$P \lg P$	$P \lg P$	$P \lg P$	$P \lg P$
$MN\ /\ L$	$P \lg P$	$\frac{P \lg P}{4}(\lg P + 1)$	$\frac{4}{3} - \frac{17}{12}P + \frac{5}{4}P \lg P^{\dagger}$	$\frac{3P \lg P}{2} - P$
$CN_{max}\ /\ L$	P	$\lg P$	$\lg P$	$\lg P$
$MN_{max}\ /\ L$	$\frac{P}{2}$	$\frac{\sqrt{2P}^{\dagger}}{4}$	$\frac{P - 1^{\dagger}}{2\sqrt{P}}$	$\lg P - 1$
$\frac{\overline{MN}}{L}$	$4P \lg P$	$P \lg P(\lg P + 1)$	$\frac{16}{3} - \frac{17}{3}P + 5P \lg P^{\dagger}$	$6P \lg P - 4P$
$\overline{MN_{max}}\ /\ L$	$2P$	$\sqrt{2P}^{\dagger}$	$\frac{2(P - 1)^{\dagger}}{\sqrt{P}}$	$4(\lg P - 1)$

Table 4.3. Using the "hypertree" structure with perfect shuffle interconnections among the leaves (Fig. 2). Implementation of four methods for eliminating duplicates assuming that no duplicates exist. There are $2P - 1$ processors and $4P - 3$ links. The normalization factor, A, is 4. CN is the total number of comparisons done in all processors. MN is the total number of message element links. CN_{max} is the maximum number of comparisons done in one processor. $\overline{MN_{max}}$ is the maximum number of message elements passing through any one node. $\overline{MN} = MNA$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{MN_{max}} = MN_{max}A$ is the normalized worst case link traffic.

† This formula is correct only where P is not a power of 4. If P is a power of 4 the formula is slightly different.

	Method			
	2	3	4	5
$C1$	$2(P-1)$	$2(P-1)$	$P \lg P$	$2(P-1)$
$M1$	$2(P-1)$	$2(P-1) - \lg P$	$\frac{4}{3} - \frac{17}{12}P + \frac{5}{4}P \lg P$†	$2P-3$
$C1_{max}$	2	$2 \lg P$	$P \lg P$	$2 \lg P$
$M1_{max}$	1	$(\lg P) - 1$	$P-2$	$\lg P$
$\overline{M1}$	$8(P-1)$	$8(P-1) - 4 \lg P$	$\frac{16}{3} - \frac{17}{3}P + 5P \lg P$†	$8P-12$
$\overline{M1_{max}}$	4	$4 \lg P$	$4(P-2)$	$4 \lg P$

Table 4.4. Using the "hypertree" structure with perfect shuffle interconnections among the leaves (Fig. 2). Implementation of four methods for eliminating duplicates assuming that all elements are identical. There are $2P - 1$ processors and $4P - 3$ links. The normalization factor, A, is 4. $C1$ is the total number of comparisons done in all processors. $M1$ is the total number of message element links. $\overline{M1} = M1A$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{M1_{max}} = M1_{max}A$ is the normalized worst case link traffic.

† This formula is correct only where P is not a power of 4. If P is a power of 4 the formula is slightly different.

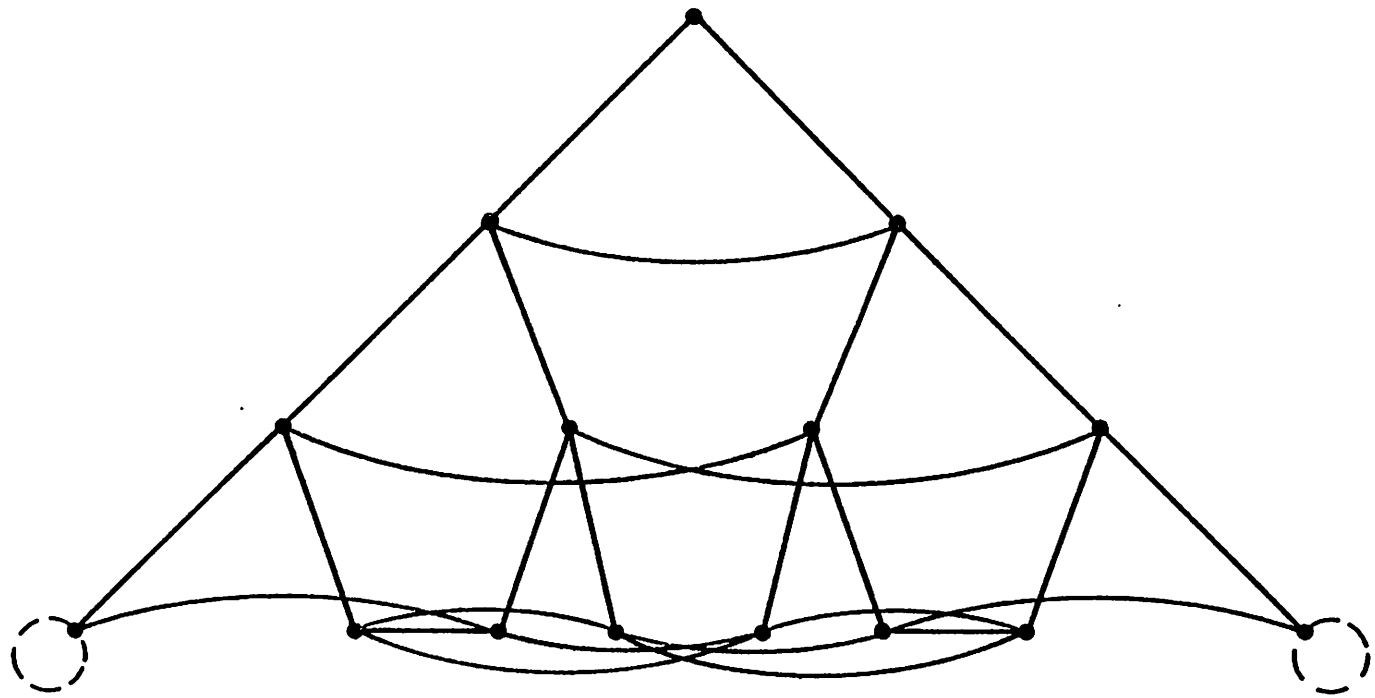

Figure 4.3. Perfect Shuffle Interconnection Superimposed on the Leaves of Hypertree.

For the case of low duplication, the results are not so clear-cut. Up to about 128 leaf nodes, methods 3 and 4 are best, with method 4 slightly preferable if total traffic is the consideration, and method 3 creating somewhat less total message traffic. Method 4 is generally superior at 128 nodes.

Above 128 nodes, method 5, the perfect shuffle, becomes the best method because of its attractive balanced link traffic. Method 2, the binary tree algorithm, has slightly less total traffic, but must be rejected because of the excessive bottleneck occurring at the root, both in link traffic and in computation.

4.4. Conclusions Concerning Structure

The Hypertree structure is able to implement the best algorithm for a given situation. Under the assumptions made, performance is nearly equal, and in some cases superior, to the structure for which the algorithm was originally proposed. The total message traffic, $\frac{\overline{MN}}{L}$ is actually improved, approximately by a factor of $\frac{P}{lg\ P}$ for the algorithm using the complete interconnection (method 4), though the worst case link traffic for the same model is increased by a factor of $\sqrt{P}$ for the case of no duplicates. Since method 4 is the method of choice only for values of $P < 128$ this would seem to be unimportant.

Method 3 shows some degradation in performance. The worst case link traffic is increased, for the case of no duplicates, by a factor of $\frac{\sqrt{2P}}{lg\ P}$, but for large values of P the perfect shuffle algorithm predominates anyway.

Methods 2 and 5 show only slight, linear degradation due to the increase in the value of A resulting from the unused links for that algorithm. Thus the Hypertree structure is able to achieve the same order of performance as the best of the methods considered for virtually all circumstances under the assumptions

made. If other assumptions are made, a different conclusion could also be drawn, as is evident from the results presented in Table 4.1.

The power of the tree structure is clear for the problem of eliminating duplicates. However, the importance of flexibility in choosing the method is apparent. The best structure is clearly one which can handle both extreme cases (and hopefully the cases in between) reasonably well. Only the Hypertree structure can claim this flexibility.

The elimination of duplicates is an important data base operation which provides significant insight into the requirements for a data base computer. Its efficient implementation over a wide range of situations is critical for achieving high performance. Since it can also implement the join operation over a wide range of conditions as well, as will be demonstrated in the next chapter, it is the structure of choice. Other considerations as well have influenced the choice of Hypertree. For example, unlike some other models, the tree structure is expandable without a modification to the processor itself, which surely makes it more attractive if it is a single component. Another issue is the fact that adjacent processors may wish to communicate heavily at times. The binary tree, with nodes having few ports, each with high bandwidth, is clearly advantageous in this case.

The X-Tree "Hypertree" interconnection with the perfect shuffle interconnection among the leaves has been shown to provide an attractive compromise of the models considered. It gives essentially the same performance as the best of the other structures over the range of conditions considered. It will be studied in more detail in the following chapter to determine its performance in the join operation.

CHAPTER 5

The Join Operation

In the relational model of a data base management system, the join operation is involved in all operations involving more than one relation. It is the only common operation which may require more than one access to the same element in a relation. The most difficult operations that can be performed with selection or restriction require no more than a single scan through the relation. While this may not be trivial for very large relations, it is far easier than many operations requiring a join operation, the implementation of which may include many scans under certain conditions.

The join operation can be implemented in many different ways, all producing the equivalent result, differing only in their ordering in the result relation. The best method in a particular case depends on the indices and access methods available, the parameters of the relations involved, and the context in which the query is presented. For example, if the result is an intermediate one, sort order may not be important, even though the final result must be sorted. Also, a join is usually associated with at least one restriction and often a projection. The effect of these operations also varies, depending on the method.

5.1. Synchronizing Data from Disk

Because data stored on a disk is not accessible in a random fashion some method of buffering is necessary in order to synchronize the processor with the data read from the disk. The traditional method of buffering the data from a direct access storage device, such as a disk, is to divide it into blocks and bring only a few into main storage at a time. These blocks are analyzed, perhaps parts are saved, then deleted to make room for more blocks fetched by means of another I/O request. While this method is widely used, it requires substantial

main storage, and it makes disk accesses expensive because many accesses to the disk are required to search a single large file.

If the operation to be performed on the data is a simple sequential search, and if the processor is capable of performing the search at a rate somewhat greater than the rate at which data comes in to main storage from the disk, then the possibility exists that the data can be processed in real-time while it is being read in. For example, if two blocks can be held in main storage, the processor could wait until one is filled. Then while the second is being filled, it could proceed with the operations on the first. Assuming that it can complete the operation before the second block is filled or, barring that, that it can halt the disk read when necessary, it can finish the first and arrange for the disk to fill the first again before the second is filled. By repeating this procedure, the processing can go on simultaneously with the I/O activity, limited only by the rate at which it can be read from the disk. Although the data may still be blocked on the disk for interruption capability, the file can be searched very quickly if it exists sequentially on the disk.

If the main storage is large enough to store an entire track, the seek time and latency time for the following access can also be used to process the data. Thus the processing time can be extended into the next access without loss in performance, and can even be extended beyond the beginning of the next read operation. Of course, at this point, the danger exists that the following operation may not be completed in time.

If one such processor exists for each head on the disk, an entire cylinder can be read in and processed, the speed again limited only by (1) the rate data can be read off the disk and (2) the synchronization of all the processes in the system that are using the data.

The X-Tree environment [see chapter 4] is particularly suited for this type of operation. The synchronization required to overlap processing with the I/O operation is already necessary to allow efficient communication among cooperating processors. In addition, there is reason to think that the processing power of the nodes may be sufficient to keep up with the disk for most simple operations. For a record size of 100 bytes, if the data is being read from the disk at a rate of 800,000 bytes per second, approximately 120 micro-seconds is available for the processing of each record. If the processing required is too great for the leaf node, and if the operation to be performed can be split up, the leaf node can perform the first stage and send the remainder on to another node up the tree. A reduction in data is required, of course, since each non-leaf node may have more than one child sending data, so the leaf node must act as a data filter.

Many of the set operations on a data base can be conveniently broken up in this way. Some algorithms to do this are proposed here for the join model of Eswaran and Blasgen. [Blasgen 77] As will be shown, this technique has great advantages under many circumstances. However, the cooperation of the various processors is a significant problem. In particular, the possibility that one or more processors may not be able to do its assignment in real time dictates that a method must be capable of being stopped or aborted and restarted. For example, on a scan of a cylinder, if any processor gets behind, one solution is to use a second revolution to complete the operation. This requirement must be communicated to the processor controlling the arm. That communication may actually take longer than the time to scan the cylinder, so an overflow of data coming from any single head causes a significant increase in scan time for the entire cylinder. A better solution, when possible, is to arrange for other processors to take care of the overflow. The tree structure can handle this nicely by passing the result of an overflow situation up the tree. A parent only gets into trouble if both its children

require its help. Even then, it may be able to push the overflow up the tree farther.

5.2. The Eswaran-Blasgen Query Model

Eswaran and Blasgen [Blasgen 76], [Blasgen 77] have studied in considerable detail the properties of various join algorithms. They defined a particular operation which included a join of two relations, R and S, and a restriction, not necessarily the same, on both R and S in some field other than the join field. They examined the cases where various indices were present, where the restriction had various effects, and for various sizes of the relations. They parameterized a large number of properties of the data base, such as the number of records in each relation, the sizes of records and indices, and the proportion of records from each relation participating in an unrestricted join. They also parameterized some properties of the query, such as the percentage of the record being sought, and the proportion of the records being eliminated as a result of the restrictions. They evaluated the cost in terms of disk operations and showed that many different algorithms were best under a particular set of conditions.

That study has been expanded in three ways here to evaluate the X-Tree environment for join algorithms. First, the algorithms of Blasgen and Eswaran were expanded in a logical way to be implemented on multiple processors, each of which has too little memory to perform the operation alone. Second, new algorithms were proposed which take advantage of the multiprocessor environment present in X-Tree. Third, a pair of additional cost functions, like those of the previous chapter for measuring interprocessor communication cost, were defined to allow the analysis of the interprocessor communication for the distributed model. Initially the total amount of communication required among the nodes to implement those algorithms proposed by Eswaran and Blasgen was determined. The

danger of bottlenecks occurring in the Hypertree structure motivated the definition of a second cost function, that being the maximum total traffic occurring in a single link.

The four algorithms of Blasgen and Eswaran [Blasgen 77] are presented here. In addition, the algorithm is expanded to describe the communication among the various processors. Three additional algorithms are then developed which are particularly suited for X-Tree. For each algorithm, the cost functions for disk accesses and for interprocessor communication are derived. It is assumed in all cases that the result relation need not be collected together in any particular way, i.e., that it can exist as a distributed file, just as the initial relations are. This is certainly valid if the query is in fact only part of a larger query. If not, then the result must be collected together at the node where it is required. This cost is not directly a function of the algorithm used and has been ignored. Limits and capabilities of each algorithm are discussed and they are compared under a variety of conditions.

The Eswaran and Blasgen model poses a query on two relations, R and S. The query consisted of the following parts:

(1) Generate R' through the application of a simple restriction on some column of R.

(2) Generate S' through the application of a simple restriction on some column of S.

(3) Generate relation T, by performing the join of R and S.

(4) Project some columns of T to produce the result relation.

Some additional assumptions have been made in the case where the system is X-Tree. It has been assumed that each relation is spread across a large number of surfaces of a disk, possibly every one. It is assumed that the entire relation will

not fit on a single cylinder, but is clustered over a few cylinders, hopefully physically close. The analysis here assumes a single disk, but it can be easily extended to include multiple disks, where the relation is spread over all of them, and the index still occupies no more than one cylinder on each disk. The assumption has also been made that, in the case of the clustering index if it exists, a tuple will reside on the same surface as the index entry for that tuple. Thus, the same processor that finds a particular tuple in the index may access the tuple directly when the head is properly positioned. It is not assumed, however, that two relations which have fields capable of being joined will be aligned in such a way as to allow the join without inter-processor communication, even if each relation has a clustering index on the join field.

It has been assumed further that each processor knows how each relation is partitioned with respect to the clustering field, so that it can directly communicate with the processor having access to any given entry. In some cases this assumption may make too heavy a demand on the system. The tree organization again provides an attractive alternative for those cases where maintaining the information at every node require too much memory. If the non-leaf nodes maintain precise information only for their descendents and much cruder information about nodes in other parts of the tree, the address can be refined along the way with minimal increase in path length.

5.3. The Four Methods of Eswaran and Blasgen

The following section extends the four algorithms of Eswaran and Blasgen for a multiprocessor distributed system, including X-Tree. The Blasgen-Eswaran paper introduced a large amount of notation which has been used here unchanged whenever possible. Those parameters are summarized in Appendix A. Some additional parameters and notation are introduced in this section, resulting from

the extensions to their model. They are explained as they are introduced.

5.3.1. Method 1: Indexes on Join Columns

For this method it is assumed that indices exist for the join columns of both R and S. These two indices are scanned concurrently. When a common value is found in both indices, the tuple from one, say R, is fetched and the appropriate restriction performed. For a qualifying tuple, the scan is continued along the S index to identify all candidates to be joined with the tuple. These tuples are then fetched and, after the proper restriction and projection, stored in temporary storage. Scanning along S is now continued to find all tuples with the same key value, and these tuples are retrieved. Restriction and projection are performed, each resulting sub-tuple being concatenated with each corresponding element of the temporary storage and placed in the output relation. The temporary storage must be sufficiently large to hold all of the sub-tuples of S to be joined with any single sub-tuple of R.

This algorithm poses some difficulty for the distributed case, since the indices for R and S are not in general partitioned in the same way. Thus, portions of the two indices to be compared may be read into different processing nodes, and additional message traffic is necessary to get them to the same node. The same is also true when the tuples are fetched. If this method proves otherwise useful, however it is reasonable to think, at least for limited cases, that the relations could be maintained with a common partitioning. Given this assumption, the join indices for both R and S can be scanned concurrently, and no interprocessor communication of data is necessary at all. Unfortunately, this restriction places rather severe constraints on the data base organization and can at most be implemented on one field of a relation. Here it has been assumed that this is not possible, and that R and S are not

partitioned in the same way. Thus a substantial cost in interprocessor communication is required for the initial comparisons.

An alternative possibility which would improve the performance of method 1 is that a joint index be maintained for the join column for both R and S, i.e., each key value is followed by two lists of *TID*'s, one for R and one for S. Of course, such a join index can be built quickly from the two join indices, but this index can exist instead of the two join indices, and would in fact be smaller. Under this assumption the index entry for a particular tuple will not in general occur on the same surface with the tuple itself, at least for one index. This is one case, then, when the index entry and the tuple itself are on different surfaces even if the index is the clustering index. In this case it must be assumed that each tuple fetched from one of the relations requires an interprocessor message to request it and another to return it. The assumption has been made here that whenever both join indices exist and are the clustering indices, that the joint index is available. In addition, in this case it is also assumed that the larger relation is aligned with the index so that no interprocessor communication is required for its retrieval.

For each key for which both R and S have tuples, the tuple itself for one, say R is now fetched and the restriction performed. The processor fetching the tuple is in general different from the requesting processor if the join field is not the clustering index. In such a case, the restriction and a projection could be performed in the processor fetching the tuple, and only those tuples qualifying would be sent back to the requesting processor. A request is now sent out for each of the tuples of S having the same key value as a tuple of R surviving the restriction. Again the restriction and projection could be done in the node fetching the tuple if that is a different processor, and the

are tightly clustered on certain pages. In fact, if every other tuple of a relation participates in the join, then every page must be fetched, but the Blasgen/Eswaran assumption would estimate only half that.

Appendix D shows that the expected number of pages fetched from relation R when P_1 is small is

$$\frac{N_1}{E_1} \cdot \left\{ 1 - \frac{\binom{N_1 - E_1}{P_1 \cdot N_1}}{\binom{N_1}{P_1 \cdot N_1}} \right\}.$$

Similarly, when $P_2 \cdot F_1$ is small, the expected number of pages fetched from relation S is

$$\frac{N_2}{E_2} \cdot \left\{ 1 - \frac{\binom{N_2 - E_2}{P_2 \cdot F_1 \cdot N_2}}{\binom{N_2}{P_2 \cdot F_1 \cdot N_2}} \right\}.$$

Because the assumption was made here, and in the Blasgen/Eswaran work, that all tuples participated in the join, this difference is unimportant for relation R. Not all the tuples of relation S participate in the join, however, because a restriction is performed first. Blasgen and Eswaran made the assumption that the restriction was a simple one, *i.e.*, that it could be stated as an equality or inequality. This has the effect, if the relation is clustered on the restriction index, that the participating tuples occur consecutively in the file, and therefore the Blasgen/Eswaran assumption is correct. Unfortunately, however, this cannot be true since the assumption was that the relation is clustered on the join index. The participating tuples therefore will be scattered in an unpredictable manner throughout the pages of the relation. Define

$$\tau(p,n,e) = \frac{n}{e} \cdot \left\{ 1 - \frac{\binom{n-e}{p \cdot n}}{\binom{n}{p \cdot n}} \right\}$$

if $p < 1 - \frac{e}{n}$ and let $\tau(p,n,e) = \frac{n}{e}$ if $1 - \frac{e}{n} \leqslant p \leqslant 1$. The cost function C_D then is

$$A \cdot \left\{ \frac{N_1}{L} + \tau(P_1,N_1,E_1) + \frac{N_2}{L} + \tau(P_2 \cdot F_1,N_2,E_2) \right\}$$

The cost C_C, of interprocessor communication in terms of pages transferred, is calculated as follows: There will be $P_1 \cdot \frac{N_1}{K}$ pages of *TID*'s sent in requesting the tuples for R. The sending of those tuples will require that $P_1 \cdot N_1 \cdot F_1 \cdot \frac{H_1}{C_1}$ pages be sent in return. However, if the join indices of both are clustering indices and the joint index on the join columns of R and S exists and is partitioned in the same way as relation R, then this cost is zero. In addition, $F_1 \cdot P_2 \cdot \frac{N_2}{K}$ pages will be sent in requesting the tuples of S which will require $N_2 \cdot P_2 \cdot F_1 \cdot F_2 \cdot \frac{H_2}{C_2}$ pages to be sent in return. The average distance these pages will travel depends heavily on whether the join column indices are the clustering indices. If they are, then the average distance will be small, (zero if fully aligned, i.e., if the two relations are partitioned in the same way), though how small is hard to predict. If they are not the clustering index, then the average distance will be very near the average distance between leaf nodes in the structure, which will be called d.

5.3.2. Method 2: Sorting Both Relations

This algorithm utilizes sorting to expedite the join. Two temporary files are created. File W_1 is created by performing the restriction on R and elim-

inating all fields but those required either for the join operation or for the output relation. File W_2 is created in the same way from S. Using the join field, each of these relations is sorted. The resulting files, W_1' and W_2', are then scanned concurrently to perform the join.

The method varies somewhat, depending on the indices available. In particular, the disk cost function is determined by which of four types of access is used. These four types and γ_1, their cost for a single scan of relation R are:

(1) The restriction index is the clustering index and is used for the scan. The cost, in pages accessed from the disk, is

$$\gamma_1 = F_1 \cdot (\frac{N_1}{L} + \frac{N_1}{E_1}).$$

(2) The restriction index is used, but is not the clustering index. Thus,

$$\gamma_1 = F_1 \cdot (\frac{N_1}{L} + N_1).$$

(3) The clustering index is used, but it is not the restriction index. Here,

$$\gamma_1 = (\frac{N_1}{L} + \frac{N_1}{E_1}).$$

(4) A *segment scan* is used to access the relation. A segment is a file containing all the tuples of one or more relations. It may include pages which do not have any relevant tuples, but they will be fetched regardless during a segment scan. In this case,

$$\gamma_1 = M_1.$$

Blasgen and Eswaran stated that if an index exists for the restriction field, and it is the clustering index, then it was always used. If not, then from the other three methods that one was selected which was possible and had the minimum disk transfers.

In the distributed case, as the tuples for relations W_1 and W_2 are obtained, they are sent to the appropriate node for the selected partitioning of W_1' and W_2'. At that node they are stored on the local disk surface if necessary, and sorted to produce W_1' and W_2'.

Disk activity is minimized if the restriction index exists and is the clustering index. However, the link traffic is minimized if the join index exists and is the clustering index, since the relations W_1' and W_2' are partitioned according to the join field. When the latter is not true, then the tuples to be placed in relations W_1' and W_2' must be sent to the appropriate node, determined by their join value.

Except for that special case where the access method is the clustering index which happens to be the join field, all tuples must be exchanged in the formation of relations W_1' and W_2'. Even in this special case, it cannot be assumed that R and S relations will be aligned on the clustering index. So at best, the tuples for one relation, the larger if both otherwise qualify, need not be moved initially if the access method is the join field index and it is the clustering index.

The cost of the disk accesses is

$$C_D = A \cdot (\gamma_1 + \gamma_2) + 2 \cdot B \cdot \left\{\beta_1 \cdot \log_z \beta_1 + \beta_2 \cdot \log_z \beta_2\right\},$$

where $\beta_1 = \dfrac{N_1 \cdot F_1 \cdot H_1}{C_1 \cdot Q}$ is the number of blocks of size Q needed to sort relation W_1 and $\beta_2 = \dfrac{N_2 \cdot F_2 \cdot H_2}{C_1 \cdot Q}$ is the number of blocks of size Q needed to sort relation W_2 (see [Blasgen 77]). Also, $\log_z \beta_1$ is the number of passes necessary to sort relation W_1 and $\log_z \beta_2$ is the number of passes necessary to sort relation W_2. Z is the degree of merging in the sort, as defined by Blas-

gen and Eswaran, and γ_2 is the cost function for relation S equivalent to γ_1.

Assuming that $N_1 > N_2$, the number of pages of W_2 sent between processors is

$$\frac{F_2 \cdot N_2 \cdot H_2}{C_2}$$

in all cases (exception: if the two relations are aligned on the clustering join indices). In addition, if the clustering index is not the access path or is not the join index, then

$$\frac{F_1 \cdot N_1 \cdot H_1}{C_1}$$

pages of W_1 are transmitted as well. Thus if the relations are scattered randomly over the leaf nodes of the tree, an inter-processor cost function of

$$C_C = d \cdot (\theta_1 + \theta_2)$$

can be assigned, where

$$\theta_1 = \frac{F_1 \cdot N_1 \cdot H_1}{C_1}$$

and

$$\theta_2 = \frac{F_2 \cdot N_2 \cdot H_2}{C_2}$$

unless the access path is the clustering join index on R, in which case $\theta_2 = 0$. If S is larger than R, then θ_1 will be zero if the access path is the clustering join index on R. A sophisticated system might minimize the interprocessor traffic further by selecting the smaller of θ_1 and θ_1, rather than just selecting the smaller of N_1 and N_2.

The values determining θ_1 and θ_2 are of three types:

(1) Parameters determined by the properties of the relations: C_x, F_x, and N_x. These values usually remain constant for long periods and are frequently retained by the DBMS.

(2) Parameters determined by the query: H_x. These values can often be estimated. If an accurate estimate is important, it may be possible to sample a portion of the data base to determine their value.

5.3.3. Method 3: Multiple Passes

As with the preceding method, no particular indices are required. The tuples of S are obtained by means of a scan. A temporary file W_2 is created to hold those sub-tuples resulting from the proper restriction and projection of S. If the main storage available is insufficient to hold all of W_2, then the tuples are ordered by join value and the smallest values are retained. Relation R is now scanned and the restriction performed. As each tuple is obtained, W_2 is checked for the presence of tuples having the same join value. For each such case the resulting tuple is produced and placed in the output relation.

If the main storage was insufficient, all those tuples which were retained in previous passes are eliminated from W_2. The procedure is repeated until W_2 fits in main storage.

This method shows dramatic performance differences depending on whether W_2 fits in main storage or not. It is therefore quite important which relation is used, if the two relations vary significantly in size. Blasgen and Eswaran assumed that R was much larger than S, so this algorithm works fine. If R is smaller than S, then W_1 should be retained instead of W_2. Again, a sophisticated system could make the truly optimal choice if it could accurately estimate the sizes of W_1 and W_2. That is, it would select W_2 if $N_1 \cdot F_1 \cdot C_1 < N_2 \cdot F_2 \cdot C_2$.

In the distributed version of this algorithm, the qualifying tuples of one relation, say S, are extracted and sent to the appropriate processor, i.e., that

processor having the corresponding tuples of R, if they exist. This method has a problem, however, in that it is difficult to tell when memory is full. For the single processor case, the tuples to be thrown out are easily identified — all those smaller than a certain value. In the distributed case, each processor must keep track of its portion of the list W_2, making sure that it throws out the proper tuples. Unfortunately, this requires that the complete file W_2 be transmitted among the processors at every pass.

The possible modes of access for method 3 are exactly the same as for method 2. The same algorithm has been assumed for choosing the access path. Unlike the previous method, however, these access methods are utilized repeatedly to access W_1 and W_2. Each file is accessed β times, where

$$\beta = \frac{F_2 \cdot N_2}{P \cdot \frac{C_2}{H_2}}$$

is the number of scan passes required. Thus the total cost of disk accesses is

$$C_D = (\gamma_1 + \gamma_2) \cdot \beta$$

where γ_1 and γ_2 were defined in method 2.

Considering now the interprocessor communication costs for method 3, again as in method 2 the link traffic is minimized if the relation is clustered on the join field. The total link traffic cost, assuming that all projected sub-tuples of relation S surviving the restriction are sent to some other node of average distance d for each pass of the algorithm, is

$$C_C = \beta \cdot (\theta_1 + \theta_2) \cdot d,$$

where θ_1 and θ_2 were defined in method 2. Again, θ_1 or θ_2 may be 0 under the proper conditions.

5.3.4. Method 4: Simple TID Algorithm

This algorithm requires indices on both the restriction and join columns of both relations. It also makes use of the unique value assigned to each tuple, the *TID*. The relation R', consisting of *TID*'s of tuples satisfying the restriction for R, is generated by scanning the restriction index for R, applying the restriction, and sorting the resulting list of *TID*'s. The relation S', is produced from S in the same way. The join column indices are then scanned concurrently to produce *TID* pairs corresponding to tuples participating in the unconstrained join. As these pairs are identified, the *TID* for relation R is examined to determine if it is also present in R'. If it is, and the remaining *TID* is present in S', the two tuples are fetched and the projection performed to produce a tuple for the output relation.

A key consideration in the performance of this algorithm is whether the relations are clustered on the join column index. If they are, then the relations R' and S' are probed sequentially and the cost is much less than if they are not, in which case the relations R' and S' must be probed randomly. This makes a considerable difference in I/O operations if these temporary relations don't fit in main storage.

In the distributed case, parts of relations R' and S' accrue at each node participating in the scan of the restriction index. These parts can be partitioned and sent directly to the nodes containing the corresponding tuples of the relation. If both relations are clustered on the join field and partitioned identically, the scanning of the join indices and their comparison against R' and S' can now proceed at individual nodes without the requirement for further intercommunication. As noted earlier, however, this is assumed not to be the case. Thus the algorithm becomes complex. The join index on S is

fetched and the (KEY,TID) pairs sent to the location where the corresponding values of the R join index reside. These pieces of the index are now compared against the R join index, and matching (TID,TID) pairs are assembled to form T'. If the join index exists, the previous step is much simplified, since that index can be scanned once quickly to generate T'. These pairs making up the relation T' must now be sent to the node controlling the relevant S tuple. If the TID for the S relation exists in S', the tuple is fetched, the projection performed to eliminate unneeded fields, and the result is sent, along with the TID for the R relation, to the node containing the R relation. At that node the tuple corresponding to the TID is fetched from R, the appropriate projection performed, and the output relation is assembled.

The disk access cost function depends on whether the join column indices are the clustering indices. If they are, then

$$C_D = A \cdot \left\{ \tau(F_1,N_1,L) + \tau(F_2,N_2,L) + \frac{N_1}{L} + \frac{N_2}{L} \right.$$

$$\left. + \tau(F_1 \cdot F_2 \cdot P_1,N_1,E_1) + \tau(F_1 \cdot F_2 \cdot P_2,N_2,E_2) \right\} + \psi$$

where $\psi = 0$ if both R' and S' fit in main storage. If they don't, then an external sort is required, and

$$\psi = A \cdot \left\{ \tau(F_1,N_1,L) + \tau(F_2,N_2,L) \right\}$$

$$+ B \cdot (2 \cdot \alpha_1 \cdot \log_z \alpha_1 + 2 \cdot \alpha_2 \cdot \log_z \alpha_2 + \alpha_1 + \alpha_2),$$

where $\alpha_1 = \frac{F_1 \cdot N_1}{Q \cdot I}$ is the number of blocks of R' and $\alpha_2 = \frac{F_2 \cdot N_2}{Q \cdot I}$ is the number of blocks of S' to be sorted. If the join column indices are not the clustering indices, then

$$C_D = A \cdot \left\{\tau(F_1,N_1,L) + \tau(F_2,N_2,L) + \frac{N_1}{L} + \frac{N_2}{L}\right\}$$

$$+ B \cdot \left\{ G \cdot N_1 \cdot N_2 \cdot \max[0,(1 - P \cdot \frac{I}{2N_1 \cdot F_1})] \right.$$

$$\left. + F_1 \cdot \max[0,(1 - P \cdot \frac{I}{2N_2 \cdot F_2})]\right\}$$

$$+ A \cdot F_1 \cdot F_2 \cdot (P_1 \cdot N_1 + P_2 \cdot N_2) + \psi,$$

where again $\psi = 0$ if both R' and S' fit in main storage. Otherwise,

$$\psi = A \cdot \left\{\tau(F_1,N_1,L) + \tau(F_2,N_2,L)\right\}$$

$$+ B \cdot (2 \cdot \alpha_1 \cdot \log_z \alpha_1 + 2 \cdot \alpha_2 \cdot \log_z \alpha_2).$$

The cost of internal communication has been analyzed for method 4 in the following two cases.

(1) If both relations are clustered on the join column indices, and the joint index for the join fields of R and S is available.

(2) If neither relation is clustered on the join column index.

For case 1, four stages of the algorithm contribute to the interprocessor communication cost.

(1) From the restriction index on each relation, each *TID* satisfying the restriction is sent an average distance d to the location of the join index of R, determined from the *TID*. This results in a total cost of

$$\left(F_1 \cdot \frac{N_1}{I} + F_2 \cdot \frac{N_2}{I}\right) \cdot d.$$

(2) Each *TID* in the join index of S is sent a distance d to the location of the join index of R. The added cost is $N_2 \cdot d$.

(3) The *TID*'s to be fetched for relation S must now be sent to the appropriate node at a cost of

$$F_1 \cdot F_2 \cdot \left(P_1 \cdot \frac{N_1}{I} \right) \cdot d.$$

(4) These fetched tuples of S must now be sent back to the node from which the request was sent, at a cost of

$$F_1 \cdot F_2 \cdot \left(P_1 \cdot \frac{N_1}{E_1} \right) \cdot d.$$

For case 2, where neither relation is clustered on the join column, the interprocessor cost is composed of the following terms.

(1) From the restriction index on each relation, each *TID* satisfying the restriction is sent a distance d to the node where the corresponding tuple in the R relation will be fetched. This results again in a total cost of

$$\left(F_1 \cdot \frac{N_1}{I} + F_2 \cdot \frac{N_2}{I} \right) \cdot d.$$

(2) For each relation, the (KEY, TID) pairs in the join index are sent to the node specified in the *TID* to be checked against R' and S'. Thus the entire join indices for both R and S are sent across the network to some node. The cost for this is

$$\left(\frac{N_1}{K} + \frac{N_2}{K} \right) \cdot d.$$

(3) The surviving (KEY, TID) pairs (those present also in R' or S'), are sent to a predetermined node to be sorted and matched against the other relation, forming T'. The cost here is

$$\left(\frac{F_1 \cdot N_1}{K} + \frac{F_2 \cdot N_2}{K} \right) \cdot d.$$

(4) The request for all tuples in the output relation must be sent from the node containing part of R' and S' to the node which can fetch the tuple itself. The tuple, stripped of unwanted fields, is then returned to the

requestor. The cost for these actions is

$$F_1 \cdot F_2 \cdot \left\{ P_1 \cdot N_1 \cdot (\frac{1}{I} + \frac{H_1}{C_1}) + P_2 \cdot N_2 \cdot (\frac{1}{I} + \frac{H_2}{C_2}) \right\} \cdot d.$$

5.4. Distributed Algorithms

Several methods are developed here which take advantage of the possibilities of processing the data as quickly as it is read off the disk. A powerful idea due to Babb [Babb 79] for implementing the join operation was employed in CAFS, which was described in chapter 1. It involves the concept of hashing a field, using the resulting number as an index into a boolean array, and thus eliminating quickly most of those parts of the data base which are not required. In the following algorithms, that idea is used extensively to allow arriving at the proper subset of the needed data quickly. It is assumed that the relation is spread approximately evenly across the tracks of the cylinder and that the processors connected to those heads are the leaves of the binary tree. Each leaf node maintains a hash table which can be combined with the others quickly by migrating it up to the root of the tree or subtree, merging the two tables at each level. This is a very quick operation, requiring only that both tables be scanned in parallel. This merged table can then be propagated down the tree to the leaves very quickly if required.

The hash table must be large enough to map uniquely most of the values of the function being hashed. In general, it has been assumed that the hash table is large enough so that only one can exist in a single processor at any one time. Define T_1 to be the ratio of unique values in the join field of R to N_1, and T_2 to be the ratio of unique values in the join field of S to N_2. It will be assumed also that at least one of the relations being hashed contains all the unique values, so that for a page size of S bytes, the hash table for the join field for the unconstrained join of relations R and S must have

$$2 \cdot \max(T_1 \cdot N_1, T_2 \cdot N_2)$$

binary locations. This meets the hash table criteria of two locations per unique value.

Note that it is not true, however, that the number of unique values found in the join field of relation R after the restriction has been performed will be $T_1 \cdot N_1 \cdot F_1$ unless T_1 is very close to 1. If it is assumed that in a collection of N items there are T unique items, all equally represented, (not a very good assumption in the case here) then the probability that a given unique value will not be chosen, when X items are randomly selected, is

$$\frac{\binom{N-T}{X}}{\binom{N}{X}}.$$

This formula is derived in Appendix D. Thus, the expected number of unique values encountered would be

$$X \cdot \left\{1 - \frac{\binom{N-T}{X}}{\binom{N}{X}}\right\}.$$

It will be assumed, therefore, that the number of unique values encountered is the lesser of

(1) the total number of unique values in all the fields being hashed, or

(2) the total number of values to be hashed.

While it is to be expected that the former would usually be larger, the latter might dominate under certain conditions, for example, when the join field is a *primary key, i.e.,* all values of the join field are unique. If a restriction is performed in this situation before the join value is hashed, and the restriction is very effective, then the total number of values being hashed may be substantially smaller than the total number of unique values in all the fields being hashed, which is quite large.

When a series of tables residing in the leaves is merged, as described above, the amount of communication resulting depends in part on the location of the tables at the leaves. If U leaves have hash tables, then the amount of communication resulting will be the length of the hash table times μ, the number of links over which some form of the hash table is sent. If U is a power of 2, then $\mu = 2 \cdot (U - 1)$, since every link in the basic tree structure transmits one hash table. If U is not a power of 2, then μ is variable, depending on the precise location of the tables among the leaves. In general, the worst case is when the U tables are distributed throughout the leaves so that the entire tree must be used. It will be assumed that the U leaves having hash tables are adjacent, in which case the number of tables transmitted at each level is half the number of the level below, plus one half if that number is odd.

5.4.1. Method P1: Hashing on the Join Fields

Each cylinder of relation R is scanned, at each node the restriction is applied to those tuples read in, and the join field is hashed, the result being used as an index to set an entry in table R_i^r at node i. Table R^r is then formed by merging the individual R_i^r tables and saved in the root, or some other convenient location.

The same is now done for relation S, first forming individual tables S_i^r, then integrating them into a single table S^r. By merging S^r and R^r at the root, a new table is now formed and propagated to the leaves. Merging is accomplished with the bit-wise 'AND' of the two tables S^r and R^r. The resulting table is called $RS^{r,j}$. Each relation is now scanned again. For each relation, the restriction is applied, the join field is hashed, the result is used to find the appropriate entry in the $RS^{r,j}$ array. If the entry is present, the tuple is stripped of unneeded fields and sent to a predetermined node, based on

some criterion which guarantees that all equal join values will be sent to the same node. The hash index could be conveniently used. The sub-tuples of R and S are each collected at these predetermined nodes and sorted, forming the relations R^o and S^o respectively, which are distributed over a number of nodes evenly. (Their distribution can be guaranteed to some extent by choosing the node number from the hashing value). At each node, these two relations are scanned jointly, and the phantom tuples of the result relation, resulting from the collisions in the hash table, if any, are eliminated.

The total disk activity required is precisely two complete scans of each relation, R and S. If it is assumed that that there are V pages per track on the disk, that the relation is clustered on a small number of cylinders so that $U \cdot V$ pages of the disk can be read in during a single revolution of the disk at a cost of B, then the disk cost can be computed as

$$C_D = 2 \cdot B \cdot (\rho + \sigma),$$

where

$$\rho = \left\lceil \frac{M_1}{U \cdot V} \right\rceil$$

and

$$\sigma = \left\lceil \frac{M_2}{U \cdot V} \right\rceil$$

are the number of cylinders of the disk that must be accessed in order to scan R and S, respectively.

The interprocessor communication cost may be computed in the following way:

(1) When the hash table R' is formed, each processor except the root sends exactly one copy of the table up the tree one level. The hash table contains approximately $2Y$ binary locations, where Y is the smaller of

$$\max[T_1 \cdot N_1, T_2 \cdot N_2]$$

and

$$\max[N_1 \cdot F_1, N_2 \cdot F_2].$$

Thus a message of length $\frac{2 \cdot Y}{b \cdot S}$ is sent up from μ processors, where S is the number of bytes in a page, and b is the number of bits in a byte.

(2) The same total communication is invoked again in the formation of S^r.

(3) The same total communication is required a third time to propagate the table $RS^{r,j}$ to the leaves.

(4) On the second scan of each relation, the necessary fields of all tuples participating in the final relation are sent to an arbitrary node in the structure. Though the exact distance will be variable, it may be assumed that on the average, the length of this path will be no more than d, the average distance between nodes in the X-Tree structure. Letting ψ be the ratio of the total number of tuples sent including phantoms to the number required for the final output relation, and assuming that the restrictions on the two relations are independent, $F_1 \cdot F_2 \cdot P_1 \cdot N_1 \cdot \psi$ tuples of relation R and $F_1 \cdot F_2 \cdot P_2 \cdot N_2 \cdot \psi$ tuples of relation S survive. Thus it is concluded that the final scans of R and S will result in a total interprocessor communication of

$$d \cdot F_1 \cdot F_2 \cdot \left(\frac{N_1 \cdot P_1 \cdot H_1}{C_1} + \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right) \cdot \psi$$

pages. Thus the total interprocessor communication cost is

$$C_C = 3 \cdot \mu \cdot \frac{2 \cdot Y}{b \cdot S} + d \cdot F_1 \cdot F_2 \cdot \left(\frac{N_1 \cdot P_1 \cdot H_1}{C_1} + \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right) \cdot \psi.$$

5.4.2. Method P2: Join Index Only

This method makes use of the join index only to generate the unconstrained join. (It can be very inefficient if many of the join values exist in both R and S, but one or the other fails to qualify because of the restriction.) Table R_i^j is formed by scanning the join index of R, hashing the join field, and entering a one in those entries indexed by the hashed values. Table R^j is created at the root by merging the separate tables R_i^j as they are sent up the tree. Table S_i^j is created in the same way, and the table corresponding to the unconstrained join, RS^j, is formed by performing the bit-wise 'AND' of the tables R^j and S^j. This table is then propagated to the leaves where it is referenced during the scans of R and S, whenever a tuple is found which satisfies the restriction. If the hashed join value of that tuple is represented in the table RS^j, the tuple is projected and sent to a predetermined node. Once again, it is easy to guarantee that the tuples from R and S to be concatenated will appear at the same node. At that node the join is performed to eliminate the phantom tuples and that portion of the output relation is formed.

The disk access cost results from a complete scan of both relations R and S and a complete scan of both join indices. Thus,

$$C_D = B \cdot (\rho + \sigma + \rho_x + \sigma_x).$$

The interprocessor communication cost is calculated as follows:

(1) When the hash table R_i^j is formed, each processor except the root sends exactly one copy of the table up the tree one level. The hash table contains approximately $2Y$ binary locations, where this time Y is the smaller of

$$\max[T_1 \cdot N_1, T_2 \cdot N_2]$$

and

$$\max[N_1 \cdot P_1, N_2 \cdot P_2].$$

Thus a hash table of length $\frac{2 \cdot Y}{b \cdot S}$ pages is sent up one level from each of μ processors.

(2) The same total communication is invoked again in the formation of S^j from S_l^j.

(3) The same total communication is required a third time to propagate the table RS^j to the leaves.

(4) On the scan of each relation, the necessary fields of all tuples participating in the final relation are again sent to an arbitrary node in the structure. Again it is concluded that the final scans of R and S will result in a total interprocessor communication of

$$d \cdot F_1 \cdot F_2 \cdot \left(\frac{N_1 \cdot P_1 \cdot H_1}{C_1} + \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right) \cdot \psi$$

pages, where ψ is the ratio of the total number of tuples sent including phantoms to the number required for the final output relation. Thus the total interprocessor communication cost is

$$C_C = 3 \cdot \mu \cdot \frac{2 \cdot Y}{b \cdot S} + d \cdot F_1 \cdot F_2 \cdot \left(\frac{N_1 \cdot P_1 \cdot H_1}{C_1} + \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right) \cdot \psi.$$

5.4.3. Method P3: Hashing the TID in Join and Restriction Indices

This method requires the join and restriction index for both relations, but it requires only one scan each of the relations R and S and each of the four indices. The restriction index of R is scanned first and the hashed *TID* of qualifying tuples is entered into the boolean table R', which is merged and propagated to the leaves. The join index of R is now scanned and the hashed value of the *TID* is checked in the table R'. If the hashed *TID* is present,

the hashed value of the join field is sent to the parent node to be entered into another table, $R^{r\cdot j}$, which is also merged and saved at the root. The same thing is done to S, producing S^r and $S^{r\cdot j}$. The bit-wise intersection of $R^{r\cdot j}$ and $S^{r\cdot j}$ is now generated, and called $RS^{r\cdot j}$ representing all join values which have qualifying tuples in R and S. $RS^{r\cdot j}$ is then propagated to the leaves. The relations R and S are both scanned now, the appropriate restriction performed, and the join field hashed. If that value is present in $RS^{r\cdot j}$ then the proper fields are extracted from the relation and sent to a predetermined node. Because that node is determined by the hashed value of the join field, the portion of each result tuple coming from R and that portion coming from S will always appear at the same node. The normal join is now performed to eliminate the phantom tuples, producing the result relation.

The accesses to the disk required, using this method, include one complete scan each of R and S, and one scan each on the restriction and join indices of each relation. Again assuming that there are V pages per track on the disk, that the relation is clustered on a small number of cylinders so that $U \cdot V$ pages of the disk can be read in during a single revolution of the disk at a cost of B, then the disk cost can be computed as

$$C_D = B \cdot (\rho + \sigma + 2 \cdot \rho_x + 2 \cdot \sigma_x),$$

where

$$\rho_x = \left\lceil \frac{N_1}{L \cdot U \cdot V} \right\rceil$$

and

$$\sigma_x = \left\lceil \frac{N_2}{L \cdot U \cdot V} \right\rceil$$

are the number of cylinders of the disk that must be accessed in order to scan an index of R and S, respectively.

The cost, C_C, of the interprocessor communication is the sum of the following:

(1) Since the hashing in this algorithm is done on the *TID*, there will be as many unique values as there are *TID*'s entered in the table. Thus a table of at least

$$Y_1^{TID} = \frac{2 \cdot F_1 \cdot N_1}{b \cdot S}$$

pages is needed for table R_i^f and

$$Y_2^{TID} = \frac{2 \cdot F_2 \cdot N_2}{b \cdot S}$$

pages for table S_i^f. One table of each of these sizes is sent from $2 \cdot (U - 1)$ processors in creating R^r and S^r at the root. The same amount of traffic is required to send each of the merged tables back to the leaves after it is merged.

(2) The hash table $R_i^{f,j}$ contains approximately $2Y$ binary locations, where Y is the same as defined for P1. The hashed join value of each *TID* found to be present in R^r is sent over a single link, a total of

$$\frac{N_1 \cdot F_1}{\Lambda_1}$$

pages, where

$$\Lambda_1 = \frac{b \cdot S}{\lceil \log_2(2Y_1) \rceil}$$

is the number of hashed join values from R that fit on a page and Y_1 is the smaller of $T_1 \cdot N_1$ and $N_1 \cdot F_1$. Likewise,

$$\frac{N_2 \cdot F_2}{\Lambda_2}$$

pages are sent over one link to check for the join value in $S^{r,j}$, where

$$\Lambda_2 = \frac{b \cdot S}{\lceil \log_2(2Y_2) \rceil}$$

is the number of hashed join values from S that fit on a page and Y_2 is the smaller of $T_2 \cdot N_2$ and $N_2 \cdot F_2$.

(3) The tables $R^{r,j}$ and $S^{R,j}$ are of size $\frac{2Y}{b \cdot S}$ pages and are merged to form $R^{r,j}$ and $S^{R,j}$, each through a total of U - 2 links, at a total cost of

$$(\frac{U}{2} - 2) \cdot \frac{2Y}{b \cdot S}$$

pages.

(4) Table $RS^{r,j}$ is propagated from the root to the leaves at a cost of

$$(U - 2) \cdot \frac{2Y}{b \cdot S}$$

pages.

(5) As in algorithm P1, all result tuples must be sent to a predetermined place for the final join and elimination of phantoms. The cost again is

$$F_1 \cdot F_2 \cdot \left(\frac{N_1 \cdot P_1 \cdot H_1}{C_1} + \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right) \cdot \psi.$$

5.5. Significance of Clustered Join Index

For each of the three algorithms described using hashing, it was assumed that all the tuples of the relation R (plus some phantoms) had to be sent to an arbitrary node to be matched against the surviving tuples from S. Since the expected distance for these communications is unpredictable, it has been assumed that it was the average distance between leaf nodes in the tree. This is a rather pessimistic assumption, since it should be possible, in general, to set up the data base so that many or most of the join fields would be partially aligned. This is hard to predict, however. On the other hand, if the partitioning of one relation, say R, is known, it should be easy to arrange to send the tuples of the other rela-

tion directly to the appropriate node where their counterparts from the other relation will appear from the disk. This achieves two things.

(1) It substantially reduces the interprocessor communication, since this traffic is the bulk of the total communication required in the algorithms, and

(2) It reduces the storage requirements in the processors, since only the tuples from one relation need be saved. Of course something must be done with the output relation when it is formed, but that is true in any case.

Therefore, in the previous examples, it has been assumed that if the join index is the clustering index, then the partitioning is known by all the nodes, so that the total communication required is only that necessary to send the tuples from one relation, namely the smaller one, to the appropriate node. Thus the last term in the formula for C_C in each case is

$$F_1 \cdot F_2 \cdot \left\lceil \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right\rceil \cdot \psi,$$

instead of

$$F_1 \cdot F_2 \cdot \left\lceil \frac{N_1 \cdot P_1 \cdot H_1}{C_1} + \frac{N_2 \cdot P_2 \cdot H_2}{C_2} \right\rceil \cdot \psi.$$

5.6. Extension of Cost Function to Busiest Link

In the previous sections, the total communication among the links was derived. While this is surely the broadest measure of the requirements of the various methods, the potential for a bottleneck exists in the X-Tree structure because it is not completely symmetric around every node. In order to assess the significance of this problem, a second cost function was defined, namely, the total traffic occurring in the busiest link. This analysis can be derived with relative ease from the previous section. All messages in the algorithms considered are of four types.

(1) A set of hash tables was propagated up the tree, being merged at each level, so that every link which was connecting two ancestors of a node receiving data directly from a disk would transmit exactly one copy of the hash table (assumed to be an integral number of pages).

(2) Leaf node to parent traffic. In algorithm P3, it was necessary to send some results from a leaf node to its parent, because of memory size limits.

(3) A general exchange of data. Each processor was sending messages to various other processors. The assumption was made that the messages were randomly distributed, with average path length the same as that of a tree large enough to have U leaves.

(4) The I/O links between the leaf processors and the disk must transmit all the data read in from the disk (and written to it). In this case, the cost is per page, not per block, since the issue here is bandwidth. For numerous cases when the interprocessor communication was small, these links were the most congested.

While the I/O links may in fact have additional constraints which force them to have even lower bandwidth than inter-processor links, they will clearly limit the communication if they are also the busiest links. Therefore, they are included here whenever they have more page traffic than any other link.

While the first type of exchange contributed equally to all of the links most likely to be busy except the I/O links, the second and fourth only contributed to a specific subset of links. The third contributed much more traffic on some links than others. In Appendix B the relative amount of messages going through various classes of links is derived for the Hypertree. In short, The busiest links in a general exchange among leaf nodes are those nodes at the middle levels of the tree. In certain cases, namely, when the number of leaves in the complete subtree

is a power of 4, the horizontal and vertical links in the middle of the tree are equally busy. Otherwise, the horizontal links are the busiest. In each case, a percentage of the total message traffic passing through each node is calculated and multiplied by the total message traffic. The total traffic in each link is determined in this way, and the worst case cost function is this value.

5.7. Selection of Parameter Values

In addition to the plethora of parameter values selected by Blasgen and Eswaran, which are detailed in Appendix A, a number of new parameters were defined and estimated in this work. In addition, some of the Blasgen and Eswaran parameters were varied to reflect expectations of change due to technological progress. The rationale for modifications to Blasgen and Eswaran's values and the selection of new values is given in this section.

5.7.1. Modifications to Parameters Defined by Blasgen and Eswaran

The parameters selected by Blasgen and Eswaran have been assumed here to have the same values, with a few exceptions. The value P, the total number of pages of storage available, was assumed by them to be only 25. A more realistic number for the middle 1980's for a large main frame computer is probably between 10 and 100 times larger than that, and both the original value, and the value of 4096 were used. The assumption was that each of the U processors having a direct port to a disk surface contained $\left\lceil \frac{P}{U} \right\rceil$ pages available for sort buffers, *TID* pair lists, hash tables, etc. The only apparent significance to this seemingly very significant change is that when P was only 25, the hash tables became too large to fit in a single processor under some conditions. When P was increased to 4096, however, the problem didn't appear until nearly one million unique records occurred in a relation.

The parameter B, the cost of reading a block off the disk, was extended in the model to include the cost of reading all the relevant pages on an entire cylinder. Blasgen and Eswaran assumed that B was the same as the cost of accessing a single page, i.e., $B = A = 1$. Since it could be argued that this is no longer reasonable when an entire cylinder is read, the value of B was increased to various values. This parameter is in fact quite crucial to this analysis, since the choosing of it critically affects the relative cost of the hashing methods. Since the basic four methods cannot take advantage of the large block size, variation in the value of B has little effect on their cost. The hashing methods, on the other hand, are directly affected by this parameter, since most accesses are cylinder-at-a-time.

For a disk with V pages per track, in terms of access time, the cost of reading in an entire cylinder under no conditions is more than V times the cost of reading in a single page and generally the cost is somewhat less. In this range ($B \approx 6$) the results are not affected significantly. For large values of B ($\approx V \cdot U$), the penalty is severe, and the hashing methods are clearly worthless.

The ranges of values for the sizes of the relations (N_x) have been extended significantly, particularly toward larger values. This is clearly the direction of modern data base systems, and it might be argued that the success of an architecture such as that proposed here could accelerate this process. Thus the maximum size of the smaller relation (S) has been increased from about 30,000 to 1,000,000.

The size of the segment is significant in methods 2 and 3. For instance, if M_1 is not somewhat larger than $\frac{N_1}{E_1}$, then a segment scan almost always requires fewer disk accesses than using the clustering index, because the index

itself must be accessed. Blasgen and Eswaran did not state their assumptions about the size of the segment, so it is not possible to duplicate their results precisely. In order to consider the range of possibilities, the assumption has been made that M_1 and M_2 are each a power of 2 and that the relation will always fill up no less than half the pages in the segment.

5.7.2. Values for New Parameters

The assumption that an entire cylinder of a disk could be read at once is not a novel idea, and such an approach is feasible with current technology. A reasonable value for U, the number of heads read in parallel, is 20 or more. If multiple drives are used, this number could be much larger. This value has been assumed. The number of pages of size 4096 bytes currently possible on disks is about 5 or 6 per track. It is reasonable to expect that density improvements will increase that number in the next five years, but a conservative estimate for V of 6 is made.

For the hashing methods, an important consideration is the size of the hash table required. Too small a hash table results in many collisions which translates to a large number of phantom tuples. One of the important parameters in determining the required size for the hash table is the number of unique values present in the join field, T_x. This number was varied from 0.01 to 1.0 and, surprisingly, this was found to have very little effect on the communication. This is understandable, however, after a little thought. Since the movement of the hash tables is a small portion of the total I/O required in most cases, it is dwarfed both by the disk I/O and, in most cases, by the movement of the tuples to assemble the joined tuples.

The number of phantom tuples resulting from collisions in the hash table is a number of considerable interest. In all the algorithms, with the

organization proposed, or any other the author has been able to think of, the major element of communication is the final operation to assemble the result relation from the participating tuples from the two relations. This communication is proportional to $1 + R$, where R is the proportion of phantom tuples generated by collisions. Babb [Babb 79] derived the formula for the proportion of phantom tuples generated as a function of the number of unique values in the domain (using his terminology, r), the number of values meeting the criteria to be entered into the hash table (h), and the number of bits in the hash table (B). He showed that a substantial reduction in the number of spurious keys selected (e_h) could be gained by splitting the B bits into n separate hash tables, each of size $b = \frac{B}{n}$, and using an independent hash function to generate each hash table. He then demonstrated that

$$e_h = (1 - e^{-h/b})^n \cdot (r - h)$$

if $b >> 1$, i.e., $b \geqslant 10$. Since R, the number of phantoms expressed as a percent of those stored in the table, is just $\frac{e_h}{h}$,

$$R = \frac{e_h}{h} = (1 - e^{-h/b})^n \cdot (\frac{r}{h} - 1).$$

A tabulation of these values is given in Tables 5.1 and 5.2. As may be seen from the table, a significant number of spurious records are selected if the hash table is no larger than the number of unique elements in the field. Even for the case where the hash table is twice as large as the number of unique elements, the number of phantom records is unacceptably high. However, if the hash table is split into two tables, each half as large, the number of phantom records occurring drops dramatically. Surprisingly, even for the case where the hash table is no larger than the number of unique elements in the field, a reasonable number of phantoms occur. For the worst case, when

$100\frac{h}{r}$	Ratio of $\frac{B}{r}$					
	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8
.01	400	200	100	50	25	12.5
.1	399	200	99.9	49.9	25	12.5
1	388	196	98.5	49.4	24.7	12.4
5	344	181	92.7	46.9	23.6	11.8
10	297	163	85.6	43.9	22.2	11.2
15	256	147	78.9	40.9	20.9	10.5
20	220	132	72.5	38.1	19.5	9.88
25	190	118	66.4	35.3	18.2	9.23
30	163	105	60.5	32.5	16.9	8.59
35	140	93.5	54.8	29.8	15.6	7.95
40	120	82.6	49.5	27.2	14.3	7.32
45	102	72.5	44.3	24.6	13	6.69
50	86.5	63.2	39.3	22.1	11.8	6.06
90	10.8	9.27	6.59	4.03	2.24	1.18
99	.991	.871	.635	.394	.221	.118

Table 5.1. Percentage of spurious records R resulting from hashing collisions. Single hash table. B is the number of bits available in the hash tables. h is the number of entries to be made in the hash table. r is the number of unique values present to be hashed.

$100\frac{h}{r}$	Ratio of $\frac{B}{r}$					
	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8
.01	.639	.16	.04	.01	.0025	.000625
.1	6.34	1.59	.399	.0998	.025	.00624
1	58.5	15.2	3.88	.98	.246	.0617
5	207	62.4	17.2	4.52	1.16	.293
10	273	97.8	29.6	8.15	2.14	.549
15	277	115	38.1	11	2.96	.768
20	255	121	43.5	13.1	3.62	.951
25	224	120	46.4	14.7	4.14	1.1
30	193	114	47.5	15.7	4.53	1.22
35	164	105	47.1	16.2	4.79	1.3
40	138	95.5	45.5	16.3	4.93	1.36
45	116	85.2	43	16	4.96	1.38
50	96.4	74.8	40	15.5	4.89	1.38
90	11.1	10.5	7.74	3.91	1.46	.451
99	1.01	.972	.75	.399	.154	.0486

Table 5.2. Percentage of spurious records R resulting from hashing collisions. Two hash tables, each of size $\frac{B}{2}$. B is the number of bits available in the hash tables. h is the number of entries to be made in the hash table. r is the number of unique values present to be hashed.

approximately 30% of the records qualify to be inserted in the hash table, the number of phantoms R is around 48%. If two hash tables are used, the worst case number is only about 17%. For larger tables the numbers get even smaller. In some cases the use of more than two hash tables improves this figure as well.

The implication of this is that if the hash table is divided into two parts and two independent hash functions are used for the tables, then it is reasonable to expect that less than 20% additional traffic will result from the spurious records and this is the value that has been used for this study.

For purposes of establishing the size of the hashing table required, it is necessary to predict the number of entries to be made in the table. Although the join field may in fact be a primary key, in which case all values are unique, it is much more likely that a large amount of redundancy will be found in the join field. Most of the analysis was done with the assumption that $T_1 = T_2 = .01$. However, it has been discovered that little of the traffic in any of the algorithms results from the merging of the hash tables. Although the presence of many more unique values may make the hashing methods unwieldy because of the large table size, making storage in a single processor of even one difficult, this does not significantly affect the cost functions defined. In addition, a trade-off exists in that keeping the hash tables smaller than is desirable only increases the number of phantoms, which may slow down response, but will not cause a failure.

5.8. Interpretation of Results

The cost functions generated in the previous section were tested over a wide range of values for numerous parameters. A computer program was generated to evaluate the expressions derived in the previous sections. Some of the most

significant results are given in Appendix C. Blasgen and Eswaran considered three situations which they believed to be typical. They are:

A. Indexes exist for the join columns, but they are not the clustering indices. No indices exist for the restriction columns.

B. Indexes exist for the join columns and for the restriction columns, but none are the clustering indices.

C. There are clustering indices on the join columns, and indices on the restriction columns.

They also talked about a fourth situation and cited results from that situation, though they did not show data for it. This was the same as situation C above except that the clustering index is on the restriction column rather than the join column. This will be called situation D. These four situations have been studied in considerable detail, and results which Blasgen and Eswaran presented were compared where possible, though their results were presented only graphically, so only rough comparisons could be made. In addition, two other situations have been studied:

E. Indexes exist for the restriction columns but not for the join columns, and they are not the clustering index.

F. No applicable indices exist.

One inconsistency was noted in the work of Blasgen and Eswaran [Blasgen 77]. Although the graphs in their Fig. 4 showed that method 4 was never best, they nevertheless concluded that it was the best except for certain circumstances which they specified. The data generated here confirms their conclusions, not their presented data. Presumably, the wrong graph was printed.

5.8.1. Disk Access Cost Function

The most surprising result of the present work is that in all situations, the disk cost is higher for the methods of Blasgen and Eswaran than for the hashing methods. This is particularly true for situation A, shown in Figs. 5.1-5.4, where the difference is nearly two orders of magnitude over the range of relation sizes even for the case where $B = 6$. In situations B and C, Figs. 5.5-5.8, it is not true for some of the very small size relations, particularly if the restriction is effective, so that there are not many disk accesses at all, or if B is quite large, say 20.

5.8.2. Interprocessor Communication Cost Function

The processor communication cost was not so clear cut, at least for small relations. For cases where F_1 and F_2 are near 1, the total traffic was nearly equal for all methods except method 3, which is higher both for disk cost and interprocessor communication cost. For smaller values of F_1 and F_2, the cost for the hashing algorithms was much more uniform than methods 1-4 over the range of relation sizes studied, which is to say that it was higher for small relations and generally lower for large ones. For situation A, shown in Figs. 5.9-5.10, the hashing algorithms were clearly superior for large relations while performance was very close for small ones. For situations B and C, the hashing algorithms were clearly inferior for small relations, while cost was similar for the larger ones.

5.8.3. Busiest Link Cost Function

In order to compare the link traffic for all links — including those between the disk and processors — the number of pages transferred between the disk and the processors was calculated. The assumption was made that

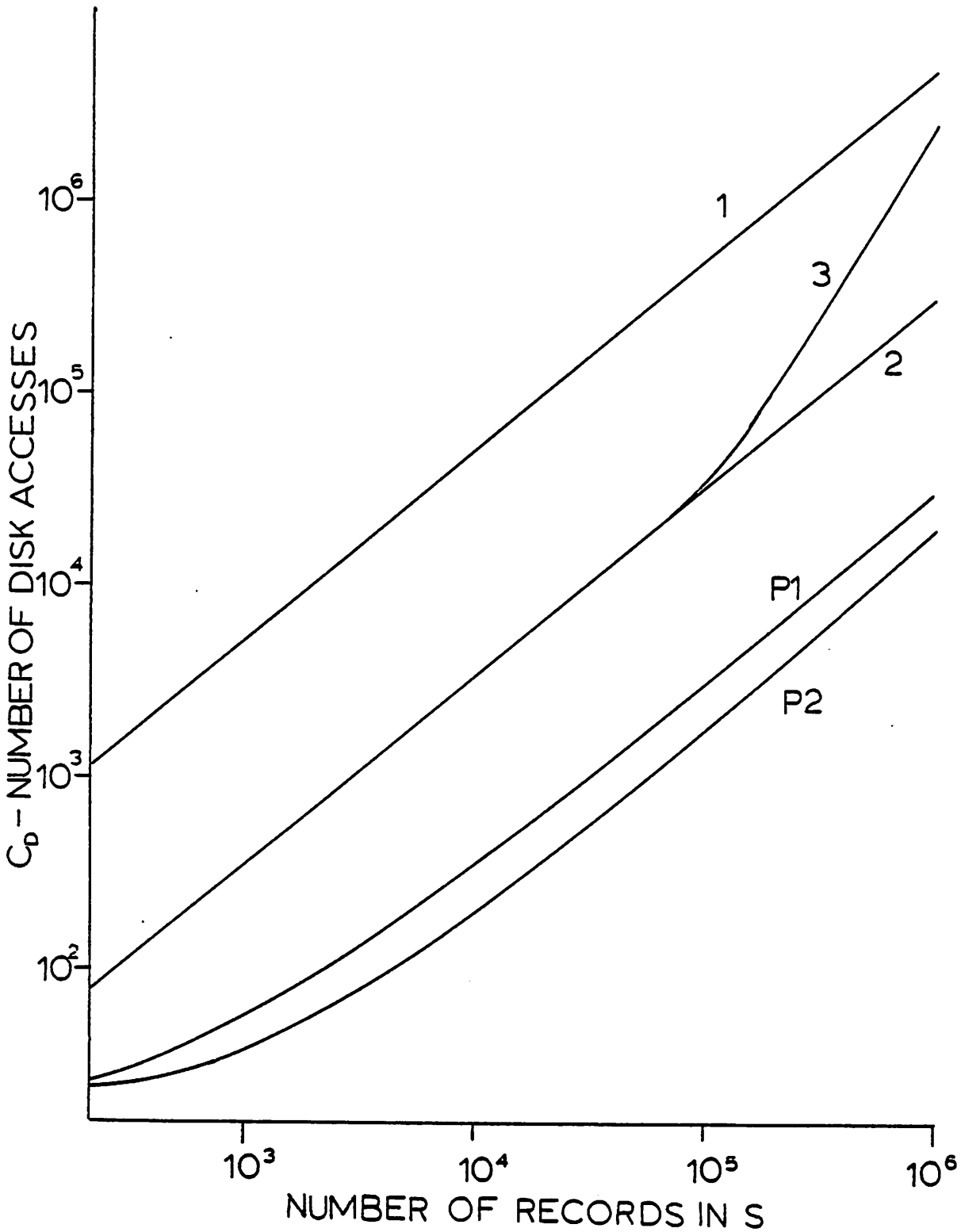

Figure 5.1. Total Disk Communication Cost for Situation A. $F_1 = F_2 = 1.0$. $B = 6$, $P = 4096$.

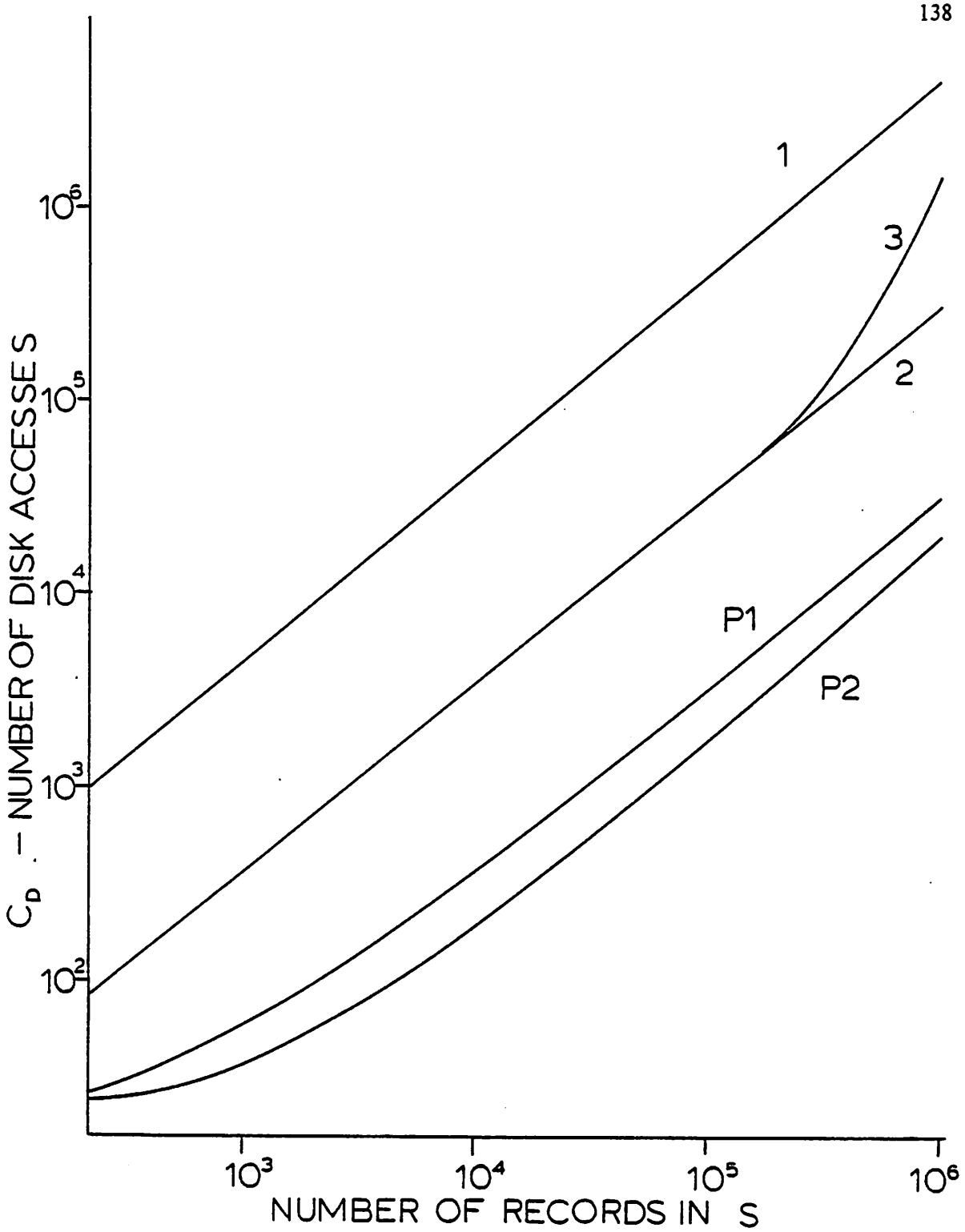

Figure 5.2. Total Disk Communication Cost for Situation A. $F_1 = F_2 = 0.5$. $B = 6$, $P = 4096$.

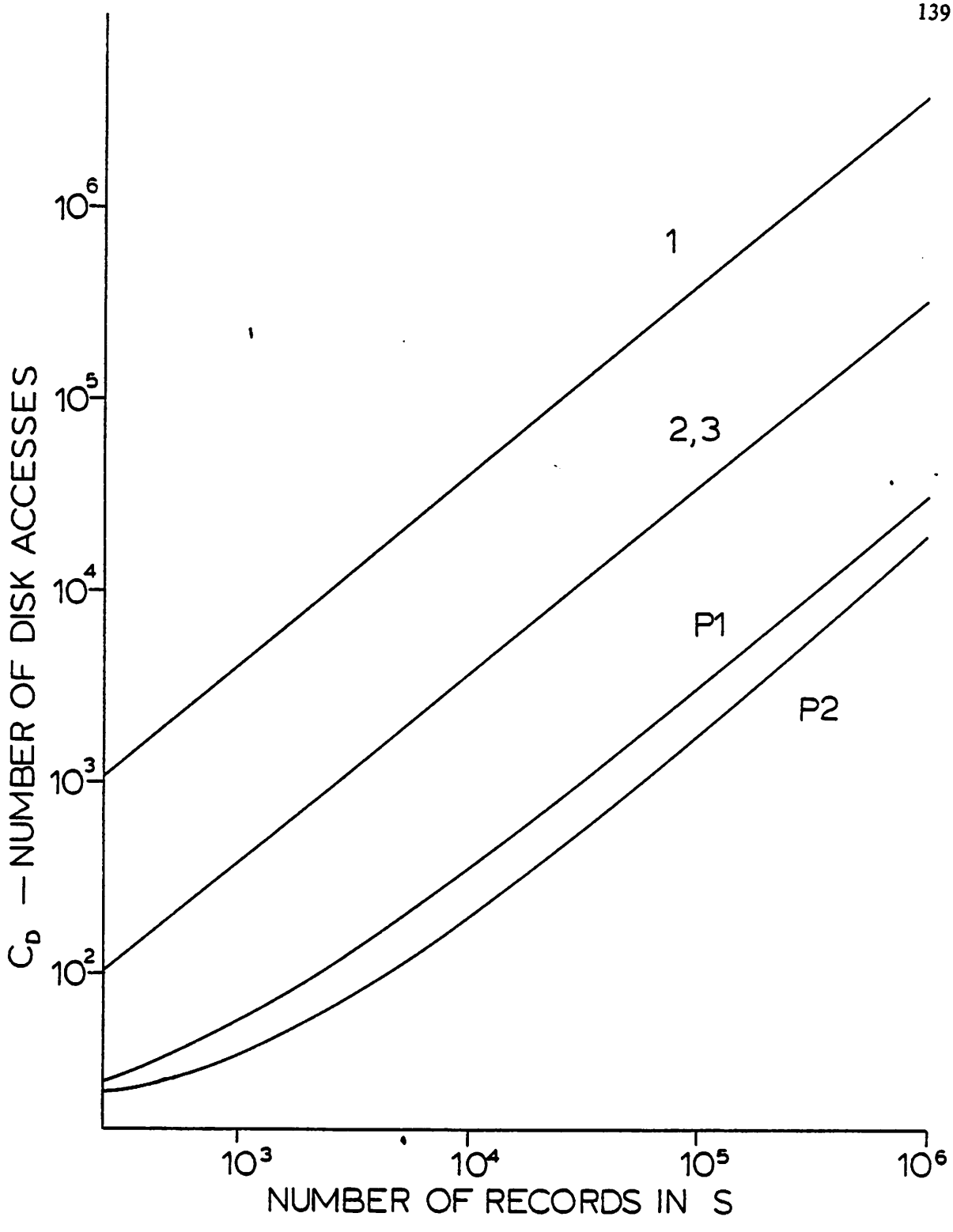

Figure 5.3. Total Disk Communication Cost for Situation A. $F_1 = F_2 = 0.1$. $B = 6$, $P = 4096$.

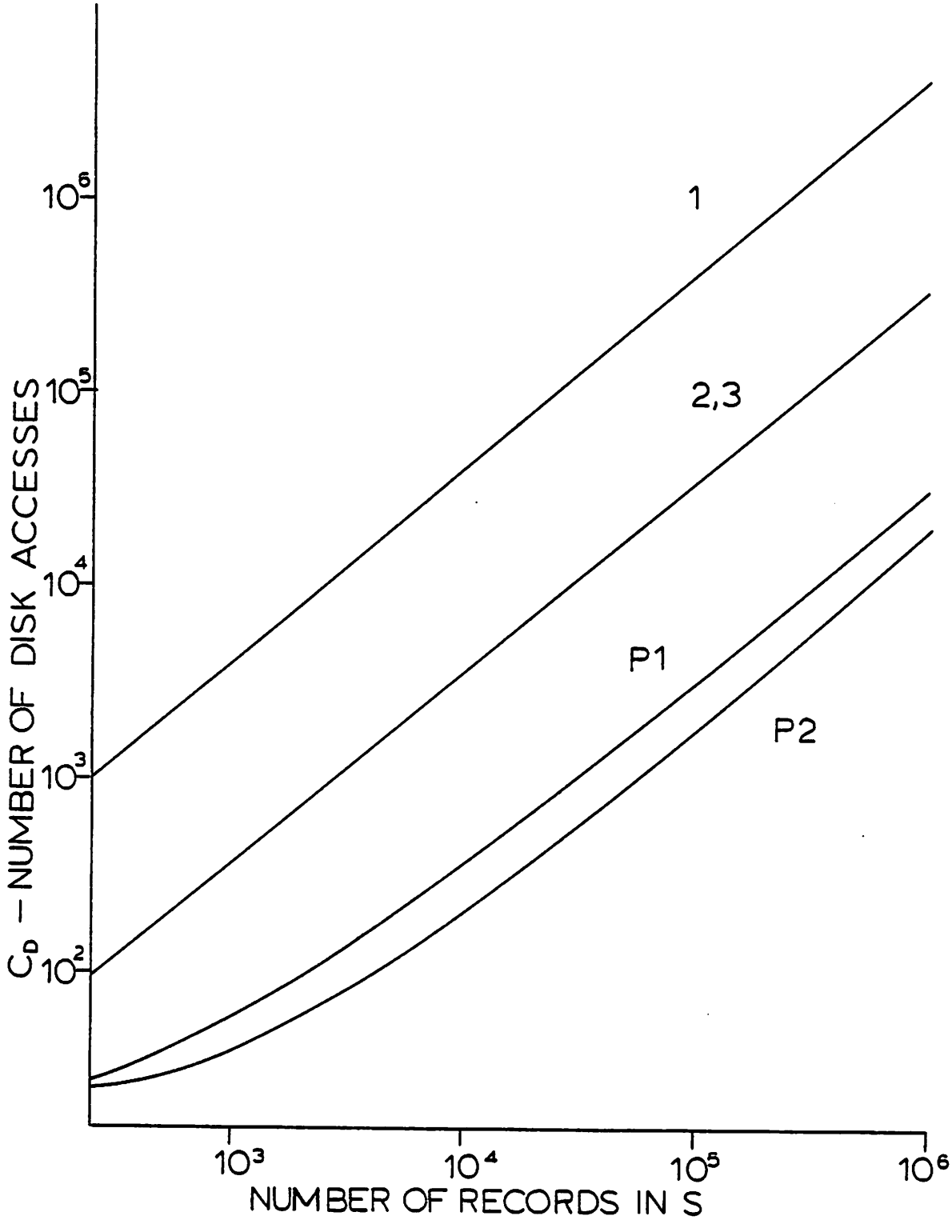

Figure 5.4. Total Disk Communication Cost for Situation A. $F_1 = F_2 = 0.01$. $B = 6$, $P = 4096$.

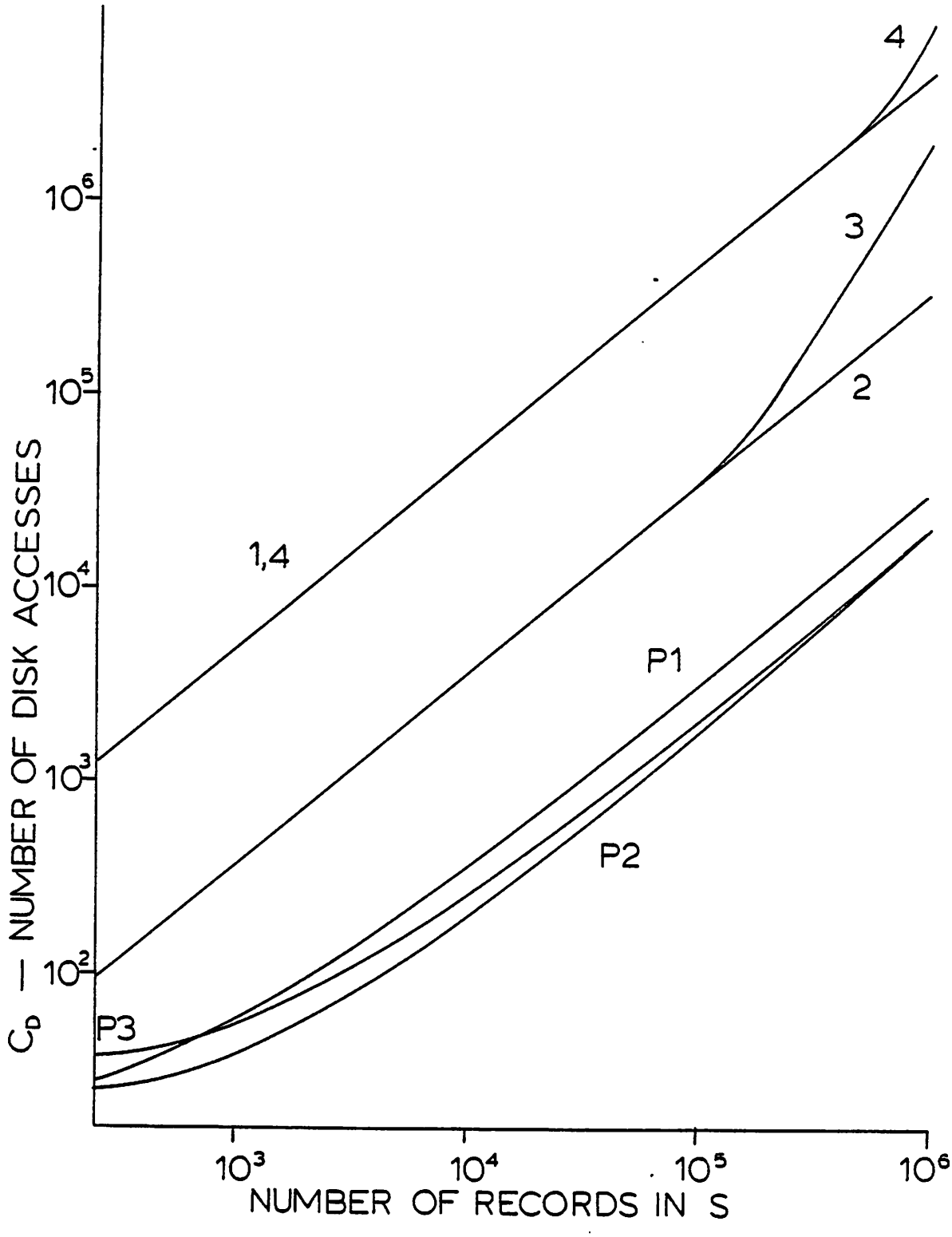

Figure 5.5. Total Disk Communication Cost for Situation B. $F_1 = F_2 = 1.0$. $B = 6$, $P = 4096$.

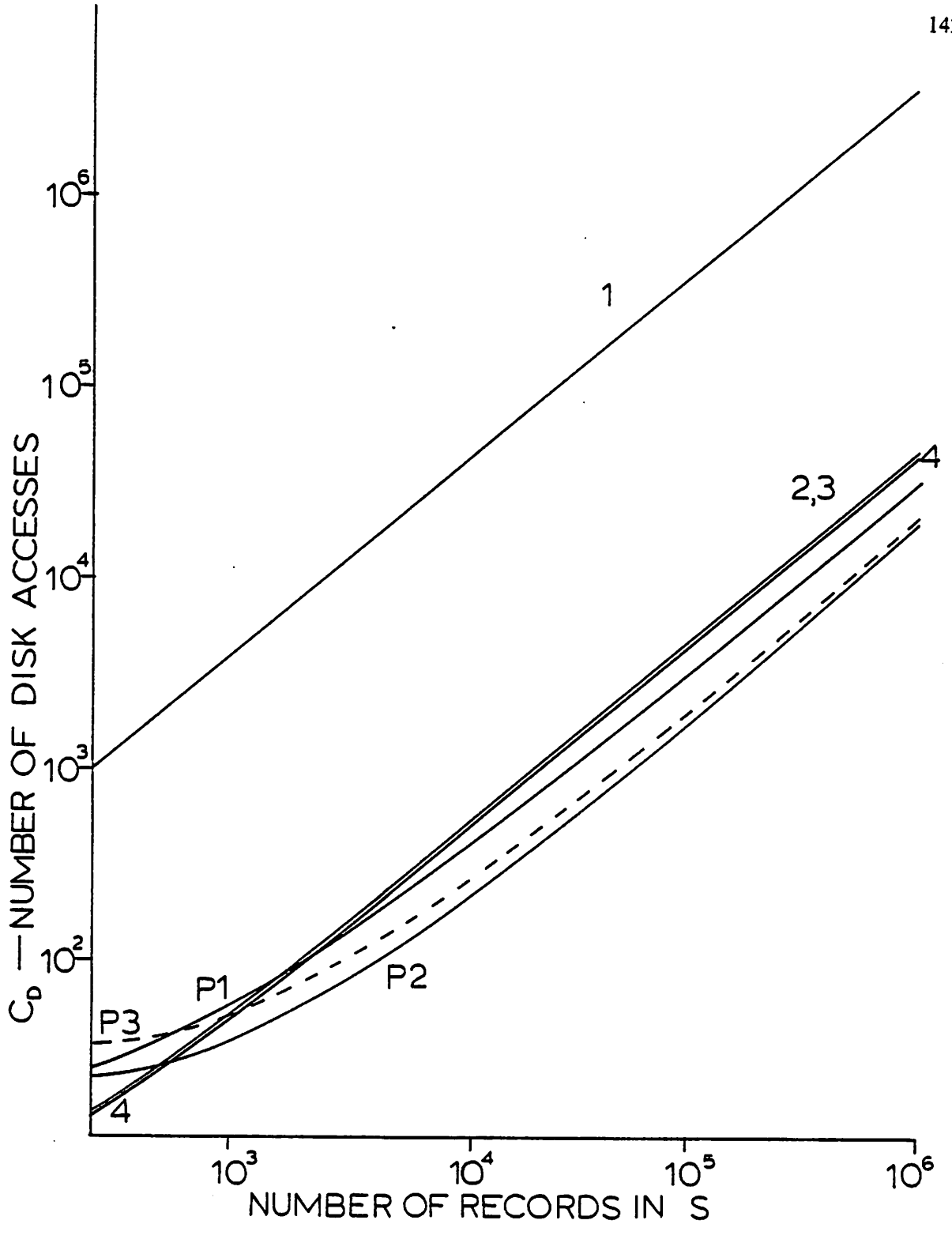

Figure 5.6. Total Disk Communication Cost for Situation B. $F_1 = F_2 = 0.01$. $B = 6$, $P = 4096$.

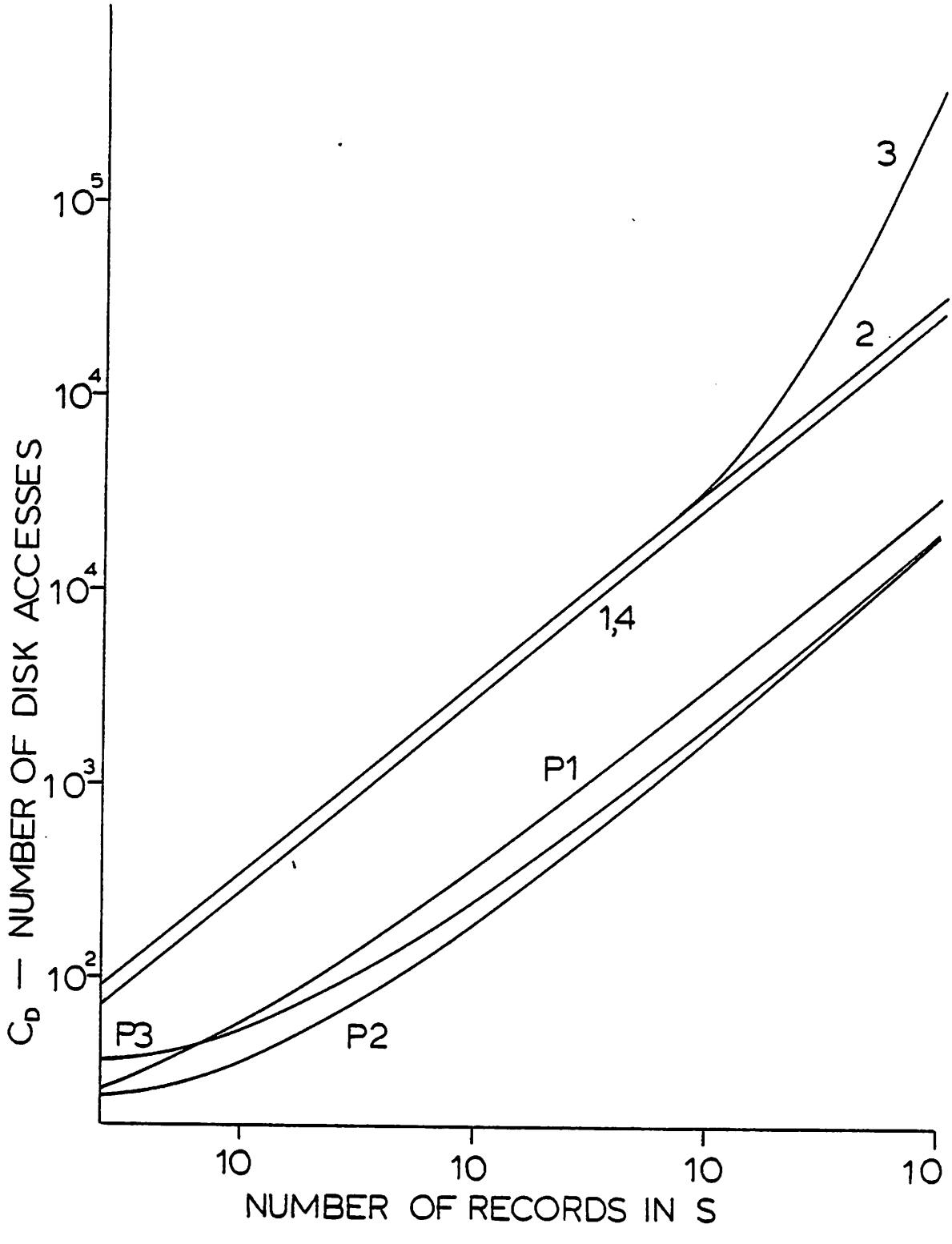

Figure 5.7. Total Disk Communication Cost for Situation C. $F_1 = F_2 = 1.0$. $B = 6$, $P = 4096$.

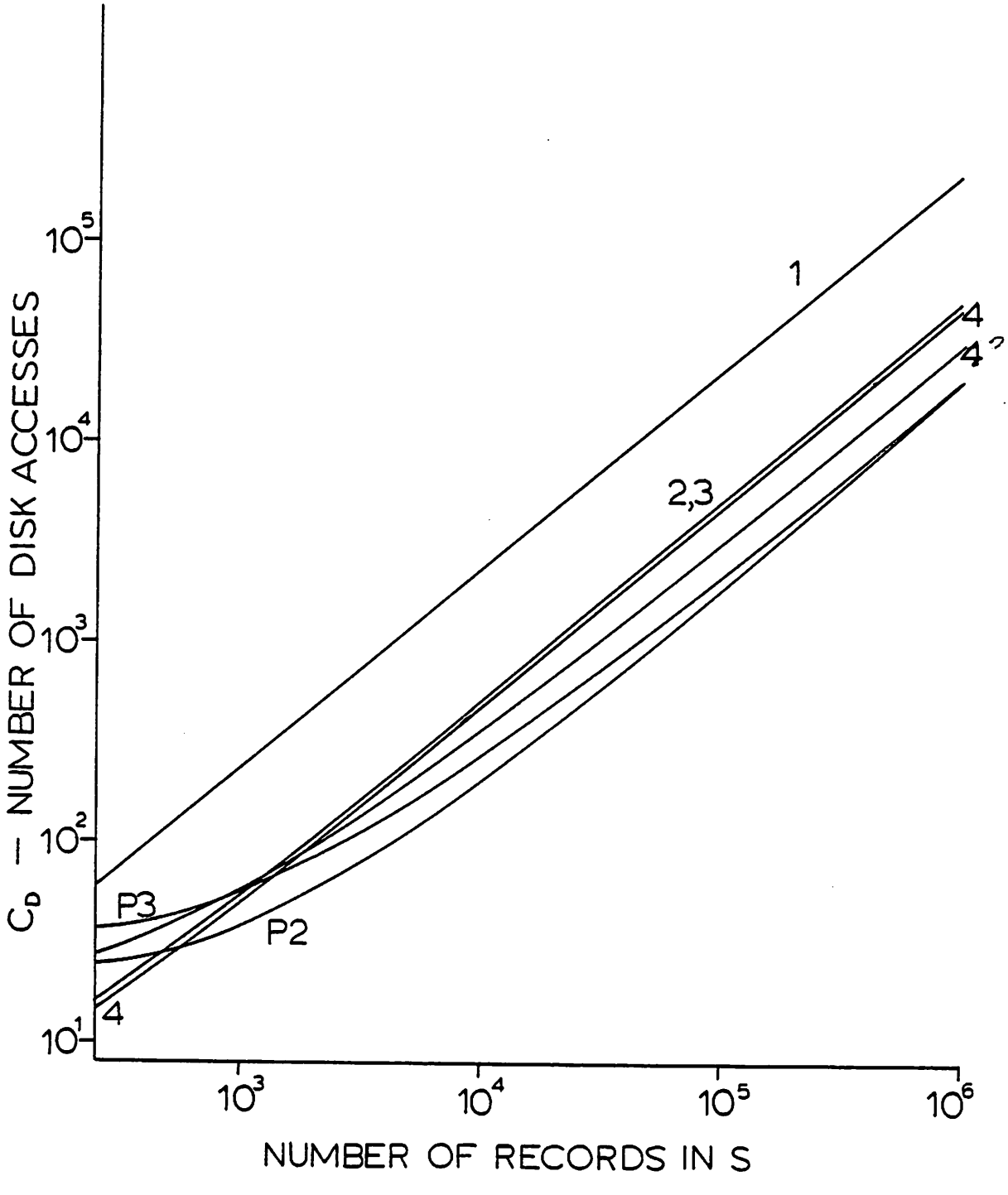

Figure 5.8. Total Disk Communication Cost for Situation C. $F_1 = F_2 = 0.01$. $B = 6$, $P = 4096$.

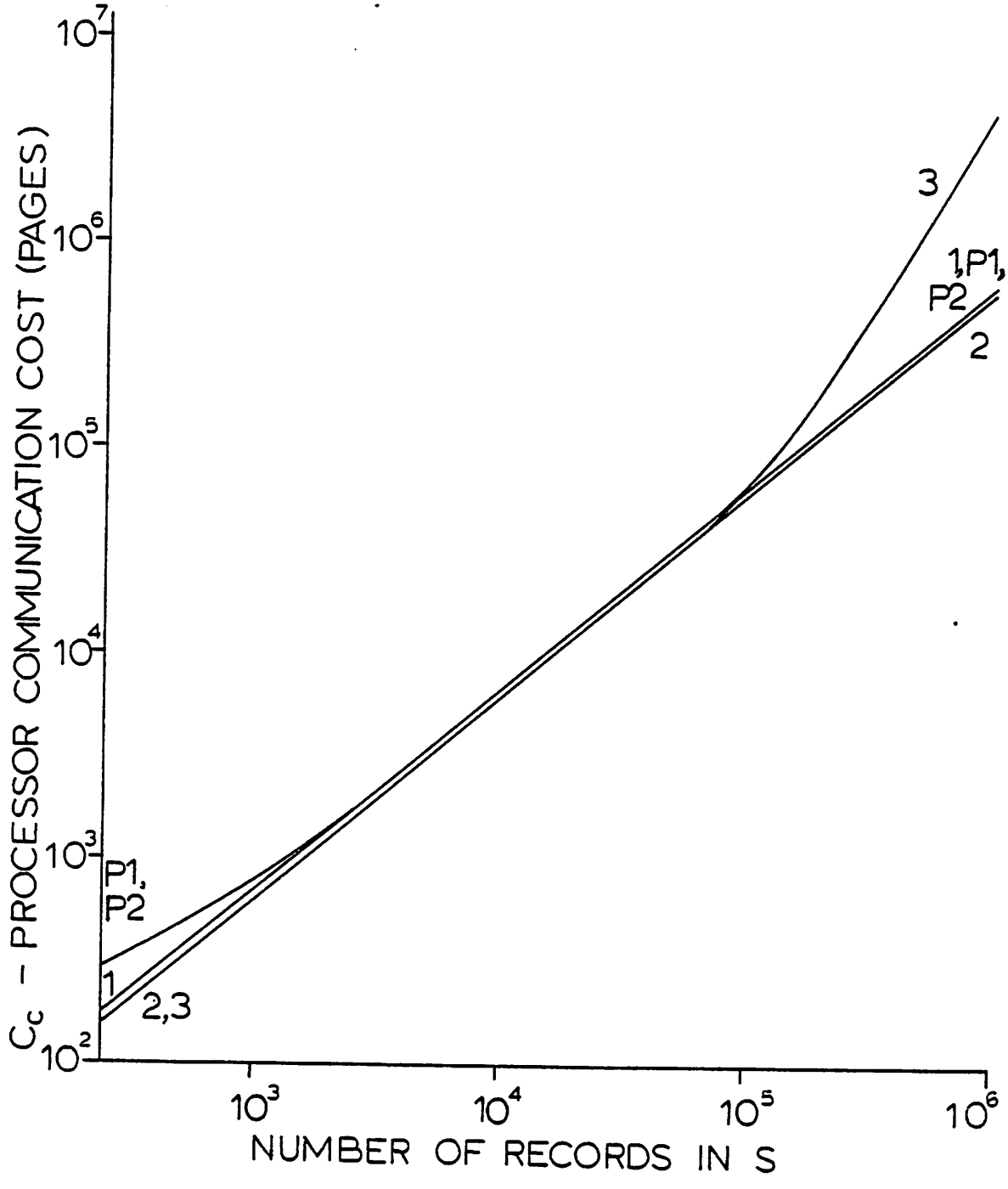

Figure 5.9. Total Processor Communication Cost for Situation A. $F_1 = F_2 = 1.0$. $B = 6$, $P = 4096$.

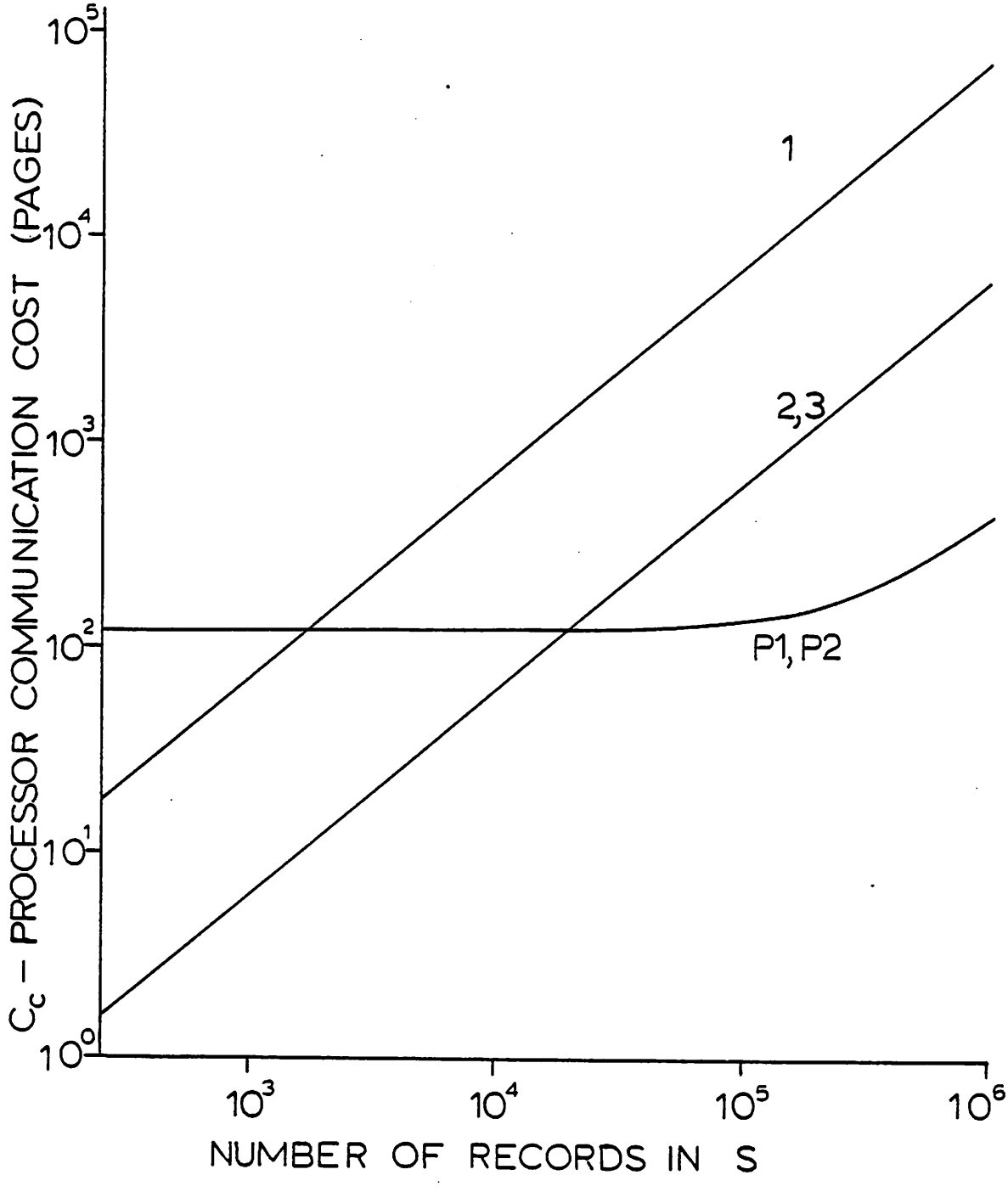

Figure 5.10. Total Processor Communication Cost for Situation A. $F_1 = F_2 = 0.01$. $B = 6$, $P = 4096$.

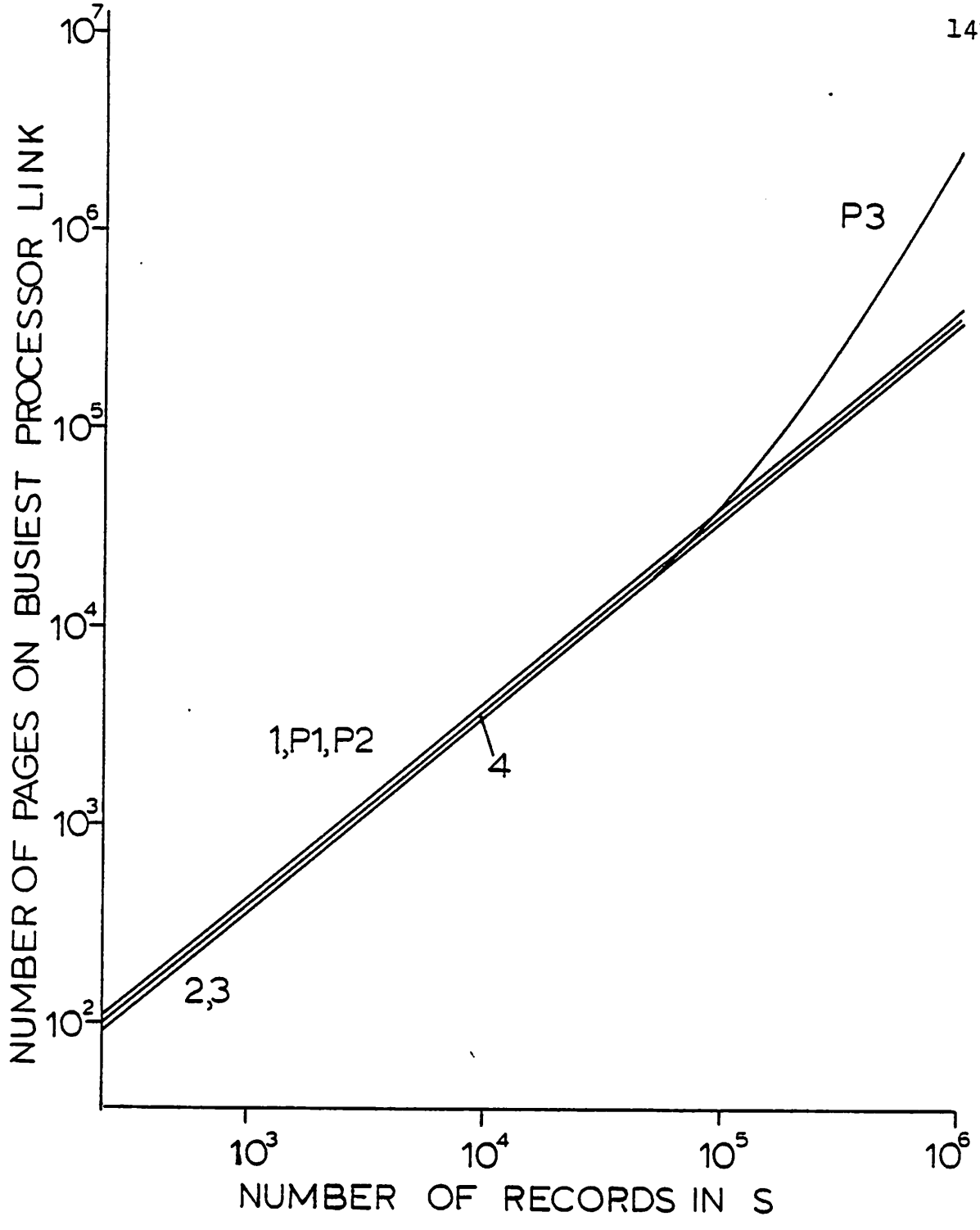

Figure 5.11. Communication Cost on Busiest Link for Situation A. $F_1 = F_2 = 1.0$. $B = 6$, $P = 4096$.

these links would carry equal traffic, so that the I/O link traffic was this number of pages divided by the number of disk heads, U. For methods 1-4 the I/O link traffic was quite low, and the links which carry the most pages were certain links connecting processors. This is misleading, however, because although the number of pages transferred for these methods was smaller than for the hashing methods, the cost of the disk transfers, C_D, is substantially larger, as shown above. This paradox results from the fact that most transfers for methods 1-4 are one page only, while for the hashing methods they are much larger, and therefore cheaper per page. Thus the methods of Eswaran and Blasgen are limited either by C_C or by C_D, either of which is worse than for the hashing methods, except for very small relations.

For the hashing algorithms in all situations, if F_1 and F_2 are each less than 0.2 then the busiest links are the links between the leaves and the disk. Of course the bandwidth between a disk head and a processor can be no higher than that between two processors, so the conclusion is that, with small F_1 and F_2 for the situations and parameters considered, at least, the traffic in the tree is not a problem, since communication is strictly limited by the disk traffic.

For the case where F_1 and F_2 were relatively large, the interprocessor links were the limiting factor. The most extreme example of this, shown in Figs. 5.11 and 5.12, is for situation B. For the best algorithm, method P2, the busiest processor link requires more than 23 times the bandwidth required of a disk link when relation R has 4 million records and relation S has 1 million records. Method 2 actually requires slightly less traffic on the busiest link, but its disk traffic is even higher than its busiest processor link, so it is probably not a good choice in this situation. In situation C, where the join index is the

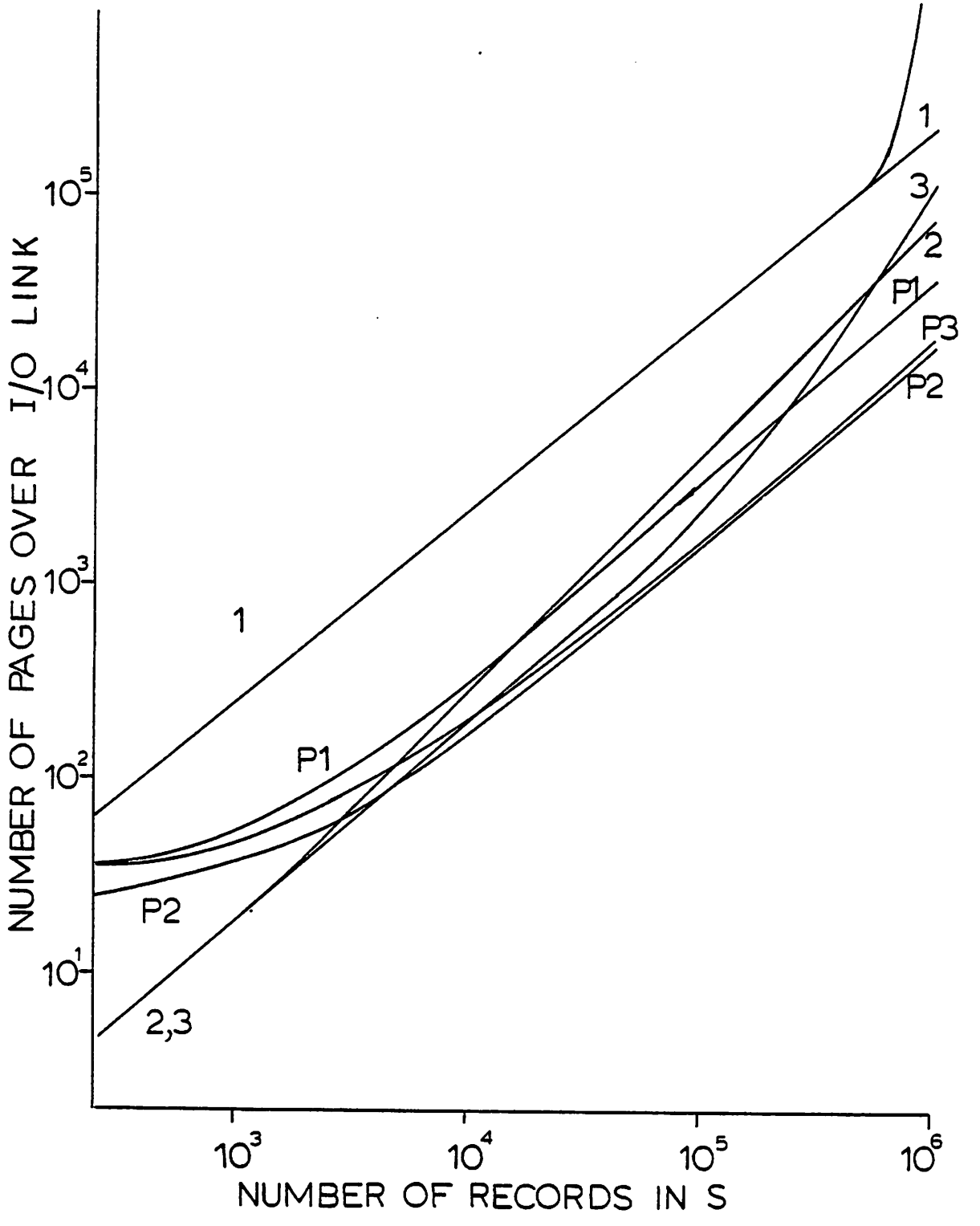

Figure 5.12. Communication Cost on I/O Link for Situation A. $F_1 = F_2 = 1.0$. $B = 6$, $P = 4096$.

clustering index, the ratio is only about 5 to 1. The severity of the bottleneck increases, of course, as the size of the data base increases. A strong argument can be made, however, that this situation is unimportant. The bottleneck results from the huge amount of traffic generated in the tree in forming the result relation, which is extremely large. The question then occurs, what will be done with this enormous relation? It is far too much to be displayed on a screen or otherwise be quickly analyzed by a human. It may possibly be a report of some type, but more likely, if it is useful at all, it will be further reduced with another program. Given the power and flexibility of a relational query language, it is reasonable to expect that a single query could be formed which would produce the smaller final result without producing the large intermediate relation.

Not surprisingly, the hashing methods are more effective in every respect when no indices at all exist.

5.9. Conclusions Regarding the Join Operation

The hashing approach has been shown to be significantly superior to the algorithms of Blasgen and Eswaran as extended to the multiprocessor environment, except for the case of very small relations. Of the three hashing algorithms, method P2 was always as good as the others for those cases for which it was applicable, and should be employed whenever possible. Method P1 is always applicable, and should be applied in those cases for which method P2 cannot be. Method P3 is never superior to method P2, and since it is never applicable when method P2 is not, it may be discarded.

The preceding analysis has demonstrated that algorithms exist which will allow the Hypertree structure to perform join operations at speeds at or near the speed limitation of the disk. This conclusion is for the range of values studied.

For a much larger relation, for example, spread over more disk surfaces, this may no longer be true, since the interprocessor communication cost is growing in proportion to $\sqrt{U}$, while the disk cost grows linearly for some methods. Nevertheless, over the range studied, the Hypertree topology is sufficient to process the data as quickly as it can be delivered from the disk. Given that it performed the best over a wide range of conditions for the problem of eliminating duplicates, it must be considered the structure of choice for the data base environment.

It has been demonstrated that the hashing algorithms proposed can show significant performance enhancement over the other methods considered, primarily because they are better able to utilize the capability to access large blocks of data sequentially. Since the hashing methods employ techniques not widely implemented to date, the potential pitfalls of this approach may not become apparent until more experience is gained in their use. The evidence presented here is sufficient to justify the building of a system to implement the hashing algorithms.

CHAPTER 6

Implications for Processor Architecture

The work discussed above has determined the algorithms and structures to support a high performance data base management system. This chapter discusses the capabilities and performance required to support the DBMS and the implications for the processor architecture.

6.1. Functional Capabilities

6.1.1. Broadcast Capability

Many of the algorithms studied require the capability to send some information to a large number, possibly all of the processors. Particularly in the case of an extraordinary event, such as when an overrun occurs and one processor is unable to complete its task in time, or if a hashing algorithm fails, so that too many records are sent to the same node, it is necessary to broadcast a message to halt the current task and restart. In addition, if the broadcast capability exists, it can be used to communicate quickly and effectively to all the processors the nature of the task to be performed.

The binary tree has certain inherent properties which make broadcast relatively easy. Of course, it is not as straightforward as a bus, since many separate messages must be created, but the hierarchal structure makes it easy to propagate quickly a message to a collection of processors. It is easy to imagine a broadcast from the root of the tree, where each node receives a message from its parent and forwards it to both children. A broadcast can be originated from any other node by taking advantage of the fact that any node of

a binary tree can be designated as root and the structure is still a binary tree. This scheme is not optimal for the X-Tree structure, or any other tree structure containing additional links, but it will work, though it fails in the case of a broken link.

A slightly less general scheme which has been considered for X-Tree is the capability to send messages to a list of nodes. Many questions have not been resolved, and it is not the intent here to resolve them, but such a capability might well satisfy the requirements identified here. Some of the relevant questions are:

(1) Should the list indicate the order in which the message was to be transmitted to the addressees?
(2) Should it be split into multiple lists whenever the routing would be different for two addressees on the list?
(3) Should it be sent as a daisy chain?

Clearly to have the effect of a broadcast, speed is important, and that probably rules out a daisy chain. The splitting of the lists is desirable however, if it doesn't create too much complexity in the routing controller.

6.1.2. Efficient Message Passing: Pipes

Interprocess communication overhead is a very serious concern in virtually any computer system of significant complexity. The INGRES work [Hawthorn 79] has shown that in the data base environment, a major portion of the processor time is devoted to the handling of such communication even with a single processor. For multiple processors the problem can only become more severe. It is clear that for such a system to perform efficiently, this problem must be solved well.

While the problem is difficult, it is by no means hopeless. The solution lies in the implementation of the appropriate low-level primitives to ensure

the efficient execution of the variety of operations which must be performed to dispatch a message. In the X-Tree environment, a simple scheduling algorithm can be implemented to run very fast, since only a small number of processes should be active at any time in a single processor. This is important in systems for which a major delay in message passing is caused by calling the scheduler. Stonebraker has pointed out [Stonebraker 79] that a special processor, which never originates messages but simply returns answers, can have a drastically lower instruction overhead compared to a more flexible protocol. He estimates, for example, that the D-cells in MUFFIN might have an overhead to send or receive a message of 100 instructions, while for the A-cells he estimates 5000. Thus there is hope that this problem can be solved. This is a very important issue and the success of the proposed system clearly depends on how well it is resolved.

Gray [Gray 78] differentiates between what he calls "one shot" messages, where he says a very rigid protocol exists, and messages via an established *session,* where many of the protocols are agreed to, thus potentially simplifying the message handling overhead.

6.1.3. Efficient String Comparisons

The operation of string comparison is used in a variety of ways in a data base management system. String comparisons are performed repeatedly in sorting and merging operations, of course. In addition, whenever a file scan is required, string comparisons are performed in large numbers. This would seem to be reason to make the string comparison operation capable of extremely efficient operation. This can be done in several ways.

(1) An instruction can be defined which allows the operation to be aborted at the earliest possible moment — for example, when a differing

character is found — without requiring an explicit test and branch.

(2) Increased buffering may be useful to allow the rapid testing of a long string.

(3) In many circumstances, the capability of comparing a single value against many values can provide significant advantages. This was used effectively in RAP [Ozkarahan 77], for example, and resulted in nearly linear speedup for join operations.

6.1.4. Family of Hashing Functions

It is well known that hashing is a useful technique for distributing data more uniformly when it does not occur that way naturally. Hashing is frequently used in data base systems today to implement efficient access methods. In addition, it has been shown that it can be used effectively to implement certain join algorithms. However, hashing functions are much less effective if they do not randomize the data well. Unfortunately, good hashing functions also require considerable computation — division, for example. Many interesting hashing functions which can be easily implemented in specially designed hardware have been virtually unused because they are so difficult to program on a conventional computer [Lum 71]. In the data base environment, it would seem clear that a hashing instruction would be quite valuable. That instruction should have these parameters:

(1) the size of the field to be hashed,

(2) the size of the hash table, and

(3) a third number to specify one of a family of hash functions to be generated.

A number of possible families of hashing functions are possible. Lum *et al* [Lum 71] concluded that the widely known division technique is best in general. They found that, in many cases, it actually performs better than a truly randomizing function because it apparently preserves the uniformity in the key set to some extent. Knuth [Knuth 73, pp. 506-513] defined a slightly more general hashing function involving only multiplication. An interesting method has been proposed based on algebraic coding theory [Peterson 57]. Analogous to the division algorithm above, it uses division by a polynomial. It was not thought applicable until recently, however, because of the difficulty of its implementation without special instructions. However, some newer machines have special instructions which can be used in this way [VAX 77] and this approach is now possible at a reasonable cost. Lum, however, did not find this method to be superior to the standard division technique.

The division approach has some limitations due to the fact that the size of the hashing table is determined, to some extent, by the value of the divisor. Knuth's multiplication technique and the polynomial arithmetic appear to have no such constraints.

6.2. Performance Requirements

6.2.1. Disk Bandwidth Requirements

For simple operations such as restriction, the dominant delay in response is the time required to retrieve the required data from the disk. It was demonstrated in chapter 5 that the join operation can be supported with the Hypertree structure for a reasonable number of disk heads in such a way that the disk transfer rate limits the response of the DBMS to the query. Thus the bandwidth of the disk is of paramount importance.

Unfortunately, the disk transfer rate is largely limited by technology. Although some increase in speed may occur in disk transfer rates, most of the increase in density has resulted from the addition of more tracks rather than denser ones, and the rotational velocity of disks is currently limited by the speed of sound. Thus it is not reasonable to expect a substantial increase in disk transfer rates in the near future. It is important, however, to take advantage of all the bandwidth that is available from the disk, *i.e.*, to make use of the data coming under every head as was done in this work.

6.2.2. Communication Rates

It has been assumed that the individual processors can send and receive data at the speed of disk transfers. For structures of the size studied here, that speed will be adequate, since for nearly all cases of interest, for the difficult operations the highest communication occurs between the disk heads and the adjacent processors. For larger networks of processors, the communication among the processors may be expected to grow more rapidly than linearly. In general, the traffic over the busiest links will grow in proportion to the square root of the number of processors, so that if the number of processors is quadrupled the busiest link traffic will double. At some point, of course, the interprocessor links will require higher transfer rates than the disk, and it is important that the processors have this higher bandwidth capability if the number of disk heads read in parallel is increased substantially.

6.2.3. Processing Rates

In chapter 5 the assumption was made that the processing required to keep up with the disk was available at each node. If that assumption is not valid now, it clearly will be at some point in the near future as the processing

speed of a single-chip computer increases with advances in semiconductor technology. A processor must be able to perform one or more comparisons, several hashing operations and move some data for each relevant record read into it. Using current disk transfer speeds of about 800,000 bytes per second, and assuming 20 tuples per page of 4096 bytes, the processor must process four tuples every millisecond. Of course, if the entire cylinder can be stored, as discussed in section 5.1, then the processor can take about twice as long to handle each tuple. This requirement is beyond the capability of current microprocessors, but mainly because they generally have no provision for hashing, so that this function would have to be implemented as a subroutine.

The algorithms developed in this work require close cooperation among a large number of processors, and in some cases created large numbers of messages, many of which might be quite small. The processor must have efficient mechanisms to send and receive messages very efficiently in order to insure that this will not limit the speed of the system.

6.3. Architectural Features Required

The features required of the processor architecture are distinctly different for the data base environment than for generaly purpose computing. Arithmetic operations are generally not used a great deal, and floating point manipulations are extremely rare. However, the sorting requirements, the join algorithms, and the restriction operation create the requirement that string comparisons be implemented efficiently. Also, arithmetic performed directly on ASCII characters, or whatever format of characters is used for the data base itself, would save multiple conversions and the possible errors they create whenever fields are modified.

In addition, the hashing algorithms derived here place additional burdens on the processor, and pose additional opportunities to modify the architecture to

support them efficiently. In particular, the use of single bits for the hashing table entries makes desirable the ability to address the memory down to the bit level. As mentioned before, a family of hashing functions is of vital importance in supporting the join algorithms developed, and these functions must be implemented to execute very quickly, because they will be used a great deal during the join operation.

The compactness of the code is important because of the limited amount of memory available for each processor in X-Tree and because of the large amount of code making up a DBMS. Through the support of special data structures for arrays of strings and hashing tables, it is probably possible to produce a storage-to-storage organization which would nevertheless have compact code. Because of the large amount of data being referenced and the limited amount of locality on that data in the DBMS environment [Hawthorn 79a], storage-to-storage operations would probably be the format of choice.

The processors operating at the leaf nodes will be essentially dedicated to the task of processing the data coming off the disk. Therefore they will probably not be required to support a high degree of multiprogramming, though it is certainly desirable for them to be able to maintain a number of processes in parallel to balance the load and handle operating system functions. This observation can perhaps be used to implement a very fast scheduling algorithm, since only a few choices exist, and hence to implement efficient system calls for message passing and other functions.

Because of the requirement that the processors handle the data fetched from the disk on the fly, it is important that the disk operations occur independently and simultaneously with the processing of the data. This means sophisticated capabilities for the I/O port and overlapped I/O.

CHAPTER 7

Summary and Conclusions

7.1. Summary

This work has investigated the nature of data base operations and identified some of the most important ones, which were expressed in the framework of a relational data base. Analysis of models of these operations was seen to be an appropriate way to evaluate the requirements they place on the architecture, and a series of models were developed to study these operations in detail.

Algorithms were developed for a number of different topologies for the elimination of duplicates problem resulting from the restriction operation and weakness of the structures were identified. The join operation was studied in great detail, and the algorithms developed by Blasgen and Eswaran were extended to operate in the multi-computer environment being studied. In addition, new algorithms, using a hashing technique were proposed and shown to perform well over a variety of conditions.

A variety of topologies were considered, and the close interaction of their structure and the algorithms employed were verified. Thus it is concluded that over the range of operations considered, the Hypertree structure [Goodman 79] provides a promising solution to the problem studied, and effective algorithms were found to implement all the operations studied on this topology.

7.2. Conclusions

DBMS systems are very important and are thus deserving of special architectural consideration. Many queries that a user would like to make are simply too

expensive on current machines, and as a result, either redundant data bases for different purposes are maintained, or those queries are answered in other ways, *i.e.*, without the use of computers, or through human interpretation of computer data. To gain improvements, many DBMS system architectures have been proposed, most apparently in an *ad hoc* fashion, since little analysis of the system choices generally appears.

An examination of query operations led to the conclusion that the restriction operation, requiring the primitive function of the elimination of duplicates, was the critical, dominant operation of a DBMS. The Hypertree structure with added links implementing the perfect shuffle interconnection among the leaves was shown to be the optimal structure for connecting a collection of identical, cooperating processors performing this task. A different structure quite possibly would be superior if specialized nodes were to be employed.

The join function was found to be a critical operation which is frequently not employed because of its cost. Therefore, principles for implementing it were derived, and new join algorithms were derived and shown to be superior to more traditional algorithms. Especially important is the proper use of hashing techniques to implement the join algorithm in linear time.

The required capabilities of the node processors were found to be about as would be expected: general purpose capabilities with optimized performance on interprocess communications of messages (including broadcast capabilities), efficient string comparison operations, and support for hashing with a family of hashing functions and bit addressability of the memory. No exotic features, such as associative memories, 'logic-in-memory,' special I/O devices, or other novel hardware were found to be useful, though some might be useful in the implementation of the required functions if they are found to be sufficiently inexpensive.

Also, to maximize I/O bandwidth, which was found to be the limit on performance, it is necessary to read from all the heads of the disk at once.

In general, it appears that a very powerful, high performance DBMS can be constructed from a large Hypertree array of micro-computers and standard magnetic disk drives, using the principles and techniques developed in this dissertation.

7.3. Future Research

This dissertation considered the theory of DBMS machines for high performance on queries. Updating the underlying data base is generally a minor extension to this work, since the major task is identifying the data to be modified. However, a number of related issues become important when the data base is being changed. In particular, recovery of the DBMS after a system crash is enormously complicated by the fact that a complicated transaction may have been partially completed, leaving the data base in an inconsistent state. Concurrency also becomes a problem only when the data base is being changed, and provisions for locking out readers may result in many potential cases of deadlock.

The techniques and principles developed here could be extended to this class of DBMS machines. Important issues of update synchronization, crash recovery, and incremental locking could be attacked using the methodology employed here.

References

[Anderson 76] G. A. Anderson and R. Y. Kain, "A content-addressed memory designed for data base applications," *Proceedings of the International Conference on Parallel Processing,* (August 1976), pp. 191-195.

[Astrahan 76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management," *ACM Transactions on Database Systems,* Vol. 1, No. 2, (April 1976), pp. 97-137.

[Babb 79] E. Babb, "Implementing a relational database by means of specialized hardware," *ACM Trans. Database Syst.* Vol. 4, No. 1, (March 1979), pp. 1-29.

[Banerjee 76] J. Banerjee and D. K. Hsiao, "The Architecture of a Database Computer — Part I: Concepts and Capabilities," Technical Report OSU - CISRC - TR - 76 - 1, The Ohio State University, Columbus, Ohio, (September 1976).

[Banerjee 78] J. Banerjee, D. K. Hsiao, and R. I. Baum, "Concepts and capabilities of a database computer," *ACM Transactions on Database Systems,* Vol. 3, No. 4, (December 1978), pp 347-384.

[Banerjee 79] J. Banerjee and D. K. Hsiao, "DBC — A Database Computer for Very Large Databases" *IEEE Transactions on Computers,* (June 1979), pp. 414-429.

[Batcher 68] K. E. Batcher, "Sorting networks and their applications," *Proc. AFIPS Spring Joint Computer Conference,* (1968), pp. 307-314.

[Berra 74] P. B. Berra, "Some problems in associative processor applications to data base management," *Proceedings of AFIPS National Computer Conference,* Vol. 43, (May 1974), pp. 1-5.

[Berra 76] P. B. Berra and A. K. Singhania, "A multiple associative memory organization for pipelining a directory of a very large data base," *Digest of Papers from COMPCIN 76,* (February 1976), pp. 109-112.

[Blasgen 76] M. W. Blasgen and K.P. Eswaran, "On the evaluation of queries in a data base system," *IBM Research Report RJ1745,* IBM Research Laboratory, San Jose, California 95193, (April 1976).

[Blasgen 77] M. W. Blasgen and K.P. Eswaran, "Storage and access in relational data bases," *IBM Systems Journal,* No. 4, (1977), pp. 363-377.

[Bush 76] J. A. Bush, G. J. Lipovski, S. Y. W. Su, J. K. Watson, and S. J. Ackerman, "Some implementations of segment sequential functions," *Proceedings of the Third Symposium on Computer Architecture,* (January 1976), pp. 178-185.

[Canaday 74] R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Ryder, and L. A. Wehr, "A back-end computer for data base management," *Communications of the ACM,* Vol. 17, No. 10, (October 1974), pp. 575-582.

[Chamberlin 74] "D. D. Chamberlin and R. F. Boyce, "SEQUEL: A structured English query language," *Proceedings of the ACM SIGFIDET Workshop,* Ann Arbor, Michigan, (May 1974), pp. 249-264.

[Chang 78] H. Chang, "On bubble memories and relational data base," *Proceedings of the 4th International Conference on Very Large Data Bases,* Berlin, (September 1978), pp. 207-229.

[CODASYL 71] "Data Base Task Group Report," Association for Computing Machinery, New York, New York, (1971).

[Codd 70] E. F. Codd, "A relational model for data for large shared data banks," *CACM 13,* No. 6, (June 1970), pp. 377-397.

[Copeland 73] G. P. Copeland, G. J. Lipovski, and S. Y. W. Su, "The architecture of CASSM: A cellular system for non-numeric processing," *Proceedings of the First Annual Symposium on Computer Architecture,* (December 1973), pp. 121-128.

[Copeland 74] G. P. Copeland, "A cellular system for non-numeric processing," Tech. Rep. No. 1, Project CASSM, University of Florida, Gainsville, Florida, (1974).

[Copeland 77] G. P. Copeland, *Private communication,* (1977).

[Coulouris 72] G. F. Coulouris, J. M. Evans, and R. W. Mitchell, "Towards content-addressing in data bases," *The Computer Journal,* Vol. 15, No. 2, (February 1972).

[Date 75] C. J. Date, *An Introduction to Database Systems,* Addison-Wesley Publishing Co., Reading, Massachusetts, (1975).

[DeFiore] C. R. DeFiore and P. B. Berra, "A quantitative analysis of the utilization of associative memories in data management," *IEEE Transactions on Computers,* Vol. c-23, No. 2, (February 1974), pp. 121-132.

[Despain 78] A. M. Despain, and D. A. Patterson "X-TREE: A tree structured multi-processor computer architecture", *Proceedings of the Fifth Annual Symposium Computer Architecture,* (April 1978), pp. 144-150.

[Despain 79] A. M. Despain, "THE COMPUTER AS A COMPONENT: Powerful Computer Systems from Monolithic Microprocessors," *unpublished paper,* (1979).

[DeWitt 78] D. J. DeWitt, "DIRECT - A multiprocessor Organization for supporting relational data base management systems" *Proceedings of the Fifth Annual Symposium Computer Architecture,* (April 1978), pp. 182-189.

[DeWitt 78a] D. J. DeWitt, *Private communication,* (April 1978).

[Dolotta 76] T. A. Dolotta, M. I. Bernstein, *et al, Data processing in 1980-1985,* A. Wiley Interscience Publication, (1976), p. 191.

[Dugan 66] J. A. Dugan, R. S. Green, J. Minker, and W. E. Shindle, "A study of the utility of associative memory processors", *Proceeding of the ACM National Conference* (August 1966), pp. 347-360.

[Edelberg 76] M. Edelberg, and L. R. Schissler, "Intelligent Memory," *Proceedings of the National Computer Conference,* (1976), pp. 393-400.

[Epstein 80] R. Epstein, "Analysis of distributed data base processing strategies," Ph. D. Dissertation, also available as Memorandum No. UCB/ERL M80/25, Electronics Research Laboratory, University of California, Berkeley, (April 1980).

[Eswaran 76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and L. I. Traiger, "On the notions of consistency and predic at locks in a data base system," *Communications of the ACM,* Vol. 19, No. 11, (November 1976), pp. 624-633.

[Frost 73] *Markets for Data-Base Services,* Frost and Sullivan Inc., New York, (July 1973).

[Fuller 65] R. H. Fuller, R. W. Bird, and R. M. Worthy, "Study of associative processing techniques", Defense Documentation Center, AD-621516, (August 1965).

[Goldberg 62] J. Goldberg and M. W. Green, "Large file for information retrieval based on simultaneous interrogation of all items", *Large-capacity Memory Techniques for Computing Systems,* pp. 63-67, Macmillan, New York, New York, (1962).

[Goodman 79] J. R. Goodman and C. H. Sequin "HYPERTREE: A multiprocessor interconnection topology," submitted to *Communications of the ACM,* (1979).

[Gray 78] J. Gray, "Notes on data base operating systems," *IBM Research Report RJ2188,* IBM Research Laboratory, San Jose, California 95193, (February 1978).

[Hawthorn 79] P. Hawthorn and M. Stonebraker, "Use of technological advances to enhance data base management system performance," Memorandum No. UCB/ERL M79/3, Electronics Research Laboratory, University of California, Berkeley, (January 1979).

[Hawthorn 79a] P. B. Hawthorn, "Evaluation and enhancement of the performance of relational database management systems," Ph.D. Dissertation, also available as Memorandum No. UCB/ERL M79/70, Electronics Research Laboratory, University of California, Berkeley, (November 1979).

[Healy 72] L. D. Healy, G. J. Lipovski, and K. L. Doty, "The architecture of a context addressed segment-sequential storage", *Proceedings of the Fall Joint Computer Conference,* Vol. 41, part 1, (1972), pp. 691-702.

[Held 75] G. D. Held, M. R. Stonebraker, and E. Wong, "INGRES – A relational data base system," *Proceedings of AFIPS National Computer Conference,* Vol. 44, (May 1975), pp. 409-416.

[Held 78] G. Held and M. Stonebraker, "B-trees re-examined," *Communications of the ACM,* Vol. 21, No. 2, (February 1978), pp. 139-143.

[Hoagland 76] A. S. Hoagland, "Magnetic recording storage," *IEEE Transactions on Computers,* C-25, Vol. 12, (December 1976), pp. 1283-1288.

[Hollaar 75] L. A. Hollaar, "A list merging processor for inverted file information retrieval systems," Technical Report UIUCDCS-R-75-762, Department of Computer Science, The University of Illinois at Urbana-Champaign, Urbana, Illinois, (October 1975).

[Hollaar 77] L. A. Hollaar, "A specialized architecture for textual information retrieval," *Proceedings of AFIPS National Computer Conference,* Vol. 46, (May 1977), pp. 697-702.

[Hsiao 76] D. K. Hsiao and K. Kannan, "The Architecture of a Database Computer – Part III: The Design of the Mass Memory and its Related Processors," Technical Report OSU - CISRC - TR - 76 - 3, The Ohio State University, Columbus, Ohio, Dec. 76.

[Hsiao 77] D. K. Hsiao and S. F. Madnick, "Database machine architecture in the context of information technology evolution," *[Proceedings on the 3rd International Conference on Very Large Data Bases,* (October 1977), pp. 63-84.

[Hsiao 78] D. K. Hsiao, "Data base machines are coming, data base machines are coming!" *Computer,* Vol. 12, No. 3, (March 1979), pp. 7-9.

[Kannan 76] K. Kannan, D. K. Hsiao, and D. S. Kerr, "The Architecture of a Database Computer – Part II: The Design of the Structure Memory and its Related Processors," Technical Report OSU - CISRC - TR - 76 - 2, The Ohio State University, Columbus, Ohio, (October 1976).

[Katzan 71] H Katzan, *Computer Organization and the System/370,* Van Nostrand Reinhold Co., New York, New York, (1971), pp. 133.

[Kerr 79] D. S. Kerr, "Data base machines with large content-addressable blocks and structural Information processors," *Computer,* Vol. 12, No. 3, (March 1979), pp. 64-79.

[Knuth 73] D. E. Knuth, *The Art of Computer Programming,* Vol. 1: *Fundamental Algorithms,* Addison-Wesley, (1973).

[Knuth 73a] D. E. Knuth, *The Art of Computer Programming,* Vol. 3: *Sorting and Searching,* Addison-Wesley, (1973).

[Lin 76] C. S. Lin, D. C. P. Smith, and J. M. Smith, "The design of a rotating associative memory for relational database applications", *ACM Transactions on Database Systems,* Vol. 1, No. 1, (March 1976), pp. 53-65.

[Lipovski 78] G. J. Lipovski, "Architectural features of CASSM: a context addressed segment sequential memory," *Proceedings of the Fifth Annual Symposium Computer Architecture,* (April 1978), pp. 31-38.

[Linde 73] R. R. Linde, R. Gates, and T. Peng, "Associative processor applications to real-time data management," *NCC,* (1973), pp. 187-195.

[Liu 79] "Intelligent magnetic bubble memories and their applications in data base management systems," *IEEEE Transactions on Computers,* Vol. c-28, No. 12, (December 1979), pp. 888-906.

[Love 73] H. H. Love, "An efficient associative processor using bulk storage," *Sagamore Computer Conference on Parallel Processing,* (1973), pp. 103-112.

[Lowenthal 76] E. I. Lowenthal, "The backend computer, part I and II," *Auerback (Data Management) Series 24-01-04 and 24-01-05,* (1976).

[Lowenthal 77] E. I. Lowenthal, "A Survey – The application of data base management computers in distributed systems," *Proceedings of the Third International Conference on Very Large Data Bases,* Tokyo, Japan, (October 1977).

[Lum 71] V. Y. Lum, P. S. T. Yuen, and M Dodd, "Key-to-address transform techniques: a fundamental performance study on large existing formatted files," *Communications of the ACM*, Vol. 14, No. 4, (April 1971), pp. 228-239.

[Madnick 75] S. E. Madnick, "INFOPLEX – Hierarchical decomposition of a large information management system using a microprocessor complex," *Proceeding of the National Computer Conference*, (1975), pp. 581-586.

[Maller 78] V. A. J. Maller, "An architecture for a data base machine," unpublished manuscript, International Computers Limited, Research and Advanced Development Centre, Fairview Road, Stevenage, Hertfordshire, England, (1978).

[Marill 75] T. Marill and D. Stern, "The datacomputer – A network data utility," *Proceedings of the National Computer Conference*, (1975), pp. 389-395.

[Martin 75] James Martin, *Computer Data-Base Organization*, Prentice-Hall, Englewood Cliffs, New Jersey, (1975).

[McGregor 76] D. R. McGregor, R. G. Thomson, and W. N. Dawson, "High performance hardware for database systems," *Systems for Large Data Bases*, North Holland Publishing Company, (1976), pp. 103-116.

[Minsky 72] N. Minsky, "Rotating storage devices as partially associative memories," *Proceedings of the Fall Joint Computer Conference*, (1972), pp. 587-595.

[Moulder 73] R. Moulder, "An implementation of a data management system on an associative processor," *Proceedings of the National Computer Conference*, (1973), pp. 171-175.

[Noyce 77] R. N. Noyce, "Microelectronics," *Scientific American*, Vol. 237, No. 3, (September 1977), pp. 63-69.

[Ozkarahan 75] E. A. Ozkarahan, S. A. Schuster, and K. C Smith, "RAP – Associative processor of database management," *Proceedings of AFIPS National Computer Conference*, Vol. 44, (May 1975), pp. 379-388.

[Ozkarahan 77] E. A. Ozkarahan, S. S Schuster, and K. C. Sevcik, "Performance evaluation of a relation associative processor," *Transactions on Database Systems*, Vol. 2, No. 2, (June 1977), pp 175-195.

[Parhami 72] B. Parhami, "A highly parallel computing system for information retrieval," *Proceedings of the Fall Joint Computer Conference*, (1972), pp. 681-690.

[Parker 71] J. L. Parker, "A logic per track retrieval system," *Proceedings of the IFIPS Conference*, pp. TA-4-146 to TA-4-150, (1971).

[Patterson 79] D. A. Patterson, E. S. Fehr, and C. H. Sequin, "Design Considerations for the VLSI Processor of X-Tree," *Proceedings of the Sixth Annual Symposium Computer Architecture,* (April 1979), pp. 90-101.

[Peterson 57] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development,* Vol. 7, No. 2, (April 1963), pp. 127-129.

[Ries 77] D. R. Ries and M. Stonebraker, "Effects of locking granularity in a database management system," *ACM Transactions on Database Systems,* Vol. 2, No. 3, (September 1977), pp. 233-246.

[Rosenthal 77] R. S. Rosenthal, "The data management machine — a classification," *Proceedings of the Third Workshop on Computer Architecture of Non-Numeric Processing,* (1977).

[Schuster 76] S. A. Schuster, E. A. Ozkarahan, and K. C. Smith, "A virtual memory system for a relational associative processor," *Proceedings of the National Computer Conference,* Vol. 45, (1976), pp. 855-862.

[Schuster 77] *Private communication,* (November 1977).

[Schuster 78] "RAP.2 - An associative processor for data bases," *Proceedings of the Fifth Annual Symposium Computer Architecture,* (April 1978), pp. 52-59.

[Sibley 77] E. H. Sibley, "Standardization and database systems," *Proceedings of the Third International Conference on Very Large Data Bases,* (October 77), pp. 144-155.

[Siegel 79] H. J. Siegel, "Interconnection networks for SIMD machines," *Computer,* Vol. 12, No. 6, (June 1979), p. 59.

[Slotnick 70] D. L. Slotnick, "Logic per track devices", *Advances in Computers,* Vol. 10, Academic Press, New York, (1970), pp.291-296.

[Stellhorn 74] W. H. Stellhorn, "A specialized computer for information retrieval," Technical Report UIUCDCS-R-74-637, Department of Computer Science, The University of Illinois at Urbana-Champaign, Urbana, Illinois, (October 1974).

[Stone 71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computing,* Vol. C-20, No. 2, (February 1971), pp. 153-161.

[Stonebraker 75] M. Stonebraker, "Implementation of integrity constraints and views by query modification," *Proceedings of the 1975 ACM-SIGMOD Conference on Management of Data,* San Jose, California, (June 1975).

[Stonebraker 76] M. Stonebraker, E. Wong, and P. Kreps, "The design and implementation of INGRES," *ACM Trans. Database Syst.* Vol. 1, No. 3, (September 1976), pp. 189-222.

[Stonebraker 79] M. Stonebraker, "MUFFIN: A distributed data base machine," *Proceedings of the Irst International Conference on Distributed Computing Systems,* Huntsville, Alabama, (October 1979), pp. 459-469. Also available as Memorandum No. UCB/ERL M79/28, Electronics Research Laboratory, University of California, Berkeley, (May 1979).

[Stonebraker 80] "Retrospection on a database system," *ACM Trans. Database Syst.* Vol. 5, No. 2, (June 1980) pp. 225-240.

[Su 75] S. Y. W. Su, and G. J. Lipovski, "CASSM: A cellular system for large data bases," *Proceedings of the International Conference on Very Large Data Bases,* Framingham, Mass., (September 1975), pp. 456-472.

[VAX 77] VAX-11/780 Architecture Handbook, Digital Equipment Corp., Maynard Massachusetts, 1977.

[Watson 74] J. K. Watson, G. J. Lipovski, and S. Y. W. Su, "A multiple head disc system for fast context addressing for large data bases," *Workshop in Computer Architecture for Non-Numeric Processing,* Dallas, Texas, (1974).

[Yau 66] S. S. Yau, and C. C. Yang, "A cryogenic associative memory system for information retrieval", *Proceedings of the National Electronics Conference,* pp. 764-769, (October 1966).

APPENDIX A

Notation and Parameter Values

A.1. Parameters of Blasgen and Eswaran

Blasgen and Eswaran[Blasgen 77] introduced a large amount of notation which has been used extensively here with essentially no modification. The following are parameters which they used. In many cases, the same parameter is defined for relations R and S. In those cases, designated here with a subscript x, $x = 1$ refers to relation R and $x = 2$ refers to relation S.

A.1.1. Query-independent parameters

N_x Cardinality of relation.

E_x (Average) Number of tuples from R in a data page. It is obtained by dividing N_x by the number of data pages that contain at least one tuple from R.

M_x Total number of data pages in the segment which contains R. M_x is never smaller than $N_x \cdot E_x$ and may be larger.

L Average number of (KEY, TID) pairs per page.

C_x Number of tuples of R that fit in a page of a (temporary) file, as obtained by dividing the size of a page in the file by the average size of a tuple. C_x may be different than E_x because a data page in the data base may contain tuples from more than one relation.

P_x Effectiveness of the *join filter*, i.e., the fraction of tuples of the relation that participate in the unconditional equijoin.

G Ratio between the number of tuples in the unconditional equijoin and $N_1 \cdot N_2$.

I Number of *TID*s that fit in a page of a temporary file.

K Number of (KEY, TID) pairs that fit in a page of a temporary file.

P Main storage space in page frames that are available for sort buffers, *TID* lists, *TID* pair lists, W_2', etc.

Z Merge factor for sort-merge algorithms.

Q The block size (in pages) for temporary storage during a sorting operation. $Z + 1$ blocks must fit in main storage at once, so

$$Q = \left\lfloor \frac{P}{Z+1} \right\rfloor.$$

A Cost (in time) of a page transfer between a processor and disk.

B Cost (in time) of a block transfer between a processor and disk. A block is the unit of transfer for a file. In the current context, it has been assumed that it is never larger than one cylinder.

A.1.2. Query-dependent parameters

H_x Ratio between the average size of the subtuple of interest from the relation and the average size of a tuple in that relation.

F_x Effectiveness of the *predicate filter*, i.e., the ratio between the number of tuples that satisfy the predicate and the cardinality of the relation.

A.1.3. Values of Parameters of Blasgen and Eswaran

Most of the parameters defined above were assumed by Blasgen and Eswaran to be fixed for all the work reported. Those include the following:

$N_1 = 4 \cdot N_2$

$E_1 = E_2 = 20$

I - 1000

$C_1 = C_2 = 20$

$L = 200$

$K = 300$

$P_1 = P_2 = 1.0$

$H_1 = H_2 = 0.5$

$G = 7 / N_2$

$P = 25$

$A = 1$

$B = 1$

$Z = 3$

The parameters that were varied were the following:

$$200 < N_2 < 30000$$

$$F_1 = F_2 = (1.0,0.5,0.1,0.01)$$

A.2. Other Notation

Many other quantities were defined throughout this work. The following is a list of those which were referenced.

μ The number of links over which a hash table is sent during a merge or broadcast of hash tables.

ρ The number of cylinders which must be read to scan the entire relation R.

ρ_x The number of cylinders which must be read to scan an entire index of the relation R.

σ The number of cylinders which must be read to scan the entire relation S.

σ_x The number of cylinders which must be read to scan an entire index of the relation S.

$\tau(p,n,e)$
The expected number of pages which must be accessed in order to examine $p \cdot n$ records of a relation containing n records distributed evenly, e per page. It is equal to

$$\frac{n \cdot p}{e} \cdot \left\{ 1 - \frac{\binom{n - e}{p \cdot n}}{\binom{n}{p \cdot n}} \right\}.$$

b The number of bits in a byte, usually 8.

$C1$ The total number of comparisons done in all processors for the case where all elements are identical.

$C1_{max}$ The total number of comparisons done in the busiest processor for the case where all elements are identical.

CN The total number of comparisons done in all processors for the case where all elements are unique.

CN_{max} The total number of comparisons done in the busiest processor for the case where all elements are unique.

C_C The total cost, in pages, of all communications among processors for the join methods.

C_D The total cost, in pages, of all disk accesses for the join methods.

d The average number of links traversed by a message sent randomly from one leaf node to another in Hypertree. In [Goodman 79] it was shown that

$$d = \frac{5 \cdot m}{4} - \frac{4}{3} + \frac{2}{3} \cdot 2^{-m} - \frac{a}{12},$$

where m is the number of levels of the tree, not including the root, and $a = m \bmod 2$.

γ_x The cost in terms of disk accesses of accessing relation R one time.

$M1$ The total number of message element links required for the case where all elements are identical.

$\overline{M1}$ The total number of message element links required for the case where all elements are identical, normalized for number of ports per processor.

$M1_{max}$ The total number of message element links required for the case where all elements are identical.

$\overline{M1_{max}}$
The total number of message element links required for the case where all elements are identical, normalized for number of ports per processor.

MN The total number of message element links required for the case where all elements are unique.

$\overline{MN}$ The total number of message element links required for the case where all elements are unique, normalized for number of ports per processor.

MN_{max}
The total number of message element links required for the case where all elements are unique.

$\overline{MN_{max}}$
The total number of message element links required for the case where all elements are unique, normalized for number of ports per processor.

R The ratio of the number of phantom tuples incorrectly included in the preliminary result relation to the number of tuples in the correct result relation.

S The number of bytes of data in a page.

T_x The ratio of unique values existing in the join field to N_x

θ_x The total number of pages of relation W_x which must be transmitted in join methods 2 and 3.

Y The expected number of unique values to be entered into a hash table.

U The number of heads read in parallel from a disk surface into processors.

V The number of pages existing on each track of the disk.

APPENDIX B

The Busiest Links in Hypertree

In order to calculate the busiest link for a given algorithm, it is necessary to determine the amount of traffic in each link. This can be done without great difficulty if the assumption is made that the message traffic among the various leaf nodes is symmetric, i.e., the lengths of their lists are equal, the sizes and number of messages they send are equal, and the amount of data read from the disk into each node is the same. Under this assumption, the symmetric nature of Hypertree guarantees that every link connecting level i to level j will have the same amount of traffic as every other link connecting level i to level j. This includes the horizontal links, i.e., when $i = j$. The problem then becomes one of determining the amount of traffic for each of these classes of links for each communication occurring in the handling of a query and dividing by the number of links in the class.

All messages in the algorithms considered were of four types.

(1) A set of hash tables was propagated up the tree, being merged at each level, so that every link which was connecting two ancestors of a node receiving data directly from a disk would transmit exactly one copy of the hash table

(2) Leaf node to parent traffic. In algorithm P3, it was necessary to send some results from a leaf node to its parent, because of memory size limits.

(3) A general exchange of data. Each processor sent messages to various other processors. The assumption was made that the messages were randomly distributed, with average path length the same as that of a tree large enough to have U leaves.

(4) The I/O links between the leaf processors and the disk transmitted all the data read in from the disk (and written to it).

For the case of a hash table being propagated from the leaves to the root, or *vice versa*, all the vertical links, for which $i \neq j$, carry the entire hash table exactly once, and all the horizontal links, for which $i = j$ carry no traffic at all. For the case where leaf nodes found it necessary to send a portion of their work to a parent, the lowest level vertical links each carried the amount of traffic sent, an no other links carried any traffic. For data read from the disk to the leaf nodes, no traffic occurred on any interprocessor links, and the traffic on the I/O links was assumed to be equally distributed. This leaves the calculation for the case where a general exchange took place among the leaf nodes.

In [Goodman 79] the average path length of the Hypertree structure was derived for uniform distribution of messages among the leaf nodes. That number was shown to be

$$d = \frac{5m}{4} - \frac{4}{3} + \frac{4}{3} \cdot 2^{-m} - \frac{a}{12}$$

where d is the average path length between pairs of leaf nodes communicating, m is the level number of the node, the root being 0, and $a = m \bmod 2$, The derivation was based on the observation that the message density through a link was uniform at a given level in the tree and was distributed in a particular way. The busiest links in a general exchange among leaf nodes are those nodes near the middle levels of the tree. This can be shown by considering separately the vertical links (the binary tree) and the horizontal links adding the redundancy for two cases:

(1) the number of leaves in the complete tree is a power of 4, i.e., the level number of the leaf nodes, starting with the root level as 0, is even, and

(2) the number of leaves in the complete tree is twice a power of 4, i.e., the level number of the leaf nodes is odd.

In both cases, it has been shown [Goodman 79] that for the bottom half of the tree, the horizontal links at each level collectively carry precisely half of all messages. For the top half of the tree, the horizontal links carry no messages at all. For the first case, where for a k-level tree, i.e., $P = 2^k$ leaf nodes, k is even, the links at the highest level carrying messages are at level $\frac{k}{2} + 1$. Where k is odd, the links at the highest level carrying messages are at level $\frac{k+1}{2}$. Since there are $\sqrt{P}$ and $\sqrt{P}/2$ links respectively at the highest levels carrying traffic, and twice as many at each succeedingly lower level, the traffic per horizontal link at each level can easily be determined.

Table B.1 shows the number of messages links occurring for the vertical links between adjacent levels for k even. Table B.2 shows the same data for k odd.

Level Number	Number of Links	Messages Traversing Levels	Communications Per Link
1	2	0	0
2	4	0	0
. . .	. . .	. . .	. . .
$\frac{k}{2}$	$2^{\frac{k}{2}}$	0	0
$\frac{k}{2}+1$	$2^{\frac{k}{2}+1}$	$\frac{2^k}{2}$	$\frac{\sqrt{P}}{2}$
$\frac{k}{2}+2$	$2^{\frac{k}{2}+2}$	$\frac{7}{8}2^k$	$\frac{7}{16}\sqrt{P}$
$\frac{k}{2}+3$	$2^{\frac{k}{2}+3}$	$\frac{31}{32}2^k$	$\frac{31}{128}\sqrt{P}$
$\frac{k}{2}+4$	$2^{\frac{k}{2}+4}$	$\frac{127}{128}2^k$	$\frac{127}{1024}\sqrt{P}$
. . .	. . .	. . .	. . .
$k-2$	2^{k-2}	2^k-32	$8-\frac{256}{P}$
$k-1$	2^{k-1}	2^k-8	$4-\frac{32}{P}$
k	2^k	2^k-2	$2-\frac{4}{P}$

Table B.1. Density of communication traffic for vertical links of Hypertree for an odd number of levels. Level i refers to links connecting the nodes at level i to the level above. The number of messages traversing the links is the number of messages sent from one leaf processor which pass through a link at that level. The communications per link is determined by dividing the number of messages traversing the links by the number of links at that level and multiplying by two, since each message passing through the level must pass both up and down. To determine total link traffic, the communications per link entry must be multiplied by the number of leaf nodes.

Level Number	Number of Links	Messages Traversing Levels	Communications Per Link
1	2	0	0
2	4	0	0
. . .	. . .	. . .	. . .
$\frac{k-1}{2}$	$2^{\frac{k-1}{2}}$	0	0
$\frac{k+1}{2}$	$2^{\frac{k+1}{2}}$	0	0
$\frac{k+3}{2}$	$2^{\frac{k+3}{2}}$	$\frac{3}{4}2^k$	$\frac{3}{4}\sqrt{2 \cdot P}$
$\frac{k+5}{2}$	$2^{\frac{k+5}{2}}$	$\frac{15}{16}2^k$	$\frac{15}{32}\sqrt{2 \cdot P}$
$\frac{k+7}{2}$	$2^{\frac{k+7}{2}}$	$\frac{63}{64}2^k$	$\frac{63}{256}\sqrt{2 \cdot P}$
. . .	. . .	. . .	. . .
$k-2$	2^{k-2}	$2^k - 32$	$8 - \frac{256}{P}$
$k-1$	2^{k-1}	$2^k - 8$	$4 - \frac{32}{P}$
k	2^k	$2^k - 2$	$2 - \frac{4}{P}$

Table B.2. Density of communication traffic for vertical links of Hypertree for an even number of levels. Level i refers to links connecting the nodes at level i to the level above. The number of messages traversing the links is the number of messages sent from one leaf processor which pass through a link at that level. The communications per link is determined by dividing the number of messages traversing the links by the number of links at that level and multiplying by two, since each message passing through the level must pass both up and down. To determine total link traffic, the communications per link entry must be multiplied by the number of leaf nodes.

APPENDIX C

Results of Join Analysis

The cost functions for the join algorithms described in chapter 5 were calculated under a variety of conditions. Initially the calculations were done using the values suggested by Blasgen and Eswaran for the parameters they defined. Each of four cost functions were determined, all measured in the number of pages transferred:

(1) Total interprocessor communication. The total number of pages passing through all links except those between processors and the disk.

(2) Total disk communication. The total number of pages read from the disk. This was the cost function defined by Blasgen and Eswaran.

(3) Worst case interprocessor communication. The total number of pages crossing the busiest link between processors.

(4) Worst case disk communication. The total number of pages crossing the busiest link between a processor and the disk. Since disk traffic was assumed to be evenly distributed, this was just the total disk communication divided by the number of heads being read.

Each of these cost functions was calculated for four possible values of the F_1 and F_2, the effectiveness of the predicate filter: 1.0, 0.5, 0.1, and 0.01. In addition, five different situations were analyzed:

A Relations R and S have join column indices, and indices on irrelevant columns X and Y respectively. They are not clustered on the join indices, but rather on columns X and Y.

B Relations R and S have join indices, restriction column indices, and indices on irrelevant columns X and Y respectively. They are not clustered on the join indices, nor on the restriction column indices, but rather on columns X and Y.

C Relations R and S have join indices and indices on the restriction columns. They are clustered on the join column indices.

D Relations R and S have join indices and indices on the restriction columns. They are clustered on the restriction column indices.

E Relations R and S do not have join column indices, but have indices on the restriction columns. They are not clustered on the restriction column indices.

E Relations R and S do not have join column indices and indices on the restriction columns. They are not clustered on either the restriction column or on the join column.

The same set of calculations were run again after changing two variables:

(1) The cost B for accessing a block on the disk was increased from 1 to 6.

(2) The number of pages available for temporary storage was increased from 25 to 4096.

Tables C.1.1.1 through C.2.4.5 present the results of these computations. Selected results were also presented in Chapter 5 in graphical form.

TOTAL INTERPROCESSOR COST							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	207	183	183	--	321	321	--
1000	691	609	609	--	790	790	--
3000	2072	1828	5484	--	2131	2131	--
10000	6906	6094	60938	--	6823	6823	--
30000	20719	18281	548438	--	20229	20229	--
100000	69063	60938	6093750	--	67151	67151	--
300000	207188	182813	54843750	--	201214	201214	--
1000000	690625	609375	609375000	--	670673	670673	--
$F_1 = F_2 = 0.500$							
300	104	91	91	--	170	170	--
1000	347	305	305	--	288	288	--
3000	1042	914	1828	--	623	623	--
10000	3473	3047	15234	--	1796	1796	--
30000	10420	9141	137109	--	5147	5147	--
100000	34734	30469	1523438	--	16878	16878	--
300000	104203	91406	13710938	--	50393	50393	--
1000000	347344	304688	152343750	--	167938	167938	--
$F_1 = F_2 = 0.100$							
300	35	18	18	--	122	122	--
1000	117	61	61	--	127	127	--
3000	350	183	183	--	140	140	--
10000	1166	609	609	--	187	187	--
30000	3498	1828	5484	--	321	321	--
100000	11659	6094	60938	--	790	790	--
300000	34978	18281	548438	--	2131	2131	--
1000000	116594	60938	6093750	--	7063	7063	--
$F_1 = F_2 = 0.010$							
300	21	2	2	--	120	120	--
1000	70	6	6	--	120	120	--
3000	210	18	18	--	120	120	--
10000	700	61	61	--	121	121	--
30000	2101	183	183	--	122	122	--
100000	7005	609	609	--	127	127	--
300000	21015	1828	5484	--	140	140	--
1000000	70050	6094	60938	--	427	427	--

Table C.1.1.1. Total processor communication cost C_C. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL INTERPROCESSOR COST							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	207	183	183	197	321	321	445
1000	691	609	609	658	790	790	930
3000	2072	1828	5484	1974	2131	2131	2321
10000	6906	6094	60938	6581	6823	6823	7396
30000	20719	18281	548438	19744	20229	20229	22051
100000	69063	60938	6093750	65813	67151	67151	73516
300000	207188	182813	54843750	197438	201214	201214	221978
1000000	690625	609375	609375000	658125	670673	670673	745760
$F_1 = F_2 = 0.500$							
300	104	91	91	51	170	170	292
1000	347	305	305	171	288	288	418
3000	1042	914	1828	512	623	623	778
10000	3473	3047	15234	1706	1796	1796	2142
30000	10420	9141	137109	5119	5147	5147	6038
100000	34734	30469	1523438	17063	16878	16878	20120
300000	104203	91406	13710938	51188	50393	50393	60795
1000000	347344	304688	152343750	170625	167938	167938	205462
$F_1 = F_2 = 0.100$							
300	35	18	18	3	122	122	242
1000	117	61	61	9	127	127	249
3000	350	183	183	26	140	140	267
10000	1166	609	609	88	187	187	336
30000	3498	1828	5484	263	321	321	547
100000	11659	6094	60938	878	790	790	1455
300000	34978	18281	548438	2633	2131	2131	4227
1000000	116594	60938	6093750	8775	7063	7063	14568
$F_1 = F_2 = 0.010$							
300	21	2	2	0	120	120	240
1000	70	6	6	0	120	120	240
3000	210	18	18	1	120	120	241
10000	700	61	61	3	121	121	244
30000	2101	183	183	9	122	122	253
100000	7005	609	609	31	127	127	285
300000	21015	1828	5484	92	140	140	394
1000000	70050	6094	60938	307	427	427	1134

Table C.1.1.2. Total processor communication cost C_C. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL INTERPROCESSOR COST							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	41	183	183	82	160	160	284
1000	138	609	609	273	254	254	394
3000	414	1828	5484	819	522	522	712
10000	1381	6094	60938	2730	1461	1461	2034
30000	4144	18281	548438	8190	4142	4142	5964
100000	13813	60938	6093750	27300	13526	13526	19891
300000	41438	182813	54843750	81900	40339	40339	61103
1000000	138125	609375	609375000	273000	134423	134423	209510
$F_1 = F_2 = 0.500$							
300	12	91	91	22	130	130	252
1000	39	305	305	74	154	154	284
3000	116	914	1828	223	221	221	375
10000	386	3047	15234	743	455	455	802
30000	1158	9141	137109	2230	1125	1125	2016
100000	3859	30469	1523438	7434	3472	3472	6714
300000	11578	91406	13710938	22303	10175	10175	20577
1000000	38594	304688	152343750	74344	33876	33876	71399
$F_1 = F_2 = 0.100$							
300	1	18	18	1	120	120	241
1000	3	61	61	5	121	121	243
3000	9	183	183	15	124	124	251
10000	28	609	609	49	133	133	283
30000	85	1828	5484	148	160	160	386
100000	284	6094	60938	492	254	254	919
300000	853	18281	548438	1477	522	522	2619
1000000	2844	60938	6093750	4924	1701	1701	9205
$F_1 = F_2 = 0.010$							
300	0	2	2	0	120	120	240
1000	0	6	6	0	120	120	240
3000	1	18	18	1	120	120	241
10000	2	61	61	3	120	120	243
30000	5	183	183	8	120	120	251
100000	17	609	609	27	121	121	280
300000	52	1828	5484	81	124	124	378
1000000	175	6094	60938	269	373	373	1080

Table C.1.1.3. Total processor communication cost C_C. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL INTERPROCESSOR COST							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	207	183	183	197	321	321	445
1000	691	609	609	658	790	790	930
3000	2072	1828	5484	1974	2131	2131	2321
10000	6906	6094	60938	6581	6823	6823	7396
30000	20719	18281	548438	19744	20229	20229	22051
100000	69063	60938	6093750	65813	67151	67151	73516
300000	207188	182813	54843750	197438	201214	201214	221978
1000000	690625	609375	609375000	658125	670673	670673	745760
$F_1 = F_2 = 0.500$							
300	104	91	91	51	170	170	292
1000	347	305	305	171	288	288	418
3000	1042	914	1828	512	623	623	778
10000	3473	3047	15234	1706	1796	1796	2142
30000	10420	9141	137109	5119	5147	5147	6038
100000	34734	30469	1523438	17063	16878	16878	20120
300000	104203	91406	13710938	51188	50393	50393	60795
1000000	347344	304688	152343750	170625	167938	167938	205462
$F_1 = F_2 = 0.100$							
300	35	18	18	3	122	122	242
1000	117	61	61	9	127	127	249
3000	350	183	183	26	140	140	267
10000	1166	609	609	88	187	187	336
30000	3498	1828	5484	263	321	321	547
100000	11659	6094	60938	878	790	790	1455
300000	34978	18281	548438	2633	2131	2131	4227
1000000	116594	60938	6093750	8775	7063	7063	14568
$F_1 = F_2 = 0.010$							
300	21	2	2	0	120	120	240
1000	70	6	6	0	120	120	240
3000	210	18	18	1	120	120	241
10000	700	61	61	3	121	121	244
30000	2101	183	183	9	122	122	253
100000	7005	609	609	31	127	127	285
300000	21015	1828	5484	92	140	140	394
1000000	70050	6094	60938	307	427	427	1134

Table C.1.1.4. Total processor communication cost C_C. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL INTERPROCESSOR COST							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	183	183	--	321	--	--
1000	--	609	609	--	790	--	--
3000	--	1828	5484	--	2131	--	--
10000	--	6094	60938	--	6823	--	--
30000	--	18281	548438	--	20229	--	--
100000	--	60938	6093750	--	67151	--	--
300000	--	182813	54843750	--	201214	--	--
1000000	--	609375	609375000	--	670673	--	--
$F_1 = F_2 = 0.500$							
300	--	91	91	--	170	--	--
1000	--	305	305	--	288	--	--
3000	--	914	1828	--	623	--	--
10000	--	3047	15234	--	1796	--	--
30000	--	9141	137109	--	5147	--	--
100000	--	30469	1523438	--	16878	--	--
300000	--	91406	13710938	--	50393	--	--
1000000	--	304688	152343750	--	167938	--	--
$F_1 = F_2 = 0.100$							
300	--	18	18	--	122	--	--
1000	--	61	61	--	127	--	--
3000	--	183	183	--	140	--	--
10000	--	609	609	--	187	--	--
30000	--	1828	5484	--	321	--	--
100000	--	6094	60938	--	790	--	--
300000	--	18281	548438	--	2131	--	--
1000000	--	60938	6093750	--	7063	--	--
$F_1 = F_2 = 0.010$							
300	--	2	2	--	120	--	--
1000	--	6	6	--	120	--	--
3000	--	18	18	--	120	--	--
10000	--	61	61	--	121	--	--
30000	--	183	183	--	122	--	--
100000	--	609	609	--	127	--	--
300000	--	1828	5484	--	140	--	--
1000000	--	6094	60938	--	427	--	--

Table C.1.1.5. Total processor communication cost C_C. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL INTERPROCESSOR COST							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	183	183	--	321	--	--
1000	--	609	609	--	790	--	--
3000	--	1828	5484	--	2131	--	--
10000	--	6094	60938	--	6823	--	--
30000	--	18281	548438	--	20229	--	--
100000	--	60938	6093750	--	67151	--	--
300000	--	182813	54843750	--	201214	--	--
1000000	--	609375	609375000	--	670673	--	--
$F_1 = F_2 = 0.500$							
300	--	91	91	--	170	--	--
1000	--	305	305	--	288	--	--
3000	--	914	1828	--	623	--	--
10000	--	3047	15234	--	1796	--	--
30000	--	9141	137109	--	5147	--	--
100000	--	30469	1523438	--	16878	--	--
300000	--	91406	13710938	--	50393	--	--
1000000	--	304688	152343750	--	167938	--	--
$F_1 = F_2 = 0.100$							
300	--	18	18	--	122	--	--
1000	--	61	61	--	127	--	--
3000	--	183	183	--	140	--	--
10000	--	609	609	--	187	--	--
30000	--	1828	5484	--	321	--	--
100000	--	6094	60938	--	790	--	--
300000	--	18281	548438	--	2131	--	--
1000000	--	60938	6093750	--	7063	--	--
$F_1 = F_2 = 0.010$							
300	--	2	2	--	120	--	--
1000	--	6	6	--	120	--	--
3000	--	18	18	--	120	--	--
10000	--	61	61	--	121	--	--
30000	--	183	183	--	122	--	--
100000	--	609	609	--	127	--	--
300000	--	1828	5484	--	140	--	--
1000000	--	6094	60938	--	427	--	--

Table C.1.1.6. Total processor communication cost C_C. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL DISK COST							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	1508	104	80	--	4	4	--
1000	5025	442	320	--	8	6	--
3000	15075	1758	3840	--	24	14	--
10000	50250	4566	25600	--	46	26	--
30000	150750	17490	307200	--	174	94	--
100000	502500	69302	4096000	--	686	365	--
300000	1507500	179420	24576000	--	1368	747	--
1000000	5025000	694358	327680000	--	5464	2941	--
$F_1 = F_2 = 0.500$							
300	1358	86	80	--	4	4	--
1000	4525	362	320	--	8	6	--
3000	13575	1458	2560	--	24	14	--
10000	45250	3526	12800	--	46	26	--
30000	135750	13744	153600	--	174	94	--
100000	452500	54726	2048000	--	686	365	--
300000	1357500	129420	12288000	--	1368	747	--
1000000	4525000	506868	163840000	--	5464	2941	--
$F_1 = F_2 = 0.100$							
300	1238	80	80	--	4	4	--
1000	4125	324	320	--	8	6	--
3000	12375	1304	1280	--	24	14	--
10000	41250	2682	2560	--	46	26	--
30000	123750	10718	30720	--	174	94	--
100000	412500	42966	409600	--	686	365	--
300000	1237500	89170	2457600	--	1368	747	--
1000000	4125000	356022	32768000	--	5464	2941	--
$F_1 = F_2 = 0.010$							
300	1211	80	80	--	4	4	--
1000	4035	320	320	--	8	6	--
3000	12105	1280	1280	--	24	14	--
10000	40350	2564	2560	--	46	26	--
30000	121050	10264	10240	--	174	94	--
100000	403500	41082	40960	--	686	365	--
300000	1210500	82398	245760	--	1368	747	--
1000000	4035000	329686	3276800	--	5464	2941	--

Table C.1.2.1. Total disk communication cost C_D. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL DISK COST							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	1508	104	80	1516	4	4	6
1000	5025	442	320	5050	8	6	8
3000	15075	1758	3840	15150	24	14	16
10000	50250	4566	25600	243032	46	26	29
30000	150750	17490	307200	1394140	174	94	101
100000	502500	69302	4096000	5668138	686	365	387
300000	1507500	179420	24576000	17879900	1368	747	810
1000000	5025000	694358	327680000	60622174	5464	2941	3150
$F_1 = F_2 = 0.500$							
300	1358	86	80	391	4	4	6
1000	4525	362	320	1300	8	6	8
3000	13575	1458	2560	3900	24	14	16
10000	45250	3526	12800	118000	46	26	29
30000	135750	13744	153600	774066	174	94	101
100000	452500	54726	2048000	3805308	686	365	387
300000	1357500	129420	12288000	12466150	1368	747	810
1000000	4525000	506868	163840000	42779848	5464	2941	3150
$F_1 = F_2 = 0.100$							
300	1238	80	80	31	4	4	6
1000	4125	324	320	100	8	6	8
3000	12375	1304	1280	300	24	14	16
10000	41250	2682	2560	1000	46	26	29
30000	123750	10718	30720	3000	174	94	101
100000	412500	42966	409600	1935032	686	365	387
300000	1237500	89170	2457600	8045140	1368	747	810
1000000	4125000	356022	32768000	29675638	5464	2941	3150
$F_1 = F_2 = 0.010$							
300	1211	17	17	18	4	4	6
1000	4035	52	52	50	8	6	8
3000	12105	152	152	145	24	14	16
10000	40350	507	503	473	46	26	29
30000	121050	1532	1508	1416	174	94	101
100000	403500	5147	5025	4717	686	365	387
300000	1210500	15553	45225	14147	1368	747	810
1000000	4035000	52256	502500	19297184	5464	2941	3150

Table C.1.2.2. Total disk communication cost C_D. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL DISK COST							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	83	104	80	91	4	4	6
1000	275	442	320	300	8	6	8
3000	825	1758	3840	900	24	14	16
10000	2750	4566	25600	3041	46	26	29
30000	8250	17490	307200	9165	174	94	101
100000	27500	69302	4096000	30722	686	365	387
300000	82500	179420	24576000	92650	1368	747	810
1000000	275000	694358	327680000	310508	5464	2941	3150
$F_1 = F_2 = 0.500$							
300	83	86	80	91	4	4	6
1000	275	362	320	300	8	6	8
3000	825	1458	2560	900	24	14	16
10000	2750	3526	12800	3000	46	26	29
30000	8250	13744	153600	9079	174	94	101
100000	27500	54726	2048000	30351	686	365	387
300000	82500	129420	12288000	91275	1368	747	810
1000000	275000	506868	163840000	305266	5464	2941	3150
$F_1 = F_2 = 0.100$							
300	82	80	80	31	4	4	6
1000	270	324	320	97	8	6	8
3000	807	1304	1280	288	24	14	16
10000	2690	2682	2560	957	46	26	29
30000	8068	10718	30720	2867	174	94	101
100000	26893	42966	409600	9594	686	365	387
300000	80677	89170	2457600	28823	1368	747	810
1000000	268922	356022	32768000	96246	5464	2941	3150
$F_1 = F_2 = 0.010$							
300	71	17	17	18	4	4	6
1000	235	52	52	50	8	6	8
3000	703	152	152	144	24	14	16
10000	2342	507	503	473	46	26	29
30000	7024	1532	1508	1416	174	94	101
100000	23411	5147	5025	4717	686	365	387
300000	70232	15553	45225	14147	1368	747	810
1000000	234105	52256	502500	47193	5464	2941	3150

Table C.1.2.3. Total disk communication cost C_D. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL DISK COST							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	1508	107	83	1516	4	4	6
1000	5025	397	275	5050	8	6	8
3000	15075	1303	2475	15150	24	14	16
10000	50250	4756	27500	243032	46	26	29
30000	150750	15500	247500	1394140	174	94	101
100000	502500	55842	2750000	5668138	686	365	387
300000	1507500	180000	24750000	17879900	1368	747	810
1000000	5025000	641678	275000000	60622174	5464	2941	3150
$F_1 = F_2 = 0.500$							
300	1358	48	42	391	4	4	6
1000	4525	180	138	1300	8	6	8
3000	13575	591	826	3900	24	14	16
10000	45250	2341	6875	118000	46	26	29
30000	135750	7629	61875	774066	174	94	101
100000	452500	27516	687500	3805308	686	365	387
300000	1357500	88750	6187500	12466150	1368	747	810
1000000	4525000	316688	68750000	42779848	5464	2941	3150
$F_1 = F_2 = 0.100$							
300	1238	10	10	31	4	4	6
1000	4125	32	28	100	8	6	8
3000	12375	107	83	300	24	14	16
10000	41250	397	275	1000	46	26	29
30000	123750	1303	2475	3000	174	94	101
100000	412500	4756	27500	1935032	686	365	387
300000	1237500	15500	247500	8045140	1368	747	810
1000000	4125000	55842	2750000	29675638	5464	2941	3150
$F_1 = F_2 = 0.010$							
300	1211	4	4	18	4	4	6
1000	4035	5	5	50	8	6	8
3000	12105	10	10	145	24	14	16
10000	40350	32	28	473	46	26	29
30000	121050	107	83	1416	174	94	101
100000	403500	397	275	4717	686	365	387
300000	1210500	1303	2475	14147	1368	747	810
1000000	4035000	4756	27500	19297184	5464	2941	3150

Table C.1.2.4. Total disk communication cost C_D. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL DISK COST							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	104	80	--	4	--	--
1000	--	442	320	--	8	--	--
3000	--	1758	3840	--	24	--	--
10000	--	4566	25600	--	46	--	--
30000	--	17490	307200	--	174	--	--
100000	--	69302	4096000	--	686	--	--
300000	--	179420	24576000	--	1368	--	--
1000000	--	694358	327680000	--	5464	--	--
$F_1 = F_2 = 0.500$							
300	--	86	80	--	4	--	--
1000	--	362	320	--	8	--	--
3000	--	1458	2560	--	24	--	--
10000	--	3526	12800	--	46	--	--
30000	--	13744	153600	--	174	--	--
100000	--	54726	2048000	--	686	--	--
300000	--	129420	12288000	--	1368	--	--
1000000	--	506868	163840000	--	5464	--	--
$F_1 = F_2 = 0.100$							
300	--	80	80	--	4	--	--
1000	--	324	320	--	8	--	--
3000	--	1304	1280	--	24	--	--
10000	--	2682	2560	--	46	--	--
30000	--	10718	30720	--	174	--	--
100000	--	42966	409600	--	686	--	--
300000	--	89170	2457600	--	1368	--	--
1000000	--	356022	32768000	--	5464	--	--
$F_1 = F_2 = 0.010$							
300	--	17	17	--	4	--	--
1000	--	52	52	--	8	--	--
3000	--	152	152	--	24	--	--
10000	--	507	503	--	46	--	--
30000	--	1532	1508	--	174	--	--
100000	--	5147	5025	--	686	--	--
300000	--	15553	45225	--	1368	--	--
1000000	--	52256	502500	--	5464	--	--

Table C.1.2.5. Total disk communication cost C_D. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL DISK COST							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	104	80	--	4	--	--
1000	--	442	320	--	8	--	--
3000	--	1758	3840	--	24	--	--
10000	--	4566	25600	--	46	--	--
30000	--	17490	307200	--	174	--	--
100000	--	69302	4096000	--	686	--	--
300000	--	179420	24576000	--	1368	--	--
1000000	--	694358	327680000	--	5464	--	--
$F_1 = F_2 = 0.500$							
300	--	86	80	--	4	--	--
1000	--	362	320	--	8	--	--
3000	--	1458	2560	--	24	--	--
10000	--	3526	12800	--	46	--	--
30000	--	13744	153600	--	174	--	--
100000	--	54726	2048000	--	686	--	--
300000	--	129420	12288000	--	1368	--	--
1000000	--	506868	163840000	--	5464	--	--
$F_1 = F_2 = 0.100$							
300	--	80	80	--	4	--	--
1000	--	324	320	--	8	--	--
3000	--	1304	1280	--	24	--	--
10000	--	2682	2560	--	46	--	--
30000	--	10718	30720	--	174	--	--
100000	--	42966	409600	--	686	--	--
300000	--	89170	2457600	--	1368	--	--
1000000	--	356022	32768000	--	5464	--	--
$F_1 = F_2 = 0.010$							
300	--	80	80	--	4	--	--
1000	--	320	320	--	8	--	--
3000	--	1280	1280	--	24	--	--
10000	--	2564	2560	--	46	--	--
30000	--	10264	10240	--	174	--	--
100000	--	41082	40960	--	686	--	--
300000	--	82398	245760	--	1368	--	--
1000000	--	329686	3276800	--	5464	--	--

Table C.1.2.6. Total disk communication cost C_D. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	128	113	113	--	127	127	--
1000	425	375	375	--	416	416	--
3000	1275	1125	3375	--	1241	1241	--
10000	4250	3750	37500	--	4128	4128	--
30000	12750	11250	337500	--	12378	12378	--
100000	42500	37500	3750000	--	41253	41253	--
300000	127500	112500	33750000	--	123753	123753	--
1000000	425000	375000	375000000	--	412509	412509	--
$F_1 = F_2 = 0.500$							
300	64	56	56	--	34	34	--
1000	214	188	188	--	106	106	--
3000	641	563	1125	--	312	312	--
10000	2138	1875	9375	--	1034	1034	--
30000	6413	5625	84375	--	3097	3097	--
100000	21375	18750	937500	--	10316	10316	--
300000	64125	56250	8437500	--	30941	30941	--
1000000	213750	187500	93750000	--	103134	103134	--
$F_1 = F_2 = 0.100$							
300	22	11	11	--	4	4	--
1000	72	38	38	--	7	7	--
3000	215	113	113	--	15	15	--
10000	718	375	375	--	44	44	--
30000	2153	1125	3375	--	127	127	--
100000	7175	3750	37500	--	416	416	--
300000	21525	11250	337500	--	1241	1241	--
1000000	71750	37500	3750000	--	4134	4134	--
$F_1 = F_2 = 0.010$							
300	13	1	1	--	3	3	--
1000	43	4	4	--	3	3	--
3000	129	11	11	--	3	3	--
10000	431	38	38	--	3	3	--
30000	1293	113	113	--	4	4	--
100000	4311	375	375	--	7	7	--
300000	12932	1125	3375	--	15	15	--
1000000	43108	3750	37500	--	50	50	--

Table C.1.3.1. Number of pages transferred on busiest interprocesor link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	128	113	113	122	127	127	131
1000	425	375	375	405	416	416	420
3000	1275	1125	3375	1215	1241	1241	1245
10000	4250	3750	37500	4050	4128	4128	4136
30000	12750	11250	337500	12150	12378	12378	12398
100000	42500	37500	3750000	40500	41253	41253	41317
300000	127500	112500	33750000	121500	123753	123753	123939
1000000	425000	375000	375000000	405000	412509	412509	413123
$F_1 = F_2 = 0.500$							
300	64	56	56	32	34	34	38
1000	214	188	188	105	106	106	110
3000	641	563	1125	315	312	312	316
10000	2138	1875	9375	1050	1034	1034	1040
30000	6413	5625	84375	3150	3097	3097	3107
100000	21375	18750	937500	10500	10316	10316	10350
300000	64125	56250	8437500	31500	30941	30941	31035
1000000	213750	187500	93750000	105000	103134	103134	103442
$F_1 = F_2 = 0.100$							
300	22	11	11	2	4	4	8
1000	72	38	38	5	7	7	11
3000	215	113	113	16	15	15	19
10000	718	375	375	54	44	44	48
30000	2153	1125	3375	162	127	127	131
100000	7175	3750	37500	540	416	416	424
300000	21525	11250	337500	1620	1241	1241	1261
1000000	71750	37500	3750000	5400	4134	4134	4198
$F_1 = F_2 = 0.010$							
300	13	1	1	0	3	3	7
1000	43	4	4	0	3	3	7
3000	129	11	11	1	3	3	7
10000	431	38	38	2	3	3	7
30000	1293	113	113	6	4	4	8
100000	4311	375	375	19	7	7	11
300000	12932	1125	3375	57	15	15	19
1000000	43108	3750	37500	189	50	50	58

Table C.1.3.2. Number of pages transferred on busiest interprocesor link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	26	113	113	50	28	28	32
1000	85	375	375	168	86	86	90
3000	255	1125	3375	504	251	251	255
10000	850	3750	37500	1680	828	828	836
30000	2550	11250	337500	5040	2478	2478	2498
100000	8500	37500	3750000	16800	8253	8253	8317
300000	25500	112500	33750000	50400	24753	24753	24939
1000000	85000	375000	375000000	168000	82509	82509	83123
$F_1 = F_2 = 0.500$							
300	7	56	56	14	9	9	13
1000	24	188	188	46	24	24	28
3000	71	563	1125	137	65	65	69
10000	238	1875	9375	458	209	209	215
30000	713	5625	84375	1373	622	622	632
100000	2375	18750	937500	4575	2066	2066	2100
300000	7125	56250	8437500	13725	6191	6191	6285
1000000	23750	187500	93750000	45750	20634	20634	20942
$F_1 = F_2 = 0.100$							
300	1	11	11	1	3	3	7
1000	2	38	38	3	4	4	8
3000	5	113	113	9	5	5	9
10000	18	375	375	30	11	11	15
30000	53	1125	3375	91	28	28	32
100000	175	3750	37500	303	86	86	94
300000	525	11250	337500	909	251	251	271
1000000	1750	37500	3750000	3030	834	834	898
$F_1 = F_2 = 0.010$							
300	0	1	1	0	3	3	7
1000	0	4	4	0	3	3	7
3000	0	11	11	0	3	3	7
10000	1	38	38	2	3	3	7
30000	3	113	113	5	3	3	7
100000	11	375	375	17	4	4	8
300000	32	1125	3375	50	5	5	9
1000000	108	3750	37500	165	17	17	30

Table C.1.3.3. Number of pages transferred on busiest interprocesor link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	128	113	113	122	127	127	131
1000	425	375	375	405	416	416	420
3000	1275	1125	3375	1215	1241	1241	1245
10000	4250	3750	37500	4050	4128	4128	4136
30000	12750	11250	337500	12150	12378	12378	12398
100000	42500	37500	3750000	40500	41253	41253	41317
300000	127500	112500	33750000	121500	123753	123753	123939
1000000	425000	375000	375000000	405000	412509	412509	413123
$F_1 = F_2 = 0.500$							
300	64	56	56	32	34	34	38
1000	214	188	188	105	106	106	110
3000	641	563	1125	315	312	312	316
10000	2138	1875	9375	1050	1034	1034	1040
30000	6413	5625	84375	3150	3097	3097	3107
100000	21375	18750	937500	10500	10316	10316	10350
300000	64125	56250	8437500	31500	30941	30941	31035
1000000	213750	187500	93750000	105000	103134	103134	103442
$F_1 = F_2 = 0.100$							
300	22	11	11	2	4	4	8
1000	72	38	38	5	7	7	11
3000	215	113	113	16	15	15	19
10000	718	375	375	54	44	44	48
30000	2153	1125	3375	162	127	127	131
100000	7175	3750	37500	540	416	416	424
300000	21525	11250	337500	1620	1241	1241	1261
1000000	71750	37500	3750000	5400	4134	4134	4198
$F_1 = F_2 = 0.010$							
300	13	1	1	0	3	3	7
1000	43	4	4	0	3	3	7
3000	129	11	11	1	3	3	7
10000	431	38	38	2	3	3	7
30000	1293	113	113	6	4	4	8
100000	4311	375	375	19	7	7	11
300000	12932	1125	3375	57	15	15	19
1000000	43108	3750	37500	189	50	50	58

Table C.1.3.4. Number of pages transferred on busiest interprocesor link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	113	113	--	127	--	--
1000	--	375	375	--	416	--	--
3000	--	1125	3375	--	1241	--	--
10000	--	3750	37500	--	4128	--	--
30000	--	11250	337500	--	12378	--	--
100000	--	37500	3750000	--	41253	--	--
300000	--	112500	33750000	--	123753	--	--
1000000	--	375000	375000000	--	412509	--	--
$F_1 = F_2 = 0.500$							
300	--	56	56	--	34	--	--
1000	--	188	188	--	106	--	--
3000	--	563	1125	--	312	--	--
10000	--	1875	9375	--	1034	--	--
30000	--	5625	84375	--	3097	--	--
100000	--	18750	937500	--	10316	--	--
300000	--	56250	8437500	--	30941	--	--
1000000	--	187500	93750000	--	103134	--	--
$F_1 = F_2 = 0.100$							
300	--	11	11	--	4	--	--
1000	--	38	38	--	7	--	--
3000	--	113	113	--	15	--	--
10000	--	375	375	--	44	--	--
30000	--	1125	3375	--	127	--	--
100000	--	3750	37500	--	416	--	--
300000	--	11250	337500	--	1241	--	--
1000000	--	37500	3750000	--	4134	--	--
$F_1 = F_2 = 0.010$							
300	--	1	1	--	3	--	--
1000	--	4	4	--	3	--	--
3000	--	11	11	--	3	--	--
10000	--	38	38	--	3	--	--
30000	--	113	113	--	4	--	--
100000	--	375	375	--	7	--	--
300000	--	1125	3375	--	15	--	--
1000000	--	3750	37500	--	50	--	--

Table C.1.3.5. Number of pages transferred on busiest interprocesor link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	113	113	--	127	--	--
1000	--	375	375	--	416	--	--
3000	--	1125	3375	--	1241	--	--
10000	--	3750	37500	--	4128	--	--
30000	--	11250	337500	--	12378	--	--
100000	--	37500	3750000	--	41253	--	--
300000	--	112500	33750000	--	123753	--	--
1000000	--	375000	375000000	--	412509	--	--
$F_1 = F_2 = 0.500$							
300	--	56	56	--	34	--	--
1000	--	188	188	--	106	--	--
3000	--	563	1125	--	312	--	--
10000	--	1875	9375	--	1034	--	--
30000	--	5625	84375	--	3097	--	--
100000	--	18750	937500	--	10316	--	--
300000	--	56250	8437500	--	30941	--	--
1000000	--	187500	93750000	--	103134	--	--
$F_1 = F_2 = 0.100$							
300	--	11	11	--	4	--	--
1000	--	38	38	--	7	--	--
3000	--	113	113	--	15	--	--
10000	--	375	375	--	44	--	--
30000	--	1125	3375	--	127	--	--
100000	--	3750	37500	--	416	--	--
300000	--	11250	337500	--	1241	--	--
1000000	--	37500	3750000	--	4134	--	--
$F_1 = F_2 = 0.010$							
300	--	1	1	--	3	--	--
1000	--	4	4	--	3	--	--
3000	--	11	11	--	3	--	--
10000	--	38	38	--	3	--	--
30000	--	113	113	--	4	--	--
100000	--	375	375	--	7	--	--
300000	--	1125	3375	--	15	--	--
1000000	--	3750	37500	--	50	--	--

Table C.1.3.6. Number of pages transferred on busiest interprocesor link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TRAFFIC ON DISK LINK (PAGES)							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	75	11	4	--	24	24	--
1000	251	53	16	--	48	36	--
3000	754	207	192	--	144	84	--
10000	2513	730	1280	--	276	156	--
30000	7538	2687	15360	--	1044	564	--
100000	25125	10551	204800	--	4116	2190	--
300000	75375	33346	1228800	--	8208	4482	--
1000000	251250	126387	16384000	--	32784	17646	--
$F_1 = F_2 = 0.500$							
300	68	6	4	--	24	24	--
1000	226	29	16	--	48	36	--
3000	679	117	128	--	144	84	--
10000	2263	418	640	--	276	156	--
30000	6788	1563	7680	--	1044	564	--
100000	22625	6178	102400	--	4116	2190	--
300000	67875	18346	614400	--	8208	4482	--
1000000	226250	70140	8192000	--	32784	17646	--
$F_1 = F_2 = 0.100$							
300	62	4	4	--	24	24	--
1000	206	17	16	--	48	36	--
3000	619	71	64	--	144	84	--
10000	2063	165	128	--	276	156	--
30000	6188	655	1536	--	1044	564	--
100000	20625	2650	20480	--	4116	2190	--
300000	61875	6271	122880	--	8208	4482	--
1000000	206250	24887	1638400	--	32784	17646	--
$F_1 = F_2 = 0.010$							
300	61	4	4	--	24	24	--
1000	202	16	16	--	48	36	--
3000	605	64	64	--	144	84	--
10000	2018	129	128	--	276	156	--
30000	6053	519	512	--	1044	564	--
100000	20175	2085	2048	--	4116	2190	--
300000	60525	4239	12288	--	8208	4482	--
1000000	201750	16986	163840	--	32784	17646	--

Table C.1.4.1. Number of pages transferred on average disk link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TRAFFIC ON DISK LINK (PAGES)							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	75	11	4	76	24	24	36
1000	251	53	16	253	48	36	48
3000	754	207	192	758	144	84	96
10000	2513	730	1280	60285	276	156	174
30000	7538	2687	15360	380367	1044	564	606
100000	25125	10551	204800	1574191	4116	2190	2322
300000	75375	33346	1228800	4985220	8208	4482	4860
1000000	251250	126387	16384000	16924152	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	68	6	4	20	24	24	36
1000	226	29	16	65	48	36	48
3000	679	117	128	195	144	84	96
10000	2263	418	640	32150	276	156	174
30000	6788	1563	7680	222470	1044	564	606
100000	22625	6178	102400	1109092	4116	2190	2322
300000	67875	18346	614400	3642345	8208	4482	4860
1000000	226250	70140	8192000	12508954	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	62	4	4	2	24	24	36
1000	206	17	16	5	48	36	48
3000	619	71	64	15	144	84	96
10000	2063	165	128	50	276	156	174
30000	6188	655	1536	150	1044	564	606
100000	20625	2650	20480	578010	4116	2190	2322
300000	61875	6271	122880	2406042	8208	4482	4860
1000000	206250	24887	1638400	8877691	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	61	1	1	1	24	24	36
1000	202	3	3	3	48	36	48
3000	605	8	8	7	144	84	96
10000	2018	26	25	24	276	156	174
30000	6053	83	75	71	1044	564	606
100000	20175	288	251	236	4116	2190	2322
300000	60525	897	2261	707	8208	4482	4860
1000000	201750	3114	25125	5777367	32784	17646	18900

Table C.1.4.2. Number of pages transferred on average disk link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TRAFFIC ON DISK LINK (PAGES)							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	4	11	4	5	24	24	36
1000	14	53	16	15	48	36	48
3000	41	207	192	45	144	84	96
10000	138	730	1280	162	276	156	174
30000	413	2687	15360	500	1044	564	606
100000	1375	10551	204800	1717	4116	2190	2322
300000	4125	33346	1228800	5295	8208	4482	4860
1000000	13750	126387	16384000	18152	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	4	6	4	5	24	24	36
1000	14	29	16	15	48	36	48
3000	41	117	128	45	144	84	96
10000	138	418	640	150	276	156	174
30000	413	1563	7680	474	1044	564	606
100000	1375	6178	102400	1605	4116	2190	2322
300000	4125	18346	614400	4883	8208	4482	4860
1000000	13750	70140	8192000	16580	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	4	4	4	2	24	24	36
1000	14	17	16	5	48	36	48
3000	40	71	64	14	144	84	96
10000	135	165	128	48	276	156	174
30000	403	655	1536	143	1044	564	606
100000	1345	2650	20480	490	4116	2190	2322
300000	4034	6271	122880	1482	8208	4482	4860
1000000	13446	24887	1638400	4993	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	4	1	1	1	24	24	36
1000	12	3	3	3	48	36	48
3000	35	8	8	7	144	84	96
10000	117	26	25	24	276	156	174
30000	351	83	75	71	1044	564	606
100000	1171	288	251	236	4116	2190	2322
300000	3512	897	2261	707	8208	4482	4860
1000000	11705	3114	25125	2370	32784	17646	18900

Table C.1.4.3. Number of pages transferred on average disk link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TRAFFIC ON DISK LINK (PAGES)							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	75	11	4	76	24	24	36
1000	251	50	14	253	48	36	48
3000	754	185	124	758	144	84	96
10000	2513	739	1375	60285	276	156	174
30000	7538	2588	12375	380367	1044	564	606
100000	25125	9878	137500	1574191	4116	2190	2322
300000	75375	33375	1237500	4985220	8208	4482	4860
1000000	251250	123753	13750000	16924152	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	68	4	2	20	24	24	36
1000	226	20	7	65	48	36	48
3000	679	74	41	195	144	84	96
10000	2263	359	344	32150	276	156	174
30000	6788	1257	3094	222470	1044	564	606
100000	22625	4817	34375	1109092	4116	2190	2322
300000	67875	16313	309375	3642345	8208	4482	4860
1000000	226250	60631	3437500	12508954	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	62	1	1	2	24	24	36
1000	206	3	1	5	48	36	48
3000	619	11	4	15	144	84	96
10000	2063	50	14	50	276	156	174
30000	6188	185	124	150	1044	564	606
100000	20625	739	1375	578010	4116	2190	2322
300000	61875	2588	12375	2406042	8208	4482	4860
1000000	206250	9878	137500	8877691	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	61	0	0	1	24	24	36
1000	202	0	0	3	48	36	48
3000	605	1	1	7	144	84	96
10000	2018	3	1	24	276	156	174
30000	6053	11	4	71	1044	564	606
100000	20175	50	14	236	4116	2190	2322
300000	60525	185	124	707	8208	4482	4860
1000000	201750	739	1375	5777367	32784	17646	18900

Table C.1.4.4. Number of pages transferred on average disk link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TRAFFIC ON DISK LINK (PAGES)							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	11	4	--	24	--	--
1000	--	53	16	--	48	--	--
3000	--	207	192	--	144	--	--
10000	--	730	1280	--	276	--	--
30000	--	2687	15360	--	1044	--	--
100000	--	10551	204800	--	4116	--	--
300000	--	33346	1228800	--	8208	--	--
1000000	--	126387	16384000	--	32784	--	--
$F_1 = F_2 = 0.500$							
300	--	6	4	--	24	--	--
1000	--	29	16	--	48	--	--
3000	--	117	128	--	144	--	--
10000	--	418	640	--	276	--	--
30000	--	1563	7680	--	1044	--	--
100000	--	6178	102400	--	4116	--	--
300000	--	18346	614400	--	8208	--	--
1000000	--	70140	8192000	--	32784	--	--
$F_1 = F_2 = 0.100$							
300	--	4	4	--	24	--	--
1000	--	17	16	--	48	--	--
3000	--	71	64	--	144	--	--
10000	--	165	128	--	276	--	--
30000	--	655	1536	--	1044	--	--
100000	--	2650	20480	--	4116	--	--
300000	--	6271	122880	--	8208	--	--
1000000	--	24887	1638400	--	32784	--	--
$F_1 = F_2 = 0.010$							
300	--	1	1	--	24	--	--
1000	--	3	3	--	48	--	--
3000	--	8	8	--	144	--	--
10000	--	26	25	--	276	--	--
30000	--	83	75	--	1044	--	--
100000	--	288	251	--	4116	--	--
300000	--	897	2261	--	8208	--	--
1000000	--	3114	25125	--	32784	--	--

Table C.1.4.5. Number of pages transferred on average disk link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TRAFFIC ON DISK LINK (PAGES)							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	11	4	--	24	--	--
1000	--	53	16	--	48	--	--
3000	--	207	192	--	144	--	--
10000	--	730	1280	--	276	--	--
30000	--	2687	15360	--	1044	--	--
100000	--	10551	204800	--	4116	--	--
300000	--	33346	1228800	--	8208	--	--
1000000	--	126387	16384000	--	32784	--	--
$F_1 = F_2 = 0.500$							
300	--	6	4	--	24	--	--
1000	--	29	16	--	48	--	--
3000	--	117	128	--	144	--	--
10000	--	418	640	--	276	--	--
30000	--	1563	7680	--	1044	--	--
100000	--	6178	102400	--	4116	--	--
300000	--	18346	614400	--	8208	--	--
1000000	--	70140	8192000	--	32784	--	--
$F_1 = F_2 = 0.100$							
300	--	4	4	--	24	--	--
1000	--	17	16	--	48	--	--
3000	--	71	64	--	144	--	--
10000	--	165	128	--	276	--	--
30000	--	655	1536	--	1044	--	--
100000	--	2650	20480	--	4116	--	--
300000	--	6271	122880	--	8208	--	--
1000000	--	24887	1638400	--	32784	--	--
$F_1 = F_2 = 0.010$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	129	128	--	276	--	--
30000	--	519	512	--	1044	--	--
100000	--	2085	2048	--	4116	--	--
300000	--	4239	12288	--	8208	--	--
1000000	--	16986	163840	--	32784	--	--

Table C.1.4.6. Number of pages transferred on average disk link. Parameters as assumed by Blasgen and Eswaren: P = 25, B = 1.

TOTAL INTERPROCESSOR COST							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	207	183	183	--	321	321	--
1000	691	609	609	--	790	790	--
3000	2072	1828	1828	--	2131	2131	--
10000	6906	6094	6094	--	6823	6823	--
30000	20719	18281	18281	--	20229	20229	--
100000	69063	60938	60938	--	67151	67151	--
300000	207188	182813	365625	--	201214	201214	--
1000000	690625	609375	4265625	--	670673	670673	--
$F_1 = F_2 = 0.500$							
300	104	91	91	--	170	170	--
1000	347	305	305	--	288	288	--
3000	1042	914	914	--	623	623	--
10000	3473	3047	3047	--	1796	1796	--
30000	10420	9141	9141	--	5147	5147	--
100000	34734	30469	30469	--	16878	16878	--
300000	104203	91406	91406	--	50393	50393	--
1000000	347344	304688	1218750	--	167938	167938	--
$F_1 = F_2 = 0.100$							
300	35	18	18	--	122	122	--
1000	117	61	61	--	127	127	--
3000	350	183	183	--	140	140	--
10000	1166	609	609	--	187	187	--
30000	3498	1828	1828	--	321	321	--
100000	11659	6094	6094	--	790	790	--
300000	34978	18281	18281	--	2131	2131	--
1000000	116594	60938	60938	--	7063	7063	--
$F_1 = F_2 = 0.010$							
300	21	2	2	--	120	120	--
1000	70	6	6	--	120	120	--
3000	210	18	18	--	120	120	--
10000	700	61	61	--	121	121	--
30000	2101	183	183	--	122	122	--
100000	7005	609	609	--	127	127	--
300000	21015	1828	1828	--	140	140	--
1000000	70050	6094	6094	--	427	427	--

Table C.2.1.1. Total processor communication cost C_C. Modified parameters: P = 4096, B = 6.

TOTAL INTERPROCESSOR COST							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	207	183	183	197	321	321	445
1000	691	609	609	658	790	790	930
3000	2072	1828	1828	1974	2131	2131	2321
10000	6906	6094	6094	6581	6823	6823	7396
30000	20719	18281	18281	19744	20229	20229	22051
100000	69063	60938	60938	65813	67151	67151	73516
300000	207188	182813	365625	197438	201214	201214	221978
1000000	690625	609375	4265625	658125	670673	670673	745760
$F_1 = F_2 = 0.500$							
300	104	91	91	51	170	170	292
1000	347	305	305	171	288	288	418
3000	1042	914	914	512	623	623	778
10000	3473	3047	3047	1706	1796	1796	2142
30000	10420	9141	9141	5119	5147	5147	6038
100000	34734	30469	30469	17063	16878	16878	20120
300000	104203	91406	91406	51188	50393	50393	60795
1000000	347344	304688	1218750	170625	167938	167938	205462
$F_1 = F_2 = 0.100$							
300	35	18	18	3	122	122	242
1000	117	61	61	9	127	127	249
3000	350	183	183	26	140	140	267
10000	1166	609	609	88	187	187	336
30000	3498	1828	1828	263	321	321	547
100000	11659	6094	6094	878	790	790	1455
300000	34978	18281	18281	2633	2131	2131	4227
1000000	116594	60938	60938	8775	7063	7063	14568
$F_1 = F_2 = 0.010$							
300	21	2	2	0	120	120	240
1000	70	6	6	0	120	120	240
3000	210	18	18	1	120	120	241
10000	700	61	61	3	121	121	244
30000	2101	183	183	9	122	122	253
100000	7005	609	609	31	127	127	285
300000	21015	1828	1828	92	140	140	394
1000000	70050	6094	6094	307	427	427	1134

Table C.2.1.2. Total processor communication cost C_C. Modified parameters: P = 4096, B = 6.

TOTAL INTERPROCESSOR COST							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	41	183	183	82	160	160	284
1000	138	609	609	273	254	254	394
3000	414	1828	1828	819	522	522	712
10000	1381	6094	6094	2730	1461	1461	2034
30000	4144	18281	18281	8190	4142	4142	5964
100000	13813	60938	60938	27300	13526	13526	19891
300000	41438	182813	365625	81900	40339	40339	61103
1000000	138125	609375	4265625	273000	134423	134423	209510
$F_1 = F_2 = 0.500$							
300	12	91	91	22	130	130	252
1000	39	305	305	74	154	154	284
3000	116	914	914	223	221	221	375
10000	386	3047	3047	743	455	455	802
30000	1158	9141	9141	2230	1125	1125	2016
100000	3859	30469	30469	7434	3472	3472	6714
300000	11578	91406	91406	22303	10175	10175	20577
1000000	38594	304688	1218750	74344	33876	33876	71399
$F_1 = F_2 = 0.100$							
300	1	18	18	1	120	120	241
1000	3	61	61	5	121	121	243
3000	9	183	183	15	124	124	251
10000	28	609	609	49	133	133	283
30000	85	1828	1828	148	160	160	386
100000	284	6094	6094	492	254	254	919
300000	853	18281	18281	1477	522	522	2619
1000000	2844	60938	60938	4924	1701	1701	9205
$F_1 = F_2 = 0.010$							
300	0	2	2	0	120	120	240
1000	0	6	6	0	120	120	240
3000	1	18	18	1	120	120	241
10000	2	61	61	3	120	120	243
30000	5	183	183	8	120	120	251
100000	17	609	609	27	121	121	280
300000	52	1828	1828	81	124	124	378
1000000	175	6094	6094	269	373	373	1080

Table C.2.1.3. Total processor communication cost C_C. Modified parameters: P = 4096, B = 6.

TOTAL INTERPROCESSOR COST							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	207	183	183	197	321	321	445
1000	691	609	609	658	790	790	930
3000	2072	1828	1828	1974	2131	2131	2321
10000	6906	6094	6094	6581	6823	6823	7396
30000	20719	18281	18281	19744	20229	20229	22051
100000	69063	60938	60938	65813	67151	67151	73516
300000	207188	182813	365625	197438	201214	201214	221978
1000000	690625	609375	4265625	658125	670673	670673	745760
$F_1 = F_2 = 0.500$							
300	104	91	91	51	170	170	292
1000	347	305	305	171	288	288	418
3000	1042	914	914	512	623	623	778
10000	3473	3047	3047	1706	1796	1796	2142
30000	10420	9141	9141	5119	5147	5147	6038
100000	34734	30469	30469	17063	16878	16878	20120
300000	104203	91406	91406	51188	50393	50393	60795
1000000	347344	304688	1218750	170625	167938	167938	205462
$F_1 = F_2 = 0.100$							
300	35	18	18	3	122	122	242
1000	117	61	61	9	127	127	249
3000	350	183	183	26	140	140	267
10000	1166	609	609	88	187	187	336
30000	3498	1828	1828	263	321	321	547
100000	11659	6094	6094	878	790	790	1455
300000	34978	18281	18281	2633	2131	2131	4227
1000000	116594	60938	60938	8775	7063	7063	14568
$F_1 = F_2 = 0.010$							
300	21	2	2	0	120	120	240
1000	70	6	6	0	120	120	240
3000	210	18	18	1	120	120	241
10000	700	61	61	3	121	121	244
30000	2101	183	183	9	122	122	253
100000	7005	609	609	31	127	127	285
300000	21015	1828	1828	92	140	140	394
1000000	70050	6094	6094	307	427	427	1134

Table C.2.1.4. Total processor communication cost C_C. Modified parameters: P = 4096, B = 6.

TOTAL INTERPROCESSOR COST							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
F_2=1.0							
300	--	183	183	--	321	--	--
1000	--	609	609	--	790	--	--
3000	--	1828	1828	--	2131	--	--
10000	--	6094	6094	--	6823	--	--
30000	--	18281	18281	--	20229	--	--
100000	--	60938	60938	--	67151	--	--
300000	--	182813	365625	--	201214	--	--
1000000	--	609375	4265625	--	670673	--	--
$F_1 = F_2 = 0.500$							
300	--	91	91	--	170	--	--
1000	--	305	305	--	288	--	--
3000	--	914	914	--	623	--	--
10000	--	3047	3047	--	1796	--	--
30000	--	9141	9141	--	5147	--	--
100000	--	30469	30469	--	16878	--	--
300000	--	91406	91406	--	50393	--	--
1000000	--	304688	1218750	--	167938	--	--
$F_1 = F_2 = 0.100$							
300	--	18	18	--	122	--	--
1000	--	61	61	--	127	--	--
3000	--	183	183	--	140	--	--
10000	--	609	609	--	187	--	--
30000	--	1828	1828	--	321	--	--
100000	--	6094	6094	--	790	--	--
300000	--	18281	18281	--	2131	--	--
1000000	--	60938	60938	--	7063	--	--
$F_1 = F_2 = 0.010$							
300	--	2	2	--	120	--	--
1000	--	6	6	--	120	--	--
3000	--	18	18	--	120	--	--
10000	--	61	61	--	121	--	--
30000	--	183	183	--	122	--	--
100000	--	609	609	--	127	--	--
300000	--	1828	1828	--	140	--	--
1000000	--	6094	6094	--	427	--	--

Table C.2.1.5. Total processor communication cost C_C. Modified parameters: P = 4096, B = 6.

TOTAL INTERPROCESSOR COST							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	183	183	--	321	--	--
1000	--	609	609	--	790	--	--
3000	--	1828	1828	--	2131	--	--
10000	--	6094	6094	--	6823	--	--
30000	--	18281	18281	--	20229	--	--
100000	--	60938	60938	--	67151	--	--
300000	--	182813	365625	--	201214	--	--
1000000	--	609375	4265625	--	670673	--	--
$F_1 = F_2 = 0.500$							
300	--	91	91	--	170	--	--
1000	--	305	305	--	288	--	--
3000	--	914	914	--	623	--	--
10000	--	3047	3047	--	1796	--	--
30000	--	9141	9141	--	5147	--	--
100000	--	30469	30469	--	16878	--	--
300000	--	91406	91406	--	50393	--	--
1000000	--	304688	1218750	--	167938	--	--
$F_1 = F_2 = 0.100$							
300	--	18	18	--	122	--	--
1000	--	61	61	--	127	--	--
3000	--	183	183	--	140	--	--
10000	--	609	609	--	187	--	--
30000	--	1828	1828	--	321	--	--
100000	--	6094	6094	--	790	--	--
300000	--	18281	18281	--	2131	--	--
1000000	--	60938	60938	--	7063	--	--
$F_1 = F_2 = 0.010$							
300	--	2	2	--	120	--	--
1000	--	6	6	--	120	--	--
3000	--	18	18	--	120	--	--
10000	--	61	61	--	121	--	--
30000	--	183	183	--	122	--	--
100000	--	609	609	--	127	--	--
300000	--	1828	1828	--	140	--	--
1000000	--	6094	6094	--	427	--	--

Table C.2.1.6. Total processor communication cost C_C. Modified parameters: P = 4096, B = 6.

TOTAL DISK COST							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	1508	80	80	--	24	24	--
1000	5025	320	320	--	48	36	--
3000	15075	1280	1280	--	144	84	--
10000	50250	2560	2560	--	276	156	--
30000	150750	10276	10240	--	1044	564	--
100000	502500	41356	40960	--	4116	2190	--
300000	1507500	83552	163840	--	8208	4482	--
1000000	5025000	334460	2293760	--	32784	17646	--
$F_1 = F_2 = 0.500$							
300	1358	80	80	--	24	24	--
1000	4525	320	320	--	48	36	--
3000	13575	1280	1280	--	144	84	--
10000	45250	2560	2560	--	276	156	--
30000	135750	10264	10240	--	1044	564	--
100000	452500	41104	40960	--	4116	2190	--
300000	1357500	82556	81920	--	8208	4482	--
1000000	4525000	330600	1310720	--	32784	17646	--
$F_1 = F_2 = 0.100$							
300	1238	80	80	--	24	24	--
1000	4125	320	320	--	48	36	--
3000	12375	1280	1280	--	144	84	--
10000	41250	2560	2560	--	276	156	--
30000	123750	10240	10240	--	1044	564	--
100000	412500	40960	40960	--	4116	2190	--
300000	1237500	81956	81920	--	8208	4482	--
1000000	4125000	328076	327680	--	32784	17646	--
$F_1 = F_2 = 0.010$							
300	1211	80	80	--	24	24	--
1000	4035	320	320	--	48	36	--
3000	12105	1280	1280	--	144	84	--
10000	40350	2560	2560	--	276	156	--
30000	121050	10240	10240	--	1044	564	--
100000	403500	40960	40960	--	4116	2190	--
300000	1210500	81920	81920	--	8208	4482	--
1000000	4035000	327680	327680	--	32784	17646	--

Table C.2.2.1. Total disk communication cost C_D. Modified parameters: P = 4096, B = 6.

TOTAL DISK COST							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	1508	80	80	1516	24	24	36
1000	5025	320	320	5050	48	36	48
3000	15075	1280	1280	15150	144	84	96
10000	50250	2560	2560	50500	276	156	174
30000	150750	10276	10240	151500	1044	564	606
100000	502500	41356	40960	505000	4116	2190	2322
300000	1507500	83552	163840	1515000	8208	4482	4860
1000000	5025000	334460	2293760	87034096	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	1358	80	80	391	24	24	36
1000	4525	320	320	1300	48	36	48
3000	13575	1280	1280	3900	144	84	96
10000	45250	2560	2560	13000	276	156	174
30000	135750	10264	10240	39000	1044	564	606
100000	452500	41104	40960	130000	4116	2190	2322
300000	1357500	82556	81920	390000	8208	4482	4860
1000000	4525000	330500	1310720	1300000	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	1238	80	80	31	24	24	36
1000	4125	320	320	100	48	36	48
3000	12375	1280	1280	300	144	84	96
10000	41250	2560	2560	1000	276	156	174
30000	123750	10240	10240	3000	1044	564	606
100000	412500	40960	40960	10000	4116	2190	2322
300000	1237500	81956	81920	30000	8208	4482	4860
1000000	4125000	328076	327680	100000	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	1211	17	17	18	24	24	36
1000	4035	52	52	50	48	36	48
3000	12105	152	152	145	144	84	96
10000	40350	503	503	473	276	156	174
30000	121050	1508	1508	1416	1044	564	606
100000	403500	5025	5025	4717	4116	2190	2322
300000	1210500	15075	15075	14147	8208	4482	4860
1000000	4035000	50250	50250	47152	32784	17646	18900

Table C.2.2.2. Total disk communication cost C_D. Modified parameters: P = 4096, B = 6.

TOTAL DISK COST							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	83	80	80	91	24	24	36
1000	275	320	320	300	48	36	48
3000	825	1280	1280	900	144	84	96
10000	2750	2560	2560	3000	276	156	174
30000	8250	10276	10240	9000	1044	564	606
100000	27500	41356	40960	30000	4116	2190	2322
300000	82500	83552	163840	90000	8208	4482	4860
1000000	275000	334460	2293760	300126	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	83	80	80	91	24	24	36
1000	275	320	320	300	48	36	48
3000	825	1280	1280	900	144	84	96
10000	2750	2560	2560	3000	276	156	174
30000	8250	10264	10240	9000	1044	564	606
100000	27500	41104	40960	30000	4116	2190	2322
300000	82500	82556	81920	90000	8208	4482	4860
1000000	275000	330500	1310720	300000	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	82	80	80	31	24	24	36
1000	270	320	320	97	48	36	48
3000	807	1280	1280	288	144	84	96
10000	2690	2560	2560	957	276	156	174
30000	8068	10240	10240	2867	1044	564	606
100000	26893	40960	40960	9553	4116	2190	2322
300000	80677	81956	81920	28658	8208	4482	4860
1000000	268922	328076	327680	95524	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	71	17	17	18	24	24	36
1000	235	52	52	50	48	36	48
3000	703	152	152	144	144	84	96
10000	2342	503	503	473	276	156	174
30000	7024	1508	1508	1416	1044	564	606
100000	23411	5025	5025	4717	4116	2190	2322
300000	70232	15075	15075	14147	8208	4482	4860
1000000	234105	50250	50250	47152	32784	17646	18900

Table C.2.2.3. Total disk communication cost C_D. Modified parameters: P = 4096, B = 6.

TOTAL DISK COST							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	1508	83	83	1516	24	24	36
1000	5025	275	275	5050	48	36	48
3000	15075	825	825	15150	144	84	96
10000	50250	2750	2750	50500	276	156	174
30000	150750	8286	8250	151500	1044	564	606
100000	502500	27896	27500	505000	4116	2190	2322
300000	1507500	84132	165000	1515000	8208	4482	4860
1000000	5025000	281780	1925000	87034096	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	1358	42	42	391	24	24	36
1000	4525	138	138	1300	48	36	48
3000	13575	413	413	3900	144	84	96
10000	45250	1375	1375	13000	276	156	174
30000	135750	4149	4125	39000	1044	564	606
100000	452500	13894	13750	130000	4116	2190	2322
300000	1357500	41886	41250	390000	8208	4482	4860
1000000	4525000	140320	550000	1300000	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	1238	10	10	31	24	24	36
1000	4125	28	28	100	48	36	48
3000	12375	83	83	300	144	84	96
10000	41250	275	275	1000	276	156	174
30000	123750	825	825	3000	1044	564	606
100000	412500	2750	2750	10000	4116	2190	2322
300000	1237500	8286	8250	30000	8208	4482	4860
1000000	4125000	27896	27500	100000	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	1211	4	4	18	24	24	36
1000	4035	5	5	50	48	36	48
3000	12105	10	10	145	144	84	96
10000	40350	28	28	473	276	156	174
30000	121050	83	83	1416	1044	564	606
100000	403500	275	275	4717	4116	2190	2322
300000	1210500	825	825	14147	8208	4482	4860
1000000	4035000	2750	2750	47152	32784	17646	18900

Table C.2.2.4. Total disk communication cost C_D. Modified parameters: P = 4096, B = 6.

TOTAL DISK COST							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10276	10240	--	1044	--	--
100000	--	41356	40960	--	4116	--	--
300000	--	83552	163840	--	8208	--	--
1000000	--	334460	2293760	--	32784	--	--
$F_1 = F_2 = 0.500$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10264	10240	--	1044	--	--
100000	--	41104	40960	--	4116	--	--
300000	--	82556	81920	--	8208	--	--
1000000	--	330500	1310720	--	32784	--	--
$F_1 = F_2 = 0.100$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10240	10240	--	1044	--	--
100000	--	40960	40960	--	4116	--	--
300000	--	81956	81920	--	8208	--	--
1000000	--	328076	327680	--	32784	--	--
$F_1 = F_2 = 0.010$							
300	--	17	17	--	24	--	--
1000	--	52	52	--	48	--	--
3000	--	152	152	--	144	--	--
10000	--	503	503	--	276	--	--
30000	--	1508	1508	--	1044	--	--
100000	--	5025	5025	--	4116	--	--
300000	--	15075	15075	--	8208	--	--
1000000	--	50250	50250	--	32784	--	--

Table C.2.2.5. Total disk communication cost C_D. Modified parameters: P = 4096, B = 6.

TOTAL DISK COST							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10276	10240	--	1044	--	--
100000	--	41356	40960	--	4116	--	--
300000	--	83552	163840	--	8208	--	--
1000000	--	334460	2293760	--	32784	--	--
$F_1 = F_2 = 0.500$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10264	10240	--	1044	--	--
100000	--	41104	40960	--	4116	--	--
300000	--	82556	81920	--	8208	--	--
1000000	--	330500	1310720	--	32784	--	--
$F_1 = F_2 = 0.100$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10240	10240	--	1044	--	--
100000	--	40960	40960	--	4116	--	--
300000	--	81956	81920	--	8208	--	--
1000000	--	328076	327680	--	32784	--	--
$F_1 = F_2 = 0.010$							
300	--	80	80	--	24	--	--
1000	--	320	320	--	48	--	--
3000	--	1280	1280	--	144	--	--
10000	--	2560	2560	--	276	--	--
30000	--	10240	10240	--	1044	--	--
100000	--	40960	40960	--	4116	--	--
300000	--	81920	81920	--	8208	--	--
1000000	--	327680	327680	--	32784	--	--

Table C.2.2.6. Total disk communication cost C_D. Modified parameters: P = 4096, B = 6.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	128	113	113	--	127	127	--
1000	425	375	375	--	416	416	--
3000	1275	1125	1125	--	1241	1241	--
10000	4250	3750	3750	--	4128	4128	--
30000	12750	11250	11250	--	12378	12378	--
100000	42500	37500	37500	--	41253	41253	--
300000	127500	112500	225000	--	123753	123753	--
1000000	425000	375000	2625000	--	412509	412509	--
$F_1 = F_2 = 0.500$							
300	64	56	56	--	34	34	--
1000	214	188	188	--	106	106	--
3000	641	563	563	--	312	312	--
10000	2138	1875	1875	--	1034	1034	--
30000	6413	5625	5625	--	3097	3097	--
100000	21375	18750	18750	--	10316	10316	--
300000	64125	56250	56250	--	30941	30941	--
1000000	213750	187500	750000	--	103134	103134	--
$F_1 = F_2 = 0.100$							
300	22	11	11	--	4	4	--
1000	72	38	38	--	7	7	--
3000	215	113	113	--	15	15	--
10000	718	375	375	--	44	44	--
30000	2153	1125	1125	--	127	127	--
100000	7175	3750	3750	--	416	416	--
300000	21525	11250	11250	--	1241	1241	--
1000000	71750	37500	37500	--	4134	4134	--
$F_1 = F_2 = 0.010$							
300	13	1	1	--	3	3	--
1000	43	4	4	--	3	3	--
3000	129	11	11	--	3	3	--
10000	431	38	38	--	3	3	--
30000	1293	113	113	--	4	4	--
100000	4311	375	375	--	7	7	--
300000	12932	1125	1125	--	15	15	--
1000000	43108	3750	3750	--	50	50	--

Table C.2.3.1. Number of pages transferred on busiest interprocesor link. Modified parameters: P = 4096, B = 6.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	128	113	113	122	127	127	131
1000	425	375	375	405	416	416	420
3000	1275	1125	1125	1215	1241	1241	1245
10000	4250	3750	3750	4050	4128	4128	4136
30000	12750	11250	11250	12150	12378	12378	12398
100000	42500	37500	37500	40500	41253	41253	41317
300000	127500	112500	225000	121500	123753	123753	123939
1000000	425000	375000	2625000	405000	412509	412509	413123
$F_1 = F_2 = 0.500$							
300	64	56	56	32	34	34	38
1000	214	188	188	105	106	106	110
3000	641	563	563	315	312	312	316
10000	2138	1875	1875	1050	1034	1034	1040
30000	6413	5625	5625	3150	3097	3097	3107
100000	21375	18750	18750	10500	10316	10316	10350
300000	64125	56250	56250	31500	30941	30941	31035
1000000	213750	187500	750000	105000	103134	103134	103442
$F_1 = F_2 = 0.100$							
300	22	11	11	2	4	4	8
1000	72	38	38	5	7	7	11
3000	215	113	113	16	15	15	19
10000	718	375	375	54	44	44	48
30000	2153	1125	1125	162	127	127	131
100000	7175	3750	3750	540	416	416	424
300000	21525	11250	11250	1620	1241	1241	1261
1000000	71750	37500	37500	5400	4134	4134	4198
$F_1 = F_2 = 0.010$							
300	13	1	1	0	3	3	7
1000	43	4	4	0	3	3	7
3000	129	11	11	1	3	3	7
10000	431	38	38	2	3	3	7
30000	1293	113	113	6	4	4	8
100000	4311	375	375	19	7	7	11
300000	12932	1125	1125	57	15	15	19
1000000	43108	3750	3750	189	50	50	58

Table C.2.3.2. Number of pages transferred on busiest interprocesor link. Modified parameters: P = 4096, B = 6.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	26	113	113	50	28	28	32
1000	85	375	375	168	86	86	90
3000	255	1125	1125	504	251	251	255
10000	850	3750	3750	1680	828	828	836
30000	2550	11250	11250	5040	2478	2478	2498
100000	8500	37500	37500	16800	8253	8253	8317
300000	25500	112500	225000	50400	24753	24753	24939
1000000	85000	375000	2625000	168000	82509	82509	83123
$F_1 = F_2 = 0.500$							
300	7	56	56	14	9	9	13
1000	24	188	188	46	24	24	28
3000	71	563	563	137	65	65	69
10000	238	1875	1875	458	209	209	215
30000	713	5625	5625	1373	622	622	632
100000	2375	18750	18750	4575	2066	2066	2100
300000	7125	56250	56250	13725	6191	6191	6285
1000000	23750	187500	750000	45750	20634	20634	20942
$F_1 = F_2 = 0.100$							
300	1	11	11	1	3	3	7
1000	2	38	38	3	4	4	8
3000	5	113	113	9	5	5	9
10000	18	375	375	30	11	11	15
30000	53	1125	1125	91	28	28	32
100000	175	3750	3750	303	86	86	94
300000	525	11250	11250	909	251	251	271
1000000	1750	37500	37500	3030	834	834	898
$F_1 = F_2 = 0.010$							
300	0	1	1	0	3	3	7
1000	0	4	4	0	3	3	7
3000	0	11	11	0	3	3	7
10000	1	38	38	2	3	3	7
30000	3	113	113	5	3	3	7
100000	11	375	375	17	4	4	8
300000	32	1125	1125	50	5	5	9
1000000	108	3750	3750	165	17	17	30

Table C.2.3.3. Number of pages transferred on busiest interprocesor link. Modified parameters: P = 4096, B = 6.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION D
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	128	113	113	122	127	127	131
1000	425	375	375	405	416	416	420
3000	1275	1125	1125	1215	1241	1241	1245
10000	4250	3750	3750	4050	4128	4128	4136
30000	12750	11250	11250	12150	12378	12378	12398
100000	42500	37500	37500	40500	41253	41253	41317
300000	127500	112500	225000	121500	123753	123753	123939
1000000	425000	375000	2625000	405000	412509	412509	413123
$F_1 = F_2 = 0.500$							
300	64	56	56	32	34	34	38
1000	214	188	188	105	106	106	110
3000	641	563	563	315	312	312	316
10000	2138	1875	1875	1050	1034	1034	1040
30000	6413	5625	5625	3150	3097	3097	3107
100000	21375	18750	18750	10500	10316	10316	10350
300000	64125	56250	56250	31500	30941	30941	31035
1000000	213750	187500	750000	105000	103134	103134	103442
$F_1 = F_2 = 0.100$							
300	22	11	11	2	4	4	8
1000	72	38	38	5	7	7	11
3000	215	113	113	16	15	15	19
10000	718	375	375	54	44	44	48
30000	2153	1125	1125	162	127	127	131
100000	7175	3750	3750	540	416	416	424
300000	21525	11250	11250	1620	1241	1241	1261
1000000	71750	37500	37500	5400	4134	4134	4198
$F_1 = F_2 = 0.010$							
300	13	1	1	0	3	3	7
1000	43	4	4	0	3	3	7
3000	129	11	11	1	3	3	7
10000	431	38	38	2	3	3	7
30000	1293	113	113	6	4	4	8
100000	4311	375	375	19	7	7	11
300000	12932	1125	1125	57	15	15	19
1000000	43108	3750	3750	189	50	50	58

Table C.2.3.4. Number of pages transferred on busiest interprocesor link. Modified parameters: P = 4096, B = 6.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	113	113	--	127	--	--
1000	--	375	375	--	416	--	--
3000	--	1125	1125	--	1241	--	--
10000	--	3750	3750	--	4128	--	--
30000	--	11250	11250	--	12378	--	--
100000	--	37500	37500	--	41253	--	--
300000	--	112500	225000	--	123753	--	--
1000000	--	375000	2625000	--	412509	--	--
$F_1 = F_2 = 0.500$							
300	--	56	56	--	34	--	--
1000	--	188	188	--	106	--	--
3000	--	563	563	--	312	--	--
10000	--	1875	1875	--	1034	--	--
30000	--	5625	5625	--	3097	--	--
100000	--	18750	18750	--	10316	--	--
300000	--	56250	56250	--	30941	--	--
1000000	--	187500	750000	--	103134	--	--
$F_1 = F_2 = 0.100$							
300	--	11	11	--	4	--	--
1000	--	38	38	--	7	--	--
3000	--	113	113	--	15	--	--
10000	--	375	375	--	44	--	--
30000	--	1125	1125	--	127	--	--
100000	--	3750	3750	--	416	--	--
300000	--	11250	11250	--	1241	--	--
1000000	--	37500	37500	--	4134	--	--
$F_1 = F_2 = 0.010$							
300	--	1	1	--	3	--	--
1000	--	4	4	--	3	--	--
3000	--	11	11	--	3	--	--
10000	--	38	38	--	3	--	--
30000	--	113	113	--	4	--	--
100000	--	375	375	--	7	--	--
300000	--	1125	1125	--	15	--	--
1000000	--	3750	3750	--	50	--	--

Table C.2.3.5. Number of pages transferred on busiest interprocesor link. Modified parameters: P = 4096, B = 6.

BUSIEST INTERPROCESSOR LINK (PAGES)							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	113	113	--	127	--	--
1000	--	375	375	--	416	--	--
3000	--	1125	1125	--	1241	--	--
10000	--	3750	3750	--	4128	--	--
30000	--	11250	11250	--	12378	--	--
100000	--	37500	37500	--	41253	--	--
300000	--	112500	225000	--	123753	--	--
1000000	--	375000	2625000	--	412509	--	--
$F_1 = F_2 = 0.500$							
300	--	56	56	--	34	--	--
1000	--	188	188	--	106	--	--
3000	--	563	563	--	312	--	--
10000	--	1875	1875	--	1034	--	--
30000	--	5625	5625	--	3097	--	--
100000	--	18750	18750	--	10316	--	--
300000	--	56250	56250	--	30941	--	--
1000000	--	187500	750000	--	103134	--	--
$F_1 = F_2 = 0.100$							
300	--	11	11	--	4	--	--
1000	--	38	38	--	7	--	--
3000	--	113	113	--	15	--	--
10000	--	375	375	--	44	--	--
30000	--	1125	1125	--	127	--	--
100000	--	3750	3750	--	416	--	--
300000	--	11250	11250	--	1241	--	--
1000000	--	37500	37500	--	4134	--	--
$F_1 = F_2 = 0.010$							
300	--	1	1	--	3	--	--
1000	--	4	4	--	3	--	--
3000	--	11	11	--	3	--	--
10000	--	38	38	--	3	--	--
30000	--	113	113	--	4	--	--
100000	--	375	375	--	7	--	--
300000	--	1125	1125	--	15	--	--
1000000	--	3750	3750	--	50	--	--

Table C.2.3.6. Number of pages transferred on busiest interprocesor link. Modified parameters: P = 4096, B = 6.

TRAFFIC ON DISK LINK (PAGES)							SITUATION A
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	75	4	4	--	24	24	--
1000	251	16	16	--	48	36	--
3000	754	64	64	--	144	84	--
10000	2513	128	128	--	276	156	--
30000	7538	819	512	--	1044	564	--
100000	25125	5427	2048	--	4116	2190	--
300000	75375	18022	8192	--	8208	4482	--
1000000	251250	74240	114688	--	32784	17646	--
$F_1 = F_2 = 0.500$							
300	68	4	4	--	24	24	--
1000	226	16	16	--	48	36	--
3000	679	64	64	--	144	84	--
10000	2263	128	128	--	276	156	--
30000	6788	717	512	--	1044	564	--
100000	22625	3277	2048	--	4116	2190	--
300000	67875	9523	4096	--	8208	4482	--
1000000	226250	40448	65536	--	32784	17646	--
$F_1 = F_2 = 0.100$							
300	62	4	4	--	24	24	--
1000	206	16	16	--	48	36	--
3000	619	64	64	--	144	84	--
10000	2063	128	128	--	276	156	--
30000	6188	512	512	--	1044	564	--
100000	20625	2048	2048	--	4116	2190	--
300000	61875	4403	4096	--	8208	4482	--
1000000	206250	19763	16384	--	32784	17646	--
$F_1 = F_2 = 0.010$							
300	61	4	4	--	24	24	--
1000	202	16	16	--	48	36	--
3000	605	64	64	--	144	84	--
10000	2018	128	128	--	276	156	--
30000	6053	512	512	--	1044	564	--
100000	20175	2048	2048	--	4116	2190	--
300000	60525	4096	4096	--	8208	4482	--
1000000	201750	16384	16384	--	32784	17646	--

Table C.2.4.1. Number of pages transferred on average disk link. Modified parameters: P = 4096, B = 6.

TRAFFIC ON DISK LINK (PAGES)							SITUATION B
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	75	4	4	76	24	24	36
1000	251	16	16	253	48	36	48
3000	754	64	64	758	144	84	96
10000	2513	128	128	2525	276	156	174
30000	7538	819	512	7575	1044	564	606
100000	25125	5427	2048	25250	4116	2190	2322
300000	75375	18022	8192	75750	8208	4482	4860
1000000	251250	74240	114688	699850119	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	68	4	4	20	24	24	36
1000	226	16	16	65	48	36	48
3000	679	64	64	195	144	84	96
10000	2263	128	128	650	276	156	174
30000	6788	717	512	1950	1044	564	606
100000	22625	3277	2048	6500	4116	2190	2322
300000	67875	9523	4096	19500	8208	4482	4860
1000000	226250	40448	65536	65000	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	62	4	4	2	24	24	36
1000	206	16	16	5	48	36	48
3000	619	64	64	15	144	84	96
10000	2063	128	128	50	276	156	174
30000	6188	512	512	150	1044	564	606
100000	20625	2048	2048	500	4116	2190	2322
300000	61875	4403	4096	1500	8208	4482	4860
1000000	206250	19763	16384	5000	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	61	1	1	1	24	24	36
1000	202	3	3	3	48	36	48
3000	605	8	8	7	144	84	96
10000	2018	25	25	24	276	156	174
30000	6053	75	75	71	1044	564	606
100000	20175	251	251	236	4116	2190	2322
300000	60525	754	754	707	8208	4482	4860
1000000	201750	2513	2513	2358	32784	17646	18900

Table C.2.4.2. Number of pages transferred on average disk link. Modified parameters: P = 4096, B = 6.

TRAFFIC ON DISK LINK (PAGES)							SITUATION C
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	4	4	4	5	24	24	36
1000	14	16	16	15	48	36	48
3000	41	64	64	45	144	84	96
10000	138	128	128	150	276	156	174
30000	413	819	512	450	1044	564	606
100000	1375	5427	2048	1500	4116	2190	2322
300000	4125	18022	8192	4500	8208	4482	4860
1000000	13750	74240	114688	16075	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	4	4	4	5	24	24	36
1000	14	16	16	15	48	36	48
3000	41	64	64	45	144	84	96
10000	138	128	128	150	276	156	174
30000	413	717	512	450	1044	564	606
100000	1375	3277	2048	1500	4116	2190	2322
300000	4125	9523	4096	4500	8208	4482	4860
1000000	13750	40448	65536	15000	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	4	4	4	2	24	24	36
1000	14	16	16	5	48	36	48
3000	40	64	64	14	144	84	96
10000	135	128	128	48	276	156	174
30000	403	512	512	143	1044	564	606
100000	1345	2048	2048	478	4116	2190	2322
300000	4034	4403	4096	1433	8208	4482	4860
1000000	13446	19763	16384	4776	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	4	1	1	1	24	24	36
1000	12	3	3	3	48	36	48
3000	35	8	8	7	144	84	96
10000	117	25	25	24	276	156	174
30000	351	75	75	71	1044	564	606
100000	1171	251	251	236	4116	2190	2322
300000	3512	754	754	707	8208	4482	4860
1000000	11705	2513	2513	2358	32784	17646	18900

Table C.2.4.3. Number of pages transferred on average disk link. Modified parameters: P = 4096, B = 6.

TRAFFIC ON DISK LINK (PAGES)						SITUATION D	
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	75	4	4	76	24	24	36
1000	251	14	14	253	48	36	48
3000	754	41	41	758	144	84	96
10000	2513	138	138	2525	276	156	174
30000	7538	720	413	7575	1044	564	606
100000	25125	4754	1375	25250	4116	2190	2322
300000	75375	18051	8250	75750	8208	4482	4860
1000000	251250	71606	96250	699850119	32784	17646	18900
$F_1 = F_2 = 0.500$							
300	68	2	2	20	24	24	36
1000	226	7	7	65	48	36	48
3000	679	21	21	195	144	84	96
10000	2263	69	69	650	276	156	174
30000	6788	411	206	1950	1044	564	606
100000	22625	1916	688	6500	4116	2190	2322
300000	67875	7490	2063	19500	8208	4482	4860
1000000	226250	30939	27500	65000	32784	17646	18900
$F_1 = F_2 = 0.100$							
300	62	1	1	2	24	24	36
1000	206	1	1	5	48	36	48
3000	619	4	4	15	144	84	96
10000	2063	14	14	50	276	156	174
30000	6188	41	41	150	1044	564	606
100000	20625	138	138	500	4116	2190	2322
300000	61875	720	413	1500	8208	4482	4860
1000000	206250	4754	1375	5000	32784	17646	18900
$F_1 = F_2 = 0.010$							
300	61	0	0	1	24	24	36
1000	202	0	0	3	48	36	48
3000	605	1	1	7	144	84	96
10000	2018	1	1	24	276	156	174
30000	6053	4	4	71	1044	564	606
100000	20175	14	14	236	4116	2190	2322
300000	60525	41	41	707	8208	4482	4860
1000000	201750	138	138	2358	32784	17646	18900

Table C.2.4.4. Number of pages transferred on average disk link. Modified parameters: P = 4096, B = 6.

TRAFFIC ON DISK LINK (PAGES)							SITUATION E
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	128	128	--	276	--	--
30000	--	819	512	--	1044	--	--
100000	--	5427	2048	--	4116	--	--
300000	--	18022	8192	--	8208	--	--
1000000	--	74240	114688	--	32784	--	--
$F_1 = F_2 = 0.500$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	128	128	--	276	--	--
30000	--	717	512	--	1044	--	--
100000	--	3277	2048	--	4116	--	--
300000	--	9523	4096	--	8208	--	--
1000000	--	40448	65536	--	32784	--	--
$F_1 = F_2 = 0.100$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	128	128	--	276	--	--
30000	--	512	512	--	1044	--	--
100000	--	2048	2048	--	4116	--	--
300000	--	4403	4096	--	8208	--	--
1000000	--	19763	16384	--	32784	--	--
$F_1 = F_2 = 0.010$							
300	--	1	1	--	24	--	--
1000	--	3	3	--	48	--	--
3000	--	8	8	--	144	--	--
10000	--	25	25	--	276	--	--
30000	--	75	75	--	1044	--	--
100000	--	251	251	--	4116	--	--
300000	--	754	754	--	8208	--	--
1000000	--	2513	2513	--	32784	--	--

Table C.2.4.5. Number of pages transferred on average disk link. Modified parameters: P = 4096, B = 6.

TRAFFIC ON DISK LINK (PAGES)							SITUATION F
Size of N_2	Method						
	1	2	3	4	P1	P2	P3
$F_1 = F_2 = 1.0$							
300	–	4	4	–	24	–	–
1000	–	16	16	–	48	–	–
3000	–	64	64	–	144	–	–
10000	–	128	128	–	276	–	–
30000	–	819	512	–	1044	–	–
100000	–	5427	2048	–	4116	–	–
300000	–	18022	8192	–	8208	–	–
1000000	–	74240	114688	–	32784	–	–
$F_1 = F_2 = 0.500$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	128	128	--	276	--	--
30000	--	717	512	--	1044	--	--
100000	--	3277	2048	--	4116	--	--
300000	--	9523	4096	--	8208	--	--
1000000	--	40448	65536	--	32784	--	--
$F_1 = F_2 = 0.100$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	128	128	--	276	--	--
30000	--	512	512	--	1044	--	--
100000	--	2048	2048	--	4116	--	--
300000	--	4403	4096	--	8208	--	--
1000000	--	19763	16384	--	32784	--	--
$F_1 = F_2 = 0.010$							
300	--	4	4	--	24	--	--
1000	--	16	16	--	48	--	--
3000	--	64	64	--	144	--	--
10000	--	128	128	--	276	--	--
30000	--	512	512	--	1044	--	--
100000	--	2048	2048	--	4116	--	--
300000	--	4096	4096	--	8208	--	--
1000000	--	16384	16384	--	32784	--	--

Table C.2.4.6. Number of pages transferred on average disk link. Modified parameters: P = 4096, B = 6.

APPENDIX D

The Expected Number of Pages Fetched

In determining their disk cost functions, Blasgen and Eswaran assumed that the number of pages which had to be read in from the disk was proportional to the percentage of the records required. This estimate is quite good if the records being retrieved are clustered on a set of pages. This will generally be true in the case where the records required are the result of a simple restriction, and the restriction is on the clustering index, ignoring fragmentation. If, however, the records are specified as the result of a restriction on any other index, or if they are specified as those participating in a join operation, then there is no basis for expecting them to be clustered neatly on a small number of pages. In this section will be derived the expected number of page accesses required to fetch X records from a relation having N records distributed evenly with T records on each page.

For the selection of a single record, the probability that a particular page, i, will not be retrieved is $\frac{N - T}{N}$. The probability that page i will not be retrieved when two records are retrieved at random is

$$\frac{N - T}{N} \cdot \frac{N - T - 1}{N - 1}$$

In general, for X such retrievals of distinct records, if $N \geq T + X$ the probability P_i that a given page i is not selected is

$$P_i = \frac{N - T}{N} \cdot \frac{N - T - 1}{N - 1} \cdot \ldots \cdot \frac{N - T - X + 1}{N - X + 1}$$

$$= \frac{\binom{N - T}{X}}{\binom{N}{X}}.$$

For $N \leq T + X$, all pages are retrieved, so $P_i = 0$.

The expected number of pages fetched E is the sum of the probabilities that each page is fetched, so

$$E = \sum_i (1 - P_i) = \frac{N}{T} \cdot (1 - P_i)$$

$$= \frac{N}{T} \cdot \left\{ 1 - \frac{\binom{N - T}{X}}{\binom{N}{X}} \right\}.$$

In the Blasgen/Eswaran work, it was assumed that $E = \frac{X}{N}$. This estimate is a lower bound. In most cases, the difference between these two values was small enough that it made no difference, but under certain circumstances, the difference was substantial. In particular, in situations B, C, and D, when F_1 and F_2 were less than 1, methods 1 and 4 resulted in somewhat higher values than Eswaran and Blasgen predicted. For method 1, this occurred only in situation C, resulting in a 10 - 20 % increase in the disk access cost. For method 4, however, it resulted in the cost being increased by more than a factor of 3 in many cases for situation C and by as much as a factor of 2 for situations B and D. This is particularly significant because Blasgen and Eswaran concluded that method 4 was the method of choice under situation C! With this adjustment, method 4 is still superior to the other Blasgen/Eswaran methods in some cases, but not by so clear a margin.

The same formula may also be used to predict the number of unique values occurring in a hash table if T is interpreted as the number of unique values present in the hash field of N tuples, of which X are selected. However, this does require the somewhat unrealistic assumption that all values are equally represented.