MARCH 1989 VOL. 12 NO. 1

a quarterly bulletin of the IEEE Computer Society technical committee on

Data Engineering

CONTENTS

Letters to the TC Members 1
S. Jajodia and W. Kim (issue editors), and Larry Kerschberg (TC Chair)

Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience 2
R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos, and H. Young

Parallelizing FAD Using Compile-Time Analysis Techniques 9
B. Hart, P. Valduriez, and S. Danforth

JAS: A Parallel VLSI Architecture for Text Processing 16
O. Frieder, K.C. Lee, and V. Mak

Parallel Query Evaluation: A New Approach to Complex Object Processing 23
T. Haerder, H. Schoning, and A. Sikeler

Multiprocessor Transitive Closure Algorithms 30
R. Agrawal, and H.V. Jagadish

Exploiting Concurrency in a DBMS Implementation for Production Systems 37
L. Raschid, T. Sellis, and C. Lin

Checkpointing and Recovery in Distributed Database Systems 44
S. Son

Robust Transaction-Routing Strategies in Distributed Database Systems 51
Y. Lee, P. Yu, and A. Leff

Sharing the Load of Logic-Program Evaluation 58
O. Wolfson

SPECIAL ISSUE ON DATABASES FOR PARALLEL AND DISTRIBUTED SYSTEMS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE

IEEE Computer Society

Editor-in-Chief, Data Engineering
Dr. Won Kim
MCC
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3439

Associate Editors
Prof. Dina Bitton
Dept. of Electrical Engineering
and Computer Science
University of Illinois
Chicago, IL 60680
(312) 413-2296

Prof. Michael Carey
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
(608) 262-2252

Prof. Roger King
Department of Computer Science
campus box 430
University of Colorado
Boulder, CO 80309
(303) 492-7398

Prof. Z. Meral Ozsoyoglu
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106
(216) 368-2818

Dr. Sunil Sarin
Xerox Advanced Information Technology
4 Cambridge Center
Cambridge, MA 02142
(617) 492-8860

Chairperson, TC
Prof. Larry Kerschberg
Dept. of Information Systems and Systems Engineering
George Mason University
4400 University Drive
Fairfax, VA 22030
(703) 323-4354

Vice Chairperson, TC
Prof. Stefano Ceri
Dipartimento di Matematica
Universita' di Modena
Via Campi 213
41100 Modena, Italy

Secretary, TC
Prof. Don Potter
Dept. of Computer Science
University of Georgia
Athens, GA 30602
(404) 542-0361

Past Chairperson, TC
Prof. Sushil Jajodia
Dept. of Information Systems and Systems Engineering
George Mason University
4400 University Drive
Fairfax, VA 22030
(703) 764-6192

Distribution
Ms. Lori Rottenberg
IEEE Computer Society
1730 Massachusetts Ave.
Washington, D.C. 20036-1903
(202) 371-1012

The LOTUS Corporation has made a generous donation to partially offset the cost of printing and distributing four issues of the Data Engineering bulletin.

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

From the Issue Editors

Sushil Jajodia and Won Kim

On December 5-7, 1988, an IEEE-sponsored symposium named the International Symposium on Databases for Parallel and Distributed Systems was held in Austin, Texas. The symposium was an attempt to encourage interested professionals to focus their research on extending the technology developed thus far for homogeneous distributed databases into two major related directions: databases for parallel machines and heterogeneous distributed databases.

We selected seven papers from the symposium, and added two new papers to form this special issue on Databases for Parallel and Distributed Systems. The selection of papers in this issue was based on our decision to maximize the breadth of research topics to be introduced to the readers. We regret that we did not have enough space to include a paper on heterogeneous databases. The papers selected from the symposium had to be condensed because of page limits on our bulletin. The interested reader may obtain the proceedings of the symposium from IEEE for a broader perspective on this area.

Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience by Lorie, et. al., and *Parallelizing FAD Using Compile-Time Analysis Techniques* by Hart, et. al. describe approaches to exploit parallelism in databases in two major research efforts in parallel database machines. Frieder, et. al. describe a text-retrieval subsystem which uses a parallel VLSI string-search algorithm in *JAS: A Parallel VLSI Architecture for Text Processing.*

Parallel Query Evaluation: A New Approach to Complex Object Processing by Haerder, et. al., and *Multiprocessor Transitive Closure Algorithms* by Agrawal and Jagadish discuss issues in exploiting parallelism in operations involving complex data structures, namely, complex objects and transitive closures, respectively. *Exploiting Concurrency in a DBMS Implementation for Production Systems* by Raschid, et. al. describe parallelism in a database implementation of a production system. In *Checkpointing and Recovery in Distributed Database Systems*, Son outlines an approach to checkpointing in distributed databases and its adaptation to systems supporting long-duration transactions.

Robust Transaction-Routing Strategies in Distributed Database Systems by Lee, et. al., and *Sharing the Load of Logic-Program Evaluation* by Wolfson discuss approaches to load sharing in distributed and parallel systems.

The authors who contributed papers to this issue were very prompt in meeting our tight deadlines; they were all very professional. The printing and distribution of this issue has been made possible by a generous grant from the Office of Naval Research.

From the TC Chairman

Larry Kerschberg

I am pleased to welcome Don Potter as Secretary of our TC. Further, on behalf of our TC, I want to congratulate John Carlis, Richard L. Shuey and their team on the excellent organization and program of the Fifth International Conference on Data Engineering, held February 6-10, 1989 at the Los Angeles Airport Hilton and Towers. Over 315 people attended the conference.

Adding Intra-transaction Parallelism to an Existing DBMS: Early Experience

Raymond Lorie, Jean-Jacques Daudenarde
Gary Hallmark, James Stamos, Honesty Young

IBM Almaden Research Center, San Jose, CA, 95120-6099, USA

Abstract: A loosely-coupled, multiprocessor backend database machine is one way to construct a DBMS that supports parallelism within a transaction. This software architecture was the basis for adding intra-transaction parallelism to an existing DBMS. The result is a configuration-independent system that should adapt to a wide variety of hardware configurations, including uniprocessors, tightly-coupled multiprocessors, and loosely-coupled processors. This paper evaluates our software-driven methodology, presents the early lessons we learned from constructing an operational prototype, and outlines our future plans.

1 Introduction

A database machine based on multiple processors that share nothing is one way to provide the functionality of a conventional DBMS. Proponents of the loosely-coupled approach claim such an architecture can achieve scalability, provide good cost-performance, and maintain high availability [DGG*86,DHM86,Nec87,Tan87]. Current database machine activity, both in the lab and in the marketplace, is often driven by an emphasis on customized hardware or software. Although hardware and software customizations may improve performance, they reduce the portability and maintainability of the software, increase the cost of developing the system, and reduce the leverage one gets by tracking technology with off-the-shelf hardware and software.

We believe the costs of customization outweigh the performance benefits and have taken a software-driven approach to database machine design that focuses on intra-transaction parallelism. Our approach is to make minimal assumptions about the hardware; design the DBMS for a generic hardware configuration; support intra-transaction parallelism; and show how to map the system to particular hardware configurations. To test our beliefs we are prototyping a configuration-independent relational DBMS that is applicable to individual uniprocessors, to tightly-coupled multiprocessors, and to loosely-coupled multiprocessors. We intend to use simulation, modeling, and empirical measurements to evaluate this approach to database machine design.

The rest of the paper is structured as follows. Section 2 discusses parallelism in the context of a DBMS. Section 3 presents the goals of our project, which is called ARBRE, the Almaden Research Backend Relational Engine. Section 4 discusses the ARBRE design and shows how to apply it to different hardware configurations. Section 5 compares ARBRE to existing work, and Section 6 presents and evaluates the research methodology used in the project. Section 7 relates our early experiences and lessons from putting our methodology into practice. The last section describes the current status of the ARBRE prototype and outlines future plans. Throughout the paper we shall use the words *transaction* and *query* interchangeably.

2 Parallelism in a DBMS

Most currently available database systems have been implemented to run on a single processor and use multiprogramming to support inter-transaction parallelism: while some transactions are waiting

for I/O's, another transaction may execute CPU instructions. If the processor is a multiprocessor system with N engines, then N transactions may execute CPU instructions simultaneously. Most systems execute each transaction as a single thread and thus do not support intra-transaction parallelism. Intra-transaction parallelism could be achieved by having multiple threads run on behalf of the same transaction in order to reduce the response time for that transaction. On a uniprocessor, the threads not waiting for I/O share the one processor. On a multiprocessor, several processors could simultaneously execute the threads in parallel.

3 Goals

To gain insight into the costs and benefits of intra-transaction parallelism, we established four goals for the ARBRE project. First, we wanted to use parallel processing in a full-function, relational DBMS to reduce the response time for a single data-intensive SQL request. This includes exploiting parallel disk I/O and CPU-I/O overlap inside the request. Second, we wanted to be able to use additional processors to reduce the response time further for data-intensive operations. Third, we wanted to be able to use additional processors for horizontal growth to increase throughput. Fourth, we wanted to maintain an acceptable level of performance for on-line transaction processing (OLTP) environments.

To meet these goals we could first propose various hardware configurations with different numbers of processors, different speeds, and different communication topologies and primitives. For each configuration we could then design the most appropriate software organization. Such a methodology would be very time-consuming, especially if simulation and prototyping activities were needed to evaluate and validate the various possibilities.

We instead designed the DBMS software to be independent of the hardware configuration, hoping to demonstrate that the approach is viable, and that the performance can be almost as good as if the software had been customized for each hardware configuration—provided the communication scheme has enough bandwidth, low latency, and reasonable cost.

Our intention is to reuse most of the code of a single-site relational DBMS with no parallelism and to use several instances of such a DBMS to exploit intra-transaction parallelism. Each DBMS instance is responsible for a portion of the database. It may execute on a private processor, or it may be one of several instances sharing a large processor. We call the latter approach *virtualization*, because each instance of the DBMS is associated with a virtual processor. Code to support the distribution of functions must be added to the existing DBMS base under both approaches.

Since we are strictly interested in the parallelism issues, we are not trying to improve the performance of local operations performed on a single processor. We accept current systems as they are and assume that the hardware and software technology will improve with time.

4 System Overview

ARBRE is best viewed as being a multiprocessor backend database machine that is connected to one or more hosts. Connections to local area networks are also possible. The interface to the database machine is assumed to be at a sufficiently high level so that we can exploit parallelism within a query and minimize the communication delays incurred by separating the backend database machine from the host.

We discuss the ARBRE system in three steps. First, we present our assumptions about the processor and communication hardware. Then we focus on the software and execution strategy. Finally, we describe how to map ARBRE onto real hardware configurations.

4.1 A Generic Hardware Configuration

We assume, but do not require, that ARBRE runs on a loosely-coupled multiprocessor. The hardware of this multiprocessor consists of a fixed number of processing sites interconnected by a communication network that lets each pair of sites communicate. We make no further assumptions about the network. Each site has its own CPU, memory, channels, disks, and operating system. The sites run independently, share nothing, and communicate only by sending messages.

4.2 ARBRE Software and Execution Strategy

We assume every site runs the same software. Every site has one instance of the DBMS, and this instance alone manages the data kept at that site. The data is partitioned horizontally [RE78]: each table in the relational model is partitioned into subsets of rows, and each subset is stored at one site. The partitioning can be controlled by hashing or by key ranges. Key ranges can be determined by the user, or can be derived automatically by the system as in Gamma [DGG*86]. ARBRE supports both local and global indexes. A local index contains entries for tuples stored at the site containing the index. A global index is a binary relation associating a secondary key with a primary key. That binary relation is itself partitioned as is any base table.

Since data is not shared, a site executing a request that involves data managed by another site uses function shipping [CDY86] to manipulate remote data. A function that returns a small amount of data returns the result directly to the caller. For example, a function that fetches a unique tuple or computes an aggregate function falls into this category. Other functions may return large sets of tuples in the form of tuple streams. A tuple stream is a first-in-first-out queue whose head and tail may reside at different sites.

The host runs the application program, which contains SQL calls to the database. Each call to the database causes an asynchronous request in the host, so it is important to minimize the interaction between the host and the backend database machine. Fortunately, relational queries are at a high level and tend to return all and only the information requested. If host-backend interaction is a problem, one simple way to reduce it is to have the host batch requests inside the same transaction as long as no processing is done between requests. A more general approach is to have the host batch requests from different transactions if the resulting increase in response time is tolerable. Raising the level of the query language can also reduce host-backend interaction. For example, the query language could express complex object fetch and recursion. The ultimate step is to have the backend do general computation, and we have chosen this approach in our prototype to give us maximum flexibility.

Before being executed the application program and the SQL statements it contains must be compiled. The query compiler, which converts an SQL statement into a set of one or more compiled query fragments, uses the database machine for interrogating the catalogs and storing the query fragments.[1] Some compiled query fragments are executed at one site, and other fragments are sent to multiple sites and executed in parallel. One fragment is called *coordinator* fragment, and it is responsible for coordinating the execution of the other fragments, which are called *subordinate* fragments.

Each compiled query fragment is executed as a separate lightweight thread. Threads at the same site or at different sites communicate by sending messages and by using tuple streams. When the host sends a request to some site in the database machine, this site fetches the corresponding coordinator fragment and executes it as a thread. This thread becomes the coordinator for the transaction and receives all further calls the host sends on behalf of this transaction.

[1] The compiler can reside in the host or in the database machine; there are arguments in favor of both approaches, but the final decision is irrelevant to the paper.

The coordinator uses function shipping to execute subordinate fragments. Each subordinate fragment executes as a separate thread and generally involves one base table. To decide the site(s) that execute a subordinate fragment when that fragment involves a base table, the coordinator consults the hashing function or key-range table that indicates how the table is horizontally partitioned.

How the results of an SQL statement are returned to the host depends on the expected size of the results. If the amount of data produced by executing the query is small, the results are returned to the coordinator which then assembles them and forwards them to the host. On the contrary, if the amount of returned data is large, and if the data does not need to be combined with other data in order to be returned to the host, we send it directly from each subordinate to the host without involving the coordinator.

A dataflow approach, similar to the one used in Gamma and proposed in [BD82], controls the simultaneous work of many query fragments on behalf of the same data-intensive transaction. Fragments may collectively produce a stream, send their substreams to others, receive the substreams sent by others, and consume them. The communication software uses message buffering and a windowing mechanism to prevent stream producers from flooding stream consumers.

When fragments must exchange large amounts of data, the communication may become a bottleneck. One way to reduce communication is by a judicious choice of algorithms. For example, a hash-based join works well in a distributed environment, but it requires sending practically both tables on the network. We are also investigating other ideas such as the use of semi-join, the possibility of completing a join in the host, and the use of algorithms that tolerate skewed data access patterns.

4.3 Mapping Sites to Processors

Most database machine research projects and commercial products use the simplest mapping from sites to processors: these systems devote an entire physical processor to each site. This approach is also applicable to ARBRE. In this approach, each site has an operating system that supports a single instance of the DBMS executing in its own address space. Each DBMS instance supports multiprogramming for inter-transaction parallelism, but it has no intra-transaction parallelism. Intersite communication corresponds to interprocessor communication.

Alternatively, one can map several sites to a single processor. The processor then contains as many instances of the DBMS as there are sites, and all the instances share a single copy of the code. The same communication interface is used, but the implementation exploits fast memory-to-memory transfer, rather than actual communication via a network, among sites that are mapped to a single processor.

4.4 Other Issues

To keep our task manageable, we postponed detailed consideration of several important issues. In particular, we examined the following issues only superficially: automatic query planning; catalog management; management and replication of key-range tables; data replication and reorganization; operational management of a large number of sites; and fault tolerance.

5 Related Work

Several projects, both in universities and industrial labs, are concerned with using multiple processors to improve performance of relational systems. Among the systems that are most comparable to ARBRE are Gamma [DGG*86], Tandem's NonStop SQL[2] product [Tan87], and the

[2] NonStop SQL is a trademark of Tandem Computers Incorporated.

DBC/1012[3]machine built by Teradata [Nec87]. All three systems use loosely-coupled general purpose processors, employ customized operating systems, and support one or more kinds of horizontal partitioning and some degree of intra-transaction parallelism. The unusual features of these systems are listed below. Gamma has diskless processors to add processing power for operations such as joins. Tandem's NonStop SQL is a stand-alone computing system that executes applications and supports end users. There is no support, however, for intra-transaction parallelism except for FastSort, which uses several processors for a single sort. Teradata's DBC/1012 has a proprietary Ynet[3], which implements reliable broadcast and tournament merge in hardware. The DBC/1012 exhibits non-uniformity of processors: each processor module has specialized software and controllers and is connected to different kinds of peripherals.

The ARBRE project differs clearly from these other systems on two accounts. First, ARBRE is the only project we are aware of that is studying multiple mappings of logical sites onto real processors. Second, unlike other multiprocessor backend database machines, ARBRE tries to increase the level of parallelism in the return of data to the host by avoiding the coordinator whenever possible. Another feature of ARBRE is that no site is distinguished by having special hardware or special software, at least at execution time.

6 Methodology

We chose a research methodology to support our main objective, which is to draw some conclusions, as quickly as possible, on the architecture of a configuration-independent parallel DBMS, its feasibility, and its expected performance. As a result our methodology was designed around three principles: (1) build an operational prototype by using sturdy components for the hardware, operating system, and access method instead of constructing our own; (2) concentrate on the run-time environment, postponing any development of the query compiler; and (3) complement the prototype evaluation with simulation and modeling. The rest of this section discusses each principle in turn.

We reused existing components rather than construct new specialized ones because the incremental benefits would not justify the cost of construction. We used a general purpose, existing operating system (MVS) that supports multiple processes in a single address space. We also used the low-level data manager and transaction manager in System R [B*81], an experimental relational database management system. In addition, we used a prototype high-performance, interprocessor communication subsystem (Spider) implemented by our colleague Kent Treiber. For hardware we used brute force, relying on a channel-to-channel communication switch interconnecting multiple IBM 4381 machines, which are midrange, System/370 mainframes.

We postponed the development of a query compiler and concentrated on query execution strategies that exploit parallelism without causing communication bottlenecks. We believe the development of a query compiler should be relatively straightforward once we have determined a repertoire of good execution strategies. To support our investigation of execution strategies, we implemented a toolkit of relevant abstractions. These abstractions fall into 4 categories: a generalization of function shipping, virtual circuits and datagrams, single-table-at-a-time database access, and primitives dealing with the horizontal partitioning of data. We used the same programming language (C++) to implement these abstractions as we do to write compiled query fragments. This will make it easy to migrate useful algorithms from query execution strategies into the database machine interface.

An operational prototype will give us enough information to drive simulations and validate the results. First, we will instrument and measure a working environment. The information obtained will then be submitted to a simulator to predict how the same workload will behave on different

[3]DBC/1012 and Ynet are trademarks of Teradata Corporation.

configurations. To obtain meaningful results we plan to record events produced by executing real applications as well as those produced by executing synthetic workloads. From the event traces we will determine data and processing skews and produce probability distributions that concisely describe these skews. The probability distributions, and not the raw event traces, will drive the simulations. Given our flexibility in mapping logical sites onto multiple configurations, we anticipate validating the simulation results on multiple physical configurations that are easy to produce.

Configuration independence has improved our programming and debugging productivity because we do not work exclusively with the target hardware, operating system, access method, and communication system. Most of the time we use an IBM RT PC running AIX, which is IBM's implementation of the UNIX operating system.[4] We use a single address space on the RT PC and a simple, main-memory based access method to emulate a multiple site system. Almost all software is developed and thoroughly debugged in this user-friendly environment before it is run on a target system.

The following are drawbacks to our methodology: (1) The simulations are based on probability distributions rather than actual data dependencies. (2) Simulation runs may be time consuming. For this reason we plan to use modeling which, when validated with a more detailed simulation, may be used to extrapolate our results to other configurations in much less time. (3) Our methodology does not consider configuration-specific optimizations; these should be identified and studied independently. Nevertheless, we believe that these drawbacks are tolerable and that our methodology is appropriate for gaining valuable insight into DBMS parallelism in a short time period.

7 Early Lessons

Over two years of preliminary research, design, and prototyping have taught us three things: good building blocks are indispensable, language design is hard, and simulation has its limitations.

One lesson we learned is not to start from scratch even though software simplification because of specialization is often highlighted as an important advantage of backend database machines. It is less fruitful to spend time rewriting mature, highly-tuned code than it is to implement intra-transaction parallelism.

If you don't start from scratch, you will most likely modify existing code, in which case it is important to have modifiable software components. For example, the transaction manager we used already had hooks for two-phase commit and distributed recovery, and adding a two-phase commit protocol was straightforward. We have added message queues and timers to the DBMS thread scheduler, and if we implement global deadlock detection we must be able to extract the transaction waits-for graph from the lock manager.

A second lesson we learned is that language design is hard. We initially tried to design a custom language for coding the query fragments, but discovered that language design without sufficient experience in the domain of discourse is too slow and required too many iterations. Instead we are using an existing programming language (C++) and have built a toolkit of useful abstractions. The toolkit lets us experiment with algorithms without designing and freezing a language and its interpreter. As we gain experience we will progressively develop our toolkit, using more predefined constructs and less ad-hoc programming in the fragments. Eventually, a "language" will emerge that succinctly expresses good execution strategies for query fragments. This language will be the target of the query optimizer and compiler.

A third lesson we learned is that some interesting issues may be difficult to study in simulation. We initially thought we could use the raw event traces in simulations of different hardware configurations, but simulating the exact data dependencies would make the simulations too expensive to

[4]RT PC and AIX are trademarks of the IBM Corporation. UNIX is a trademark of the AT&T Corporation.

run. Instead, data dependencies and other nonuniformities will be approximated with probability distributions.

8 Status and Plans

The prototype is operational on three interconnected dyadic-processor 4381 systems. Although we have begun measuring the system for complex queries involving sorts and joins, the results are too preliminary to be reported here. Suffice it to say that for a single data-intensive transaction we have illustrated all aspects of parallelism. We used multiple sites on a single 4381 processor (i.e., virtualization) to exploit I/O parallelism; we used multiple sites on tightly-coupled dyadic processors to exploit CPU parallelism; and finally we used multiple sites on separate 4381 systems to exploit loose coupling.

The prototype will be extremely useful as we begin to study issues that are inherent to DBMS parallelism, including: the need for sophisticated parallel algorithms; load balancing and process scheduling; and communication problems, such as convoys, network congestion, and deadlock. We are also beginning to investigate query optimization and support for high rates of simple transactions. Skewed data access patterns and a larger number of smaller processors will exacerbate some of the above problems and may demand innovative solutions.

Our approach to DBMS parallelism, which distinguishes logical sites from physical processors, is a promising approach that can adapt to different hardware configurations, different cost-performance trade-offs, and different levels of required performance. We envision a single code base that is applicable to a cluster of high-end mainframes as well as to a network of powerful microprocessors.

References

[B*81] M. W. Blasgen et al. System R: An architectural overview. *IBM Systems Journal*, 20(1):41–62, January 1981.

[BD82] Haran Boral and David J. DeWitt. Applying data flow techniques to data base machines. *IEEE Computer*, 15(8):57–63, August 1982.

[CDY86] D. W. Cornell, D. M. Dias, and P. S. Yu. On multisystem coupling through function request shipping. *IEEE Transactions on Software Engineering*, SE-12(10):1006–1017, October 1986.

[DGG*86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma–a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, August 1986.

[DHM86] Steven A. Demurjian, David K. Hsiao, and Jai Menon. A multi-backend database system for performance gains, capacity growth and hardware upgrade. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 542–554, 1986.

[Nec87] Philip M. Neches. The anatomy of a data base computer system. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 102–104, 1987.

[RE78] D. Ries and R. Epstein. *Evaluation of Distribution Criteria for Distributed Database Systems.* UCB/ERL Technical Report M78/22, University of California—Berkeley, May 1978.

[Tan87] The Tandem Database Group. Non-stop SQL, a distributed, high-performance, high availability implementation of SQL. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, September 28–30 1987.

PARALLELIZING FAD
USING COMPILE-TIME ANALYSIS TECHNIQUES

Brian Hart, Patrick Valduriez, Scott Danforth

Advanced Computer Architecture Program
Microelectronics and Computer Technology Corp.
Austin, Texas 78759

ABSTRACT

FAD is a database programming language with much higher expressive power than a query language. FAD programs are to be executed efficiently on Bubba, a parallel computer system designed for data-intensive applications. Therefore, parallelism inherent in a FAD program must be automatically extracted. Because of the expressive power of FAD, traditional distributed query-optimization techniques are not sufficient. In this paper, we present a general solution to the parallelization of FAD programs based on compile-time analysis techniques.

1. Introduction

FAD [Ban87, Dan89] is a strongly typed functional-programming language designed for manipulating transient and persistent data within Bubba, a highly parallel database system [Bor88] developed at MCC. As a database programming language, FAD reduces the "impedance mismatch" of the traditional approach that embeds a query language (e.g., SQL) into a programming language (e.g., C). The FAD data model allows arbitrarily complex combinations of data structures based on atomic values, tuples, sets, disjuncts and objects. In particular, referential object sharing [Kho87] is fully supported. FAD incorporates a blending of proven concepts from the worlds of functional programming and relational databases. The result is a strongly typed language with clean semantics whose implementation on Bubba directly benefits from progress in both parallel processing and relational-database technology. To increase performance, FAD is compiled into low-level code to be executed on the parallel database system.

The FAD compiler extracts parallelism inherent in ("parallelizes") a FAD program by transforming it into a set of communicating FAD subprograms, called components, which can be executed in a parallel (SIMD or MIMD) fashion. The problems to be addressed are to determine the most efficient division of a program into components and the most efficient location of their execution. Traditional distributed query-optimization techniques serve as the basis for the compiler but these must be extended considerably due to expressiveness of FAD. In particular, the use of object identity can create difficult aliasing problems. Also, the presence of a number of powerful programming constructs, such as iteration and conditionals, adds complexity. In this paper, we give a solution to these problems based on compile-time analysis techniques.

In this paper, we provide a short overview of the compile time analysis techniques employed for parallelizing FAD. After an introduction of Bubba and of the most salient features of the FAD programming language, we focus on FAD parallelization which plays a central role in compiling FAD for execution on Bubba.

2. Bubba

Bubba is a parallel computer system for data-intensive applications. Bubba is intended as a replacement for mainframe systems providing scalable, continuous, high-performance per-dollar access to large amounts of shared data for a large number of concurrent application types.

Three constraints shape the problems to be addressed by Bubba: concurrent multiple workload types on shared data, large databases, and high-availability requirements. Multiple workloads, particularly transaction processing and knowledge-based searches for complex patterns, imply the need to support a powerful programming language through a rich environment for program management and execution. Large databases imply minimization of data movement and thus program execution where the data lives. High-availability requirements imply the need for redundancy and real-time fault recovery mechanisms.

[Bor88] gives the rationale for picking the army of ants approach for the Bubba architecture. The simplified hardware architecture is illustrated in Figure 1. Each node, called Intelligent Repository (IR), includes one or more microprocessors, a local main memory (RAM) and a disk unit on which resides a local database. Diskless nodes are also used to interface Bubba with other machines. An IR is believed to be "small" for two reasons: 1) they provide cheap units of expandibility and conversely, the loss of an IR is likely to have little impact on overall performance, and 2) knowledge that IRs are "hefty" will lead to attempts to exploit locality through clever physical-database design, thereby limiting the class of applications for which Bubba would be useful.

The only shared resource is the interconnect which provides a message-passing service. Each IR runs a copy of a distributed operating system which, among other things, provides low-level support for task management, communication and database functions. In particular, local object identity is supported within an IR but global object identity is not supported.

To favor parallel computation on data, the database consists of relations which are *declustered* across IRs. Declustering is a placement strategy which horizontally partitions and distributes each relation across a number of IRs. This number is a function of the size and access frequency of the relation [Cop88]. The basic execution strategy of Bubba is to execute programs where the data is [Kho88] to avoid moving data. Therefore, the degree of parallelism in an individual program is determined by the number of nodes the data referenced by the program occupies.

High availability is provided through the support of two on-line copies of all data on the IRs as well as a third copy on checkpoint-and-log IRs.

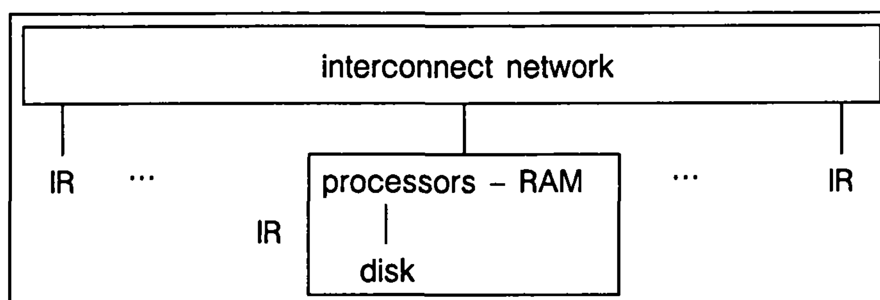


Figure 1: Simplified Hardware Organization of Bubba

3. FAD

The central ideas of FAD are relatively few: types, data, actions, and functions (action abstractions). The FAD type system corresponds closely to that of an order-sorted algebra [Dan88b]. A type in FAD provides a domain of data, as well as functions for creating and performing actions on elements of that domain. FAD *data* are distinguished between values and objects: only objects can be shared and updated [Kho87]. The structure of FAD data can be simple or complex (using constructors such as tuple, set and disjunct). The term *action* is used to indicate a computation that accesses data, returns data, and may change existing objects. Application of a function to its arguments denotes an action. Abstraction in FAD allows the creation of user-defined first-order functions, actions that are parameterized with respect to the data on which they act. In addition, FAD provides a fixed set of higher order functions, called *action constructors,* for writing programs. These are operators that construct aggregate actions from actions, data, and functions. A number of FAD action constructors are provided for operating on sets in a parallel fashion. The most important set-oriented action constructor is the *filter* statement, which applies a function in parallel to each element of the Cartesian product of a number of sets to produce a new set, thus providing a generalized select-project-join (SPJ) capability. Other action constructors are provided for set manipulation (group, pump), variable definition (let), conditional (if-then-else), iteration (whiledo), and control (do-end, begin-end, abort).

FAD essentially provides Bubba users with a centralized execution model. Parallel FAD (or PFAD) [Har88] is an enhancement to the FAD language that captures aspects of the parallel execution model of Bubba not visible in FAD. PFAD is an abstraction of Bubba that supplements FAD with the concepts of *component* and inter-component *communication primitives*. PFAD is used as an intermediate language by the FAD compiler to reflect decisions concerning the locations at which actions will be executed, and the manner in which actions at different locations communicate. The similarity between FAD and PFAD eliminates the difficulty of translation to a very different language. A FAD program is partitioned into one or more components and because the data is declustered, each one may be executed by the parallel system at one or more IRs. A component may produce transient data which are used as input to other components. Those transient data establish dataflow dependencies between components.

4. The FAD Compiler

The FAD compiler [Val89] transforms a FAD program into a low-level object program that may be executed on Bubba. Designing a FAD compiler is a challenging research project that combines compilation, parallel processing and distributed query-optimization issues. A FAD program expresses a computation on conceptual objects based on a centralized execution model and may use constructs such as set operators that favor parallel computation. The compiled program accesses physical objects (actually stored in the database) based on a parallel execution model with explicit intra-program communication.

The compiler performs static type checking, correct transformation and optimization of a FAD program. To achieve those functions, the compiler has precise and concise knowledge of the Bubba database which includes schema (typing), statistics, cost functions and data placement information. Utilization of this information leads to efficient low-level programs for execution on Bubba. Static type checking avoids expensive run-time type checks while helping the FAD programmer to write correct programs. The compiler will infer transient types when appropriate. The major characteristic of the compiler is to make a number of crucial optimization decisions. The compiler optimizes a FAD program with respect to communication, disk

access, main-memory utilization and CPU costs [Val88]. Optimization may be biased towards minimizing response time or total work. The latter is more suitable to maximize throughput.

The compiler comprises four subsequent phases: static type checking (type inferencing and type assignment), optimization, parallelization, and object-code generation. The optimization decisions are conveyed using FAD (e.g., filter ordering) and annotations to be used by the next compilation phases.

The parallelizing approach is to use a dataflow analysis to flag potential correctness problems that may arise when parallelizing a FAD program, followed by the heuristic application of several parallelization techniques that provide a tradeoff space between conservative (and always correct) and speculative (and sometimes correct) options. Selection of a parallelization technique in a given instance is influenced by several factors: 1) correctness, using the dataflow-analysis results; 2) performance, using input from the optimizer when available; and 3) performance, using a heuristic evaluation of a search tree of possibilities when input from the optimizer is unavailable or irrelevant because of correctness constraints

5. Analysis Techniques for Parallelizing FAD

The parallelizer (see [Har88] for more details) transforms a FAD program into an equivalent PFAD program. It does most of its work incrementally with a local viewpoint. Because of the complexity of the input and output languages and aggressiveness of the techniques used, and similar to the problem that this local viewpoint may produce locally optimal decisions, but not globally optimal, there are some correctness issues discussed below such that this local viewpoint may produce only locally correct decisions. So the parallelizer needs to pursue alternate translations.

The parallelizer explores a search tree whose nodes are possible translations (PFAD programs). The root of the search tree is a trivial translation which is simple and guaranteed correct, but only minimally parallel. It generates a *centralized PFAD program* which retrieves all needed persistent data at the IRs it is stored on, sends it to a single *central IR*, executes all the operations from the FAD program at that central IR, sends all persistent data updates back to the IRs it is stored on, and performs the updates there. Successor nodes are incremental transformations from their parents, generated using a set of *strategies* (they generate the choices in the search tree). The optimizer uses heuristics to explore the search tree. Because of the problems discussed below, the choices must also be checked; the parallelizer uses a set of *analyses* to check the choices.

The strategies generally involve moving another increment through the program and transforming it so that some (more) operations will be executed in parallel at the IRs which hold some persistent data, rather than at the central IR. Then any data the operations need, but is not already at those IRs, must be sent to those IRs. For example, consider a FAD program for the relational algebra expression: $T \bowtie S \bowtie \sigma R$. The parallelizer translates this to the centralized PFAD program in which relations R, S, and T are retrieved from the IRs they are stored on and sent to a central IR, where the select and the two joins are performed. (There are no updates to send back.) Then the parallelizer considers executing the select at IRs other than the central IR. In general, the set of IRs holding data from relation R is a good choice. So now there are two possible translations. The parallelizer proceeds as illustrated in Figure 2; the select at R, join1 at S, and join2 at T is the resulting translation.

But there are several possible problems. One is to determine whether the operations involved make sense to be run in parallel, with the data that they will be getting at run time. Others are to determine what data the operations need, to parallelize operations that are sequentialized in the input FAD program, to

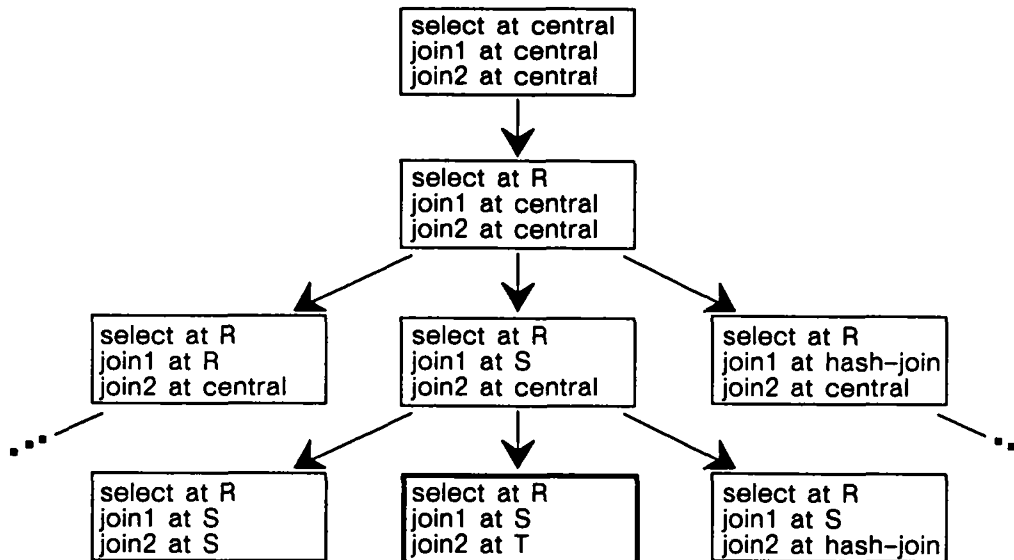


Figure 2: Search Tree Example

handle updates to aliases and non-local objects, and to emulate global object identity using only local object identity.

5.1 Abstract Evaluation

The analyses that check the problems discussed above are based on *abstract evaluation* (called symbolic or abstract interpretation elsewhere), a tool for reasoning about a program at compile-time. It abstracts the domain of data that the programs execute on, and then evaluates the program using that abstraction. The particular abstraction is based upon the particular program properties we wish to reason about. The results of the abstract evaluation tell us something about the program, specifically whether the transformations are correct and how they might be corrected if not.

The analyses are *data-distribution (DD) analysis* and *object-sharing (OS) analysis*. A data-distribution analysis checks whether operations make sense to be run in parallel, with the data that they will be getting at run time. An object-sharing analysis checks the others.

5.2 Data-Distribution Analysis

The motivation for a data-distribution analysis is best illustrated with the following FAD program:

```
prog()
 let x = f()
 in  if p(x) then g(x) else h(x)
```

If the "if-then-else" executes at several IRs, then "x" might have different values at different IRs, meaning "p(x)" might have different values at different IRs, meaning that "g(x)" might be executed at some IRs and "h(x)" might be executed at other IRs. While this might give us equivalent (to the non-parallel program) results, we cannot say for sure when it will and when it will not. So the data items used by an if-part must be the same or "wholly" present at each IR executing the if-part.

An abstract evaluation of *data-distribution (DD)* determines this. Before we describe DD, we define two terms. With respect to a PFAD data item, *placed* means that (1) if the data item is an atom, then the

atom's value can be considered with respect to the database relation's declustering function for the IR it is on; (2) if the data item is a tuple, then the tuple contains an attribute that is an atom and is placed; or (3) if the data item is a set, then each data item in the set is an atom or tuple and is placed. With respect to a PFAD data item, *placed correctly* means that the data item is *placed* and the atom's value involved agrees with the database relation's declustering function for the IR it is on. There are six DD values.

W *whole item.* The whole data item is present at each IR, and either it is placed correctly or there is only one IR containing the data item. It does not contain duplicates.

W_W *whole item, wrong place.* The whole data item is present at each IR, but it is not placed correctly. It does not contain duplicates.

D *distributed item.* The data item is a set, is placed correctly, and is fragmented over more than one IR. It does not contain duplicates.

D_W *distributed item, wrong place.* The data item is a set, is placed but not correctly, and is fragmented over more than one IR. It does not contain duplicates.

D_{WD} *distributed item, wrong place, with duplicates.* The data item is a set, is placed but not correctly and is fragmented over more than one IR. It may contain duplicates.

O *other.* This is anything else and means useless data.

The operations on DD correspond to FAD operations. As an example, consider the "if-then-else" in the above FAD program. If the DD of "p", "g", and "h" were all "W", then the DD of the "if-then-else" would be "W". If the DD of "p", "g", and "h" were "W", "D", and "D_W", respectively, then the DD of the "if-then-else" would be "D_W".

5.3 Object-Sharing Analysis

When a data item is updated, and that data item aliases another data item (possibly at another IR), the update must be performed on the proper IR. This is done by either placing the update so that it is executed at the IR where the data is, or by placing the update somewhere else, but also placing a compensating update so that it is executed at the IR where the data is.

An abstract evaluation of object sharing is used to determine this. Let P be the set of paths to objects in FAD. For example, "*db.D1*" is the path to the database relation "db.D1", "*db.D1@1*" is the path to the element of "db.D1" with the key "1", and "*db.D1@1.wage*" is the path to the wage of the element of "db.D1" with the key "1". The paths form a partial order. For example, "*db.D1*" includes the object "*db.D1@1*" which includes the object "*db.D1@1.wage*". Unnamed objects are not a problem because they cannot be aliased.

The operations on paths correspond to the FAD operations. For example, the OS of the union of two sets of objects is the union of the objects in the two sets. The OS of the difference of two sets of objects is the set of objects in the left argument.

Aliases are also kept track of. For example, if the variable "x" aliased the object "*db.D1@1*", then the OS for "x" is "*x=db.D1@1*". This object-sharing information determines the data needed by the operations and the operations which may be parallelized. Further, when an object is updated, it and all the objects it aliases are marked as updated with a subscript "u". For example, if the variable "x" above was

updated, then its OS would be "x_u=*db.D1@1*$_u$". When an object is sent to another IR, it and all the objects it aliases are marked as copied with a subscript "*c*" and a unique *copy number*. For example, if the variable "x" above was sent to another IR, then its OS would be "x_{c16}=*db.D1@1*$_{c16}$" (the "16" is an example). If the variable "x" above was updated and sent to another IR, then its OS would be "x_{uc16}=*db.D1@1*$_{uc16}$". If the variable "x" above was sent to another IR and updated, then its OS would be "x_{c16u}=*db.D1@1*$_{c16u}$", which determines that an aliased or non-local object is updated. We have updated an object in the "wrong" place. This can be corrected by either placing the update so that it is executed at the IR where the data is, or by placing the update somewhere else, but also placing a compensating update so that it is executed at the IR where the data is.

Another problem is "*db.D1@1*$_{c16}$=*db.D1@1*$_{c17}$", which means that we have the same objects aliased twice, but by different copies of the same object, because global object identity is not emulated faithfully. At run-time, these will be represented as different objects.

6. Status

The FAD compiler, including the parallelizer has been operational since November 1988, and work continues on it. Bubba is being implemented on a 40-node Flexible Computers multiprocessor.

References

[Ban87] F. Bancilhon, T. Briggs, S. Khoshafian, P. Valduriez, "FAD, a Simple and Powerful Database Language", Int. Conf. on VLDB, Brighton, England, September 1987

[Bor88] H. Boral, "Parallelism in Bubba", Int. Symp. on Databases in Parallel and Distributed Systems, Austin, Texas, December 1988.

[Cop88] G. Copeland, B. Alexander, E. Boughter, T. Keller, "Data Placement in Bubba", ACM SIGMOD Int. Conf., Chicago, Illinois, May 1988.

[Dan89] S. Danforth, S. Khoshafian, P. Valduriez, "FAD, a Database Programming Language, Rev.3", MCC Technical Report DB-151-85, Rev.3, January 1989.

[Har88] B. Hart, S. Danforth, P. Valduriez, "Parallelizing a Database Programming Language", Int. Symp. on Databases in Parallel and Distributed Systems, Austin, Texas, December 1988.

[Kho87] S. Khoshafian, P. Valduriez, "Persistence, Sharing and Object Orientation: a database perspective", Int. Workshop on Database Programming Languages, Roscoff, France, September 1987.

[Kho88] S. Khoshafian, P. Valduriez, "Parallel Execution Strategies for Declustered Databases", *Database Machines and Knowledge Base Machines*, Kitsuregawa and Tanaka Ed., Kluwer Academic Publishers, Boston, 1988.

[Val88] P. Valduriez, S. Danforth, "Query Optimization in FAD, a Database Programming Language", MCC Technical Report ACA-ST-316-88, Austin, Texas, September 1988.

[Val89] P. Valduriez, S. Danforth, T. Briggs, B. Hart, M. Cochinwala "Compiling FAD, a Database Programming Language", MCC Technical Report ACA-ST-019-89, Austin, Texas, February 1989.

JAS : A PARALLEL VLSI ARCHITECTURE FOR TEXT PROCESSING

O. Frieder, K. C. Lee, and V. Mak
Bell Communications Research
445 South Street
Morristown, New Jersey 07960-1910

Abstract. A novel, high performance subsystem for information retrieval called JAS is introduced. The complexity of each JAS unit is independent of the complexity of a query. JAS uses a novel, parallel, VLSI string search algorithm to achieve its high throughput. A set of macro-instructions are used for efficient query processing. The simulation results demonstrate that a gigabyte per second search speed is achievable with existing technology.

1. Introduction

Many recent research efforts have focussed on the parallel processing of relational (formatted) data via the use of parallel multiprocessor technology [Bar88, Dew86, Goo81, Got83, Hil86, Kit84]. In [Bar88], the use of dynamic data redistribution algorithms on a hypercube multicomputer is described. The exploitation of a ring interconnection is discussed in [Dew86, Kit84]; modified tree architectures are proposed in [Goo81, Hil86]; and a multistage interconnection network as a means of supporting efficient database processing is described in [Got83]. However, except for a few efforts [Pog87, Sta86], relatively little attention has focussed on the parallel processing of unformatted data.

For unformatted data, most of the previous efforts have relied on low-level hardware search support (associative memory). Even the software approaches on parallel machines [Pog87, Sta86] have relied on algorithms best suited for low level enhancements. In [Pog87], parallel signature comparisons were studied on the ICL Distributed Array Processor (DAP) architecture, and [Sta86] discusses the utilization of the Connection Machine for parallel searching. Critical reviews of [Sta86] are found in [Sto87] and [Sal88].

Associative-memory search architectures are based on VLSI technology. VLSI technology supports the implementation of highly parallel architectures within a single silicon chip. In the past, hardware costs exceeded software development costs. Thus, software indexing approaches were used to reduce the search time. Currently, since the design and maintenance of software systems is more costly[1] than repetitively structured hardware components, using VLSI technology to implement an efficient associative storage system seems advantageous. Furthermore, besides the cost differential, VLSI searching reduces the storage overhead associated with indexing (300 percent if word level indexing is used [Has81]) and can reduce the time required to complete the search.

Two critical problems associated with supporting efficient searching are the *I/O bottleneck* and the *processor incompatibility*. The I/O bottleneck is the inability of the storage subsystem to supply the CPU with query-relevant data at an aggregate rate which is comparable to the aggregate processing rate of the CPU. Processor incompatibility is the inconsistency of the instruction set of a general-purpose CPU, e.g., add, subtract, shift, etc.; and the needed search primitives, e.g., compare two strings masking the second, sixth, and eleventh character. To remedy these problems, special-purpose VLSI processing elements called data filters have been proposed [Cur83, Has83, Hol83, Pra86, Tak87]. The search time is further reduced by combining multiple data filters on multiple data streams to form a virtual associative storage system. Thus, the advantages of an associative memory can be exploited without incurring the associated costs.

With the continued growth of unformatted, textual databases[2], a large virtual associative memory should be based on unconventional I/O subsystems and very high filtering rates to continue supporting adequate response times. Currently proposed filtering rates have hovered at roughly 20 MBytes per second. We propose an I/O subsystem called JAS, with filtering rates comparable to the next-generation optical disks and/or silicon memory systems. JAS consists of a general-purpose microprocessor which issues the search and control instructions that the multiple VLSI data filters execute. Only the portion of data that is relevant to the query, e.g., related documents in a text database environment, are forwarded to the microprocessor. In this paper, we discuss the design and usage of a parallel VLSI text data filter to construct a subsystem for very large text database systems.

The remainder of this paper is organized as follows. Section 2 briefly describes the JAS architecture. A description of the Data Parallel Pattern Matching (DPPM) algorithm, which forms the basis for the design of our

---

[1] We measure cost in terms of both finances and human effort.

[2] The legal database Lexis is estimated at over 125 GBytes of information [Sta86]. It is reported that information retrieval databases have been growing at a rate of 250,000 documents per year [Hol79].

data filter, is presented in Section 3. A performance study of the JAS system is presented in Section 4. Section 5 concludes this paper with a discussion of the JAS system.

2. JAS System Architecture

Customized VLSI filters are used to perform high-speed substring-search operations. The novel string-search algorithm used in JAS improves the search speed by an order of magnitude as compared to prior CMOS filter technology (e.g., [Tak87]). We decouple the substring-search operation from high level predicate evaluation and query resolution. Thus, complex queries can be evaluated but do not nullify the simplicity and efficiency of the search hardware.

A JAS system is comprised of a single "master" Processing Element (PE) controlling a set of gigabyte per second "slave" Substring Search Processors (SSPs). While previously proposed text filters [Cur83, Has83, Tak87] evaluate complex queries via integrated custom hardware, in JAS the predicate evaluation and query resolution is decoupled from the primitive substring-search operations. A complex query is decomposed by the PE into basic search primitives. In [Cur83, Has83], complicated circuitry is required to support state transition logic and partial results communication for cascaded predicate evaluation. In JAS, since the complexity of an individual query is retained at the PE level, and only a substring-match operation is computed at the SSPs, only simple comparator circuitry is required.

Figure 1 demonstrates the processing of a query within a JAS system. Each PE forwards a sequence of patterns to its associated SSPs. Each SSP compares the data against a given pattern: one pattern per SSP; multiple SSPs per PE. Whenever a match is detected at a given SSP, the document Match ID (MID) consisting of the address of the match (Addr), the document identifier (Doc_ID), and the query id (Que_ID) is forwarded to the PE. Once the MID reaches the PE, the actual information-retrieval instruction which was decomposed to generate the match is evaluated, and if relevant, the results are forwarded to the host.

Table I presents the match-based JAS PE macro-instruction set and the match sequence which implements each of the JAS instructions. The JAS instruction set is based on the text-retrieval instruction set presented in [Hol83]. In the table, the leftmost column presents the actual instruction. A semantic description of the instruction is provided in italics followed by the control structure implementing the instruction. As seen in Table I, the entire text-retrieval instruction set, including the variable-length separation match instruction, "**A *.n.* B**", which can not be efficiently implemented directly via FSA, cellular, or CAM&SLC implementations, can be implemented via a coordinated sequence of substring-search primitives. Several clarifications are required. It is assumed that the evaluation of each sequence of subinstructions terminates upon encountering an end-of-document indicator (END_OF_DOC). The match(set of strings) instruction returns true whenever a match is detected. False is returned once detecting an END_OF_DOC. Type(match(strings)) returns the pattern type of the match. Address(match(A)) returns the starting address of the match. If no match is encountered before END_OF_DOC, the function returns default. Note that the match instructions "hang" until a match or END_OF_DOC is encountered. The pseudocode provided is for explanation purposes. For better performance, many optimization are possible. In all instructions, pattern overlap is forbidden. For queries comprised of multiple text-retrieval instructions, several sequences of substring search primitives must be employed.

The internal JAS control structure, PE to SSPs, is similar to that of the Query Resolver to Term Comparator of the PFSA system [Has83]. In the JAS system, however, the PE is responsible for the actual evaluation of information-retrieval instruction (a sequence of match primitives - see table I); whereas in the PFSA system, the Query Resolver is involved in the evaluation of the overall query (a sequence of information retrieval instructions).

3. Substring Search Processors

In examining prior work [Cur83, Fos80, Has83, Mea76, Pra86, Tak87], we found that the search speeds of existing approaches are all constrained by the single-byte comparison speed of the implementation. Further, we observed that prior approaches typically exhibit great percentages of redundant comparisons. Recognizing the byte-comparison upper bound for sequential algorithms and realizing the importance of early detection of the mismatch condition, we have designed a Data Parallel Pattern Matching (DPPM) algorithm to be executed at each SSP. The DPPM algorithm broadcasts the target pattern one character at a time, comparing each character against an entire input block in parallel. Each block consists of K bytes- one byte per comparator. The simultaneous processing of an entire input block from input string **W** differs from the systolic array, cellular array, and finite-state automata approaches which operate on **W** on a byte-by-byte basis. Rather than broadcasting the input data to many comparators, DPPM broadcasts the characters in pattern **Q** one by one into all the comparators on a demand-driven basis. A mismatch-detection mechanism, which inputs a new block immediately upon detecting a mismatch, is used to improve the throughput achievable for string searching.

For example, a match of q_1 of pattern **Q** with an element w_j of the current block of **W** will trigger the loading into position **j + 1** and comparison with q_2 of the next target character. Subsequent comparison outputs (in this case, q_2 with w_{j+1}) are 'and'ed with the previous results, in parallel, to generate new comparison results. The previous results are shifted one position before the 'and' operation to emulate the shifting of the input string. If q_1, $q_2, q_3, \ldots, q_h$ match $w_j, w_{j+1}, w_{j+2}, \ldots, w_{j+h-1}$, respectively, then a full match has occurred. On

the other hand, if, after any *comparison cycle* (the broadcast of q_i, and the 'and' of the current results with the past history), all the comparison results are zero and no partial-match traces generated from the previous input block are waiting, an early-out flag will be set to indicate that further comparison of the current block of **W** is unnecessary.

On detection of the early-out flag, the next block of input data is loaded and the search operation restarted from the first byte of the pattern. Thus, redundant comparisons are eliminated. In our example, if q_1 fails to match any element of the current block of W, then the next block is fetched and loaded immediately. In practice, only the first one or two characters in Q usually need to be tested against the current block of W; a block size of 16 characters yields roughly an order of magnitude speedup in search throughput over traditional sequential algorithms, assuming the same comparator technology.

Figure 2 illustrates the algorithm via a concrete example. Assume that a 4-byte comparator array is used, the pattern to be detected is "filters" and the incoming input stream is "file,filters". After "file" is compared with the first character of the pattern string, "f", a partial match trace is initiated and the next pattern character is compared against the same input string block. This process continues until the comparison on the fourth pattern character generates a mismatch. An early-out flag is set, and a new input block is retrieved to resume the search process. It is necessary to temporarily store the comparison result of the rightmost comparison in register V(i) since the generated result represents a partial match. This temporary result is used as a successful/unsuccessful partial match indicator for the comparison of the next input block. The next block to be loaded is ", fil", and the pattern matching process resumes. This trace crosses over the input block boundary and continues until it reaches the end of the pattern string. This time, the V(i) register is marked with a partial-match success indicator. Eventually, the last character of the pattern is compared, and the HIT flag is set. Note that if multiple occurrences of the pattern are overlapped within the input stream, all occurrences will be detected, as shown in figure 3. In figure 3, the pattern to be matched is "fifi" and the input stream is "XfififiX". As shown, both the patterns starting at position 2 and position 4 are detected.

The DPPM algorithm has several notable characteristics. First, the mismatch-detection capability reduces redundant comparisons, increasing throughput significantly. The throughput achieved by the parallel algorithm reduces the need for expensive high speed GaAs or ECL devices. Second, the parallel execution of the algorithm detects all occurrences of partial matches; therefore no backup is required in either the pattern or the input data.

Three critical implementation aspects of the DPPM engine are the realization of the comparator array, the required high pattern broadcast rate, and the chip input ports. The propagation delay of the comparator array is proportional to the log of the number of inputs into the array. Therefore, the comparator array supports high comparison rates, even when it includes many comparators. The pattern characters are broadcast by Cascade Buffers [Wes85] via the double-layer metal runs to minimize the propagation delay. Using input-buffer design similar to [Cha87] would allow very high-speed communication between the storage devices and the chip.

4. JAS Performance Evaluation

To evaluate the performance of the SSP, a functional simulator was written for measurement on an existing database at Bellcore. The 3.7 MBytes database consists of abstracts of 5,137 Bellcore technical memoranda with topics from communications, computer science, physics, devices, fiber optics, signal processing, etc.. One hundred different patterns were evaluated. Each set of 25 patterns were randomly selected by sampling vocabulary from each of four disciplines: linguistics, computer science, electrical engineering, and device physics. The selected patterns represent typical keywords commonly encountered in queries to the database. A list of the 100 sample patterns is presented in Appendix A. The pattern lengths vary from 3 to 14 with an average of 7.34 characters. The starting character of the patterns were roughly uniformly distributed among the 26 English case-insensitive characters.

The patterns were used as inputs to the simulator which measured and collected the number of comparisons used for each pattern in searching the database. Figure 5 shows the average number of comparison cycles per block, C, at different block sizes, from 1 to 1024. For all block sizes tested, C is less than 3.2 despite an average pattern length of 7.34 characters. At a block size of 1, the DPPM algorithm degenerates to a sequential comparison. As the block size increases, the chance of matching the pattern also increases, and thus requires more comparison cycles.

Figure 6 shows the histogram of the number of comparison cycles used for the pattern "processor" at a block size of 16. 71% of all blocks require only one comparison, and 93% require two or fewer comparisons. The early mismatch detection of the DPPM algorithm is effective in eliminating redundant comparisons. From the simulation experiment, it is observed that the average number of comparison cycles used is almost independent of the pattern length, but rather depends on how frequently the first character in the pattern appears in the database. Patterns starting with "a", "e", "s", and "t" require more comparison cycles, while patterns starting with "x" and "z" always require fewer, regardless of the pattern length.

The filter rate of the SSP is defined as the number of bytes that can be searched in one second, and can be computed as

$$\text{Filter Rate} = \frac{\text{Block Size}}{C \times (\text{Cycle Time})}$$

Using 50 ns as the cycle time, the filter rates at different block sizes are shown in Figure 7. At a block size of 16, the filter rate is 222 MBytes per second. This has already exceeded the predicted optical disk transfer rate of 200 MBytes per second and existing memory bandwidth of supercomputers (CRAY, CDC). At a block size of 128, the filter rate reaches 1.2 GBytes per second. Figure 8 shows the speedup at different block sizes which is defined as

$$\text{Speedup} = \frac{\text{Filter Rate at Block Size K}}{\text{Filter Rate at Block Size 1}}$$

The speedup curve shows that the DPPM algorithm exhibits a high degree of parallelism, thus speedup can be achieved effectively by just increasing the block size. Since the predicate evaluation and query resolution are performed at the PE, only very simple comparators and control circuitry are required in each SSP.

5. Conclusion

We presented an information retrieval subsystem called JAS. JAS incorporates several novel features. In JAS, instructions are decomposed into substring-match primitives. The decomposition of the individual instructions into search primitives provides a high degree of flexibility, several storage and retrieval schemes that can be efficiently supported, independence of the query complexity, and easy implementation of previously difficult instructions such as "***A.n.B*** ".

In conjunction with the decomposition of instructions, a novel Data Parallel Pattern Matching (DPPM) algorithm and its associated Substring Search Processor (SSP) is proposed. In contrast to previous approaches, the DPPM algorithm operates on an input block (instead of byte) at a time and incorporates an early mismatch-detection scheme to eliminate unnecessary comparisons. The SSP, a hardware realization of the DPPM algorithm, demonstrates the feasibility of a gigabyte-per-second search processor. A simulation study of the SSP was described. The study demonstrated the potential for very high-speed text filtering.

References

[Bar88] Baru, C. K. and Frieder, O., "Database Operations in a Cube-Connected Multicomputer System", *to appear in the IEEE Transactions on Computers.*

[Cha87] Chao, H. J., Robe, T. J., and Smoot, L. S., "A CMOS VLSI Framer Chip for a Broadband ISDN Local Access System", *Proceedings of the 1987 VLSI Circuits Symposium*, May, 1987.

[Cur83] Curry, T. and Mukhopadhyay, A., "Realization of Efficient Non-Numeric Operations Through VLSI", *Proceedings of VLSI '83*, 1983.

[Dew86] DeWitt, D. J., et. al., "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Twelvth Int'l Conf. on Very Large Data Bases*, pp 228-237, 1987.

[Fos80] Foster, M. J. and Kung, H. T., "The Design of Special Purpose Chips", *IEEE Computer*, 13 (1), pp 26-40, January, 1980.

[Goo81] Goodman, J. R. and Sequin, C. H., "HYPERTREE: A Multiprocessor Interconnection Topology," *IEEE Transactions on Computers*, Vol. c-30, No. 12, pp 923-933, December, 1981.

[Got83] Gottlieb, A., et al., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, " *IEEE Transactions on Computers*, Vol. c-32, No. 2, pp 175-189, February, 1983.

[Has81] Haskin, R. L., "Special-purpose Processors for Text Retrieval", *Database Engineering 4*, 1, pp. 16-29, September, 1981.

[Has83] Haskin, R. L. and Hollaar, L. A., "Operational Characteristics of a Hardware-based Pattern Matcher", *ACM Transactions on Database Systems*, Vol. 8, No. 1, pp 15-40, March, 1983.

[Hil86] Hillyer, B. and Shaw, D. E., "NON-VON's Performance on Certain Database Benchmarks," *IEEE Transactions on Software Engineering*, se-12, 4, pp 577-583, April, 1986.

[Hol79] Hollaar, L. A., "Text Retrieval Computer", *IEEE Computer*, 12 (3), pp 40-50, March, 1979.

[Hol83] Hollaar, L. A., Smith, K. F., Chow, W. H., Emrath, P.A., and Haskin, R. L., "Architecture and Operation of a Large, Full-text Information-retrieval System", in Advanced Database Machine Architecture, Englewood Cliffs, N.J. : Prentice/Hall, 1983, pp 256-299.

[Kit84] Kitsuregawa, M., Tanaka, H., and Moto-Oka, T.,"Architecture and Performance of Relational Algebra Machine GRACE", *Int'l Conf. on Parallel Processing Proceedings*, pp 241-250, August, 1984.

[Mea76] Mead, C. A., Pashley, R. D., Britton, L. D., Yoshiaki, T., and Sando, Jr., S. F., "128-Bit Multicomparator", *IEEE Journal of Solid-State Circuits*, SC-11, No. 5, October, 1976.

[Pog87] Pogue, C. A. and Willett, P., "Use of Text Signatures for Document Retrieval in a Highly Parallel environment." *Parallel Computing* 4 (1987), pp 259-268, Elsevier (North-Holland).

[Pra86] Pramanik, Sakti, "Perfomance Analysis of a Database Filter Search Hardware", *IEEE Transactions on Computers*, Vol. c-35, No. 12, December, 1986.

[Sal88] Salton, G. and Buckley, C., "Parallel Text Search Methods", *Communications of the ACM*, 31(2), pp 202-215, 1988.

[Sta86] Stanfill, C. and Kahle, B., "Parallel Free-text Search on the Connection Machine System", *Communications of the ACM*, 29(12), pp 1229-1239, 1986.

[Sto87] Stone, H. S., "Parallel Querying of Large Databases: A Case Study", *IEEE Computer*, 20 (10), pp 11-21, October, 1987.

[Tak87] Takahashi, K., Yamada, H., and Hirata, M. "Intelligent String Search Processor to Accelerate Text Information Retrieval", *Proceedings of Fifth Int'l Workshop on Database Machines*, pp 440-453, October, 1987.

[Wes85] Weste, N and Eshraghian, K., Principles of CMOS VLSI Design: A Systems Perspective, Reading, Massachusetts: Addison-Wesley, 1985.

APPENDIX A

acoustic	allocation	amplitude	architecture	banyan	basic	bell
broadband	circuit	communication	computer	concurrent	design	distortion
distributed	domain	ear	efficiency	energy	environment	erlang
fiber	field	fine-grain	frequency	gallium	glottis	greedy
ground	hertz	high	hopfield	hypercube	image	information
intelligent	intensity	japanese	jaw	jitter	junction	k-map
kernel	keyboard	knowledge	language	limited	locality	loudness
markov	message	momentum	multi-computer	network	neural	noise
nuclear	object	optic	oscillator	output	packet	phoneme
processor	protocol	quadrature	quantum	query	queue	recognition
research	resource	retrieval	speech	standard	superconduct	synthesis
system	telephone	time	timestamp	transform	ultra-violet	unix
user	utilization	verification	vlsi	voice	voltage	watt
wide	window	word	x-ray	y-net	yield	z-transform
zero	zone					

Table I. JAS PE Instruction Set

A [*Find any document containing the string A*]

```
if match( A ) then return true else return false
```

A B [*Find any document containing the string A immediately followed by the string B*]

```
C := A B      { concatenate A and B }
if match( C ) then return true else return false
```

A ?? B [*Find any document containing string A followed by any two characters followed by string B*]

```
C := A## B { concatenate A , ##, and B }
if match( C ) then return true else return false
```

(A, B, C) % n [*Find any document containing at least n different patterns of the strings A, B, or C*]

```
count_A := 0;
count_B := 0;
count_C := 0;
While not ( END_OF_DOC ) do
    Case type ( match( string ) ) of   { CASE statement used only for clarity }
A : count_A := 1;
B : count_B := 1;
C : count_C := 1
      end;
if count_A + count_B + count_C ≥ n then return true else return false
```

A OR B [*Find any document containing either of the strings A or B*]

```
if ( match( A ) or match( B ) ) then return true else return false
```

```
A  AND  B     [ Find any document containing both the strings A and B                    ]
            found_A := false
            found_B := false
            While not ( END_OF_DOC ) do begin
                Case type ( match( string ) ) of   {  CASE statement used only for clarity }
        A :  begin
               if found_B then
                 if address( match() ) - addr_B > length ( B ) then return  true
               else if not found_A then begin
                 addr_A := address( match() );
                  found_A := true
                 end
             end;
        B :  begin
               if found_A then
                 if address( match() ) - addr_A > length ( A ) then return  true
               else if not found_B then begin
                 addr_B := address( match() );
                  found_B := true
                 end
             end;
                   end;
            return  false
```

```
A ... B       [ Find any document containing the string A followed either immediately  or after an arbitrary
              number of characters by string B                                                           ]
            addr_B := default
            addr_A  :=  address( match( A ) )        { addr_A = default if A is not found }
            While not ( END_OF_DOC ) and ( addr_A <> default ) do begin
              temp  :=  address( match( B ) )        { find last B }
               if temp <> default then addr_B := temp
              end
            if ( addr_A = default ) or ( addr_B = default ) then return false
            if addr_B - addr_A > length( A ) then return true   else   return false
```

```
A .n. B       [ Find any document containing the string A followed by string B  within n characters        ]
            addr_A := address( match( A ) );
      if addr_A = default then return false;
            length_A := length( A );
            While not ( END_OF_DOC ) do
                Case type ( match( string ) ) of   {  CASE statement used only for clarity }
        A :  begin                               { ignore possible overlap A with A  }
               temp := address( match() );
               if ( temp <> default ) and ( temp - addr_A > length_A ) then
                   addr_A := temp;
             end;
        B :  begin                               { ignore possible overlap B with A  }
               temp := address( match() );
               if ( temp <> default ) and ( temp - addr_A > length_A ) then
                   if temp - addr_A - length_A ≤ n  then return  true
             end;
                     end;
            return  false
```

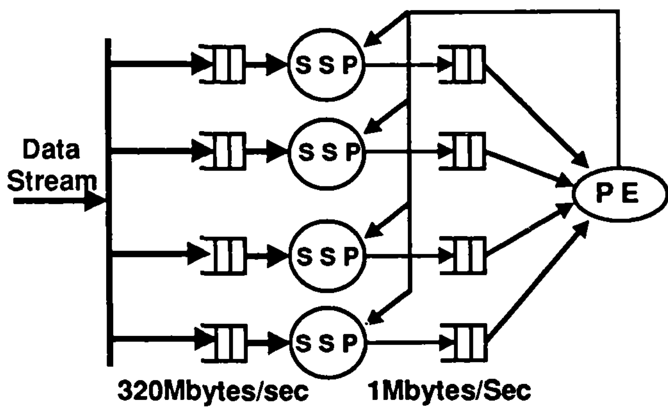


Figure 1. JAS System Architecture

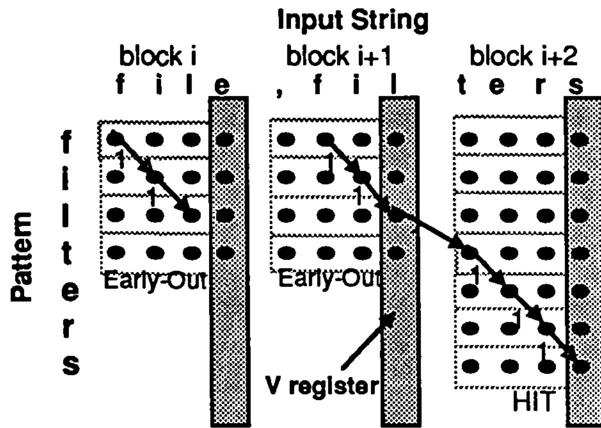


Figure 2. Example without Overlap

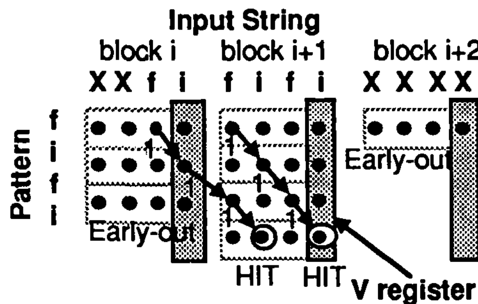


Figure 3. Example with Overlap

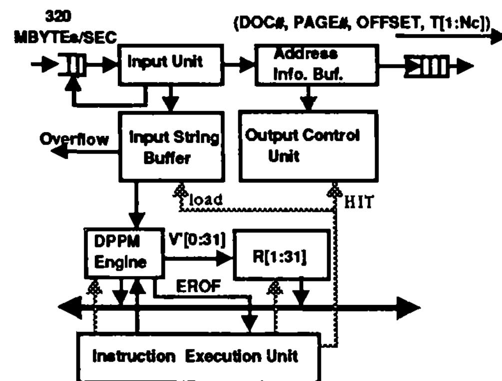


Figure 4. Substring Search Processor

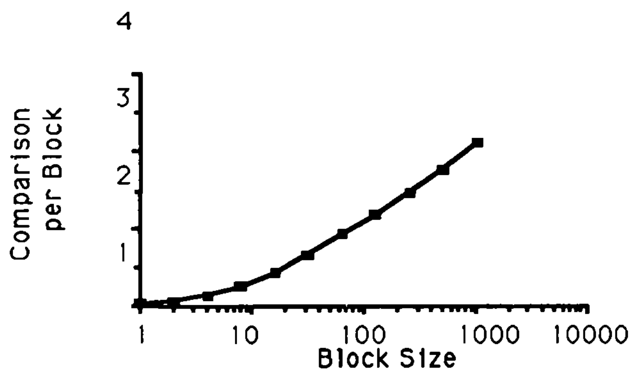


Figure 5. Compare Cycle vs. Block Size

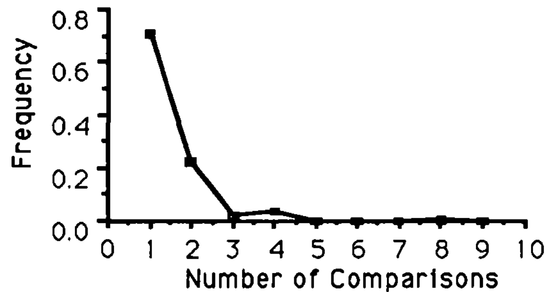


Figure 6. Number of Comparisons

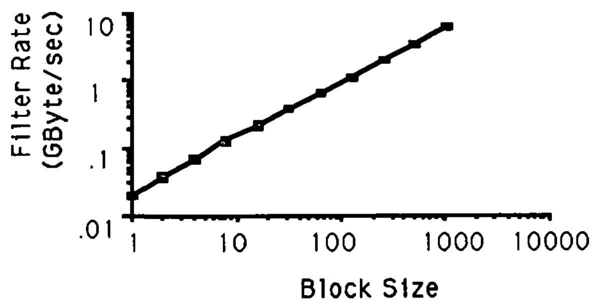


Figure 7. Filter Rate vs. Block Size

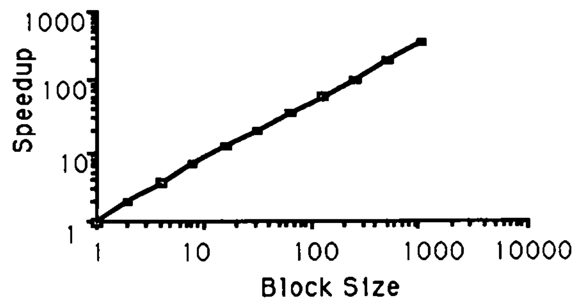


Figure 8. Speedup vs. Block Size

Parallel Query Evaluation: A New Approach to Complex Object Processing

T. Härder H. Schöning A. Sikeler

University Kaiserslautern, Department of Computer Science, P.O. Box 3049, D-6750 Kaiserslautern, West Germany

Abstract

Complex objects to support non-standard database applications require the use of substantial computing resources because their powerful operations must be performed and maintained in an interactive environment. Since the exploitation of parallelism within such operations seems to be promising, we investigate the principal approaches for processing a query on complex objects (molecules) in parallel. A number of arguments favor methods based on inter-molecule parallelism as against intra-molecule parallelism. Retrieval of molecules may be optimized by multiple storage structures and access paths. Hence, maintenance of such storage redundancy seems to be another good application area to explore the use of parallelism.

1. Introduction

Non-standard database applications such as 3D-modeling for workpieces or VLSI chip design [1] require adequate modeling facilities for their application objects for various reasons. Enhanced data models provide many of such desired features; above all they support forms of data abstraction and encapsulation (e.g. ADTs) which relieve the application from the burden of maintaining intricate object representations and checking complex integrity constraints. On the other hand, the more powerful the data model the longer the DBMS's execution paths, since all aspects of complex object handling have to be performed inside the DBMS. Hence, appropriate means to concurrently execute "independent" parts of a user operation are highly desirable [2].

The use of intra-transaction parallelism for higher-level operations was investigated in a number of database machine projects [3]. These approaches focus on the exploitation of parallelism in the framework of the relational model. Complex relational queries are transformed into an operator tree of relational operations in which subtrees are executed concurrently (evaluation of subqueries on different relations) [4]. Other approaches utilize special storage allocation schemes by distributing relations across multiple disks. Parallelism is achieved by evaluating the same subquery on the various partitions of a relation [5, 6].

We investigate possible strategies to exploit parallelism when processing complex objects. In order to be specific, we have to identify our concepts and solutions in the framework of a particular data model and a system design facilitating the use of parallelism. Therefore, we refer to the molecule-atom data model (MAD model [7]) which is implemented by an NDBS kernel system called PRIMA [8]. We use the term NDBS to describe a database system tailored to the support of non-standard applications.

2. A Model of NDBS Operations

The overall architecture consists of a so-called NDBS kernel and a number of different application layers, which map particular applications to the data model interface of the kernel. Our application-independent kernel is divided into three layers:

- The storage system provides a powerful interface between main memory and disk. It maintains a database buffer and enables access to sets of pages organized in segments [8].
- The access system manages storage structures for basic objects called atoms and their related access paths. For performance reasons, multiple access paths and redundant storage structures may be defined for atoms.
- The data system dynamically builds the objects available at the data model interface. In our case, the kernel interface is characterized by the MAD model. Hence, the data system performs composition and decomposition of complex (structured) objects called molecules.

The application layer uses the complex objects and tailors them to (even more complex) objects according to the application model of a given application. This mapping is specific for each application area (e.g. 3D-CAD). Hence, different application layers exist which provide tailored interfaces (e.g. in form of a set of ADT operations) for the corresponding applications.

The NDBS architecture lends itself to a workstation-server environment in a smooth and natural way. The application and the corresponding application layer are dedicated to a workstation, whereas the NDBS kernel is assigned either to a single server processor or to a server complex consisting of multiple processors. This architectural subdivision is strongly facilitated by the properties of the MAD model: Sets of molecules consisting of sets of heterogeneous atoms may be specified as processing units.

Before we start to evaluate our concepts for achieving parallelism to perform data system and access system functions, we briefly sketch our process (run-time) environment. In order to provide suitable computing resources, PRIMA is mapped to a multi-processor system, i.e. the kernel code is allocated to each processor of our server complex (multiple DBMSs). The DB operations to be considered are typically executed on shared (or overlapping) data which requires synchronization of concurrent accesses. Due to the frequency of references (issued from concurrent tasks) accessibility of data and synchronization of access must be solved efficiently.

For this reason, we have designed a closely coupled multiprocessor system as a server complex. Each instance of PRIMA (running on a particular processor with private memory) uses an instruction-addressable common memory [9] for buffer manage-

ment, synchronization, and logging/recovery. Furthermore, each instance of PRIMA is subdivided into a number of processes which may initiate an arbitrary number of tasks serving as run-units for the execution of single requests. Cooperation among processes is performed by establishing some kind of client-server relationship; the calling task in the client process issues a request to the server process where a task acts upon the request and returns an answer to the caller. In our model, a client invokes a server asynchronously, i.e. it can proceed after the invocation, and hence, can run concurrently with this server. To facilitate such complex and interleaved execution sequences we have designed a nested transaction concept [10] which serves as a flexible dynamic control structure and supports fine grained concurrency control as well as failure confinement within a nested subtransaction hierarchy. Due to space limitations we can not refine our arguments on these system issues [11].

2.1 The Data System Interface

In order to describe the concepts for achieving parallelism in sufficient detail, we have to refine our view of the kernel architecture and the interfaces involved. It is obvious that the data model plays the major role and determines many essential factors which enable reasonable parallelism: sufficiently large data granules, set orientation of request, dynamic construction of objects (result sets), flexible selection of processing sequences, etc.

In our system, the data model interface is embodied by the MAD model and its language MQL which is similar to the well-known SQL language. Here, we cannot introduce this model with all its complex details, but only illustrate the most important concepts necessary for our discussion. In the MAD model, atoms are used as a kind of basic element (or building block) in order to represent entities of the real world. In a similar way to tuples in the relational model, they consist of an arbitrary number of attributes. The attributes' data types can, however, be chosen from a richer selection than in the relational model, i.e. apart from the conventional types the type concept includes

- the structured types RECORD and ARRAY,
- the repeating group types SET and LIST, both yielding a powerful structuring capability at the attribute level as well as
- the special types IDENTIFIER (serving as surrogates) for identification purposes and REF_TO for the connection of atoms.

Atoms are grouped to atom types. Relationships between atoms are expressed by so-called connections and are defined as connection types between atom types. Connection types are treated in a symmetric way, i.e. connections may be used in either direction in the same manner. Such connection types directly map all types of relationships (1:1, 1:n, n:m). The flexibility of the data model is greatly increased by this direct and symmetric mapping. Connection types are represented by a pair of REF_TO attributes (reference and "back-reference") one in either involved atom type, e.g.:

- FIDs: SET_OF (REF_TO(Face.EIDs)) in an atom type Edge
- EIDs: SET_OF (REF_TO(Edge.FIDs)) in an atom type Face.

In the database, all atoms connected by connections form meshed structures (atom networks) as illustrated in Fig. 1a.

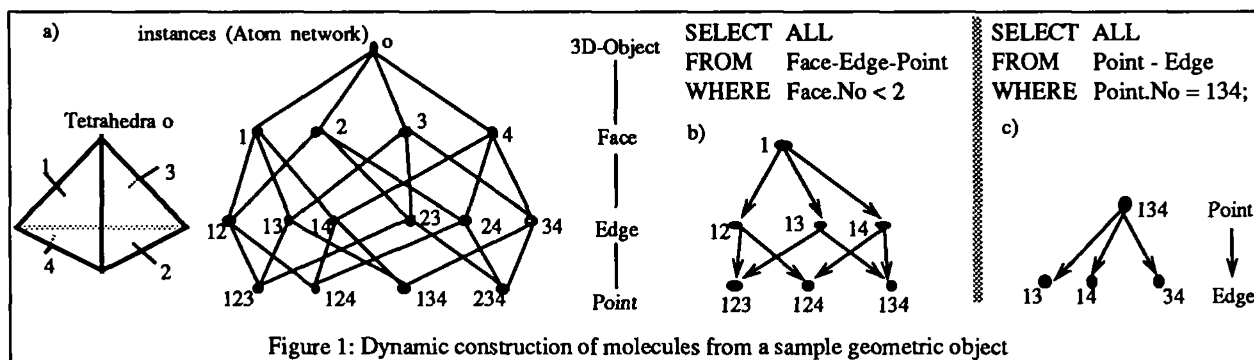


Figure 1: Dynamic construction of molecules from a sample geometric object

Molecules are defined dynamically by using MQL statements and have to be derived at run-time. Each molecule belongs to a molecule type. The type description establishes a connected, directed and acyclic type graph (cycles occur when recursive types are specified), in that a starting point (i.e. root atom type) and all participating atom and connection types (for short "-") are specified. A particular example of a molecule type is Face-Edge-Point. Such a molecule type determines both the molecule structure as well as the molecule set which groups all molecules with the same structure. At the conceptual level, the dynamic construction of molecules proceeds in a straight-forward manner using the molecule type description as a template: For each atom belonging to the root atom type all children, grandchildren and so on are connected to the molecule structure, terminating after all leaves of the molecule structure are reached. Connecting children to the molecule structure means performing the hierarchical join which is supported by the connection concept. Hence, for each root atom a single molecule is constructed. Fig. 1b shows the result of a molecule construction for Face-Edge-Point molecules, where the set of molecules was restricted. Furthermore, it illustrates the most important properties of the MAD interface:

- An MQL request handles a set of molecules.
- The molecules as complex objects consist of sets of atoms of different type, i.e. they are embodied by sets of interrelated heterogeneous record structures.
- Molecule construction is dynamic and allows symmetric use of the atom networks (e.g. Point-Edge (Fig. 1c)).

2.2 The Access System Interface

The access system provides an atom-oriented interface which allows for direct and navigational retrieval as well as for the manipulation of atoms. To satisfy the retrieval requirements of the data system, it supports direct access to a single atom as well as atom-by-atom access to either homogeneous or heterogeneous atom sets. Manipulation operations (insert, modify, and delete) and direct access operate on single atoms identified by their logical address (or surrogate) which is used to implement the IDENTIFIER attribute as well as the REF_TO attributes. Performing manipulation operations, the access system is responsible for the automatic maintenance of the referential integrity defined by the REF_TO attributes. Thus, a manipulation operation on such an attribute requires implicit manipulation operations on other atoms in order to adjust the appropriate back references.

Different kinds of scan operations are introduced as a concept to manage a dynamically defined set of atoms, to hold a current position in such a set, and to successively deliver single atoms. Some scan operations are added in order to optimize retrieval access. Therefore, they may depend on the existence of a certain storage structure (defined by the database administrator):

- The atom-type scan delivers all atoms in a system-defined order utilizing the basic storage structure existing for each atom type.
- The access-path scan provides fast value-dependent access based on different access path structures such as B-trees, R-trees, and grid files.
- The sort scan processes all atoms following a specified sort criterion also utilizing the basic storage structure of an atom type. However, sorting an entire atom type is expensive and time consuming. Therefore, a sort scan may be supported by an additional storage structure, namely the sort order.
- The atom-cluster scan speeds up the construction of frequently used molecules by allocating all atoms of a corresponding molecule in physical contiguity using a tailored storage structure as a so-called atom cluster. For example, in Fig. 1 each Face atom and all its associated Edge and Point atoms may be organized to form an atom cluster [12]. On a logical level, an atom cluster corresponds to a molecule. It is described by a special so-called characteristic atom which consists of references to all atoms belonging to the molecule. This characteristic atom together with all the referenced atoms is mapped onto a single physical record which in turn is stored in a set of pages.

The underlying concept is to make storage redundancy available outside the access system by offering appropriate retrieval operations (i.e. the choice of several different scans for a particular access decision by the optimizer of the data system), whereas in the case of manipulation operations storage redundancy has to be hidden by the access system. As a consequence, maintaining storage redundancy in an efficient way is a major task of the access system. However, sequential update of all storage structures existing for a corresponding atom results in a lack of efficiency which is not acceptable. Therefore, exploiting parallelism seems to be a natural way to speed up a single manipulation operation.

3. Using Parallelism in Query Processing

Query processing operates on sets of molecules which are either extracted from the database (retrieval) or updated, inserted, or deleted (manipulation). In either case, the complex operation must be evaluated using operations on a single atom at a time. In this chapter we discuss some techniques to exploit parallelism in executing these atom-oriented operations as well as higher-order operations on intermediate results.

Three phases of query processing can be isolated [13]. The compilation phase checks for the syntactic and semantic correctness of a query and performs some obviously simplifying query transformations. Finally, it derives an operator tree consisting of several operator types, most of which accept and produce sets of molecules. The leaves of this operator tree transform heterogeneous atom sets to molecules (construction of simple molecules for retrieval) and vice versa (molecule modification by atom manipulation). This operator tree is the input of the optimization phase which is expected to reorganize it somehow into an "optimal" form. This includes reordering, combination, and splitting of operator tree nodes. Furthermore, methods have to be chosen for each operator, i.e. evaluation strategies (e.g. nested-loop join), storage structure usage (e.g. B-tree), and amount of parallelism (as described below). Although there is much to be said about these two phases, we concentrate on the discussion of the third phase, i.e. query evaluation. The input of this phase is the operator tree introduced above. For each node, we need an active unit to compute its result. Since there are several operator types, we assume a process for each of them. As a consequence, more than one node of an operator tree may be assigned to the same process.

We discuss the operator tree interpretation separately for retrieval and manipulation. In either case, we have to handle a set of molecules, each of which consists of a set of atoms. Thus, we investigate parallel handling of molecules (inter-molecule parallelism) as well as parallel handling of a molecule's components (intra-molecule parallelism).

3.1 Parallelism in Retrieval Evaluation

Evaluation starts at the root operator, which needs results from all of its children in the operator tree to compute its own result. Depending on the operator type and the method chosen, children can be evaluated in parallel, e.g. the children of a binary merge-join operator can be accessed concurrently, while those of a nested-loop join cannot. The evaluation of leaf operators requires a lot of access system calls to build up simple molecules (hierarchical, non-recursive molecules). Of course, we assume that the access system is able to handle an arbitrary number of asynchronous calls in parallel. Therefore, construction of simple molecules seems to be a good candidate for using parallelism within the handling of each molecule.

Parallelism Within Construction of Simple Molecules

An obvious strategy to construct simple molecules is to call the root atom of each molecule via access system scan. Following this all child atoms of the root atom (which are identifiable by reference attributes) are called, then their children, and so on. These accesses to the children of an atom may be done in parallel (example 1a).

Query a	Query b	Query c
SELECT ALL FROM Face-Edge-Point WHERE Face.No = 123;	SELECT ALL FROM Face-Edge-Point WHERE FOR_ALL Edge: Length > 10;	SELECT ALL FROM Face-Edge-Point WHERE (FOR_ALL Point: x-coordinate = 5) OR (EXISTS Edge: Length > 10);
Strategy: Call Face; Call all edges and all points in parallel;	Strategy: Call Face; Call all edges sequentially; If all edges fulfil the restriction: call all points in parallel;	Strategy: Call Face; • call edges sequentially; then call points sequentially; or • call one edge, call its points sequentially; or • call n edges in parallel, call their points sequentially;

Example 1: Three sample queries and parallelism strategy choices

However, in many cases the user is not interested in all molecules of a certain type, but strongly restricts the molecules he wants to see. In this situation, it would be inefficient to fetch all atoms of all molecules and then throw away most of them by a separated restriction operator. Instead, we want to integrate the restriction facility into the operator "construction of simple molecules" leading to a more efficient evaluation strategy. Restrictions on the root atom are passed on to the access system which allows scan restrictions. All other restrictions on dependent atoms are evaluated as early as possible. As soon as it becomes evident during molecule construction that a molecule will be disqualified, none of its atoms has to be fetched any more, thus saving many access system calls (example 1b).

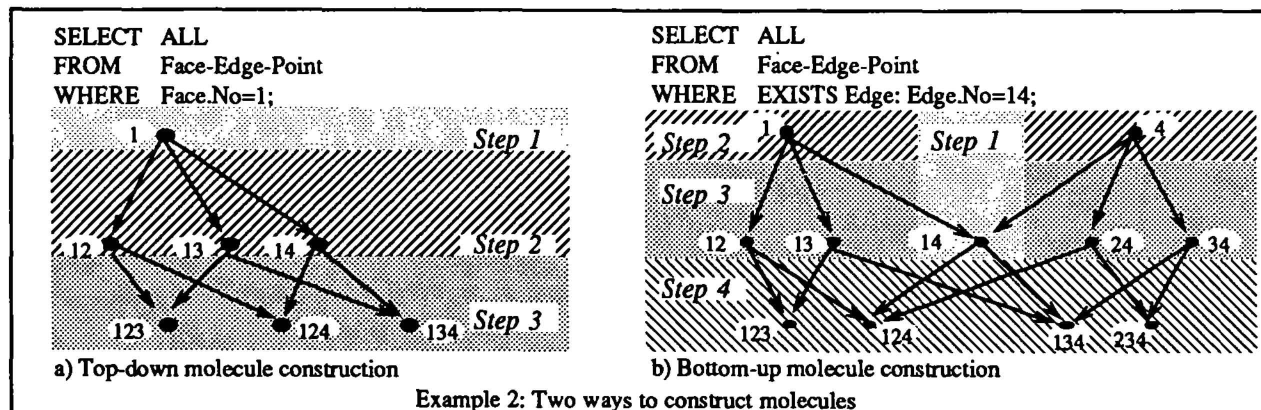
Of course, this approach is contradictory to the parallel molecule construction proposed above, because we want to fetch as few atoms as possible, if a molecule is disqualified. Therefore, we combine both techniques: Atom types that do not contribute to molecule qualification should be treated last. Their atoms can be called in parallel. Atom types restricted by an ALL-quantifier should be called sequentially, since each atom of this type can indicate molecule disqualification. While good strategies for these extreme cases are easy to find, much more complicated situations can be thought of (example 1c). They raise the question whether in some situation a compromise on the amount of parallelism and unnecessary atom accesses should be made, e.g. limitation of parallel atom calls to a constant n, thereby limiting unnecessary atom calls to n-1 (third choice in example 1c). We are still investigating this case for generally applicable rules to decide the optimal amount of parallelism for each atom type as well as the best sequence of atom accesses.

The top-down approaches suggested above are sometimes not the most efficient strategies. When highly selective restrictions are defined on some child types, a bottom-up approach may be more promising. In this case, the first step evaluates the qualifying child atoms. Since some of these atoms may be orphans, it is necessary to explicitly check the existence of a related root atom. Finally, the whole molecule is constructed for each of the identified root atoms following the same guidelines as sketched above (example 2b).

So far, we have discussed parallelism within the construction of one molecule. Since queries deal with sets of molecules, we should consider inter-molecule parallelism, too.

Inter-Molecule Parallelism

The most simple model for the computation of a set of molecules is to build up the first molecule completely, then the second and so on, thereby preserving the order of molecules induced by construction of simple molecules. Following this control scheme, there cannot be any parallelism among an process and its descendants or ancestors. To enable this kind of parallelism, we pro-



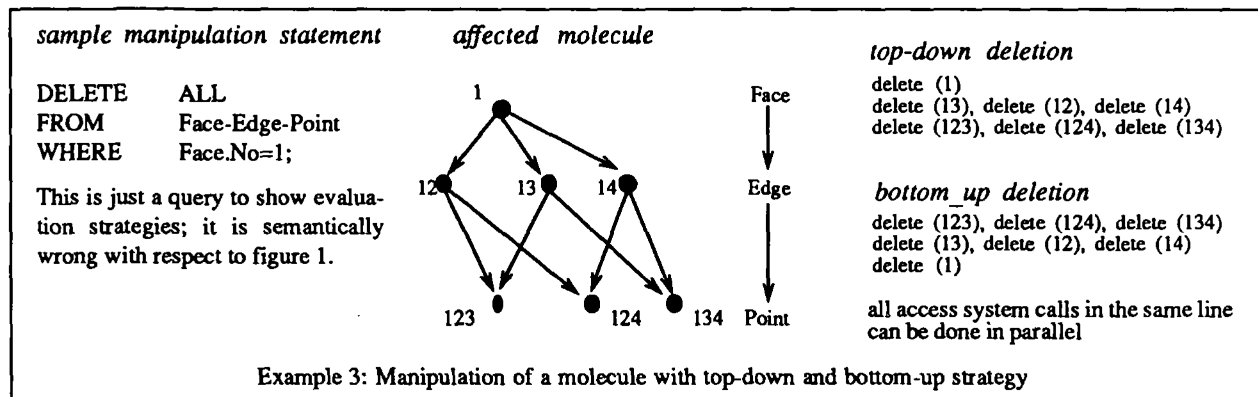
Example 2: Two ways to construct molecules

pose a pipeline mechanism. In particular, whenever the process for construction of simple molecules finds a root atom for a molecule, it builds up this molecule in a separate task, while another concurrent task calls the next root atom.

The pipeline structure defined this way (which at this point of the discussion is introduced as a model of computation and not as schedule for hardware-assignment), is very dynamic and complex, since the number of pipeline stages to run through is data dependent for many operator types and may vary for each molecule. Since this results in varying construction times, order-preservation cannot be guaranteed. As a consequence, there must not be any operator with a varying number of pipeline stages in the operator tree between a sort operator and the corresponding operator that relies on the sort order.

3.2 Parallelism in Manipulation Evaluation

As for retrieval evaluation, we consider intra- and inter-molecule parallelism. Parallelism among several molecules by creation of a separate task for each of them is possible for manipulation, too. When existing molecules are to be manipulated, tasks emerging from retrieving them can be continued for manipulation. Within one molecule, either a top-down or a bottom-up strategy can be applied, both of them allowing parallelism among most of the atoms of a molecule (example 3).



Example 3: Manipulation of a molecule with top-down and bottom-up strategy

4. Maintaining Redundancy by Parallel Operations

To speed up data system operations we have introduced some algorithms for the parallel construction/maintenance of complex objects represented as sets of heterogeneous atoms. In the following, we discuss the implementation of concurrent maintenance operations on redundant storage structures used for such atoms. As in the data system, two kinds of parallelism may be distinguished within the access system.

- The inter-operation parallelism allows for the parallel execution of an arbitrary number of independent access system calls. This kind of parallelism is a prerequisite for the parallelism introduced in the data system.
- The intra-operation parallelism however, exploits parallelism in executing a single access system call.

In this chapter, we will concentrate on intra-operation parallelism, since inter-operation parallelism is easily achieved by the underlying processing and transaction concept. For this purpose, however, the mapping process performed by the access system has to be outlined in some more detail in order to reveal purposeful suboperations to be executed in parallel.

In order to conceal the storage redundancy resulting from the different storage structures we have introduced the concept of a logical record (i.e. atom) made available at the access system interface and physical records stored in the "containers" offered by the storage system, i.e. each physical record represents an atom in either storage structure. As a consequence, an arbitrary number of physical records may be associated with each atom. For example, the creation of an atom cluster for each Face-Edge-Point molecule in Fig. 1 would imply that all Edge atoms belong to two atom clusters and all Point atoms to three (due to the properties of a tetrahedra). Furthermore, they always belong to the basic storage structure.

The relationship between a single atom and all its associated physical records is maintained by a sophisticated address structure related to each atom type. This address structure maps the logical address identifying an atom onto a list of physical addresses each indicating the location of a corresponding physical record within the "containers" (page address).

In contrast to the data system, however, the exploitation of parallelism within the access system is limited to the manipulation operations. Although most of the retrieval operations are also decomposed into further suboperations (e.g. in the case of an access-path scan on a tree structure: read the next entry in order to obtain the next logical address, access the address structure for the associated physical addresses, access either physical record), these suboperations cannot be executed in parallel due to the underlying precedence structure. Furthermore, each retrieval operation is tailored to a certain storage structure, thus operating not only on a single atom, but also on a single physical record.

On the other hand, each manipulation operation on an atom may be decomposed in quite a natural way into corresponding manipulation operations on the associated physical records. These lower-level manipulation operations, however, should be executed in parallel due to performance reasons. There exist (at least) two alternatives to perform such a parallel update:

Deferred Update

Deferred update means that during a manipulation operation on an atom initially only one of the associated physical records (e.g. in the basic storage structure of the atom type) is altered. All other physical records as well as the access paths are marked as invalid. Finally, a number of "processes" is initialized which alter the invalid structures in a deferred manner, whereas the manipulation operation itself is finished. Thus, low-level manipulation operations on additional storage structures may still run, although the manipulation operation on the corresponding atom or even each higher-level operation initializing the modification is already finished. This, however, strongly depends on the embedding of deferred update into the underlying transaction concept.

In order to mark a physical record as invalid the address structure may be used to indicate whether or not the corresponding physical record is valid. Therefore, all operations which utilize the address structure in order to locate a physical record may determine the valid records, whereas all operations which do not utilize the address structure will access invalid records unless the appropriate storage structure was already altered by the corresponding "process". Hence, the corresponding storage structures themselves (access paths, sort orders, and atom clusters) have to be marked as invalid and when performing a scan operation on such an invalid structure each physical record has to be checked as to whether or not it is valid. This, however, requires an additional access to the address structure in order to locate a valid record. Consequently, the speed of a scan operation degrades, since each access to the address structure may result in an external disk access. In order to avoid this undesired behaviour, all invalid atoms (or their logical addresses) may be collected in a number of special pages assigned to each storage structure. These pages may be kept in the database buffer throughout the whole scan operation thus avoiding extra disk accesses. Nevertheless, each physical record has to be compared with the atoms collected in these pages. However, this is not sufficient, since each manipulation operation may require a modification of the whole storage structure, e.g. modifying an attribute which establishes a sort criterion requires the rearrangement of the corresponding physical record within the sort order. This fact also has to be considered during a scan operation. As a consequence, some of the scan operations may become rather complex and thus inefficient. For all these reasons, deferred update seems to be a bad idea.

Concurrent Update

The problem of maintaining invalid storage structures, however, is avoided by concurrent update. Concurrent update means that each manipulation operation on an atom invokes a number of "processes" which alter the associated physical records and access paths in parallel. The manipulation operation is finished when all "processes" are completed. When sufficient computing resources are available, concurrent update may not be more expensive, in terms of response time, than update of a single physical record if we neglect the cost of organizing parallelism.

Depending on the software structure of the access system, there are different ways to perform a concurrent update:

Autonomous Components

Each manipulation operation on an atom is directly passed to all components maintaining only a single storage structure type. Each component checks which storage structures of the appropriate type are affected by the manipulation operation. The corresponding storage structures are then modified either sequentially or again in parallel.

As a consequence, all components have to provide a uniform interface including all manipulation operations offered by the access system (i.e. insert, modify, and delete of a single atom identified by its logical address) as well as all retrieval operations. A quite simple distribution component directs each request to all components maintaining a storage structure type and collects their responses. This means, each component initially performs an evaluation phase during which it checks

- whether or not it has to perform the desired operation and if so,
- which storage structures of the appropriate type are really affected.

For this purpose, the addressing component (maintaining the common address structure) and the meta-data system (maintaining all required description information) are utilized. After the evaluation phase the proper operation is performed on each affected storage structure either sequentially or in parallel, thereby again utilizing two common components: the addressing component in order to notify the modification of a physical address and the mapping component in order to transform a logical record (i.e. atom) into a physical record and vice versa (thus achieving a uniform representation of physical records which is mandatory for some retrieval operations which use one of the physical records when accessing an atom (e.g. direct access)).

Thus, it is quite easy to add a new storage structure type (e.g. a dynamic hash structure as an additional access path structure) by simply integrating a corresponding component into the overall access system. However, there may be some drawbacks regarding performance aspects. During each operation, all components have to perform the evaluation phase although in many cases only a few or even only one component are affected. Moreover, the addressing component may become a bottleneck, since access to the address structure has to be synchronized in order to keep it consistent.

General Evaluation Component

These problems, however, may be avoided by a general evaluation component which replaces the simple distribution component as well as the evaluation phases in each of the components maintaining a storage structure type. Additionally, it solely maintains the address structure. As a consequence, this general evaluation component becomes much more complex. It requires dedicated

information about each component in order to decide whether or not a component is affected by an operation, and it has to know the operations offered by either component in order to invoke the corresponding component in the right way. Although these operations may be tailored to the corresponding storage structure type (e.g. insert (key, logical address) in the case of an access path structure), it seems to be useful again to provide a uniform interface to all components in order to allow for a certain degree of extensibility. Such an interface has to consider the different characteristics of each storage structure type and the corresponding component (e.g. maintaining logical addresses instead of physical records) in an appropriate way.

In our initial design, we have decided to implement concurrent update based on a general evaluation component. In our opinion, concurrent update seems to be the better solution due to the invalidation problem. Although both software structures, autonomous components and general evaluation components, have their pros and cons with respect to performance and extensibility aspects [14], we prefer the general evaluation component, since it promises better performance. However, more detailed investigations are still necessary in order to determine the best way which may be a mixture of all. In particular, the influence of the underlying hardware architecture has to be investigated in more detail.

5. Conclusion

We have presented a discussion of the essential aspects of parallel query processing on complex objects. The focus of the paper has primarily been on the investigation of a multi-layered NDBS to achieve reasonable degrees of parallelism for a single user query. We have derived several design proposals embodying different concepts for the use of parallelism. In the data system, intra- and inter-molecule parallelism were explored. To exploit the former kind of parallelism seems to be more difficult because it turns out that it is very sensitive to the optimal degree of parallelism which may vary dynamically depending on the complex object characteristics. The latter concept is considered more promising because it allows simpler solutions. In the access system two approaches were investigated. Deferred update seems to provoke more problems than the solutions it might provide whereas concurrent update on redundant storage structures seems to incorporate a large potential for successful application.

Currently, we have finished the PRIMA implementation (single user version) and are integrating the proposed concepts for achieving parallelism in order to have a testbed for practical experiments. Performance analysis will reveal their strength and weaknesses at a more detailed level.

In the future, we wish to investigate further concepts for exploitation of parallelism. Another possibility of parallel execution on behalf of a single user would be the simultaneous activation of multiple requests within the application; for example, by means of a window system a user could issue several concurrent calls inherently related to the same task in a construction environment. Other possibilities to specify concurrent actions exist in the application layer where a complex ADT operation could be decomposed into compatible (non-conflicting) kernel requests. Usually multiple kernel requests are necessary for the data supply of an ADT operation; hence, these data requests can be expressed by MQL statements and issued concurrently to kernel servers when they do not conflict with each other or do not require a certain precedence structure.

References

[1] Dittrich, K.R., Dayal, U. (eds.): Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986.

[2] Duppel, N., Peinl, P., Reuter, A., Schiele, G., Zeller, H.: Progress Report #2 of PROSPECT, Research Report, University Stuttgart, 1987.

[3] Special Issue on Database Machines, IEEE Transactions On Computers, Vol. C-28, No. 6, 1979.

[4] DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K., Muralikrishna, M.: GAMMA - A High Performance Dataflow Database Machine, in: Proc. VLDB 86, pp. 228-237.

[5] Neches, P.: The Anatomy of a Database Computer System, in: Proc. IEEE Spring Compcon, San Francisco, Feb. 1985.

[6] Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., Young, H.: Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience, IBM Research Report, RJ 6165, San Jose, CA, 1988.

[7] Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. Second Int. Conf. on Expert Database Systems, Tysons Corner, Virginia, 1988, pp. 33-49.

[8] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. VLDB 87, pp. 433-442.

[9] SEQUENT Solutions: Improving Database Performance, Sequent Computer Systems, Inc., 1987.

[10] Moss, J.E.B.: Nested Transactions: An Approach to Reliable Computing, M.I.T. Report MIT-LCS-TR-260, M.I.T., Laboratory of Computer Science, 1981.

[11] Härder, T., Schöning, H., Sikeler, A.: Parallelism in Processing Queries on Complex Objects, in: Proc. Int. Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, 1988, pp. 131-143.

[12] Schöning, H., Sikeler, A.: Cluster Mechanisms Supporting the Dynamic Construction of Complex Objects, to appear in: Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms (FODO'89), June 21-23, 1989.

[13] Freytag, J.C.: A Rule-Based View of Query Optimization, in: ACM SIGMOD Annual Conference, 1987, pp. 173-186.

[14] Carey, M. (ed.): Special Issue on Extensible Database Systems, Database Engineering, Vol. 10, No. 2, 1987.

MULTIPROCESSOR TRANSITIVE CLOSURE ALGORITHMS

Rakesh Agrawal
H. V. Jagadish

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We present parallel algorithms to compute the transitive closure of a database relation. Experimental verification shows an almost linear speed-up with these algorithms.

1. INTRODUCTION

The transitive closure operation has been widely recognized as a necessary extension to relational query languages [1, 12, 16]. In spite of the discovery of many efficient algorithms [3, 5, 10, 13, 17], the computation of transitive closure remains much more expensive than the standard relational operators. Considerable research has been devoted in the past to implementing standard relational operators efficiently on multiprocessor database machines and there is need for similar research in parallelizing the transitive closure operation.

Given a graph with n nodes, the computation of its transitive closure is known to be a problem requiring $O(n^3)$ effort. Transitive closure is also known to be a problem in the class *NC*, implying that it can be solved in poly-log time with a polynomial number of processors. From a practical point of view, however, there are likely to be only a small number of processors — even less than $O(n)$. Therefore, the parallel algorithms that we seek in this paper are ones that require only m ($m \ll n$) processors, and for which the total execution time is no more than $O(n^3/m)$. We also present their implementation on a multiprocessor database machine [15] and report on experimentally observed speed-ups.

The organization of the rest of the paper is as follows. Our endeavor has been to develop the parallel transitive algorithms in an architecture-independent manner. However, to keep the discussion concrete, we consider two generic multiprocessor architectures: *shared-memory* and *message-passing*. These architectures are briefly described in Section 2. We also present primitives that we use in algorithm description in this section. Our parallel algorithms are presented in Section 3. Section 4 describes the implementation of these algorithms on the Silicon Database Machine (SiDBM) [15], and presents performance measurements. We discuss related work in Section 5, and close with some concluding remarks in Section 6.

2. PRELIMINARIES

We seek parallel algorithms that are independent of the exact nature of the underlying multiprocessor so that they may be implemented on different types of multiprocessors. Of course, the costs of the individual operations will differ with the machine and communication model used, affecting the resultant performance of the algorithms. We recognize at the same time that it is impossible to *completely* divorce the execution of a parallel algorithm for a multiprocessor from any architectural assumptions [6]. We, therefore, concentrate on two generic multiprocessor architectures and keep our architectural assumptions as general as possible.

2.1 Generic Architectures

We are interested in two multiprocessor architectures: *shared-memory* and *message-passing* (also referred to as *shared-nothing*). Each processor has some local memory and local mass storage where the database resides. Processors are connected with some communication fabric. In the case of a message-passing architecture, the system interconnect is the only shared hardware resource, whereas in the case of a shared-memory architecture, processors have access to a shared memory.

We assume that the database relation whose transitive closure is to be computed consists of a "source" field, a "destination" field, and other data fields that represent labels (properties) on the arc from a specific source to destination such as distance, capacity, reliability, quantity, etc. The database relations have been partitioned across processors, so that each processor "owns" a part of the relation and there is no replication. Partitioning is horizontal; each processor has all the tuples in the relation (both original and result) with certain specified values for the source or destination field.

† This paper is a condensed version of [2], presented at the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, December 1988.

2.2 Basic Primitives

To present algorithms in an architecture-independent manner, we first define a few primitives that we use in algorithm description in Section 3.

Remote-Get: *Access data from non-local memory.*

A remote-get is executed by a processor to access a piece of data not owned by the processor. If the remote data is unavailable, the remote-get is blocked. We shall write `remote-get(data)` where `data` is the data that needs to be remotely accessed.

Show: *Make data available to remote processors*

The show operation is complimentary to the remote-get operation. A show is executed by a processor to make available a piece of data "owned" by the processor to other processors. A processor may not gain access to remote data unless it has been shown by its owner. We shall write `show(data, processor_list)` to mean show the `data` to processors in `processor_list`. The `processor_list` could be empty.

Enable-Interrupt: *Set up an interrupt event and the interrupt-handling routine*

A processor may receive notification of an external event provided that it has enabled an interrupt. We write `enable(event, action)` to mean upon the occurrence of the interrupt `event`, execute the action specified in `action`.

There are different ways in which a pair of show and remote-get operations may be implemented. A processor doing show may write the data in a remote location and the other processor(s) may then access it remotely. This form of implementation normally exists in a shared-memory system, where the remote location in question is in the shared memory. A second alternative is to do a show by sending (broadcasting or multicasting if multiple receivers are involved) the data to the other processors. The remote-get then requires a local access. This form of communication is found in most message-passing systems. A third alternative is the inverse of the second scheme. A processor may do a show by writing locally to its own memory and provide this address to the intended receivers. The remote-get is then accomplished by a remote access to this location.

Irrespective of the type of architecture used, a show and remote-get pair of operations is considerably more expensive than a local access. This expense may simply be the longer latency of a remote access, but may also include synchronization costs, the effects of contention for shared resources such as a bus, etc. The parallel algorithms that we devise minimize the number of show and remote-get pairs in favor of local accesses.

3. PARALLEL TRANSITIVE CLOSURE ALGORITHMS

We present three parallel transitive closure algorithms: one iterative and two matrix-based direct algorithms.

3.1 Iterative Algorithms

The essential idea of iterative algorithms is to evaluate repeatedly a sequence of relational algebraic expressions in a loop, until no new answer tuple is generated. Included in this family are algorithms such as semi-naive [5], logarithmic [10, 17] and variations thereof [10, 13]. We consider parallelization of the semi-naive algorithm; other iterative algorithms can be parallelized similarly.

If R_0 is the initial relation and R_Δ the set of tuples generated in an iteration, then the *semi-naive* algorithm computes the transitive closure R_f of R_0 as shown below. Drawing upon the results in [4], R_0 was partitioned on the source field, as also R_Δ and R_f. The steps executed by the processor p in the ith iteration are shown below.

Semi-Naive Algorithm (Uniprocessor):

$R_f \leftarrow R_0$
$R_\Delta \leftarrow R_0$
while $R_\Delta \neq \phi$ do
  $R_\Delta \leftarrow R_\Delta \cdot R_0$
  $R_\Delta \leftarrow R_\Delta - R_f$
  $R_f \leftarrow R_f \cup R_\Delta$

Algorithm I.1 (Parallel Semi-Naive):

1) if $(R_\Delta^p)^{i-1} \neq \phi$ then
2) $(R_\Delta^p)^i \leftarrow (R_\Delta^p)^{i-1} \cdot$ remote-get(R_0)
3) $(R_\Delta^p)^i \leftarrow (R_\Delta^p)^i - (R_f^p)^{i-1}$
4) $(R_f^p)^i \leftarrow (R_f^p)^{i-1} \cup (R_\Delta^p)^i$

The closure computation is partitioned in such a way that the set of result tuples owned by a particular processor are generated at the same processor, so that communications and synchronization is minimized. The set-difference and union steps (steps 3 and 4 respectively) can be performed locally without remote access to any tuple in R_Δ. The composition step (step 2), however, requires R_Δ to be joined with the complete R_0 because R_Δ has been partitioned on the source field and it may have a tuple for every destination value. The relation R_0 will have to be remotely accessed. However, provided enough storage is available locally, it may be possible to remotely access R_0 only *once* at the beginning of the iteration, since R_0 does not change from iteration to iteration. All subsequent computation can then be performed locally at each processor. There is no need for synchronizing iterations, and different processors may even compute for different numbers of iterations, since they independently evaluate their termination conditions. The algorithm terminates when all processors are done.

In terms of the graph corresponding to the given relation, this algorithm hands over the complete graph to every processor, but makes a processor responsible for determining reachability from a specified set of nodes. Since a processor has access to the complete graph, it can determine this reachability without any communication with any other processor. The disadvantage is that there may be significant redundant computation in this algorithm. For example, suppose the graph has an arc from i to j and the reachability determination for i and j has been delegated to different processors, then both the processors will end up determining complete reachability for node j.

Thus, this algorithm completely avoids communication and synchronization during the transitive closure computation. The price paid is a relatively more expensive composition step and extra storage requirement with each processor. As such, this algorithm can be very attractive in systems in which communication costs are high, such as loosely-coupled multicomputers.

3.2 Matrix-Based Algorithms

Warshall [19] proposed a uniprocessor algorithm for computing the transitive closure of a Boolean matrix that requires only one pass over the matrix. Given an $n \times n$ adjacency matrix of elements a_{ij} over a n-node graph, with a_{ij} being 1 if there is an arc from node i to node j and 0 otherwise, the Warshall algorithm requires that every element of this matrix be "processed" column by column from left to right, and from top to bottom within a column. "Processing" of an element a_{ij} involves examining if a_{ij} is 1, and if it is, then making every successor of j a successor of i.

It was shown in [3] that the matrix elements can be processed in *any* order, provided the following two constraints are maintained:

1. In any row i, processing of an element a_{ik} precedes processing of the element a_{ij} iff $k < j$, and
2. For any element a_{ij} in row i, processing of the element a_{jk} precedes processing of a_{ij} iff $k < j$.

Various processing orders can be derived subject to these two constraints, giving rise to a whole family of *Warshall-derived* algorithms. We now develop algorithms in which the matrix elements are processed in parallel, while maintaining the above two constraints.

3.2.1 Algorithm M.1

Partition the original relation on the source field so that each processor owns all the successors of a contiguous number of nodes (in terms of adjacency matrix, each processor owns a contiguous set of rows[1]). Processors are numbered and the processor p owns the pth partition. Let there be m processors, and hence m partitions, and let b_p and e_p be the beginning and end nodes of the pth partition. Each processor executes in parallel the following algorithm, written for the processor p:

```
1)    for q from 1 to m do
2)        if p equals q then   /* p drives the computation */
3)            for i from b_p to e_p do
                  /* process elements below the diagonal */
                  for j from b_p to i-1 do process a_ij
4)                copy successors_of(i) into copy_of(i)
5)                show(all, copy_of (i))
6)            for i from b_p to e_p do
                  /* process elements above the diagonal */
                  for j from i+1 to e_p do process a_ij
7)        else  /* q (≠ p) is driving the computation */
8)            for j from b_p to e_p do
9)                remote-get(copy_of(j)) /* from q */
10)               for i from b_p to e_p do process a_ij
```

As noted before, processing of an element a_{ij} involves examining if a_{ij} is 1, and if it is, then making every successor of j a successor of i (that is, ORing row j into row i). Clearly, for processing an element a_{ij}, a processor needs access to both rows i and j. It is guaranteed that the processor will have available to it the row i, since a processor processes only those elements that it owns and the matrix has been partitioned among processors row-wise, but it may have to access remotely the row j.

Algorithm M.1 is an asymmetric algorithm. For each value of q, only one processor (the processor whose number is q) executes the *if* part of step 2 and all other processors execute the *else* part. The processor executing the *if* part drives the computation, and every processor gets to assume this role once.

1. The row i of the adjacency matrix corresponds to the successor list of node i (that is, all tuples of the relation that have i in the source field). Similarly, the column j of the matrix corresponds to the predecessors of the node j. We shall freely alternate between the tuple, graph, and adjacency matrix representations of a relation.

When the processor p is executing the *if* part, it does not require any remote access at step 3 or step 6, since the rows i and j needed for processing the element a_{ij} belong to the partition owned by p. The *if* part is executed in two steps: first the elements below the diagonal are processed, followed by a processing of elements above the diagonal. Elements are processed in row-order from top to bottom. During the processing of elements below the diagonal, as soon as the diagonal element is reached in a row, a copy of this row is made for showing it to other processors (for reasons that will become clear shortly), and processing continues with the next row. Note that there need not be two physical copies; a simple one-bit public/private mark associated with each tuple will be sufficient. Furthermore, a copy is required only in those systems in which other processors have direct access to p's latest successor lists. For example, in a message-passing system in which a show involves a broadcast, the copy created at step 4 is really the content of the broadcast message.

When the processor p is executing the *if* part, all other processors must be executing the *else* part of the algorithm. (This is not strictly true. It is possible that while the processor p is still executing the *if* part, the processor $p+1$, if it has completed executing the *else* part, may move on to start executing its *if* part, since the processors need not synchronize for every new value of q). Any of these processors, say r, will need access to row j that is owned by p before it can execute step 10, which is made available by p at step 5. However, what p shows is a copy of row i made at step 4. This avoids unnecessary synchronization since p does not have to wait for every other processor to see row i before making additions to it at step 6. Precedence constraint 2 will be violated if the processor r gets at step 9 row i that has some nodes added to it by p at step 6. Note that r processes elements in the column order (from left to right) so that as soon as a row is shown by p, all elements in the partition owned by r are processed that have the same column number as the number of this row.

Finally, processors do not have to synchronize at step 1 for every value of q. The blocking remote-get at step 9 performs synchronization, if necessary.

Algorithm M.1 can be modified so that each successor list is not shown as soon as it was ready, but rather a number of lists are clubbed together and shown (and accessed) as a unit. See [2] for details.

3.2.2 Algorithm M.2

A disadvantage of Algorithm M.1 is that if a processor p is slow in creating and showing the successor list of i at step 5, all other processors may be blocked at step 9 for the successor list of i. Instead of blocking such a processor q, Algorithm M.2 attempts to schedule processing of some other element within the partition belonging to q. Instead of assigning contiguous successor lists to the processors, lists are assigned in a round-robin fashion. Assume, as before, that the graph has n nodes and there are m processors. The following algorithm, written for processor p, is executed in parallel by every processor:

```
start:
set the row_received flag to false

/* first process elements below the diagonal */
for (i = p ; i ≤ n ; i = i + m ) do
    if row_received flag is true then go to start
    for j from 1 to i-1 do
        if a_ij has not been processed then
            if j does not belong to the partition of p then
                /* get the row j */
                if copy_of(j) has not been shown then
                    /* do not block */
                    enable(on show of copy_of(j),
                    set the row_received flag to true)
                    continue outer loop with next i
                else/* get it */
                    remote-get(copy_of(j))
            process a_ij
    copy successors_of(i) into copy_of(i)
    show(all, copy_of (i))

/* now process elements above the diagonal */
for (i = p ; i ≤ n ; i = i + m ) do
    if row_received flag is true then go to start
    for j from i+1 to n
        if a_ij has not been processed then
            if j does not belong to the partition of p then
                if copy_of(j) has not been shown then
                    enable(on show of copy_of(j),
                    set the row_received flag to true)
                    continue outer loop with next i
                else
                    remote-get(copy_of(j))
            process a_ij

wait if there is any pending enabled interrupt
```

If a processor p finds that it cannot process an element a_{ij}, since p does not own j, and the successor list of j has not yet been shown by its owner, then p does not block as in Algorithm M.1. Rather, p enables an interrupt to receive notification when the successor list of j is shown, and moves on to process the next row in its partition.

A processor p shows a row i only after p has processed all the elements of row i that belong to the lower-triangular half of the matrix. To minimize blocking, every processor tries to show the rows it owns as early as possible. A processor shows the rows it owns from top to bottom. Thus, whenever a processor p is interrupted on a new row being shown by some other processor, p comes back to processing the top most row in its partition that it has not yet shown to other processors. As in Algorithm M.1, a copy of the successor list is saved to show to the other processors.

A final note with regard to the parallel algorithms presented in this section. Although the algorithms have been presented for reachability computation for the ease exposition, they may be trivially adapted to solve path problems using the techniques presented in [3]. Path computation is a generalization of the simple reachability computation. While reachability computation can tell whether a point p can be reached from q, path computation can additionally determine some properties of the path from q to p. Indeed, most of the applications of practical interest require path computation, and the experimental results presented in the next section are for the performance of these algorithms in computing a bill of materials.

4. EXPERIMENTAL RESULTS

4.1 The Set Up

The algorithms presented in Section 3 have been implemented on the Silicon Database Machine (SiDBM) [15] that presently consists of eight 14 Mhz AT&T 32100-based single board computers interconnected with the standard VMEbus. Each processor has one megabyte of local, on-board memory, and they all have access to 16 megabyte global, off-board memory on the bus. Algorithms were coded in Concurrent C [7].

Two different implementations of communication mechanisms were tried for our algorithms. The first utilized shared memory. The second was a message passing scheme. In this paper, we present results only for the shared memory configuration; see [2] for the results for the message passing configuration.

4.2 The Methodology

Synthetic graphs were used in the performance evaluation experiments. Two parameters of a graph were identified as important: the number of nodes, and the degree of each node. These two parameters were varied to create a set of random graphs.

Since most practical database applications of transitive closure involve path properties [1, 16], we decided to experiment with the computation of transitive closure with path properties. One common use of transitive closure in databases is to compute a bill of materials in a manufacturing situation. All our experiments were run for this problem. Since a bill of materials cannot be computed from a graph with cycles, only acyclic graphs were used.

A standard metric for the performance of a parallel algorithm is its *speed-up* defined as the time taken by the best serial algorithm to perform the same computation divided by the time taken by the given parallel algorithm. Since our parallel algorithms can be considered the parallelization of the well-known sequential algorithms, we have taken speed-up to be the time required for the appropriate sequential algorithm divided by the time taken by the parallelized version. Thus, our speed-up numbers for I.1 compare it with the semi-naive, and our numbers for M.1 and M.2 compare them with the Warshall algorithm, all memory-resident.

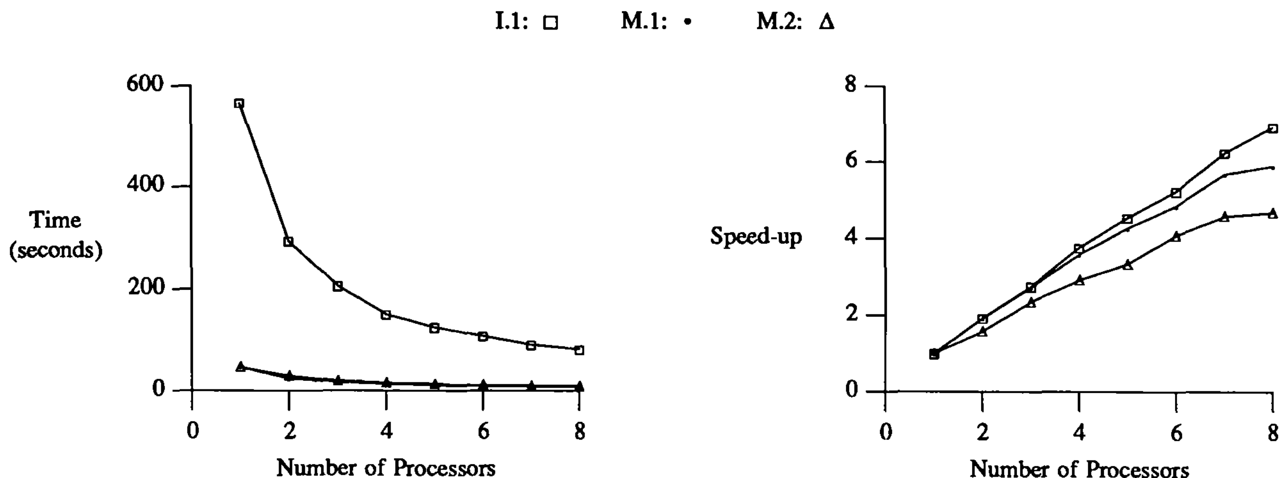
4.3 The Experiments

Figure 1 shows the performance of the three algorithms on a directed acyclic graph (DAG) with 250 nodes and an average degree of 5. The graph had 1236 arcs, and its closure had 28422 arcs. The time taken by all three algorithms follows an approximate hyperbola, as one would hope if a linear speed-up were being achieved. The two direct algorithms perform considerably better than the iterative algorithm, irrespective of the number of processors used. The poor performance of the iterative algorithm can be attributed to the large number of iterations it performs.

Figure 1 also shows the corresponding speed-up obtained. Notice that all three algorithms parallelize quite well. Comparing the three algorithms, we find that the speed-up obtained by M.1 was consistently better than that obtained by M.2, which we shall explain shortly. The speed-up obtained by I.1 was consistently better than that obtained by M.1. The reason for the better speed-up with I.1 is the longer time that it takes to perform the same computation, making even small problems appear "large" in comparison to the overheads of parallelization. Thus, by using 8 processors, we were able to obtain a speed-up of 6.9 with I.1 and a speed-up of 5.9 with with M.2.

Figure 1. Comparative performance of the three algorithms on a random DAG (nodes = 250, degree = 5)

I.1: □ M.1: • M.2: Δ



We should also remark at this stage that we are using the *fixed size* measure of speed-up[2] in which the problem size is kept fixed and the number of processors is varied. As pointed out in [9], a problem with this measure is that as the number of processors is increased, the work assigned to each processor reduces. Thus, when 8 processors are used on a 250-node graph, each processor is assigned only 31 nodes to work on. This causes severe under-utilization of the local memory available with each processor and large amplification in any load imbalance between processors due to nodes having different number of successors. The 250-node graph was just about the largest graph whose closure could be computed using a single processor. In practice, the fixed quantity often is not the problem size but rather the amount of time a user is willing to wait for an answer: given more computing power, the user expands the problem to use the available resources.

We also studied the sensitivity of the speed-up numbers to the graph being used for the computation, by varying the number of nodes and the average degree. We also carefully metered the time spent in performing different activities at each processor, and found that a large fraction of the CPU time was spent in computing and little time in "control", which includes synchronization waits for other processors. In fact, the control time is zero for I.1 since it runs with no synchronization whatsoever. The communication time was also small. See [2] for the detailed results.

In view of the computation time dominating the total time taken, for suitably large problems, we expect that an almost linear speed-up should be possible even as the number of processors grows. Also, a more complex algorithm such as M.2 is of little value since all it can do is to reduce the amount of time spent in synchronization waits while adding some overhead for a more complicated control strategy. In fact, the overhead added appears in most cases to exceed the idle wait time eliminated, resulting in performance slightly worse than M.1 in most cases.

5. RELATED WORK

Valduriez and Khoshafian proposed in [18] a hash-join-based parallelization of the semi-naive algorithm. As discussed in [2], the primary drawback of this technique is that (almost) every tuple computed must be shown to a different processor and remotely accessed. What is worse, this communication occurs prior to the elimination of duplicates, and thus could actually involve numbers of tuples far larger than those actually present in the final result. In fact, in the worst case, $O(n^3)$ tuples are remotely accessed at each iteration, resulting in large communication. Furthermore, the processors need synchronization at every iteration. We, therefore, expect our parallelization of the semi-naive algorithm to perform better than the hash-join-based parallelization.

Valduriez and Khoshafian also proposed another parallel transitive closure algorithm in [18]. This algorithm partitions the given graph among processors, and each processor computes the closure of the subgraph assigned to it. The closure of these "transitively closed" subgraphs is then computed by recursively considering two components, merging them into one component, till one final graph is obtained. A two-way merge tree, similar to the one used in parallel sorting algorithms, is used for the merging process. Valduriez and Khoshafian found that this algorithm always has inferior response time characteristics when compared to the hash-join-based parallel semi-naive algorithm. Furthermore, although this point was not addressed in [18], unless the components have a certain *convexity* property so that paths between two nodes do not criss-cross across components, computing the closure of two "transitively closed" graphs would be as hard as when the graphs were not transitively closed. Dividing a given graph into such convex components is non-trivial.

Jenq and Sahni [11] have presented a parallel implementation of the all-pairs shortest path algorithm on a NCUBE hypercube. The Jenq-Sahni algorithm computes the closure in n rounds of processing (n is the number of nodes). A broadcast and a synchronization of all the processors is required after each round. The largest reported speed up was about 15 using 32 processors on a 96 node graph. Experience with parallelizing the shortest path problem on Denelcor HEP were reported by Deo, Pang, and Lord in [8], and by Quinn and Yoo in [14]. The former reported a speed up of about 3.5 with 8 processors on a 100 node graph. The best speed up reported by the latter was about 9 with 16 processors on a 500 node graph.

6. CONCLUSIONS

There are several, not all independent, issues to be considered when devising a parallel algorithm: how well the load is balanced among the processors, how much time is spent by each processor waiting for data or a synchronization message from another processor, how much inter-processor communication is required, and what speed-up is achieved with multiple processors. In this paper we have presented three parallel transitive closure algorithms that effectively address all of these issues. We have a good load balance, high processor utilization, low communication and control overhead, and an almost linear speed-up.

7. ACKNOWLEDGEMENTS

We are grateful to Rudd Canaday, Narain Gehani, Eric Petajan, and Bill Roome for their help at different stages of this work.

2. As an alternative, a *scaled* measure of speed-up was proposed in [9] in which the problem size is increased with an increase in number of processors. We decided not to use this measure, partly because the fixed size measure makes it simpler to put our work in perspective with other work, and partly because we felt that there was no readily acceptable way for scaling up the transitive closure problem with the number of processors.

REFERENCES

[1] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590. Also in *IEEE Trans. Software Eng.* 14, 7 (July 1988), 879-885.

[2] R. Agrawal and H. V. Jagadish, "Multiprocessor Transitive Closure Algorithms", *Proc. Int'l Symp. Databases in Parallel and Distributed Systems*, Austin, Texas, Dec. 1988, 56-66.

[3] R. Agrawal, S. Dar and H. V. Jagadish, "Direct Transitive Closure Algorithms: Design and Performance Evaluation", *ACM Trans. Database Syst.*, 1989. To appear. (Preliminary version appeared as: R. Agrawal and H.V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 255-266)..

[4] R. Agrawal, S. Dar and H. V. Jagadish, "Composition of Database Relations", *Proc. IEEE 5th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1989.

[5] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations", Tech. Rept. DB-004-85, MCC, Austin, Texas, 1985.

[6] D. Bitton, H. Boral, D. J. DeWitt and W. K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations", *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 324-353.

[7] R. F. Cmelik, N. H. Gehani and W. D. Roome, "Experience with Multiple Processor Versions of Concurrent C", AT&T Bell Laboratories, Murray Hill, New Jersey, 1987. To appear in the *IEEE Trans. Software Eng.*.

[8] N. Deo, C. Y. Pang and R. E. Lord, "Two Parallel Algorithms for Shortest Path Problems", *Proc. IEEE Int'l Conf. Parallel Processing*, 1980, 244-253.

[9] J. L. Gustafson, G. R. Montry and R. E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube", *SIAM Journal on Scientific and Statistical Computing 9*, 4 (July 1988), .

[10] Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators", *Proc. 12th Int'l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 403-411.

[11] J. F. Jenq and S. Sahni, "All Pairs Shortest Paths on a Hypercube Multiprocessor", *Proc. IEEE Int'l Conf. Parallel Processing*, Aug. 1987, 713-716.

[12] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro and M. Stonebraker, "Heuristic Search in Data Base Systems", *Proc. 1st Int'l Workshop Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984, 96-107.

[13] H. Lu, "New Strategies for Computing the Transitive Closure of a Database Relation", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987.

[14] M. J. Quinn and Y. B. Yoo, "Data Structure for the Efficient Solution of the Graph Theoretic Problems on Tightly Coupled MIMD Computers", *Proc. IEEE Int'l Conf. Parallel Processing*, 1984, 431-438.

[15] W. D. Roome and M. D. P. Leland, "The Silicon Database Machine: Rationale, Design, and Results", *Proc. 5th Int'l Workshop on Database Machines*, Karuizawa, Nagano, Japan, Oct. 1987.

[16] A. Rosenthal, S. Heiler, U. Dayal and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 166-176.

[17] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices", *Proc. 1st Int'l Conf. Expert Database Systems*, Charleston, South Carolina, April 1986, 197-208.

[18] P. Valduriez and S. Khoshafian, "Parallel Evaluation of the Transitive Closure of a Database Relation", *Int'l J. of Parallel Programming 17*, 1 (Feb. 1988), 19-42.

[19] S. Warshall, "A Theorem on Boolean Matrices", *J. ACM 9*, 1 (Jan. 1962), 11-12.

Exploiting Concurrency in a DBMS Implementation for Production Systems[1]

Louiqa Raschid[2]
Department of Information Systems
School of Business and Management

Timos Sellis[2], **Chih-Chen Lin**
Department of Computer Science
and Systems Research Center

University of Maryland, College Park, MD 20742

ABSTRACT

In this paper, we tailor DBMS concurrent execution to a production system environment and investigate the resulting concurrent execution strategies for productions. This research is carried out in conjunction with a novel DBMS mechanism for testing if the left-hand side conditions of productions are satisfied. We demonstrate the equivalence of a serial and a concurrent (interleaved) execution strategy and define requirements for a correct, serializable execution. We also compare the number of possible serial and parallel execution schedules.

1. Introduction

The integration of artificial intelligence (AI) and database management (DBMS) technology has been the focus of recent research [3,4]. An important aspect of this integration is identifying functional similarities in database processing and reasoning with rules. This will allow techniques designed for use in either technology to be used in a functionally integrated environment. In this paper, we focus on tailoring DBMS concurrent-execution techniques to a production system environment. Production systems represent knowledge in the form of productions and are a good example of the rule-based reasoning paradigm. We choose the OPS5 production system [1] because of its popularity in the AI domain. Our research on concurrency in production systems has been carried out in conjunction with a novel DBMS mechanism for testing if the left-hand side conditions of productions are satisfied [6]. In this paper, we identify several potential instances for concurrency, and demonstrate the equivalence of a serial and a concurrent (interleaved) execution strategy. We specify the requirements for serializability, assuming a 2-Phase Locking scheme, and compare the number of serial and parallel execution schedules. For more details, the reader is referred to [5].

2. Production Systems

A production system is a collection of *Condition-Action* statements, called *productions*. The condition part on the left-hand side (LHS) is satisfied by data stored in a database, composed of *working memory* (WM) elements. The action part on the right-hand side (RHS) executes operations that can modify the WM. A production system repeatedly cycles through the following operations:

Match: For each production r, determine if LHS(r) is satisfied by the current WM contents. If so, add the qualifying production to the *conflict set*.

Select: Select one production out of the conflict set; if there is no candidate, halt.

Act: Perform the actions in the RHS of the selected candidate. This will change the content of the WM and, as a result, additional productions may be fired, or some productions may be deleted.

The following OPS5 production removes Mike from the WM class `Emp` if he works on the first floor, in the Toy department:

---

[1] This research was partially sponsored by the National Science Foundation under Grant CDR-85-00108 and by the University of Maryland Office of Graduate Research and Studies under a Summer Research Award.

[2] Also with the University of Maryland Institute for Advanced Computer Studies (UMIACS).

```
(p Rule1
  (Emp ↑Name Mike ↑ Salary <S> ↑Dno <D>)
  (Dept ↑Dno <D> ↑Dname Toy ↑Floor 1 ↑Mgr <M>)
  →  (remove 1))
```

The execution efficiency of OPS5 has been attributed to the Rete algorithm [2]. This algorithm exploits temporal redundancy and compiles the LHS condition elements into a binary discrimination network. The network is an inherently redundant storage structure [6]. This redundancy results in a decrease of processing efficiency of the Rete network implementation with a large database. The lack of support for universal quantification is also a drawback of the Rete implementation.

3. A Novel DBMS Implementation of OPS5

In the DBMS implementation, we treat the LHS condition of each production as a query to be evaluated against the WM classes. Each WM class is simulated using a WM relation, such as Emp(Name,Salary,Dno) as used in production Rule1. Attributes, represented by the ↑ symbol and a name, whose values are bound to constants or value ranges are equivalent to selection predicates. Variables that are common to two WM classes, e.g., <D> in classes Emp and Dept, and occur in the same production, are equivalent to a join.

We introduce a new data structure, a COND relation, which is used to link partially matched tuples from the WM relations. A COND relation is required for each WM relation (class) and will store matching tuple information for all productions that refer to that WM-class. For example, WM classes Emp and Dept occur on the LHS of the production Rule1, and are related through the join variable <D>, in this production. When a tuple <Mike,10000,D12> is input into the WM relation Emp, we store this information in the COND relation for Dept, COND-Dept. Now, when a tuple is input into the WM relation Dept with a value of Dno=D12, the link information stored in the relation COND-Dept reflects the existence of matching tuples in Emp. The COND relation obviates the need for the join operation, which would otherwise be required. A brief description of the COND relations follows; details can be found in [6].

The COND relation has the following attributes: (1) RID to record the unique production identifier. (2) Condition Element Number (CEN) to differentiate among conditions of the same production. (3) Restrictions on each attribute of the corresponding WM relation. (4) A list of Related Condition Elements (RCE), each RCE being represented by a (RID,CEN) pair. (5) A Mark bit register, comprising one bit per RCE, with a default value of zero. We illustrate the use of these attributes in the COND relation(s) through an example. Assume four relations A, B, C, D, with attributes Ai, Bi, Ci, and Di, for i=1,2,3, respectively, and the following productions (actions omited):

```
(p R1
  (A  ↑A1 <x>  ↑A2 'a'  ↑A3 <z>)
  (B  ↑B1 <x>  ↑B2 <y>  ↑B3 'b')
  (C  ↑C1 'c'  ↑C2 <y>  ↑C3 <z>)
  →  (   ))
```

```
(p R2
  (A  ↑A1 <x>  ↑A2 'a'  ↑A3 <u>)
  (B  ↑B1 <x>  ↑B2 <y>  ↑B3 'b')
  (D  ↑D1 'd'  ↑D2 <y>  ↑D3 <x>)
  →  (   ))
```

There will be four COND relations: COND-A, COND-B, COND-C, and COND-D. These relations are related to each other by variables <x>, <y> and <z> occurring on the LHS of productions R1 and R2. Variables capture linkages between tuples of the WM relations A, B, C, and D that must be stored in the COND relations. The initial contents of these COND relations describe the LHS conditions of the productions. In the COND relations that are shown below, these initial tuples have their Mark bit values set to 0.

The RCE list indicates which conditions (involving other WM relations) of the same production are affected by insertions or deletions in the current relation being examined. A tuple in a COND relation with at least one Mark bit set is called a *matching-pattern*. There is one Mark bit for each RCE, which if set indicates that the matching–pattern is created by a tuple that satisfies the corresponding condition element. This implies that there is already some tuple(s) in a (related) WM relation having the property of the matching-pattern and therefore it can be joined with tuples in the current WM relation. Thus, when a tuple is inserted later in the current WM relation which matches that pattern, we know immediately

that there is a match without having to examine the other WM relation(s).

COND-A

RID	CEN	A1	A2	A3	RCE	Mark bit
R1	1	<x>	'a'	<z>	(B,2),(C,3)	00
R2	1	<x>	'a'	<u>	(B,2),(D,3)	00
R1	1	4	'a'	<z>	(B,2),(C,3)	10
R2	1	4	'a'	<u>	(B,2),(D,3)	10
R1	1	<x>	'a'	8	(B,2),(C,3)	01
R1	1	4	'a'	8	(B,2),(C,3)	11
R2	1	4	'a'	<u>	(B,2),(D,3)	01
R2	1	4	'a'	<u>	(B,2),(D,3)	11

COND-B

RID	CEN	B1	B2	B3	RCE	Mark bit
R1	2	<x>	<y>	'b'	(A,1),(C,3)	00
R2	2	<x>	<y>	'b'	(A,1),(D,3)	00
R1	2	<x>	7	'b'	(A,1),(C,3)	01
R1	2	4	<y>	'b'	(A,1),(C,3)	10
R1	2	4	7	'b'	(A,1),(C,3)	11
R2	2	4	<y>	'b'	(A,1),(D,3)	10
R2	2	4	7	'b'	(A,1),(D,3)	01
R2	2	4	7	'b'	(A,1),(D,3)	11

COND-C

RID	CEN	C1	C2	C3	RCE	Mark bit
R1	3	'c'	<y>	<z>	(A,1),(B,2)	00
R1	3	'c'	7	<z>	(A,1),(B,2)	01
R1	3	'c'	<y>	8	(A,1),(B,2)	10
R1	3	'c'	7	8	(A,1),(B,2)	11

COND-D

RID	CEN	D1	D2	D3	RCE	Mark bit
R2	3	'd'	<y>	<x>	(A,1),(B,2)	00
R2	3	'd'	7	4	(A,1),(B,2)	01
R2	3	'd'	<y>	4	(A,1),(B,2)	10
R2	3	'd'	7	4	(A,1),(B,2)	11

When a tuple is inserted into, say, WM relation A, two tasks are performed. First, we have to examine the existing tuples in COND-A to determine if this new A tuple satisfies any productions; this would be the case if COND-A already had a matching-pattern tuple with appropriate linkages (for the common variables) and with a RID value of R1. A matching-pattern tuple with both Mark bits set indicates the existence of tuples in the WM relations B and C that together would satisfy the production R1. The second task is to store the information on linkages through the common variables, i.e., how tuples in the WM class A interact with B and C, or B and D, as indicated on the LHS of the productions. This information is stored in the form of matching-patterns in COND-B and COND-C.

Suppose that we insert the tuples B(4,7,b), C(c,7,8), A(4,a,8) and D(d,7,4) in that order. Several tuples will be inserted into the COND relations. Tuple B(4,7,b) will insert two tuples in COND-A (corresponding to R1 and R2), one tuple in COND-C for R1 and one tuple in COND-D for R2. Tuple C(c,7,8) will insert two tuples in COND-A for R1. The second tuple will have both Mark bits set indicating the presence of WM elements from B and C simultaneously satisfying R1. One tuple will also be inserted into COND-B for R1. Tuple A(4,a,8) will match with a matching-pattern for R1 with both bits set; this implies there now exist tuples from A, B and C satisfying R1 and this pattern must be placed in the conflict set. This tuple also introduces three tuples in COND-B, two tuples in COND-C, and two in COND-D. Finally, tuple D(D,7,4) will match a matching-pattern in COND-D for R2 with both bits set; this pattern for R2, is placed in the conflict set. This tuple also inserts two tuples in COND-A and in COND-B. The contents of the relations are shown in the above tables. Details of the algorithm described above are in [6].

4. Processing Applicable Productions

In our DBMS implementation, all the tasks associated with the execution of a candidate production (from the conflict set) will be defined as a single (complex) transaction. Within this transaction, the first task is retrieval from the WM relations; the matching-pattern tuple for a selected production does not store pointers to, or identifiers of, the actual tuples of the WM relations satisfying this production. The attribute values in each matching-pattern will provide the selection criterion that must be applied to the WM relations. The next task is executing the corresponding RHS actions. These actions represent changes

to the WM classes and include insertions, deletions and updates of the WM elements. RHS actions that add or delete tuples from the WM relations trigger the insertion or deletion algorithm, respectively. Both algorithms will update the conflict set and also execute the maintenance tasks that update the COND relations. The deletion algorithm performs searches similar to the insertion algorithm; the difference is that it deletes matching-patterns from the conflict set, and resets the Mark bits in the COND relations. An update is equivalent to a delete followed by an insert, and triggers both algorithms. Conceptually, execution of the production (transaction) completes after updating the WM relations, the conflict set and the COND relations.

In the Rete network implementation of OPS5, productions placed in the conflict set are executed in a serial order. In each cycle, a single production is selected and its RHS actions are then executed; this may result in changes to the WM. In the next match phase, these updates to the WM are propagated through the discrimination net. Consequently, productions in the conflict set may be deleted, or productions may be added. When several combinations of the WM elements satisfy a single production, the Rete implementation stores each combination as a separate instantiation of the production in the conflict set, and each is executed independently. However, in our implementation, a set-oriented selection will retrieve all possible combinations of the WM tuples satisfying each LHS condition. Thus, a selected production can be simultaneously applied to all possible combinations of the WM tuples, that are retrieved. This leads to potential *intra-production* concurrency (within a single production) when executing productions.

Similarly, combinations of the WM elements could simultaneously satisfy different productions. This leads to potential *inter-production* concurrency for executing the entire conflict set. In the next section, we explore a concurrent execution strategy that exploits both forms of concurrency.

4.1. Equivalence of a Serial and Parallel Execution

Given an initial set Ψ_1 of transactions, each of which corresponds to an already satisfied production in the conflict set, we compare the serial execution of these transactions, e.g., in OPS5, with their interleaved execution in a concurrent environment. The serializability criterion is used to show the equivalence of both execution strategies.

In a serial production system, in each step i, a single transaction, T_i is *arbitrarily* selected from the conflict set and applied (**Select** and **Act**). We use the term "arbitrarily", because the OPS5 conflict resolution strategies are syntactic. Subsequently, the production system will determine (**Match**), if, as a result of applying T_i, some other transactions in this conflict set are no longer applicable; if so, these transactions will be deleted from the set. Let the set of transactions deleted in step i be del_i. Also as a result of applying T_i, the production system will determine (**Match**) if some additional transactions are now applicable as well. Let the set of transactions added in step i be add_i. The new set of candidate transactions in step $i+1$ is $\Psi_{i+1} = \Psi_i - \{T_i\} - del_i \cup add_i$. This process will continue until in step F, the set Ψ_F is empty.

The selection of each T_i is arbitrary; thus, it is entirely possible that in step 2, T_2 is selected from the set $\Psi_1 - \{T_1\} - del_1$ which is the set $\Psi_2 - add_1$. In other words, T_2 could also be selected from the initial set Ψ_1 and not from the added set of transactions add_1. Similarly, in subsequent steps i, T_i can be selected from the set $\Psi_1 - \bigcup_{j=1}^{i-1} (\{T_j\} \cup del_j)$ which is the same as the set $\Psi_i - \bigcup_{j=1}^{i-1} add_j$. If the selection is as described, then after some f_1 steps the serial production system will have executed a sequence of f_1 transactions $T_1, T_2, \ldots, T_{f_1}$, where each T_i happens to be an element of the initial set Ψ_1. After step f_1, all transactions in Ψ_1 are either executed or deleted and the set of applicable transactions for step (f_1+1), Ψ_{f_1+1} is the set $\bigcup_{j=1}^{f_1} add_j$, i.e., all the transactions added in the f_1 previous steps, which were not selected previously. In step (f_1+1), T_{f_1+1} is chosen from this set Ψ_{f_1+1}.

Given this same initial set Ψ_1, a concurrent execution strategy would interleave the execution of this set of transactions. If an appropriate protocol is used, and the resulting schedule is serializable, then it must be equivalent to some serial schedule $T_1, T_2, \ldots$, etc., where each T_i must be from the initial set Ψ_1.

In other words, the concurrent production system will execute an equivalent serial schedule which will be the same as some serial schedule arbitrarily selected by the serial production system.

4.2. Concurrent Execution with the DBMS Implementation

In a DBMS environment, a transaction commits all its changes after it has terminated its execution normally. Once the transaction commits, these changes are physically made in the database. In a concurrent environment, appropriate locks must be obtained to satisfy the following: First, the interleaved execution of a set of productions must maintain consistency of the database. i.e., two transactions that update the same WM relation must be serializable. Second, transactions that are inter-related and affect each other's execution, i.e., transactions that delete each other's matching-pattern tuples from the conflict set, must interact correctly. For example, when a transaction T_i executes, the selected commit point must be chosen to enforce a delay in the execution (and commit) of the transactions in the set del_i. i.e., transactions that are deleted as a result of previously applying T_i. Transactions in this set must either not be executed or, if executed, their changes must not be committed to the database.

A transaction (production) is *positively dependent* on a WM relation if the LHS of the production is satisfied by the existence of some specific tuples of a WM relation. A transaction is *negatively dependent* on a WM relation if it is satisfied by the absence of some specific tuples. A transaction is *independent* of a WM relation if it is unaffected by the existence or absence of specific tuples. In the current definition of the OPS5 language, the RHS actions of a production can only delete or update tuples from the WM relations on which it is positively dependent. However, a transaction can insert tuples into any WM relation. The production R3 shown below, is positively dependent on the WM relations A and B. It is negatively dependent on C, and is independent of D.

```
(p R3
   (A  ↑A1 <x>  ↑A2 'a'  ↑A3 <z>)
   (B  ↑B1 <x>  ↑B2 <y>  ↑B3 'b')
   ¬ (C  ↑C1 'c'  ↑C2 <y>  ↑C3 <z>)
 →  ( (remove 2)
      (make (D  ↑D1 'd'  ↑D2 'd'  ↑D3 'd') )
```

The following requirements must be met to produce a serializable execution schedule:

(1) Each transaction T_i must obtain an **R lock** for specific tuples of the WM relations it positively depends on and are used to satisfy the LHS of the production. This prevents the deletion or update of these tuples by other transactions. Note that if any of these retrievals returns an empty set of tuples, then the transaction is aborted. This may happen if T_i is in the delete set of a previously committed transaction.

(2) Each transaction T_i must obtain an **R lock** for the entire WM relations(s) it negatively depends on. It must subsequently verify that there are no tuples satisfying the search criterion for this negative dependency. Note that if the verification fails, T_i is aborted since it is in some delete set.

(3) Each transaction T_i must obtain a **W lock** for specific tuples of the WM relation that it deletes or updates. These would necessarily be tuples for which it would have previously obtained an **R lock**; consequently, these tuples must exist and could not have been deleted. However, it could wait indefinitely in case of a deadlock.

(4) Each transaction T_i must obtain a **W lock** for an entire WM relation if there is an insertion of tuples. Again, an indefinite wait implies a deadlock situation.

(5) Once these locks are obtained, transaction T_i can update the matching-pattern tuples in the conflict, modify the WM relations and update the COND relations. It then commits all of its changes and releases all locks.

In [5], we have proved the correctness of our requirements for serializability by examining all cases of inter-dependencies among productions (through the WM relations).

Consider the case of transactions T_i and T_j that are positively dependent on the WM relation R. T_i deletes specific tuples from R and this may affect the execution of T_j. If T_j obtains an **R lock** for the WM relation R before T_i attempts to obtain a **W lock**, then, T_j will precede T_i in the equivalent serial execution and the database will be consistent.

If T_i obtains a **W lock** on the WM relation R (and thus, completes execution) before T_j requests an **R lock**, then T_j may be in the set del_i, so its execution must be delayed until after the update or delete from R. Changes made to R trigger the maintenance process and propagate changes to the COND relations. The maintenance process can potentially delete the matching-pattern tuple for T_j from the conflict set. For this reason, T_i must not commit and release its locks on the WM relations until the maintenance process completes.

If the matching-pattern tuple corresponding to T_j is unaffected or is deleted before T_j starts execution, then, no further action is required. If T_j has already started execution, it will be delayed since it will not be able to obtain a **R lock** until T_i releases its **W lock** on the tuples of R. Now, even if the matching-pattern tuple for T_j is deleted, T_j will still be executed. However, T_j will not be able to process those tuples of R that have already been deleted by T_i so the database will remain consistent.

It is also possible that both T_i and T_j delete or update tuples from R, and that T_i is in the set del_j and vice versa. This could lead to a deadlock of the two transactions.

4.3. Special Case of Intra–Production Concurrency

Maintaining serializability in the case of intra–production concurrency requires further attention. Consider the following sets of tuples satisfying R3:

{A(4,a,8),B(4,7,b)} and {A(4,a,38),B(4,7,b)}

If the tuple B(4,7,b) was deleted by R3 (with the first set of tuples), then it would affect the subsequent execution of R3 (with the second set of tuples). However, if the following combinations of tuples were to satisfy R3 simultaneously:

{A(4,a,8),B(4,7,b)} and {A(4,a,8),B(4,9,b)}

then, execution of R3 with each combination of tuples in any sequence is always serializable.

One solution is to treat each instantiation of a production and one combination of the WM tuples as a nested transaction within the transaction representing the production and all combinations of tuples. Nested transactions, however, are expensive to support and this solution may even negate the advantages of intra–production concurrency. A preferable solution would be for each transaction to count the number of times it attempts to delete a tuple. If it attempts to do so more than once, then there is clearly inconsistency in the execution of the transaction. Similarly, if a transaction is negatively dependent on a WM relation and if the transaction also inserts tuples into the same relation, then there is a possibility for inconsistent execution within the transaction. These issues require further consideration.

4.4. Estimating the Number of Execution Schedules

The benefits of concurrent execution can be measured in several ways. First, the number of operations that must execute in a non–interleaved fashion reflects the time of execution. In the best case, neglecting the locking overhead, this will be proportional to the maximum number of updates to any WM relation or COND relation. In the worst case, this will reduce to the time taken for a serial execution.

A second measure involves an estimation of the number of possible choices for selecting a transaction in any step and the resulting number of different execution schedules in each case. This measure is of interest since different execution schedules result in different final states of the knowledge base. See [5] for a detailed discussion.

A serial system is not constrained to execute in the fashion that we just described, where the initial sequence of transactions is limited to be in the initial set Ψ_1. In fact, in each step i, all transactions in the set Ψ_i, which was previously defined, are candidates for execution. If we use $|\Psi_i|$ to denote the size of the set Ψ_i, then in each step i, $|\Psi_i|$ is the number of available choices for selecting a transaction. Let

NS_f represent the number of choices of possible serial schedules, corresponding to a sequence of f transactions, $T_1, T_2, \ldots, T_f$. Then NS_f is as follows:

$$NS_f = \prod_{i=1}^{f} |\Psi_i| \quad \text{where} \quad |\Psi_{i+1}| = |\Psi_i| - 1 - |del_i| + |add_i|.$$

We now estimate the number of possible execution schedules for the concurrent PS; we can measure the number of equivalent serial schedules, $NC(eq)_f$, for a sequence of f transactions. The final state of the knowledge base is determined by the equivalent serial schedule. Let $|\Psi_1|$ be the number of transactions in the initial set and let f_1 of these transactions be actually executed, i.e., the LHS of the rest are not satisfied. At each step i, the number of choices for the concurrent system is $\left(|\Psi_1| - i - |del_i|\right)$. After the f_1 transactions are executed we use the operations executed by them, collectively, to form a new initial set Ψ_{f_1+1} of added transactions. THen, $NC(eq)_f$ is as follows

$$NC(eq)_f = \prod_{i=0}^{f_1}\left(|\Psi_1| - i - |del_i|\right) \times \prod_{i=0}^{f_2}\left(|\Psi_{f_1+1}| - i - |del_{f_1+i}|\right) \times \cdots$$

where, $\Psi_{f_1+1} = \bigcup_{i=0}^{f_1} add_i$ is a new initial set of transactions made applicable after executing the set Ψ_1 and $f = f_1 + f_2 + \cdots$:

5. Conclusions

In this paper we studied the problem of concurrent execution of a set of applicable productions in a DBMS implementation of a production system. We showed the equivalence of a serial and an interleaved execution. The latter may improve execution efficiency and in the worst case it will be no worse than a serial strategy (neglecting any locking overhead). Assuming 2-Phase Locking, we specified the requirements for a correct serializable execution. We also estimated the number of possible execution schedules for a serial production system compared to the concurrent one.

6. References

[1] Forgy, C.L., OPS5 User's Manual, Tech. Report CMU-CS-81-135, Carnegie-Mellon University (1981).

[2] Forgy, C.L., Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence* (19) (1982).

[3] Kershberg, L., Ed., *Expert Database Systems: Proc. From the First International Workshop,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1986).

[4] Kershberg, L., Ed., *Expert Database Systems: Proc. From the First International Conference.* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1987).

[5] Raschid, L., Sellis, T. and Lin, C-C., Exploiting Concurrency in a DBMS Implementation for Production Systems, *International Symposium on Databases in Parallel and Distributed Systems,* Austin, TX (1988).

[6] Sellis, T., Lin, C-C., and Raschid, L., Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms, *Proc. of ACM-SIGMOD,* Chicago, IL (1988).

Checkpointing and Recovery in Distributed Database Systems

Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

1. Introduction

The need for a recovery mechanism in a database system is well understood. In spite of powerful database integrity checking mechanisms which detect errors and undesirable data, it is possible that some erroneous data may be included in the database. Furthermore, even with a perfect integrity checking mechanism, failures of hardware and/or software at the processing sites may destroy consistency of the database. In order to cope with those errors and failures, database systems provide recovery mechanisms, and checkpointing is a technique frequently used in database recovery mechanisms.

The goal of checkpointing in database systems is to read and return current values of the data objects in the system. A checkpointing procedure would be very useful, if states it returns are guaranteed to be consistent. In a bank database, for example, a checkpoint can be used to audit all of the account balances (or the sum of all account balances). It can also be used for failure detection; if a checkpoint produces an inconsistent system state, one assumes that an error has occurred and takes appropriate recovery measures. In case of a failure, previous checkpoints can be used to restore the database. Checkpointing must be performed so as to minimize both the costs of performing checkpoints and the costs of recovering the database. If the checkpoint intervals are very short, too much time and resources are spent in checkpointing; if these intervals are long, too much time is spent in recovery.

For a checkpoint process to return a meaningful result (e.g., a consistent state), the individual read steps of the checkpoint must not be permitted to interleave with the steps of other transactions; otherwise an inconsistent state can be returned even for a correctly operating system. However, since checkpointing is performed during normal operation of the system, this requirement of non-interference will result in poor performance. For example, in order to generate a commit consistent checkpoint for recovery, user transactions may suffer a long delay waiting for active transactions to complete and the updates to be reflected in the database [CHA85]. A transaction is said to be *reflected* in the database if the values of data objects represent the updates made by the transaction. It is highly desirable that transactions are executed in the system concurrently with the checkpointing process. In distributed systems, the desirable properties of non-interference and global consistency make checkpointing more complicated because we need to consider coordination among autonomous sites of the system.

Recently, the possibility of having a checkpointing mechanism that does not interfere with transaction processing, and yet achieves consistency of the checkpoints, has been studied [CHA85, FIS82, SON86b]. The motivation for non-interfering checkpointing is to improve system availability, that is, the system must be able to execute user transactions concurrently with the checkpointing process. The principle behind non-interfering checkpointing mechanisms is to create a diverged computation of the system such that the checkpointing process can view a consistent state that could result by running to completion all of the transactions that are in progress when the checkpoint begins, instead of viewing a consistent state that actually occurs by suspending further transaction execution. Figure 1 shows a diverged computation during checkpointing.

Non-interfering checkpointing mechanisms, however, may suffer from the fact that the diverged computation needs to be maintained by the system until all of the transactions, that are in progress when the checkpoint begins, come to completion. This may not be a major concern for a database system in which all the transactions are relatively short. However, for database systems with many long-lived transactions, checkpointing of this kind might not

This work was supported in part by the Office of Naval Research under contract number N00014-86-K-0245, by the Department of Energy under contract number DEFG05-88-ER25063, and by the Federal Systems Division of IBM Corporation under University Agreement WF-159679.

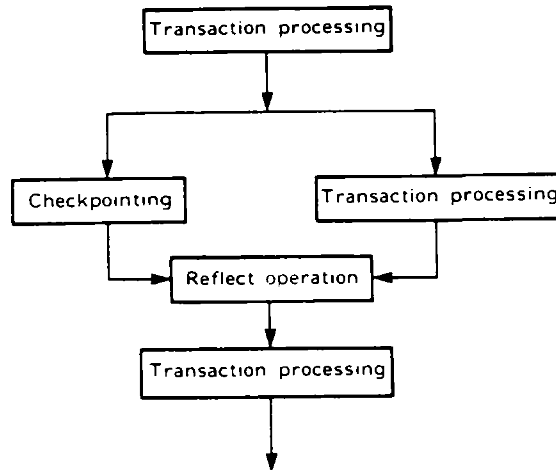


Fig. 1. Diverged computation for checkpointing

be practical for the following reasons:

(1) It takes a long time to complete a non-interfering checkpoint, resulting in high storage and processing overhead.

(2) If a crash occurs before the results of a long-lived transaction are included in the checkpoint, the system must re-execute the transaction from the beginning, wasting all the resources used for the initial execution of the transaction.

In the rest of this paper, we briefly discuss one approach for checkpointing which efficiently generates a consistent database state, and its adaptation for systems with long-lived transactions. Given our space limitations, our objective is to intuitively explain this approach and not to provide details. The details are given in separate papers [SON86b, SON88].

2. Non-interfering Approach

In order to make each checkpoint consistent, updates of a transaction must either be included in the checkpoint completely or not at all. To achieve this, transactions are divided into two groups according to their relations to the current checkpoint: *after-checkpoint transactions* (ACPT) and *before-checkpoint transactions* (BCPT). Updates belonging to BCPT are included in the current checkpoint while those belonging to ACPT are not included. In a centralized database system, it is an easy task to separate transactions for this purpose. However, it is not easy in a distributed environment. To separate transactions in a distributed environment, a special timestamp which is globally agreed upon by the participating sites is used. This special timestamp is called the *Global Checkpoint Number* (GCPN), and it is determined as the maximum of the Local Checkpoint Numbers (LCPN) through coordination of all participating sites.

An ACPT can be reclassified as a BCPT if its timestamp requires that the transaction must be executed before the current checkpoint. This is called the *conversion* of transactions. The updates of a converted transaction are included in the current checkpoint.

Two types of processes are involved in the checkpoint execution: *checkpoint coordinator* (CC) and *checkpoint subordinate* (CS). The checkpoint coordinator starts and terminates the global checkpointing process. Once a checkpoint has started, the coordinator does not issue the next checkpoint request until the first one has terminated. At each site, the checkpoint subordinate performs local checkpointing by a request from the coordinator. We assume that site m has a local clock LC_m which is manipulated by the clock rules of Lamport[LAM78].

Execution of a checkpoint progresses as follows. First, the checkpoint coordinator broadcasts a Checkpoint Request Message with a timestamp LC_{CC}. The local checkpoint number of the coordinator is set to LC_{CC}. The coordinator sets the Boolean variable CONVERT to false, and marks all transactions at the coordinator site with timestamps not greater than $LCPN_{CC}$ as BCPT.

On receiving a Checkpoint Request Message, the local clock of site m is updated and $LCPN_m$ is set to LC_m. The checkpoint subordinate of site m replies to the coordinator with $LCPN_m$, and sets the Boolean variable CONVERT to false. The coordinator broadcasts the GCPN which is determined as the maximum of the local checkpoint numbers.

In all sites, after the LCPN is fixed, all transactions with timestamps greater than the LCPN are marked as temporary ACPTs. If a temporary ACPT updates any data objects, those data objects are copied from the database to the buffer space of the transaction. When a temporary ACPT commits, updated data objects are not stored in the database as usual, but are maintained as *committed temporary versions* (CTV) of the data objects. The data manager in each site maintains permanent and temporary versions of data objects. When a read request is made for a data object which has committed temporary versions, the value of the latest committed temporary version is returned. When a write request is made for a data object which has committed temporary versions, another committed temporary version is created for it rather than overwriting the previous committed temporary version.

When the GCPN is known, each checkpointing process compares the timestamps of the temporary ACPTs with the GCPN. Transactions that satisfy the following condition become BCPTs; their updates are reflected in the database, and are included in the current checkpoint.

$$\text{LCPN} < \text{timestamp}(T) \leq \text{GCPN}$$

The remaining temporary ACPTs are actual ACPTs; their updates are not included in the current checkpoint. These updates are included in the database after the current checkpointing has been completed. After the conversion of all eligible BCPTs, the checkpointing process sets the Boolean variable CONVERT to true. Local checkpointing is executed by saving the state of data objects when there is no active BCPT and the variable CONVERT is true. After the execution of local checkpointing, the values of the latest committed temporary versions are used to replace the values of data objects in the database. Then, all committed temporary versions are deleted. Execution sequences of two different types of transactions are shown in Figure 2.

As an example, consider a three-site distributed database system. Assume that $LC_{CC} = 5$, $LC_{CS1} = 3$, and $LC_{CS2} = 8$. CC sets its LCPN as 5, and broadcasts a checkpoint request message. On receiving the request message, LCPN of each CS is set to 6 and 9, respectively. After another round of message exchange, the GCPN of the current checkpoint will be set to 9 by the CC and will be known to each CS. If transaction T_i with the timestamp 7 was initiated at the site of CS1, it is treated as an ACPT. All updates by T_i are maintained as CTV. However, when GCPN is known, T_i will be converted to a BCPT and its updates will be included in the current checkpoint.

3. Adaptive Approach for Long-lived Transactions

It can be shown that a non-interfering checkpointing process will terminate in a finite time by selecting an appropriate concurrency control mechanisms [SON87]. However, the amount of time necessary to complete one

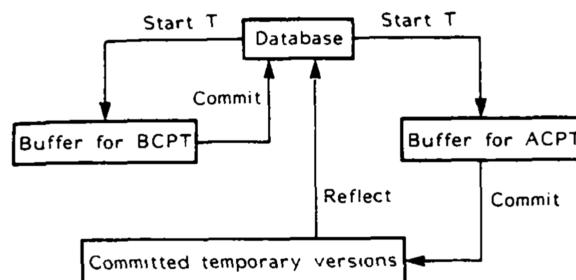


Fig. 2. Execution sequences of ACPT and BCPT

checkpoint cannot be bound in advance; it depends on the execution time of the longest transaction classified as a BCPT. Therefore the storage and processing cost of the checkpointing algorithm may become unacceptably high if a long-lived transaction is included in the set of BCPTs. We briefly discuss the practicality of non-interfering checkpoints in the next section. In addition, all resources used for the execution of a long-lived transaction would be wasted if the transaction must be re-executed from the beginning due to a system failure.

These problems can be solved by using an adaptive checkpointing approach. We assume that each transaction must carry a flag with it, which tells whether it is a normal transaction or a long-lived transaction. The threshold to separate two types of transactions is application-dependent. In general, transactions that need hours of execution can be considered as long-lived transactions.

An adaptive checkpointing procedure operates in two different modes: *global mode* and *local mode*. The global mode of operation is basically the procedure sketched in the previous section. In the local mode of operation, a mechanism is provided to save consistent states of a transaction so that the transaction can resume execution from its most recent checkpoint.

As in the previous approach, the checkpoint coordinator begins checkpointing by sending out Checkpoint Request Messages. Upon receiving this request message, each site checks whether any long-lived transaction is being executed at the site. If so, the site reports it to the coordinator, instead of sending its LCPN. Otherwise (i.e., no long-lived transaction in the system), non-interfering checkpointing begins. If any site reports the existence of a long-lived transaction, the coordinator switches to the local mode of operation, and informs each site to operate in the local mode. The checkpoint coordinator sends Checkpoint Request Messages to each site at an appropriate time interval to initiate the next checkpoint in the global mode. This attempt will succeed if there is no active long-lived transaction in the system.

In the local mode of operation, each long-lived transaction is checkpointed separately from other long-lived transactions. The coordinator of the long-lived transaction initiates the checkpoint by sending Checkpoint Request Messages to its participants. A checkpoint at each site saves the local state of a long-lived transaction. For satisfying the correctness requirement, a set of checkpoints, one per each participating site of a global long-lived transaction, should reflect the consistent state of the transaction. Inconsistent set of checkpoints may result from a non-synchronized execution of associated checkpoints. For example, consider a long-lived transaction T being executed at sites P and Q, and a checkpoint taken at site P at time X, and at site Q at time Y. If a message M is sent from P after X, and received at Q before Y, then the checkpoints would save the reception of M but not the sending of M, resulting in a checkpoint representing an inconsistent state of T.

We use message numbers to achieve consistency in a set of local checkpoints of a long-lived transaction. Messages that are exchanged by participating transaction managers of a long-lived transaction contain message number tags. Transaction managers of a long-lived transaction use monotonically increasing numbers in the tag of its outgoing messages, and each maintains the tag numbers of the latest message it received from other participants. On receiving a checkpoint request, a participant compares the message number attached to the request message with the last tag number it received from the coordinator. The participant replies OK to the coordinator and executes local checkpointing only if the request tag number is not less than the number it has maintained. Otherwise, it reports to the coordinator that the checkpoint cannot be executed with that request message.

If all replies from the participants arrive and are all OK, the coordinator decides to make all local checkpoints permanent. Otherwise, the decision is to discard the current checkpoint, and to initiate a new checkpoint. This decision is delivered to all participants. After a new permanent checkpoint is taken, any previous checkpoints will be discarded at each site.

4. Performance Considerations

There are two performance measures that can be used in discussing the practicality of non-interfering checkpointing: extra storage and extra workload required. The extra storage requirement of the algorithm is simply the CTV file size, which is a function of the expected number of ACPTs of the site, the number of data objects updated by a typical transaction, and the size of the basic unit of information:

$$\text{CTV file size} = N_A \times (\text{number of updates}) \times (\text{size of the data object})$$

where N_A is the expected number of ACPT of the site.

The CTV file may become unacceptably large if N_A or the number of updates becomes very large. Unfortunately, they are determined dynamically from the characteristics of transactions submitted to the database system, and hence cannot be controlled. Since N_A is proportional to the execution time of the longest BCPT at the site, it

would become unacceptably large if a long-lived transaction is being executed when a checkpoint begins at the site. The only parameter we can change in order to reduce the CTV file size is the granularity of a data object. The size of the CTV file can be minimized if we minimize the size of the data object. By doing so, however, the overhead of normal transaction processing (e.g., locking and unlocking, deadlock detection, etc) will be increased. Also, there is a trade-off between the degree of concurrency and the lock granularity[RIE79]. Therefore the granularity of a data object should be determined carefully by considering all such trade-offs, and we cannot minimize the size of the CTV file by simply minimizing the data object granularity.

There is no extra storage requirement in intrusive checkpointing mechanisms[DAD80, KUS82, SCH80]. However this property is balanced by the cases in which the system must block the execution of an ACPT or abort transactions because of the checkpointing process.

The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT (move data objects to the CTV file), (3) reflecting the CTV file (move committed temporary versions from the CTV file to the database), and (4) clearing the CTV file when the reflect operation is finished. Among these, the workload for (2) and (3) dominates the total extra workload. As in the estimation of extra storage, the workload for (2) and (3) is determined by the number of ACPTs and the number of updates. Therefore, as long as the values of these variables can be maintained below a certain threshold level, non-interfering checkpointing would not severely degrade the performance of the system. A detailed discussion of the practicality of non-interfering checkpointing is given in [SON86b].

5. Site Failures

So far, we assumed that no failure occurs during checkpointing. This assumption can be justified if the probability of failures during a single checkpoint is extremely small. However, it is not always the case, and we now consider the method to make the algorithm resilient to failures.

During the global mode of operation, the checkpointing process is insensitive to failures of subordinates. If a subordinate fails before the broadcast of a Checkpoint Request Message, it is excluded from the next checkpoint. If a subordinate does not send its LCPN to the coordinator, it is excluded from the current checkpoint. When the site recovers, the recovery manager of the site must determine the GCPN of the latest checkpoint. After receiving information about transactions which must be executed for recovery, the recovery manager brings the database up to date by executing all transactions whose timestamps are not greater than the latest GCPN. Other transactions are executed after the state of the data objects at the site is saved by the checkpointing process.

An atomic commit protocol guarantees that a transaction is aborted if any participant fails before it sends a Precommit message to the coordinator. Therefore, site failures during the execution of the algorithm cannot affect the consistency of checkpoints because each checkpoint reflects only the updates of committed BCPTs.

In the local mode of operation, the failure of a participant prevents the coordinator from receiving OKs from all participants, or prevents the participants from receiving the decision message from the coordinator. However, because a transaction is aborted by an atomic commit protocol, it is not necessary to make checkpointing robust to failures of participants.

The algorithm is, however, sensitive to failures of the coordinator. In particular, if the coordinator crashes during the first phase of the global mode of operation (i.e., before the GCPN message is sent to subordinates), every transaction becomes an ACPT, requiring too much storage for committed temporary versions.

One possible solution to this involves the use of a number of *backup* processes; these are processes that can assume responsibility for completing the coordinator's activity in the event of its failure. These backup processes are in fact checkpointing subordinates. If the coordinator fails before it broadcasts the GCPN message, one of the backups takes control. A similar mechanism is used in SDD-1 [HAM80] for reliable commitment of transactions.

6. Recovery

A recovery from site crashes is called a *site recovery*. The complexity of a site recovery varies in distributed database systems according to the failure situation[SCH80]. If the crashed site has no replicated data objects and if all recovery information is available at the crashed site, local recovery is sufficient. Global recovery is necessary because of failures which require the global database to be restored to some earlier consistent state. For instance, if the transaction log is partially destroyed at the crashed site, local recovery cannot be executed to completion.

When a global recovery is required, the database system has two alternatives: a *fast* recovery and a *complete* recovery. A fast recovery is a simple restoration of the latest global checkpoint. Since each checkpoint is globally

consistent, the restored state of the database is assured to be consistent. However, all transactions committed during the time interval from the latest checkpoint to the time of crash would be lost. A complete recovery is performed to restore as many transactions that can be redone as possible. The trade-offs between the two recovery methods are the recovery time and the number of transactions saved by the recovery.

Quick recovery from failures is critical for some applications of distributed database systems which require high availability (e.g., ballistic missile defense or air traffic control). For those applications, the fate of the mission, or even the lives of human beings, may depend on the correct values of the data and the accessibility to it. Availability of a consistent state is of primary concern for those applications, not the most up-to-date consistent state. If a simple restoration of the latest checkpoint could bring the database to a consistent state, it may not be worthwhile to spend time in recovery by executing a complete recovery to recover some of the transactions.

For the applications in which each committed transaction is so important that the most up-to-date consistent state of the database is highly desirable, or if the checkpoint intervals are large such that a lot of transactions cannot be recovered by a fast recovery, a complete recovery is appropriate. The cost of a complete recovery is the increased recovery time which reduces availability of the database. Searching through the transaction log is necessary for a complete recovery. The property that each checkpoint reflects all updates of transactions with earlier timestamps than its GCPN is useful in reducing the amount of searching, because the set of transactions whose updates must be redone can be determined by a simple comparison of the timestamps of transactions with the GCPN of the checkpoint. Complete recovery mechanisms based on the special timestamp of checkpoints (e.g., GCPN) have been proposed in [KUS82, SON86a].

After site recovery is completed using either a fast recovery procedure or a complete recovery procedure, the recovering site checks whether it has completed local-mode checkpointing for any long-lived transactions. If any local-mode checkpoint is found, those transactions can be restarted from the saved checkpoints. In this case, the coordinator of the transaction requests all participants to restart from their checkpoints if and only if they all are able to restart from that checkpoint. The coordinator decides whether to restart the transaction from the checkpoint or from the beginning based on responses from the participants, and sends the decision message to all participants. Such a two-phase recovery protocol is necessary to maintain consistency of the database in case of damaged checkpoints at the failure site. A transaction will be restarted from the beginning if any participant is not able to restore the checkpointed state of the transaction for any reason.

7. Concluding Remarks

During normal operation, checkpointing is performed to save information for recovery from failure. For better recoverability and availability of distributed databases, checkpointing must allow construction of a globally consistent database state without interfering with transaction processing. Site autonomy in distributed database systems makes checkpointing more complicated than in centralized systems.

The role of the checkpointing coordinator is simply that of getting a uniformly agreed GCPN. Apart from this function the coordinator is not essential to the operation of the proposed algorithm. If a uniformly agreed GCPN can be made known to individual sites, then the centralized nature of the coordinator can be eliminated. One way to achieve this is to preassign the clock values at which checkpoints will be taken. For example, we may take checkpoints at clock values as a multiple of 1000. Whenever the local clock of a site crosses a multiple of this value, checkpointing can begin.

If the frequency of checkpointing is related to load conditions and not necessarily to clock values, then the preassigned GCPN will not work as well. In this case a node will have to assume the role of the checkpointing coordinator to initiate the checkpoint. A unique node has to be identified as the coordinator. This may be achieved by using solutions to the mutual exclusion problem[RIC81] and making the selection of the coordinator a critical section activity.

The properties of global consistency and non-interference of checkpointing results in some overhead and reduces the processing time of transactions during checkpointing. For applications where continuous processing is so essential that the blocking of transaction processing for checkpointing is not feasible, we believe that a non-interfering approach provides a practical solution to the problem of checkpointing and recovery in distributed database systems.

Acknowledgement

The author would like to thank Dr. Won Kim and Professor Robert Cook for their valuable suggestions and comments on the previous version of this paper.

REFERENCES

[CHA85] Chandy, K. M., Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, ACM Trans. on Computer Systems, February 1985, pp 63-75.

[DAD80] Dadam, P. and Schlageter, G., Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints, Information Processing 80, North-Holland Publishing Company, Amsterdam, 1980, pp 457-462.

[FIS82] Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

[HAM80] Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

[KUS82] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. ACM SIGMOD, 1982, pp 293-302.

[LAM78] Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, Commun. ACM, July 1978, pp 558-565.

[RIC81] Ricart, G. and Agrawala, A., An Optimal Algorithm for Mutual Exclusion in Computer Networks, Commun. of ACM, Jan. 1981, pp 9-17.

[RIE79] Ries, D., The Effect of Concurrency Control on The Performance of A Distributed Data Management System, 4th Berkeley Conference on Distributed Data Management and Computer Networks, Aug. 1979, pp 221-234.

[SCH80] Schlageter, G. and Dadam, P., Reconstruction of Consistent Global States in Distributed Databases, International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 191-200.

[SON86a] Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 532-539.

[SON86b] Son, S. H. and Agrawala, A., Practicality of Non-Interfering Checkpoints in Distributed Database Systems, Proceedings of IEEE Real-Time Systems Symposium, New Orleans, Louisiana, December 1986, pp 234-241.

[SON87] Son, S. H., "Synchronization of Replicated Data in Distributed Systems," *Information Systems 12*, 2, June 1987, pp 191-202.

[SON88] Son, S. H., An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions, Proceedings of Fourth International Conference on Data Engineering, Los Angeles, February 1988, pp 528-535.

Robust Transaction-Routing Strategies in Distributed Database Systems

Yann-Hang Lee Philip S. Yu Avraham Leff

IBM Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

Abstract In this paper, we examine the issue of robust transaction routing in a heterogeneous distributed database environment. A class of dynamic routing strategies which use estimated response times to make routing decisions has previously been proposed. Since response time estimation and decision making depend on the assumed parameter values, it is important to examine the robustness or sensitivity to the accuracy of parameter values. Two refinements are proposed which improve system performance as well as robustness of routing decisions. One is the threshold strategy and the other is the discriminatory strategy.

1. Introduction

The locally distributed heterogeneous database environment is shown in Figure 1.1. The database is partitioned among the various processing systems, and the incoming transactions are routed to one of the processing systems by a common front-end system. If a transaction issues a database request which references a non-local database partition, the request, referred to as a remote database call, must be shipped to the system owning the referenced partition for processing. With regard to the destinations of database requests issued by a transaction, we can often identify one system as the preferred system to which the transaction sends most of its requests. In studying the performance of a transaction-processing system, the reference-locality distribution, i.e. percentage of database calls issued by a transaction to each database partition, has to be considered. As there is an additional overhead associated with remote database calls, routing an incoming transaction to the system with the lightest load may provide worse performance than routing the transaction to its "preferred" system. Thus transaction-routing strategies need to strike a balance between sharing the load among systems and reducing the number of remote calls.

Dynamic routing strategies for this environment have been studied in [Yu88]. In previous studies on dynamic load-balancing approaches, such as in [Wang85, Eagr85], it is assumed that incoming tasks can be completely serviced at any processing system. Under the heterogeneous database model, of course, these assumptions are invalid. A class of dynamic strategies based on an attempt to minimize each incoming transaction's response time, referred to as the MRT strategy, was proposed and studied in [Yu88]. It uses readily available information at the front-end system such as previous routing decisions of transactions currently in the complex. It has been demonstrated that system performance can be greatly improved if this concept of minimizing the response time of incoming transactions is used. In [Ferr86, Zhou87], a response-time-oriented load index based on mean-value equation is proposed which is a linear combination of queue lengths. The experiments done in [Zhou87] take samples of queue length and use smoothed queue length to calculate a load index for determining the placement of UNIX commands.

An issue of great significance from a practical view point is how critically the quality of the routing decision depends upon the accuracy of the assumed parameter values. The behavior of each transaction can vary from the assumption and even the average behavior may change from time to time. Thus a practical scheme has to be robust to the variation in parameter values.

In this paper, we examine two refinements to the MRT strategy to improve its robustness. One is a strategy that imposes a threshold criterion on the load condition before non-preferred-system routing, based on MRT, is considered; the other applies a policy which discriminates long transactions in applying non-preferred-system routing. A common idea underlies these schemes is that we seek to reduce the risks of making a non-preferred-system routing decision by being more selective about either the precondition or the candidate for non-preferred-system routing. Because the number of remote calls is reduced, the communications-bandwidth requirement is also reduced. These strategies are shown to be more robust than the original MRT with respect to parameter accuracy.

In the next section, the MRT strategy is briefly described. In Section 3, we introduce the threshold strategy. Its performance is evaluated with simulations and compared with the MRT strategy. Another refinement, the discriminatory strategy, is introduced and examined in Section 4. We summarize the results in Section 5.

2. Response-Time-Based Dynamic Routing Strategy

We now examine the MRT strategy proposed in [Yu88]. The strategy first estimates the average queue length or utilization of each processing system P_i. Then, the expected response times of an incoming transaction, if it were routed to the processing system P_i, for $i = 1,...,N$, are estimated. The processing system which provides the minimum expected response time is chosen to execute the transaction.

Under the MRT strategy, the routing decisions of active transactions are maintained by the front-end system in a routing-history table. Each time a transaction tx_k is routed to P_i, the entry in the k-th row and the i-th column of the routing-history table is incremented by one to reflect the new arrival and its routing. Furthermore, when a transaction is completed, the entry in the corresponding row and column of the table is decremented by one to reflect the departure. Note that there is a negligible overhead to maintain the table in the front-end system, and that no sampling of instantaneous state information from the processing systems is required.

The expected response time of an incoming transaction depends upon the transient behavior of the system and the future arrivals. For efficient implementation at the front-end, a steady-state analysis is applied to estimate the mean response time using transaction characteristics and mean queue length at each processing system. There are different approaches to estimate the mean queue length/response time as studied in [Yu88]. In this paper, we focus on the MRT strategy based upon the residence-time calculation. This approach has shown to provide the best performance over other approaches considered in [Yu88]. It regards the numbers of active transactions indicated by the routing-history table as fixed populations in a closed-queueing network. Naturally, it is possible to calculate the exact queue lengths based on a mean-value algorithm. However, this approach is impractical when we consider the complexity of the mean-value analysis. The MRT strategy uses an approximation based on Bard-Schweitzer's algorithm [Schwe79, Bard80] to calculate the residence time of each transaction at each processing system. Then, the queue length of each processing system is computed.

3. Threshold Strategy

Threshold approach has been considered by Eager, et. al., for load sharing in distributed systems with identical nodes [Eage86, Eage85]. Threshold is used in a location policy to determine the destination of transferring jobs. We demonstrate that a threshold can also be applied to the MRT in the environment studied. The new approach not only shortens the response time but, most significantly, reduces the sensitivity to accuracy in the assumed reference-locality distributions.

Under the threshold strategy, a more conservative approach is taken that recognizes the preferred system as the default system for the routing decision. A non-preferred-system routing is considered only when the preferred system is comparatively overloaded. However, when non-preferred-system routing is considered, the routing decision is again based on the MRT strategy. That is to say that a transaction is routed to a non-preferred system only when a sizable gain on response time can be achieved. Marginal response-time gain may not provide sufficient reward to make a non-preferred-system routing desirable. We in fact avoid individual optimization in pursuing global optimization. More precisely, our threshold strategy tests whether the ratio of the estimated response times between that of the preferred system and that of the system with minimum estimated response time is within a given threshold. If it is, preferred routing is taken. Otherwise, the MRT algorithm is used to make the routing decision.

In order to study performance of the routing strategies, a simulation is developed [Lee88] which is an extension of the one reported in [Yu88]. We consider an environment consisting of three transaction-processing systems with three transaction classes. Based on data from some IBM IMS systems [Corn86, Yu87], the average number of database requests per transaction is set to 15 for all

transaction classes. Geometric distribution is assumed for the number of database calls unless otherwise specified. The reference-locality distribution is given in Table 3.1.

Database Partition	1	2	3
Transaction class 1	0.75	0.11	0.14
Transaction class 2	0.07	0.82	0.11
Transaction class 3	0.11	0.06	0.83

Table 3.1 The reference locality distribution used in simulations

In addition, we assume that the processor speed is 7.5 MIPS and the pathlengths of each database call and the application processing between database calls are 9K and 21K instructions, respectively. The additional communication overhead of serving a remote database call, c, is chosen to be 3K and 15K to represent low and high communications overheads. The IO access time and the probability of having an IO during a database call are assumed to be 40 ms and 0.7 for all transaction classes, respectively. The arrival rates are adjusted so that the processing load of database calls on each system is balanced and the total processing load is as indicated.

To study the robustness of system performance, we would consider two additional measures besides response time. One is the percentage of transactions routed to non-preferred systems. This is referred to as *NPR*, the non-preferred-system routing ratio. Although non-preferred-system routing improves the balance of loads among the processors, it has side effects because it incurs remote calls. This puts more communications overhead on the processors and higher communications-bandwidth requirement on the links. It can also create a vicious cycle in that after transaction tx_1, with a preferred system P_1, is routed to P_2, the change in load conditions forces the arrival of a next transaction tx_2, with a preferred system P_2, to be routed to P_1. This vicious cycle is referred to as a *swapping phenomenon*. Thus, we are concerned with a sequence of 'locally' optimal decisions (in terms of estimated response time) that is sub-optimal 'globally'-- i.e. over the entire period of processing-- in that the preferred-system routing would have been the correct decision. The swapping ratio, *SWR*, defined as the percentage of non-preferred routings in which we observe a swapping phenomenon, is the other measure considered. This measure attempts to provide an indication as to whether a routing strategy makes too many sub-optimal non-preferred routings.

Let S represent the average processor utilization in the entire system excluding communications-processing overhead. Simulation results are shown in Table 3.2, for $S = 0.71$ and 0.81, where the overall response time, *NPR*, and *SWR* are presented. The original MRT strategy is equivalent to the case with threshold equal to 1. For a threshold of 1.15, slightly smaller response times are obtained in all cases examined than those for the MRT. Also we obtain more improvement for the case of high communications overhead under high load. However, the improvement in average response time is still less than 10%. To a lesser degree, this trend can be observed when the threshold is 1.2. The reduction on *NPR* and *SWR* are very significant in all cases.

We also consider the case with a threshold of 1.3 in Table 3.2. By ignoring opportunities to balance the load by routing to non-preferred systems (the strength of the MRT) we can worsen the response time significantly. In the extreme, the response time can reach 0.801 and 1.175 if we always route transactions to their preferred systems (i.e. a bigger threshold is used), for $S = 0.81$ and $c = 3K$ and $15K$, respectively. This demonstrates that the MRT policy of emphasizing load balancing is important.

In analyzing the results of the threshold strategy, we make the following observation. As *NPR* is very low compared to that of the MRT, we expect the system to be less balanced. This is verified by comparing differences between utilizations of the processing systems through simulation. However, the response time of the threshold strategy is as good as or better than that of the MRT. In a further study we examine the probability distribution of system unbalance, defined as $\max\{l_1,l_2,l_3\} - \min\{l_1,l_2,l_3\}$ where l_i is the instantaneous queue length of P_i, and is sampled during simulations. The distributions under the MRT strategy and the threshold strategy with a threshold of 1.2 are plotted in Figure 3.1. We note that the mean unbalance of the threshold strategy is higher than that of the MRT. Also, there is a difference in the tail of the distributions. That is, the threshold

strategy allows the system to become more unbalanced than the MRT does. Since there is a window of uncertainty as to how unbalanced the system will remain over the course of a transaction, allowing a small unbalance-- coupled with the reduction in utilization due to a remote-call overhead-- leads to better response time under the threshold approach.

	$S = 0.71$				$S = 0.81$			
threshold	1.(MRT)	1.15	1.2	1.3	1.(MRT)	1.15	1.2	1.3
c=3K								
RT	0.618	0.616	0.616	0.627	0.700	0.694	0.701	0.711
NPR	46.6%	13.6%	7.0%	3.6%	46.3%	17.7%	10.2%	6.3%
SWR	78.2%	43.3%	8.7%	3.3%	85.0%	49.0%	15.9%	6.6%
c=15K								
RT	0.707	0.705	0.701	0.716	0.986	0.933	0.916	0.941
NPR	19.4%	8.2%	4.2%	2.1%	17.9%	11.0%	6.9%	4.4%
SWR	37.6%	17.5%	4.1%	1.0%	43.4%	24.3%	8.4%	2.7%

Table 3.2 The performance comparisons of the threshold and the MRT strategies

Consider the cases in which the router uses inaccurate locality distributions to estimate the queue length and response time when making routing decisions. Simulations were conducted for cases where an inaccurate locality distribution $[q_{ki}']$ is assumed in the router for making routing decisions. Transactions issue database calls according to the actual reference distribution $[q_{ki}]$ during execution. In these simulations, $[q_{ki}]$ is set to the distribution defined in Table 3.1. The estimated distribution, $[q_{ki}']$, is set according to the following: given a percentage of inaccuracy x, if $(1 + x)q_{kk} < 1$, then $q_{kk}' = (1 + x)q_{kk}$ and for all $i \neq k$, $q_{ki}' = q_{ki}(1 - xq_{kk}/(1 - q_{kk}))$ (i.e. locality of the preferred system increases with a ratio x, whereas localities of non-preferred systems decrease proportionally); otherwise $q_{kk}' = 1$ and $q_{ki}' = 0$ for all $i \neq k$. The response times and the swapping ratios are plotted in Figure 3.2. It shows that, when weaker localities are assumed in making routing decisions, swapping ratio can reach 80% under the MRT as $x = -0.3$. The performance of the MRT degrades substantially in this case. However, when a threshold of 1.2 is applied, variations in response times are quite small when x is in the range of -0.3 to 0.2. It is clear that the threshold approach is much more robust than the MRT when less accurate reference-locality distribution is assumed.

4. Discriminatory Strategy

The MRT strategy implicitly assumes, over the course of a transaction, that the system load will remain the same as at the time when we make the decision. Actually the system load changes over the lifetime of a transaction because of departures and arrivals. The longer the transaction, the larger the load deviation is likely to be. Hence, a decision made at transaction arrival may not be optimal during its entire execution period. With uncertainty about the future, making a decision that involves only a few remote calls is less riskier than one involving a large number of remote calls. When the load is unbalanced, the router should try to improve the situation through many small corrections instead of one large correction.

Based on the above observation, a refinement of the MRT is to distinguish between short and long transactions when making routing decisions. We call this the discriminatory strategy. For example, we can apply a larger threshold to longer transactions in the threshold approach. In the following, we consider an ideal situation. Each transaction class consists of two subclasses: one long and one short which have 48 and 8 database calls, respectively, as in a Bernoulli distribution. Note that a Bernoulli distribution consisting of 87.5% short transactions with 8 database calls and 12.5% long transactions with 48 database calls has a mean of 15 database calls just as the geometric distribution considered before. Assume we can distinguish between the long and short transactions from the input parameters associated with each transaction. We discriminate against a long transaction

by automatically route it to its preferred system. We only adopt the MRT strategy when short transactions arrive -- i.e. we attempt to minimize the response times in situations where, even if we make a mistake, the exposure is low.

By using preferred-system routing for long transactions, the discriminatory strategy gives up some opportunities that MRT can use to balance the system. On the other hand, the discriminatory strategy avoids making non-preferred-system routing decisions for long transactions which can result in a large number of remote calls. In terms of response time we thus have a trade-off: the discriminatory strategy increases system unbalance, but decreases total processor utilization. In Table 4.1, we examine the case for both high and low communications overheads. We find that, over all transaction-length categories, and over all cases, the discriminatory algorithm has a lower response time than that of the MRT. The reductions can reach 8% for short transactions and 11% for long transactions when the processing load and communication overhead are high. The larger the transaction-processing load and the higher the communication overhead, the larger the exposure is for non-preferred routing: the improvement in response time is thus more apparent.

	$S = 0.71$		$S = 0.81$	
	MRT	Discriminatory	MRT	Discriminatory
c=3K				
RT(short)	0.334	0.328	0.384	0.372
RT(long)	1.963	1.917	2.244	2.152
RT(overall)	0.619	0.606	0.709	0.684
NPR	46.6%	40.9%	46.3%	40.9%
c=15K				
RT(short)	0.384	0.373	0.529	0.486
RT(long)	2.249	2.152	3.062	2.741
RT(overall)	0.711	0.684	0.972	0.885
NPR	19.5%	19.5%	18.2%	19.5%

Table 4.1 The performances of the discriminatory and the MRT strategies

In addition to improved performance, the discriminatory approach shows more robustness than the MRT when inaccurate locality distribution is used. With the same parameters as in Figure 3.2, we plot the response times and swapping ratios of the discriminatory approach and of the MRT in Figure 4.1, given that the number of database calls is a Bornoulli distribution as described before. The discriminatory approach shows little sensitivity to inaccuracy in the assumed locality. The smaller swapping ratio of the discriminatory approach, compared with that of the MRT, indicates that the risk of non-preferred routing is reduced by selecting good candidates for non-preferred routing, i.e., short transactions.

5. Conclusion

Our study shows that although the MRT strategy leads to pretty good performance, it is sensitive to the accuracy of the assumed reference-locality distribution. We have proposed a threshold approach which follows the MRT's routing decision only when the estimated load unbalancing is above a given threshold, and which otherwise uses preferred-system routing. There are two significant results from the threshold approach. The first is the ability to improve transaction response time even with a dramatic reduction in non-preferred system routing. The other one is, more importantly, the robustness that the threshold approach provides when the assumed reference-locality distribution is inaccurate. Because the risk of doing non-preferred-system routing may be greater for long transactions than for short transactions, we suggested a discriminatory approach in which the MRT strategy is applied only

to short transactions, and the preferred system for long transactions. Interestingly, both the short and long transactions show improvements in their response times. Furthermore, the robustness to inaccuracy in locality distribution is improved.

References

[Bard80] Bard, Y., "A Model of Shared DASD and Multipathing," *Comm. of the ACM,* Vol. 23, No. 10, (Oct. 1980), pp. 564-572.

[Corn86] Cornell, D.W., Dias, D.M., and Yu, P.S., "Analysis of Multi-system Function Request Shipping", *IEEE Tran. on Software Eng.,* Vol. SE-12, No. 10, Oct. 1986, pp. 1006-1017.

[Eage85] Eager, D L., Lazowska, E.D. and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", *Performance Evaluation Review,* Vol. 13, No. 2 (Aug. 1984), pp. 1-3.

[Eage86] Eager, D L., Lazowska, E.D. and Zahorjan, J., "Adaptive Load Sharing in Homogenous Distributed Systems", *IEEE on Soft. Eng.,* Vol. SE-12, No. 5, May 1986, pp. 662-675.

[Ferr86] Ferrari, D., "A study of Load Indices for Load Balancing Schemes," *Proc. of FJCC,* Nov. 1986.

[Lee88] Lee, Y.H., Yu, P.S., and Leff, A., "Robust Transaction Routing in Distributed Database Systems". *Proc. Intl. Symposium on Databases in Parallel and Distributed Systems* Dec. 1988, pp. 210-219.

[Schwe79] Schweitzer, P., "Approximate Analysis of Multiclass Queueing Networks of Queues", *Int. Conf. on Stochastic Control and Optimization* North Holland, Amsterdam, 1979.

[Wang85] Wang, Y.-T., and Morris, R.J.T., "Load Sharing in Distributed Systems", *IEEE Trans. on Computers,* Vol. C-34, No. 3, (March 1985), pp. 204-217.

[Yu87] Yu, P.S., Dias, D.M., Robinson, J. T., Iyer, B. R., and Cornell, D. W., "On Coupling Multi-systems Through Data Sharing," *IEEE Proceeding,* Vol. 75, No. 5, May 1987, pp. 573-587.

[Yu88] Yu, P.S.. Balsamo, S., and Lee, Y.-H., "Dynamic Transaction Routing in Distributed Database Systems," *IEEE Trans. on Soft. Eng.,* Vol. SE-14, No. 9, Sept. 1988, pp. 1307-1318.

[Zhou87] Zhou, S. , and Ferrari, D., "A Measurement Study of Load Balancing Performance," *Proc. of the 7th Conference on Distributed Computing Systems* Sep. 1987, pp.490-497.

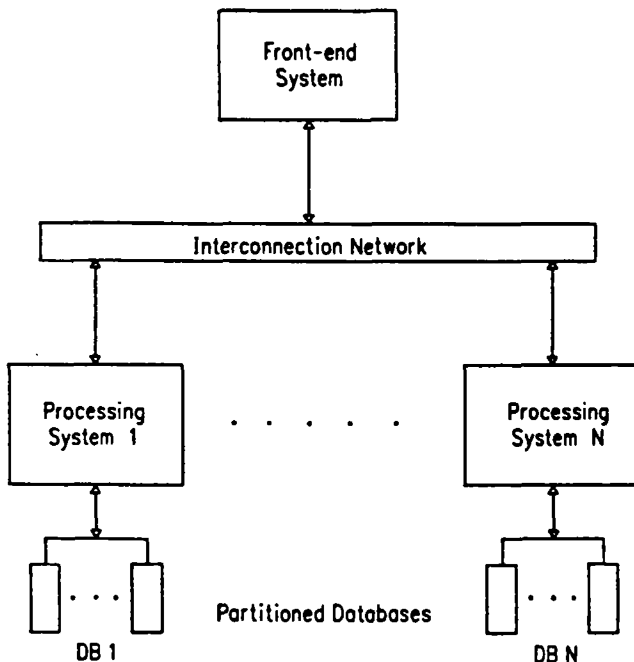


Figure 1.1 The configuration of a distributed transaction processing system

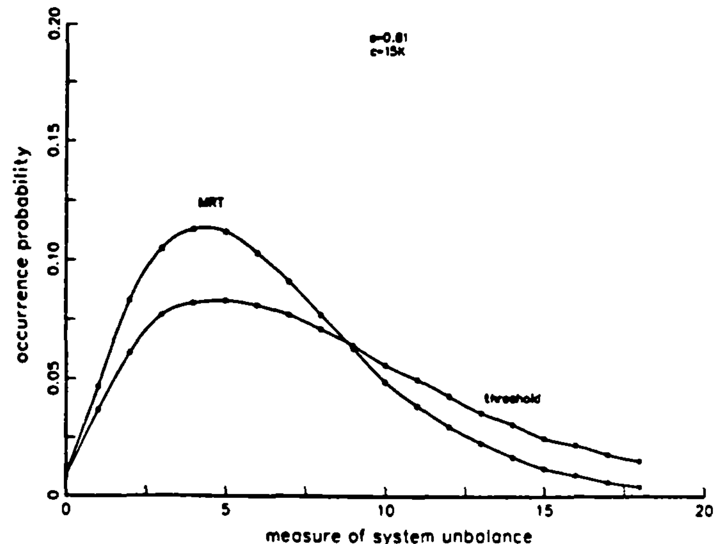


Figure 3.1 Occurrence probabilities of sampled system unbalance

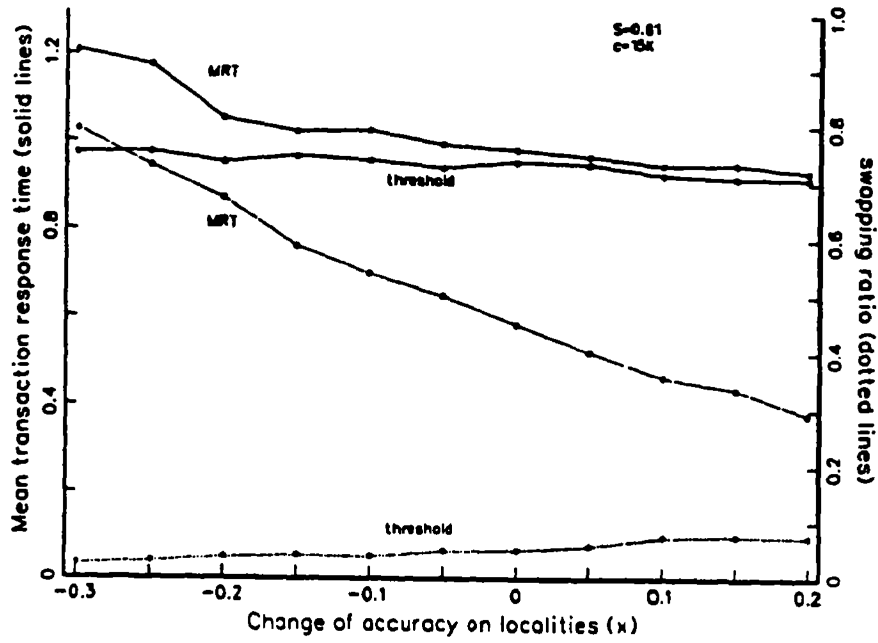


Figure 3.2 Performances of the MRT and threshold strategies with inaccurate locality distribution

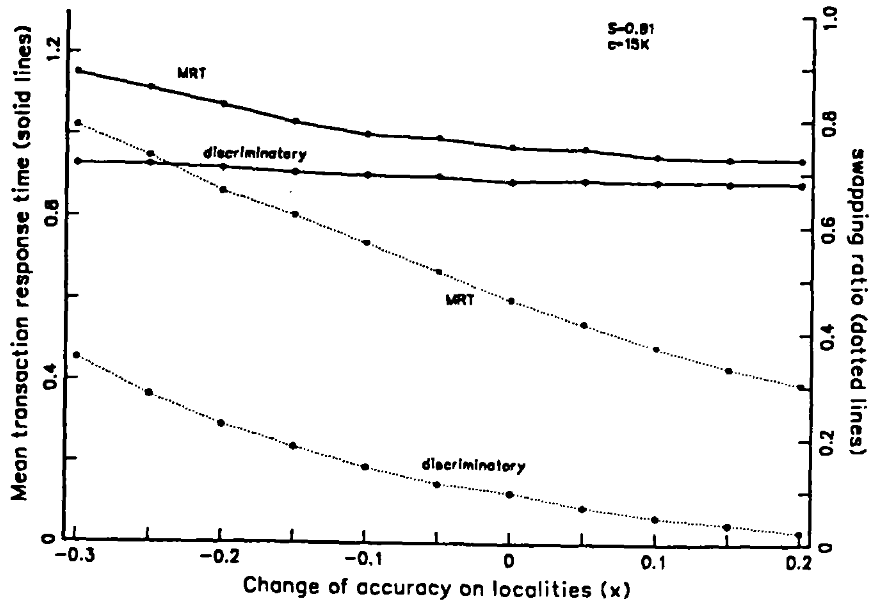


Figure 4.1 Performances of the MRT and discriminatory strategies with inaccurate locality distribution (no. of database calls is with Bornoulli distribution).

SHARING THE LOAD OF LOGIC-PROGRAM EVALUATION

Ouri Wolfson
Computer Science Dept., The Technion, Haifa 32000, Israel

ABSTRACT

We propose a method of parallelizing bottom-up-evaluation of logic programs. The method does not introduce interprocess communication, or synchronization overhead. We demonstrate that it can be applied when evaluating several classes of logic programs, e.g., the class of linear single rule programs. This extends the work reported in [WS] by significantly expanding the classes of logic programs that can be evaluated in parallel. We also prove that there are classes of programs to which the parallelization method cannot be applied.

1. INTRODUCTION

The efficient bottom-up-evaluation of intentional database relations, defined by means of recursive logic programs, has recently emerged as a very active area of research ([U], [BR], [K]). Two main methods of improving performance have received most of the attention. One is selection propagation, and the other is parallel evaluation.

Selection propagation reduces the number of relevant input-database tuples, by using constants passed as parameters to the database query processor. This usually necessitates a rewriting of the logic program which defines the intentional relation. The best known rewriting algorithm for this purpose is "magic sets" (see [BMSU]).

Parallel evaluation uses multiple cooperating processors, to reduce the overall evaluation time form start to finish. Most efforts in this area have been devoted to characterization of the logic programs which belong to the NC complexity class ([UV], [K], [AP]). If a program is in NC, it means that its intentional relations can be evaluated very fast, given a polynomial (in the number of input-database tuples) number of processors; they have to communicate extensively, usually through common memory. Unfortunately, this research means very little as far as utilizing a constant number of processors, particularly if they do not share common memory (e.g. a hypercube multiprocessor system[1] having 1024 nodes).

In this paper we assume an environment with a constant number of processors, which either communicate by message passing, or have common memory. The method that we propose is to create rewritten versions of the original logic program (a la selection propagation), and assign to each processor a different version. Each processor executes its version on a local copy of the input database, without communicating with the other processors. At the end, the union of outputs comprises the output of the original program (completeness). Therefore, if these outputs are sent to the same device or stored in the same file, the result is equivalent to a single-processor evaluation. Based on the paradigm of less-generated-tuples-implies-less-work, that lies at the heart of all selection propagation methods, and to which we also subscribe, each processor completes its evaluation before a single processor would have done so. The next example demonstrates our method.

Example 1: Consider the following DATALOG (see [MW]) program called in [MPS] the *canonical strongly linear (csl):*

$$S(x,y):-\ UP(x,w), S(w,z), DOWN(z,y)$$
$$S(x,y):-\ FLAT(x,y)$$

Assume that the extensional-database relations UP, FLAT, and DOWN represent a directed graph with three types of arcs. The csl program defines a tuple (a,b) to be in S, if and only if there is a path from a to b having k UP arcs, one FLAT arc, and k DOWN arcs, for some integer k.

Given processors { 0,...,r-1 } we propose that they share the load as follows. Processor i executes the *csl* program, with the predicate $i = x \bmod r$ added to the second rule of the program[2]. In other words, processor i computes the tuples (a,b) for which the path goes through a FLAT arc (c,d), with $i = c \bmod r$[3]. It is intuitively clear that for a large random graph, each one of the processors generates less tuples.

To demonstrate the time saving for a specific input to the csl program, consider the extensional database relations of Figure 1. UP consists of the tuples (i,i+1) for i = 1,...,4, FLAT consists of the tuples (i,6) for i = 1,...,5 and DOWN consists of the tuples (i, i+1) for i = 6,...,9.

1. see [H]

2. i = (x+y) mod r works as well

3. This works for character-strings as well, since the binary representation can be regarded as a natural number

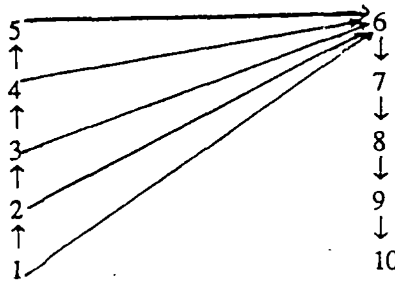


Figure 1: Sample input to the csl program.

The set NEW, defined below, consists of the tuples of S which are not in FLAT.

$$\text{NEW} = \begin{array}{llll} \{(4,7), & (3,7), & (2,7), & (1,7), \\ & (3,8), & (2,8), & (1,8), \\ & & (2,9), & (1,9), \\ & & & (1,10)\} \end{array}$$

Assume that S is computed by the naive evaluation method (see [B]). It assigns FLAT to S and then iteratively adds to S the tuples in the (projection of) *UP join S join DOWN*. Then in the first iteration, a single processor evaluating csl performs the join of a 4-tuples relation (UP), with a 5-tuples relation (S), with a 4-tuples relation (DOWN). In the second iteration the relations UP, S, DOWN are of sizes 4,9,4, respectively (first row of the set NEW has been added to S); third iteration 4,12,4 (second row has been added); fourth iteration 4,14,4; fifth and last iteration 4,15,4.

However, if two processors share the load by having processor i execute the csl program with i = x mod 2 added to the nonrecursive rule, then the arcs (1,6), (3,6), (5,6) will be assigned to processor 1, and the rest to processor 0. The maximal computation burden is placed on processor 1, performing five iterations with relations of sizes 4,3,4, 4,5,4, 4,7,4, 4,8,4, 4,9,4. Due to the smaller S-relation at each iteration, a significant time saving compared to the single processor case occurs. Processor 0 has a lower computation burden then processor 1, and completes even faster. If there are five processors instead of two a greater time saving results. In this case the maximum burden is placed on processor 0, performing five iterations, with relations of sizes 4,1,4, 4,2,4, 4,3,4, 4,4,4, 4,5,4, respectively.

Similar observations can be made if the evaluation is semi-naive ([B]) rather than naive. □

Let us emphasize that our method removes what for many problem domains constitutes the main obstacle to efficient parallelization, namely synchronization. In a multiprocessor system, often there is economic justification for increasing the number of processors, as long as performance can be increased. However, the problem is that as the number of processors grows, the required synchronization may slow processing to the single-processor level, and even below that. Therefore, even though there are other ways and operations performed in the process of evaluating logic programs that can be parallelized by using multiple processors, our method prevents the synchronization that is usually involved in parallelization. For example, assume that multiple processors are used to parallelize the join operation, instead of using them as proposed above. Then at each iteration of the naive or semi-naive evaluation, each processor would have to exchange its newly generated tuples with the newly generated tuples of every other processor. This procedure involves a lot of message passing or synchronization in accessing common memory.

The purpose of this paper is to determine to which programs the load sharing method described above can be applied. Specifically, we formally define what it means for a program to have a load-sharing scheme, and explore which programs do have such schemes, and which ones do not, i.e., are not amenable to parallel evaluation by the method described. We determine that almost all linear programs (each rule has at most one intentional predicate in the body) have such a scheme. In the class of single rule programs (sirups), defined in [CK], the pivoting ones (see [WS]) do have a load sharing scheme; so does a certain subclass of the simple chain programs. We define a class of sirups for which we prove that a load sharing scheme cannot exist. Several famous sirups belong to this class (e.g. path systems, introduced in [C]).

Presently, the only other method that we are aware of for speeding up bottom-up evaluation of logic-programs by using a constant number of processors, is the one introduced in [WS]. It resembles the one we proposed above, except for an important difference. The method is applied only when it can be guaranteed that each new tuple generated in the evaluation process is computed by a *unique* processor. The purpose is to partition (rather than share, as

in our method) the evaluation load. But consequently, the [WS] method is applicable only to a very restricted class of logic programs, called *decomposable*. For example, in the class of simple chain programs ([UV], [AP]) only to the regular ones are decomposable. Therefore, the csl program of example 1 is not decomposable. Intuitively, the reason for this is that since there may be more than one path between a and b, it is not guaranteed that each tuple is computed by a unique processor. For instance, in the sample input of example 1, if, in addition to the listed tuples, the tuple (2,9) is also in FLAT, then the tuple S(2,9) is computed by both processors 0 and 1.

One last comment regarding comparison with relevant literature concerns the parallel versions of PROLOG (e.g. [G], [S]). There has been a lot of research on the subject, but because of the fundamental difference between bottom-up (or forward chaining) and top-down (or backward chaining) evaluation of logic programs, this research is not applicable to our problem. Specifically, parallelization methods for PROLOG, which takes the top-down approach are not applicable in database query processing, which employs bottom-up. The reason for bottom-up is mainly because database applications are looking for *all* answers to a query.

The rest of this paper is organized as follows. In section 2, we provide the preliminaries, and in section 3 we define the concepts of a load sharing scheme, and its potential speedup, and prove initial results about them. In section 4 we determine that every program in a large subclass of all linear programs has a load sharing scheme, and in section 5 we prove that a whole class of sirups cannot have a load sharing scheme. In section 6 we discuss future work.

2. PRELIMINARIES

An *atom* is a predicate symbol with a constant or a variable in each argument position. We assume that the constants are the natural numbers. An R-atom is an atom having R as the predicate symbol. A *rule* consists of an atom, Q, designated as the *head*, and a conjunction of one or more atoms, denoted $Q^1, \ldots, Q^k$, designated as the *body*. Such a rule is denoted $Q :- Q^1, \ldots, Q^k$, which should be read "Q if Q^1 and Q^2, and, ...,and Q^k." A rule or an atom is an *entity*. If an entity has a constant in each argument position, then it is a *ground* entity. For a predicate R, a finite set of R-ground-atoms is a *relation* for R.

A DATALOG program, or a program for short, is a finite set of rules whose predicate symbols are divided into two disjoint subsets: the *extensional* predicates, and the *intentional* predicates. The extensional predicates are distinguished by the fact that they do not appear in any head of a rule. An *input* to P is a relation for each extensional predicate. An *output* of P is a relation for each intentional predicate of P. A *substitution* applied to an entity, or a sequence of entities, is the replacement of each variable in the entity by a variable or a constant. It is denoted $x1/y1, x2/y2, \ldots, xn/yn$ indicating that xi is replaced by yi. A substitution is *ground* if the replacement of each variable is by a constant. A ground substitution applied to a rule is an *instantiation* of the rule. When we talk about an instantiation we refer either to the ground rule, or to the substitution; which reference, will be clear from the context.

A *database* for P is a relation for each predicate of P. The output of P given an input I, is the set of relations for the intentional predicates in the database, obtained by the following procedure, called *bottom up evaluation*.

BUE1. Start with an initial database consisting of the relations of I.

BUE2. If there is an instantiation of a rule of P such that all the ground atoms in the body are in the database generated so far, and the one in the head is not, then: add to the database the ground atom in the head of the instantiated rule, and reexecute BUE2.

BUE3. Stop.

This procedure is guaranteed to terminate, and produce a finite output for any given P and I ([VEK]). The output is unique, in the sense that any order in which *bottom up evaluation* adds the atoms to the database will produce the same output.

Let ground atom a be in the output for P, and let s be a minimal sequence of iterations of BUE2 for deriving a. To s corresponds a *derivation tree* for a; it is a rooted tree with ground atoms as nodes, and a as the root. Node b has children $b_1, \ldots, b_k$, if and only if $b :- b_1, \ldots, b_k$ is an instantiation in s.

For some rule, a variable which appears in the the head, is called a *distinguished* variable. For simplicity we assume that each rule of a program is *range restricted*, i.e. every distinguished variable also appears in the body of the rule; additionally, we assume that none of the rules of a program has constants.

An *evaluable predicate* is an arithmetic predicate (see [BR]). Examples of evaluable predicates are sum, greater than, modulo, etc. A rule re is a *restricted version* of some rule r, if r and re have exactly the same variables, and r can be obtained by omitting zero or more evaluable predicates from the body of re. In other words, re is r with some evaluable predicates added to the body, and the arguments of these evaluable predicates are variables of r. For example, if r is: $S(x,y,z) :- S(w,x,y), A(w,z)$

then one possible re rule is: $S(x,y,z) :- S(w,x,y), A(w,z), x-y=5$

A program P_i is a *restricted version* of program P if each one of its rules is a restricted version of some rule of P.

Note that P_i may have more than one restricted version of a rule r of P. To continue the above example, if P has the rule r, then P_i may have the rule re as well as the rule re':

$$S(x,y,z) :- S(w,x,y), A(w,z), x-y=6$$

Throughout this paper, only restricted versions of a program may have evaluable predicates. The input of a program with evaluable predicates, i.e. a restricted version, is defined as before. The output is also defined as before, except that BUE2 also verifies that the substitution satisfies[4] the evaluable predicates in the ground rule; only then the atom in the head is added to the database and BUE2 is reexecuted. In other words, in considering instantiations for a restricted version of a rule, *bottom up evaluation* disregards database atoms which do not satisfy the additional evaluable predicates. Observe that if P' is a restricted version of P, then for every input, and for every intentional predicate R of P, the relation for R output by P' is a (possibly improper) subset of the relation for R output by P.

A predicate Q in a program P *directly derives* a predicate R if it occurs in the body of a rule whose head is a R-atom. Q is *recursive* if (Q,Q) is in the nonreflexive transitive closure of the "directly derives" relation. Predicate Q *derives* predicate R if (Q,R) is in the reflexive transitive closure of the "directly derives" relation (particularly, every predicate derives itself). A program is recursive if it has a recursive predicate. A rule is recursive if the predicate in its head transitively derives some predicate in its body.

3. LOAD SHARING SCHEMES

In this section we define and discuss the concept of a load sharing scheme, and establish that it is weaker than decomposability. Then the notion of the potential speedup of a load sharing scheme is defined, and we establish the potential speedup of a class of sirups called pivoting.

Assume that P is a program, and $P_1, \ldots, P_r$ are restricted copies of P, for $r>1$. Given and input, for an intentional predicate R of P, we denote by R_i the relation output by P_i for R; the relation output by P is denoted R. Observe that this is a somewhat unconventional notation, since for P_i the relation name is different than the predicate name. The set $D = \{P_1, \ldots, P_r\}$ is a *load sharing scheme* for evaluating some predicate T in P, if the following two conditions hold:

1. For each input I to $P, P_1, \ldots, P_r$ $\bigcup_i T_i \supseteq T$ (completeness).
2. There is an input such that for some intentional predicate Q which derives T, each $|Q_i| < |Q|$ (nontriviality).

In order to intuitively explain the above definition, we assume that each processor has a restricted copy of the program P, and the whole database, i.e. the set of input base relations, is replicated at each one of r processors. Alternatively, the database may reside in memory common to all the processors.

The completeness requirement in the definition is that no T-atoms are lost by evaluating the relation for T in each P_i, rather than the relation for T in P. Although the requirement is for inclusion in one direction only, the fact that $\bigcup_i T_i$ does not contain any atoms which are not in T is implied by the fact that each P_i is a restricted version of P. Thus, by using multiple processors and taking the union of the T_i's, the exact relation for T is obtained.

The nontriviality requirement refers to some predicate Q which derives T, i.e. has to be evaluated to determine the output relation for T. Nontriviality says that for some input, say I, the output of each P_i for Q is smaller than the output of P for Q. If, along the lines suggested in [BR, Section 4], the load of evaluating an intentional relation is measured in terms of the number of new tuples generated in the process, then the evaluation by the load sharing scheme completes sooner for the input I. Any doubt an implementor may have, concerning the load sharing scheme performing worse for some input than a single processor, can be removed by having one processor (of the thousand or so) perform the nonrestricted version of P.

In [WS] a decomposable predicate is defined. Decomposability with respect to restricted copies $P_1, \ldots, P_r$ is similar to the set being a load sharing scheme, with two exceptions. First, decomposability imposes an additional restriction, called lack-of-duplication. It requires that for each input I to $P, P_1, \ldots, P_r$, and for each $i \neq j$, the relations Q_i and Q_j are disjoint for any intentional predicate Q which derives T in P. Second, for decomposability, nontriviality requires that for some arbitrary input each T_i is nonempty.

Proposition 1: If a predicate T in a program P is decomposable with respect to $D = \{P_1, \ldots, P_r\}$, then D is a load sharing scheme for evaluating T in P.
Proof: Nontriviality and lack-of-duplication imply that there is an input for which each $|T_i| < |T|$. []

Next we define the notion of potential speedup. Let P be a program, and T an intentional predicate in P. Denote by $Q^1, \ldots, Q^t$ the set of intentional predicates which derive T. The *output of P for T given I*, denoted

4. for example, the substitution x/14,y/8 satisfies the evaluable predicate x-y=6, whereas the substitution x/13,y/9 does not.

$O(P,T,I)$, is $\bigcup_{i=1}^{t} Q^i$. In other words, the output contains T-ground-atoms and ground-atoms of other intentional predicates which derive T. Given a load-sharing scheme $D = \{P_1, \ldots, P_r\}$ for evaluating T in P, the *potential speedup* of D, denoted $Ps(D)$, is the maximal number M for which the following condition is satisfied. For every integer n and every ε, there is an input I for which $|O(P,T,I)| > n$, and $|O(P,T,I)|/\max_i |O(P_i,T,I)| \geq M-\varepsilon$. Intuitively, the potential speedup is the number to which the ratio $|O(P,T,I)|/\max_i |O(P_i,T,I)|$ can come arbitrarily close, when I is an arbitrarily large input. The definition is somewhat complicated since there are load-sharing schemes (the ones discussed in section 6) for which the potential speedup cannot be achieved, but to which the ratio can come arbitrarily close. Note that the fact that D is a load-sharing scheme implies that $1 \leq Ps(D) \leq r$.

The potential speedup means that for each one in an infinite set of inputs, the output of each P_i is at least $Ps(D)$ times smaller than the output of P; also, this output reduction occurs for arbitrarily large outputs. When the load to evaluate T is measured in terms of new ground atoms generated in the evaluation process, $Ps(D)$ is the ratio between the load of evaluating T by P, and the maximum load of a processor, when the sharing scheme is used. Although we defined the potential speedup based on some infinite set of inputs, for the load sharing schemes that we are discussing in this paper, it is intuitive that time saving can be achieved for the "average input". The reason is that each load-sharing scheme discussed in this paper is obtained by adding the evaluable predicate $i = (x_1 + \ldots + x_k) \bmod r$ to one of the rules, where $x_1, \ldots, x_k$ are distinguished variables. For an input which is distributed evenly across a range of natural numbers, this reduces the number of newly generated tuples at each processor.

A *single rule program* (see [CK]), or a sirup for short, is a DATALOG program which has a single intentional predicate, denoted S in this paper. The program consists of two rules. A nonrecursive rule:
$S(x1,\ldots,xn) :- B(x1,\ldots,xn)$
where the xi's are distinct variables; and one other, possibly recursive, rule in which the predicate symbol B does not appear.

Assume that R is a set of atoms with each atom having a variable in each argument position. The set R is *pivoting* if there is a subset d of argument positions, such that in the positions of d:
1. the same variables appear (possibly in a different order) in all atoms of R, and
2. each variable appears the same number of times in all atoms of R.
A member of d is called a *pivot*. Note that a variable which appears in a pivot may or may not appear in a nonpivot position. The recursive rule of a sirup is *pivoting* if all the occurrences of the recursive predicate in the rule constitute a pivoting set. For example, the rule: $S(w,x,x,y,z) :- S(u,y,x,x,w), S(v,x,y,x,w), A(u,v,z)$
is pivoting, with argument positions 2, 3 and 4 of S being the pivots.

Theorem 1: If the recursive rule of a sirup is pivoting, then the sirup has a load-sharing scheme of any size. The potential-speedup equals the size of the scheme.
Proof: Assume that argument positions $i_1, \ldots, i_k$ of S are the pivots. Consider restricted version P_j of P which has the same recursive rule as P, and a nonrecursive rule

$$S(x1,\ldots,xn) :- B(x1,\ldots,xn),\ j = (xi_1 + xi_2 + \ldots + xi_k) \bmod r$$

for $j=0, \ldots, r-1$. It is easy to see that $\{P_0, \ldots, P_{r-1}\}$ is a load-sharing scheme. Nontriviality and potential-speedup can be demonstrated using as input a prefix of the sequence $\{ B(1,\ldots,1,q,1,\ldots,1) \mid q \geq r$, and q appearing in position $i_1 \}$. □

4. LINEAR PROGRAMS

In this section we discuss linear programs. A program is *linear* if the body of each rule contains at most one intentional predicate. A rule of a program P is an *exit* rule if its body consists of extensional predicates only. An exit rule r_e, with extensional predicate symbols $B_1, \ldots, B_k$, is *distinct*, if there is no other rule r of P for which the following condition is satisfied: in the body of r there are some extensional predicates, and every such extensional predicate belongs to the set $\{B_1, \ldots, B_k\}$. In other words, r_e is not distinct if there is another rule of P in which a subset of the B_i's appears, but no other extensional predicate does. Note that the exit rule of a sirup is distinct. An exit rule r_e in P *derives* a predicate R if the predicate in r_e's head derives R. An intentional predicate of a linear program is distinct if it is derived by a distinct rule.

Theorem 2: If T is a distinct predicate of a linear program P, then there is a load-sharing scheme of any size for evaluating T in P. The potential-speedup equals the size of the scheme.
Proof: Assume without loss of generality that the the first variable of the atom in the head of each exit rule is x. Let restricted version P_j of P be obtained by adding the predicate $j = x \bmod r$ to each exit rule, for $j=0, \ldots, r-1$ (all

the other rules stay the same).
To show completeness, assume that a T-atom, a, is in the output of P. Consider a minimal-length sequence s of iterations of BUE2, for deriving a. It can be shown by induction on the length of s, that s has exactly one instantiation of an exit rule. In that instantiation, x is substituted for some constant, n. Let $l = n \bmod r$. Then s is a derivation sequence for a in P_l. Thus, a is in at least one T_i. (It may be in more than one T_i if a can be obtained by a sequence of iterations with a different instantiation of an exit rule).
Now assume that the distinct exit rule which derives T in P is:

$$R(x,......) :- B_1(.........), B_2(.........), \ldots, B_k(.....)$$

Nontriviality and potential-speedup can be demonstrated by using as input the set of atoms $\{ B_1(i,...,i), B_2(i,...,i), \ldots, B_k(i,...,i) \mid 1 \le i \le N \}$ for a large enough N. []

In [WS] we have shown that a linear sirup without repeated variables in the recursive predicate, is not decomposable if it is not pivoting. In contrast, observe that theorem 2 implies that *every* linear sirup has a load-sharing scheme of any size. Another comment is that the above proof does not work if the predicate T of Theorem 2 is not distinct. The reason for this is given in [W].

5. PROGRAMS WITHOUT A LOAD SHARING SCHEME

In this section we demonstrate that not every program has a load sharing scheme. Specifically, we provide a necessary condition for a sirup to have a load sharing scheme (Theorem 3). It turns out that some famous sirups do not satisfy the condition. An example is the first P-complete problem, *path-systems* ([C]). The sirup for the problem is:

$S(x) :- S(y), S(z), H(x,y,z)$
$S(x) :- B(x)$

Another example of a sirup without a load sharing scheme, is a variant of path systems called the *blue blooded frenchman* ([CK]):

$BBF(x) :- BBF(m), BBF(f), MOTHER(x,m), FATHER(x,f)$
$BBF(x) :- FRENCH(x)$

Some other variations which have not been defined previously, as far as we know, are (nonrecursive rule obvious, thus specification omitted):

$S(x,u) :- H_1(x,y,u), H_2(x,z,w), S(y,u), S(z,w)$

$S(x,u) :- H(x,y,z,u,w), S(y,u), S(z,w)$

$S(x) :- H_0(x,w), H_1(w,y), H_2(w,z), S(y), S(z).$

$S(x,w,y) :- UP(x,t,u), S(t,u,v), FLAT(v,w,z), S(z,r,s), DOWN(r,s,x)$

What do the above sirups have in common? This is what the next theorem establishes. Before stating it we need the following definition.

Given a sirup P, denote by $A(P)$ the set of atoms in the body of the recursive rule, and by $V(P)$ the set of variables in $A(P)$. Let $R(P) = \{x \mid x \text{ is in } V(P), \text{ and } x \text{ appears in some } S\text{-atom of } V(P)\}$. Let the *extensional graph* of P, denoted $G(P)$, be an undirected graph defined as follows. Its set of nodes is $V(P) - R(P)$, in other words, variables which do not appear in any S-atom in the body of the recursive rule. For two distinct nodes of $G(P)$, x and y, the edge $x - y$ is in the graph if and only if there is an extensional-predicate atom, A, in the body of the recursive rule such that x and y are variables of A.

Theorem 3: Given a sirup P, with the recursive rule r, denote by ST the set of atoms in the body of r. Then P does not have a load sharing scheme if the following conditions are satisfied.

1. Except for the S-atoms, there are no two atoms of $A(P)$ which have the same predicate symbol.
2. There are at least two S-atoms in $A(P)$, and the S-atoms in $A(P)$ have pairwise disjoint variables, and none of them has repeated variables.
3. Each extensional predicate atom in $A(P)$ has a variable which is not in $R(P)$, and each variable in $R(P)$ appears in some extensional predicate atom.

4. The graph $G(P)$ has a distinguished variable in each one of its connected components.

Proof: see [W].

It is easy to verify that *path systems* and the other sirups that have been discussed in this section satisfy the requirements of Theorem 3.

6. FUTURE WORK

An obvious direction for future research is to extend the class of programs which have load sharing schemes, and the class for which we can prove nonexistence. However, we conjecture that determining whether a program does or does not have a load sharing scheme, is undecidable. Another question is the following. If a load sharing scheme exists, does there always exist one with a potential speedup equal to the size of the scheme? For the programs discussed in this paper we have seen that this is the case. Must it be true? Finally, we would like to determine how to distribute the load when communication among the processors participating in the evaluation cannot be avoided, and what architectures are preferable. Some answers are provided in [CW].

Acknowledgement: The author thanks Nissim Francez and Oded Shmueli for helpful discussions and comments.

References

[AP] F. Afrati and C. H. Papadimitriou "The Parallel Complexity of Simple Chain Queries", *Proc. 6th ACM Symp. on PODS*, pp. 210-213, 1987.

[B] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations" in *On Knowledge Base Management Systems - Integrating Database and AI Systems, Brodie and Mylopoulos, Eds., Springer-Verlag, pp.165-178, 1986.*

[BMSU] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc. 5th ACM Symp. on PODS*, pp. 1-15, 1986.

[BR] F. Bancilhon and R. Ramakrishnan "An Amateur's Introduction to Recursive Query Processing", *Proc. SIGMOD Conf.* pp. 16-52, 1986.

[C] S. A. Cook "An Observation on Time-Storage Trade-off" *JCSS 9(3)*, pp. 308-316, 1974.

[CK] S. S. Cosmodakis and P. C. Kanellakis "Parallel Evaluation of Recursive Rule Queries", *Proc. 5th ACM Symp. on PODS*, pp. 280-293, 1986.

[CW] S. Cohen, O. Wolfson, "Why a Single Parallelization Strategy is Not Enough in Knowledge-Bases", to appear, Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, PA, March 1989.

[G] S. Gregory, Parallel Logic Programming in PARLOG, Addison Wesley Publishing Co.

[H] M. T. Heath "Hypercube Multiprocessors 1986", Society for Industrial and Applied Mathematics, Philadelphia PA, 1986.

[K] P. C. Kanellakis "Logic Programming and Parallel Complexity", *Proc. ICDT '86, International Conference on Database Theory*, Springer-Verlag Lecture Notes in CS Series, no. 243, pp. 1-30, 1986.

[MPS] A. Marchetti-Spaccamela, A. Pelaggi, D. Sacca "Worst Case Complexity Analysis of Methods for Logic Program Implementation" *Proc. 6th ACM Symp. on PODS*, pp. 294-301, 1987.

[MW] D. Maier and D. S. Warren "Computing with Logic: Introduction to Logic Programming", Benjamin-Cummings Publishing Co., 1987.

[S] E. Y. Shapiro "A Subset of Concurrent Prolog and Its Interpreter", TR-003 ICOT, Tokyo, Japan.

[U] J. D. Ullman "Database Theory: Past and Future", *Proc. 6th ACM Symp. on PODS*, pp. 1-10, 1987.

[UV] J.D. Ullman and A. Van Gelder, "Parallel Complexity of Logic Programs", TR STAN-CS-85-1089, Stanford University.

[VEK] M. H. Van Emden and R. A. Kowalski "The Semantics of Predicate Logic as a Programming Language", *JACM* 23(4) pp. 733-742, 1976.

[W] O. Wolfson, "Sharing the Load of Logic Program Evaluation", Proc. of the Intl. Symp. on Databases in Parallel and Distributed Systems, pp. 46-55, Austin, TX, Dec. 1988.

[WS] O. Wolfson and A. Silberschatz, "Distributed Processing of Logic Programs" TR466 The-Technion, CS Dept. Also, Proc. of the ACM-SIGMOD Conf., pp. 329-336, 1988.

IEEE Computer Society

1730 Massachusetts Avenue, N W
Washington, DC 20036-1903

Non-profit Org
U.S Postage
PAID
Silver Spring, MD
Permit 1398

Dr. David B Lomet
Digital Equipment Corporation
9 Cherry Lane
Westford, MA 01886
USA