INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Distributed Parallel Query Processing on Network of Workstations

by

Cyrus Soleimany, B.Sc.

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University
Ottawa, Ontario
September 1998

© Copyright
1998, Cyrus Soleimany

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file   Votre référence

Our file   Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36880-7

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Distributed Parallel Query Processing on

Network of Workstations

submitted by

Cyrus Soleimany, B.Sc.

in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

September 1998

Abstract

Several commercial parallel database systems have been introduced for speeding up database activities; however, the cost associated with them is one of the factors that most companies have not been able to afford them. A Network of Workstations (NOW) environment is an attractive alternative to parallel database systems due to the cost advantages. Furthermore, the availability of public domain software, such as Parallel Virtual Machine (PVM) and Message Passing Interface (MPI), for use as message-passing tools also facilitate the use of NOWs for database query processing. The structure of the query in Relational Database Management Systems (RDBMS) presents opportunities to exploit several aspects of parallelism, such as concurrent query execution, inter-operation and intra-operation parallelism.

This thesis focuses on the detailed design of a distributed architecture and associated performance issues like processor allocation, job scheduling, and load coordination. The thesis provides a performance evaluation of the distributed architecture for parallel query processing on NOWs. The results show that the distributed architecture provides excellent speedups, scaleups, and load sharing.

Acknowledgments

I would like to take this opportunity to thank my supervisor, **Professor Sivarama Dandamudi**, whose continuous help and support have been essential to the completion of the thesis.

Also, I feel great appreciation to my wife, **Minoo Nouri**, who is my primary source of inspiration and support and my lovely sons **Sina** and **Nima** for their patience when I needed a quiet place at home.

I would like to thank **Sonia Takkar**, **Gautam Jain** and **Jemal Abawajy** for their ideas, their time, many hours of discussion, and useful comments on my thesis.

Thanks, also, to **Peter Choynowski** and **Andrew Miles** for system support and making the parallel machine available, and all the staff of the *School of Computer Science*.

Finally, I dedicate this thesis to my parents, who have always given me strength and encouragement.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Parallel Database Systems	3
1.2 Network of Workstations (NOWs)	4
1.3 Motivation	6
1.4 Objectives	8
1.5 Overview of the Thesis	9
2 Background	10
2.1 Related Work	10
2.2 Query Processing in Shared-Nothing Parallel DBMS	14
2.2.1 Load Distribution and Node Allocation	20
2.2.2 Scheduling Parallel Jobs	22
2.3 Load Sharing	25
2.3.1 Chunk self-scheduling (CSS)	25

2.3.2	Guided self-scheduling (GSS)	26
2.3.3	Factoring scheduling (FS)	27
2.3.4	Trapezoid self-scheduling (TSS)	27
2.4	Chapter Summary	29
3	Distributed Architecture for Query Processing	30
3.1	Overview	31
3.1.1	Centralized Architecture	32
3.1.2	Distributed Architecture	33
3.1.3	Hierarchical Architecture	35
3.2	Distributed Architecture	35
3.2.1	Block Diagram of Distributed Architecture	36
3.2.2	Load Sharing Within a Query	45
3.3	Chapter Summary	49
4	Experimental Environment	51
4.1	NOW-based system Environment	51
4.2	Parallel Virtual Machine (PVM)	52
4.3	Database Model	54
4.4	Workload Model	55
5	Performance Evaluation	57
5.1	Selection of Parameters	57
5.2	Upper and Lower Bound Thresholds	59
5.3	Performance as a Function of Inter-arrival Time	60
5.4	System Speedup and Scaleup	62
5.5	Sensitivity to Fragmentation Size	65
5.6	Impact of Background Load on Performance	67
5.7	Performance with Different Query Types	67

6	Conclusion	71
6.1	Summary	71
6.2	Contribution of the Thesis	72
6.3	Future Work	73
	References	75
A	Procedure	80
A.1	Procedure 1 <i>Scheduler</i>	80
A.2	Procedure 2 <i>Transfer Policy</i>	81
A.3	Procedure 3 <i>Location Policy</i>	81
A.4	Procedure 4 <i>Load Coordinator</i>	82
A.5	Procedure 5 <i>Result Matcher</i>	84

List of Tables

3.1	Message Tags	39
4.1	Selectivity Factors	55
5.1	Parameter Values	58
5.2	Processor Power in <i>Triton</i>	65
5.3	Four Relations in Balance and Left-deep Query	69
5.4	Comparison Between Balance and Left-deep Query	70

List of Figures

1.1	Two Types of Parallel Processing Systems	2
1.2	Three Different Parallel Database Architectures	3
1.3	Network of Workstations (NOWs) Architecture	5
2.1	The three-level architecture of DBMS [16]	11
2.2	A Typical Query Processor [16]	13
2.3	Inter-query Parallelism	16
2.4	Intra-query Parallelism	17
2.5	Intra-operation Parallelism	19
2.6	$T(1)$ of the $TSS(I/2P)$	28
3.1	Centralized Database System in NOW-based System	32
3.2	Work transfer process in the hierarchical architecture	34
3.3	High-Level Block Diagram of the Distributed Architecture	37
3.4	Flow of Messages and Data	38
3.5	Two Types of Execution Plan	44
4.1	Network of Workstations Configuration	52
5.1	Performance Sensitivity to Probe Limit	58
5.2	Sensitivity to Different Threshold Values	60
5.3	System Performance on Different Inter-arrival Time	61
5.4	System Speedup	62

5.5	System Scaleup	63
5.6	Sensitivity to Fragmentation Size	66
5.7	Impact of Background Load on Performance	68
5.8	System Performance with Two Types of Queries	69

Chapter 1

Introduction

During the 1970s and 1980s, the input/output bottleneck was a major problem in database machines due to high data access time from disks. To avoid the severe I/O bottleneck, researchers proposed various techniques. One of these techniques is increasing the I/O bandwidth by a disk striping method. Using this technique, the response time of any disk access can be reduced since the transmitted data is reduced to smaller chunks. This idea of disk interleaving was implemented in the Cray operating system and some early Unix systems. The other approach is de-clustering in which a file is divided into a number of sub-files and each sub-file is stored on one disk unit. Round-robin, hashing, and range partitioning are three different techniques that have been used in this approach. These ideas of de-clustering were implemented in the Gamma [17] and the Bubba [2] database machines.

In 1983, the relational database management system (RDBMS) was introduced and today it dominates the database industry. Relational database management systems and parallel processing technologies have reached a point where they can be combined to support various applications. The main advantages of relational database management systems are data independence and high-level query languages, which increase programmer productivity and facilitate optimization. Be-

side these improvements in RDBMS, parallel processing uses several techniques to improve the response time of requests. Today, the competitive world with different kinds of users expects more speed in database systems. Information volumes are growing 25-35% per year and there is also an increase in the rate of transactions in applications. Thus, complex queries will increase the demand for acceptable response times in database systems [27].

Figure 1.1: Two Types of Parallel Processing Systems

Parallel processing systems can be broadly divided into two architectures: shared-memory and distributed-memory as shown in Figure 1.1. In shared-memory systems, processors communicate with each other through an interconnection network by using shared memory variables. The lack of scalability due to network bandwidth limitation is one of the major limitations of this architecture. On the other hand, in distributed-memory systems each processor has its own local memory. Communication among processors is accomplished by means of message passing. Scalability is one of the advantages of this system. However, the lack of load balancing restricts the use of this architecture.

Figure 1.2: Three Different Parallel Database Architectures

1.1 Parallel Database Systems

The goal of parallel processing is to make a set of computers to collectively work on certain problem and reduce the time required to obtain the result. Basically, parallel database systems are classified into three architectures. These are the *shared-everything*, *shared-disk*, and *shared-nothing* architectures (Figure 1.2). In the *shared-everything* architecture a set of CPUs are connected to a high speed bus and common memory bank and disks. The commercially available systems of this type are the IBM/370, Digital VAX and the Sequent Symmetry. In the *shared-disk* architecture, each node has an individual memory but direct access to all disks. The commercially available systems of this type are the IBM Sysplex and original Digital VAX cluster. In the *shared-nothing* architecture, all nodes have their own CPU, cache, disks and memory and communicate with each other by sending messages across the network. This kind of a parallel database system has been the most pop-

ular design and has led to the emergence of many parallel machines like Teradata, Tandem, Bubba, Gamma and so on. There has been a lot of debate in the database community about the suitable architecture for parallel processing. The paper [36] provides a performance evaluation of these three architectures under a uniform workload. The result shows that shared-everything performs better than shared-nothing and shared-disk fairly competes with shared-nothing architecture only in certain applications. But both shared-disk and shared-everything architectures suffer from low I/O bandwidth and memory limitation. On the other hand, the shared-nothing architecture can be scaled extensively as none of the components are shared. However, the presence of individual memory and disk increases the complexity of the system and can lead to potential load balancing problems. The performance is not the only issue in recent parallel database systems, but other factors such as scalability, and reliability also matter. Therefore, the shared-nothing architecture has emerged as the most commonly used architecture in all the commercially available products and it is the best choice among the other architectures.

1.2 Network of Workstations (NOWs)

Network of Workstations (NOWs) is a special class of parallel processing systems, which falls in the category of distributed-memory architecture, where a set of workstations are interconnected through a Local Area Network (LAN). Each workstation has its own private memory, disk, and local communication network and has access to a global interconnection network as shown in Figure 1.3. Communication among the workstations is done by means of message passing. In the area of parallel processing there is a trend to use NOWs as a virtual parallel machine since NOWs can be defined as a set of heterogeneous computers connected via an interconnection network. Thus, in recent years, NOWs have attracted a lot of attention from researchers for three major reasons:

- High-powered workstations connected via fast communication networks have been considered as an alternative to special-purpose parallel computers.

- Networks of workstations offer a cost-effective and scalable alternative to supercomputers, and also provide a general-purpose environment for both parallel and non-parallel applications.
- The availability of many idle workstations in the network leads the researcher to use this source of computational power for the purpose of parallel processing (up to 80% processing power of a workstation is available depending on the time of day [5]).

Figure 1.3: Network of Workstations (NOWs) Architecture

The success of NOWs is also due to the availability of public domain software such as PVM, and MPI, which allow the system to be viewed as a large distributed-memory machine. These software packages make it convenient for developers to create parallel applications by virtualizing the workstation and providing the communication facilities. PVM supports a heterogeneous network by providing a unified framework for developing parallel programs in an efficient and straightforward manner using the existing hardware. Since PVM handles all of the message routing, data conversion, and task scheduling across the network of incompatible computers, we can essentially view the distributed network as a single parallel virtual machine.

Until recently, only high performance parallel processors and supercomputers were able to satisfy the computing requirements for certain applications. Fortunately, modern network of workstations connected by high-speed networks have the ability to run most application that run on supercomputers with reasonable performance, but at a lower cost. The traditional NOW-based environments have several limitations. For example, most environments need the intervention of the operating system to make the simplest exchange information between workstations, which leads to a significant overhead. But these public domain software take this overhead off from the system. PVM allows heterogeneous networks (parallel, serial, or vector computers) to appear as a single resource. Since the architecture of each machine can be potentially different from each other, PVM converts internal data representations of each machine to a standard form to facilitate communication. A significant advantage of PVM is that when PVM spawns a task it tries to spawn it on the least loaded machine. However, the user can override this default by specifying the particular machine. Another advantage of PVM is that when it detects a faulty machine it will remove the faulty one from the virtual machine.

An important limitation that PVM suffers from is the lack of communication among multiple distributed virtual machines. Several distributed virtual machines cannot collaborate, merge, or split. This means there is only one single user per virtual machine.

1.3 Motivation

Some of the motivations for parallel query processing are [5, 6]:

- An increase in the size of relational databases.
- Data intensive read-only queries on these large databases.
- Support for high-level ad hoc query languages like SQL, allowing users to submit complex queries at their terminals.

- The need for supporting non-standard applications like VLSI CAD, Computer Aided Software Engineering (CASE) tools, Multimedia, etc.

All of the above applications demand high performance.

The performance of database systems can be improved by applying parallel processing techniques. There are several commercial database systems on the market but these are expensive. In 1997, Dandamudi [5] proposed the use of network of workstations to improve the performance of database query processing. This led us to further research this area. However, using a network of workstations for database query processing poses several problems and performance issues. As each workstation in a NOW is under an individual control, (or in other words, there is no central controller), it is not possible to distribute the database among workstations. As a result, database relations are stored in the central database node as a traditional database system to provide better security and to maintain the integrity of data. However, we can use NOWs to speed up query processing involving only read-only access of data.

The database itself is not partitioned across the workstations as in parallel database systems; this causes additional overhead as the data has to be sent to the other workstations to be worked on. Load imbalance is the other problem in NOW-based database systems due to the heterogeneity of the system and lack of a central scheduler. Since each workstation receives a job at any time, maintaining load balance among all workstations requires a dynamic load balancing scheme. Job migration is another issue, which degrades the performance of applications. In a NOW-based system, a user may remove his/her workstation from the pool of shared workstations used for database query processing. In this situation, the incomplete job has to be either killed and restarted on another workstation or the current state of the job has to be saved and the job has to be transferred along with the partial result and resumed on another workstation. Both of these cases cause a delay in response. Fault tolerance is another important issue in this system, because either a malfunction of one workstation or a broken link connection affects the overall system performance. The last important issue is communication network latency. NOW-

based systems use a slow communication network compared to the interconnection network used in a parallel database system. Clearly, a faster network improves the performance since NOW-based database systems depend on message passing. For database applications, if data is in main memory, speed of the communication network can have serious impact on performance, but if data is on disk, the speed of the communication network can only have a marginal impact on performance.

1.4 Objectives

The main goal of this thesis is to develop the *distributed architecture* and address the problems associated with parallel query processing in a NOW-based environment and conduct experiments to evaluate the performance of this architecture. By analyzing the simulation results, we would like to show that the distributed architecture could be an attractive alternative for parallel database systems.

Unlike parallel database systems, query processing on NOWs poses several problems. The lack of a central controller forces the system to store the database tables in the traditional way, which introduces an additional overhead due to the shipment of relations to the workstations. Load balancing is the second problem due to the heterogeneity of the system. Since all workstations can introduce background load (non-query load) at any time, load balancing is difficult in a NOW-based environment. Workstation or link failure is another issue, which degrades the performance of the system due to job migration. The delay caused by this problem affects the overall response time since the job has to be restarted in another node.

The distributed architecture described in detail in Section 3.2 presents a solution to all the above problems and performance issues. We, however, do not consider node failures. Furthermore, the performance results in Chapter 5 show that the system can handle both situations (with and without background load) effectively.

In this thesis we investigate the distributed architecture for parallel query processing on a network of workstations and demonstrate the usefulness of a *sender-initiated* policy as one of the components of the dynamic policies for load distribution

and node allocation. We provide a detailed implementation of this technique in Section 3.2.1, which solves the problem of load imbalance in a distributed environment. Also, based on the performance analysis of some load sharing policies described in Section 2.3 (from [33]), we implement the *Chunk self-scheduling (CSS)* for the purpose of load sharing. This technique is also used in Section 3.2 to give a solution to the load sharing problem, when the load of the system changes (under-loaded, OK, overloaded states). To accomplish this job, we also need to develop a database and workload model to handle the arrival and optimization of the query.

Most existing research in this area uses either synthetic or simulation data. This thesis develops and implements an application program for distributed architecture for parallel query processing and studies the performance of the system in different scenarios by adopting some of the algorithms and techniques proposed in the literature.

1.5 Overview of the Thesis

In this thesis, we briefly describe the three architectures (centralized, distributed, and hierarchical architectures) used for database query processing on network of workstations. We will study the behavior of the *distributed architecture* in detail. This is followed by the performance evaluation of simulation results for the distributed query processing. The organization of the remainder of the thesis is as follows: Chapter 2 gives a review of the background work in DBMS query processing, load sharing algorithms, loop scheduling policies and query processing in a NOW-based environment. Chapter 3 focuses on three different architectures and describes in detail the distributed architecture and its implementation. Chapter 4 describes the simulation environment, the workload model, and the PVM software. The experimental results are given in Chapter 5 along with the performance evaluation. Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

Background

2.1 Related Work

A database is a collection of data items stored together in an organized manner for shared use by various applications. A DBMS is a software package that retrieves the data from databases. This package performs numerous activities behind the scenes and allows many different users to have concurrent access to the database. The three-level architecture of DBMS shown in Figure 2.1 provides a highly organized management of data in a database. It maintains the mapping from the user views to the actual data in the physical storage device. This is what makes data independence and integration possible. This approach allows many users with different perspectives to view the data differently. Also, a DBMS provides a facility for users to send their queries without knowing how the data is stored.

At the external level or user view, which is the highest level of the database, the only portion of data that is of concern to the user will be viewed. At the conceptual level, the entire database is represented by one conceptual view, which describes all the records and relationships among them. The last or the lowest level, called internal view level, indicates the actual definition of the stored data and their representation in terms of fields [16].

A relational database is based on the simplest way of viewing data items in one or more tabular structures (table of rows and columns). Basically, a relational database

Figure 2.1: The three-level architecture of DBMS [16]

is a set of relations, each of which consists of a set of tuples. Relational database systems are ideally suited for parallel query processing since the uniform operation can be applied to uniform data. Results of one or many operations can be pipelined into another operation or many operations can work on series of data in parallel. Data partitioning, one of the parallel approaches for query processing, successfully emerged with distributed database architecture and is exploited in massively parallel database processing.

Another approach beside the hardware improvement is possible with the progress of the SQL language in relational database model in which relations can be created, modified, and viewed by SQL statements from a simple set of relational algebra. Also, several operators can be combined to produce the result, or the output of one operator can be sent to the input of the other. Also, the simplicity of SQL statements as an embedded SQL improves the productivity of procedural languages.

To take full advantage of parallel database systems, the database should be partitioned across the system. This technique has its origin in centralized database

because of the size of files. Partitioning a table of data, which is called a relation, implies distribution of data across the system. The reason for this distribution is to be able to parallelize the I/O operations among several disk units. To store large relations, the relations can be fragmented into a number of partitions and can be stored on the different disks. Partitioning should ensure that the database does not undergo any changes during fragmentation. If a relation has been fragmented, it should be fragmented without any loss of data. Partitioning can be either *horizontal* or *vertical*. Horizontal partitioning distributes tuples across the disks by using predicates on relation but vertical partitioning distributes data columnwise, which is the attribute of data. Partitioning technique allows the system to work in parallel in reading and writing to the disk as a normal I/O operation without needing any specialized hardware. Today, the commercial databases available in the market use horizontal partitioning in three different ways: *round robin*, *hashing* or *range* partitioning [18].

The *Round robin* technique distributes tuples across the disks in a cyclic fashion. Thus, accessing data by query command is simple and fast if the query considers the whole relation. However, this technique doesn't work well when tuples have a common value for the same attribute.

The *Hashing* technique distributes tuples across disks by using a hash function applied on a set of attributes of each tuple. The main advantage of this technique is that of reducing the overhead of starting queries on multiple disks. This mechanism has been used in current database systems such as Arbre, Gamma, Bubba, and Teradata.

The *Range Partitioning* technique clusters tuples based on their range of attributes that are frequently accessed. In this technique, there is a risk of *data skew*, which is caused by storing all the data with similar attributes in the same fragment and *execution skew* in which all the execution is limited to one particular fragment. Hashing and Round robin are less sensitive to data and execution skew since they do not cluster data. This mechanism has also been used in current database systems such as Arbre, Gamma, Bubba, and Teradata.

Figure 2.2: A Typical Query Processor [16]

Besides all these techniques and methods for increasing the speed, the other concern in DBMS is *query processing*. One of the components of DBMS is the Query Processor, which is responsible for generating the best plan to execute the query. A query submitted by the user has to be transformed to a relational algebraic expression and then the *parser* during this transformation checks the syntax and attributes used in the query and the existence of database structures. The next step is generating *Access Plans*. The method of accessing and the physical characteristics of the data are taken into account and from these plans, the optimal one will be chosen. In general, query processing involves the costs of processing, input/output, and communication. A query may be embedded in an application or directly submitted by the user. No matter how the query is entered, the goal of the DBMS is to minimize the cost of the operations. Thus, the parser first distinguishes all the tokens and converts these to internal code. An optimizer gets this code, tries to optimize the query by removing redundant code and generates an access plan. The access plan is sent to a data manager, which retrieves the data, manipulates it and returns the

result [16]. A typical query processor is shown in Figure 2.2.

Querying a database is the most important part of any database activity since all tasks require retrieving the data. The purpose of querying the database is not only satisfying the query but in a minimum amount of time. Therefore, maintaining a reasonable level of performance is essential. The response time of a query (the time difference between the time the query arrives and finishes) is the sum of waiting time and execution time. The overall response time essentially can be reduced by:

- Reducing the average waiting time of a query. This refers to the time difference between when a query arrives and when it starts executing.
- Reducing the execution time of a query. This refers to the time difference between when the execution of a query is started and when it is completed.
- Increasing overall throughput. This refers to the number of queries executed in a unit of time.

2.2 Query Processing in Shared-Nothing Parallel DBMS

The combination of parallel processing and database management gave rise to the concept of *parallel databases*. Parallel database systems exploit the parallelism available in multiprocessor systems to bring high performance database with high data availability. Parallel database systems use a message-based client/server operating system to connect the parallel processors interconnected by a high-speed network [18, 36]. Like distributed databases, parallel databases have been primarily designed to handle large amount of data. In the original single processor database systems, an increase in the amount of data to be checked increases the response time of a query significantly. With a parallel database system, the response time can be kept under control. Another advantage of parallel databases is the use of conventional processors, disk and memories to construct these systems. The modular architectures used today also enable the systems to grow linearly by adding more

hardware. In addition to the hardware advantage, these systems can also take the advantage of the parallelism inherent in the software.

Generally, the join operations are part of all operations in a database system and provide scope for potential parallelism in query processing. Basically, there are two approaches to parallelism:

- **Program Parallelism (PP):** which involves parallelizing several queries and can be exploited in inter- and intra-query parallelism.
- **Data Parallelism (DP):** which involves parallelizing one operation with different range of data and can be exploited in intra-operator or partial operator parallelism.

We discuss the significance of both forms of query parallelism in the following paragraphs.

In the flow of user accesses to the system, called transactions, three types of parallelism are inherent [32]. *Inter-query parallelism* enables the parallel execution of queries generated by concurrent transactions. *Intra-query parallelism* enables the parallel execution of concurrent and independent operations within the same query. The third type of parallelism, called *Intra-operator parallelism* enables the parallel execution of many sub-operations, within a single operation, that can be executed independently. Inter-query and Intra-query forms of parallelism can be obtained by using data partitioning. However, for our purpose of using shared-nothing architecture in terms of Network of Workstations for parallelizing query processing, distributing data across the system or workstations cannot be effectively done because of many reasons such as: security, integrity of data, and reliability. Therefore, these two approaches are slightly different than the real parallel database systems. Intra-operation parallelism can be obtained using program parallelism as well as data partitioning.

Inter-query Parallelism: Let us assume two queries similar to QUERY1 and QUERY2 in Figure 2.3 are submitted to the system. Our goal is to minimize the overall response time of each query. By allocating one workstation for the second query, the

Figure 2.3: Inter-query Parallelism

overall waiting time of that query will be minimized. Moreover, as part of the DBMS responsibility to optimize the query, each query can be divided into sub-queries and those independent sub-queries can be executed in parallel on other available workstations. More precisely, the common sub-queries can also share their results without executing twice. Another approach that significantly reduces the execution time of queries in practice is to store the result of a more frequent executed join operation for a predetermined amount of time. This of course depends on the availability of storage space which is part the responsibility of the DBMS [15]. The above techniques of identifying common sub-queries and more frequent sub-queries are part of the scheduler in DBMS. In fact, reducing waiting time of a sub-query results in decreasing the execution time of the other dependent sub-queries, which automatically reduces the execution time of the entire query and the overall response time.

Intra-query Parallelism: Let us consider an example which involves a query joining four relations R_1 , R_2 , R_3 , and R_4 . The two possible execution strategies, as shown in Figure 2.4, lead to two different approaches to parallelism. The first strategy shows

two independent task executions, where two branches of the tree can be executed concurrently on two idle workstations. Using multiple processors to join R_1 and R_2 in parallel with the join of R_3 and R_4 reduces the execution time of join operations within a single query [5]. The second strategy is an asynchronous pipelined approach, where the result of task S_1 (join operation of two relations R_3 and R_4) is sent to task S_2 , which then performs the join operation with R_2 . Finally, the result is sent to task S_3 which then performs the join operation of R_1 . The reasoning behind the name of the approach has to do with the fact that tuples are piped between tasks. But, unlike the synchronous pipelining used in sequential plans, different stages of the pipeline are not executed in lock-step mode.

Figure 2.4: Intra-query Parallelism

An execution plan is a partially ordered set of operators, such as index or data access, and predicate evaluators, sort, join, aggregation, etc. [24]. The number of operators depends on the complexity of queries. Obviously, the number of operators used in query limits the degree of parallelism obtainable by program parallelism. In fact, the actual degree is usually less than the upper bound due to dependencies between operators. For example, the merged join of R_1 and R_2 cannot start until the access and sort of R_1 and R_2 are completed.

Intra-operation Parallelism: The join operation is closely related to the Cartesian product. The Cartesian product of two relations concatenates each tuple of the

first relation with every tuples of the second relation. The join operation is used to combine related tuples from two relations into single tuples that are stored in the result relation. The join operation has been studied and discussed extensively in the literature because it is one of the most time consuming and data-intensive operations in any relational query processing. It is also important that joins be performed efficiently because they are executed frequently. Join operations may be implemented in many ways, and it has been found that certain techniques are more efficient than others in some computing environments.

The nested-loops join is the simplest join technique in query processing. One of the relations being joined is designated as the inner relation, and the other one as the outer relation. In practice, a nested-loops join is implemented as a nested-block join; that is, tuples are retrieved in units of blocks rather than individually. The exhaustive matching performed in this method makes it unsuitable for joining large relations unless the join selectivity factor is high. The simplicity of this algorithm has made it a popular choice for hardware implementation in database machines. Also, it has been found that this method can be parallelized with great advantage. The parallel version of this algorithm is more efficient than the most other methods [23, 30]. Considering a simple join operation between two relations, involving R_1 and R_2 with 1000 records each, the join operation takes millions unit of time to be evaluated. As a result, allocating one single workstation to process this operation is not efficient, due to an increase in the execution time of the query, which has a serious impact on the overall performance of the query. However, the hash-join technique takes less than nested-loops technique, if it is used in single node environment, so it is not suitable for parallel processing techniques. One of the fundamental approaches for nested-loops join operation is to fragment relations equally amongst all available workstations and send the join operation to each of them. Then, each workstation works on its own set of the fragmented relations and holds the partial results, which later on will be collected by a sender (master) processor to obtain the final result. The advantage of this technique is a reduction in execution time. However, there is a debate on how each relation should be fragmented and shipped

to all other workstations to obtain better performance results. Finer granularity of relations will lead to an increase in the communication delay. On the other hand, coarse granularity of the relations is not beneficial due to an increase in the execution time. Therefore, the system has to provide a balance between better performance and communication delay. There are several approaches proposed to handle this part of parallelism, these will be discussed in Section 2.3.

Performance Issues in PP and DP Parallelism: After a discussion of the three types of parallelism, we can conclude that:

Figure 2.5: Intra-operation Parallelism

- Inter-query parallelism reduces waiting time and execution time of a query by exploiting the common sub-queries within several queries.
- Intra-query parallelism reduces execution time of a query by employing several workstations to execute independent operations within single a query in parallel.
- Intra-operation parallelism reduces execution time of a query from the Data Parallelism point of view.

Finally employing these techniques, which have a direct consequence on the waiting and execution times, will increase the overall performance of a system.

Query processing in a NOW-based environment can get the benefits of parallelism by sharing load among several workstations. However, additional overhead is introduced since all workstations are under individual control, but we can get substantial performance improvement by using additional processor cycles and main memory rather than disk access. This improves performance without additional cost.

2.2.1 Load Distribution and Node Allocation

Basically, there are two techniques in parallel query processing for improving the performance of the jobs:

- In a time-sharing technique, a set of processors is shared among several jobs.
- In a space-sharing technique, a set of processors works together on a single job.

Paper [10] proposed a hierarchical processor scheduling policy, which takes the advantages of both time and space sharing policies. It is shown that the combination of both policies is beneficial in making scheduling decisions for large-scale systems. The hierarchical scheduling policy is based on the hierarchical run queue organization described in [7, 8]. In this organization, a set of queues is organized as a tree with all processors (with their local queues) attached to the bottom level of the tree. All incoming jobs are placed in the root queue. When a processor is looking for work, it consults the task queue at the upper level. If that queue is empty, then it turns the request to the parent node until it reaches the root queue. However, in order to reduce access contention at higher level, a set of jobs is moved one level down the tree and at the bottom of the tree this set size is reduced to just one task. The hierarchical policy exhibits superior performance than the time-sharing and space-sharing policies. The improved performance of the hierarchical policy over

the space-sharing is due to the fact that even though it uses space-sharing, because of self scheduling nature of the policy, the entire partition of free processors is not necessary to start executing a job.

Paper [3] shows the performance of multi-join queries based on the space sharing technique. It divides the scheduling of a query into two steps (i) scheduling the execution sequence of multiple joins within a query, and (ii) determining the number of processors to be allocated for the execution of each join operation. The cost of the query is related to the cost of each operation and the cost of each operation is usually proportional to the size of its operand and the selectivity factor. Since the result of a join operation may be larger than the size of each relation in the join, it is important to explore the general join sequence to reduce the cost of a join operation. To determine a join sequence of minimal cost, this paper uses the Huffman algorithm, which finds a join operation with a minimal cost and repeatedly applies the algorithm until all join operations are exhausted. Based on this heuristic, it first finds the sequence of join operations that minimize the cost and then applies the second step for processor allocation. The selection of the number of processors to perform the join operation is based on the execution time required for a join. However, it is important to find the optimum number, which optimizes processor utilization and avoids system fragmentation as much as possible. The top-down approach has been used in such a way that all processors are allocated to the root node and each internal node keeps the corresponding total execution time of the tree, which involves to the processor allocation procedure, then processors are divided into two sets for each sub-tree. However, insufficient number of processors compared to the depth of the tree makes the processor allocation algorithm complex and increases the overhead.

Paper [9] studied the performance of a distributed system by using load sharing policy and a dynamic load distribution policy and its two important components have been discussed. Dynamic load sharing tends to take the load of the system into account for any job transfer decision. Typically, for the transfer policy, one of the components uses a fixed value of load index threshold to determine whether the node

is overloaded or not. Therefore, the choice of load index has a considerable effect on the performance. Location policy, the second component, is responsible for finding a suitable node to transfer the load. Thus, it has to consult other nodes and collect their states. Location policy can be either centralized or distributed. In a centralized policy, the state information is collected by one node and other nodes would have to consult with this node for any transfer decision. In a distributed policy, the system state information is distributed to all nodes. The centralized policy has the advantage of providing near perfect load sharing because it has access to the entire system state information. However, the obvious disadvantage is a potential bottleneck and the failure of the coordinator (The central coordinator is responsible to manage all the activities in the system). The distributed policy eliminates these disadvantages; however, it may introduce extra overhead due to transmitting system state to all nodes. To overcome these problems, this paper proposed a hierarchical load sharing policy that combined the merits of both centralized and distributed policies. In this policy, instead of a single node maintaining the entire system state information, a set of nodes is given this responsibility. The system is divided into clusters and each cluster has a single node responsible for collecting the state information of its own cluster.

2.2.2 Scheduling Parallel Jobs

In a locally distributed system, the random arrival of tasks causes some workstations to be heavily loaded, while others are idle or lightly loaded. Load distributing improves performance of the system by transferring tasks from heavily loaded workstations to lightly loaded ones, where the tasks can take the advantage of a computing capacity that would otherwise go unused. The purpose of load distribution is basically to improve the performance, or in other words, to reduce the average response time of tasks. There are several approaches available for load distribution in distributed system, which are described in the following sections. Beside those techniques, our purpose is to extend load sharing within the operation, especially in database query processing.

Shivaratri, Krueger, and Singhal [37] describe several algorithms which are related to load scheduling. Generally, the algorithms for scheduling jobs can be broadly characterized as dynamic, static, or adaptive. The dynamic load distributing algorithms take the system state into account to make any scheduling decision, while in static algorithms, a priori knowledge of the system state is necessary for future decision. The adaptive load distributing algorithms are a special class of dynamic load distributing algorithms, in which the system adapts its activities by dynamically changing the parameters or policies. The most interesting part of load scheduling in parallel database system is the one that deals with dynamic load distribution. A dynamic load distribution algorithm has four components:

Transfer policy: A transfer policy determines whether a node is in a suitable state to participate in a task transfer either as a sender or a receiver. Many proposed transfer policies are threshold policies, which means that if a new task arrives at the node, the transfer policy decides that the node is in the sender mode if the node's load exceeds threshold T_1 . On the other hand, if the load at the node falls below T_2 , the transfer policy decides that the node is in the receiver mode (T_1 and T_2 are threshold indices). Alternative to threshold transfer policy, there is a relative transfer policy which decides whether the node is in a sender or a receiver mode based on the load of the other system nodes.

Selection policy: When the transfer policy decides that a node is in the sender mode, a selection policy selects a task for transfer. The simplest approach is to choose the newly arrived task that caused the node to become a sender. Such a task is relatively cheap to transfer, since it hasn't started processing yet. However, there are several factors involved in a selection policy:

1. The overhead incurred by the transfer should be minimal. For example, a small task carries less overhead.
2. The selected task should be long lived to compensate the transfer overhead.

3. The number of location-dependent system calls should be minimal. For example, the task may need to use resources from the original node such as clock, windows, etc.

Location policy: Once the transfer policy has decided to transfer the task, the location policy has to find a suitable partner for a node. The polling approach is widely used in most policies. A node polls the other nodes to find out whether it is suitable for load sharing. Polling can be either serial or parallel (multi-cast). The second approach is polling on a random basis. A node randomly peeks another node from the set of available workstations. The advantage of this technique is that the probability of finding a partner from a set of under-loaded nodes is high. But, if the system is overloaded, finding a partner will require many probes that will introduce more communication and overhead.

Information Policy: This policy decides when a node has to collect the information about the state of the other nodes in the system. There are three types of information policies:

1. *Demand-driven policy:* The node decides to collect the state of the other nodes when it becomes either a sender or a receiver.
2. *Periodic Policy:* This policy collects the information of the system periodically and it generally does not adapt to the rate of activities in the system. If all nodes are busy in this system, the load distributing is minimal. Thus, systems with very high loads are more suitable for this policy.
3. *State-change-driven Policy:* In this policy, nodes report their states to a state information collector (in centralized) or other nodes (in decentralized) whenever they have been changed to some degrees.

2.3 Load Sharing

Most join operations in queries can be parallelized since a nested-loops algorithm basically performs this operation. The body of a loop may execute several times and it often comprises a large portion of a program's parallelism. Moreover, loops provide a very rich opportunity for exploiting parallelism [28]. However, to exploit this technique, we need to identify an independent operation and explore data independence. Dependencies can be categorized into three different types: *resource*, *data*, and *control*. Resource dependency occurs when two different operations attempt to control one physical resource, such as two statements attempting access for a memory port. Data dependency exists when two statements reference the same memory location. Finally, control dependency occurs when one statement is waiting for the result of another statement. The purpose of load sharing at the lower level of a query, which is the join operation, is to distribute data amongst all available workstations to improve performance. The idea of loop scheduling will be studied in the following techniques: *Chunk self-scheduling (CSS)*, *Guided self-scheduling*, *Factoring scheduling*, and *Trapezoid self-scheduling*.

2.3.1 Chunk self-scheduling (CSS)

This technique assigns a fix-sized chunk to each processor at every iteration. Since the size of each chunk is fixed, this method produces high overhead in a heterogeneous environment. The trade-off for better performance is to find an optimum size of each chunk. Kruskal and Weiss [28] used probabilistic analysis and derived the equation for chunk scheduling.

$$T_p \approx (N\mu/p) + (Nh/pc) + \sigma(2c \log p)^{1/2} \quad (2.1)$$

where T_p is total time required on P processors, and

N is the number of iterations, and

μ is mean execution time with variance σ^2 , and

c is the chunk size, and

p is the number of processors, and
 h is the mean scheduling overhead.

The first term in this expression represents the expected time to execute N tasks with no scheduling overhead. The second term indicates the additional time for task scheduling and the last term represents the extra delay from idle processors due to unequal finishing times. This approximation shows that as c increases, the scheduling overhead decreases, proportional to $1/c$. But larger tasks result in an increase in the load imbalance, proportional to $c^{1/2}$. From the equation 2.1, we can see that, if the scheduling overhead h is zero, the optimal value for $c = 1$. If the scheduling overhead is large compared to task execution time, the optimum chunk size is $c = \lceil N/p \rceil$, since this chunk minimizes the last term. Although variation in execution time might tend to cancel out statistically, it may not be wise to rely on it in practice. Therefore, in an actual implementation, where we should probably consider the variable chunk size approach to reduce scheduling overhead, we could allocate a large chunk size at the start time and send small chunks toward the end, for minimizing unequal finishing times.

2.3.2 Guided self-scheduling (GSS)

The scheduling technique in GSS performs decreasing chunk-size allocation [28, 35]. GSS allocates a chunk of size $c_i = \lceil R_i/p \rceil$, to the processor making the i^{th} request for the chunk. The term R_i is the number of iteration to the processor making the i^{th} request for a chunk of work. Initially, $R_1 = N$ and the next term will be $R_{i+1} = R_i - c_i$. Therefore, it allocates larger chunks at the beginning, which reduces the scheduling overhead. GSS may introduce an unreasonable amount of scheduling overhead, if the chunk assigned is getting smaller towards the end of execution time. A simple variation specifies the minimum chunk size, c_{min} , to ensure that each processor is always assigned at least c_{min} chunk size.

2.3.3 Factoring scheduling (FS)

Factoring scheduling techniques proposed in [28, 26] can be viewed as a generalization of GSS and CSS. GSS is factoring where each batch contains one single chunk, and CSS is factoring where there is a single batch. Factoring schedules independent iterations in decreasing-sized chunks to each processor and later, smaller chunks are used to smooth over the unevenness of the earlier larger ones. The size of the chunks are chosen so that there are a high probability that they will be completed before the optimal finishing time, and hence, there will be enough work remaining to smooth over their unevenness.

2.3.4 Trapezoid self-scheduling (TSS)

In order to achieve both load balancing and small run time scheduling overhead, this technique assigns a large number of iterations to the first processor and linearly decreases the number of iterations until all iterations are exhausted. TSS obtains the best tradeoff between small overhead and load balancing, while reducing the number of mutually accesses to loop indexes. But the most important part of the scheduling is to balance the load among all workstations. Therefore the performance of Trapezoid self-scheduling is dependent on synchronization of the result and execution time of the loop iterations [39].

This approach compared with chunk self-scheduling (CSS) and Guided self-scheduling (GSS) has the advantage of both in terms of minimizing scheduling overhead and balancing workloads. The key principle of the TSS(f, l) method is that the area below the linear curve is a trapezoid. f and l are the size of the first and last chunks that can be selected by the programmer or assigned by the compiler. The number of chunks, N , is determined by f, l and I (total number of iterations) using the following equation:

$$N = \lceil \frac{2I}{f+l} \rceil \quad (2.2)$$

and the decreasing chunk size between scheduling steps, δ , can be obtained by

$$\delta = \frac{f-l}{N-1} \quad (2.3)$$

where f is the size of first chunk,

l is the size of last chunk,

I is the total number of iterations.

$$T(i) = \sum_{i=\alpha(i)}^{\beta(i)} L(i) \quad (2.4)$$

where $T(i)$ is the load of the i^{th} chunk,

$L(i)$ is execution time of the i^{th} iteration,

α is the *chunk starting point* at iteration i ,

β is the *chunk ending point* at iteration i .

Figure 2.6: $T(1)$ of the $TSS(I/2P)$

To apply $TSS(f, l)$, the values of f and l have to be predefined. However, these values are dependent on the execution time of the loop and would have a serious side effect on load imbalancing if they were chosen incorrectly. In order to obtain the size of the i^{th} chunk, it will be selected in such a way that $T(i)$, equation 2.4, is less than or equal to L/P . Thus, the first chunk is set to be $I/2P$ and the last chunk will have a value between the range 1 and $I/2P$. By substituting $(f, l) = (I/2P, 1)$ and $(f, l) = (I/2P, I/2P)$ in equation 2.2, the range of the number of chunks, N , is

deduced to be from $4P$ to $2P$. Figure 2.6 shows the size of first chunk under the loop with decreasing order of loop indexes.

2.4 Chapter Summary

In this chapter, we reviewed the related literature on DBMS and parallel processing systems. We mainly focused on shared-nothing system as a scalable and reliable architecture in parallel database systems with various partitioning techniques. The three useful parallelization techniques for query processing was also discussed.

Load distribution as one of the important tasks for load balancing has been focused on. Also, several techniques such as hierarchical processor scheduling policy, scheduling the execution sequence of multiple joins, and dynamic load scheduling in a distributed system were discussed.

Finally, the load sharing aspect of parallel query processing was studied in the context of several techniques. These techniques were mainly classified as Chunk Self-Scheduling (CSS), Guided Self-Scheduling (GSS), Factoring Scheduling (FS), and Trapezoid Self-Scheduling (TSS) and their detailed characteristics were discussed.

The distributed architecture we propose in the next chapter adopts these techniques to parallel query processing on a network of workstations.

Chapter 3

Distributed Architecture for Query Processing

One of the important performance requirements of a database system, from the user standpoint, is the response time of a query. Users expect the result of a query, even a complex one, available quickly. Processing large and complex queries in a reasonable time requires the use of parallel processing techniques. The response time of the query (R) is $R = W + E$, where W is the waiting time of the query, which is the time difference between query submission and the query execution, and E is the execution time of the query, which is the difference between starting the execution and completing the query.

The overall response time of a query can be reduced by reducing the average waiting time and execution time. These two factors can be reduced if a set of idle workstations is available for scheduling queries. The use of a network of workstations for parallel query processing has been proposed in [5]. The main motivation for using NOWs for parallel query processing, in general, comes from several studies that indicate that a large fraction (up to 80% depending on the time of the day) of the processing power available on a Network of Workstations (NOW) is wasted by idling cycle [15]. Substantial performance gains can be obtained by using these idle processing cycles. It has been demonstrated in [5] that the performance benefits from parallelizing a four-way join operation by using a set of workstations. However, there are several performance issues associated with using NOWs for database

query processing. These performance issues, mentioned in Section 1.3, include load balancing, fault tolerance, job migration and communication network latency. Communication network latency is the most important one, as NOW-based systems rely on message passing for communication.

In this chapter, we introduce the three architectures used in parallel query processing on NOW-based environment. Our contribution of the thesis is discussed in Section 3.2 along with the three major parallelization techniques (i.e., inter-query, intra-query, intra-operation parallelism) with detailed description of the components of the system. The overview presented in the next section is based on [6]. However, the detailed distributed architecture presented in the remaining of this chapter is a contribution of this thesis.

3.1 Overview

In a NOW-based architecture, the database management system (DBMS) would be centralized since every workstation has access to data in read-only mode. Updating data requested by any workstation has to go through the central database node, which is responsible for updating the database. A perspective of the system is similar to Figure 3.1 where one workstation has control over the database and others have access to database through this workstation. Every query indicates which relations are involved and the central database node transfers all required relations to the processor initiating the query. Once the data is selected for each query, all workstations can process the query independently with their own local resources.

A network of workstations is a kind of distributed-memory architecture in which no central control is present. Since this architecture is not centralized or in other words, each workstation is individually controlled, the database can neither be distributed among the workstations (like distributed database systems) nor be de-clustered along the relations (like parallel database systems). Furthermore, like in the traditional databases the security and integrity of data should be maintained [5]. As a result, using a network of workstations for database processing leads to a new

Figure 3.1: Centralized Database System in NOW-based System

configuration, in which the database is centrally controlled and the rest of the workstations communicate with each other by sending messages across an interconnection network to use an idle cycle of workstations for speeding up the query processing. This technique leads to some other aspects of parallelism, which were already discussed in section 2.2. Generally, parallel query processing can be classified into three different architectures: (i) Centralized architecture (ii) Distributed architecture (iii) Hierarchical architecture. The following sections briefly discuss each architecture and compare their advantages and disadvantages.

3.1.1 Centralized Architecture

Basically this architecture works in a master-slave mode. One workstation acts as a master (in this case the database node) and the rest of workstations work as slaves. The database node performs all the work related to query processing (i.e., query optimization, interpretation, coordination, etc.). Queries submitted at various workstations are forwarded to the database node and this node, by taking the system state information into account, allocates processors for the query. The system state

information could be non-query processing load and current query load. In this architecture, the database node is responsible for maintaining the state information of the system, whereas other workstations are responsible for informing the database node about their state changes.

The processor allocation decisions are made dynamically and could vary during the lifetime of a query. According to the workload information in the system, the database node allocates a set of processors to the job until the job is finished. Since the database node has the information on the entire system, load balancing can be done effectively in this architecture. The other advantage is that if two queries arrive with common join operations, the result of these join operations can be shared by both queries, without the operations being executed twice. The centralized architecture, however, suffers from a bottleneck problem in the database node as the size of the system increases. Therefore, this architecture has better performance for only small-sized systems.

3.1.2 Distributed Architecture

In this architecture, all query preprocessing activities (i.e., query interpretation, optimization, processor allocation, and scheduling) are done locally at the workstation that received the query. Due to the dynamic query arrivals and query complexity, some nodes might be overloaded while others are under-loaded [6, 37, 38]. In NOW-based systems, we have to consider a weighted combination of the background and query loads as an indicator of system load. Then, a simple threshold can be used to determine if a node is overloaded. Using threshold values classifies the state of the node as under-loaded (i.e., in the receiver state), or overloaded (i.e., in the sender state), or normally loaded (OK state). [20, 11, 37, 6] studied the load sharing in distributed system in different techniques and policies. The sender-initiated policy attempts to transfer work to an under-loaded receiver node by a probe mechanism. On the other hand, in the receiver-initiated policy, an under-loaded node sends a probe message to the overloaded node to get the load. The adaptive policy has also been studied for the case when the system acts as both a receiver and a sender in two

different situations. The two sender-/receiver-initiated policies will be extensively discussed in Section 3.2.

The distributed architecture introduces several performance problems in comparison with the centralized architecture. Since the load of the system varies among all workstations, load balancing is the major issue in this system. Unlike the centralized architecture, in this policy the load information is distributed and has to be gathered periodically in order to make proper processor allocation decisions, which introduces more communication overhead. However, unlike the centralized architecture, this system is scalable and is able to tolerate the presence of faulty nodes by providing fault tolerance.

Figure 3.2: Work transfer process in the hierarchical architecture

3.1.3 Hierarchical Architecture

The centralized architecture is simple and communication overhead is lower, but, the scalability is limited. On the other hand, the distributed architecture solves the scalability problem but the additional communication overhead and the associated performance degradation are undesirable. [6, 7, 8, 9, 10, 11] propose the hierarchical architecture that inherits the merits of the centralized and the distributed architecture while minimizing the associated disadvantages.

The hierarchical architecture is based on the hierarchical run queue organization. A set of queues is organized as a tree structure with all the nodes as leaf nodes. When a node is looking for work, it first requests from the parent queue. If that queue is empty, then it turns the request to the parent of the queue at one level up until the request reaches the root queue. However, in order to reduce access contention at higher levels, when there is a request, a set of work units is moved one level down the tree. The size of the set decreases progressively as the request goes down the tree. At the bottom of the tree, the set size is reduced to just one work unit. Figure 3.2 shows the process of scheduling jobs from the root queue to the processors.

3.2 Distributed Architecture

In this section, we introduce our system model of the distributed architecture. Our objectives is to implement the three parallelism techniques as described in Chapter 2 for parallel query processing. A distributed system is represented as a set of workstations connected by a local area network (LAN) and they communicate with each other by sending messages across the interconnection network. Since all workstations are general-purpose and heterogeneous machines, they are responsible for handling non-query jobs as well as query jobs (either submitted by the user or by other workstations). A job submitted by the user is either a query or a normal task. Thus, each workstation maintains three job queues, a *non-query queue*, *local-query queue*, and *remote-query queue*. The non-query queue stores all non-query jobs submitted by the user that can run locally. The local-query queue stores all query jobs

that can be executed locally and remote-query queue holds those query jobs that must be scheduled on other workstations. Each workstation gives preemptive priority to the non-query jobs, which can be calculated by a predefined function. There are additional activities at each workstation that have to be taken into account such as: communication messages, packing and unpacking of messages, initiating a probe message, sending a job, etc.

The following sections give the detailed design and implementation of distributed architecture as well as the performance issues such as load balancing, node allocation, and load sharing techniques. A priori knowledge is used to set the threshold parameters and probe limit; however, the information policy has been implemented in such a way that the system could use the load of the system dynamically to make any transfer or load sharing decisions.

3.2.1 Block Diagram of Distributed Architecture

In this section we present our system model of distributed architecture and all local processes in detailed description. Figure 3.3 shows this model and all components and processes, which execute on all workstations at the same time.

The *Workload Generator* is one of the components of this system that is responsible for generating a query and non-query load based on the workload and database model. At the initialization stage, this process reads the values of *mean inter-arrival time*, and generates loads in exponential distribution random arrival time. The generated loads are submitted to the *Scheduler* for further processing. In this system model, every process has a unique processor identifier (PID) that is used by processes to communicate with each other. Scheduler of each node is the only bridge to other workstations since it keeps the *PID* of other schedulers as well as local processes. Since the behavior of the components varies and they receive different messages, we maintain a table of message tags that are categorized in Table 3.1. In addition, for better understanding of the flow of interaction between processes, these message tags are shown in Figure 3.4. Processes can receive any kind of message tags from any other processes; however, for the process to be able

Figure 3.3: High-Level Block Diagram of the Distributed Architecture

Figure 3.4: Flow of Messages and Data

<i>Process</i>	<i>Tags Identifier</i>	<i>Tag</i>	<i>Description</i>
<i>Workload Generator</i>	SendQueryToSC	11	Sends query to Scheduler
	SendNonQuery	10	Sends non-query job to Non-query Queue
<i>Scheduler</i>	ReceivedNonQuery	10	Received Non-query jobs from Workload generator
	ReceivedQuery	11	Received query from Workload gen., Result Matcher
	SendQueryToLC	12	Sends query to Load Coordinator
	SendQueryToHost	13	Sends query to remote node
	ReceivedQueryFromHost	13	Received Query from Remote node
	ReceivedLoadRequestFromHost	21	Received Load request from remote node
	SendLoadToHost	30	Sends Load status to other node
	SendLoadToLC	31	Sends Load status to Load Coordinator
	ReceivedLoadRequestFromLC	41	Received Load request from Load Coordinator
	ReceivedFinish	42	Scheduler received a message from background job that it is done
	ReceivedJobRequest	43	Received job request from Load Coordinator
	AskTransferPolicy	47	Ask Transfer Policy for transfer decision
	ReceivedIdleHost	48	Received an under-loaded node
<i>Load Coordinator</i>	ReceivedQueryFromSC	12	Received query from Scheduler
	SendQueryToHost	13	Sends query to Remote node(Scheduler)
	SendPartialToRM	15	Sends partial result to ResultMatcher
	ReceivedLoadFromSC	31	Received load status from Scheduler
	RequestLoadFromSC	41	Requests load status from Scheduler
	RequestJob	43	Requests query from Scheduler
	ReceivedFinishedMsg	46	Received Finish Msg from Worker(Dojoin)
	AskTransferPolicy	47	Ask Transfer Policy for transfer decision
	ReceivedIdleHost	48	Received an under-loaded node
	AskLocationPolicy	49	Ask for an under-loaded node
	FirstTable	71	Sends first relation to Worker(Dojoin)
	SecondTable	72	Sends second relation to Worker(Dojoin)
<i>Transfer & Location Policy</i>	RequestLoadFromHost	21	Requests load status from other node
	ReceivedLoadFromHost	30	Received Load status from node
	ReceivedTransferRequest	47	Received transfer request from Scheduler or Load Coordinator
	SendIdleHost	48	Sends an under-loaded workstation to either Scheduler or Load Coordinator
	ReceivedLocationRequest	49	Received Location request from Load Coordinator
<i>Dojoin</i>	SendPartialToRM	15	Sends partial result to ResultMatcher
	SendFinishedMsg	46	Sends Finish message to sender of chunks
	FirstTable	71	Receives First relation from Load Coordinator
	SecondTable	72	Receives Second relation from Load Coordinator
<i>ResultMatcher</i>	ReceivedPartialQuery	15	Receives partial result from Dojoin
	RMSendToRM	15	Sends the merged result to the sender
	AllProcessesTIDs	1	Scheduler Broadcast all processes id generated in each workstation

Table 3.1: Message Tags

to differentiate between its own message and other messages, it will respond only to those whose tags are maintained. This mechanism makes the environment more secure and provides a high integrity in the flow of information.

Scheduler: This component as one of the essential part of load balancing and node allocation techniques, has three sub-components (*Load Information Gathering, Transfer Policy, and Location Policy*). These sub-components co-operate with each other in order to schedule the receiving job. When a non-query job arrives, the scheduler pushes it into the non-query queue and the job will wait there to get the service. But if the scheduler receives a query job, first it sets some information associated with the newly arrived query. This information represents the arrival time, the owner of the query (local or remote), etc. If the query is from a remote node, then it will increment the variable called *number of remote limit*. This variable keeps track of the number of times that the query has been transferred to another remote node. This value is required when the scheduler decides to send the job to another under-loaded node when it becomes overloaded. However, if the query is from a remote node and the remote limit value has reached the maximum value, then the query cannot be sent and has to be processed locally. This option prevents transferring query from one node to another if node, which holds the query, becomes overloaded. After the query arrives the scheduler pushes the new query into the local queue for further decision. Then, it asks the *Transfer policy* to decide whether to transfer the job or process it locally. In addition to scheduling the job, the scheduler has some other responsibilities such as:

- Responding to the requests of *Load Coordinator*, which are either current load status or job requests.
- Responding to the request of some other nodes about the current load status. If the scheduler sends an under-loaded status message to any node, it would expect to receive a query or a load-sharing job from that node.
- Initiating background jobs if there is any non-query job in the queue.

Transfer Policy: As we mentioned in an earlier section, load sharing improves the performance of a distributed system by distributing the load from a heavily loaded node to a lightly loaded node. A transfer policy, as one of the components of a dynamic policy, determines whether a job is processed locally or remotely. Typically, transfer policies use some kind of load threshold to determine whether the node is heavily loaded or not. Several load indices such as the CPU queue length, time averaged CPU length, CPU utilization, the amount of available memory, etc. have been proposed in paper [14, 37]. It has been shown that the choice of load indices has a considerable effect on the performance and among the above indices, CPU queue length was found to be the most effective one. The node is identified as a sender if a new task originating at the node makes the total load to exceed the threshold. The node is identified as a suitable receiver for a task if accepting the task transfer will not cause the node's load to exceed the threshold.

The transfer policy in this project uses two threshold values, T_{uload} for under-loaded status and T_{oload} for overloaded status. These thresholds are expressed in units of load. *Transfer Policy* process calculates the current load, l , of the workstation to decide whether to transfer the query or do it locally (detailed analysis of the load calculation will be discussed in section 3.2.2). Transfer Policy based on T_{uload} , T_{oload} , and *the cost of the query* schedules the query into either Local-query queue or Remote-query queue. In our system model, the load indices such as: Local-query and Non-query queues length, cost of a query, speed of workstation, interconnection bandwidth, and estimated execution time of a query are the possible values that the system requires to make a decision to do the job locally, remotely, or share with some other workstations. The procedure of the Transfer policy is given in Appendix A.2.

Location Policy: When a new job arrives at a node, the transfer policy described above decides whether to place a job in the local queue or to transfer it to a remote node. If the job is decided to process on remote node, the location policy is invoked. The location policy can be *Sender-initiated* in which the node that becomes overloaded initiates a probe up to a maximum of probe limit P_l randomly selected nodes, to locate a node with the under-loaded status. Thus, it sends a message to

the selected node and requests the load status. If the status is under-loaded, then the location policy places the job into the local queue of that under-loaded remote node. On the other hand, the *Receiver-initiated* would be invoked when the load of the current node is below the threshold. Thus, it sends a probe, up to a maximum of probe limit P_l randomly selected nodes, to locate an overloaded node. If the status is overloaded, then the overloaded node places the job into the local queue of an under-loaded node. Note that for both policies, probing stops as soon as a suitable target node is found. If all probes fail to locate a node, then the location policy reactivates the probing after a prespecified amount of time.

Both policies offer substantial performance advantages over the situation in which no load sharing is attempted. Furthermore, the sender-initiated policy is preferable for a light to moderate system load, whereas the receiver-initiated policy works better at a high system load.

In our model, which is based on the sender-initiated policy, when a transferred job arrives at a node, the node accepts the job and increments the *number of remote limit*. This job is sent to another under-loaded node if the *number of remote limit* value is less than the *transfer limit*. Otherwise, the accepted job has to be processed locally regardless of the current load. We make some assumptions for implementing this part. We assume that the lower threshold, $T_{load} = 1$. That is, load distribution is attempted only when a node is idle. The motivation is that a node is better off avoiding spending time on load distribution when there is work to do. Furthermore, the availability of an idle node increases the possibility of finding an under-loaded one in a pool of workstations [6, 5, 12]. The procedure of the location policy is given in Appendix A.3.

Load Information Gathering: This component, as a part of the scheduler, uses the *Information policy (demand-driven policy)*. When the state of a node is changed to sender, the location policy probes for an idle node. After probing a node randomly, the current load of the other nodes will be stored in *Load Information Storage* along with the time that has been probed. The location policy uses this information to probe the node if the load status is under-loaded or the last probing time is long

enough for another trial. Note that the longer time idle is better than the shorter time idle¹. Thus, location policy tries to probe those nodes that have a longer idle time.

Load Coordinator is responsible for getting the query from Local-query Queue. It maintains Intra-query, and Intra-operation parallelism based on the current load status of the node. This process provides load sharing at the join operation level, if the current load status is between the threshold values, T_{uload} and T_{oload} . But if the current load falls below T_{uload} (under-loaded status), it provides load sharing at the partial operation level. Note that if the join operation is large that can be done by several nodes, the under-loaded node will establish the master-slave mode and share the load with other nodes. Basically, the granularity of the work is dependent on the current load status of the node. At the partial operation level, Load coordinator asks the location policy for M_w idle nodes which will be calculated by dividing the size of relations by the fragmentation size subjected to the maximum of total available nodes in the system. Thus, M_w is always one less than the number of nodes in the network. After a successful probe, this process acts as a master and all *Dojoin* processes of the other nodes become slaves. Then, it sends partial operations to all *Dojoin* processes until the operation is exhausted. Then, it takes the next operation and repeats the same procedure until the query is done. Note that the intermediate result of a partial operation will be sent to *Result Matcher* by all *Dojoin* processes to be merged for the next level of the join operation. The M_w represents the maximum number of nodes that the load coordinator can work with, and the availability of M_w nodes is dependent on the load of the system at that particular time. If the load coordinator cannot get M_w *Dojoin* processes, it starts to partialize the operation with as many as nodes which can get and tries to probe at a later time if it requires more nodes. Furthermore, if the system is heavily loaded, the possibility of getting an idle node is lower. Therefore, it always sends one partial operation to its *Dojoin*

---

¹For example if a node is idle for 30 minutes, it probably will be idle for a few more seconds. On the other hand, if a node is idle for 5 seconds, the user may at this time is preparing another command.

process and tries to probe after a prespecified time. Note that the probe mechanism is always in parallel with other activities in Load Coordinator. The procedure of the Load Coordinator is given in Appendix A.4.

Figure 3.5: Two Types of Execution Plan

Result Matcher: To process a query, each workstation at the time of query arrival, maintains a *flag* for each operation of the query, that is initially set to zero. Each flag indicates the status (done or not) of the join operation at its level. If a workstation decides to transfer the entire query to another workstation, all flags will be removed. Otherwise they remain until the query result is ready. The corresponding flag to the level is set to 1 after the result of the join operation is ready. Thus, the parent of a join operation can proceed to the upper level of the query. Consider two examples with two different tree structures:

The first one deals with a query with a balance tree structure, as shown in Figure 3.5(a). The result of QUERY1 is not ready until all operations receive their results. Assume that the result of J_3 and J_5 are ready. Thus, $F_3 = 1$, $F_5 = 1$, $F_4 = 0$, and $F_6 = 0$. As a result, two join operations J_1 and J_2 cannot proceed.

The second example, as shown in Figure 3.5(b), is a query with left-deep tree. In this

case the result of join operation at level₃ cannot proceed to level₂ until F_3 becomes 1, and the result of join operation at level₂ cannot proceed to level₁ until F_2 becomes 1, and so on.

After receiving the partial results from the *Dojoin* process which reside in *Pending Queue*, *Result Matcher* matches their corresponding query information. If all the results (based on their flags) are available then it applies the parent operation to the receiving result and sends it to the local-query queue of the current workstation for further scheduling. It is also responsible for keeping track of all operations with their corresponding information.

3.2.2 Load Sharing Within a Query

In the distributed architecture, explained in Section 3.2, all query activities are done locally at each workstation to prevent the bottleneck problem for the DBMS node. As soon as a query is submitted to the workstation, the associated data (such as an execution plan, the relations' name, the size of each relation, etc.) is assumed to be transferred with it. The execution plan of the query can be either *Balanced Tree* or *Left-deep Tree*, as shown in Figure 3.5. In our system model, the aim is to balance the system load to improve the response time of the queries thereby achieving better performance. Due to the cost and dynamic arrival of the query, some workstations might be overloaded while others are under-loaded. Therefore, the scheduler as one of the components of this system is responsible for balancing the load of the system. One of the responsibilities of the scheduler is to calculate the current load of the workstation. The Scheduler of one workstation has to be able to respond to the request of another workstation whenever it has been asked for its current load. This response will be used in the load sharing decision of the requesting workstation. The Scheduler decides whether or not to send the job based on the current load of the workstation. the parameter l_i , which is a unit of load, is the current load of workstation i . l_i is a function of the length of both non-query and local-query

queues with the cost of the query as a unit of load:

$$l = f_1(B) + f_2(Q) \quad (3.1)$$

where B is the non-query queue, and

Q is the local-query queue, and

f_1, f_2 are two different functions that return an integer numbers.

If we assume a priority for non-query jobs, a , and the total number of join operations as a cost of query, then the above equation will be:

$$l = a.f_1(B) + \sum_{i=1}^{|Q|} g(q_i) \quad (3.2)$$

where a is a predefined constant value and,

f_1 returns the number of jobs in the non-query queue, and

$|Q|$ is the number of query in local-query queue, and

$g(q_i)$ is the cost of the i^{th} query in the local-query queue.

There are two prespecified threshold parameters, T_{uload} and T_{oload} which represent the threshold value for under-loaded and over-loaded workstations respectively. The current load of workstations between these two values is called *OK* status. The above equation will be evaluated at each workstation when either a new job arrives or a load status is requested. In order to maintain consistency among workstations, the following rules have to be taken into account by all workstations:

1. If an under-loaded node receives a request from any node for sharing jobs and then becomes overloaded, it will accept the request and will not transfer the remote job to any other nodes².
2. If an under-loaded node receives a remote partial operation, it wont transfer the remote job to any node under any circumstances, due to the cost of transferring.

---

²A node can send an accepted remote job to another node, if the *number of remote limit* associated with the job is less than the prespecified *transfer limit*.

3. If an overloaded node could not find any under-loaded node to share the job, it would start processing the job until the availability of some under-loaded nodes. While it is processing the job, it could send a probe message to locate an under-loaded node.

In all the above rules, we assume that there is no workstation failure and all workstations are able to finish the job in desired time.

The second parameter, which is essential in terms of communication overhead, is the maximum probe. P_l represents the maximum probe for each workstation trying to locate the under-loaded workstation for sharing the load. Irrespective of the policy (sender-initiated, receiver-initiated, or adaptive), each workstation probes up to P_l times to find a workstation to share the load. If such a workstation is found, work is transferred to that node for remote execution [6]. However, transferring the work is related to the current load of the sender workstation. Sending an entire query puts less overhead on a sender workstation than sending an operation or a partial operation. Thus, the granularity of work transferred can be defined as:

- An entire query along with the relations, if the sender workstation is overloaded, or
- An operation of a query or sub-query along with the relations, if the sender workstation is in the OK status, or
- A partial operation along with fragments of relations, if the sender workstation is under-loaded³.

The third factor, that is very important to decide whether or not the workstation can share the load with as many workstations as possible, is based on the bandwidth of the interconnection network, communication speed, processor speed, and the cost

---

³When the sender workstation is under-loaded, first it sends one partial operation to itself and while it is doing the join operation, it sends probe message to some other nodes to share the load in order to reduce the response time of a query. Note that the probe mechanism is in parallel with other activities.

of the query. The following equation is applicable to each workstation based on [5] for estimating the execution time of a query on a single node (e_1):

$$e_1 = \sum_{i=1}^J \bowtie_i \quad (3.3)$$

where J is the total number of join operations within the query, and

$\bowtie_i$ is an execution time of two relations ($R_j \bowtie R_k$).

The execution time of a query can also be calculated from:

$$\bowtie_i = B_j + (B_j * B_k) + \frac{(\alpha_{R_j R_k} * |R_j| * |R_k|)}{bfr_{R_j R_k}} * t_d \quad (3.4)$$

where $\alpha_{R_j R_k}$ is the join selectivity between R_j and R_k ,

$bfr_{R_j R_k}$ is the blocking factor,

$|R_j|$ and $|R_k|$ are the size of relations in terms of the number of tuples,

t_d is the average disk block access time,

B_j and B_k are the number of disk blocks associated with relation R_j and R_k .

Let assume that a query Q_1 with four relations R_1 , R_2 , R_3 , and R_4 , is submitted to workstation W_1 . If W_1 is overloaded then it would transfer the entire query Q_1 to other workstation (assume after probing, one idle workstation is assigned to Q_1). But if the load of W_1 is OK or under-loaded, the scheduler of W_1 calculates the execution time of a query e_1 , by applying equation 3.3. If we assume W_1 can employ one workstation to share the load, $M_w = 2$, the following equation will give an estimate of execution time for two workstations.

$$e_s = \frac{e_1}{M_w} + (M_w - 1)[C + C^R + m] \quad (3.5)$$

where

e_s is an estimate of the total execution time with load sharing,

$\frac{e_1}{M_w}$ is an estimate of the execution time on M_w workstations,

C is an estimate of the communication time for sending fragments of relations to another workstation,

C^R is an estimate of the communication time for receiving the result from other workstation,

m is an estimate of the merging time of the results, and

M_w is the maximum number of workstations that can be employed for better load sharing.

If $e_1 > e_s$, then W_1 is able to send the partial operation to another workstation. In case when the workstation is under-loaded then after estimating the execution time e_1 , W_1 finds M_w and sends the partial operations on each of available workstations. All these cases assume that after probing, W_1 is able to find an under-loaded workstation to share the load. Otherwise, it has to do the job locally. Note that the communication time and speed of workstations are assumed to be the same for all workstations, as well as the granularity of the job for all workstations.

Probing for the target workstation will stop as soon as the maximum number of under-loaded workstations needed is satisfied or exceeds probe limit. The maximum number of workstations, M_w , is the estimated number of workstations on which the scheduler can schedule the current job. However, it doesn't mean that if the scheduler could not get M_w workstations, it has to wait for the availability of workstations. This number represents that more than M_w workstations does not reduce the response time because introducing more workstations with finer grain jobs on each increases the communication time. On the other hand, having fewer workstations with coarse grain jobs increases the response time. Therefore, there has to be a balance between M_w and the granularity of the job.

3.3 Chapter Summary

This chapter presented a detailed discussion of the distributed architecture for parallel query processing and provided a detailed discussion of the responsibilities of each component. A number of basic concepts from the parallel query processing have also been adopted in the distributed architecture.

This chapter also explained how the load of each workstation can be calculated by deriving general equations and how an under-loaded node can be selected from a pool of workstations. A three-case method for load balancing and load sharing was proposed based on the granularity of the job and the current load status of the node.

Chapter 4

Experimental Environment

The network of workstations used in this experiment is based on Pentium processors. We also use a public domain software package Parallel Virtual Machine (PVM) for message passing. Our goal is to evaluate the performance of the distributed parallel query processing in NOW-based system on two types of queries (Balanced and Left-deep tree structures). In this chapter, we introduce the NOW-based system environment and the PVM package in detail, as well as the database and workload models used in these experiments.

4.1 NOW-based system Environment

The network of workstations used in this experiment is a set of Pentium processors interconnected through two communication networks as shown in Figure 4.1. The current configuration consists of 6 Pentium and 4 Pentium Pro nodes. All nodes have 32 MB of memory and node n_0 (Triton node) is the only node with disk space. This node facilitates some basic jobs for the users such as login to the system. The two interconnection networks are 10 Mbps and 100 Mbps Ethernet high-speed networks. The 10 Mbps "service network" is used for booting the system, providing the basic communication and remote file service through NFS, while the 100 Mbps network provides a high-speed communication network. This system, for the external user, looks like a set of workstations; however, from the inside it can be configured as one

parallel machine.

Figure 4.1: Network of Workstations Configuration

Nodes can work in two different modes, *Shared* or *Private*, which depends on the type of application. Node allocation allows the user to allocate the nodes if the application is desired to have no interference with other users. All nodes, except node n_0 , can be allocated or shared. Furthermore, we assume that there is no node failure in this system.

4.2 Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) is a software package that allows a heterogeneous collection of computers connected together by a network to be used as a single large parallel computer. The PVM description in this section is based on [21, 22]. The

PVM software provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware. Thus, large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one data type to be exchanged between machines having different data representations.

The PVM system is composed of two parts. The first part is daemon, called *pvmd*, that resides on all the computers making up the virtual machine. The second part of the system is a library of PVM interface routines. It contains all primitive functions that are needed for cooperation between tasks of an application. This library contains message-passing routines, spawning processes, modifying the virtual machine, and some utility commands.

The most important disadvantage of PVM is the lack of communication among multiple distributed virtual machines. Several distributed virtual machines cannot collaborate, merge, or split. It means that there is only one single user per virtual machine.

The current version of PVM 3.0 supports the following features:

Resource Management: Since PVM is dynamic in nature, hosts can be added/deleted from the virtual machine at any time either from the console or from within the user's application. It allows the application to interact and manipulate the computing environment to provide a powerful paradigm for load balancing, task migration, and fault tolerance.

Process Control: This refers to the ability to start and stop tasks, to find out which task is running, and where they are running. PVM has a set of resource control functions that allow some basic resource queries to be entered to control the flow of process.

Message Passing: PVM provides a blocking send for message passing as well as two different ways of receiving message, blocking and non-blocking receive.

Dynamic Task Groups: Different topology cannot be configured by using PVM.

However, it allows the users to organize their tasks in a group based on desired communication among them. Thus, a task can join or leave a group at any time.

Fault Tolerance: PVM is able to withstand host and network failures. It provides polling and notification primitives to allow fault-tolerant applications to be built. Task can register with PVM to be notified when the status of the virtual machine changes or when a task fails. If a task fails, the receiving task will get a notify message in place of any expected message. This allows the task an opportunity to response to the fault without hanging or failing.

PVM has been accepted by computing community and it has been used for solving large-scale problems. Therefore, in our system model, we have designed the distributed architecture based on the PVM software and used as a message passing interface.

4.3 Database Model

For the purpose of our simulation, a set of tables is created. These tables are stored as regular files under Unix file system of *Triton* node, and all nodes have access to these tables through the DBMS node (one dedicated workstation to database activities). In other words, the DBMS node reads the tables and ships them to the node when they are requested. The size of tables varies between 5000 to 9000 tuples, each tuple consisting of two attributes¹. Totally, there are 16 tables and two of these tables have unique key numbers which are sequentially generated. Thus, there is no duplicate key in these two tables. But the rest of tables have some common key values, which are generated with the normal random number distribution with the same mean value and different variance.

We maintain the *selectivity factor* among tables for the purpose of join operation, which represents the fraction of tuples that qualify compared to the number of tuples

---

¹For the simplicity of our experiments, we consider only *key* and *value* as two attributes for each tuple and small-sized tables. However, the system does not impose any restrictions on the size of tables, tuples, and attributes.

Table	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	5.9	1.7	1.4	0.6	1.5	1.6	0.7	1.6	1.2	1.1	1.7	2.0	0.9	1.1	1.6	1.6
1	1.7	4.9	1.4	0.5	1.4	1.5	0.7	1.5	1.2	1.1	4.9	1.7	0.9	1.1	1.5	1.5
2	1.4	1.4	2.9	0.5	1.2	1.3	0.7	1.3	1.1	2.2	1.4	1.4	0.8	1.0	1.3	1.3
3	0.6	0.5	0.5	2.9	0.5	0.6	0.4	0.5	0.5	0.5	0.5	0.6	0.5	0.5	0.6	0.5
4	1.5	1.4	1.2	0.5	3.5	1.3	0.7	1.3	1.1	1.0	1.4	1.5	0.8	1.0	1.3	1.4
5	1.6	1.5	1.3	0.6	1.3	3.9	0.7	1.4	1.2	1.1	1.5	1.7	0.8	1.1	1.5	1.4
6	0.7	0.7	0.7	0.4	0.7	0.7	3.4	0.7	0.6	0.7	0.7	0.7	0.6	0.6	0.7	0.7
7	1.6	1.5	1.3	0.5	1.3	1.4	0.7	4.7	1.1	1.1	1.5	1.6	0.8	1.0	1.4	1.4
8	1.2	1.2	1.1	0.5	1.1	1.2	0.6	1.1	5.0	1.0	1.2	1.3	0.7	0.9	1.1	1.2
9	1.1	1.1	2.2	0.5	1.0	1.1	0.7	1.1	1.0	2.2	1.1	1.1	0.7	0.9	1.1	1.1
10	1.7	4.9	1.4	0.5	1.4	1.5	0.7	1.5	1.2	1.1	4.9	1.7	0.9	1.1	1.5	1.5
11	2.0	1.7	1.4	0.6	1.5	1.7	0.7	1.6	1.3	1.1	1.7	4.8	0.9	1.1	1.6	1.7
12	0.9	0.9	0.8	0.5	0.8	0.8	0.6	0.8	0.7	0.7	0.9	0.9	4.6	0.7	0.8	0.8
13	1.1	1.1	1.0	0.5	1.0	1.1	0.6	1.0	0.9	0.9	1.1	1.1	0.7	3.0	1.1	1.1
14	1.6	1.5	1.3	0.6	1.3	1.5	0.7	1.4	1.1	1.1	1.5	1.6	0.8	1.1	4.8	1.4
15	1.6	1.5	1.3	0.5	1.4	1.4	0.7	1.4	1.2	1.1	1.5	1.7	0.8	1.1	1.4	4.3

Table 4.1: Selectivity Factors

in the Cartesian product of the two tables and is given by Equation 4.1. Note that the selectivity factors shown in Table 4.1 are multiplied by 1000 to be fitted in the page.

$$\alpha_{R_1 R_2} = \frac{|(R_1 \bowtie R_2)|}{|(R_1 \times R_2)|} = \frac{|R_1 \bowtie R_2|}{|R_1| \times |R_2|} \quad (4.1)$$

where $\alpha_{R_1 R_2}$ is the join selectivity factor,

$|R_i|$ is the size of relation in terms of the number of tuples.

For example, if the selectivity factor between R_1 and R_2 , with 1000 and 2000 tuples respectively, is 0.01, $R_1 \bowtie R_2$ will generate a table with an intermediate result of 20,000 tuples.

4.4 Workload Model

For the purpose of generating a workload, three types of random number generators are used. One of them generates query with the mean inter-arrival time λ_1 and coefficient $CV_1 = 1$, and second one generates a background load with the mean

inter-arrival time λ_2 with coefficient $CV_2 = 1$. These two random number generators use exponential distribution. The third random number generator uses uniform distribution and produces a number between 1 and 16 corresponding to each query.

The sixteen types of query are produced in advance and stored in a file. Based on the database model and selectivity factor among tables, these queries are either growing-sized or shrinking-sized in their result. In our experiment, two types of queries (Balanced tree and Left-deep tree structure) are studied². But our focus is on balanced query structure and evaluating the performance of the system since the parallelism can be exploited with this type of query. The left-deep tree query has no use of inter-query and intra-query parallelism in distributed architecture since one join operation is available in a query at any time.

---

²For simplicity we generate queries with balanced tree structure based on the selectivity table and store them in the file. Thus, the selectivity factors among tables do not completely cover the query with Left-deep tree structure in terms of growing-sized or shrinking-sized in results.

Chapter 5

Performance Evaluation

In this chapter we study the performance of the distributed architecture for parallel query processing. It is intended to evaluate the performance of the system for different inter-arrival times and fragmentation sizes. For all experiments, a batch strategy has been used to compute the confidence intervals (at least 30 batch runs were used). This strategy produced 95% confidence intervals that were less than 5% of the mean response times when the inter-arrival time is low to moderate (i.e., heavily loaded system) and less than 1% for moderate to high inter-arrival times (i.e., lightly loaded system).

We have used several parameters in the experiments (explained later in this chapter). Section 5.1 and Section 5.2 present the justification for selecting values for these parameters. Section 5.3 discusses the performance of the system as a function of inter-arrival time. Section 5.4 presents the scaleup and speedup of the system. In Section 5.5, we discuss the performance of the system for different fragmentation sizes. The impact of a background load is presented in Section 5.6. Finally, the performance of the system under two different query types is presented in Section 5.7.

5.1 Selection of Parameters

The performance of our system relies on several parameters that have to be predefined in order to run the simulation. These parameters are shown in Table 5.1. After

Parameter	Value	Description
T_{load}	1	Lower threshold limit
T_{oload}	4	Upper threshold limit
P_l	3	Probe limit
T_s	2	Probe message reinitiation time (seconds)

Table 5.1: Parameter Values

conducting several experiments, we derived that the minimum execution time for the smallest join operation, without load sharing, is 2.46 seconds. Also, decreasing the value for T_s results in increasing the overhead of *Transfer policy* since it reinitiates probing shortly after each probe failure. Thus, the optimum value for T_s is set to 2 seconds.

Figure 5.1: Performance Sensitivity to Probe Limit

A second preliminary experiment was conducted to study the performance sensitivity of the system to probe limits. The results are presented in Figure 5.1. For

this experiment 8 nodes were used and the inter-arrival time is set to 3.0 seconds and no intra-operation is permitted. Also, the threshold values, T_{uload} and T_{oload} , are set to 1 which causes the node to send a probe after receiving a job.

As depicted in Figure 5.1, when the probe limit is 3, the optimal performance is achieved. Note that when the probe limit increases beyond 3, the performance of the system degrades rapidly. This is due to the fact that an increase in communication and waiting times are incurred. Therefore, we use 3 as the probe limit for all the experiments unless otherwise specified.

5.2 Upper and Lower Bound Thresholds

For the purpose of finding the upper and lower threshold values, we conducted several experiments with the following parameters: $P_l = 3$, $T_s = 2$ seconds, no partial operation and inter-arrival time is set to 2.0 seconds. Figure 5.2 shows the impact of different threshold values on the performance of the system. It can be observed that the arrival of jobs forces the system to increase the load sharing activities at certain thresholds. For example, if four jobs arrive at a node, the node attempts to distribute the jobs above the threshold limit to some other nodes, while the jobs below the threshold limit wait in the queue to get the service thereby causing an increase in the response time. In general, increasing the threshold value decreases the scope of load sharing and decreases the probe rates.

Note that performance improves until the threshold reaches 4 and further increase in threshold limit beyond 4 would result in performance degradation. The reason for the initial improvement is that increasing the threshold limit from 1 increases the probability of finding a node that is below the threshold limit and therefore results in increased job transfer rate. Further increase in the threshold limit results in decreasing the number of attempts at load distribution which causes reduced job transfer rates. However, at some point (4) it can be observed that the threshold produces an optimum response time, which results in better performance. Thus, the threshold values based on this analysis, for the rest of the experiments

are set as $T_{uload} = 1$ and $T_{oload} = 4$ for the lower bound and upper bound thresholds respectively.

Figure 5.2: Sensitivity to Different Threshold Values

5.3 Performance as a Function of Inter-arrival Time

In this experiment, we evaluate the performance of the system with different inter-arrival times. A high inter-arrival time results in fewer jobs arriving at the system, thereby a given job holds on to more resources. Therefore, we expect a decrease in the average response time. Also, the system saturates if inter-arrival time is less than 2 seconds.

Figure 5.3 shows the performance of the system on 8 nodes while varying the inter-arrival time. The fragmentation size is set to 200,000 bytes, which means no

Figure 5.3: System Performance on Different Inter-arrival Time

partial operation is permitted at least at the first level of join operations. It can be observed that when the system is lightly loaded the average response time is 24.31 which is 6.62 times less than 161.04 at heavily loaded status. It is clear that waiting time forms a significant part of the total response time, whereas the execution time is almost constant in all the cases. The important point to note from the experiment is the communication time. It shows that the communication cost increases. This is because the transfer rate and node probing required by each node to transfer a job are relatively high when the system is heavily loaded, whereas in a lightly loaded situation the transfer rate reduces. According to the above analysis, it can be concluded that the major part of the response time is the waiting time. But if the arrival of the query is relatively slow, the waiting time of the query has a little effect.

5.4 System Speedup and Scaleup

In this section, experimental results for the speedup and scaleup of the distributed architecture under heavy and light loads are presented.

A parallel system demonstrates two key properties (i) *linear speedup*: twice as much hardware can perform the task in half the time, and (ii) *linear scaleup*: twice as much hardware can perform twice as large a task in the same time.

$$speedup = \frac{\text{small system response time}}{\text{large system response time}}$$

Scaleup measures the ability to grow both the system and the problem. Scaleup is defined as the ability of N_times larger system to perform an N_times larger job in the same elapsed time as the original system.

$$scaleup = \frac{\text{small system response time on small problem}}{\text{large system response time on large problem}}$$

If this equation evaluates to a constant for different system sizes, then the scaleup is said to be linear.

Figure 5.4: System Speedup

Three experiments were performed to evaluate the speedup and scaleup of the

system under heavy and light loads. The number of processors varied while the fragmentation size was fixed at 200,000 bytes. Figure 5.4 shows the performance of a lightly loaded system (the shaded bars). It can be observed that the speedup of a 2-node system compared to a 1-node system is 1.92, and the speedups of 4-node and 8-node systems are 3.37 and 3.44 respectively. Note that there is no performance difference between 8-node and 4-node systems since the system is lightly loaded. In other words, the query has enough time to use all resources in an efficient way since the gap between the arrival of the queries into the system is large and the size of relations involved in the query are small. As a result, the system with four nodes is able to do the query and by adding extra nodes, system does not gain any performance.

Figure 5.5: System Scaleup

Figure 5.4 also shows the performance of a heavily loaded system (the black bars). The job inter-arrival time is set to 10 seconds. The number of processors is varied from 2, 4, and 8 nodes, while there is no response time for 1-node system due to saturation¹. It is shown that the 4-node and 8-node system can service the

¹When the arrival of the jobs is faster than the service time, then the system goes into a

same amount of load 2.27 and 3.7 times faster than 2-node system. From these two experiments, we can conclude that the distributed architecture achieves good speedups under both heavy and light workloads.

The third experiment deals with the system scaleup. We assume that doubling the mean inter-arrival time (i.e., 2 second to 4 seconds) results in half the amount of work to be done. Thus, for 8-node, 4-node, 2-node, and 1-node system the inter-arrival time of job is set to 6, 12, 24, and 48 seconds respectively. The result in Figure 5.5 shows that the system has good linear scaleup and for 1-node the mean response time is proportionately high. One of the reasons could be that with a 1-node system most of the local processing of the job is done by only one node, while this overhead in systems with more than one node would be amortized among several nodes. The second reason is that the power of one node is not always $1/2$, $1/4$, or $1/8$ of the power of other type of nodes in 2-node, 4-node, and 8-node systems. Therefore, for 1-node system, the performance is much higher in terms of response time. To show this fact the following experiment has done to find the power of each node as a *processor unit* (PU). Since the system is heterogeneous with processors of varying speeds, this experiment is done by running 16 types of queries on all nodes without considering waiting time to find out how fast each node can process these queries. We find out that P133's, P166's, and P200's execute in 23.80, 19.05, and 9.00 seconds in average. As a result, the P200 is 2.12 times faster than P166 and 2.64 times faster than P133. Also, P166 is 1.25 times faster than P133, which is almost the same power. The Table 5.2 shows the power of each type of node in the *Triton* system.

From the scaleup result, we conclude that the ratio of the small system to large system (i.e., response time of 2-node to 4-node system) based on the scaleup equation for 2-node to 4-node and 8-node systems are 0.93 and 0.96 respectively. Thus, the distributed architecture successfully achieved linear scaleup.

Finally, as we mentioned earlier, the distributed architecture is the only scalable system among centralized and hierarchical architectures. This system has the ability

---

saturated state. In this case, the response time keeps growing as the waiting time of the job grows.

Node Type	Processor Unit
P200 (n_3, n_7, n_8)	264
P166 (n_5, n_6)	125
P133 (n_1, n_2, n_4, n_9)	100

Table 5.2: Processor Power in *Triton*

to manage all resources and use them efficiently without losing the scalability feature. However, in all experiments, we use only 8 nodes and limit the size of the relations because of the hardware limitations. This might raise a question that how we ensure the scalability issue if we increase the size of the system. This situation in a real environment will create a bottleneck problem for a DBMS node if we assign a single node to the database node. However, to rectify this problem, we can also use a distributed or a client/server technique for a DBMS node if the number of workstations increases. In other words, a DBMS node can be configured as a set of nodes that are working together to handle the request.

5.5 Sensitivity to Fragmentation Size

Until now we assumed no partial operation at least at the first level of a join operation. However, the load sharing aspect of the system can be exploited in intra-operation parallelism. Generally, a large fragmentation size results in fewer partial operations. There is a trade off between the fragmentation size and the response time. In this section we study the impact of the fragmentation size on the performance of the system. Increasing the number of partial operations increases the communication cost, which becomes a significant part of the total response time as shown in Figure 5.6. It can be observed that as the size of the fragment decreases the response time also decreases until the fragment size 20,000 bytes. After this point the response time increases again. The reason for this behavior is that the fine-grain data introduces more overhead and increases communication which directly affects

Figure 5.6: Sensitivity to Fragmentation Size

the response time.

The second observation is that as the fragmentation size decreases, the communication cost increases irrespective of the system workload. This is because without partial operation the system transfers the entire relations to other node, which is the only cost. However, by allowing partial operations, system has to calculate the size of each partial, send a probe message, wait for the respond, and transfer the fragment. These overheads in addition to the communication cost are responsible for increasing the response time. Thus, the smaller the fragment size the higher the communication cost.

Note that the response time of a heavily loaded system is higher than a lightly loaded system, whereas the communication cost of a heavily loaded system is lower than a lightly loaded system. The reason is that the system under light workloads has a higher possibility to share the load with some other nodes, while under heavy workloads, the system goes above the lower threshold limit and tries to balance the load by sending the join operation (sends only the execution plan of the join

operation, not the relations) to another under-loaded node rather than sharing the load by sending partial operation (sends join operation along with fragments of relations) to some other under-loaded nodes.

Further observation from the results for both cases shows that the optimum fragmentation size is 20,000 bytes for which the response time is low.

5.6 Impact of Background Load on Performance

So far we assumed that there is no background load in the system. This section is intended to study the impact of a background load on the performance of query processing. A background task, as a load of any general-purpose environment, increases the response time of a query job. We try to generate some background loads for each node as it might happen in a real environment. In this experiment 8 nodes are used and the fragmentation size is set to 200,000 bytes. The inter-arrival time of the query is varied between 2.0 to 10.0 seconds, while the inter-arrival time of the background load is set to 3 seconds with the mean service time of 5 seconds. Furthermore, each background load at the processing time has been set to use between 40% to 50% of the CPU time. Figure 5.7 shows the delay caused by a background load. In most cases, it is between 21% and 47% except for the case where the inter-arrival time of the system falls below 3 seconds. Also when the system is lightly loaded, the overall performance decreases by at most 47%. However, if the system is heavily loaded, this percentage will increase to above the 60% of the system without background load. This data shows that the distributed architecture is able to handle both query and non-query jobs by at most 32% expected delay in average response time.

5.7 Performance with Different Query Types

In this experiment we examine the performance of the distributed architecture with respect to left-deep and balanced queries. Initially, we expected the performance of

Figure 5.7: Impact of Background Load on Performance

left-deep queries to be worse than balance queries, since it does not exploit parallelism in general. But Figure 5.8 shows that left-deep queries perform better than balance queries. One explanation for this behavior is that changing the execution sequence of join operations affect the response time of a query. This is because the size of the intermediate results decreases as the execution of the query progresses. To demonstrate this behavior assume that there are four relations A, B, C , and D involved in a query with 6, 7, 6, and 6 tuples respectively. These relations are shown in Table 5.3. Also, assume that the cost of a join operation between two relations is a product of the number of tuples in each relation. Table 5.4 shows the result of two types of queries with the same data. Note that the cost of a balanced query is more than the cost of a left-deep query since the execution sequences are different.

Figure 5.8: System Performance with Two Types of Queries

Furthermore, the result indicates that with selecting the growing-sized² relations, the left-deep query performs better than the balanced query. Similarly, the left-deep query performs better than the balanced query when the shrinking-size relations are used.

²Based on the selectivity factor among two relations, the intermediate result of a join operation has more tuples than both relations.

Four Relations							
A		B		C		D	
Key	Value	Key	Value	Key	Value	Key	Value
1	a	1	b	1	c	1	d
2	a	2	b	1	c	2	d
2	a	2	b	2	c	2	d
4	a	3	b	2	c	3	d
5	a	4	b	3	c	3	d
7	a	7	b	3	c	3	d
		7	b				

Table 5.3: Four Relations in Balance and Left-deep Query

Balance Query Result

$A \bowtie B$	
Key	Value
1	a b
2	a b
2	a b
2	a b
2	a b
4	a b
7	a b
7	a b

$$Cost = 6 \times 7 = 42$$

$C \bowtie D$	
Key	Value
1	c d
1	c d
2	c d
2	c d
2	c d
2	c d
2	c d
3	c d
3	c d
3	c d
3	c d
3	c d

$$Cost = 6 \times 6 = 36$$

$(A \bowtie B) \bowtie (C \bowtie D)$	
Key	Value
1	a b c d
1	a b c d
2	a b c d
2	a b c d
-	-
-	-
-	-
2	a b c d
2	a b c d

$$Cost = 8 \times 12 = 96$$

Total Cost For Balance Query = 174

Left-deep Query Result

$A \bowtie B$	
Key	Value
1	a b
2	a b
2	a b
2	a b
2	a b
4	a b
7	a b
7	a b

$$Cost = 6 \times 7 = 42$$

$(A \bowtie B) \bowtie C$	
Key	Value
1	a b c
1	a b c
2	a b c
2	a b c
2	a b c
2	a b c
2	a b c
2	a b c
2	a b c
2	a b c
2	a b c

$$Cost = 8 \times 6 = 48$$

$((A \bowtie B) \bowtie C) \bowtie D$	
Key	Value
1	a b c d
1	a b c d
2	a b c d
2	a b c d
2	a b c d
-	-
-	-
-	-
2	a b c d
2	a b c d

$$Cost = 10 \times 6 = 60$$

Total Cost For Left-deep Query = 150

Table 5.4: Comparison Between Balance and Left-deep Query

Chapter 6

Conclusion

6.1 Summary

The main objective of this thesis was to design, implement, and study the performance of parallel query processing in a distributed architecture, specifically on a network of workstations. We have discussed several parallel database systems with different partitioning techniques, parallel processing strategies, load distribution, and loop scheduling policies. The advantages and disadvantages of these systems have been discussed and a comparison of commercially available ones has also been presented.

In Chapter 2, we reviewed a summary of the background work in database management systems and the progress of relational database systems along with the SQL language. A discussion of the partitioning techniques in parallel database systems has also been presented. This chapter also highlighted the differences between parallel database systems and NOW-based system for database query processing and discussed the issues encountered in a NOW-based environment. Furthermore, different policies in load scheduling and load distribution along with load sharing policies have been presented. Finally, query processing in a NOW-based environment using different parallel techniques has been discussed.

Chapter 3, along with the results in Chapter 5, are the main contributions of the thesis. This chapter briefly compared the three different architectures used

for parallel query processing in a NOW-based environment. Then, we presented a detailed description of our distributed architecture, its various components and processes, and provided methods and equations for the essential parts. A detailed description of the simulation model, PVM model, and NOW-based environment used in this study is also provided in Chapter 4.

Chapter 5 presented the results obtained from the simulation experiments. The preliminary experiments have shown that the specification of some parameters such as threshold values, probe limit, probe message reinitiation time, etc, used in distributed architecture perform better for certain system states. The results of the experiments have shown that the distributed architecture exhibits a reasonably good performance both in heavily and lightly loaded systems. It has also been observed that the small fragmentation size results in better performance, as the query can take advantage of load sharing, but finer-grained fragmentation leads to deterioration of performance, due to increased overhead and communication time. It has also been observed that background load has significant effect on the response time of the system as the mean inter-arrival time of the query increases. The mean response time of the query would drop by average 32% when the background load took between 40% to 50% of the CPU time. However, it has been shown that the distributed architecture is able to handle both query and non-query jobs. Finally, our study has shown that the distributed architecture successfully achieved both significant scaleups and speedups.

6.2 Contribution of the Thesis

Recently, researchers in the database community have been paying attention to the Network of Workstations (NOWs), as an alternative architecture to parallel database systems, for parallel query processing. Several research studies have shown that the distributed architecture is more robust, scalable, and reliable among centralized and hierarchical architectures. This thesis, using distributed architecture, is one of the few studies that provides the design, implementation, and performance evaluation

for parallel query processing. The main focus on this architecture is on three performance issues: load balancing, load sharing, and node allocation. Previous work on these issues was mainly focused on the scientific area, while we are looking at parallel query processing.

6.3 Future Work

Due to the large number of parameters, the effect of only a selected parameters was considered in the simulation experiments. However, this work can be expanded to explore other aspects of the distributed architecture. One interesting scenario to study is the effect of the execution sequence of the query on performance. The execution sequence can be maintained as part of the workload generator and use one of the known algorithms, such as Greedy, or Huffman algorithm.

Another interesting problem to study is the performance of the distributed architecture with receiver-initiated and adaptive policies. The sender-initiated policy used in this thesis provides superior performance only when the load of the system is relatively light. As the load of a system is not always light, the performance of the other two policies on heavily loaded systems can be studied.

The load sharing part of this research gets the benefit of fixed-chunk scheduling policy. This technique does not fully use the resources of the system since the power of each workstation is different. Early in this thesis, we introduced several other policies, which can be used to study the performance of distributed architecture.

In this architecture the intermediate result is kept on disk for each partial operation. This technique mainly introduces two major issues, which are the disk access time and pipeline mechanism. The pipeline mechanism provides performance improvements in query response time. In this technique, as soon as the partial intermediate result of the join operation is written into memory or disk, the upper level of the join operation can be started. If this mechanism is added to distributed architecture, it will reduce the waiting time of the join operations at upper levels

and the disk access time, which will result in an improvement in the performance of the system.

The last important issue is fault tolerance. We assume that there is no failure in distributed architecture. But, in a real environment, node failure is fairly common. The performance of the system can also be studied in the context of fault tolerance and job migration.

References

- [1] Ayachi S., Dandamudi S.P., "Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems," *IEEE International Symposium on Parallel and Distributed Processing (New Orleans)*, 1996, pp. 1-17.
- [2] Boral H., Alexander W., Clay L., "Prototyping Bubba, A Highly Parallel Database System," *IEEE Transaction on Knowledge and Data Engineering*, 1990, Vol. 2, No. 1, pp. 4-24.
- [3] Chen M.S., Yu P.S., Wu K.L., "Scheduling and Processor Allocation for Parallel Execution of Multi-join Queries," *IEEE Transactions on Parallel and Distributed Systems*, 1992, pp. 58-67.
- [4] Clark D., Casas J., Otto S., Prouty R., Walpole J., "Scheduling of Parallel Jobs on Dynamic, Heterogeneous Networks," 1995, <http://www.cse.ogi.edu/DISC/projects/cpe>, pp. 1-12.
- [5] Dandamudi S.P., "Using Network of Workstations for Database Query Operations," *Proceedings of International Conference on Computers and Their Applications (Tempe, Arizona)*, 1997, pp. 100-105.
- [6] Dandamudi S.P., Jain G., "Architecture for Parallel Query Processing on Networks of Workstations," *Proceedings of International Conference on Parallel and Distributed Computing Systems (New Orleans)*, 1997, pp. 444-451.
- [7] Dandamudi S.P., "Reducing Run Queue Contention in Shared Memory Multiprocessors," *IEEE Computer*, 1997, Vol. 30, No. 3, pp. 82-89.

- [8] Dandamudi S.P., Cheng S.P., "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor System," *IEEE Transaction on Parallel and Distributed Systems*, 1995, Vol. 6, No. 1, pp. 1-16.
- [9] Dandamudi S.P., Lo M., "Performance of Hierarchical Load Sharing in Heterogeneous Distributed Systems," *Proceedings of International Conference on Parallel and Distributed Computing Systems (Dijon, France)*, 1996, pp. 370-377.
- [10] Dandamudi S.P., Thyagaraj T.K., "A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems," *Proceedings of the 4th International Conference on High Performance Computing (HiPC'97)*, 1997, pp. 218-223.
- [11] Dandamudi S.P., Lo M., "A Hierarchical Load Sharing Policy for Distributed Systems," *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication System (Haifa, Israel)*, 1997, pp. 3-10.
- [12] Dandamudi S.P., "Sensitivity Evaluation of Dynamic Load Sharing in Distributed System," *IEEE Concurrency*, (to appear).
- [13] Dandamudi S.P., "The Effect of Scheduling Discipline on Dynamic Load Sharing in Heterogeneous Distributed Systems," *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication System (Haifa, Israel)*, 1997, pp. 17-24.
- [14] Dandamudi S.P., Hadavi H., "Performance Impact of I/O on Sender-Initiated and Receiver-Initiated Load Sharing Policies in Distributed Systems," *Proceedings International Conference on Parallel and Distributed Computing Systems (Dijon, France)*, 1996, pp. 507-514.
- [15] Dandamudi S.P., Jain G., "Exploiting Partitioned Parallelism for Parallelizing Database Queries," *Unpublished manuscript*, 1998, School of Computer Science, Carleton University.

- [16] Desai, B.C., *An Introduction to Database Systems*, West Publishing Company, 1990.
- [17] DeWitt D., Ghandeharizadeh S., Schneider D., Bricker A., Hsiao H., Rasmussen R., "The Gamma Database Machine Project," *IEEE Transaction on Knowledge and Data Engineering*, 1990, Vol. 2, No. 1, pp. 44-62.
- [18] Dewitt D., Gary J., "Parallel Database System: The Future of High Performance Database Systems," *Communications of The ACM*, 1992, Vol. 35, No. 6, pp. 85-98.
- [19] Eager D.L., Lazowska E.D., Zahorjan J., "Adaptive Load Sharing in Homogeneous Distributed System," *IEEE Transaction Software Engineering*, 1986, Vol. SE-12, No. 5, pp. 662-675.
- [20] Eager D.L., Lazowska E.D., Zahorjan J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation*, 1986, Vol. 6 No. 1, pp. 53-68.
- [21] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V., *PVM: Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.
- [22] Geist G., Kohl J., Papadopoulos P., "PVM and MPI: A Comparison of Features," *Calculateurs Paralleles*, 1996, Vol. 8 No. 2, pp. 1-16.
- [23] Harris E.P., Ramamohanarao K., "Join Algorithm Costs Revisited," *The VLDB Journal*, 1996, Vol. 5, No. 1, pp. 64-84.
- [24] Hass L., Freytag J.C., Lohman G., Pirahesh H., "Extensible Query Processing in Startburst," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1989. Vol. 18, No. 2, pp. 377-388.
- [25] Hong W., "Exploiting Inter-Operation Parallelism in XPRS," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992, Vol. 21, No. 2, pp. 19-28.

- [26] Hummel S.F., Lawrence E., Flynn E., "Factoring: A Method for Scheduling Parallel Loops," *Communication of the ACM*, 1992, Vol. 35, No. 8, pp. 91-101.
- [27] Knig-Ries B., Lockemann P., "Research in Databases and Data-Intensive Applications," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1997, Vol. 26, No. 3, pp. 67-72.
- [28] Lilja D.J., "Exploiting the Parallelism Available in Loops," *IEEE Computer*, 1994, Vol. 27, No. 2, pp. 13-26.
- [29] Liu C., Yu C., "Performance Issues in Distributed Query Processing," *IEEE Transactions on Parallel and Distributed Systems*, 1993, Vol. 4, No. 8, pp. 889-905.
- [30] Mishra P., Eich M., "Join Processing in Relational Databases," *ACM Computing Surveys*, 1992, Vol. 24, No. 1, pp. 63-113.
- [31] Mutka M., Livny M., "The Available Capacity of a Privately Owned Workstation Environment," *Performance Evaluation*, 1991, Vol. 12, No. 4, pp. 269-284.
- [32] Ozsu M.T., Valduriez P., "Distributed and Parallel Database Systems Technology and Current State-of-the-Art," *ACM Computing Surveys*, 1996, Vol. 28, No. 1, pp. 125-128.
- [33] Piotrowski A., Dandamudi S.P., "A Comparative Study of Load Sharing on Networks of Workstations," *Proceedings International Conference on Parallel and Distributed Computing Systems*, 1997, pp. 1-8.
- [34] Pirahesh H., Mohan C., Cheng J., Selinger P., "Parallelism in Relational Database Systems: Architectural Issues and Design Approaches," *Second International Symposium Database in Parallel and Distributed Systems*, 1990, RJ7724(71589), IBM Almaden Research Center, pp. 4-29.

- [35] Polychronopoulos C., Kuck D., "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, 1987, Vol. 36, No. 12, pp. 1425-1439.
- [36] Schneider D.A., DeWitt D.J., "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," *Proceedings of the ACM SIGMOD International Conference on Management of data*, 1989, Vol. 18, No. 2, pp. 110-121.
- [37] Shivaratri N.G., Krueger P., Singhal M., "Load Distributing for Locally Distributed Systems," *IEEE Computer*, 1992, Vol. 25, No. 12, pp. 34-44.
- [38] Shivaratri N.G., Krueger P., "Two Adaptive Location Policies for Global Scheduling Algorithms," *IEEE International Conference Distributed Computing Systems*, 1990, pp. 502-509.
- [39] Tzen T.H., Ni L.M., "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," *IEEE Transactions on Parallel and Distributed Systems*, 1993, Vol. 4, No. 1, pp. 87-98.
- [40] Valduriez P., "Parallel Database Systems: open problem and new issues." *International Journal on Distributed and Parallel Databases*, 1993, Vol. 1, No. 2, pp. 1-27.
- [41] Van den Berg C.A., Kersten M., Shair-Ali S., "Dynamic Parallel Query Processing," *Centre for Mathematics and Computer Science*, 1991, Report CS-R9112, pp. 1-14.

Appendix A

Procedures Used in Distributed Architecture

A.1 Procedure 1 *Scheduler*

- 1 *Scheduler* receives a job
- 2 if the job is job request from *Load-Coordinator* then
- 3   send one query to *Load-Coordinator* if there is any
- 4 if the job is load request from *Load-Coordinator* or from remote node then
- 5   send the current load of workstation
- 6 if the job is query then
- 7   if the newly arrived job is remote query then
- 8   increment *number of transfer*
- 9   endif
- 10   push the query into *Local-query Queue*
- 11   ask *Transfer Policy* to schedule the job
- 12 endif
- 13 if the job is non-query then
- 14   push the non-query job into *Non-query Queue*
- 15   Schedule one non-query if there is any
- 16 endif

A.2 Procedure 2
Transfer Policy

```

1  Transfer Policy check the load  $l$ 
2  if  $l > T_{load}$  then
3      Transfer Policy decides to transfer the newly arrived query.
4      ask Location Policy to find one available workstation.
5      if Location Policy fails then
6          try after a prespecified time
7      else
8          peek query from Local-query Queue which has the number of transfer  $< Transfer\ limit$ 
9          if the query is found then
10             transfer the query to under-loaded node
11         endif
12     endif
13 endif

```

A.3 Procedure 3
Location Policy

```

1   $P_i \leftarrow 0$ 
2  while  $p_i < p_l$  do
3       $W_i$  selects node  $j$  randomly
4      if  $l_j < T_{uload}$  then
5           $p_i \leftarrow 0$ 
6          return node  $j$ 
7      else
8           $p_i \leftarrow p_i + 1$ 
9      endif
10 enddo
11 return fail

```

A.4 Procedure 4 *Load Coordinator*

```

1  while there is any job in Local-query Queue do
2     $Q_1 \leftarrow \text{pop}(\text{Local-query Queue})$ 
1  "The status of workstation is under-loaded( $l_{W_1} < T_{load}$ )"
2  if  $W_1$  is under-loaded then
3    Compute  $M_w$  for  $Q_1$  "  $M_w$  is the maximum available nodes"
4    if  $M_w = 1$  then
5      Load-Coordinator reads corresponding data of the job
6       $W_1$  processes the job locally without load sharing
7      push the result of  $Q_1$  into Pending Queue
8      Send the finish message to sender of  $Q_1$ 
9      get the next job
10   else
11     Send partial-operation to its worker
12     for each join operation in  $Q_1$  do
13       Load-Coordinator reads the corresponding data
14       while the operation is not finished do
15         Send partial-operation to the worker of node  $j$ 
16         if Load-Coordinator receives finish message from node  $j$  then
17           Send next partial-operation to the worker of node  $j$ 
18         endif
19       enddo
20     enddo
21   endif
22 endif
23 get the next job

1  "The status of workstation is over-loaded( $l_{W_1} > T_{load}$ )"
2  if  $W_1$  is over-loaded then
3    if the number of transfer < Transfer limit then
4      Load-Coordinator asks Location Policy to find an under-loaded node  $j$ 
5      if Location Policy fails then

```

```

6      Load-Coordinator get one operation if there is any and process it locally
7      push the result into Pending Queue of sender node
8      send finish message to sender node
9      goto line 5
10     else
11         send the rest of job to node  $j$ 
12         get the next job
13     endif
14 else
15     Compute  $M_w$  for  $Q_1$  " $M_w$  is the maximum available nodes"
16     Load-Coordinator asks Location Policy to find  $M_w$  under-loaded nodes
17     if Location Policy fails then
18         Load-Coordinator get one operation if there is any and process it locally
19         push the result into Pending Queue of sender node
20         send finish message to sender node
21         goto line 16
22     else
23         for each under-loaded node  $j$  do
24             Send one join operation to node  $j$ , if there is any
25         enddo
26         if there is any operation left in  $Q_1$  then
27             goto line 16
28         endif
29     endif
30 endif
31 endif
32 get the next job

1     "The status of workstation is OK( $T_{uload} \leq l_{W_1} \leq T_{oload}$ )"
2     if  $W_1$  is OK then
3         Compute  $M_w$  for  $Q_1$  " $M_w$  is the maximum available nodes"
4         Load-Coordinator asks Location Policy to find  $M_w$  under-loaded nodes
5         if Location Policy fails then

```

```

6      Load-Coordinator get one operation if there is any and process it locally
7      push the result into Pending Queue of sender node
8      send finish message to sender node
9      goto line 4
10     else
11         for each under-loaded node  $j$  do
12             Send one join operation to node  $j$ , if there is any
13         enddo
14         if there is any operation left in  $Q_1$  then
15             goto line 4
16         endif
17     endif
18 endif
19 get the next job

```

A.5 Procedure 5 *Result Matcher*

```

1  while there is any job in Pending Queue do
2       $P_i \leftarrow \text{pop}(\text{Pending Queue})$ 
3      if  $P_i$  is the last partial result then
4          merge all partial operation results in one file using their id
5          peek the parent of  $P_i$  from Pending Queue and set to  $Q_i$ 
6          join the merged file with  $Q_i$ 
7          update the flag of  $Q_i$ 
8          if  $Q_i$  is executable then
9              send  $Q_i$  to the Scheduler of sender node
10         else
11             push  $Q_i$  into Pending Queue
12         endif
13     else
14         push  $P_i$  into Pending Queue
15     endif
16 enddo

```

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved