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

Centralized Architecture for Query Processing on Network of Workstations

by

Sijun Zeng, B.S.

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

May 1998

© Copyright

1998, Sijun Zeng

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.

0-612-32390-0

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
**Centralized Architecture for Query Processing on Network of
Workstations**
submitted by
Sijun Zeng, B.S.
in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

May 7, 1998

Acknowledgements

I would like to thank my supervisor, Dr. Sivarama P. Dandamudi for his generosity, advice and guidance, which made all things possible for me.

I would also like to dedicate this thesis to my parents for their eternal love.

Abstract

Parallel processing is one of the techniques that could be used to improve computer system performance. In the area of transaction processing, increasing demand for high transaction processing rate coupled with large databases sizes has motivated the use of parallel processing techniques for transaction or query processing. Several parallel database systems have been developed for parallel query or transaction processing. However, due to the high cost of most commercially available parallel database systems, attention has been turned to its alternative, Network of Workstations (NOW). Exploiting the otherwise wasted CPU cycles and memory of NOW-based systems for query processing is a new area that is being studied currently. Query processing on NOWs benefits from three aspects: concurrent execution of multiple queries, exploiting intra-query and intra-operation parallelism. Three query processing architectures for NOW-based systems have been proposed. These are the centralized, distributed and hierarchical architectures. This thesis presents a detailed design and implementation of a three-level strategy for the centralized architecture for query processing on NOWs. A comprehensive performance evaluation is also presented. The experimental results presented in the thesis suggest that significant performance benefits can be obtained by using network of workstations for parallel query processing.

Contents

Acknowledgements	iii
Abstract	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Parallel Database System	3
1.2 Network of Workstations	6
1.3 Motivation	8
2 Background	11
2.1 DBMS Query Processing	11
2.1.1 Database Management System (DBMS)	11
2.1.2 Query Processing	12
2.2 Load Distribution	15
2.2.1 Load Distribution Algorithm	16
2.2.2 Sender-Initiated and Receiver-Initiated Policies	18
2.2.3 Load-sharing versus Load-balancing	20
2.3 Loop Scheduling	21

2.3.1	Fixed-Size Chunk Scheduling (FSCS)	23
2.3.2	Guided Self-Scheduling (GSS)	24
2.3.3	Factoring	24
2.3.4	Trapezoid Self Scheduling (TSS)	25
2.3.5	Affinity Scheduling (AS)	26
3	Parallel Query Processing on NOW	27
3.1	Parallelism in Network of Workstations	28
3.2	Query processing architectures on NOWs	29
3.2.1	Centralized architecture	30
3.2.2	Distributed architecture	30
3.2.3	Hierarchical architecture	32
3.3	Query processing in centralized architecture	34
3.3.1	Partition Size Calculation	37
3.3.2	Processor Partitioning	41
3.3.3	Operation Scheduling	44
4	Experimental Model and Implementation	48
4.1	Triton Parallel System	48
4.2	Database Model	51
4.3	Query Workload Model	53
4.4	Parallel Virtual Machine (PVM)	54
5	Performance Analyses	56
5.1	Some Preliminary Experiments	56
5.1.1	Node Computational Capacity	57
5.1.2	System Capacity	58
5.2	Sensitivity to System Inter-Arrival Time	59
5.3	Sensitivity to Fragmentation Size	59

5.4	System Speedup	62
5.5	System Scaleup	65
5.6	Sensitivity to Query Type	67
5.7	Sensitivity to Parameter f	68
6	Conclusion and Future Work	71
6.1	Summary	71
6.2	Contribution of Thesis	72
6.3	Future Research	72
	References	74

List of Tables

4.1	Nodes of Triton System	50
4.2	Selectivities among tables	52
5.1	Time (second) of executing 16 queries on each node	57
5.2	Processor Unit of each node used in experiments	58
5.3	Three independent experiment results on system capacity	58

List of Figures

1.1	A shared-memory multiprocessor system	2
1.2	A distributed-memory multicomputer system	3
1.3	Some parallel database system architecture	10
2.1	The three-schema architecture (from [33])	12
2.2	A high level query steps (from [33])	14
2.3	A distributed system without load distributing	17
2.4	Sender-initiated load sharing using a threshold location policy (from [29])	19
2.5	Receiver-initiated load sharing (from [29])	20
3.1	Centralized architecture (from [7])	31
3.2	Distributed architecture (from [7])	32
3.3	Hierarchical architecture	33
3.4	Data and Query Flow in the Centralized Architecture	35
3.5	Three-level Centralized Architecture	36
4.1	Triton Parallel System	49
4.2	Balanced and Linear query with 4 tables	53
5.1	Performance Under Different Inter-Arrival Time	60
5.2	Performance under different fragmentation size when system is lightly loaded	61

5.3	Performance under different fragmentation size when system is heavily loaded	62
5.4	System Speedup under light load	64
5.5	System Speedup under heavy load	65
5.6	System Scaleup	66
5.7	Performance with balanced and linear queries	67
5.8	Performance Under Different Parameter f	69
5.9	Execution and Waiting Time Under Different Parameter f	70

Chapter 1

Introduction

Two basic techniques exist for improving computer system performance – using faster components to reduce processor cycle time and exploiting parallelism for concurrent execution. As the components approach their limitations, parallel processing has been receiving considerable attention in recent years.

Parallel processing requires concurrent execution of many programs in the computer in contrast to conventional sequential processing. In parallel systems, several processors work together to reduce execution time of a task. Two approaches have been proposed to improve system performance of parallel systems: *time sharing* and *space sharing*. In *time sharing*, several applications temporally share the system [19, 23, 25]. In the other approach, sharing is done spatially. In *space sharing*, subsets of processors are assigned to various parallel applications, either in a dedicated modes (i.e. one subset per program) or in time sharing mode (i.e. several programs execute on the same partition).

Parallel processing systems can be broadly divided into two basic architectures: *shared-memory* and *distributed-memory*. In shared-memory systems (often referred to as multiprocessor), the system memory is shared by all processors. This shared

memory is the medium by which processors communicate (Figure 1.1). This kind of architecture does not scale due to the network bandwidth limitations. Some examples of multiprocessors are the BBN Butterfly, Sequent Balance, IBM RP3, the Stanford DASH, and the Illinois Cedar.

Figure 1.1: A shared-memory multiprocessor system

In distributed-memory systems (often referred as multicomputers), on the other hand, each processor has its own local memory (Figure 1.2). While each processor can access its own memory, accessing other processor's local memory is impossible. As a result, these systems communicate by means of message passing. The advantage of these systems is their scalability. Parallel machines from Intel and Ncube, and machines based on Transputers are all examples of commercially available distributed-memory systems.

Query or transaction processing in database systems is one of the areas that can be improved by applying parallel processing techniques. In recent years, the demand

Figure 1.2: A distributed-memory multicomputer system

for high transaction processing rates has continued to increase substantially. Some of the motivations [6, 10, 26] for parallel query processing are

- An increasing size of relational databases consisting of billions of records.
- Data intensive read only queries on these large databases.
- Support for high level query languages like SQL, allowing users to pose complex queries at their terminals.
- The need for supporting non standard database applications like VLSI CAD, Computer Aided Software Engineering (CASE), Multimedia, etc.

A direct consequence of the above is an ever increasing demand for performance. The traditional main-frame systems do not meet the demand. The current trend is to build parallel database systems in order to effectively support these applications.

1.1 Parallel Database System

A parallel database system can be built by connecting together several processors, memory units and disk devices through a global interconnection network. Depending

on how these components are interconnected they can be categorized into a variety of architectures. Three main categories [10] are

Shared-memory In this category, all the processors share a common memory but each processor has a direct access to its disk. All processors are connected to the shared memory through an interconnection network. IBM/370, Digital VAX and the Sequent Symmetry are commercially available system of this type

Shared-disk In this system, each processor has its local memory but all processors share all the disks. All processors are interconnected to all the disks through an interconnection network. The commercially available systems of these type are the IBM Sysplex and the original Digital VAX cluster.

Shared-nothing All processors in this type have individual memories and disks. All the processors are connected via a high speed interconnection network. Communication between the processors is via a message-passing model. Many commercial and research parallel systems like Teradata, Tandem, Bubba and Gamma are examples of this type of architecture.

Each of the above architecture consists of processors (p), memory units (m), disk units (d), local buses, and a global interconnection networks as shown in Figure 1.3. The shared-memory and shared-disk systems are simple models since all the memory and disks are connected to all processors in each system respectively. However, this sharing resource increases the risk of interference between different processors and increases latency when accessing shared information, which, in turn deteriorates the performance of the system and limits their scalability.

The shared-nothing architecture can be scaled extensively as none of the components are shared. The shared-nothing architecture also removes the need for a complex interconnecton network. In the shared-memory and shared-disk systems,

the interconnection network used has to be powerful because it has to transmit large quantities of raw data across different processors. The shared-nothing architecture has its own disadvantages. The presence of individual memory and disk increases the complexity of the system and can lead to potential load-balancing problems.

Due to its various advantages and relatively few disadvantages, the shared-nothing architecture has emerged as the most commonly used architecture in all the commercially available systems. The availability of cheap and off-the-shelf components have led to a sharp increase in the development and acceptance of shared-nothing parallel databases.

To effectively use parallel database systems, we must find a way to partition the database across the system resources such that parallel processing is facilitated. For example, in relational data model, large relations can be fragmented into a number of partitions and can be stored on the different disks. Partitioning or fragmentation should ensure that the database does not undergo any changes during fragmentations.

Partitioning allows the processors to scan and write data from all the disks in parallel, thus minimizing the execution time of the transaction. The main advantage of partitioning is that each partition of data can be read and written using normal I/O hardware. The system can also be scaled indefinitely without any requirement of complex and customized hardware.

There are several ways to partition data. Some of the most important include the following:

Round Robin. Tuples are distributed among the various fragments of the relation in a round-robin fashion. The round-robin strategy works well when majority of the queries involve the whole relation to be accessed sequentially. But when

tuples have a common value for a particular attribute or a set of attributes have to be accessed in a query (associative access), bottleneck may be created.

Hash Partitioning. The tuples are distributed among the different fragments using some hash function mapped on an attribute or a set of attributes. This strategy is suited for sequential as well as associative access as the tuples will be placed according to a specific value of an attribute or attributes of the relation. The advantage of this strategy is, for associative access, scan will be directed to only one disk instead of all them, thus limiting the overhead of starting queries at multiple disks.

Range Partitioning. Tuples with similar attribute values, which are frequently accessed together, are placed together. This technique is also called *clustering*. This strategy is also suited for sequential as well as associative access. But clustering can lead to a few problems during data access, such as *data skew* and *execution skew*.

Partitioning also leads to an improvement in the response time of the system. In case of sequential access, increase in the number of partitions leads to a decrease in the response time of the system. In the case of associative research, as the number of partitions increases, the number of tuples to be accessed at each partition decreases.

1.2 Network of Workstations

Parallel processing techniques have greatly improved the performance of computer systems. However, the commercial parallel systems are very expensive, often costing millions of dollars depending on system size. Most of the small organizations may not be able to afford such expensive systems.

In the parallel processing area, there is a trend to use network of workstations (NOW) as a virtual parallel machine. A NOW-based system consists of a lot of heterogeneous computing nodes connected by some kind of communication network.

These computing nodes can be from low end PCs to high speed workstations.

Using a group of workstations connected via a LAN provides better cost/performance, or may be the only way to achieve high performance within budget constraints. Another factor in favor of NOW-based computing is the availability of many lightly loaded workstations. Several studies have shown that a large fraction (up to 80% depending on the time of day [25]) of the processing power available on a network of workstations is wasted by idling workstations. Substantial performance gains can be obtained by exploiting these idle processing cycles [11, 24, 30].

NOW has been recognized as the primary computing infrastructure for science and engineering [1]. Several commercial and noncommercial tools have been developed to support network computing. Widely used tools include the Parallel Virtual Machine (PVM) software, the P4 system, Express, and the Message Passing Interface (MPI) [2, 4, 15, 30]. The availability of these tools to configure a parallel virtual machine over a network of workstations has led several researchers to study performance issues in this area.

Due to its characteristics, there are several performance issues associated with using Network of Workstations. These performance issues include:

- Load Balancing

Load balancing on a NOW-based system causes several problems mainly due to workstation heterogeneity and dynamic load variations as each workstation can receive jobs locally. In order to achieve load balancing on a NOW-based system, dynamic load balancing schemes have to be used.

- Job Migration

In a NOW-based system, a user might want to take his/her workstation out of the pool of workstations used for database query processing. This could be the result of high load on the workstation or for other reasons. In such instances,

there are two options: either kill the task and reinitiate it on another workstation or migrate the task to another workstation. The first option involves duplicating the work done on the task being lost whereas task migration is expensive but preserves the work.

- **Fault-Tolerance**

Fault-tolerance is another key issue with NOW-based systems. In these systems, failures can occur not only due to system faults but also because of workstation owners' actions. For example, if workstation owners do not like their machines to be part of a NOW, they can turn the machine off. Thus, fault-tolerance is very important in NOW-based systems.

- **Communication Network Latency**

Communication networks used in NOW-based systems are slow compared to the interconnection networks used in parallel database systems. Improved performance of communication subsystems is due to better networks, low-overhead protocols and efficient switching methods. For parallel processing applications, slow communication network can seriously degrade performance of applications. We should also note that LANs are improving substantially. As a result, the parallel algorithms devised for parallel database systems will have to be reevaluated for the NOW-based systems.

1.3 Motivation

Commercial parallel database systems have been extensively used for improving database system performance by processing queries in parallel. However, due to their high cost, not many organizations are able to afford parallel systems. On the other hand, the immense and untapped computing power of network of workstations for parallel processing has not received attention in the database community. A new concept, that of using network of workstations (NOW) for parallel query processing has been proposed [6] for improving the performance of database system.

Dandamudi [6] has demonstrated the performance benefits of parallelizing a four way join query using a set of workstations.

In this thesis, we describe three query processing architectures for NOW-based systems. Particularly, we focus our research on the query processing in centralized architecture. Our objective is to design our parallel query processing strategy and analyze its performance.

The remainder of the thesis is organized as follows:

Chapter 2 reviews background on DBMS query processing, load distribution and loop scheduling. In Chapter 3, we briefly describe three query processing architectures for NOW-based systems: centralized, distributed and hierarchical. We present our three-level strategy for the centralized architecture in detail. Chapter 4 describes the database model, system model, workload model and the PVM software that are used for the implementation and experimentation. Chapter 5 presents the experimental results and discusses performance of the centralized architecture for query processing on a NOW-based system. Conclusions are given in chapter 6. Chapter 6 also presents the contributions of the thesis and the directions for future research.

Figure 1.3: Some parallel database system architecture

Chapter 2

Background

Query processing is one of the most important operations in a database management system (DBMS). With the increasing demands for data intensive queries from large databases, performance issue of database query has been received attention in recent years. This chapter presents an overview of DBMS query processing, load distribution, and loop scheduling.

2.1 DBMS Query Processing

2.1.1 Database Management System (DBMS)

Databases have been the primary source of information repository for a very long time. A database is actually a set of data item stored in a manageable way. A database management system (DBMS) is a software package that facilitates the database operation and maintainence. Figure 2.1 shows a three-schema architecture of DBMS.

Internal Level includes an **internal schema**, which describes the physical storage structure of the database.

Conceptual Level has a **conceptual schema** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities,

Figure 2.1: The three-schema architecture (from [33])

data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

External or view level includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group.

Most DBMSs do not separate the three levels completely, but several of them support the three-schema architecture to some extent.

2.1.2 Query Processing

There are several types of operations can be applied to a DBMS. The support for querying stored information is known as *database query processing*. In the world of

relational database, which is basically a set of relations and each relation is a set of homogeneous tuples, relations are created, updated and queried using a database language such as SQL. Querying a database is the most important part of any database management system (DBMS). A query comprises of user conditions on the database relations, and the tuples satisfying the condition(s) are retrieved from the corresponding relations in the database.

A high-level query, such as expressed by SQL, must first be scanned, parsed, and validated. An internal representation of a query is then created, usually a data structure, called *query tree* or *query graph*. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as *query optimization*.

Figure 2.2 shows the different steps of processing a high-level query. The *query optimizer* module has the task of producing an execution plan, and the *code generator* generates the code to execute that plan. The *runtime database processor* has the task of running the query code, whether compiled or interpreted, to produce the query result.

The requirement of querying the database is not just to retrieve all tuples satisfying a user query but in a reasonable amount of time. At the mean time, with computers approaching high execution speeds and several users querying the database simultaneously, maintaining a reasonable level of performance is essential. Some of the important terminologies are given below:

- Response Time (RT)

This is the time difference between when a job enters the system and the time the job is completed. This is one of the metric that is most often used in performance evaluation.

- Waiting Time (WT)

Figure 2.2: A high level query steps (from [33])

This is the time period from when a job is submitted to the system to the time it is processed for execution.

- **Execution Time (ET):**

This refers to the time difference between when the job is started to execute and when it is finished.

- **System Throughput (ST):**

This refers to the average number of jobs completed in a unit time.

The objective to maintain a reasonable level of performance implies a reduction in the mean response time, which can be also defined as sum of the waiting time (WT)

and the actual execution time (ET) of a query, in most cases. This is essentially reduced by:

- Reducing the average waiting time (WT) of a query.
- Reducing execution time (ET) of query.
- Increasing overall throughput.

Parallelism in database queries provide immense potential for speeding up complex database queries. The idea behind parallel query processing/optimization is to process multiple queries using a set of available processors efficiently. Parallelism can be exploited at the database query level and/or database operator level. In the former the system allocates one processor per query that arrives in the system, ignoring the fact that some queries are less complex than the other. In the latter, the system takes into account the complexity of the query and may allocate a different number of processors to different queries. As a simple example, consider two database queries one of which consists of only one join operation and the other consists of several join operations. It is necessary to allocate larger fraction of the available processors to the second query if a reasonable response time is required. The database system also exploits the inherent parallelism available in database queries to divide a query into several independent subqueries and schedule each subquery for parallel execution. The partial results from each of the subquery are then combined to generate the final result.

2.2 Load Distribution

A locally distributed computing system consists a collection of autonomous computers linked by a local area communication network. Users submit tasks at their host computers for processing. The random arrival of tasks in such an environment can cause some computers to be heavily loaded while other computers are idle or only lightly loaded as being shown in Figure 2.3. This situation is referred to as *load*

imbalance which degrades overall system performance as computing resources are not fully utilized. Load distribution improves the performance and reduces the level of load imbalance by transparently transferring jobs from heavily loaded computers, where service is poor, to lightly loaded or idle computers, while the tasks can take advantage of computing capacity that would otherwise go unused.

If workload at some computers are typically heavier than at others, or if some processors execute tasks more slowly than others, the situation shown in Figure 2.3 is likely to occur often. The usefulness of load distributing is not so obvious in systems in which all processors are equally powerful and have equally heavy workloads over the long term. However, studies show that, even in a homogeneous distributed system, system performance can potentially be improved by appropriate transfers of workload from heavily loaded computers (*senders*) to idle or lightly loaded computers (*receivers*). This is due to the statistical fluctuations in the arrival of tasks and task-service-time requirements, which causes at least one computer likely to be idle while other computers are heavily loaded.

2.2.1 Load Distribution Algorithm

Load distribution algorithm can be broadly characterized [29] as *dynamic*, *static* and *adaptive*. Dynamic algorithm makes use of system-state to make load distribution decisions, while static algorithm uses a priori knowledge of the system. Adaptive algorithms adapt their activities by dynamically changing their parameters or even their policies to suit the changing system state.

Basically, a load distribution algorithm has four components: a transfer policy, a selection policy, a location policy, and an information policy.

Transfer Policy: Determines whether a node is in a suitable state to join in a task transfer, either as a sender or a receiver.

Figure 2.3: A distributed system without load distributing

- **Threshold Policies:** If the load exceeds a threshold T_1 at a site, it becomes a sender. If the load falls below T_2 , it becomes a receiver of remote tasks. Depending on the algorithm, T_1 and T_2 may have the same value.
- **Relative Transfer Policy:** Load of a node in relation to loads at other nodes in the system is taken into account to decide if it is a suitable sender or receiver

Selection Policy: Selects a task for transfer. Several factors are considered in selecting a task.

- The overhead incurred by the transfer should be minimal
- The selected task should be long lived so that it is worthwhile to incur the transfer overhead
- The number of location dependent system calls made by the selected task should be minimal

Location Policy: Finds a suitable site to transfer to.

- Polling: other nodes are polled to find out if they are suitable for load sharing.
- Query Broadcast: broadcasting a query seeking for any nodes available for sharing

Information Policy: Decides when, from where and what information about other nodes to collect. Three types information policies are:

- Demand-Driven: information is collected only when a node becomes a sender or receiver.
- Periodic: nodes exchange information periodically.
- State-Change-Driven: nodes disseminate information only when their state change.

2.2.2 Sender-Initiated and Receiver-Initiated Policies

Load distributing algorithms are also categorized as sender-initiated and receiver-initiated. In Sender-initiated algorithms, load-distributing activity is initiated by an overloaded node (sender) trying to send a task to an underloaded node (receiver). Three simple, yet effective algorithms were studied by Eager, Lazowska, and Zahorjan [29]. Each of the algorithms uses the same threshold transfer policy, based on the CPU queue length. All three algorithms have the same selection policy, considering only newly arrived tasks for transfer. Different location policies in the three algorithms are random, threshold (see Figure 2.4) and shortest. Information policy is demand-driven.

In receiver-initiated algorithms, load distributing activity is initiated from an underloaded node (receiver), which tries to get a task from an overloaded node (sender). An algorithm studied by Livy and Melman [22], and Eager, Lazowska, and Zahorjan [13] is shown in Figure 2.5.

Figure 2.4: Sender-initiated load sharing using a threshold location policy (from [29])

The algorithm's threshold transfer policy bases its decision on the CPU queue length. The policy is triggered when a task departs. If the local queue length falls below the threshold T , then the node is identified as a receiver for obtaining a task from a node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold T . In the selection policy, the algorithm considers all tasks for load distributing, and can use any of the approaches discussed. The location policy selects a node at random and polls it to determine whether transferring a task would place its queue length below the threshold level. If not, then the polled node transfers a task. Otherwise, another node is selected at random, and the procedure is repeated until either a node that can transfer task is found or a static poll limit number of tries has failed to find a sender. The information policy is demand-driven, since polling starts only after a node becomes receiver.

Figure 2.5: Receiver-initiated load sharing (from [29])

2.2.3 Load-sharing versus Load-balancing

Dynamic load-distributing algorithms can be further classified as being load-sharing or load-balancing algorithms. The goal of a load-sharing algorithm is to maximize the rate at which a distributed system performs work when work is available. The load-sharing algorithms strive to avoid unshared states, states in which some computer lies idle while tasks contend for service at some other computers.

If task transfers are instantaneous, unshared states could be avoided by transferring tasks only to idle computers. Because of the time required to collect and package a task's state and because of communication delays transfers are not instantaneous. The lengthy unshared states that would otherwise result from these delays can be partially avoided through anticipatory transfers, which are transfers from overloaded to lightly loaded computers, under the assumption that lightly loaded computers are likely to become idle soon.

Load-balancing algorithms also strive to avoid unshared states, but go a step beyond load sharing by attempting to equalize the load at all computers. Krueger and Livny [17] have shown that load balancing can potentially reduce the mean and standard deviation of task response times, relative to load-sharing algorithms. Load balancing requires a higher transfer rate than load sharing, however, the higher overhead incurred may outweigh this potential performance improvement.

2.3 Loop Scheduling

The problem of mapping or assigning different tasks of the same program to different processors is known as *scheduling*. To take advantage of the capability of parallel processing, jobs must have sufficient parallelism to be exploited. Data dependencies define a partial order or precedence relation on the statements of a program. There are three major types of data dependences, *flow*, *anti* and *output*. Loops without dependencies among their iterations are a rich source of parallelism in scientific codes. Restructuring compilers for sequential programs have been particularly successful in determining when loop iterations are independent and can be executed in parallel. Because of the prevalence of parallel loops, optimal parallel-loop scheduling has received considerable attention in both academic and industrial communities.

The fundamental trade-off in scheduling parallel-loop iterations is that of maintaining balanced processor workload vs. minimizing scheduling overhead. Static parallel loop scheduling, or pre-scheduling, assigns iterations to specific processors at compile time or load time. Each processor determines which iterations it will execute based on its processors number. This scheduling strategy evenly distributes iterations across processors to attempt to balance the computational load. Dynamic loop scheduling, also called self-scheduling, can be used to move scheduling decisions from compile time to runtime by making each processor responsible for allocating its own work. A shared iteration counter variable indicates which iteration number

should be executed by the next available processor.

A variety of parallel computer architectures and compilation techniques have been proposed to exploit this parallelism at different granularities. For example, pipelined processors and multiple instruction-issuing processors, such as superscalar and very-long instruction word (VLIW) machines, exploit fine-grained parallelism at the instruction set level. These processors use both runtime and compile-time techniques, such as dynamic lookahead and software pipelining, to find independent operations to execute concurrently. In contrast, shared-memory multiprocessors exploit coarse-grained parallelism by distributing entire loop iterations to different processors.

Fine-grained parallel architectures exploit loop parallelism at the instruction set level by issuing several instructions or operations in a single cycle. At runtime, dependencies between operations must be checked either statically by the compiler or dynamically by the hardware to ensure that only independent operations are issue simultaneously. To extract maximum parallelism, the dependence-checking techniques must be capable of looking beyond basic block boundaries to find independence operations both within the body of a single iteration and from several different iterations. Techniques such as branch prediction and static loop unrolling can be applied.

Coarse-grained parallel architectures exploits parallelism by scheduling entire iterations on separate processors. On shared-memory multiprocessors, for example, the independent tasks to be scheduled are the individual instantiations of the iterations, each with a unique value of the loop index. The loop scheduling strategy then determines which tasks will be executed on which processors at what time.

Static parallel loop scheduling, or *prescheduling*, assigns iterations to specific processors at compile time or load time. Each processor determines which

iterations it will execute based on its processor number. This scheduling strategy evenly distributes iterations across processors to attempt to balance the computational load. However, since the compiler could not accurately predict all execution times, the computational load could not be perfectly balanced to minimize total execution time.

Dynamic loop scheduling, also called *self-scheduling*, can be used to move scheduling decisions from compile time to runtime by making each processor responsible for allocating its own work. Idle processors assign iterations to themselves at runtime by accessing a shared iteration-counter variable. This shared variable indicates which iteration number should be executed by the next available processor. The number of access can be reduced by allocating a chunk of iterations for each access. Next we introduce some of the scheduling schemes [16, 20, 27, 31].

2.3.1 Fixed-Size Chunk Scheduling (FSCS)

As the name implies, this scheduling method assigns fixed-size chunks to processors. Kruskal and Weiss analyzed the scheduling of iterations in fixed-size chunks [19]. They argue that an acceptable fixed chunk-size K can be found, and give an approximation and an upper bound for the expected time for P processors to execute N iterations in chunks of size K . An optimal fixed-chunk size K can be found by differentiating these formulae with respect to K . Unfortunately, their results are only applicable when both P and K are large.

Kruskal and Weiss speculate that an optimal chunking scheme would use chunks that are of decreasing (rather than fixed) size. The initial larger chunks incur very little relative overhead, whereas the later smaller chunks smooth over the unevenness of the earlier chunks. If the initial chunks are not too large and there are a sufficient number of smaller chunks, then the final unevenness of processor finishing time have

the same magnitude as the execution times of the smaller chunks. Several decreasing method have been proposed in the literature [14, 27, 31].

2.3.2 Guided Self-Scheduling (GSS)

One scheduling strategy that performs decreasing-chunk size allocation is *guided self-scheduling* [3]. In GSS, the size C_i of the chunk scheduled on the next idle processor is R_i/P , where R_i is the number of iterations remaining. Thus,

$$R_0 = N, R_{i+1} = R_i - C_i$$

$$C_i = \lceil \frac{R_i}{P} \rceil = \lceil (1 - \frac{1}{P})^i \frac{N}{P} \rceil$$

The value of R_i is decremented at each scheduling step. Allocating larger blocks of iterations to the first few processors reduces the scheduling overhead. As loop execution progresses, allocating smaller chunks smoothes out the variations in chunk execution times and thereby provides good load balancing. A simple variation specifies a minimum chunk size, C_{min} , to ensure that each processor is always assigned at least C_{min} iterations.

A feature of GSS is that approximately two-thirds of the remaining iterations are allocated every P chunks. Sometimes this results in the allocation of too much of the work in early chunks, so that they are too big and are not finished by the optimal time.

2.3.3 Factoring

Hummel et al. [16], suggest an alternate strategy, called *factoring*, that schedules iterations in batches of P equal-sized chunks. The total number of iterations per batch is a fixed ratio of those remaining (hence name *factoring*). After batch j is scheduled, a new chunk size C_{j+1} ($\leq C_j$) is determined, and P chunks of size C_{j+1}

are placed at the head of the scheduling queue.

A feature of factoring is that when iterations are constant-length, all processors execute an equal portion of the work (N/P). With factoring, the number of iterations allocated to each of the P processors at scheduling step i is $C_i = \lceil R_i/2P \rceil$. R_i is decremented at each scheduling step to give $R_{i+1} = R_i - PC_i$. This algorithm evenly distributes half of the remaining iterations among the P processors at each scheduling step, so all processors execute approximately the same number of iterations at the same time.

2.3.4 Trapezoid Self Scheduling (TSS)

Trapezoid Self Scheduling offers a compromise between fixed chunk scheduling and GSS and factoring, where there is a trade-off between scheduling overhead and better load balancing.

TSS uses a simple linear function to determine the size of the chunk allocated at each scheduling step. At the beginning, appropriate values of C_F and C_L , which are the sizes of the first and last chunks, respectively, are determined by the compiler or programmer. The decrease in the chunk size between scheduling steps is precalculated to be $s = (C_F - C_L)/(M - 1)$, where $M = \lceil 2N/(C_F + C_L) \rceil$ is the number of chunks to be scheduled and N is the total number of iterations. Thus, the first chunk size is $C_1 = C_F$ and the second is $C_2 = C_1 - s$. In general, $C_{i+1} = C_i - s$.

The flexibility of selecting (C_F, C_L) is an advantage of the TSS over the GSS. An optimal choice of (C_F, C_L) is dependent on the execution time distribution of the parallel loop. To have small M in the TSS, the (C_F, C_L) pair should be chosen as large as possible. However, the load imbalancing will be a serious side effect especially when the workload is not uniformly distributed. Nonuniformly distributed loops are often generated in real application programs. In order to ensure workload in every chunk is less than or equal to $\frac{N}{P}$, the first chunk size F is set to $\frac{N}{2P}$ in the

conservative TSS method. In this case, the load of first chunk (T_1) could be derived from the formula

$$T_1 = (N + (N - \frac{N}{2P})) * (\frac{N}{2P}) * \frac{1}{2} = \frac{(4P-1)*L}{4P^2}$$

L is the total load of a loop. It can be seen that T_1 is always less than $\frac{N}{P}$, thus the workload of the rest iterations is still adequate to keep other processors busy even under the decreasing workload. After the F is fixed by $\frac{N}{2P}$, due to the property of the linearly decreasing chunk size in the TSS, the range of the last chunk size, L , is from 1 to $\frac{N}{2P}$.

The name *trapezoid* comes from the observation that this linearly decreasing chunk-size function produces a trapezoid when chunk size C_i is plotted against scheduling step i .

2.3.5 Affinity Scheduling (AS)

Affinity Scheduling combines static and dynamic scheduling strategies. It maintains good load balancing while providing greater memory locality than a purely dynamic strategy.

With Affinity scheduling, $\lceil N/p \rceil$ iterations of the parallel loop are statically assigned to each processor such that the m th chunk is always initially allocated to processor m . At runtime, each processor maintains a local work queue of its assigned iterations. At each scheduling step, a processor removes a fractions of its queued iterations for execution. After executing all assigned iterations, a processor "snoops" the other processors to find additional iterations to execute. This ability to "steal" work lets the processors balance the load dynamically when the initial static assignment leads to an imbalance, but it also reduces memory locality.

Chapter 3

Parallel Query Processing on NOW

Query processing can also benefit from parallelism in NOW-based systems. Though it is not possible to distribute database over a NOW because each workstation is under the individual control, we can use the workstations to speed up query processing, which only requires read-only access to data.

Despite the additional overhead to transfer partial table among workstations, NOW-system is beneficial not only because it provides additional processor cycles but also adds substantial main memory for query processing by reducing the number of disk access. The improved performance is often obtained without any additional cost.

For query processing, there are significant differences between a parallel database system (usually based on multiprocessor systems) and a NOW-based system, though NOW is considered cost-effective alternative to multiprocessing systems. The main differences are the following:

- The database itself is not partitioned across the processing modules as in a parallel database. This causes additional overhead as the data have to be sent to the workstations to work on.
- The workstations in a NOW-based system are heterogeneous. The workstations could be of different types (ranging from low-end PCs to powerful work-

stations).

- In parallel database systems, load on each processing module is stable. This is a direct consequence of having a central scheduler and load balancer. In NOW-based systems, the load on each workstation can vary dynamically even while a query is being processed. This poses several problems to balance load across the system.
- Parallel database systems have a dedicated, special purpose fast communication network, whereas the NOW-based systems use general purpose LANs, which are slow and involve high communication overhead.

The above differences clearly indicate that performance issues in NOWs are different from those in a parallel database system.

3.1 Parallelism in Network of Workstations

The purpose of NOW is to simply use its additional processor cycles and memory available for database processing on a temporal basis. In particular, we focus on the query processing operations. Using idle workstations for database processing can improve performance of the system [6].

There are two types of parallelism in queries, namely *pipelined* and *partitioned* parallelism. Pipelined parallelism is beneficial in parallel database systems because of the high speed interconnection networks used to connect the processing modules and the ease of achieving **synchronous execution time** (synchronizing the processing of different processors involved in the processing of a particular query, eliminates the need for storing partial result), which is important for performance improvement using pipelining. However, the benefits of this parallelism will not be achieved on NOWs. The main reason is that the load variations are unpredictable and it becomes difficult to estimate the execution time of each query. Consequently,

synchronizing the dependent processing of the workstations involved in the processing of a particular query may not be possible. This may require temporary storage of partial results. On the other hand, a great deal of partitioned parallelism can be obtained in NOWs. These include:

- **Concurrent execution of multiple queries**

When multiple processors are available for query processing, we can schedule several queries concurrently. This improves system performance by reducing the waiting time of a query.

- **Exploiting intra-query parallelism**

The execution time of a query can be reduced by applying multiple workstations to work on a single query in parallel. This can be done by looking at the available parallelism in the query and scheduling the operations (e.g., joins) in parallel on different workstations.

- **Exploiting intra-operation parallelism**

The execution time can be reduced by applying multiple workstations to work on a single join operation in a query as well. This can be done in sending fragments of an operation to different workstations and working in parallel.

3.2 Query processing architectures on NOWs

In a NOW-based system, we assume that there is a special node responsible for running DBMS software. All other related operations are performed by this special node so that there will be no concurrency and recovery problems.

There are three major steps involved in managing parallel query processing: query interpretation and optimization, processor allocation and operation scheduling. The first step receives input queries (e.g., in SQL) and generates optimized query execution plans. This step is the same as that in traditional DBMS. The second step, processor allocation, determines the set of workstations that is used

for processing either an entire query or a part of it. The operation scheduler may actually schedule the workloads of the query. The operation scheduler may schedule an entire operation or can parallelize and load balance the operation. Three query processing architectures have been discussed [7].

3.2.1 Centralized architecture

The centralized architecture works in a master-slave mode. The master node performs all the work related to query preprocessing. The other workstations act as slaves and perform work assigned to them by the master node (Figure 3.1).

Queries are submitted at various workstations and forwarded to the master node for interpretation and processing. The master node interprets all queries received by the system and generates optimized query execution plans. Processor allocator takes the system state (i.e., non-processing load and the current query load) into account in making processor allocation decisions. In this architecture the master node is responsible for maintaining the latest system state information.

The processor allocation decisions can be made either statically at the beginning of the execution of the queries or can vary during the lifetime of a query. Load balancing (scheduling) can be done effectively in the centralized architecture, because allocation decisions are typically made at the operation level. We give more detailed description of the centralized architecture in Section 3.3.

3.2.2 Distributed architecture

In this architecture, all query preprocessing activities (i.e., query interpretation and optimization, processor allocation and operation scheduling) are done locally at the workstation that received the query. As a result, it eliminates the bottleneck problem which may exist in the centralized architecture.

Figure 3.1: Centralized architecture (from [7])

In the distributed architecture (as shown in Figure 3.2), processor allocation and load scheduling can be incorporated into load sharing. Load sharing has been extensively studied in the context of distributed systems. Ideas from those studies, though not directly applicable to the query processing, can be adapted to query distribution.

In the distributed architecture, each node receives and preprocesses the queries locally. Due to the dynamics of the query complexity and query arrivals, some nodes might be overloaded while others are underloaded. As discussed before, load on a node consists of a background load (i.e., non-query processing load on workstations) and query processing load. In distributed systems, CPU queue length is often used as an indicator of system load and threshold-based method is used to detect if a node is overloaded. In NOW-based systems, a weighted combination of the background and query loads is considered as an indicator of system load. Then a simple

Figure 3.2: Distributed architecture (from [7])

threshold can be used to determine if a node is overloaded.

This mechanism can classify a node as underloaded or overloaded or normally loaded. Load sharing achieved by transferring work from overloaded nodes to underloaded nodes. Typical policies on load sharing are sender-initiated, receiver-initiated, adaptive and hierarchical.

3.2.3 Hierarchical architecture

The hierarchical architecture is based, in concept, on the hierarchical run queue organization. In this architecture, a set of queues is organized as a tree with all the

nodes attached to the bottom level of the tree (i.e., as leaf node).

The hierarchical architecture is a self-allocation scheme. There is no central allocator making processor allocation decisions. The hierarchical architecture works as follows. As in the centralized architecture, all incoming queries are placed in the root queues (Figure 3.3). Let L be the leaf node level (i.e., node level) in the tree. When a node is looking for work, it first queries its associated query queue at level $(L-1)$. If that queue is empty it in turn queries its parent node of this node at level $(L-2)$ and the process is repeated up the tree until it finds work to be scheduled (unless the root queue is empty). However, in order to reduce access contention at higher levels, when a queue is queried, a set of work units is moved one level down the tree. The size of the set decreases progressively as one goes down the tree (taking into account the availability of the increasing number of queues per level). At the bottom of the tree this set size is reduced to just one work unit (corresponding to scheduling that work on the associated node).

Figure 3.3: Hierarchical architecture

It can be seen that the set of query queues that form the tree can all be distributed to different nodes without causing any hot spots. For a system with P nodes, the maximum number of query queues are required when the branching factor B is 2 and this number is equal to $(P-1)$. Since there are P nodes these $(P-1)$ queues can all be distributed without causing any communication and other contention problems.

3.3 Query processing in centralized architecture

In the centralized architecture, there are three types of nodes :

Coordinator: This is also called "master" node. All queries generated in the system are first sent to this node for processing. This node also maintains the whole system state information. In Coordinator node, there is a coordinator process, which is responsible for receiving queries from other nodes and performing processor scheduling. When there are no processors available, jobs are put into a *waiting queue*. All the system information is maintained in a *SystemInfoTable*, which includes each node's computational capacity, background load and current state. Three states are used:

- WORKING state means node is assigned to a job
- IDLE state means node is lightly-loaded and ready to accept a new job
- BUSY state means node is heavily-loaded and busy with local jobs

Ordinary node: These nodes perform the normal tasks of the system. These tasks may be workload (queries) generating, task scheduling and operations executing. Since these nodes send queries, report status to and expect tasks from the Coordinator, they are regarded as "slaves". There are four basic processes in an ordinary node: A query generator generates jobs at an arbitrary time intervals. A worker process is to do the actual job. In our case, it does the join operation in the query. A scheduler receives a query information and the partition of nodes assigned to the query from the Coordinator. It then

gets the tuples of involving tables from the Repository node, and schedules those nodes of given partition to work on the query. Upon completion, the query result is sent back. A background load info. process periodically create some processes locally as background loads.

Repository node: A special node where the database is physically located. Upon request, it sends the tuples of required tables in the database to node sending request. This is done by a database process.

Figure 3.4: Data and Query Flow in the Centralized Architecture

Queries are generated at ordinary nodes and sent to the Coordinator. The Coordinator, once receives a query, adds it at the end of the waiting queue. When there is a new query comes in or completed query leaves the system, the Coordinator recalculates the partition size and decides how many processors are to be given to each

query in the queue based on the current system state. It then starts to assign processors to queries waiting in queue until no more processors are available. Here we assume that requests for service are served in first-come-first-served order. Among those processors selected for a query, one of them is designated as *Scheduler*. The partition information along with the query is sent to the Scheduler. The scheduler schedules the tasks among those partitioned nodes to complete the job and result will be send back to the originator.

Figure 3.5: Three-level Centralized Architecture

Our scheduling procedure is a three-level strategy. The task at first level is done by the Coordinator. It determines how many processors are going to work for a

query, called *processor scheduling*. The task at second level, also done by the Coordinator, is to find which specific processors are to be allocated; though this task is easy in multiprocessor systems, more work needs to be done in NOWs. How those allocated processors work together on a given query to achieve load sharing is decided at third level, which is purely local to the assigned partition.

The reminder of this chapter gives details about these tasks.

3.3.1 Partition Size Calculation

Various techniques for multiprogramming parallel multiprocessor systems have been proposed as a way to improve performance. A natural approach is to divide the set of processing nodes into independent partitions and simultaneously execute a different parallel program in each partition. Three different strategies were proposed to determine the partition size [28].

- *Static strategies* assume that the number of processors allocated per job is a system configuration parameter which does not change. Static policies are simple and efficient to implement, but at a performance cost, since they do not adapt to various workloads. They work well when the workload is known and constant.
- *Adaptive strategies* adjust to various workloads by computing a job's partition size when it is scheduled. The partition size does not change until the job completes execution. The number of simultaneously executing jobs varies according to the specific strategy, the workload arrival patterns, and the service demands. The overhead is minimal, since the partition size computation is simple. Adjusting the partition size depends upon the current state of the processor set and the number of jobs requesting service increases the system's flexibility.
- *Dynamic schemes* allow the number of processors assigned to a job to vary during its execution. Processors can be taken away from any executing program

and reassigned to a newly arriving program. Similarly when a job finishes, the freed processors can be allocated to the other jobs that are still running. Depending on the underlying operating system platform, the overhead for re-allocating processors to processes during execution may be considerable.

An adaptive policy has been selected as our processor allocation policy because of the flexibility and efficiency. The algorithm we used to calculate partition size is derived from the basic formula.

$$PartitionSize = Max(1, \lfloor \frac{N}{Q} + 0.5 \rfloor) \quad (3.1)$$

N: The number of workstations in the system.

Q: The number of queries to be executed.

In a multiprocessor system, nodes are homogeneous, thus the total computational capacity of the system can be represented by the number of nodes in the system. In NOW-based systems, the computational capacity may differ significantly among these nodes due to their heterogeneity. Two low capacity nodes working together may not outperform a high capacity node. As a consequence, it is not appropriate to treat them as the same. In order to get accurate system information, a uniform criteria must be established among those nodes first. A new term, *processor unit* (PU) is introduced in instead of processor. The node with the lowest computational capacity is considered one PU. Other nodes, when compared their capacity with the lowest one, can be 2 PUs, 2.5 PUs and 3 PUs, etc.

We'll uniformly use the term *processor* in the rest of the paper. i.e. We'll use *processor* instead of *processor unit* in NOW system, while it actually refers to *processor unit*.

We then design our first formula for computational partition size in NOW systems [28].

$$PartitionSize = Max(1, \lfloor \frac{P}{Q} + 0.5 \rfloor) \quad (3.2)$$

P: The number of processor units in the NOW system.

Q: The number of queries to be executed.

However, the formula 3.2 overreacts to changes in the queue length [28]. A more conservative method, tends toward premature fragmentation is [28] :

$$PartitionSize = Max(1, \lfloor \frac{P}{Q+1} + 0.5 \rfloor) \quad (3.3)$$

That is, when a single job arrive at an empty system, only 50% of the available processors will be allocated to it.

Partition size calculation depends not only on the jobs in the waiting queue, but also on those being executed [34]. This means the jobs of the whole system are taken into account. The formula is improved as [34]

$$PartitionSize = Max(1, \lfloor \frac{P}{Q + f * E + 1} + 0.5 \rfloor) \quad (3.4)$$

P,Q: Same as previous formulas

E: The number of job currently being executed

f: A parameter used in the experiment. When $f = 0$,

formula 3.4 is equivalent to formula 3.3.

At the first level of our strategy, we consider the system capacity as *static*, which is a constant value achieved at system configuration time. We assume the workload at each node is low as idle when deciding the partition size. Thus we simplify the computation of partition size without loss of generality. The total number of processors P in the NOW systems can be known at the configuration time. Workload issues are left to be handled later within the next two steps. Also at the first level, the workload information is collected from ordinary nodes and system state is kept updated.

Algorithm at the First Level :

Input P: Number of processors in the system

f: A preassigned value

system information is maintained in SystemInfoTable

Output: PartitionSize

$Q \leftarrow 0$ (waiting queue length)

while (message coming from ordinary nodes) do

begin

if (workload information)

update SystemInfoTable accordingly

else

if (new query arrival)

{

add new query at end of waiting queue

$Q \leftarrow Q + 1$

}

if (query completion)

{

$Q \leftarrow Q - 1$

}

$E \leftarrow \text{NumberOfQueryBeingExecuted}$

$\text{PartitionSize} \leftarrow \text{Max}(1.0, \lfloor \frac{P}{Q+f \cdot E+1} + 0.5 \rfloor);$

}

PartitionAllocation (see level 2)

end

3.3.2 Processor Partitioning

Once the partition size is determined, the next step at second level is to actually partition the available nodes into subsets and assign them to each query. By saying "available", we mean the node should be idle (background workload lower than a pre-assigned value) and it should not have been assigned to other query in the system already (Here we assume each node will dedicate to only one job at a time). In multiprocessor system, it is easy to do partitioning as all nodes are identical to each other. One can pick up any partition whose size is equal to or larger than the PartitionSize achieved at the first level.

In a NOW-based system, however, it is much more complex. This is due to the heterogenous computational capacity and workload on those system nodes. There are at least two issues we have to consider:

How to partition? Despite the fact that there exist several ways of partitioning, the candidate partitions obviously should have size not less than the required PartitionSize. Among these appropriate candidate partitions, our goal is to find the set of nodes, the sum of whose processor units has the least difference from the required size. For example, if PartitionSize = 6, meaning minimum 6 processors are requested; and we have four nodes available with 3, 4, 4, 5 processors respectively, we will choose the first two nodes in our policy.

Choosing among partitions with same size. The second issue we encounter is if, two or more partitions have exactly same size, which should be selected? This is not a problem in a multiprocessor system. Since all nodes are similar, it makes no difference which partition is selected. In NOW-base systems, we do have some concerns in making such decisions. Our policy is to choose the partition with the fewest nodes due to the following considerations:

- The overhead of communication among those nodes
- The less possibility of the node involvement

To enact our policy, we sort all the system nodes in a decreasing order of their available computational capacity. We consider the availableComputingCapacity (aCC) of a node as

$$\text{availableComputingCapacity} = \text{ComputingCapacity} * (1 - \text{CurrentWorkload}) \quad (3.5)$$

When no partition that meets the PartitionSize requirement can be found, queries will stay in the queue until next new query comes in or a query completes. (see algorithm below).

Algorithm at Second Level:

Input : PartitionSize, SystemInfoTable, Number of Queries in waiting queue

Output: Partitioning Result

Sort system nodes in decreaseing order of avaialbleComputingCapacity

availableProcessors $\leftarrow$ *getFreeProcessors(SystemInfoTable)*

QueueLength $\leftarrow$ *number of queries in waiting queue*

while (QueueLength > 0 AND availableProcessors $\geq$ PartitionSize) do
begin

SelectedPartition $\leftarrow$ *Partitioning(SystemInfoTable, PartitionSize)*

If (SelectedPartition is empty) goto end;

Scheduler $\leftarrow$ *node with most aCC in SelectedPartition*

send SelectedParititon and query information to Scheduler to execute

availableProcessors $\leftarrow$ *availableProcessors - getFreeProcessors(SelectedPartition)*

queuelength $\leftarrow$ *queuelength - 1*

update state of nodes in SelectedPartition to WORKING

end

getFreeProcessors

Input: a set of nodes

Output: total available processors in the node sets

freeProcessors $\leftarrow 0$

For each nodes in the set of nodes do

sum $\leftarrow$ *sum* + *availableComputingCapacity*

return sum;

*Partitioning Algorithm*¹

Input: SystemInfoTable. PartitionSize

Output: a set of nodes (partiton)

PNumber $\leftarrow$ *total number of nodes in the system*

finalPartition, tempPartition $\leftarrow \emptyset$

finalSum, tempSum $\leftarrow 0$

for (*k*=0; *k* < *PNumber*; *k*++) *do*

Begin

for (*i*=*k*; *i* < *PNumber*; *i*++) *do*

Begin

if (*tempSum*+*aCC*[*i*]) $\geq$ *PartitionSize*) {

for (*j*=*i*+1; *j* < *PNumber*; *j*++)

if (*tempSum*+*aCC*[*j*]) $\leq$ *PartitionSize*) *break*;

i=*j*-1;

}

tempSum += *aCC*[*i*];

add node i to tempPartition

End

if (*tempSum* < *PartitionSize*) *return finalPartition*;

if (*finalSum* == 0 OR *tempSum* < *finalSum*) {

finalSum $\leftarrow$ *tempSum*;

finalPartition $\leftarrow$ *tempPartition*;

¹In experiment, a simplified version of algorithm is applied to reduce the scheduling overhead


```
        tempPartition  $\leftarrow \phi$  ;  
        tempSum  $\leftarrow 0$ ;  
    }  
End  
return finalPartition;
```

3.3.3 Operation Scheduling

Early in this chapter, we mentioned three types of parallelism pursued in NOW-based systems. In our three-level strategy, concurrent query execution is implemented by partitioning system nodes into subset and executing a query in each partition. This is done at first and second levels as discussed in the previous two sections. Within a partition, intra-query and intra-operations are exploited by parallelizing join operations in a query and doing partial operations of a join. These functions are done at the third level between the Scheduler node and the partition nodes.

A query is made up of several join operations. A join operation involves two relations with thousands or millions of records each. Basic idea for operation scheduling is to determine

- how partition nodes work together on a query's join operations and
- how partial operations on a join are managed to be performed.

The decision is made by the Scheduler. The Scheduler determines how many join operations at each level (in balanced query) and which nodes are going to work on a particular join. For example, in a 8-relation query, there are four joins at the bottom level. Depending on the number of nodes in the partition, the four joins could be executed concurrently by different groups of nodes; or in another way, all nodes work intensively on one or two joins first. The rest will be done afterwards.

When executing a particular join, partial operation is processed by "chunking" the involved records and assigning chunks of records in turn to each node for execution (called load distribution). Initially, each node is given same chunk size records. Upon receiving a partial result, the Scheduler assigns another chunk, if there are, to the node. This chunk size maybe a fixed or variable one according to the nodes' performance. For example, if the node finishes its last assignment faster than other nodes do, it is considered having high capacity, and are therefore assigned a larger chunk size. Several methods of determining chunk size are surveyed in Chapter 2. By this way, the query load is shared by all the partition nodes to achieve overall good system performance.

The scheduler continues to assign workload on each nodes until the whole query is done; it then sends the result back to the query originator and, at the same time, informs the Coordinator to release this partition for other queries. Each node working on a join operation acts as "slaves". it simply receives chunks of records and doing the database join and send the join result back to Scheduler upon completion.

The Algorithm at third level:

The Scheduler process²

Input S: a given partition

Q: the query to be executed

Output: query result

result $\leftarrow \phi$

while (unfinished join exists) do

Begin

records $\leftarrow$ *GetQueryRecords;*

²This is a simplified algorithm while the algorithm applied in the implementation is more complete

```

/*This is done by asking the repository nodes */
send initial chunk of records to each node in S
while (partial result coming from a node ) do
{
    result  $\leftarrow$  result + partial result
    chunk  $\leftarrow$  getNextChunk
    if (chunk exists)
        send chunk of records to the node for execution
}
send result back to the node where query is generated
notify the Coordinator the completion of the query
End

```

Working nodes (slave) process

Input: chunk of records of relations in a join

Output: partial join result

```

partialResult  $\leftarrow$   $\phi$ 
for each record i in chunk of first relation do
Begin
    for each record j in chunk of second relation do
Begin
        if ( i.key == j.key )
        {
            record.key  $\leftarrow$  i.key;
            record.attribute  $\leftarrow$  i.attribute combine j.attribute
            partialResult  $\leftarrow$  partialResult + record
        }
    End
End

```

End

return partialResult

Chapter 4

Experimental Model and Implementation

In last chapter, we described our three-level strategy for query processing in the centralized architecture of NOW-based systems. We implemented this strategy on the Triton system, a network of workstation. In following, we give details on the design and implementation.

4.1 Triton Parallel System

The Triton Parallel System we used in the experiments is a Network of Workstations. This system is composed of a number of Pentium processors linked with 10Mb/s and 100Mb/s networks (Figure 4.1). Each node has 32Mb of RAM. The main node, known as "triton" from the outside, is a P6/200 with a hard disk. This is the only node that can be accessed from the outside, and the only node that supports X11.

Nodes are listed in Table 4.1. From the table we can see, besides triton (n_0), there are 4 different types of nodes in Triton. Nodes 1,2, 9 are Pentium 133 with slow caches. Node 4 is also Pentium 133 but with fast cache. Nodes 5,6 are Pentium 166. Node 3, 7, 8 are Pentium 200. The effect of the processor type and speed on their computational capacity is given in the next chapter.

Figure 4.1: Triton Parallel System

Triton is a research system, the status of nodes changes from time to time. The command *sysinfo* display the status of each node in the system as in:

```
%sysinfo
```

```
triton up 46 days, 23:44. 3 users, load average: 0.16 0.05 0.01
n1 up 34 days, 9:18, 0 users, load average: 0.00 0.00 0.00
n2 up 68 days, 19:29, 0 users, load average: 0.00 0.00 0.00
n3 up 0 users, load average: 0.06 0.01 0.00
n4 up 0 users, load average: 0.00 0.00 0.00
n5 up 45 days, 17:57. 0 users, load average: 0.00 0.00 0.00
n6 up 6 days, 1:46, 0 users, load average: 0.00 0.00 0.00
```

<i>Node</i>	<i>CPU</i>		
	Type	RAM	Cache
N_0	P200	32	n/a
N_1	P133	32	slow
N_2	P133	32	slow
N_3	P200	32	n/a
N_4	P133	32	fast
N_5	P166	32	n/a
N_6	P166	32	n/a
N_7	P200	32	n/a
N_8	P200	32	n/a
N_9	P133	32	slow

Table 4.1: Nodes of Triton System

```

n7 up 6 days, 19:18, 0 users, load average: 0.00 0.00 0.00
fn1 up 34 days, 9:18, 0 users, load average: 0.00 0.00 0.00
fn2 up 68 days, 19:29, 0 users, load average: 0.00 0.00 0.00
fn3 up 0 users, load average: 0.05 0.01 0.00
fn4 down
fn5 up 45 days, 17:57, 0 users, load average: 0.00 0.00 0.00
fn6 up 6 days, 1:46, 0 users, load average: 0.00 0.00 0.00
fn7 up 6 days, 19:18, 0 users, load average: 0.00 0.00 0.00

```

This shows that all the 10Mb/s connections are up, i.e. nodes known as nX are connected through 10Mb cable and the same nodes connected through 100Mb/s are known as fnX (fast node). The above display shows that n4 is up, but the 100Mb connection to that node is down i.e., fn4 is down. It is possible for a fnX to be down, but if nX is down the node can not boot, and fnX is not possible. In summary there are currently 9 Pentium motherboards, connected through multiple networks, with 1 node with a disk, and all others diskless.

In order to allow multiple users at the same time, when doing time critical ex-

periments, a user can allocate nodes for his/her exclusive use. This allows users to share the system, and have as little impact on each other as possible.

Node allocation is done with command *allocate*. It has 5 modes of operation

- allocating a number of nodes and creating `/hosts` file: `allocate -c 4`
- as above but node 0 is not included in `/hosts` file: `allocate -c -e 4`
- allocating a number of nodes: `allocate 4`
- allocating specific nodes: `allocate n2 n4 n7`
- displaying allocated nodes: `allocate -s`

Node freeing is done with command *release*. It has 2 modes of operation:

- release selected nodes: `release n3 n5`
- release all nodes: `release -a`

Please note that node 0 (triton) can not be allocated, as it has to be used by all users.

4.2 Database Model

The model of the database considered consists of a number of tables. Each table contains a large number of tuples. Each tuple has several attributes. In our database model, tables are actually regular Unix files, each file has 5000 to 9000 records. A record is made up of two fields, as two attributes in a tuple. The first field is key attribute and second field are considered as other attributes. Both fields are actually long integers with size 4 bytes. Thus a tuple have size 8 bytes. We created 16 tables for the purpose of experimentation.

Among these tables, two of them are different from the others. The key value in this two tables are unique, sequential long integers. There is no key duplication. In the rest fourteen tables, however, the key values in each are normally distributed with the same mean value and different variance for the keys in every table. By having these two kinds of tables, we intend to obtain different selectivities for the join operations between different tables.

	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	0.8	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	3.4	0.7	0.6	0.7	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	2.4	0.5	1.4	1.5	0.7	1.5	1.2	1.1	4.9	2.7	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	0.8	0.8	0.5	0.8	0.8	0.6	0.8	0.7	0.7	0.8	0.9	4.6	0.7	0.8	0.8
13	1.1	1.1	1.0	0.5	1.0	1.0	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 4.2: Selectivities among tables

The selectivities of join operations among the 16 tables in the database model are shown in Table 4.2. The first row and first column give the id (number) of the database tables. The rest are the selectivities of join operations between the tables and the matrix of selectivities is symmetric around the diagonal. The unit of selectivity values in the table is 0.01%. The selectivities are from 0.0043% to 0.059% and the average value of all the selectivities is 0.029%. Table 2 and 9 have unique key values and other tables have their key values normally distributed. The variance of the key values in tables 3,6,12 and 13 are higher than those in the rest

of the tables. With higher variance, the probability that the keys in a table have the same value becomes less. Thus, the selectivities of the join operations between this table and other tables become less. It can be noticed that the selectivities of join operation in row 3, 6, 12, 13 and column 3, 6, 12, 13 are relatively smaller.

4.3 Query Workload Model

Based on the database model and given selectivities among tables, we generated 16 different type of queries as our jobs. Each query type in our model contains the information of how many tables and which specific tables in the database are involved in the query. Each of the 16 queries have 2, 4, or 8 tables involved. In half of these query types, tables of join operations with higher selectivities are selected, so that we can model queries with growing-sized query result. In the other half query types, tables are selected from those with lower selectivities to produce queries with shrinking-sized result. All the queries generated during the experiment are randomly selected from one of these queries.

Figure 4.2: Balanced and Linear query with 4 tables

We also consider two types of query, namely *balanced query* and *linear query*.

In a balanced query, the query structure is a balanced binary tree as shown in Figure 4.3(a). The linear query creates a left side binary tree (Figure 4.3(b)). Performance sensitivity to query structure is given in the next chapter. Theoretically, balanced query tree takes more advantage of parallel processing. Because inter-query, inter-operation and also intra-operation parallelism can be exploited, while linear queries can not exploit inter-operation parallelism. Most of our experiments are done on balanced queries. The number of original tables in the query determines the number of operation levels in the query. There could be 1-3 levels of operations in each balanced query and 1-7 levels of operations in each linear query.

In query generator, queries are generated by randomly picking up one of the 16 query types as either a balanced query or a linear query. This is done by using specific set of tables in the selected query type. The query generator sleeps for certain time after each query generation and continues to generate next query. In the simulation, the arrivals of queries are exponentially distributed.

4.4 Parallel Virtual Machine (PVM)

Our implementation is done on Linux operating system with the help of Parallel Virtual Machine software package. PVM allows a heterogeneous network of parallel, serial and vector computers to appear as a single concurrent computational resource (i.e., a distributed-memory system). PVM consists of two parts:

- A daemon process that any user can install on a machine; and
- A user library that contains routines for initializing tasks on other machines, communication and changing configuration of the PVM machine.

Applications using PVM can be written in Fortran77 or C/C++ and parallelized by using message-passing constructs common to most distributed-memory computers. PVM supports heterogeneity at the application, machine, and network level. In other words, PVM allows application tasks to exploit the architecture which is best

suited to their solution. For example, PVM may include a parallel machine and a graphical workstation. Computation can be done on the parallel machine and the results may be displayed in graphical form on the workstation.

PVM handles all data conversion that may be required if two computers use different integer or floating point representations. If PVM machine is to contain machines with different architectures, programs need to be compiled on each one of different architecture.

Chapter 5

Performance Analyses

In this chapter, we analyze our experimental results and evaluate the performance of the centralized architecture for parallel query processing on NOWs. The performance metric used in these experiments is the mean response time of the queries. For each experiment, 31 batches of queries with a batch size of 50 - 100 queries are used. The results of the first batch are discarded as the system starts cold. All the experiments achieve 95% confidence intervals to ensure the independent data values. The value of the confidence interval for each statistical data are, while none exceeds 10%, less than 5% in most cases.

Several parameters play an important role in the performance, such as number of processors, system inter-arrival time. fragmentation size for partial operation, query types etc. Next we discuss how the system performance is sensitive to these parameters.

5.1 Some Preliminary Experiments

Before the proceeding with the actual performance experiments, some preliminary system data need to be obtained.

Node	n7	n8	n3	n5	n6	n1	n2	n9	n4(1)	n4(2)
0	11.8	11.8	11.9	24.7	24.9	30.7	30.5	29.6	34.4	34.4
1	3.4	3.4	3.5	7.5	7.0	9.1	8.6	9.0	10.1	10.1
2	3.4	3.4	3.4	7.3	7.9	8.8	8.6	9.0	10.1	10.1
3	29.5	29.6	29.7	65.4	61.8	75.4	76.6	74.3	86.3	86.4
4	19.1	19.1	19.1	40.9	41.1	47.8	49.2	47.9	55.8	55.8
5	6.3	6.3	6.3	13.1	12.9	16.3	16.2	15.9	18.5	18.5
6	8.4	8.4	8.5	19.0	17.3	21.1	21.6	22.2	24.6	24.6
7	2.5	2.5	2.5	5.0	5.0	6.1	6.1	6.5	7.2	7.2
8	9.7	9.7	9.7	20.4	20.6	25.3	25.4	24.7	28.3	28.3
9	8.0	8.0	8.0	16.6	16.4	21.8	21.2	19.9	24.7	23.4
10	13.0	13.0	13.0	27.2	26.7	32.5	33.4	33.3	38.2	37.9
11	3.4	3.5	3.4	7.5	7.5	9.2	8.8	9.5	10.1	10.0
12	12.7	12.8	12.8	26.7	27.2	31.9	31.9	32.6	37.2	37.2
13	2.5	2.5	2.5	5.0	5.5	6.1	6.4	6.6	7.2	7.2
14	7.0	7.1	7.1	14.6	14.9	17.5	17.9	18.7	20.6	20.6
15	2.8	2.8	2.8	5.8	6.0	7.0	7.0	7.0	8.2	8.2
mean	9.0	9.0	9.0	19.2	18.9	22.9	23.1	22.9	26.3	26.2

Table 5.1: Time (second) of executing 16 queries on each node

5.1.1 Node Computational Capacity

The first experiment is given to find out the computational capacity of each node in the Triton system. Despite the fact that the CPU type is given, each node has to be mapped onto our node capacity measure metric, the *processors unit*, i.e. the number of *PUs* of each node in the Triton system needs to be figured out first. The experiment is done by executing the 16 different types of queries consecutively (without queueing) on each node and compare the average execution time of each node on the queries. The results of this experiment are shown in Table 5.1.

Experiment on node 4 is done twice since there is only one node of this type. We find out that a node with Pentium 200 processor executes nearly 2.8 times as fast as a node with Pentium 133 processor. We give the computational capacity of the slowest node n4 as 100 *processor units*. The computational capacity of each node in our experiment as shown in Table 5.2.

CPU	PU
P200	280
P166	138
P133 (slow cache)	116
P133 (fast cache)	100

Table 5.2: Processor Unit of each node used in experiments

Node	R1(sec.)	R2(sec.)	R3(sec.)
0	5.3	5.5	5.6
1	1.7	1.9	1.9
2	1.8	1.6	1.6
3	13.7	13.3	13.9
4	9.5	9.4	9.5
5	3.2	3.1	3.2
6	4.0	4.1	4.0
7	1.2	1.1	1.2
8	4.8	4.8	4.8
9	4.0	4.1	4.1
10	6.6	6.8	6.8
11	1.7	1.6	1.6
12	6.6	6.7	6.7
13	1.3	1.2	1.2
14	3.8	3.6	3.8
15	1.4	1.3	1.3
mean	4.41	4.38	4.45

Table 5.3: Three independent experiment results on system capacity

5.1.2 System Capacity

Another experiment is to check the system's capacity; that is, how fast the system can execute a query. The experiment is done on the 8-node system. We have used the same 16 queries used in the last section. All nodes are given the same number of tuples. The result (Table 5.3) of three independent experiments shows that the system can process one query per 4.4 seconds. As a result, our first assumption is that if the queries come into the system at a rate of one every 4.4 seconds or faster, the system will not be able to handle and become saturated. However, experiments show that this is not the case when load is not equally distributed.

Since all 8 nodes are given same workload, fast nodes apparently finish earlier than the slow nodes. However, these experiments do not take advantage of the fast node as work is equally distributed. Since the fast node of Pentium 200 is almost three times as fast as that of P133, further improvement in performance is possible by using load sharing techniques. Our experiment shows at the rate of one query every two seconds, the system still works, but with a high response time. The system saturates when the inter-arrival time is less than two seconds.

5.2 Sensitivity to System Inter-Arrival Time

System inter-arrival time refers to the average length of time for the system to generate a new query. If the queries enter the system with short inter-arrival times, the workload of the system increases. If, on the other hand, queries enter into the system with a long interval between each other, the system could be regarded as lightly loaded.

Results for the 8-node centralized architecture is shown in Figure 5.1. The fragmentation size used is 200000 bytes. This means no partial operation, at least at first join level. From this figure we can see that when inter-arrival time is small, queries enter into the system at a high rate and these queries have to wait in the queue because the system is busy with other queries. Experimental data suggests that the increase in the mean response time is mainly caused by the increase of query waiting time.

5.3 Sensitivity to Fragmentation Size

The fragmentation size determines the number of partial operations allowed. The larger the fragmentation size, the fewer the number of partial operations created. A smaller fragmentation size leads to more partial operations and larger scale of parallelism. Thus, load can be distributed to achieve fairly good load sharing. On

Figure 5.1: Performance Under Different Inter-Arrival Time

Figure 5.2: Performance under different fragmentation size when system is lightly loaded

the other hand, with smaller partial operations, the overhead of communication and operation scheduling increases. This trade-off implies that system response time may not always decrease as the fragmentation size becomes smaller.

Figure 5.2 shows the system performance under different fragmentation size when the system is relatively lightly loaded. It can be observed that for most of fragmentation sizes, the larger it is, the higher mean response time. The smaller the fragmentation size, the better system performance. As shown in the figure, mean response time under fragmentation size 200000 bytes is 19.5 seconds. The mean response time under fragmentation size 40000 bytes is 16.6, 1.5% lower than the former. The mean response time under fragmentation size 24000 and 5000 bytes are 10.1 and 7.6 seconds respectively, 48.2% and 61% lower than that under fragmentation

Figure 5.3: Performance under different fragmentation size when system is heavily loaded

size 200000 bytes. When the fragmentation size decreases to 1000 bytes, it shows that the response time increases to 14.8 seconds, which indicates that the benefits obtained from partial operations are less than the communication and scheduling overhead.

In the next experiment, system inter-arrival time is set to 6 seconds. This means the system is heavily loaded. We get similar result (Figure 5.3). The experiment also shows that a smaller fragmentation size leads to better performance.

5.4 System Speedup

Ideal parallel systems demonstrate two key properties, namely *linear speedup* and *linear scaleup*. Linear speedup refers to the fact that twice as much hardware can

perform the same task in half the time. More formally, given a fixed job, the *speedup* is given by

$$Speedup = \frac{small - system - elapsed - time}{big - system - elapsed - time} \quad (5.1)$$

Speedup is said to be linear, if an N-times larger system yields a speedup of αN , for some constant α .

Two experiments have been done to evaluate the system speedup. In the first experiment, systems with 1, 2, 4 and 8 nodes are given the same amount of light load that each system is able to handle. Figure 5.4 shows the performance result of the experiment. The fragmentation size used is 200000 bytes. Queries arrive into the system every 24 seconds on average. The throughput obtained for all the systems are around 24 seconds per query. With this workload, the mean response time on 1, 2, 4 and 8 nodes systems are 89.5, 26.9, 23.4 and 19.3 seconds, respectively. This gives the speedup of 2-nodes system to 1-node system is 3.4, while speedup of 4-nodes and 8-nodes systems to 1-node system is 3.8 and 4.6, respectively. From these data, it can be concluded that the centralized system is successful in providing speedups under light load.

In the second experiment (Figure 5.5), the inter-arrival time for systems with 1, 2, 4 and 8 nodes are set to be 6 seconds. This causes all the systems to be in a heavily-loaded state. For this experiment, the 1-node and 2-nodes systems get saturated. Both 4-nodes and 8-nodes system work fine under this workload. The 4-nodes system achieves mean response time of 25.1 seconds; while the 8-nodes system achieves a response time of 19.9 seconds under the same load. The speedup of 8-nodes system to 4-nodes system is 1.3.

From the two experiments, we conclude that centralized architecture achieves good speedup under light load; while in the heavy load condition, 8-nodes system achieve 1.3 speedup to 4-nodes. The 4 nodes we chose in the experiment have a total computational capacity greater than half of the 8-nodes. This might be the

Figure 5.4: System Speedup under light load

Figure 5.5: System Speedup under heavy load

reason why we do not achieve a more linear speedup. It is speculated that if the 8-nodes system has exact double or even more capacity than the 4-nodes system, we can get a better speedup.

5.5 System Scaleup

Scaleup refers to the fact that twice as much hardware can perform twice as large a task in the same time. While speedup holds the problem size constant, and grows the system, scaleup measures the ability to grow both the system and the problem. Scaleup is defined as the ability of an N-times larger system to perform an N-times larger job in the same elapsed time as the original system as given in the metric

$$Scaleup = \frac{small - system - elapsed - time - on - small - problem}{big - system - elapsed - time - on - big - problem} \quad (5.2)$$

Figure 5.6: System Scaleup

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

A longer inter-arrival time means lighter workload. In our experiment, we assume that twice of the inter-arrival time leads to half the amount of work. Performance results of 8, 4, 2 and 1 node systems are given in Figure 5.6. The 8-nodes system has 6 seconds inter-arrival time, leading to twice as much work as that in the 4-nodes system whose inter-arrival time is 12 seconds. The inter-arrival time of 2-nodes and 1-node systems is set to 24 and 48 seconds, respectively. The mean response time of these systems are 20.0, 26.7, 29.5 and 31.8 seconds. This indicates that the 8-nodes, 4-nodes and 2-nodes system can finish 8 times, 4 times and 2 times amount of work of 1-node system but within a less time. It can be concluded from this data that centralized architecture can achieve good system scaleup.

Figure 5.7: Performance with balanced and linear queries

5.6 Sensitivity to Query Type

So far we have analyzed the system performance sensitivity to number of nodes, fragmentation size and system speedup and scaleup of the centralized architecture. In all these experiments, the queries generated are of balanced query type. In this experiment, linear queries are generated by using the same database set. Figure 5.7 gives the experimental result of linear queries under different inter-arrival time, in comparison to that of balanced queries. The fragment size is 200000 bytes. Thus no intra-operation tasks exist in linear queries.

The result shows that linear queries gain better performance than the balanced queries under same parameters. This might be due to our implementation policy that assigns nodes with higher capacity to workload with a higher priority. For example, in our experiment with 8 nodes, we have 3 Pentium 200 processors which are able to handle most of the load without causing the waiting queue getting too

long. Their execution speed matched the query incoming rate, thus nodes with lower capacity have less chance to work on query. This, though wastes some system resource (some low capacity nodes), increases the system's overall performance. While in balanced query cases, though many of nodes may be assigned to work on a query, but due to the lack of partial operation, only part of the partition is used. At the same time, when fast nodes work together with slow nodes, the system performance is actually worsen by the latter.

It could be expected that with smaller fragmentation size and optimal processor allocation policy, the balanced queries will outperform the linear queries.

5.7 Sensitivity to Parameter f

Formula 3.4 decides the partition size in our strategy. There is a parameter f in the formula that affects the partition size. When the other parameters remain the same, a larger f leads to a smaller partition size. A small partition size gives a better chance for the processor allocator to find a free partition of nodes for a query. A large partition size indicates that, though there might be some low capacity nodes that are idle, they could not be assigned to a query. So queries may have to wait in the queue for longer time for more nodes to be available. This increases the query waiting time which increases the response time. On the other hand, assigning smaller partition to a query may reduce its waiting time, but execution time might increase.

Figure 5.8 shows the system performance sensitivity to different parameter f . The fragmentation size used in the experiment is 24000 bytes, which means partial operations are used. The system inter-arrival time is 6 seconds. Our experimental data (Figure 5.9) shows that when f increases, the query waiting time decreases. This leads to decreased response time. When $f=1.0$, though the waiting time is the smallest, the increase in its execution time brings a worse overall performance than

Figure 5.8: Performance Under Different Parameter f

Figure 5.9: Execution and Waiting Time Under Different Parameter f

when $f = 0.75$. So our conclusion is that the centralized architecture obtains the best performance when $f = 0.75$. In most of our experiments, we set f to 0.75. It has also been observed that $f = 0.75$ also provides the best performance even in distributed-memory multicomputer system [34].

Chapter 6

Conclusion and Future Work

6.1 Summary

This thesis has studied the parallel query processing on Network of Workstations (NOWs). Our main objective was to design a three-level parallel query processing strategy for the centralized architecture on NOWs and evaluate its performance.

Early in the thesis, we have introduced both parallel processing techniques and parallel database system. The motivation for our research was also stated. Background information on query processing, load distribution and loop scheduling has been presented in Chapter 2. The main contribution of the thesis is in Chapters 3 and 4. In Chapter 3, we presented a detailed description of our strategy for the centralized architecture. Our experimental results are given in Chapter 5.

From our experiments, it has been observed that small system inter-arrival time causes the increase of response time. This is mainly due to the increase of query waiting time. Small fragmentation size leads to a better chance of partial operation and larger scale of parallelism. In this case, the mean response time decreases because of the decreases in query execution time. However, when the fragmentation size is too small, the overhead in scheduling large number of partial operations dominates the benefits obtained from this parallelism.

Linear query in our experiments outperforms the balanced queries. This is mainly due to the parameter values we have chosen that are not in favor of partial operations. We believe that with carefully selected parameters, the balanced queries will demonstrate a better performance.

6.2 Contribution of Thesis

This section states the contribution of the thesis in the area of parallel query processing on Network of Workstations.

Most of the previous studies regarding parallel processing in the database area have focused on parallel database systems, which are mostly built on multiprocessor systems. On the other hand, though some attention has been paid to the usage of Network of Workstations recently, very little work has been done in the database area. This thesis is one of the few studies on parallel query processing using NOWs. Particularly, this is the first time a three-level query strategy is designed and implemented for the centralized architecture. And it is the first to give a comprehensive performance evaluation of the centralized query processing on NOWs.

Our study has shown that centralized architecture achieves both system speedup and scaleup. It has also been found that the system performance is very sensitive to fragmentation size.

6.3 Future Research

Due to the time constraints, we could not complete all possible simulation experiments. But the existing simulator can provide a good test bed for any later simulation experiments. In the following a few research issues are identified.

Our implementation uses a fixed-size operation scheduling policy. This results in poor usage of the system resource in certain situations. For example, in a 2-node system, if the computational capacity difference between the two nodes is big, the fast node will waste a lot of CPU time to wait for slow node to finish. There might exist some other policy (as described in Chapter 2) which can achieve a better overall performance.

The main advantage of centralized system is that system state is kept in a central node which makes it possible to achieve good load sharing. However, since the fragmentation size we used in most of the experiments are so large that there is almost no partial operation and leads to small scale of parallelism. Experiments show that a small fragmentation size leads to more partial operation. However, the overhead in handling large number of fragments increases at the same time. Future research should focus on the optimal fragmentation size.

Also, performance comparison among the centralized, distributed and hierarchical architecture is interesting. This would be helpful in understanding when and under what situation an architecture is more appropriate.

Some other issues in NOWs are not considered in the experiments, such as fault tolerance, communication overhead etc. The impact of these issues on the overall system performance in the centralized architecture needs further research.

References

- [1] T.Anderson, D.Culler, and D. Patterson, "A Case for Networks of Workstations: NOW", *IEEE Micro*, Feb. 1995.
- [2] A. Giest, A.Beguelin, J. Dongarra. W. Jiang, R.Mancheck, V.Sonderam, "PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing", *The MIT Press*, 1994
- [3] H. Boral, W.Alexander, Lclay, G. Copeland. S.DanForth, M. Franklin, B.Hart, M. Smaith. and P. Valduriez, "Prototyping Bubba, A Highly Parallel System", *IEEE Transactions on Knowledge and Data Engineering*. Vol. 2, No. 1, March 1990, pp 4-23
- [4] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Mancheck, K.Moore, and V.S. Sunderam, "Tools for Heterogeneous Network Computing", *Proc. 6 th SIAM conf. On Parallel Processing For Scientific Computing*, 1993
- [5] Y-N Chan, S.Dandamudi. "Performance Comparison of Processor Scheduling Strategies in a Distributed-Memory Multicomputer System", *Technical Report TR-97-1*, School of Computer Science, Carleton University, 1997
- [6] S. Dandamudi, "Using Networks of Workstations for Database Query Operations", *Int. Conf. On Computers and Their Applications*, Tempe, Mar. 1997, pp. 100-105
- [7] S. Dandamudi and G. Jain, "Architectures for Parallel Query Processing on Networks of Workstations", *Technical Report TR-97-11*, School of Computer Science, Carleton University, Canada, 1997

- [8] S. Dandamudi, K.C.M. Lo, "A Hierarchical Load Sharing Policy for Distributed System", Technical Report TR-96-1, School of Computer Science, Carleton University, Canada, 1996
- [9] D. J. Dewitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hshiao and R.Rasmussen, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp 44-61
- [10] D. DeWitt and J.Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communications of the ACM*, Vol.35, No. 6, June 1992, pp. 85-98
- [11] F. Douglass, J. Ousterhout, "Transparent Process Migration: Design Alternative and the Sprite Implementation". *Software Practice and Experience*, Vol. 21, No. 8, pp. 757- 785, August 1991
- [12] D.L. Eager, E. D. Lazowska. and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems".*IEEE Trans. Software Eng.*, Vol. 12, No. 5, May 1986, pp. 662-675
- [13] D. L. Eager. E. D. Lazowska, and J. Zahorjan, "A comparison of Receiver-Initiated and Sender-Initiated Adaptive Load sharing". *Performance Evaluation*, Vol. 6, No. 1, Mar. 1986, pp. 53-68
- [14] L.E. Flynn and S.F. Hummel, "Scheduling variable length parallel subtasks", *IBM Research Report RC 15492*, Feb. 1990
- [15] W. Gropp and E. Lusk and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface", *The MIT Press*, 1994
- [16] S. F. Hummel, E. S. Lawrence E. Flynn, "FACTORING: A Method for Scheduling Parallel Loops". *Communications of the ACM*, Aug. 1992, Vol. 35, No. 8, pp. 90-101

- [17] P.Krueger and M.Livny, "The Diverse Objectives of Distributed Scheduling Policies", *Proc. Seventh Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order NO801, 1987, pp. 242-249
- [18] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors", *IEEE Trans. Software Eng.*, SE-11, 10 (Oct. 1985)
- [19] Leutenegger S.T., Vernon M.K., "The performance of multiprogrammed multiprocessor scheduling policies", *ACM SIGMETRICS 1990*, pp. 226-236, May 1990.
- [20] D. J.Lilja, "Exploiting the Parallelism Available in Loops", *Computer*, Feb. 1994, pp. 13-26
- [21] M.J. Litzkow, M. Livny, and M.W. Mutka. "Condor - A Hunter of Idle Workstations", *IEEE Int. Conf. Dist. Computing Systems*, 1988, pp. 104-111
- [22] M. Livny and M.Melman. "Load Balancing in Homogeneous Broadcast Distributed Systems", *Proc. ACM Computer Network Performance Symp.*, 1982, pp. 47-55
- [23] Majumdar S., EagerD.L., Bunt R., "Scheduling in multiprogrammed parallel systems", *ACM SIGMETRICS 1988*, pp 104-113, May 1988
- [24] M.M. Mutks, M.Livny, "The Available Capacity of a Privately Owned Workstation Environment", *Performance Evaluation*, Vol. 12, No. 4, pp. 269-284, July 1991
- [25] Ousterhout J. "Scheduling techniques for concurrent systems", *Proc. 3rd International Conference on Distributed Computing Systems*, pp 22-30, October 1982.
- [26] H.Pirahesh, C.Mohan, J.Cheng, T.S.Liu, and P.Selinger, "Parallelism in Relational Database Systems: Architectural Issues and Design Approaches", *Second Int'l symp. Databases in Parallel and Distriubted Systems*, 1990, pp. 4-29

- [27] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", *IEEE Transactions on Computers*, Vol.C-36, No.12, Dec. 1987, pp. 1425-1439
- [28] E.Rosti, E.Smirni, L.W.Dowdy, G.Serazzi, B.M.Carlson, "Robust Partitioning Policies of Multiprocessor Systems", *Performance Evaluation*, Vol. 19, 1994, pp. 141-165
- [29] N. G. Shivaratri, P. Krueger and M. Singhal, "Load Distributing for Locally Distributed System", *Computer*, Dec. 1992, pp. 33-44
- [30] M. Theimer, K. Landtz, D. Cheriton. "Preemptable Remote Execution Facilities for the V System", *Proc. 10th ACM Symposium on Operating System Principles*, Dec. 1985
- [31] T.H. Tzen and L.M. Ni, "Dynamic Loop Scheduling for Shared-Memory Multiprocessors", *1991 Int'l Conf. On Parallel Processing*, pp. 247-250
- [32] Y. Xu and S. Dandamudi, "Performance Evaluation of a Two-level Hierarchical Parallel Database System", *Technical Report TR-97-2*, School of Computer Science, Carleton University, 1997
- [33] R. Elmasri and S.B. Navathe, "Fundamentals of Database Systems", second edition, The Benjamin/Cummings Publishing Company, Inc. 1994
- [34] H. Yu and S. Dandamudi, "Performance Sensitivity of Space-Sharing Processor Scheduling in Distributed-Memory Multicomputers", *12th Int'l Parallel Processing Symposium*, March 1998, Orlando, Florida, pp. 403-409

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