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

Hierarchical Architecture for Parallel Query Processing on Networks of Workstations

By

Boquan Xie, B.Sc(Beijing), M.Sc(Shanghai)

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Science in Information Systems Science

School of Computer Science
Carleton University
Ottawa, Ontario

May 1998

© Copyright
1998, Boquan Xie, B.Sc(Beijing), M.Sc(Shanghai)

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.

0-612-32434-6

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis.

Hierarchical Architecture for Parallel Query Processing on Networks
of Workstations

submitted by

Boquan Xie, B.Sc(Beijing), M.Sc(Shanghai)

Dr. Evangelos Kranakis
(Director, School of Computer Science)

Dr. Sivarama Dandamudi
(Thesis Supervisor)

Carleton University

May 1998

Abstract

Idle CPU cycles of workstations in Network of Workstations (NOW) based systems can be used to process database query operations. Three architectures have been proposed for query processing on NOW-based systems. These are centralized, distributed, and hierarchical architectures. Among the three architectures, the hierarchical architecture provides good load sharing as in the centralized architecture and eliminates the bottleneck problem like the distributed architecture does.

In this thesis, we have implemented the hierarchical architecture by using the public domain software package Parallel Virtual Machine (PVM). Performance of the hierarchical architectures has been evaluated on selected parameters. The experimental results show that the hierarchical architecture is successful in achieving good scale-ups and speedups. Hierarchical system can also handle both light and heavy workload in a load sharing fashion. It has also been found that the performance is sensitive to the minimum fragmentation size (chunk size) for partial operations, and the scheduling structure of queries.

Acknowledgments

I would like to especially thank my supervisor, Dr. Sivarama P. Dandamudi, for his research guidance, and personal encouragement throughout this thesis.

I would like to dedicate this thesis to my family, my husband Xia Zhu and my parents, for their love and encouragement for all these years.

Contents

Abstract	iii
Acknowledgments	iv
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Parallel Database System Architecture	3
1.2 Network of Workstations	6
1.3 Overview of the Thesis	9
2 Related Work	11
2.1 Optimization for Parallelizing Queries	12
2.1.1 Algorithms for Intra-query Parallelism	12
2.1.2 Algorithms for Inter-query Parallelism	15
2.2 Processor Scheduling in Parallel Systems	17
2.2.1 Space-sharing Policies	17
2.2.2 Time-sharing Scheduling	18
2.2.3 Hierarchical policies	19
3 Hierarchical Architecture on Query Processing	22
3.1 Centralized Architecture	22
3.2 Distributed Architecture	23

3.3	Hierarchical Architecture	25
3.3.1	Hierarchical Task Scheduling Policy	26
3.3.2	Query Queue Mapping in Hierarchical Architecture	28
3.3.3	Structures of Local Process and Task Process	30
3.3.4	Main Functions of Local Processes and Task Processes	35
4	Implementation Environment	41
4.1	Parallel Virtual Machine	41
4.2	NOW System Setup	44
4.3	Database Model and Workload Generation	46
5	Performance Evaluation	51
5.1	Performance as a Function of Inter-arrival Time	52
5.2	Speedup of the Hierarchical System	53
5.3	Scale-up of Hierarchical System	57
5.4	Performance Sensitivity to Fragmentation Size	58
5.5	Performance Sensitivity to Query Scheduling Structures	61
6	Conclusions and Future Work	64
6.1	Summary	64
6.2	Contributions of the Thesis	65
6.3	Future Work	66
	References	68

List of Tables

4.1	Selectivities of join operations among the database tables	48
-----	--	----

List of Figures

1.1	Parallel Database System Architectures	4
1.2	Parallel Processing System Architectures	7
3.1	Work Mode Transfer in Hierarchical Architecture with $P=64$ and $B = 2$	29
3.2	Hierarchical Architecture for $P=8$ Processors with Branching Factor $B=2$	31
3.3	Structure of the Local Process	32
3.4	Structure of the Task Process	34
4.1	Network of Workstation System	45
4.2	Balance-scheduled Query and Linear-scheduled Query Created from a Query Type of 4 Tables.	49
5.1	Performance under Different Inter-arrival Time	52
5.2	System Speedup When System is lightly Loaded	54
5.3	System Speedup When System is heavily Loaded	56
5.4	System Scale-up	57
5.5	Performance Under Different Fragmentation Size When System is heav- ily Loaded	60
5.6	Performance Under Different Fragmentation Size When System is lightly Loaded	61
5.7	Performance with Balance-scheduled and Linear-scheduled Queries . .	63

Chapter 1

Introduction

The relational data model [12] has been adopted extensively in the database community. With the increase in database size and the requirement of smaller response time for queries, traditional transaction processing on a single processor is not able to meet the demand.

The time difference between when a query is submitted and when a query is completed is called response time. Two major parts of the response time are: waiting time, which is the time between when a query is submitted and when it actually starts execution, and execution time, which is the actual query execution time. Parallel query processing attempts to reduce the execution time of a query. Relational database queries are suitable for parallel execution because they have uniform operations on uniform streams of data. Three types of query parallelism can be used to improve the performance of database systems: inter-query parallelism, intra-query parallelism, and intra-operation parallelism.

When running multiple queries at the same time, queries can be scheduled to be executed concurrently on different processors. Performance of database systems can be improved by reducing the waiting time of each query. However, in inter-query parallelism, execution time of queries remains the same. In intra-query parallelism, based on the available parallelism among the operations in a single query, concurrent execution of these operations may be performed on several processors. Execution time for a single query can be further reduced by exploiting intra-operation parallelism by partitioning one operation into several partial operations which may be executed concurrently.

The ideal parallel systems expect linear speedup and linear scale-up. If twice as much hardware in a parallel system can perform the same amount of work in half the response time, the parallel system is said to have a linear speedup. Linear scale-up refers to the characteristic that having twice as much hardware, the system can complete twice the amount of work during same time period.

Parallel transaction processing systems have such good features that they are beginning to displace mainframe-used computers. In this chapter, we briefly describe two different approaches proposed for parallel transaction processing: parallel database systems and networks of workstations based systems.

1.1 Parallel Database System Architecture

Parallel database systems are built from conventional processors, memories and disks. Based on how processors, memories and disks are connected to network, the architectures of parallel database systems are divided mainly into three types [11]: shared-disk, shared-memory, and shared-nothing as shown in Figure 1.1. In this figure, p stands for processors, m represents for memory modules, and d refers to hard disks.

In the shared-disk architecture, also called data sharing architecture, each processor has its own, private memory and all the disks containing the databases are shared among different processors. Each processor, may access and update the contents of the database. Parallel database systems such as Digital's VAXcluster and IBM's Sysplex are based on this type of architecture. These systems use the shared-disk architecture for inter-query parallelism rather than intra-query parallelism.

In the shared-memory architecture, all the processors can access global shared memory and all the disks through the interconnection network. This architecture is also called shared-everything [22]. Multiprocessor IBM/370, Vax and Sequent computers are typical examples of shared-memory design.

In the shared-nothing architecture, each processor has a part of the database and only that part of data can be directly accessed by that processor. In addition, each processor has its own memory and processors communicate with each other via a high speed interconnected network. Teradata, Tandem, nCUBE, and newer VAXclusters are based on the shared-nothing architecture.

Shared-nothing

Shared-memory

Shared-disk

Figure 1.1: Parallel Database System Architectures

Shared-disk architecture with large numbers of processors and disks works fine for large read only databases and for databases where there is no concurrent sharing. However, a processor executing a query that updates the database, has to inform other processors about its intention first. After the request is acknowledged by the other processors, it can obtain the copy of that data and perform the update. This not only creates heavy traffic on the shared interconnection network, but also greatly increases the response time due to the shared resources. The interconnection network plays an important role in determining the performance of the system. It is also difficult to scale this architecture to thousands of processors and disks.

The interconnection network is also a major problem in shared-memory architecture. The network must have the bandwidth of the sum of all the processors and disks and it is hard to scale to thousands of processors. Shared-memory designed systems can be used within a node of a shared-disk or shared-nothing system [11].

Shared-nothing architecture is suitable for large parallel database systems. As processors can access its own memory and disks directly, shared-nothing architecture reduces resources sharing and interconnection network does not carry large amount of data but only requests and responses. This architecture can achieve near-linear speedup and scale-up on complex relational queries [22]. Data in shared-nothing databases are partitioned among the disks connected directly to each processor. Usually relations are partitioned horizontally into sets of tuples and these sets are distributed to disks in the system. The horizontal partitioning strategies include range

partitioning, round-robin partitioning and hashing partitioning [11]. The major problem in shared-nothing database systems is how to partition data as non-uniform distribution of data may cause load imbalance among the processors and thus affect the performance of the whole system.

1.2 Network of Workstations

Parallel processing systems can be divided into shared-memory systems (also called multiprocessor systems), and distributed-memory systems (also called multicomputer systems) as shown in Figure 1.2. Processors in multiprocessor systems communicate with each other by the shared memory variables. In multicomputer systems, nodes communicate with each other by message passing via the inter-connection network. Some systems may equip each computer with local hard disk while other systems may have disk-less computers file-served from one or more server nodes. One special case of a multicomputer system is Network of Workstations(NOW).

Commercial parallel systems are very expensive and small organizations usually could not afford. On the other hand, it is found that up to 80% of the processing power available on a network of workstation is wasted by idling [20]. Substantial performance can be obtained by making use of the idle processing cycles based on the availability of public domain software packages such as PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) which allows a heterogeneous network of computers to appear as a single concurrent computational resource. It is proposed

A shared-memory multiprocessor system

A distributed-memory multicomputer system

Figure 1.2: Parallel Processing System Architectures

by Dandamudi [6] that database query operations can be processed by using networks of workstations. Analysis of a four-way join operation is also presented in [6] to demonstrate the feasibility of this proposal.

Because each workstation in a NOW is under an independent control, it is not possible to distribute the database across the network. Therefore, relational tables are all located in the central database and NOW can be used to speedup query processing which requires read-only access to data in the central database. Since each workstation does not have direct access to data, before executing the query, each workstation has to communicate with the central database and obtain the relation tables through the communication network. After the query is finished, the result table may have to be collected from several workstations. All these activities cause traffic on the network. However, for complex queries, this communication overhead should be compensated for by the reduction in execution time.

There are several differences between NOW used for query processing and parallel database systems. First, workload on processors in parallel database systems are stable since processors are dedicated to parallel query processing. On the other hand, processors in the NOW architecture could possibly have variable background workload. Second, although the NOW architecture is similar to the shared-nothing parallel database system, database is not distributed among the processors in NOW architecture as it is the case in shared-nothing architecture. Third, NOW-based systems use the general purpose LANs, while in parallel database systems, the network

is dedicated. special purpose fast communication network.

NOW-based systems need to introduce several performance problems. As each workstation can receive jobs locally, this may cause load imbalance if there is no central scheduler and central load balancer. Job migration and fault tolerance are also important in NOW-based systems. When the load on one workstation becomes heavy, the queries running on that machine can either be killed or migrated to another machine. Fault tolerance is definitely important if one of the workstations suddenly fails or is turned off. Moreover, communication network used in NOW-based systems is much slower compared to the inter-connection network in parallel database systems. Faster networks improve performance significantly since send/rcv messages are the only ways processors communicate with each other and obtain the data from the database node.

1.3 Overview of the Thesis

This thesis is concerned with parallel query processing on a NOW. The main focus of the thesis is on the performance evaluation at the hierarchical architectures for parallel query processing.

Chapter 2 reviews the previous work on the algorithms for parallel query processing and scheduling policies used in parallel systems. Chapter 3 presents three architectures of a NOW-based system for parallel query processing and describes in detail the hierarchical policy and its implementation. The experimental environment

is described in Chapter 4, including the Parallel Virtual Machine (PVM) software package [14]. This chapter also discusses the database and query workload models. Chapter 5 presents the main experiments and the experimental results. Conclusions and future work are given in Chapter 6.

Chapter 2

Related Work

Query processing algorithms can be divided into several types depending on whether the optimization is static or dynamic; whether replication is considered or not; what are the complexities of the algorithms and so on. Since relational operations are set-oriented, there is a lot of flexibility for optimizing queries for parallel execution. Parallel query processing in distributed database systems has been studied extensively. Among the proposed algorithms, processing of distributed queries usually consists two steps: execution sequence optimization and workload scheduling among the resources available in the system. Previous work on processor scheduling in parallel systems is also relevant to our study. This chapter reviews the related work on parallel query processing in database systems and on processor scheduling in parallel systems.

2.1 Optimization for Parallelizing Queries

Among the relational operations, join operation is complex and its optimization usually includes the optimizations of other operations in relational database queries, such as project, select and so on [18]. Because natural join operation is one of the most time-consuming and data-intensive operations in relational database query processing, this thesis focuses on parallel processing of queries consisting of natural join operations. As intra-operation parallelism is less complex than intra-query and inter-query parallelism, most of the studies on parallel query processing focused on the intra-query and inter-query parallelism. We address these issues separately in the following sections.

2.1.1 Algorithms for Intra-query Parallelism

It has been observed that different execution sequences in joins in a query will result in different execution costs and the execution time of one join operation in the query strongly depends on the allocation of the number of processors in the systems. Thus, there are two issues in exploiting intra-query parallelism: 1) determining the execution sequence of the join operations in the query, and 2) determining the number of processors that should be allocated for each join operation in the query according to its size and the order of other join operations so that the total response time can be reduced [4].

For join sequence scheduling, two kinds of join sequences are investigated by Chen

and Yu [4] [3]: sequential join sequence and general join sequence. A sequential join sequence refers to a join sequence in which the resulting relation of an intermediate join can only be used in the next join. A join sequence where the resulting relation is not required to be only used in the next join is termed general join sequence. Several algorithms for generating general join sequences are proposed and evaluated to determine the efficiency of general join sequence. It has been found that the results for general join sequences is significantly better than that for optimal sequential join sequence, especially for complex queries. [3].

For processor allocation, execution dependency and system fragmentation are shown to be important for intra-query parallelism. Several processor allocation algorithms of bottom-up and top-down approaches have been proposed and evaluated in [4][5]. Among all the algorithms evaluated, Chen and Yu concluded that a two step algorithm is the solution to minimize the query execution time. This algorithm works as follows: first build a bushy tree to minimize the total amount of work, required as if in a single processor, then allocate processors to the internal node of the bushy tree in a top-down approach with the concept of synchronous execution time.

Tan and Lu [26] classified the above approach as two phase optimization: in the first phase, a join ordering for a query is generated without considering the runtime resources, and the second phase ensures that the resources are optimally scheduled to minimize the completion time of the operator tree. Tan and Lu argued that they can first transform the operator tree to task tree. Each node in task tree represents a task

to be scheduled, and arcs represent the order in which tasks have to be completed. Shelves are introduced in [26]. A shelf contains one or more tasks and the task in one shelf are to be executed concurrently by all the processors, thereby exploiting inter-operation parallelism. Beginning with a fixed number of shelves where the number is determined by the number of nodes along the longest path from leaf nodes and root node, each shelf is assigned a task first. The number of shelves is extended by adding tasks into them according to different algorithms, and the degree of intra-query parallelism can be seen after all the tasks are assigned into shelves. The strategy of keeping minimal number of shelves exploits intra-query parallelism to the full set while the algorithm works fine by introducing intelligence and flexibility in assigning the remaining tasks. Algorithms based on page connectivity graphs were used by Murthy and Rotem [19] to schedule join execution in a shared memory multiprocessor environment. First minimum number of time steps with an unbounded number of processor was calculated. According to the upper and lower bounds on the number of processors required to complete join processing in such amount time, page reading schedules were developed to minimize the number of processors and total duration of join processing. However, while the processing time was guaranteed to be minimized, the utilization of the processors may be as low as 50% [19].

All the above query processing algorithms optimize only a single query without considering other queries that are running concurrently in the system. Thus the algorithms for intra-query parallelism may not be applicable for inter-query parallelism

in a multi-user, multi-query environment, and it is necessary to consider intra-query parallelism together with inter-query parallelism.

2.1.2 Algorithms for Inter-query Parallelism

Scheduling in multi-query environment requires the determination of both processor allocation and temporal processor assignment. Wolf and Chen [28] extended Chen and Yu's work [4] to parallel multi-query scheduling. Several algorithms have been proposed to handle both intra-query parallelism and inter-query parallelism. One of the algorithms which handles the intra-query scheduling problem (with precedence) in one phase, and the inter-query scheduling (without precedence) in the other seems to have better performance than the others. In the first phase, for each possible number of processors that might be allocated to the overall query, a processor allocation for each individual query phase and an estimate of the total execution time are generated. In the second phase, the results of the first phase are used to generate an overall schedule of the full set of queries. The first phase can be done at compile time, and only the second phase has to be done at run time.

For the same database, the same operations appear quite often in the queries, and many queries will share a common set of relations. If the common operations or relations can be scheduled properly, the number of times that the common operations being processed can be reduced. It is argued in [27] that performance improvement can be achieved by grouping a set of queries from the same workload. Because of the

availability of large main memory, it is possible for several relations to fit in memory and a set of operation can be performed. Tan and Lu [27] described how a group of queries can be scheduled based on the sharing of relations and common intermediate results. First, the set of queries are optimized and a query execution plan is generated for each query. Second, a sharing graph is created to reflect data sharing between the query plans. Then, the sharing graph can be transformed to a multi-query plan. It has been shown in [27] that the algorithms proposed can provide significant savings over the non-sharing strategy.

Hypercubed multicomputer systems have also been studied for database applications, such as data redistribution and nested-loop join algorithm [1], comparison of different join algorithms [2], and so on. Frieder and Baru [13] presented an architecture of a run time system that incorporates site allocation and query scheduling algorithms in a distributed memory multicomputer system. Specific sites are not reserved in advance when a query is activated. Since it is difficult to compute the exact completion times of each operation, the site allocation policy dynamically chooses the nearest available site in terms of communication distance. When executing multiple queries, queries are selected for execution only if their resource requirements is less or equal to resources available. Frieder and Baru pointed out in [13] that the dynamic site allocation and query scheduling policies are applicable to a wide range of distributed-memory multicomputer systems, although their effects were studied by simulating data sets and queries in a hypercube multicomputer system,

2.2 Processor Scheduling in Parallel Systems

Scheduling algorithms in distributed-memory multicomputer systems, such as Network of Workstations, can be mainly divided into space-sharing and time-sharing. Space-sharing policies partition the processors in the system into disjoint parts and each part, which may include several processors, is allocated exclusively for one job. Only after the job has finished, the processors can be released for further allocation. In time-sharing policies, processors can run several jobs at the same time and the jobs share processors according to a particular policy such as round robin processes, round robin jobs and so on. In this section we will talk about space-sharing, time-sharing policies and policies combining space- and time-sharing.

2.2.1 Space-sharing Policies

In space-sharing, the main issues are determination of the number of partitions and the corresponding partition sizes. This can be decided statically, dynamically, and adaptively according to the systems and workload characteristics.

Static strategies assume that the number of processors allocated for one job is a predetermined system configuration parameter which does not change with time. Adaptive strategies compute a job's partition size according to various workloads when the job is scheduled. The partition size for each job will not change during the execution of the job. Dynamic scheduling allows the number of processors assigned to a job to change during the execution of the job. Static scheduling is simple and

has low overhead but the utilization of processors is low as it may leave processors idle when work is available. The overhead of adaptive scheduling is much less than that of dynamic scheduling but flexibility of the system using dynamic scheduling is better than that of a system using adaptive scheduling.

Rosti *et al.* studied the problems involving the partitioning of processors among two or more applications [23]. Several static scheduling policies including Fixed Processors Per Job (PPJ), Equal Partition with a Maximum (EPM), and Insurance Policies (IP), are compared with adaptive scheduling. Each adaptive policy was analyzed over a range of workload types and possible arrival rates. The comparison shows that there is trade-off between system performance and the amount of known information. If the workload and arrival rate are known, best policy can be determined. However, it may perform poorly for other workload and arrival rates. Sevcik [25] also pointed out that information about the characteristics of individual applications can be used to achieve effective scheduling. For large-scale parallel systems, more information is necessary to determine the best scheduling policy.

2.2.2 Time-sharing Scheduling

Time-sharing scheduling algorithms have been studied extensively. These policies are based on the total work including number of jobs and number of processes in each job. Majumdar *et al* [17] characterized an application by its total amount of work and the number of processes in the application. Scheduling policies such as Smallest

Number of Processes First (SNPF) and Smallest Cumulative Demand First (SCDF) have been proposed with both non-preemptive and preemptive versions. It has been shown in [17] that the shortest job first class performs better than First Come First Serve (FCFS) and Round Robin (RR).

In addition to shortest job first class and FCFS, Leutenegger and Vernon [16] evaluated co-scheduling, RRjob and RRprocess using a similar characterization of applications. Ousterhout proposed co-scheduling [21] so that groups of cooperating processes are all assigned processors at the same time and they can be executed without delay. RRjob is a round robin policy that allocates an equal fraction of the machines power to each job in the system [16] and RRprocess is a similar round robin policy that allocates an equal fraction of the processing power to each process in the system. Simulation result of mean response time shows that the general performance ordering of the policies from best to worst is: RRjob, RRprocess, co-scheduling, SNPF, FCFS. Under heavier workload and variation of the number of processes, it has been concluded that scheduling policies that allocate processing power equally per job are the best candidates for future study and implementation.

2.2.3 Hierarchical policies

Another class of scheduling policies was proposed by Setia and Squillante [24] in attempt to increase processor utilization and system throughput by timesharing a partition of processors among a group of jobs. By overlapping the communication

and waiting time of a process with computation time of other processes, it has been shown that the combination of space-sharing and time-sharing scheduling policy can provide significant performance improvements, especially from moderate to heavy workload [9].

However, all the scheduling policies mentioned above require a central scheduler, which maintains a central run queue, to load balance the systems. Due to mutually exclusive access, the central run queue causes serious contention problem as the number of the processors increases. To reduce run queue contention in the systems, Dandamudi and his co-workers proposed hierarchical task queue organization for shared-memory multiprocessor systems [5] [7] and hierarchical processor scheduling policy for shared memory systems [10] and distributed memory systems [9]. The objective of hierarchical policy is to inherit good features of space sharing and time sharing policies and to eliminate queue access contention by distributing scheduling functions among processors.

Hierarchical scheduling policy is a self-scheduling policy using a hierarchical run queue organization [10] [9]. There is no central scheduler making processor allocation decisions and processors look for work whenever they are idle. The workload used in these studies is the fork-and-join type of job structure which is similar to workloads in [16] [17]. Simulation results show that hierarchical policy provides superior performance than purely time sharing and purely space sharing policies, especially at moderate to heavy workloads. Only at low system loads at which large scale systems

are unlikely to operate. equal partition time sharing policy performs better than hierarchical policy. It is also found that hierarchical scheduling policy is less sensitive to systems overheads and the variance in job service demands [9].

From the discussion above, it can be concluded that there are a number of scheduling policies in parallel system among which the hierarchical scheduling policy provides substantial improvements over the others for all practical purposes. On the other hand, in distributed database systems, although different algorithms of parallel query processing were proposed to realize inter-query, intra-query which includes inter-operation and intra-operation parallelism. none of these works considered the application of scheduling algorithms in parallel systems to the implementation of the three kinds of parallelism in parallel query processing in database systems. The objective of this thesis is to apply hierarchical run queue organization to parallel query processing on Networks of Workstations and evaluate its performance.

Chapter 3

Hierarchical Architecture on Query Processing

Network of Workstations can be used to speed up query processing, which requires read-only access to the data in databases. There are three kinds of architectures for parallel query processing in Network of Workstations [8]. In this chapter, we introduce each of these three architecture and discuss their advantages and disadvantages. It is assumed in the study that any input of queries has been optimized by query optimizer, and we focus more on the workload scheduling for parallel query processing in hierarchical architecture.

3.1 Centralized Architecture

In centralized architecture, one of the processors (nodes) maintains the only task run queue in the system and pre-optimized queries are submitted from various workstations to this node. As a central workload scheduler in the system, it keeps the state

information, including non-query load (background load) and current query load, of all processors. If the state of a node changes, the node reports the change to the central node. Depending on the workload information in the system, the central node decides the processors that will be allocated to the jobs. The processor scheduling policies are dynamic and can change during the lifetime of a query.

Since the central node is in charge of scheduling, workload in the whole system is well shared by the available processors in the system. The centralized architecture is suitable for small systems and it provides good performance mainly due to its load sharing characteristic. As the number of processors in the system increases, the central scheduler may become a bottleneck. Several techniques were found to reduce the frequency of run queue access, such as assigning a set of tasks from queue instead of a single task each time, removing tasks for several processors at one time, and using multiple run queues. However, access contention remains a problem for the centralized architecture and it is only suitable for small sized systems.

3.2 Distributed Architecture

In distributed architecture, each workstation has its own query queue and the query queue receives the workload generated from the local workstation together with the queries transferred from other workstations. Each workstation also has the same scheduler. Depending on the transfer policy and the information the workstation receives

from the other workstations in the system. the scheduler will schedule the next operation to be performed by itself. or transfers that operation to other workstations (in this case. it determines which workstations). The queries transfered from other workstations have to be executed locally.

In a Network of Workstation system. the indicator of system load is considered as a weighted combination of the background and query loads [8]. The workstations can be classified as under-loaded (*i.e.* in a receiver state), overloaded (*i.e.* in a sender state.) or normally loaded (*i.e.* in a OK state). The scheduling policies for distributed architecture include sender-initiative. receiver-initiative and adaptive policies. For the sender-initiative policy, overloaded workstations look for under-loaded workstations and transfer work to them. If no workstation is under-loaded, it has to do the work itself. In the receiver-initiative policy, it is the opposite. The under-loaded workstations receive jobs from overloaded workstations. It has been shown that sender-initiative policy is better at low to moderate workload while receiver-initiative policy is more suitable at moderate to heavy workload. The adaptive policy behaves as a sender-initiative policy when system is at low to moderate workload and as a receiver-initiative policy at moderate to high workload in the system [8].

In the distributed architecture, access contention is avoided. However, there are several performance problems due to the distributed schedulers in systems. Because workload information of other workstations is collected periodically, the information

may not be up to date. It causes additional communication overhead since the workstations need to exchange state information and transfer jobs among them. Load imbalance is always a problem in the distributed architecture due to the lack of information about all queries and incomplete workload information of the workstations.

3.3 Hierarchical Architecture

The hierarchical architecture is proposed not only to provide good load sharing as in the centralized architecture but also to eliminate the bottleneck problem like the distributed architecture does [8]. The hierarchical architecture is based on the hierarchical run queue and hierarchical scheduling policy proposed by Dandamudi [5] [9] [7] [10].

In the hierarchical architecture, each of the processors (workstations/nodes) in the system maintains a local job queue, and some of the processors also maintain a job queue that is part of the hierarchy. Local job queues and job queues are mapped as a tree structure with all the local job queues acting as leaf nodes in the tree of job queues. Local job queues and job queues maintain queries, operations of the queries, and partial operations. The processes that maintain local job queues are called local processes and the processes that keep job queues are called task processes. Besides local process and task process, each processor maintains also another kind of process called workload generator. Each workload generator simply produces query workload and sends them to the root job queue in the mapped tree system.

In this section, we will first describe the hierarchical task scheduling policy in the

hierarchical architecture. Then, the mapping of job queues in hierarchical architecture and the structures of local processes and task processes are introduced. Finally a detailed description of the main functions of the local processes and task processes is presented.

3.3.1 Hierarchical Task Scheduling Policy

Hierarchical architecture is a self-scheduling scheme. The local processes and task processes running on the workstations actually perform self-scheduling by exchanging information through the network.

Queries are generated by workload generators on each workstation and they are sent to the root process in the system. The scheduling policy in the hierarchical architecture has the characteristic of bottom up, demand-driven scheduling. Whenever a local process is idle, it checks its local job queue to see if there are jobs waiting. If not, it sends a message to its own parent job queue in the tree and asks for jobs. If the parent job queue has jobs in the queue, the task process that maintains the parent job queue sends certain number of jobs to the child queue. Otherwise the parent task process sends message to its parent job queue up the tree requesting jobs.

When there is no job in the job queues in the tree, the process which keeps the root job queue will be idle until the arrival of queries. This root process inserts the new queries into the job queue and sends some queries or part of a query to its child queues which request jobs, if there are any. The processes that requested jobs are

invoked when their parent queues send new jobs to them.

The hierarchical policy operates in three modes: query transfer mode, operation transfer mode, and partial operation transfer mode. At the level of the root queue in the tree, jobs come as queries. As queries consist of operations, they need to be changed into operations when the number of queries in job queue is less than branching factor B . At this time, the transfer mode is changed from query transfer mode to operation transfer mode. Information tables are created to keep track of the execution of operations. Dependent operations are disabled until the operations on which they depend on have been completed.

When the number of operations in job queues becomes no greater than branching factor B , transfer mode changes from operation transfer mode to partial operation transfer mode. The number of the partial operations created is based on the size of this task and the minimum fragmentation factor. Minimum fragmentation size is the minimum size of the tables required in partial operations. It determines how many partial operations can be created for one operation. Information tables are also created to keep a record of the completion of these partial operations.

In order to explain the working of the policy more clearly, we present an example of a system with 64 processors and a branching factor B of 2 (Figure 3.1). Assume that there are 4 queries in the root job queue and all other job queues are empty. Processor 1 is idle and it looks for jobs. Since there is no job in the local job queue of processor 1, it requests its parent queue Q32 for jobs. Q32 asks Q16 for jobs, and

so on. Because the number of queries in Q1 is greater than the branching factor B, the work transfer mode stays at query transfer mode and Q1 moves $1/B$ of its jobs (*i.e.* 2 queries, q1 and q2) to Q2. Q2 moves $1/B$ of the 2 queries (*i.e.* 1 query, q1) to Q4 still using the query transfer mode. The number of jobs in Q4 now is less than branching factor B. Thus the work transfer mode changes to operation transfer mode. If we assume that q1 has 4 operations, Q4 moves 2 of q1's operations to its child Q8, and Q16 gets 1 operation from Q8. Work transfer mode changes to partial operation mode as there are less than 2 operations in Q16. Q16 partitions the only operation it has and sends $1/B$ (*i.e.* 50%) of the operation to child Q32. Processor 1 gets only 25% of 1 operation in one query. Other jobs stay as high as possible in the tree of job queues and wait for any processors looking for jobs.

In terms of the mode transfer, hierarchical policy realizes inter-query parallelism, inter-operation parallelism as well as intra-operation parallelism. There is strong indication that hierarchical architecture will perform well [9]. In this thesis, we evaluate the performance of the hierarchical architecture for paralleling query processing. In the following sections, the details of the implementation of the hierarchical architecture are described.

3.3.2 Query Queue Mapping in Hierarchical Architecture

Assume we have $P=8$ processors available with a tree branching factor of $B=2$, the height of the balanced tree is 3 and the number of non-leaf nodes is 7. Thus, 7 out of

Figure 3.1: Work Mode Transfer in Hierarchical Architecture with $P=64$ and $B=2$

the 8 processors maintain one job queue in addition to a local job queue. From Figure 3.2, we have 15 queues from Q1 to Q15. Local job queue Q8 to Q15 are maintained by processor number 1-8. The other 7 job queues are maintained among the processors and in our case, Q1-Q7 are maintained by processor number 1-7 in order because we use an Ethernet-based NOW system.

In our implementation, local job queue and job queue are kept in the local process and task process running on the processors, respectively. Therefore, in this system, we have 7 processors running one local process plus one task process and 1 processor running only one local process. The local job queues and job queues are manipulated as if they have the tree structure as shown in Figure 3.2.

3.3.3 Structures of Local Process and Task Process

In this section we introduce the structures of local process and task process and explain their most important functions in query processing in the hierarchical architecture.

The main function of a local process is to receive tasks from its parent process, and perform the real processing, for example, join operation. Inside a local process, as shown in Figure 3.3, there are a local job queue, a job manipulator, a job processing center, and several job execution information tables.

The local job queue receives jobs from its parent process. The job manipulator checks the jobs in the local job queue. If there is no jobs in the queue, it asks its

Figure 3.2: Hierarchical Architecture for $P=8$ Processors with Branching Factor $B=2$

Figure 3.3: Structure of the Local Process

parent process for jobs. If there are jobs, job manipulator that is in charge of process determines whether transfer mode changes are needed. The job processing center obtains source data either from other processes or from hard disk and perform the real join processing. The result data is written back to the disk. The information tables keep track of the execution of queries and operations. Once a query or operation is finished, the information table of this query or operation is updated. If other dependent operations at higher query level become executable because of this update, the corresponding task in the local query queue will be enabled.

Task processes manage job transferring from parent queues to child queues down to local job queues and one task process consists of a job queue, a job manipulator, several job execution information tables, and a child access queue (Figure 3.4). The job queue in a task process receives jobs from its parent process if it is an internal job queue in the tree or from workload generators if it is the root job queue in the system. The function of job manipulator and information tables are the same as, the ones in the local process. The process IDs of the child processes are added to the child access queue in the task process whenever the child processes request for jobs. Depending on the available jobs in job queue, the task process transfers certain number of jobs to the child processes whose process IDs appear in the child access queue.

Figure 3.4: Structure of the Task Process

3.3.4 Main Functions of Local Processes and Task Processes

In the following, we present in terms of pseudo-code a summary of the common responsibilities and different functions to exchange information between local processes and task processes.

Local processes and task processes have the same mode transfer control, which includes query to operations mode transfer and operation to partial operation mode transfer. Both processes also maintain the same task execution information that updates the information tables when operation or partial operations are finished. The pseudo-code for the four functions, when the process receives messages of query to operations transfer, operations to partial operations transfer, operation finished, and partial operation finished, are described as the following:

- Query To Operations Transfer:

1. When it is time to change the transfer mode from query to operation, the program decomposes the first query in job queue into operations and puts them back to the job queue;
2. It then creates job execution information table for the query. The execution table contains the execution information about each operation of the query.

- Operation To Partial Operations Transfer:

1. When it is necessary to change the transfer mode from operation to partial operation, the program decomposes the first operation in the job queue into

partial operations depending on the minimum fragmentation size specified.

The partial operations are inserted back into the job queue:

2. Then it creates job execution information table for the original operation.

The execution table contains the execution information about each partial operation of the original operation.

- Operation Finished:

1. When the process gets an operation finished message, check is made to see if the query to which the operation belongs is finished. If yes, it notifies the root task process, calculates experimental results and removes the information table for the query;
2. Otherwise, it updates the information table of this operation. If the parent operation is thus executable, the corresponding operation is enabled in the job queue.

- Partial Operation Finished:

1. When the process receives the partial operation finished message, it checks if the whole operation is completed. If so, it collects the results of the operation, and send operation finished message to the process which keeps the information table of the query. Finally it removes the information tables for the operation.

2. Otherwise, it updates the information table for the operation.

For task processes, the most important functions are looking for jobs when they receive messages asking for jobs from child processes, dividing jobs when there are jobs in the job queues, and inserting jobs if their parent sends them jobs:

- Looking for Jobs:

1. If there is no job in its job queue, its parent job queue is notified that it needs jobs by sending its process ID.
2. If there are jobs, it executes Dividing Jobs.

- Dividing Jobs:

The following is executed in a loop while the access queue of task process is not empty.

1. If there is no executable jobs (operation or partial operations):
 - (a) If there are queries and the number of queries is less than the branching factor B , execute Queries To Operations.
 - i. If the number of executable operations is not less than branching factor B , move $1/B$ of the executable operation down to the child process.
 - ii. Otherwise, invoke Operations To Partial Operations. After that, move $1/B$ of the executable jobs to child process operations if the

number of executable jobs is greater or equal to B , or move down the number of jobs equal to the number of local processes the child process is above.

(b) If the number of queries is not less than B , move down $1/B$ of the queries or the number of queries equal to the number of local processes the child process is above, whichever is less.

(c) Otherwise, do not move any jobs down.

2. If there are executable operations or partial operations.

(a) If the number of executable jobs is greater than or equal to B , move down $1/B$ of the executable jobs.

(b) Otherwise,

i. If there are executable operations, invoke Operation To Partial Operations;

ii. Else if there are queries, execute Query To Operations.

iii. After the above procedures, if the number of executable jobs is not less than B , move down $1/B$ of the executable jobs. Otherwise, move all the executable jobs down or move jobs equal to the local processes the child process is above.

- Inserting Jobs to Queue

Whenever a task process receives a job, after inserting it into the job queue, the

process invokes Dividing Jobs.

Local processes have slightly different functions of looking for jobs and inserting jobs into job queue. Local processes start looking for jobs when the processor becomes idle, and they perform the real job processing when there is jobs waiting in the job queue:

- Looking for Jobs:

1. If there is no job in its job queue, notify its parent task queue that it needs jobs by sending its process ID.
2. If there are jobs, execute Job Processing.

- Job Processing:

1. If the first job in job queue is a query, it is changed into operations by using Query To Operation and repeat executing Job Processing;
2. If the first job in job queue is an executable operation or a partial operation, after obtaining the necessary tables this job needs from the processes that maintain those tables, the join operation is performed. When the join is done, the process who keeps the execution information table for this job that it has been finished is notified.

- Inserting Job To Queue

Whenever a local process receives a job, after inserting it into the job queue,

the process invokes Job Processing.

Chapter 4

Implementation Environment

Parallel Virtual Machine (PVM) and Network of Workstations (NOW) are used in the thesis to examine the performance of parallel query processing. This chapter describes the software package PVM and the experimental NOW system used. The database and workload models are also presented in detail.

4.1 Parallel Virtual Machine

PVM is a software package for developing parallel programs executable on networked computers. The development of PVM started in 1989 by Vaidy Sunderam and Al Geist in Oak Ridge National Laboratory, and PVM 3.0 was released in 1993.

PVM uses message passing model to allow programmers to exploit distributed computing across a wide variety of computer types. The types of computers can include 386/486 PC class machines, high performance workstations, shared memory multiprocessors, vector supercomputers, and large massively parallel processors. An

important feature of PVM is that it makes the collection of different types of computers appear as one virtual machine with a large computing resource. More details on PVM can be found in [14].

PVM consists mainly of two parts. The first part is called PVM daemon, `pvmd3` or `pvmd`. One `pvmd` runs on each host of a virtual machine. `Pvmd` can be used by different users at the same time and the daemon owned by one user does not interact with those owned by other users. A virtual machine is created by starting a `pvmd`. Then PVM applications can be started from any of the host of the virtual machine.

Another part of PVM is `libpvm`, a library of PVM interface routines that provides user-callable functions for message passing, spawning processes, coordinating tasks and modifying the virtual machine. The PVM interface library is written in C and it currently supports C, C++ and Fortran languages.

A task in PVM is defined as a unit of computation similar to a UNIX process. Users write PVM application programs as a collection of cooperating tasks. One of the tasks, which is called master, is invoked under Unix prompt and other tasks, which are called slaves, are spawned by the master. Tasks access PVM resources through `libpvm`.

The `pvmd` and `libpvm` manage message buffers which potentially hold large amount of dynamic data and PVM messages are composed without declaring a maximum length ahead of time. As data formats on different computers are incompatible, the

pack and unpack functions provide the choice of XDR encoding. PVM message-passing primitives provide not only buffering and transmission for sending and receiving data structures, but also high level primitives such as broadcast, barrier synchronization and global sum.

In terms of resource management, PVM is dynamic in nature. Computer resources, i.e. hosts can be added or deleted either from a system console or from within the user's application. In other words, at any point in the execution of a concurrent application, any task may add or delete computers from virtual machine. Allowing application to interact with and manipulate their computing environment provides a powerful paradigm for load balancing, task migration [15].

Besides scalability and heterogeneity, PVM has two other important features: fault tolerance and portability. For portability, operating system and language features, such as multi-threaded processes and asynchronous I/O, are avoided because it would be hard to retrofit if unavailable. PVM programs can be copied to different architectures, compiled and executed without modification.

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

PVM has been well accepted by global computing community and used successfully for solving large-scale problems. Our implementation programs for performance evaluation of parallel query processing in NOW system are written in C++ for the PVM environment.

4.2 NOW System Setup

The NOW system used in this thesis is composed of a number of Pentium processors running LINUX operating system and linked with 10Mb/s and 100Mb/s networks. The current configuration of the system consists of 6 Pentium and 4 Pentium Pro nodes as shown in Figure 4.2. All nodes have 32 MBytes of RAM. Nodes 1, 2, and 9 are 133 MHz Pentiums, and node 4 is also a 133 MHz Pentium but with a pipeline burst cache which makes it quite faster than nodes 1, 2, and 9. Nodes 5 and 6 are 166 MHz Pentiums while nodes 0, 3, 7 and 8 are 200 MHz Pentium Pros.

The main node number 0 has a hard disk and is the only node that can be accessed from the outside of the system. To an outside user, the system looks like a set of normal workstations. The ten nodes are interconnected with each other by a 10Mbps Ethernet network and a 100Mbps Ethernet high-speed network. The 10Mbps Ethernet is used for booting the system, providing the basic communication and remote file service through Network File Service (NFS). Both 10Mbps and 100Mbps Ethernet can be used for experiments under PVM environment.

Node allocation in the system allows users to share the system and have as little

Figure 4.1: Network of Workstation System

impact on each other as possible. Once the nodes are allocated to one user, other users can not use those nodes until that user releases the nodes. This provides the users with relatively stable background workload to get more accurate experimental results. The system also supports node sharing between users when it is not necessary to use exclusive node allocation. Node 0 is shared by all users and is only used as a database node that reads from database and sending data to the processor nodes in the experiments.

4.3 Database Model and Workload Generation

The database we used in the implementation contains sixteen original tables. Each table is represented as a file under UNIX system and each file contains 5000 to 9000 records of data. Each record in the tables has two fields of data with the first field of data acting as the key field for the relational database tables and the other field does not have any real meaning ("filler" field). Both fields are represented by long integers so that the tuple size of the original database tables are all 8 bytes.

There are two kinds of key fields for the tables in the database model. In two of the sixteen tables, the keys are unique in the tables. There is no repetition of the key values in the table and the values of the keys are sequential long integers. For the remaining fourteen tables in the database model, the key values in each table are normally distributed with the same mean value and different variance for the keys in each table. By having these two kinds of tables, we intend to obtain different

selectivities for the join operations between different tables. In our experiments, the selectivity of natural join between table 1 and table 2 is defined as the number of tuples in the result table divided by the multiplication of the number of tuples in table 1 and the number of tuples in table 2 [12].

Table 4.1 shows the selectivities of join operations among the 16 tables in the database. The first row and first column are the numbers 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%. Tables 2 and 9 have unique key values and the keys in the other tables are 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.

Selectivities are used to determine the query types in the database model. 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. For each query type, the number of original tables in the database is either 2, 4, or 8. Sixteen query types are created according to the different selectivities of the join operations among the data

	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	0.7	3.4	0.7	0.6	0.7	0.7	0.7	0.6	0.6	0.7	0.7
7	1.6	1.5	1.3	0.5	1.3	1.4	0.7	4.7	1.1	1.1	1.5	1.6	0.8	1.0	1.4	1.4
8	1.2	1.2	1.1	0.5	1.1	1.2	0.6	1.1	5.0	1.0	1.2	1.3	0.7	0.9	1.1	1.2
9	1.1	1.1	2.2	0.5	1.0	1.1	0.7	1.1	1.0	2.2	1.1	1.1	0.7	0.9	1.1	1.1
10	1.7	4.9	1.4	0.5	1.4	1.5	0.7	1.5	1.2	1.1	4.9	1.7	0.9	1.1	1.5	1.5
11	2.0	1.7	1.4	0.6	1.5	1.7	0.7	1.6	1.3	1.1	1.7	4.8	0.9	1.1	1.6	1.7
12	0.9	0.8	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 4.1: Selectivities of join operations among the database tables

tables in the database model. In order to model queries with growing-sized result tables, in half of the query types, tables of join operations with higher selectivities are selected. In the other half of the query types, tables whose join operations have lower selectivities are selected to produce shrinking-sized result tables.

Balance-scheduled queries are most suitable for parallel execution and they can exploit inter-query, inter-operation and also intra-operation parallelism, while linear-scheduled queries can not exploit inter-operation parallelism. Therefore, in order to examine the performance sensitivity to query structure, we create two classes of queries for each query type: balance-scheduled query and linear-scheduled query. Figure 4.2 shows an example of balance-scheduled query and linear-scheduled query

Figure 4.2: Balance-scheduled Query and Linear-scheduled Query Created from a Query Type of 4 Tables.

created from a single query type which contains four database tables. Both balance-scheduled query and linear-scheduled query in the figure are based on the same database tables and the final results of the queries are the same. With a balance-scheduled query and a linear-scheduled query for each query type, there are 32 different kinds of queries although only 16 query types are available. 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 balance-scheduled query and 1-7 levels of operations in each linear-scheduled query.

When queries are generated, the workload generator randomly picks one of the query types and creates either a balance-scheduled query or a linear-scheduled query

by using the specific set of tables in the selected query type. The workload generator sleeps for certain time after finishing the generation of one query and then goes on to generate the next query. In our experiments, the inter-arrival time of queries are exponentially distributed.

Chapter 5

Performance Evaluation

In this chapter, performance of the hierarchical architecture for parallel query processing on a NOW is evaluated. There are several important parameters in the experiments: number of processors, number of batches, minimum fragmentation size (chunk size) for partial operations, mean inter-arrival time for the queries, and the scheduling structure of queries. The average response time of queries is presented to show the sensitivity of the hierarchical query processing system implemented in this thesis to the parameters in the experiments. Since balance-scheduled queries are more suitable for parallel execution, most of the experiments used balance-scheduled queries. As the 100Mbps network was down during the experiments, all the results in the thesis are obtained by using 10Mbps network.

Each experimentation value is averaged over 30 batches and each batch consists of 50 - 100 queries. 95% confidence intervals were obtained for the results to give an indication of the variability. The value of the confidence interval for each statistical

Figure 5.1: Performance under Different Inter-arrival Time

data value is less than 10% of the mean value reported when the system is heavily loaded and less than 5% when the systems is lightly loaded.

5.1 Performance as a Function of Inter-arrival Time

Varying the inter-arrival time of queries causes the load to change. If inter-arrival time increases, queries arrive into the system less frequently and the workload of the system decreases. If inter-arrival time decreases, the system becomes more heavily loaded.

The performance of an 8-node hierarchical system under different inter-arrival times is shown in Figure 5.1. The minimum fragmentation size for partial operations

of this hierarchical system is set to be 200,000 bytes. As the sizes of the database tables are 40,000 to 72,000 bytes, there is no partial operation at least at the first level of joins in queries.

The capacity of the system with above parameters is around 0.2 jobs per second (i.e., 5 seconds per job). Average response time is measured when the average inter-arrival time is 5.6 seconds to 30 seconds. It can be observed that the average response time is much longer when system is heavily loaded than when system is lightly loaded. When the inter-arrival time is 30 seconds, queries enter the hierarchical system less frequently, on average every 30 seconds. In this case, there is not much query queuing in the system and the average response time is around 24.8 seconds which is almost half of the response time when inter-arrival time is 5.6 seconds. Note that 24.8 seconds is the minimum mean response time for the query classes that are considered. This shows that the hierarchical system is capable of handling both heavy workload and light workload. The increasing of response time with smaller inter-arrival time is due to the query queuing in the system when the workload increases.

5.2 Speedup of the Hierarchical System

If twice as much hardware in a parallel system can perform the same amount of work in half the time, the parallel system is said to be successful in achieving linear speedup.

Since small parallel systems can not handle as heavy load as large parallel systems,

Figure 5.2: System Speedup When System is lightly Loaded

two experiments are designed to verify the speedup of hierarchical systems. In the first experiment, hierarchical systems with 1, 2, 4, 8 nodes are given the same amount of light workload that each system is able to handle. In the second experiment, each hierarchical system is given the same heavy workload as the largest hierarchical system can handle.

Figure 5.2 shows the performance of the systems when they are lightly loaded. The same work generated on one of the nodes with an inter-arrival time of 48 seconds per job is used for all the hierarchical systems. The minimum fragmentation size is 200, 000 bytes as above. The throughputs obtained for all the systems are around 0.02 jobs per second (50 seconds per job). From the results in Figure 5.2, with the same light workload, the speedup of a 2-node system to 1-node system is 3.91, while

the speedups of a 4-node system and a 8-node system to 1-node system are 5.68 and 5.37, respectively. It can be concluded that the small-sized hierarchical systems have great success in achieving good speedups under light workload.

The speedup obtained is 3.91 instead of 2 when the number of nodes in the system increases from 1 to 2. One of the reasons might be that the 1-node system is heavily loaded with an inter-arrival time of 48 seconds. In the 2-node system, there is not query generating process on one of the nodes while for the 1-node system the only node handles all the scheduling, real processing, and query generating. The overhead of the query generator may also affect the average response time for queries.

For large hierarchical systems, because the systems are relatively light-loaded and the scale of parallelism the systems can provide is not fully used, the mean response time of queries in large hierarchical systems is not one-fourth or one-eighth of that in 1-node system as expected. Moreover, as shown in Figure 5.2 that 4-node system is better in handling such workload with the sizes of database tables considered than 8-node system. The reason might be that the cost of communications for 8-node system is larger than the benefits obtained from the larger scale of parallelism for the sizes of data files considered. Similar results are obtained from the experiment for system scale-up which is shown in next section.

Figure 5.3 shows the performance of the systems when they are heavy-loaded. The inter-arrival time for each system is set to be 10 seconds, and the minimum fragmentation size is 200,000 bytes. It can be observed that there are only two data

Figure 5.3: System Speedup When System is heavily Loaded

points in the figure, one for 8-node system and the other for 4-node system. This is because the other two systems, 1-node and 2-node systems, become saturated when the arrival rate of 10 seconds per job is used. The 4-node system works fine for such workload with average response time of 46.0 seconds. 8-node system completes the same amount of workload in 29.0 second, which is 37% less than that, of 4-node system. So, for the hierarchical system, the speedup of an eight-node system to a 4-node system is about 1.59 while the smaller systems become saturated under the heavy workload.

From the above two experiments, it shows that small-sized hierarchical systems are successful in achieving good speedup under light workload and large-sized hierarchical systems have the performance of 1.59 times speedup with twice hardware under heavy workload.

Figure 5.4: System Scale-up

5.3 Scale-up of Hierarchical System

Besides linear speedup, ideal parallel systems expect the results of linear scale-up. Linear scale-up refers to the characteristic that with twice the hardware, the system can perform twice as the amount of work during the same time period.

In our experiment, we assume that twice of the inter-arrival time leads to half the amount of work. Figure 5.4 shows the response time of different hierarchical systems with different inter-arrival time. For 8-node system, the inter-arrival time is 6 seconds, and for 4-node system, the inter-arrival time is 12 seconds, twice as that of 8-node system. Thus the inter-arrival time of 2-node system is 24 seconds, and that of 1-node system is set to be 48 seconds. The throughputs of 1-node system to

8-node system are accordingly 0.02, 0.04, 0.08, and 0.17 jobs per second (48, 24, 12, and 6 seconds per job).

From the results in Figure 5.4, it is clear that hierarchical systems are successful in achieving linear scale-ups. The 2-node system, 4-node system and 8-node system can all complete 2 times, 4 times, and 8 times as the amount of work as 1-node system but in a less time period. It shows that the hierarchical systems not only succeed in linear scale-up but also succeed in combining linear scale-up with speedups of 2.54 to 3.77.

Particularly, 4-node system completes four times as the amount of work of 1-node system in only 73% of the time 1-node system needs. However, similar speedup results as in the previous section are obtained for 4-node and 8-node systems. The average response time of 8-node system under eight times the workload of 1-node system is longer than that of the 4-node system under four times the workload of 1-node system. It is believed that when choosing the number of processors to complete certain jobs, it is important to consider the size of data sent on the network compared to the total CPU time for completing the jobs. Otherwise, the results may not be as good as expected.

5.4 Performance Sensitivity to Fragmentation Size

In the hierarchical query processing system, processors directly process either operations or partial operations. Only when the workload of the system is light enough,

it is possible for a certain operation to reach the step of partial operation transfer mode. Otherwise, the operation will be processed by a single processor and there are no partial operations for that operation. While the workload determines whether partial operations are created for an operation, the number of partial operations that can be created for an operation is decided by the minimum fragmentation size. The larger the minimum fragmentation size, the fewer partial operations can be created. When the number of partial operations increases in the system, there are more processors working on one operation and on one query. It is possible for queries to be finished in less time because of the larger scale of parallelism. On the other hand, communication time increases when there are more partial operations in the system. When the minimum fragmentation size is too small, the average response time may be longer because of the large amount of communication overhead.

Figure 5.5 shows the system performance under different fragmentation sizes when the system is relatively heavily loaded. The experiments are performed with 8 processors and with inter-arrival time of 10 seconds. It can be observed from the figure that the average response time does not decrease with smaller fragmentation size. It suggests that dividing operations to partial operations does not help with the improvement of the system's performance at this system workload.

In Figure 5.6, these experiments are repeated with an inter-arrival time of 30 seconds. With such a workload, it can be observed that for some of the fragmentation sizes, the average response time for queries is shorter than that with larger

Figure 5.5: Performance Under Different Fragmentation Size When System is heavily Loaded

fragmentation size. With the fragmentation size of 40, 000 bytes, the response time is reduced by 5% of the response time with the fragmentation size of 200, 000 bytes. The response time with fragmentation size of 20, 000 and 10, 000 bytes is only 83% of the response time with fragmentation size of 200, 000. However, when the fragmentation size becomes smaller (5, 000 bytes), the average response time of queries becomes even higher than that with fragmentation size of 200, 000. These results indicate that, in lightly loaded system, dividing operations to partial operations does improve the performance of the system and there is a trade off between the number of partial operations and the communication overhead.

Figure 5.6: Performance Under Different Fragmentation Size When System is lightly Loaded

5.5 Performance Sensitivity to Query Scheduling Structures

In the experiments above, all the queries are generated as balance-scheduled queries. In this section, performance of hierarchical systems with only balance-scheduled queries is compared with performance of systems with only linear-scheduled queries. The balance-scheduled queries and linear-scheduled queries are generated from the same set of database tables as described in Chapter 4.3.

Figure 5.7 shows the experimental results of the average response time under different inter-arrival time. Number of processors used in the experiments is eight and the minimum chunk size is set to be 200, 000 bytes. For balance-scheduled queries,

inter-operation parallelism can be realized and for each arriving query, more than one processor can be assigned to work on each of its operations. Other queries arriving later than this query have to wait until one of the processors becomes free. Therefore, more query queuing appears in the system as the inter-arrival time decreases.

On the other hand, for linear-scheduled queries, there is no inter-operation parallelism, and if minimum fragmentation size is large enough, there will not be intra-operation parallelism in the system, either. Under such cases, each processor can deal with one operation of one query at a time. If there are N processors in the systems, N linear-scheduled queries can be processed at the same time. Waiting time for linear-scheduled queries is relatively shorter compared with the systems with only balance-scheduled queries. Because fewer processors work on each query at the same time, actual execution time for each linear-scheduled query is usually larger than that of balance-scheduled queries. When queries arrive on the system fast enough, the average difference of waiting time between balance-scheduled queries and linear-scheduled queries may become larger than the average difference of execution time between balance-scheduled and linear-scheduled queries. Response time for each query is the sum of waiting time and actual execution time and the difference between response time for balance-scheduled queries and that for linear-scheduled queries varies as inter-arrival time changes.

Figure 5.7 shows that the mean response time for balance-scheduled queries is not as good as that for linear-scheduled queries when inter-arrival time is small. When

Figure 5.7: Performance with Balance-scheduled and Linear-scheduled Queries

systems are light-loaded, the response time for the linear-scheduled queries becomes larger than that for the balance-scheduled queries. This is because there is less (or no) query queuing in both systems and waiting time for queries becomes less (or zero). If we take the example of no query queuing at all in the systems, for the system with only linear-scheduled queries, only one processor at a time is dealing with the only query in the current system, while the other processors are idling. For balance-scheduled query system, more processors are involved in finishing one query, and execution time is less than the sequential execution time for linear-scheduled queries. Thus, mean response time for balance-scheduled queries becomes less than that of linear-scheduled queries if workload in the system is light enough.

Chapter 6

Conclusions and Future Work

6.1 Summary

Centralized and distributed architectures for parallel query processing on Network of Workstations have their advantages and disadvantages. In the centralized architecture, workload is well shared but access contention may cause performance problems. Distributed architecture avoids access contention but load imbalance may be a problem. A new hierarchical architecture was proposed in [8]. This hierarchical architecture for query processing has been studied in this thesis.

In Chapter 1, two different approaches for parallel query processing in database systems are introduced: parallel database systems and the networks of workstations. Chapter 2 reviews the related work on parallel query processing in database systems and on processor scheduling in general parallel systems. Three architectures for parallel query processing on Network of Workstations are described in Chapter 3. Details about the implementation of the hierarchical architecture and hierarchical

task scheduling policy are also presented in Chapter 3. In Chapter 4, implementation environment and the database model used in the experiments are described in detail. Chapter 5 discussed the performance of the hierarchical systems using the experimental results.

6.2 Contributions of the Thesis

It has been found that the hierarchical systems are able to efficiently handle both light and heavy workload. Due to query queuing in the system, the mean response time of queries under heavy workload is longer than that under light workload. Hierarchical systems greatly succeed in achieving good scale-ups and speedups.

It has also been found that the mean response time of queries are very sensitive to the minimum fragmentation size (chunk size) for partial operations. When hierarchical systems are heavy-loaded, smaller chunk size does not help reduce the average response time although it increases the scale of intra-operation parallelism. When systems are relatively light-loaded, smaller chunk size helps reduce mean query response time due to the intra-operation parallelism. However, when chunk size is too small the benefit obtained from inter-operation parallelism does not compensate the cost of additional communication, and the average response time increases.

The experimental results show that the scheduling structure of queries is another important parameter that affects the performance of the hierarchical system. For a system with only balance-scheduled queries, performance is improved by reducing the

execution time of queries. However, when system load is moderate to high, waiting time of queries increases. For a system with only linear-scheduled queries, average response time for queries drops due to decreased waiting time. But when system is light-loaded, computing resource might be wasted since one processor can only deal one query at a time.

6.3 Future Work

The experimental results show that when using hierarchical system for parallel query processing, it is necessary to consider the overhead of communication compared to the total CPU time in completing the jobs. Due to time limit and the lack of experiment resources, the experiments were performed with only a set of selected parameters. In the future, we would like to do more experiments to see if the performance of hierarchical systems is also sensitive to the sizes of the database tables, the sizes of queries, the capacities of connected networks, and so on. In the hierarchical system implemented in the thesis, intermediate results are written to disk and read from disk for further operations to be performed. If pipeline is added to the hierarchical system, it is possible to improve the performance of the hierarchical system by reducing disk access time. The performance of the hierarchical system should also be compared with the performance of the centralized and distributed architectures.

In the PVM software package used in the implementation of hierarchical systems, there are several choices for data packing and routing. By using `PvmDataDefault`, it

is necessary to format the data before sending, and by using `PvmDataInPlace`, there is no need to copy the data. Using `PvmDirectRoute` can reduce round trip latency by reducing the delay on network routings. If other options for data packing and routing than default values in PVM software package are used in the experiments, performance of hierarchical systems may be improved.

References

- [1] C. K. Baru and O. Frieder. Database operation in a cube-connected multicomputer system. *IEEE Transactions on computer*, vol. 38, pp. 820-927, 1989.
- [2] C. K. Baru and S. Padmanabhan. Join and data redistribution algorithms for hypercubes. *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, pp. 161-168, February 1993.
- [3] M. S. Chen and P. S. Yu. Optimization of parallel execution for multi-join queries. *IEEE Transaction on Knowledge and data Engineering*, vol. 8, no. 3. pp. 416-428. 1996.
- [4] M. S. Chen, P. S. Yu, and K. L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *Proceedings of the Eighth International Conference on Data Engineering*, pp. 58-67, February 1992.
- [5] S. Dandamudi. Reducing run queue contention in shared memory multiprocessors. *Computer*, vol. 30, no. 3, pp. 82-89, March 1997.

- [6] S. Dandamudi. Using workstations for database query operations. *Proceedings of International Conference on Computers and Their Applications*, pp. 100-105. Tempe, Arizona. March 1997.
- [7] S. Dandamudi and S. P. Cheng. A hierarchical task queue organization for shared-memory multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 1, pp. 1-16. January 1995.
- [8] S. Dandamudi and G. Jain. Architectures for parallel query processing on networks of workstations. *Int. Conf. Parallel and Distributed Computing Systems*, pp. 444-451, New Orleans, October 1997.
- [9] S. Dandamudi and T. K. Thyagaraj. A hierarchical processor scheduling policy for distributed-memory multiprocessor systems. Technical report, School of Computer Science, Carleton University, 1997.
- [10] S. Dandamudi and S. Ayachi. A hierarchical processing scheduling policy for multiprocessor systems. *Proceedings of Eighth IEEE Symposium on Parallel and Distributed Processing*, pp. 100-109, Los Alamitos, California, 1996.
- [11] D. Dewitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communication of the ACM*, vol. 35, no. 6, pp. 85-98, June 1992.

- [12] R. Elmasri and S. B. Navathe. Fundamentals of Database Systems. *The Benjamin/Cummings Publishing Company, Inc.*, 1994.
- [13] O. Frieder and C. K. Baru. Site and query scheduling policies in multicomputer database systems. *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 4, pp. 609-619, August 1994.
- [14] G. A. Geist, A. Beguelin, J. Dongarra, R. Jiang, W. Manichuk, and V. Sunderam. PVM: Parallel Virtual Machine. *MIT Press*, 1994.
- [15] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: a comparison of features. *Calculatuers Paralleles*, pp. 137-150, vol. 8, no. 2, June 1996. (<http://www.epm.ornl.gov/pvm/>)
- [16] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. *Proceedings of ACM Sigmetric Conference*, pp. 226-236, Boulder, Colorado, May 1990.
- [17] S. Majumdar, D. L. Eager, and R. B. Bunt. scheduling in multiprogrammed parallel systems. *Proceedings of ACM Sigmetric Conference*, pp. 104-113, Santa Fe, NM, May 1988.
- [18] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, vol. 24, no. 24, pp. 63-113, 1992.

- [19] C. M. Murphy and D. Roten. Processor scheduling for multiprocessor joins. *Proceedings of IEEE fifth International Conference on Data Engineering*, pp. 140-148, 1989.
- [20] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, vol. 12, no. 4, pp. 269-284, July 1991.
- [21] J. K. Ousterhout. Scheduling techniques for concurrent systems. *Proceedings of the 3rd International Conference on Distributed Computer Systems*, pp. 22-30, October 1982.
- [22] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger. Parallelism in relational database systems: architectural issues and design approaches. *The Second International Symposium on Databases in Parallel and Distributed Systems*, pp. 4-29, 1990.
- [23] E. Rosti, E. Smirni, G. Serzizi, L. W. Dowdy, and B. M. Carlson. Robust partitioning policies for multiprocessor systems. *Performance Evaluation*, vol. 19, pp. 141-165, 1994.
- [24] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel systems. *Proceedings of ACM SIGMETICS Conference*, pp. 158-170, May 1993.

- [25] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, vol. 19, pp. 109-140, 1994.
- [26] K. L. Tan and H. Lu. On resource scheduling of multi-join queries in parallel database systems. *Information Processing Letters*, vol. 48, pp. 189-195, 1993.
- [27] K. L. Tan and H. Lu. Workload scheduling for multiple query processing. *Information Processing Letters*, vol. 55, pp. 251-257, 1995.
- [28] J. L. Wolf and M. S. Chen. A hierarchical approach to parallel multiquery scheduling. *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 578-589, 1995.

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