Architectures for Parallel Query Processing on Networks of Workstations[*]

Sivarama P. Dandamudi and Gautam Jain
Centre for Parallel and Distributed Computing
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada

TECHNICAL REPORT TR-97-13

Abstract

Networks of workstations (NOWs) are cost-effective alternatives to multiprocessor systems. Recently, NOWs have been proposed for parallel query processing. This paper explores the various architectures for using NOWs for parallel query processing. We start our discussion with two basic architectures: centralized and distributed. We identify the advantages and disadvantages associated with these architectures. We then propose a hierarchical architecture that inherits the merits of the centralized and distributed architectures while minimizing the associated pitfalls. We also discuss the performance and research issues that require attention in all three architectures.

1 Introduction

A relational database 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.

1.1 Objectives of query processing

The purpose of querying a database is not just to retrieve all tuples satisfying a user query but in a finite amount of time. With computers approaching high execution speeds and several users querying the database simultaneously, maintaining a reasonable level of performance is essential. Assuming multiple users submit several queries simultaneously, our objective for maintaining a reasonable level of performance is to reduce response time of each query. The response time of a query can be defined as the sum of the waiting time and the actual query execution time. The response time is essentially reduced by:

- Reducing the average waiting time of a query. This refers to the time difference between when a query is submitted to the system and when it is actually scheduled for execution.
- Reducing execution time of a query.

Reduction in waiting and execution times of a query automatically results in increased throughput, which refers to the average number of queries executed in a unit time. Increased performance is a direct consequence of achieving the above objectives. Parallelism in database queries provides immense potential for speeding up query processing.

1.2 Need for parallel query processing

Some of the motivations [6, 11, 3] for parallel query processing are:

- An ever increasing size of relational databases consisting of billions of records
- Data intensive read-only queries on these large databases
- Support for high-level ad hoc 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 all of the above is an ever increasing demand for performance.

1.3 System resources

There exist two hardware resources for parallel query processing, namely, parallel database machines and network of workstations. Commercial parallel database systems have been extensively used for improving database system performance by processing queries in parallel. A new concept, that of using network of workstations (NOWs) for parallel query processing has also been proposed for improving database system performance [3]. We extend the study

[*]Appears in *Proc. Int. Conf. Parallel and Distributed Computing Systems*, New Orleans, October 1997. This report can be obtained from http://www.scs.carleton.ca.

and propose different architectures for NOW–based parallel query processing.

1.4 Contributions and overview of the paper

We discuss three architectures for query processing on networks of workstations. We describe two basic architectures — centralized and distributed — and discuss their advantages and disadvantages. We also identify how processor allocation and load sharing can be achieved in these architectures. We then propose a hierarchical architecture that inherits the merits of the centralized and distributed architectures while minimizing their disadvantages. These three architectures have not been studied previously and present a variety of new research problems that need investigation.

The remainder of the paper is organized as follows. Section 2 discusses the parallel database system architecture and the following section presents the NOW-based system. This section also discusses the differences between the two types of systems. The centralized and distributed architectures are described in Section 4 and the hierarchical architecture is presented in Section 5. Section 6 is a summary.

2 Parallel Database System Architectures

A parallel database system can be built by connecting together several processors, memory units, and disk devices through a global interconnection network. Depending upon how the different components are interconnected they can be categorized under three main classes [6]: shared-nothing, shared-disk, and shared-everything. Each architecture consists of processors (p), memory units (m), disk units (d), local buses, and a global interconnection network as shown in Figure 1.

The shared-everything architecture allows all processors to have access to both the global shared memory and all the disks. In the shared-disk architecture, each processor has its own local memory but all processors have direct access to all disks in the system. Both these architectures facilitate sharing and load balancing as the disks are all shared. In shared-nothing, each processor has its own (private) disk and memory units and processors communicate with each other by means of explicit message passing.

The shared-everything and shared-disk architectures are not suitable for large systems due to the interconnection network bandwidth limitations. Share-nothing architecture for database systems is popular from the standpoint of scalability and reliability [6]. Shared-nothing architectures minimize interference by minimizing resource sharing. They also exploit commodity processors and memory without needing an incredibly powerful interconnection network [6, 11]. Some of the commercial and experimental database systems based on the shared-nothing architecture are the Teradata, NonStop SQL, Gamma, and Bubba systems.

3 Network of Workstations for Parallel Query Processing

3.1 Motivation

The use of network of workstations (NOWs) for parallel query processing has been proposed in [3]. Several studies have shown that a large fraction (up to 80% depending on time of day) of the processing power available on a network of workstation is wasted by idling cycles. Substantial performance gains can be obtained by using these idle processing cycles. Dandamudi [3] has demonstrated the performance benefits of parallelizing a four way join query using a set of workstations. However, there are several performance issues associated with using NOWs for database query processing. These performance issues include load balancing, fault-tolerance, job migration and communication network latency. Communication network latency being the most important as NOW-based systems rely on message passing for communication.

3.2 Differences with Parallel Database Systems

There are several significant differences [3] between a shared-nothing parallel database system and one that is based on a NOW. The differences give rise to several performance issues that need to be addressed when using NOWs for parallel database query processing. The main differences are:

- The database itself is not partitioned across the processing modules as in a parallel database system. This causes additional overhead as the data have to be sent to the workstations.
- The workstations in a NOW-based system are heterogeneous. The workstations could be of different types (ranging from low-end PCs to powerful workstations).
- 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 in balancing load across the system.
- Parallel database systems, have a dedicated, special purpose fast communication network. NOW-based systems use the general purpose LANs, which are slow and involve high communication overhead.

The above differences clearly indicate that performance gains are more with parallel database systems. However, the immense and untapped computing power of network of workstations for parallel processing has not received a lot of attention in the database community. The benefits of using NOWs for parallel database query processing has been reported in [3].

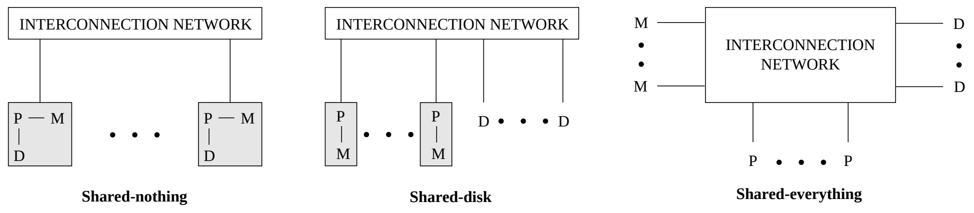

Figure 1: Some parallel database system architectures

3.3 Performance Issues in NOWs

The major performance issues that arise due to the differences in parallel database systems and one based on network of workstations can be explained in detail as follows.

3.3.1 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. Consequently, dynamic load balancing schemes have to be used for balancing the extra load that arrives at workstations during query processing. Since the data are not pre-partitioned as in the shared-nothing parallel database system, NOWs give more flexibility in terms of processor allocation and load balancing. As a result, query processing on networks of workstations can exploit various processor allocation and load sharing techniques proposed for multiprocessor systems and distributed systems, respectively [7, 8, 12, 13, 15]. Sections 4 and 5 discuss various strategies for load balancing in NOW-based parallel query processing.

3.3.2 Job Migration

In a NOW–based system, the workstations are usually owned by different users. A user may claim his/her workstation from the pool of shared workstations for reasons such as overloading. In such a situation, the incomplete remote job(s) executing on the workstation have to be either

- killed and restarted on another workstation, or
- the state of the process has to be saved and the job has to be transferred along with its partial results and resumed on another workstation

The strategy that minimizes the response time is obviously selected. In distributed systems, it has been demonstrated that job migration is useful [8]. Both strategies cause delay in response time. In the first one, the processing has to restart, resulting in duplication as the work done is lost, and in the second one, the time required to shift the job from the claimed workstation to another workstation. Both strategies require selection of a new workstation. The benefits of using heuristics for selecting an appropriate site for job transfer have been addressed by Lu and Carey [9].

3.3.3 Fault Tolerance

This problem is another key issue in a NOW–based system. There are several sources from which failures can occur. Workstation owners can intentionally generate faults, workstations can fail due to a system malfunction or links connecting different workstations can fail. This affects overall system performance because of the extra time required to recover from the faults.

4 Basic Architectures for Parallel Query Processing

In all the architectures, workstations are used only for processing of queries, which require read-only access to data from the database. There is a special node that is responsible for running DBMS software. All update-related activities are performed by this special node so that concurrency and recovery problems are minimized. The rationale for this is the observation that in most database environments, queries dominate. Thus, the following discussion focuses on software architectures for parallel query processing.

From our perspective, there are three major steps involved in managing 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 a traditional DBMS. The second step, processor allocation, determines the set of workstations that is responsible for processing either an entire query or a part of it. It is the operation scheduler that actually schedules operations of the query. Note that the operation scheduler may schedule an entire operation or can parallelize and load balance the operation. Next we discuss how the centralized and distributed architectures perform these steps.

4.1 Centralized Architecture

The centralized architecture operates in a master-slave mode. The database node performs all the work related to query preprocessing. The other workstations act as slaves and perform the work assigned to them by the master node. Figure 2 shows the centralized architecture with some of the

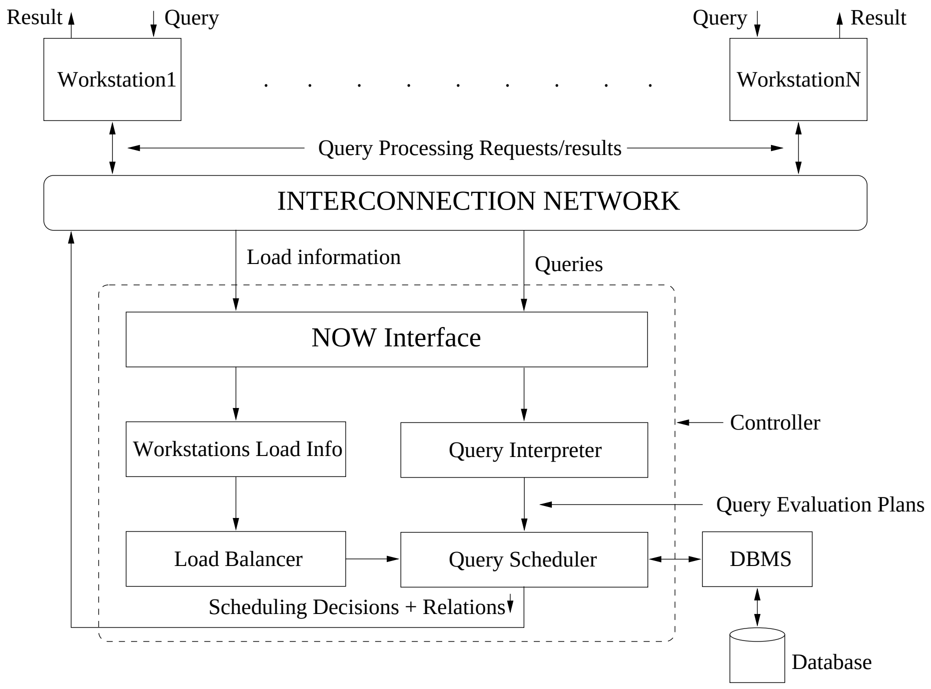

Figure 2: Centralized Architecture

main components.

Queries are submitted at various workstations and are forwarded to the database node for interpretation and processing. The database node interprets all queries received by the system and generates optimized query execution plans. Processor allocator takes the system state (i.e., non-query processing load and the current query load) into account in making processor allocation decisions. In this architecture, the database node is responsible for collecting the system state information. Whenever the state of a node changes, it informs this change in state to the database node for updating purposes.

The processor allocation decisions are dynamic and can vary during the lifetime of a query. The policies proposed for distributed-memory multicomputer systems can be adapted successfully [12, 13]. In query processing, however, allocations decisions are typically made at the operation level. As a result, load balancing can be done effectively in the centralized architecture. For example, if system background load is down or the query load at the database node is low, the next operation can get more processors allocated. Another advantage is that inter-query parallelism can be exploited as all queries are interpreted and scheduled from a central processor. For example, if two queries arrive and there is a common join operation, then both queries can share the result of this join rather than executing the join twice. Further work is needed to determine the strategies useful for the central architecture. The centralized organization, however, suffers as the database node becomes a bottleneck. Therefore, this architecture is limited to small systems.

4.2 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 shown in Figure 3. As a result, it eliminates the bottleneck problem discussed in the last section.

In the distributed architecture, processor allocation and operation scheduling can be incorporated into load sharing. Load sharing has been extensively studied in the context of distributed systems. While those studies are not directly applicable to the query processing activity, ideas from those studies can be adapted to query distribution. Here we briefly discuss some of the possibilities.

In this 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 a threshold-based method is used

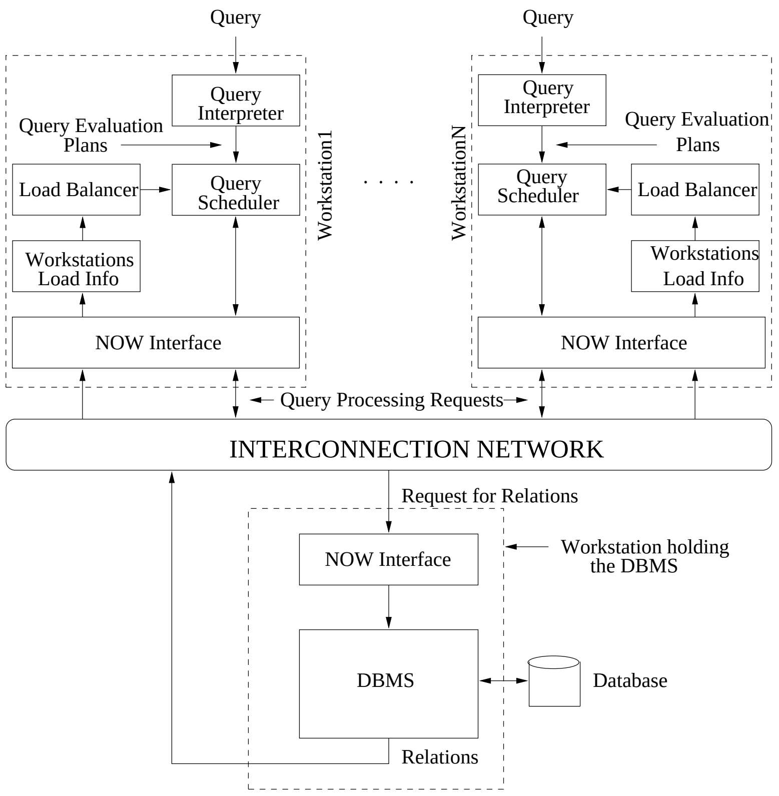

Figure 3: Distributed Architecture

to detect if a node is overloaded [14, 15]. In NOW-based systems, we have to consider a weighted combination of the background and query loads as an indicator of system load. Then, a simple threshold can be used to determine if a node is overloaded. This mechanism can classify a node as underloaded (i.e., in a receiver state), or overloaded (i.e., in a sender state), or normally loaded (OK state). Load sharing is achieved by transferring work from overloaded nodes to underloaded nodes. There are several ways in which this can be done. Here we discuss the following four types of policies [7, 15, 4].

4.2.1 Sender-initiated policy

In this policy, the sender node (i.e., overloaded node) attempts to transfer work to a receiver (i.e., underloaded node). When a node becomes a sender, the node probes (up to) a maximum of probe limit P_l randomly selected nodes to locate a node that is in the receiver state. If such a node is found, work is transferred to that node for remote execution. The granularity of work transferred can be:

- an entire query along with the required relations, or
- an operation of a query along with the required relations, or
- a partial operation along with fragments of relations

The larger the granularity, the smaller the communication and load distribution overhead. However, load imbalance increases with increasing granularity.

Note that probing stops as soon as a suitable target node is found. If all probes fail to locate a suitable node, the probing process is reinitiated after a predetermined reinitiation period if the node is still overloaded. When the transferred work arrives at the destination node, the node must accept and process even if the state of the node at that instance has changed since the probing.

4.2.2 Receiver-initiated policy

This is similar to the sender-initiated policy except that load distribution is initiated by an underloaded node. When a node is underloaded, it initiates the probing process as in the sender-initiated policy to locate a sender node. If such a node is found within P_l probes, work from that node is transferred. As described in the last section, the granularity work can vary from a query to a partial operation. If a sender node is not located within P_l probes, load distribution is reinitiated after the lapse of reinitiation period as in the sender-initiated policy.

4.2.3 Adaptive Policy

In distributed systems, it has been shown that the sender-initiated policy performs better at low to moderate system loads and the receiver-initiated policy is better from moderate to high system loads. This is because at low system loads it is easier to find an underloaded node and at moderate to high system loads probability of finding an overloaded node is high. This has motivated some researchers to devise a hybrid load sharing policy that behaves like the sender-initiated policy at low to moderate system loads and like the receiver-initiated policy at moderate to high system loads.

The adaptive policy proposed in [14] belongs to the hybrid category and has both sender-initiated and receiver-initiated components. In addition, this policy does not select random nodes for probing as in the sender-initiated and receiver-initiated policies. Instead, each node keeps the state information gathered from previous probes and uses this information to select a most likely target node. The state information is gathered at each node by maintaining three lists: a senders list, a receivers list, and an OK list. More details on this policy are presented in [14, 15]. In distributed systems, the adaptive policy has been shown to provide substantial performance improvements over the sender-initiated and receiver-initiated policies.

4.2.4 Hierarchical Policy

In the hierarchical policy, instead of a single node maintaining the entire system state, a set of nodes is given this responsibility. The system is logically divided into clusters and each cluster of nodes will have a single node that maintains the state information of the nodes within the cluster. The state information on the whole system is maintained in the form of a tree where each tree node maintains the state information on the set of processor nodes in the sub-tree rooted by the tree node. More details are given in [4, 10]. The hierarchical policy provides further performance improvements over the adaptive policy.

4.2.5 Performance Problems

Distributed architecture introduces several performance problems. Some of these are:

- In the centralized architecture, the processor allocator can dynamically make allocations decisions based on the query load as all queries are fed to the central database node. Since such information is lacking in the distributed architecture, load imbalance may occur.

- Another related problem is the background load information. Unlike the centralized architecture, this information is distributed and has to be gathered periodically in order to make proper processor allocation decisions.

- Distributed architecture introduces additional communication overhead as each workstation has to keep track of the query and background workloads of all other workstations connected to the network.

- Network delays could lead to workstations making non-optimal processor allocation decisions, thereby increasing the need for load balancing.

- Since queries are distributed, inter-query parallelism is difficult to exploit in this architecture (i.e., diminished opportunities for global query optimization).

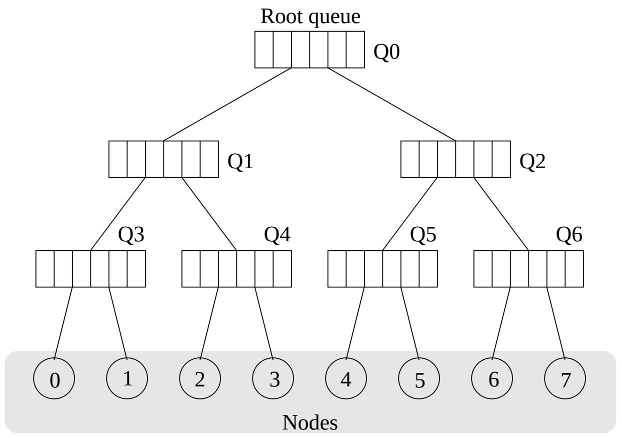

Figure 4: Hierarchical architecture for $P = 8$ nodes with a branching factor B of 2

The next section presents a hierarchical architecture that effectively combines the best features of the centralized and distributed architectures.

5 Hierarchical Architecture

The centralized architecture is simple and communication overhead is less, however, scalability is limited. On the other hand, the distributed architecture solves the scalability problem but the additional communication overhead and the associated performance degradation are undesirable. We propose a hierarchical architecture that inherits the merits of the centralized and distributed architectures while minimizing the associated disadvantages. Thus, this architecture provides a reasonable solution for using NOWs for parallel query processing.

The hierarchical architecture is based, in concept, on the hierarchical run queue organization proposed previously [1, 2]. 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 nodes). Figure 4 shows an example hierarchical architecture for 8 nodes with a tree branching factor B of 2.

The hierarchical architecture uses a self-allocation scheme. This implies that 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 queue. 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 queue at level $(L-2)$ and the process is repeated up the tree until it finds work[1] to be scheduled (unless the root queue is empty). However, in order to reduce access contention at higher levels, when a queue is accessed, 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 the account the availability of the increasing number of queues per level). At the bottom of the tree this set size reduces to just one work unit (corresponding to scheduling that work unit on the associated node).

To explain the query processing in the hierarchical architecture, let us assume that all query queues are empty and the root queue has four queries q_1 to q_4 waiting to be executed on a 64 node system (see Figure 5). An idle node (assume node 0) visits the root queue as all the queues along the path are empty. The policy operates in one of three modes: *query transfer mode*, *operation transfer mode* or *partial operation transfer mode*.

The query transfer mode is applied as long as the number of waiting queries is more than or equal to the branching factor B. Since the branching factor is 2 in this example and there are more than 1 query waiting at Q0, the policy operates in query transfer mode. In this mode, the work units transferred between successive levels of the tree are entire queries. In the example, node 0 moves 1/B of the four waiting queries one level down the tree to Q1 as shown in Figure 5 (leaving q_3 and q_4 at the root node). By this move the queries q_1 and q_2 are restricted to a partition in the left half of the system (i.e., left 32 of the 64 processors). This process is repeated at Q1 as the number of queries is greater than or equal to the branching factor B. Therefore, node 0 moves q_1 to Q3. At this point, q_1 is allocated a partition that consists of all the processors for which the queue Q3 acts as the root node. Thus, the partition allocated to q_1 consists of 16 nodes — nodes 0 through 15.

At this point in the transfer process, the mode is switched to operation transfer mode. The operation transfer mode works similar to the query transfer mode except that the operations of a query are transferred rather than queries. Assuming that q_1 has four operations $1/B$ of these operations are moved down the tree leaving two operations at Q3. This process is repeated as long as the number of operations is greater than or equal to B. Thus, of the two operations moved to Q7, one operation is left at Q7 while the other is moved to Q15. As a result, this operation is allocated 4 nodes (i.e., nodes 0 through 3).

[1] As noted before, work unit can be a query, an operation, or a partial operation.

At this point, the mode is switched to partial operation mode as the number of operations is less than B. In this mode, operation is parallelized. Thus, each transfer step moves $1/B$ fraction of the operation until the work reaches the node. Since B is 2, 50% of the q_1 operation is moved from Q15 to Q31 and 25% is finally scheduled on node 0.

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 is 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 contention problems.

Performance of the hierarchical architecture needs investigation. There are strong indications that it will perform well [5]. We are currently conducting a performance evaluation of the hierarchical architecture.

6 Conclusions

Network of workstations present an alternative to speed-up query execution. However, the system behaviour is different from that of the shared-nothing parallel database system. The main difference that is both an asset and a performance drawback is that the data in NOW are not pre-partitioned. This feature of NOWs gives more flexibility in terms of processor allocation and load balancing. On the down side, the data are to be moved after processor allocations decisions are made incurring a communication overhead.

In this paper, we have presented two basic architectures — centralized and distributed — and discussed their advantages and disadvantages. We then proposed a new hierarchical architecture that inherits the merits of the two basic architectures while limiting the associated disadvantages. The three architectures described require further research to understand the merits and their relative performance. We are currently studying the performance issues associated with these three architectures.

Acknowledgments

We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada and Carleton University.

References

[1] S. P. Dandamudi and S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems,* Vol. 6, No. 1, January 1995, pp. 1-16.

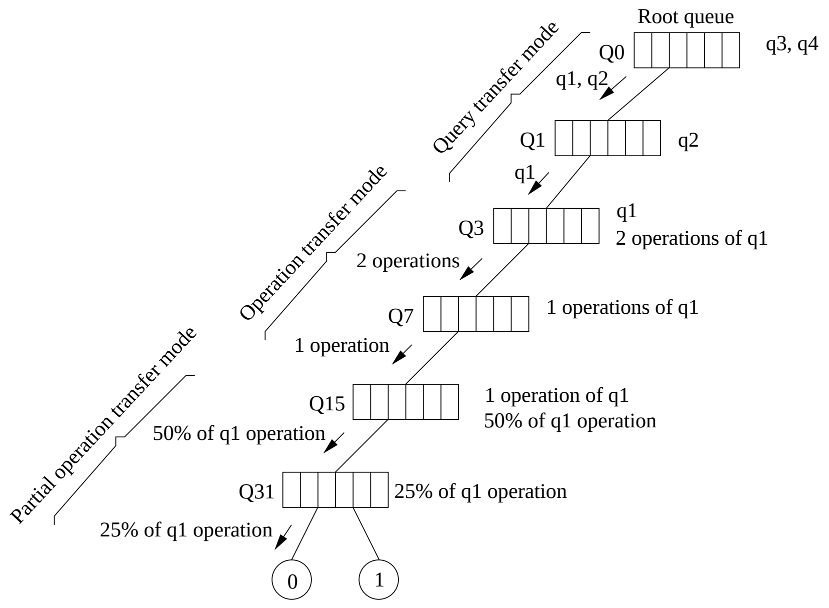

Figure 5: Work transfer process in the hierarchical architecture (number of nodes P = 64 and the branching factor B = 2)

[2] S. Dandamudi, “Reducing Run Queue Contention in Shared Memory Multiprocessors,” *IEEE Computer*, March 1997, pp. 82-89.

[3] S. P. Dandamudi, “Using Network of Workstations for Database Query Operations,” *Int. Conf. on Computers and Their Applications*, Tempe, 1997, pp. 100-105.

[4] S. P. Dandamudi and M. Lo, “A Hierarchical Load Sharing Policy for Distributed Systems,” *IEEE Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Haifa, Israel, 1997, pp. 3-10.

[5] S. P. Dandamudi and T. K. Thyagaraj, “A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems,” *4th Int. Conf. High Performance Computing*, Bangalore, India, December 1997.

[6] D. DeWitt and J. Gray, “Parallel Database Systems: The Future of High Performance Database Systems,” *Comm. ACM*, Vol. 35, No. 6, June 1992, pp. 85-98.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan, “A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing,” *Performance Evaluation*, Vol. 6, March 1986, pp. 53-68.

[8] M. Harchol-Balter and A. B. Downey, “Exploiting Process Lifetime Distributions for Dynamic Load Balancing,” *ACM Sigmetrics Conf.*, 1996, pp. 13-24.

[9] H. Lu and M. J. Carey, “Load-Balanced Task Allocation in Locally Distributed Computer Systems,” *Int. Conf. on Parallel Processing*. 1986, pp. 1037-1039.

[10] M. Lo and S. Dandamudi, “Performance of Hierarchical Load Sharing in Heterogeneous Distributed Systems,” *Int. Conf. Parallel and Distributed Computing Systems*, Dijon, France, 1996, pp. 370-377.

[11] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger, “Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches,” *Second Int. Symp. Databases in Parallel and Distributed Systems*, 1990, pp. 4-29.

[12] E. Rosti, E. Smirni, G. Serazzi, L.W. Dowdy, and B.M. Carlson, “Robust Partitioning Policies for Multiprocessor Systems,” *Performance Evaluation*, 19:141-165, 1994.

[13] S.K. Setia, M.S. Squillante, and S.K. Tripathi, “Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Systems,” *ACM SIGMETRICS Conf.*, pp.158-170, May 1993.

[14] N. G. Shivaratri and P. Krueger, “Two Adaptive Location Policies for Global Scheduling Algorithms,” *IEEE Int. Conf. Dist. Computing Systems*, 1990, pp. 502- 509.

[15] N. G. Shivaratri, P. Krueger, and M. Singhal, “Load Distributing for Locally Distributed Systems,” *IEEE Computer*, December 1992, pp. 33-44.