Appears in *Proc. Int. Conf. Computers and Their Applications*, Tempe, March 1997.

Using Networks of Workstations for Database Query Operations

Sivarama P. Dandamudi
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada
sivarama@scs.carleton.ca

TECHNICAL REPORT TR-97-3

ABSTRACT - Performance of database systems can be improved by applying parallel processing techniques. Several commercial parallel database systems are available but these are expensive. In the parallel processing area, there is a trend to use networks of workstations as a virtual parallel machine. The obvious advantage of such systems is their low cost. The availability of public domain software (such as PVM and MPI) to configure a parallel virtual machine over a network of workstations has led several researchers to study performance issues in this area. In this paper, we propose the use of network of workstations to improve the performance of database queries. We present analysis of a four-way join operation to demonstrate the feasibility of our proposal. However, using networks of workstations for database processing poses its own problems. These problems have to be solved before we can profitably employ networks of workstations for database applications. We provide a discussion of the issues involved and pointers for further research.

Key words: Networks of workstations, database systems, join cost analysis, parallel database systems, parallel virtual machines.

1. INTRODUCTION

There are several application areas that require very high transaction processing rates. For example, transaction processing rates of stock market and banking industry are very high. At the same time the size of databases is increasing. The traditional mainframe-based transaction processing systems are not able to meet this demand. The current trend is to build parallel database systems in order to effectively support these applications.

With the dominance of the relational database systems in the marketplace, parallelizing relational queries has received considerable attention. As a simple example, consider the following SQL query on an employee database:

```
SELECT emp_name, emp_address, emp_phone
FROM   employee
WHERE  emp_dept = 'Personnel'
```

Suppose that we have ten processing modules, where each processing module consists of a processor, memory, and a disk unit. If the employee table is large (say 100,000 employees), we can partition the table into 10 disjoint fragments and distribute them to the ten processing modules. Then, instead of a single processing module working on all 100,000 entries, ten of them will be working concurrently on only 10,000 entries each. This is referred to as partitioned parallelism [9].

Several parallel database systems have been available commercially, including the systems from Teradata and Tandom. We will discuss different architectures of parallel database systems in Section 2. A problem with these systems is that they are expensive and smaller organizations may not be able to afford such highly parallel database systems.

Similar problem exists in parallel processing area. Commercial parallel systems are very expensive, often costing millions of dollars depending on system size. However, several studies have shown that a large fraction (up to 80% depending on the time of day [16]) of the processing power available on a network of workstations (NOW) is wasted by idling workstations. Substantial performance gains can be obtained by using these idle processing cycles. In order to use these idle workstations for parallel processing, we need a software to configure a virtual parallel machine on a NOW. Several public domain software packages such as PVM [12,18] and MPI [10,13,19] are available for this purpose. Section 3 discusses the two basic parallel system architectures and provides some details about the PVM software.

The database area can also benefit from a similar approach. There are some significant differences between the two. For database applications, it is not possible to distribute database over a NOW because each workstation is under an individuals control. This lack central control on the system makes it undesirable to distribute relations across the workstations. Therefore, the database will have to be under the control of a traditional database system, which provides security and maintains the integrity of data. However, we can use the workstations to speed up query processing, which only requires read-only access to data.

In such a NOW-based system, since the tables are all residing with the central database, every time a query is executed on a NOW, the workstations involved in the query execution will have to receive the required (partial) tables before query processing can proceed. This overhead is not present in a parallel database system, where the required data usually resides at the processing module (for example, in a shared-nothing type architecture - see next section for details). We show in Section 4 that table transfer overhead can be amortized over a complex query operation. Thus, despite the added overhead, query processing on a NOW is beneficial. In particular, we should note that any additional benefit we get in terms improved performance is often obtained with no additional cost. Often, only the existing workstations are used. Also note that NOW not only provides additional processor cycles to be used but also adds substantial main memory for query processing. This addition of main memory is an important feature that can substantially benefit query processing by reducing the number of disk accesses. The benefit of keeping more data in main memory is well recognized (for example, the Oracle VLM (Very Large Memory) exploits the main memory). However, for NOW to become a reality for database applications, we need a (public domain) software similar to that in the parallel processing area.

These are several significant differences between a highly parallel database system and one that is based on a NOW. Section 5 addresses these issues. The issues raised in this section open several new research topics in the parallel database systems area. The paper concludes with a summary.

2. PARALLEL DATABASE SYSTEMS

A parallel database system can be built by connecting together several processors, memory units, and disk devices through a global interconnection network. Depending on how these components are interconnected there can be a variety of possible architectures. Figure 1 illustrates some parallel database system architectural types [5,20]. Each of the architecture consists of processors (p), memory units (m), disk units (d), local buses,

and a global interconnection network.

In the shared-nothing architecture, each processor has its own (private) disk and memory units. A processor can communicate with other processors by means of explicit message passing. In shared-everything, all processors have access to both the global shared memory as well as all disks. In the shared-disk architecture, each processor has its own local memory but all processors have direct access to all disks. The interconnection network in these architectures plays an important role in determining the overall system performance. A detailed description of suitable interconnection networks and their impact of performance is beyond the scope of this paper. ([11] gives a survey of interconnections networks.)

The shared-everything and shared-disk architectures are not suitable for large systems due to interconnection network bandwidth limitations. Shared-nothing database systems are attractive from the standpoint of scalability and reliability [3,4,20]. Several commercial and experimental systems such as Non-Stop SQL from Tandem [21], DBC/1012 from Teradata [22], nCube-Oracle system, Gamma at the University of Wisconsin [6,7], and Bubba at MCC [15] belong to this type of architecture. Hybrid architectures have also been proposed [24].

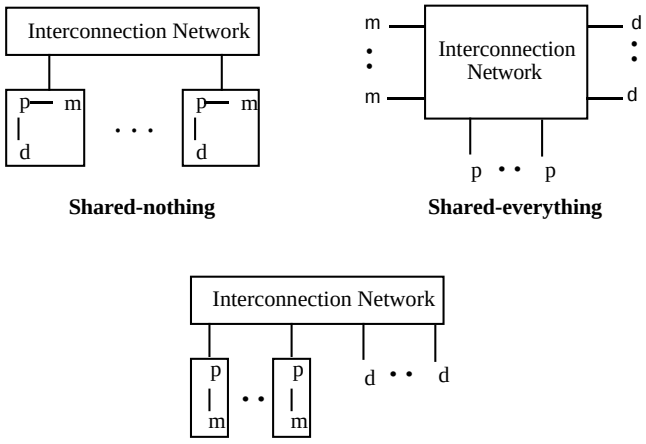

Figure 1 Some parallel database system architectures

To effectively use parallel database systems, we must find a way to partition the database across the system resources such that parallel processing is facilitated. Typically, all relations are horizontally partitioned across all disk units in the system i.e., relations are partitioned into tuples (or some set of tuples) and the tuples are distributed to disk drives in the system. There are several ways of distributing the tuples of a relation. Some of these are [8]: (a) round robin strategy in which tuples are loaded into a relation, which is distributed across all the disk drives in the system (equal to the number of processors), in a round robin fashion among all disk drives; (b) hashed strategy, which applies a randomizing function to the "key" attribute of each tuple to select a disk drive; (c) range partitioned with user-specified placement by key value, i.e., user specifies a range of key values for each processor and then the tuples are distributed to processors according to the specified range of key values; (d) range partitioned with uniform distribution, i.e., the system horizontally partitions relations with an user-specified attribute (the selected attribute may not be a key attribute of the relation). Data partitioning strategy has a significant impact on the performance of the overall system as introducing data skew (i.e., non-uniform distribution of data) can lead to execution skew.

3. USING NETWORKS OF WORKSTATIONS FOR PARALLEL PROCESSING

Parallel processing systems can be divided into either shared-memory or distributed-memory systems. Shared-memory systems, often simply called multiprocessors, provide the programmer a single address space much like that of a traditional uniprocessor system. Communication among processors is through shared memory variables (see Figure 2). These systems, like the shared-everything and shared-disk parallel database systems, do not scale to large systems due to network bandwidth limitations.

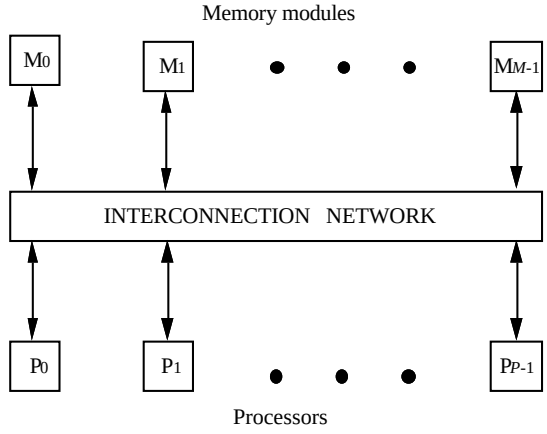

Figure 2 A shared-memory multiprocessor system

Distributed-memory systems, also called multicomputers, are the shared-nothing counterpart as shown in Figure 3. These systems communicate by means of message-passing. The advantage of these systems is their scalability. Several commercial systems are available with hundreds and thousands of processors. The systems from Intel and nCube belong to this category.

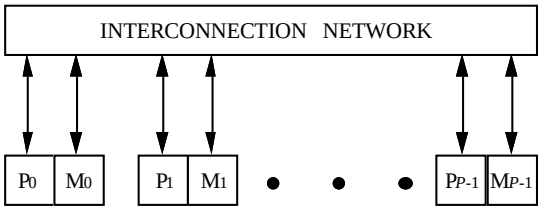

Figure 3 A distributed-memory multicomputer system

3.1. PVM: A Parallel Programming Environment

PVM is a software package that allows a heterogeneous network of parallel, serial and vector computers to appear as a single concurrent computational resource (i.e., a distributed-memory system). PVM consists of two parts:

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

Applications using PVM can be written in Fortran 77 or C and parallelized by using message-passing constructs common to most distributed-memory computers. PVM supports heteroge-

neity at the application, machine, and network level. In other words, PVM allows application tasks to exploit the architecture which is best suited to their solution. For example PVM may include parallel machine and graphical workstation. Computation can be done on the parallel machine and the results may be displayed in graphical form on the workstation.

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

PVM supplies the functions to automatically start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. A task in PVM is defined as a unit of computation analogous to a UNIX process. The PVM application can then be started from a UNIX prompt on any of the hosts. Multiple users can configure overlapping virtual machines and each user can execute several PVM applications simultaneously. By default, when the PVM is spanning a task, it tries to spawn in on the least loaded machine. But the least loaded machine might not be the fastest one to complete the task and loads are subject to change. User may override default spawning behavior by specifying architecture to spawn a task on or even a particular host. This provides greater flexibility while retaining the ability to exploit particular strengths of individual machines on the network. If some host fails PVM will automatically detect it and delete the host from the virtual machine. Complete details on the PVM are available in [12,18]. The Berkeley NOW project looks at effectively using networks of workstations for parallel processing [1].

4. USING NETWORKS OF WORKSTATIONS FOR DATABASE PROCESSING

This section discusses how the use of networks of workstations improves performance of database systems. As we have indicated, network of workstations is distributed and there is no central control. For example, users might reboot their workstations if they do not like someone else stealing processor cycles on their machines. Because of this lack of central control on the resources, they also differ significantly from the parallel database machines discussed in Section 2. We discuss these differences and their impact on database processing in the next section.

The use network of workstations for database processing leads to a system that exhibits different characteristics than those found in distributed and parallel database systems. The major difference is that the database itself is neither distributed among the workstations (as in distributed database systems) nor relations declustered as in a parallel database system. Our proposal is to simply use the additional processor cycles and memory available on these workstations for database processing on a temporal basis. In particular, we restrict our attention to query processing operations. Using idle workstations for database processing can improve performance of the system due to:

- Concurrent execution of multiple queries
- Exploiting intra-query parallelism
- Exploiting intra-operation parallelism

The following sections discuss these three topics in detail.

4.1. Concurrent Execution of Multiple Queries

Response time of a query (i.e., the time difference between when the query is done and when it was submitted) consists of two main components:

Response time of a query = $W + E$

where W is the waiting time of the query to get to the execution stage and E is the actual query execution time. When multiple processors are available for query processing, we can schedule several queries concurrently. This improves performance of the system by reducing the waiting time W of a query. For example, if three queries are waiting to be executed, if we have three idle workstations available for this work, we can schedule these three queries on the three workstations. This, however, involves shipping the required relations to these workstations to work on the query. For complex queries, this overhead can be small in comparison to the amount work involved in query execution. We will show later that this is true by means of an example.

4.2. Intra-Query Parallelism

While running multiple queries reduces the response time of a query by reducing the associated waiting time, it does not reduce the query execution time. This and the following technique attempt to reduce query execution time. One way to reduce execution time of a query is to apply multiple processors to work in parallel on a single query. This can be done by looking at the available parallelism within the query and scheduling the operations that can be done in parallel on different workstations. For example, consider a four-way join operation on relations R_1, R_2, R_3, and R_4. Figure 4 shows the structure of the four-way join operation. It can be seen from this figure that $R1 \bowtie R2$ and $R3 \bowtie R4$ can be done in parallel (at level 1). Thus if we have an additional workstation, we can schedule these two join operation in parallel. In order to see the amount of benefit one can get by such a strategy, we provide a complete analysis of the four-way join operation next.

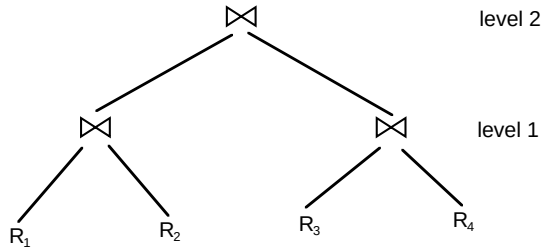

Figure 4 Four-way join execution strategy
($\bowtie$ represents the join operation)

4.2.1. Analysis of four-way join

Let B_i represent the number of disk blocks associated with relation R_i. Assume that the join is performed by the nested loop algorithm that uses two main memory buffers. Then, the cost (in disk accesses) of joining two relations R_1 and R_2 is given by

$$\text{cost of } R_1 \bowtie R_2 = B_1 + (B_1 * B_2) + (\alpha_{R1R2} * |R_1| * |R_2|) / bfr_{R1R2} \text{ disk accesses} \quad (1)$$

where $|R_i|$ represents the size (in tuples) of relation R_i, bfr represents the blocking factor of $R_1 \bowtie R_2$, and α_{R1R2} is the join selectivity of $R_1 \bowtie R_2$. Join selectivity represents the fraction of tuples that qualify compared to the number of tuples in the Cartesian product (represented by X) of the two relations and is given by

$$\alpha_{R1R2} = \frac{|(R_1 \bowtie R_2)|}{|(R_1 \text{ X } R_2)|} = \frac{|(R_1 \bowtie R_2)|}{|R_1| * |R_2|}$$

We now compute the four-way join execution time with no additional workstations.

Execution time of the query =
execution time of $R_1 \bowtie R_2$ +
execution time of $R_3 \bowtie R_4$ +
execution time of the level 2 join (2)

Execution time of $R_1 \bowtie R_2$ = number of disk accesses given by

Eq. (1) * T_d

where T_d is the average disk block access time.

Execution time of $R_3 \bowtie R_4$ can be computed in a similar way.

The execution time of the level 2 join is given by Eq. (1) by substituting the following for R_1 and R_2.

B_1 should be the size of the $|(R_1 \bowtie R_2)|/bfr_{R1R2}$, which is given by

$$\alpha_{R1R2} * |R_1| * |R_2| / bfr_{R1R2}$$

B_2 should be the size of the $|(R_3 \bowtie R_4)|/bfr_{R3R4}$, which is given by

$$\alpha_{R3R4} * |R_3| * |R_4| / bfr_{R3R4}$$

The size of each relation $|R_1|$ and $|R_2|$ should be $\alpha_{R1R2} * |R_1| * |R_2|$ and $\alpha_{R3R4} * |R_3| * |R_4|$, respectively.

Substituting these values in Eq. (2) gives the approximate execution time to perform a four-way join.

Example

Consider four relations, each with 2000 tuples and a blocking factor bfr of 10. Thus, each file requires $\lceil 2000/10 \rceil = 200$ blocks. Assume that the join selectivities α for all joins is 0.1%. In addition, all blocking factors are assumed to be 10. The cost of performing $R_1 \bowtie R_2$ is the same as that of $R_3 \bowtie R_4$. This is given by

200 + (200 * 200) + 0.001 * 2000 * 2000/10 =

40,600 disk block accesses

Assuming that the mean disk block access time is 5ms, the execution time is

40,600*5ms = 203s

Number of tuples in $R_1 \bowtie R_2$ as well as $R_3 \bowtie R_4$ is

0.001 * 2000 * 2000 = 4000 tuples

and the number of blocks is 4000/10 = 400 blocks

Thus the cost of the level 2 join is

400 + (400 * 400) + 0.001 * 4000 * 4000/10 =

162,000 disk block accesses

The execution time of the this join is given by

162,000 * 5ms = 810s

The total execution time for performing the four-way join is

203 + 203 + 810 = 1,216s = 20.27 minutes

4.2.2. Using an additional workstation

If we use one additional workstation to exploit intra-query parallelism in the four-way join operation, we can use it as a slave that receives relations from the master and returns the results back to the master. Then we can perform the two joins (at level 1 in Figure 4) in parallel after sending the two relations to the slave. The computation of four-way join involves the following steps:

(i) Send the two required relations to the slave

(ii) Both master and slave work on the joins concurrently

(iii) Slave sends its join results back to the master

(iv) Master works on the level 2 join operation

Let us assume, without loss of generality, that $R_1 \bowtie R_2$ is performed on the slave workstation. This means that the master has to read the two relations from the disk and communicate them to the slave. The cost involved is equal to scanning these two relations once. Thus, the cost is B_1+B_2 disk accesses. Since communication is faster than disk read, we can assume that communication overlaps disk reads. Therefore, we do not have account for the communication cost associated with sending these two relations to the slave. Also, disk writes at slave can overlap. In any case, we will see that these costs are small when compared to the join costs and will not significantly affect the analysis. The time for preforming the first step is $(B_1 + B_2) * T_d$

The execution time for the joins $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$ (which are performed concurrently) is given before. As in the case of transmission of relations to the slave, sending the join results back to the master involves reading the result from the disk at the slave and transmitting it to the master. Again, we can assume that communication overlaps disk reads. Thus the time to perform this step is

$$(\alpha_{R1R2} * |R_1| * |R_2| / bfr_{R1R2}) * T_d$$

The final join cost is that same as that given before.

Example

For the example given in Section 4.2.1, we can compute the execution time as follows:

Time for step i = (200 + 200) * 5ms = 2s

Time for step ii = 203s

Time for step iii = 400 * 5ms = 2s

Time for the final join = 810s

Total execution time = 1017s = 16.95 minutes.

This represents an improvement of 16.4%. Even though the improvement is not substantial, we can get this for free as we are using only idle workstations. The reason for obtaining only a small improvement by using an additional workstation is that the final join is very time consuming. If we restrict ourselves to parallelism at operations level, we cannot do any better in this example! It is clear that having additional workstations will not help us as the query's maximum parallelism is 2. Notice that the overheads associated with data distributions are very small compared to the join costs. To gain further performance improvements, we have to exploit intra-operation parallelism, which we will discuss next.

4.3. Intra-Operation Parallelism

It is clear from the previous analysis that we can get even better performance if we can parallelize the join operation. We look at the impact of a parallel join operation on response time. To illustrate this idea, we use the parallel nested loop algorithm [2]. The algorithm is given below:

Input: R_1 and R_2

Output: $R_1 \bowtie R_2$ to be computed on n processors (one master and n-1 slaves)

i. Master broadcasts R_1 to all (n-1) slaves

ii. for j = 2 to n
Master sends the fragment R_{2j} to slave j

where $\{ R_2 = \bigcup_{j=1}^{n} R_{2j} \}$

iii. All n processors compute partial joins concurrently

iv. Each of the (n-1) slaves sends its partial join results back to the master

Cost of parallel join

The cost of step i is equal to number of disk blocks in R_1. As in the previous analysis, we assume that the network supports broadcast and it can be overlapped with disk reads. Thus, the

time required for this step is $B_1 * T_d$.

The second step involves sending a fragment of R_2 to each slave. Again, assuming that we can overlap communication with disk access, the cost the second step is $(n-1)*B_2/n$. Thus the time required for this step is $(n-1)*B_2 * T_d/n$.

The cost of performing the join operation (i.e., step iii) is

$$(B_2/n) + ((B_2/n) * B_1) + (\alpha_{R1R2} * |R_1| * |R_2|/n) / bfr_{R1R2}$$

The cost of the final step is

$$(n-1) * (\alpha_{R1R2} * |R_1| * |R_2|/n) / bfr_{R1R2} * T_d.$$

Example

Consider the final join operation given in the previous example. Without parallelization, the join operation requires 810s. We now recompute this value when we apply three additional workstations (i.e., n = 4 with one master and three slaves).

Time for step i = 400 * 5ms = 2s

Time for step ii = 3* 400 * 5ms/4 = 1.5s

Time for the partial join =

(100 + (100 * 400) + 0.001 * 2000 * 500/10) * 5ms = 201s

Time for the final step =

3 * (0.001 * 2000 * 500 /10)* 5ms = 1.5s

Total time to perform the final join is 206s. An improvement by a factor of 4. We get linear speedup in preforming the join operation because the overheads are relatively small.

5. PERFORMANCE ISSUES

We have demonstrated in the last section that using NOW for query processing is beneficial. There are, however, significant differences between parallel database systems discussed in Section 2 and NOW-based systems. 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 to work on. We have, however, shown that this overhead can be amortized if the query is complex. For example, join is the most common operation that is expensive in query processing.
- The workstations in a NOW-based system are heterogeneous in two aspects. First, the workstations could be of different types (ranging from low-end PCs to powerful workstations). In addition, the load on each workstation can be different. This is mainly because each user generates load independently on his/her workstation. In a parallel database system, processing modules are homogeneous and the load on each is controlled by a central scheduler.
- In a parallel database system, 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 networks. NOW-based systems use the general-purpose LANs, which are slow and involve high communication overhead.

5.1 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. In order to achieve load balancing on NOW-based systems, we have to use dynamic load balancing schemes. As an example, consider the intra-operation parallelism discussed in Section 4.3. The discussion in that section assumed no load balancing. As a result each node is sent |R2|/n tuples where n-1 is the number of slaves working on the operation. The number of tuples sent is often referred to as the task granularity. As we have indicated this type of equal distribution of work is not suitable for NOW-based systems. In these systems, the task granularity has to be adjusted depending on the load on each workstation and its speed. There are several ways this can be done.

- First, instead of sending all |R2|/n tuples to each workstation, we could send only a fraction of these tuples and send additional tuples only after the previous tuples have been joined. For example, if R2 has 2000 tuples, instead of sending 500 tuples to each of the three additional workstations, we could send 100 tuples each time a workstation requests work. This scheme is referred to fixed task granularity scheme. A key question is: what should be the size of each task? Too small a value would increase communication overhead and too large a value causes load imbalances that deteriorate in performance.
- Second, we can use variable task granularity to reduce the communication overhead. In this strategy, we start out with a coarse task granularity and gradually decrease the granularity. For example, if R2 has 2000 tuples, we can send 200 tuples of R2 to each of the three additional workstations that participate in the join operation. When a workstation is done with the 200 tuples, it will be given the next set of, say 150, tuples and so on. Several variable task granularity algorithms have been proposed for loop scheduling in multiprocessor systems. Some examples are the guided self-scheduling [17], trapezoidal self-scheduling [23], and factoring [14]. The advantage of these schemes is that, if a workstation is particularly slow, work can be diverted to other workstations dynamically.
- Last, we can develop learning-based algorithms that set the task granularity proportional to the amount time taken to the last task sent to the workstation.

We are currently investigating some of these load balancing issues.

5.2 Job Migration and Fault-Tolerance

In NOW-based systems, a user might want to take his/her workstation out of the pool of workstations used for database query processing. This could be because of high load on the workstation or for other reasons. In such instances, there are two options: either kill the task and reinitiate it on another workstation or migrate the task to another workstation. The first option involves duplication as the work done on the task is lost whereas task migration is expensive but preserves the work.

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

5.3 Communication Network Latency

Communication networks used in NOW-based systems are slow compared to the interconnection networks used in parallel database systems. Improved performance of communication subsystem in parallel database systems is due to better networks, low-overhead protocols and efficient switching methods. For example, the nCube-Oracle system is based on the nCube distributed-memory multicomputer system. This system uses hypercube as the interconnection network. Each communication

link in this system can support 2.75MB/s. The standard Ethernet is slower by a factor between 2 and 3. For parallel processing applications, slow communication network can seriously degrade performance of applications. For database applications, if the data to be transmitted is in main memory, communication network can have serious impact. If, on the other hand, the data is to be read from a disk unit, the communication network can only have a marginal impact on performance. We should also note that LANs are improving. For example, Fast Ethernet, Sonet-based ATM improve the bandwidth substantially. As a result, the parallel algorithms devised for parallel database systems will have to be reevaluated for the NOW-based systems.

6. SUMMARY

We have proposed the use of networks of workstations for database query processing. There is an increased overhead in such systems as the data will have to move from the database server to workstations. We have shown that, by providing a complete analysis of a four-way join, such overheads can be amortized over complex query operations that involve join operations.

We have also identified several problems associated with using NOWs for database query processing. These issues include load balancing, fault-tolerance, job migration, and communication network latency. For effective use of NOW-based systems, we will have address these issues. We have started preliminary work on some of these issues.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] T. E. Anderson, D. E. Culler, D. A. Patterson et al., "A Case for NPW (Networks of Workstations)," *IEEE Micro*, Vol. 15, No. 1, February 1995, pp. 54-64.

[2] D. Bitten, H. Boral, D. DeWitt, and W. K. Wilkinson, "Parallel Algorithms for Execution of Relations Database Operations," *ACM Trans. Database Systems,* Vol. 8, No. 3, September 1983, pp. 324-353.

[3] H. Boral, "Parallelism and Data Management," *3rd Int. Conf. on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[4] M. J. Carey and M. Livny, "Parallelism and Concurrency Control Performance in Distributed Database Machines," *Proc. ACM SIGMOD Conf.*, Seattle, Washington., June 1989, pp. 122-133.

[5] G. Copeland and T. Keller, "A Comparison of High-Availability Media Recovery Techniques," *Proc. ACM SIGMOD Conf.*, Seattle, Washington., June 1989, pp. 98-109.

[6] D. DeWitt et al. , "GAMMA - A High Performance Backend Database Machine," *Proc. 12th VLDB Conf.*, Kyoto, Japan, August 1986.

[7] D. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A Performance Analysis of the Gamma Database Machine," *Proc. ACM SIGMOD Conf.*, Chicago, Ill., June 1988.

[8] D. DeWitt and D. Schneider, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proc. ACM SIGMOD Conf.*, Seattle, Washington., June 1989, pp. 110-122.

[9] 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.

[10] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, ``A Message Passing Standard for MPP and Workstations," *Comm. ACM,* Vol. 39, No. 7, July 1996, pp.~84--90.

[11] T.-Y. Feng, "A Survey of Interconnection Networks," *Computer,* Vol. 14, No. 12, December 1981, pp. 12-27.

[12] G. A. Geist and V. S. Sunderam, ``Network-Based Concurrent Computing on the PVM System,'' *Concurrency: Practice and Experience*, Vol. 4, No. 4, June 1992,pp.~293--311.

[13] W. Gropp. E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass., 1994.

[14] S. F. Hummel, E. Schonberg, and L. E. Flynn, ``Factoring: A Method for Scheduling Parallel Loops," *Comm. ACM*, Vol. 35, No. 8, August 1992, pp.~90--101.

[15] B. C. Jenq, B. C. Twichell, and T. Keller, "Locking Performance in a Shared Nothing Parallel Database Machine," *IEEE Trans. Knowledge and Data Eng*, Vol.1, No. 2, 1989, pp. 530-543.

[16] M. Mutka and M. Livny, ``The available capacity of a Privately Owned Workstation Environment,'' *Performance Evaluation*, Vol. 12, No. 4, July 1991, pp. 269-284.

[17] C. Polychronopoulos and D. Kuck, ``Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers,'' *IEEE Trans. Computers*, Vol. C-36, No. 12, December 1987, pp. 1425--1439.

[18] *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratories,1993. (Manual and source code available via anonymous ftp from netlib2.cs.utk.edu in directory /pvm3).

[19] M. Snir, S. W. Otto, S. Huss-Lederman, D. Walker, and J. J. Dongarra, *MPI:The Complete Reference*, MIT Press, Cambridge, Mass., 1996.

[20] M. Stonebraker, "The Case for Shared Nothing," *Database Eng.*, Vol. 9, No. 1, March 1986.

[21] The Tandom Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction," *Proc. ACM SIGMOD Conf.*, Chicago, Ill., June 1988.

[22] Teradata DBC/1012 Data Base Computer Systems Manual, Release 1.3, Teradata Corp., Document No. C10-0001-00, February 1985.

[23] T. H. Tzen and L. M. Ni, ``Dynamic Loop Scheduling for Shared-Memory Multiprocessors,'' *Proc. Int. Conf. Parallel Processing*, Vol. II, 1991, pp. 247--250.

[24] Y. Xu and S. Dandamudi, ``A Hierarchical Parallel Database System for Transaction Processing," *Int. Conf. Computers and Their Applications*, Tempe, Arizona, March 1997. (Expanded version of this paper is available as technical report TR-97-2 from http://scs.carleton.ca.)