Parallel Query Processing in a Cluster using MPI and File System Caching

N. Ch. S. N. Iyengar , Monis Huda, Pranav Juneja, Saurabh Jain, V Vijayasherly,
School of Computing Sciences, VIT University
Vellore 632014, Tamil Nadu, India

Summary
Data intensive applications that rely heavily on huge databases waste a lot of time in searching and retrieval especially if there is a single server retrieving data from the database. This paper proposes a Beowulf cluster for fast query processing by distributing the database horizontally over nodes through a load balancing act. A mathematical model is proposed to optimally partition data among the nodes. Communication between nodes is to be achieved through MPI(Message Passing Interface).
A file system cache has been created to further decrease the query processing time. Caching is performed with the help of Apache Lucene API. Results would be retrieved depending upon a cache hit or miss. The size of the cache would be monitored and if it exceeds a threshold value deletion operation would be performed by applying the LRU(least recently used) algorithm. Through experimental results we have found that caching reduces the query processing time substantially. We can further improve the result by performing query optimization by indexing the attributes in complex queries.
This approach has reduced the query processing time manifold as compared to a single overloaded server. With networks growing in speed and highly available secondary storage it is expected to perform even better in future.
Key words:
Fast Query Processing, MPI, Load Balancing, File System Cache

1. Introduction

Due to high performance and cost effectiveness, cluster of workstations have gained popularity in recent years. A cluster can be built either from asymmetric or symmetric processors, but generally it is built from Symmetric Multi-Processors (SMP). SMPs with more than one node are also emerging. We are using Beowulf cluster in our approach which comes under compute cluster category and is a multi-computer architecture used for parallel computations. In order to provide efficient inter-node communication using MPI, the cluster should have a high performance and scalable architecture. In our implementation we have used MPI to provide effective communication between nodes.

The main motivation behind parallel processing application is that we need to solve bigger problems with resource requirements beyond current limits. The term bigger refers to applications that are performance critical, complex in nature and computation-intensive. Parallel computing is the way because it performs work in lesser time, solve large problem easily, saves cost and provides concurrency [9]. Data intensive applications that require huge databases waste a lot of time in scanning and searching. The optimal way to run these applications is to use the computational power of more than one system by distributing the workload among the nodes in a cluster. Thus by using extensive computational power of many nodes simultaneously the work can be performed in a very quick and efficient way. Traditional serial computation has many serious limitations like memory size and speed, limited instruction level parallelism, power usage, heat problem etc. With the wide availability of parallel computing platforms like HPC centres, local linux clusters, multiple CPU's and GPU's(graphics processing unit) the above mentioned limitations can be overcome.
Our objective is to optimize query processing time by providing parallelization with the use of Java MPI over distributed database systems to store data rather than overloading a single DB Server machine. In case of a single DB server machine if the database to be processed is huge, it suffers from various limitations like memory size and speed. Our aim is to share the work of the single server machine by partitioning the load among the nodes in the cluster so that parallelization can be achieved. Query processing time is further reduced by adding an important functionality of caching, in which a file system cache would be maintained so that in order to make the process fast, the result would be first retrieved from the cache if present instead from the database and if it is not present it would be written in cache for future references [7]. To provide efficient searching Jakarta Lucene search engine version 2.4 is used [3], [4]. We have further reduced query

Manuscript received May 5, 2009
Manuscript revised May 20, 2009

processing time by creating index of attributes in complex queries.
The rest of the paper is organized as the following: In Section 2, we introduce some background knowledge, including advanced system architectures, MPI inter-node communication, Lucene search engine and file system caching. We illustrate our design and its methodology in Section 3. Performance evaluation, comparison and analysis are presented in Section 4. And finally, in Section 5 we conclude and point out future work directions.

2. Background

MPI is a message-passing application programmer interface used to program parallel computers with the help of a set of library routines which are used for distributed programming [5], [6]. It is language independent and supports both point-to-point and collective communication. Its goals are high performance, scalability, portability, and it is a dominant model used in high-performance computing today. Most MPI implementations are callable from any language capable of interfacing with routine libraries like Fortran, C, C++ or Java. The advantages of MPI over older message passing libraries are portability and speed. There are two standard versions that are currently in use version 1.2 (also known as MPI-1) which has a static runtime environment and MPI-2.1 which includes new features such as parallel I/O, dynamic process management and remote memory operations. Another difference is in the shared memory which is supported by MPI-2 only. While MPI-1 is used to exchange messages of one datatype MPI-2 provides additional functions of one way communications and language bindings. This message passing model which is based on data exchange between processes in the MPMD model provides the programmer with the advantage of high flexibility in the explicit parallel programming.

.

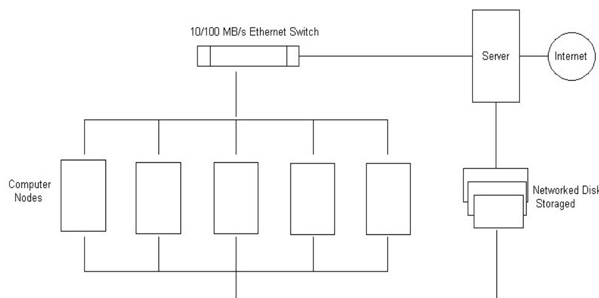


Figure1 : Beowulf Cluster

Figure 1 illustrates a typical Beowulf cluster built from SMPs. Threaded MPI execution on SMP clusters can be done in two ways, Intra Machine Communication through shared memory [2] and Inter Machine Communication through network. We are using inter node communication in our approach. It is a common belief that inter node communication is dominated by network delay, so the advantage of executing MPI nodes as threads diminishes but recent findings as described in [1] have shown that using threads can significantly reduce the buffering and orchestration overhead for inter machine communications.

In our file system caching we are using Lucene. It is an open source API supported by Apache Software Foundation originally created in java by Doug Cutting. It is suitable for any application which requires indexing and searching operations. In our approach master node is using these functionalities to create and maintain file system cache.

We are using database indexing also to reduce query processing time. A database index is a data structure that improves the speed of operations on a database table. Popular data structures used for indexing are balanced trees, B+ trees and hash tables. Indexing more than necessary can result in an application slower than usual so it has to be used in a proper manner to enhance performance.

3. Design and methodology

In this section, we provide a detailed illustration of our proposed design. Our design goal is to develop distributed system architecture for fast query processing. In the following subsections, we start with our proposed work, followed by overall design architecture and its implementation.

A. *Proposed Work*

The system architecture that is generally used has a Database (DB) Server solely responsible for the entire database. The client system interacts with the server through the user interface and requests the data records. The DB server which stores and manages the entire database retrieves the records, sends it back to the client and present it through the user interface. In this case the database is centralized and the entire overhead of the database is on the DB server. Therefore, the DB Server should have adequate resources to comply with the requirements of the clients and database of different sizes.

In our proposed system we are emphasizing the use of distributed database instead of a centralized one. Our aim is to reduce the query processing time significantly so

that result can be displayed in real time. Database is distributed by performing a load balancing act by the master node in the cluster. Master node partitions the database horizontally and distributes it equally to all the nodes present. Communication between nodes is to be achieved through MPI(Message Passing Interface) middleware which is a language-independent communication protocol used to program parallel computers. MpiJava is an object-oriented Java interface to the standard Message Passing Interface which we are going to use. User would enter the query at the master node and master node would send query to each of the nodes in the cluster. Slave nodes would execute query and send the result back to the master node where all the result would be assimilated and displayed. Further reduction in query processing time is to be achieved using file system cache. We are going to use apache lucene API to perform caching. Lucene creates index for the cache and performs searching operations. When query is entered by the user at the master node it would first search the result in the file system cache. If it is a hit the result would be retrieved from the cache and if it is a miss the result would be appended in form of a file in cache for future references. The size of the cache would be monitored and if it exceeds some threshold value deletion operation would be performed by applying an algorithm(we are using LRU). We have further reduced query processing time by creating indices of attributes in case of complex queries.

B. Overall Architecture

In the conventional system as shown in figure 2 there is only a single DB server retrieving data from the database. Figure 3 demonstrates our proposed architecture which comprises of a Beowulf cluster with the master node connected to the outside network and the slave nodes connected to the master. There is distributed database architecture with the master node distributing the load to all the nodes present. In the master node we have created a file system cache to enable faster retrieval of data.

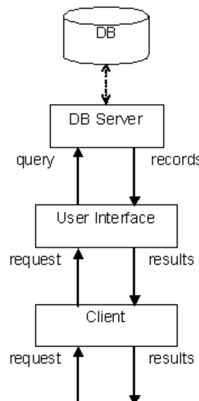


Figure 2: Existing system

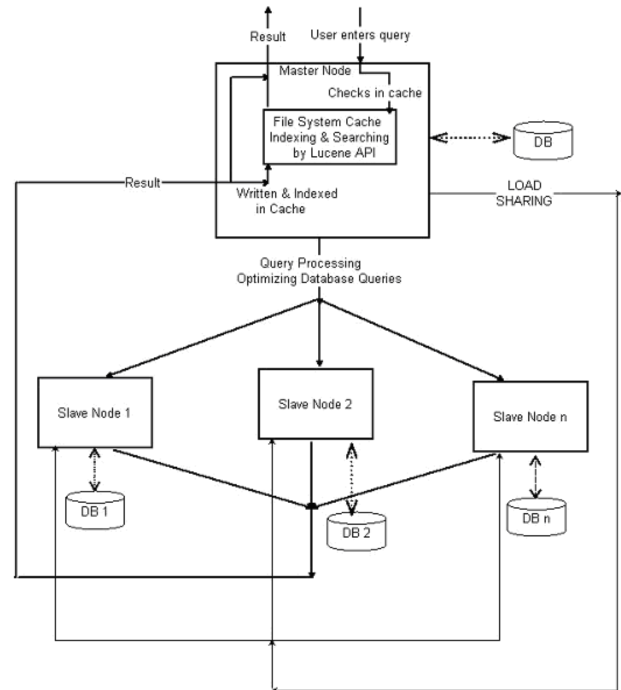


Figure 3: Proposed System

The master node would retrieve the data from the cache if present and if not, the result would be appended in cache for future references. The size of the cache would be monitored and only recently queried data would be present. The searching in file system cache is performed using Lucene search engine.

C. Mathematical Model for Load Balancing

In a heterogeneous cluster many factors have to be taken into account [8] to decide how much task is to be allotted to each node. These factors include processor speed, disk storage, input/output and network latency. While in case of homogeneous cluster we have to only take network latency into account. Distribution of database has to be done according to the execution time taken by a particular node, less the execution time more the data given to that node. We can present load sharing in a mathematical model as follows:

Let D be the database size and S be a set of heterogeneous machines; $S = \{N_1,N_2,\ldots,N_n\}$, where N denotes a machine and n be the number of nodes.

$N_i(C_i,S_i,L_i)$ represents a machine having computational power C_i, disk storage Si and estimated network bandwidth L_i; $L_i=\{\alpha_{ij},\forall\ i \neq j\}$, where α_{ij} is the average latency between nodes N_i and N_j.

Let E_i be the execution time taken by the node N_i. This can be written as:

$E_i = C_i + I_i$, where

C_i = estimated computational time

I_i = estimated I/O time

The partitioning of the database can be carried out in three steps:

1) Estimating the number of nodes in which the database has to be partitioned: To optimize the performance of the cluster all the nodes need not to be used. The number of nodes that should be used for better performance can be estimated by using the following formula:

$$T_m = max[\ C*(D/m) + I_i*\alpha_{ii}*(D/m*\alpha_{ij})]$$

where $1 \leq i \leq m$, $1 \leq m \leq n$,

T_m denotes the execution time taken by m nodes

C is average computational time

Now to find the maximum number of nodes m, we have to find the point where derivative becomes zero. We can denote it by n_{min}

$$n_{min} = \{\ m : d/dM\{max[\ C*(D/m) + I_i*\alpha_{ii}*\ (D/m*\alpha_{ij}\ )]\}=0\ \}$$

where, n_{min} is the minimum number of nodes to be used out of n.

2) Finding the rank of each node: The next step is to find the rank of each node and altering nodes.txt file according to the rank and number of nodes which are found in the first step. In the nodes.txt file hostname of slave nodes would be written in the order of their rank. The rank of each node would depend on its computational power and network latency. Rank would be calculated using this formula:

$$R_i = C_i + (\ I_i\ /\alpha_{ii}\ )*\alpha_{ij}$$

where R_i denotes the rank of the ith node.

3) Partitioning database: The database to be distributed is partitioned according to the rank of the node, that is, more the value of Ri less the part of database will be given to it. For example if there are three nodes having rank 1, 2, 3 database would be distributed in the ratio 3/6, 2/6 and 1/6 respectively.

D. Implementation

The idea has been implemented through the following modules:

1) Load Sharing: The aim of the module is to perform load sharing through database distribution. The mathematical model for load sharing has been described above. The flow is shown in figure 4. To avoid direct access to the slave node and individual entering of records into the data base, the data can be directly entered via master node. The records are retrieved from the master node and then are partitioned depending on the number of nodes present in the cluster. The data is now sent to the slave nodes in the form of an object using MPI. The slave receives the object and updates its database.

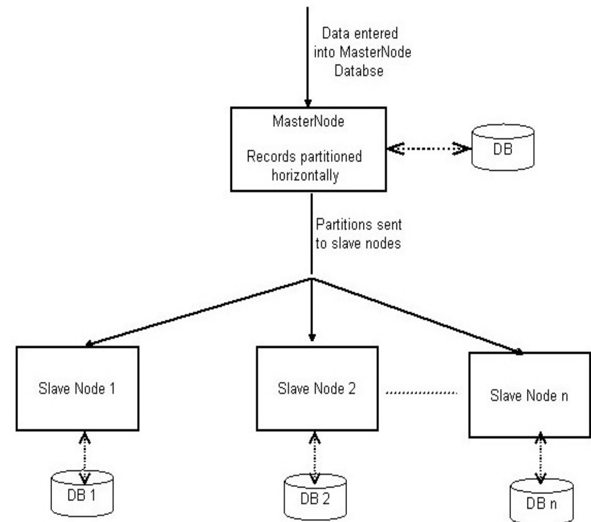


Figure 4: Load Sharing

2) MPI (Message Passing Interface): This interface means that programs can communicate between instances of themselves and other programs on remote nodes to achieve efficient parallelism and minimize the overhead associated with process migration when the load is inaccurately predicted.

The Slave nodes establish TCP/IP connection with the master node by dialing the host name. The Master node accepts the request from all the nodes based on the entries in the nodes.txt file. Once the connection is established, the slave nodes are ready to accept the query from the master node.

3) File System Caching: We are using cache in the form of file system cache which is different from in memory cache. This cache is present in the secondary storage itself in the form of directory at a specified location. So in our case its non volatile and its persistent. The attributes of the cache like cache size, expiry can be set accordingly. In our implementation the recently retrieved records are stored in the cache in form of files. The data if present in the cache itself is taken out rather than retrieving from the database. A cache might backend to a file system to load and retrieve objects locally rather than across a network. Once the size of cache exceeds certain fixed limit the least recently used files are automatically deleted before the creation of the new ones.

Lucene gets its data from a file system directory that contains a certain set of files that follow a certain structure. The data is sent into the Index, after that searching is performed on the Index to get results out. Document objects are stored in the Index, and they are put into the Index at some point. We select what data to enter in, and convert them into Documents. We read in each data file (or database entry, or whatever), instantiate a Document for it, break down the data into chunks and store the chunks in the Document as Field objects.

4) Indexing Database Queries: A database index is a data structure that can be created using one or more columns of a database table and its main advantage is that it improves the speed of operations on a database table, providing the basis for both rapid random look ups and efficient access of ordered records. Since indexes do not consist of the details that are present in the table and contains only the key fields according to which the table is arranged, it occupies much lesser space than the table and thus provides the chance of storing the index in memory of those tables which are too large.

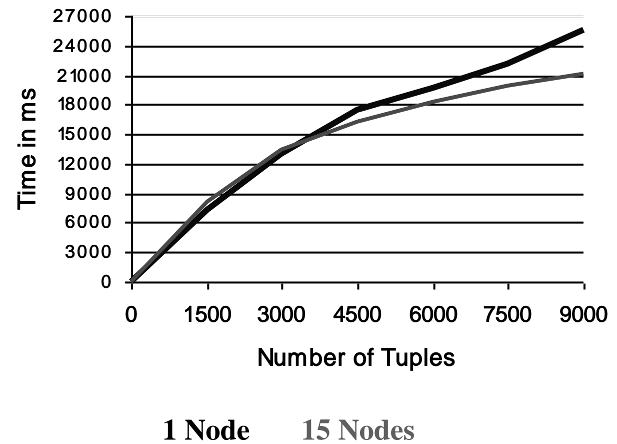
Whenever the user enters a complex query, an index containing the attribute as columns present in the query is created. Now the next time when the user enters a query containing similar attributes, the index is first accessed rather than the table and thus leading to time optimization.

4. Performance Evaluation

In this section, we present the performance evaluation of our proposed system.

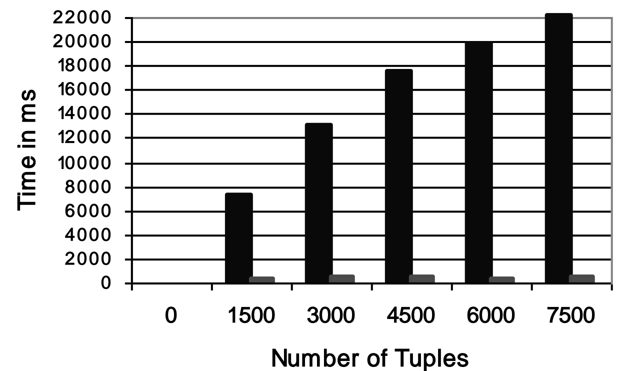
Experimental Setup: We created a Beowulf cluster comprising 15 nodes. Each node is equipped with Intel Pentium4 processor running at 1.66GHz. Slave nodes consist of 1024kb L2 cache while the master node is having 2048kb size. The nodes are connected by high speed LAN. The MPI version used is 1.2.5 and the Lucene version on the master node is 2.4. Oracle 10g is used as database server. The operating system on the cluster is Windows XP.

We compared the performance of our cluster to the single server design. Query retrieval time is measured in unit of milliseconds (ms). First we compared the query retrieval time of a single node and 15 nodes fetching the same number of tuples. It was assessed that when the number of records were less, the performance of single node was better than 15 nodes fetching results simultaneously. But as the number of tuples increased the time taken by the distributed system decreased substantially as shown in figure 5. We found out that query processing is highly dynamic in nature and depends upon the size of the database and on the number of nodes present in the cluster.



1 Node **15 Nodes**

Figure 5: Difference in latencies between single node and cluster



Query Processing time when tuples retrieve from slave nodes (Cache MISS)

Query Processing time when tuples retrieve from file system Cache (Cache HIT)

Figure 6: Difference in latencies between file system cache and cluster

We also compared the performance and query retrieval time when records are fetched from slave nodes and local file system cache as shown in figure 6. Here we found drastic reduction in time. The time taken to retrieve results from slave nodes is expected to be proportional to the number of tuples, but the time was found out to be nearly constant when retrieved from local file system cache because of fast searching performed by lucene.

We further analyzed the performance based on the number of nodes in the cluster keeping the number of tuples constant as demonstrated in figure 7. It was found that when the number of nodes was less, the performance

was low, but when it was increased to 25 or above there was a significant enhancement in the cluster performance.

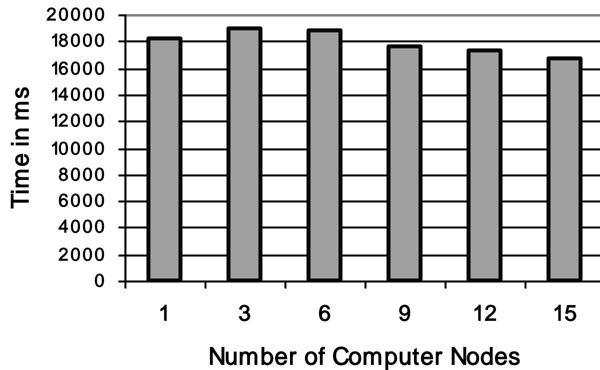


Figure 7: Difference in latencies with increasing nodes in cluster

5. Conclusion and future work

In this paper, we have designed and implemented a high performance and scalable inter node communication within a cluster for fast query processing in a distributed database environment using MPI in java. Master-node file caching has been demonstrated as a powerful tool to reduce the data transfer between master node and the slaves. A mathematical model for load sharing is proposed to distribute the data optimally to all the nodes present and since it is performed by the master node it also prevents direct access to slave nodes which is very crucial in respect of security. Our experimental results show that our proposed design and work have substantially reduced query processing time.

For future works, we will perform both vertical and horizontal partitioning of the database. Also, we intend to encrypt the data before transferring it over the network to provide secured transmission.

Acknowledgment

We would like to thank Dr. M. Khalid, Director, School of Computing Sciences and management of the VIT University, India for providing facilities to test our design in labs.

References

[1] Hong Tang and Tao Yang, Dept of Computer Science, University of California, "Optimizing Threaded MPI Execution on SMP Clusters", 2001
[2] Lei Chai ,Albert Hartono ,Dhabaleswar K. Panda, "Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters", 2006 I EEE.
[3] Lucene in Action, Second Edition" by Erik Hatcher, Otis Gospodnetić, and Michael McCandless.
[4] Building Search Applications: Lucene, LingPipe, and Gate" by Manu Konchady; Mustru Publishing; June 2008.
[5] Parallel Programming with MPI by Peter Pacheco
[6] Using MPI: portable parallel programming with the message passing interface by William Gropp, Ewing Lusk, Anthony Skjellum
[7] Wei-keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, and Alok Choudhary , "Using MPI File Caching to Improve Parallel Write Performance for Large-Scale Scientific Applications" 2007 ACM.
[8] Sanan Srakaew, Nikitas A. Alexandridis, Punpiti Piamsa-nga, George Blankenship, "Content-based Multimedia Data Retrieval on Heterogeneous System Environment," in International Conference on Intelligent Systems (ICIS-99) , Denver, Colorado, June 24-26, 1999.
[9] Philip Hatcher And Mathew Reno, "Cluster computing With Java" 2005 IEEE