

Query Execution in Distributed Text Databases

Prajakta S. Kalekar (04329008)
KReSIT, IITB

Abstract—The performance of distributed text document retrieval systems is strongly influenced by the organization of the index structures. This paper introduces the various distributed index organization schemes and the algorithms for processing in parallel batches of queries using these organizations. This paper also compares the impact of various factors such as disk I/O and network resources on query processing for the various physical organizations of the induces.

I. INTRODUCTION

The number and the size of text collections readily available in electronic form today is staggering. At the same time, there is a rapid increase in the number of users of and queries submitted to text retrieval systems. As the data volume and query processing loads increase, it becomes more and more important to have efficient information retrieval systems, which are able to provide multiple users with concurrent and efficient access to multiple text collections.

A solution to this problem is to explore parallel and distributed techniques, which can greatly enhance the ability to scale traditional information retrieval algorithms and support larger and larger number of documents and queries.

In the distributed computing approach, multiple computers connected by a network are used to solve a single problem. Some of the advantages of the network of workstations model are as follows: Firstly, networks of workstations have become extraordinarily powerful and offer a better price-performance than parallel computers. Secondly, most networks of workstations have a huge amount of memory and very fast processors, both of which sit idle most of the time. Hence, distribution of the data (in this case text documents and index structures) as well as processing (in this case search for keywords to be retrieved) among a number of processors can make use of the idle CPU cycles of the processors. Thirdly, switched networks allow bandwidth to scale with the number of processors and low overhead communication protocols have made it possible to do very fast communication among workstations.

The goal of this paper is to study parallel query processing and various distributed index organizations for information retrieval. The query processing algorithms are based on the *Block Synchronous Model for Parallel Computing* [1]

II. INDEX STRUCTURES FOR DISTRIBUTED TEXT DATABASES

This section will introduce two of the most commonly used Distributed Text Databases Index structures:

A. Inverted Index

1) *Basic Concept*: Each index entry gives the word and a list of documents, possibly with locations within the document, where the word occurs.

2) *Distributed Algorithm for generation of Inverted Index*:

1) *LL Algorithm: Local Buffer and Local Lists*

The LL algorithm works as follows.

- *Phase 1: Local Inverted Files*

In this phase, each processor builds an inverted file for its local text and stores it on the local disk.

- *Phase 2: Global Vocabulary*

In this phase, the global vocabulary and the portion of the global inverted file to be held by each processor are determined. For this, the processors are paired as illustrated in Figure 1. At the end, processor p_0 results with the global vocabulary.

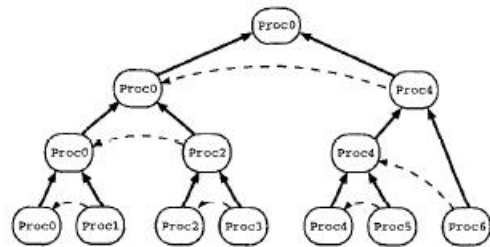


Fig. 1. Global vocabulary computation with 7 processors

- *Phase 3: Global Distributed Inverted File*

In this phase, portions of the local inverted files (as determined in phase 2) are exchanged to generate the global inverted file. For this, an all-to-all communication procedure is used. The number of rounds of processor pairings required is $p - 1$. Each portion received by a processor p_i (from every other processor p_j) is stored on disk as a separate run. As a result, each processor ends up with p separate runs on disk. At the end, a p -way merging on disk is executed at each processor to generate the final inverted lists which compose the global inverted file.

2) *LR Algorithm: Local Buffer and Remote Lists*

In the LL algorithm, when the memory buffer B fills up the partial inverted lists are stored on a local disk. As a result, those inverted lists have to merged locally through an R -way merging procedure (at the end of phase 1). Later on, at the end of phase 3, a second

multi-way merging procedure is executed (this time a p-way merging) to merge the lists received from other processors. The first one of these two multi-way merging procedures (and its resultant I/O costs) can be avoided entirely if we store the partial inverted lists directly at a remote disk (instead of at a local disk). By doing so, we save roughly half of the I/O costs associated with two separate merging procedures. This is the main idea behind the LR algorithm.

The LR parallel algorithm uses a local memory buffer B as before but sends the partial inverted lists (at the time the buffer B fills up) to other processors for storage at remote disks. This implies that the cost of storing the partial lists at local disks is saved. It also means that information on the global vocabulary must be available early on (because a processor p_i needs to know the destination of each inverted list to be sent out). In this algorithm processors have to be synchronized. This means that there is an extra hidden cost because the processors must wait for the slowest processor before proceeding.

3) RR Algorithm: Remote Buffer and Remote Lists

The LR algorithm stores the triplets $[k_i, d_j, f_i, j]$ in a local buffer B, assembles partial inverted lists in this buffer, and then sends these lists out. A potential improvement is to assemble the triplets in small messages early on and to send these messages out very soon to avoid storage at the local buffer B. By doing so, the buffer B is assembled remotely. As a result, the pR-way merging used in the LR algorithm is substituted by an R-way merging. A distinct message for every other remote processor p_j is assembled separately. These messages should be large enough to avoid penalties due to network overhead. Further, through proper programming of two threads (one for disk access and other for network access), it is possible to overlap the transmission through the network with the reading of local documents from disk. This implies that transmission of the inverted lists through the network comes at very low cost. In the RR algorithm, the buffer B is remote and the partial inverted lists are stored on remote disks. The algorithm is a slight variation of the LR algorithm and is not detailed here.

B. Suffix Arrays

1) *Basic Concepts:* Suffix Array is an array containing all the pointers to the text suffixes sorted in lexicographical (alphabetical) order. Each suffix is a string starting at a certain position in the text and ending at the end of the text. Searching a text can be performed by binary search using the suffix array.

2) *Distributed Algorithm for generation of Suffix Arrays:* The algorithms used for generation of Inverted Indices can also be used for generating Suffix Arrays, by modifying the technique of creating the Index, without modifying the distributed algorithm part.

III. AN EXAMPLE DISTRIBUTED TEXT DATABASE

The following scenario will be used for illustration purposes: Consider four sample documents, D0, D1, D2, D3, that could be stored in an information retrieval system.

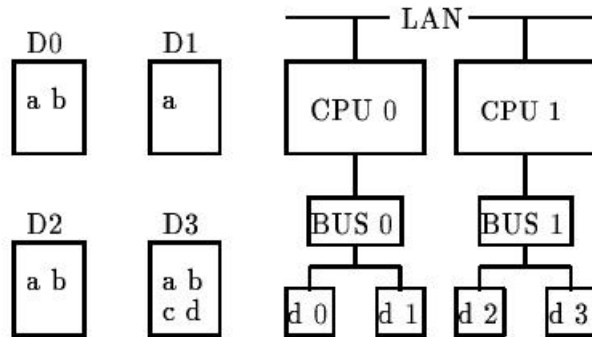


Fig. 2. An example set of four documents and an example hardware configuration

A. Nature of documents

Each document contains a set of words (the text), and each of these words will be used to index the document. In Figure 2, the words are shown within the document box, e.g., document D0 contains words a and b. This is a simple example, in practice documents are significantly larger and contain many more words.

B. Nature of Index structures

To find documents quickly, index structures such as inverted lists [4] or suffix arrays [2] are built and stored on disk. For instance, the inverted list for word b would be b: (D0,1), (D2,1), (D3,1). Each pair in the list indicates an occurrence of the word (document id, position). To find documents containing word b, the system only needs to retrieve this list. To find documents containing both a and b, the system could retrieve the lists for a and b and intersect them.

C. System Architecture

Suppose that we wish to store the inverted lists on a multiprocessor like the one shown in Figure 2. This system has two processors (CPUs), each with a disk controller and I/O bus. Each CPU has its own local memory. Each bus has two disks on it. The CPUs are connected by a local area network.

IV. INDEXING SCHEMES FOR DISTRIBUTED TEXT DATABASES

Broadly, there are two distinct Indexing techniques for text databases, namely *Local Indexing* and *Global Indexing*.

A. Global Indexing

1) *Index Structure*: In Global indexing, an inverted file (or Suffix Array) is generated for all the documents in the text database and the Inverted lists (or Suffix Arrays) are distributed among the different processors. The entire vocabulary is split into ranges for example words beginning from a-e lie in one range, f-j in the next range and so on. Each range of the vocabulary is assigned to a different processor. Hence a processor maintains a local index of the words belonging to the range assigned to it.

Apart from the local indices maintained by the individual processors, a broker machine maintains information of the intervals assigned to each processor and routes queries to the processors accordingly. This fact can be the source of load imbalance in the processors when queries tend to be dynamically biased to particular intervals. Section V-A explores ways of overcoming this problem.

2) *Query Processing Algorithm*: Let us assume that we are interested in determining the text positions in which a given substring x (of length T) is located in. This can be achieved using index structures like Suffix Arrays.

Let us assume the ideal scenario in which the queries are routed uniformly at random onto the processors. A search for all text positions associated with a batch of $Q = q * P$ queries can be performed as follows. The broker takes routes the queries to their respective target processors. This is one *super-step* of the BSP model[6]. In order to route the queries to the appropriate processor, a binary search is performed on the *Global Index* Once the processors get their q queries, in parallel each of them performs q binary searches.

Then the q array positions per processor are received by the broker to continue with the following batch and so on. However, it is not necessary to wait for a given batch to finish since the broker can start the processing of a new batch as soon as the queries for a batch are routed to the respective processors, that is after each superstep.

This forms a pipelining across supersteps in which at any given superstep we have, on average, $\log(N = P)$ batches at different stages of execution. The net effect is that at the end of every superstep we have the completion of a different batch.

A binary search on the global index approach can lead to a certain number of accesses to remote memory. To overcome this problem, a cache scheme can be implemented so as to keep in p the most frequently referenced strings from remote memory (i.e., those close to the root of the global binary search virtual tree).

B. Local Indexing

1) *Index Structure*: In the local index strategy, an Index Structure is constructed in each processor by considering only the subset of text stored in its respective processor. No references to text positions stored in other processors are made.

2) *Query Processing Algorithm*: For every query it is necessary to search in all of the processors in order to find the pieces of local arrays that form the solution for a given

interval query. As answers for interval queries, it is necessary to send to the broker QP pairs (a; b), Q per processor, where a/b are the start/end positions respectively of the local arrays. The processing of a batch of Q queries is as follows. Let us charge 1 unit to the handling of each query by the broker so it first does a work proportional to $Q = qP$. However, the broker now has to send every query to every processor.

That is, the processors get q queries from the broker and then broadcast them to all other processors. This means that the broker itself does not broadcast all the Q queries to all the processors, but instead, in the first superstep broadcasts one query q to each processor, after which each processor broadcasts the query it received to all the other processors, so that at the end of this step, all processors have all Q queries.

In the next superstep, each processor performs in parallel Q binary searches and sends Q pairs (a; b) to the broker The broker, in turn, receives QP queries.

V. INDEX ORGANIZATION SCHEMES

Following options for organization of the indices are available:

Index	Disk	Inverted Lists
Disk	d 0	a:(D0, 0); b:(D0, 1)
	d 1	a: (D1, 0)
	d 2	a: (D2, 0); b: (D2, 1)
	d 3	a: (D3, 0); b: (D3, 1);c: (D3, 2); d: (D3, 3)
Host, I/O bus	d 0	a: (D0, 0), (D1, 0)
	d 1	b: (D0, 1)
	d 2	a: (D2, 0), (D3, 0); c: (D3, 2)
	d 3	b: (D2, 1), (D3, 1); d: (D3, 3)
System	d 0	a: (D0, 0), (D1, 0), (D2, 0),(D3, 0)
	d 1	b: (D0, 1), (D2, 1), (D3, 1)
	d 2	c: (D3, 2)
	d 3	d: (D3, 3)

TABLE I
INDEX ORGANIZATIONS FOR INVERTED LISTS

- 1) *System Index Organization* In the System Index Organization, a Global Index for all documents of the text database is built. This Global Index is then split evenly across all the disks in the system. For example, the inverted list of word b discussed earlier happened to be placed on disk d1. Consider the query "Find documents with words a, c", and say the query initially arrives at CPU 0. Under the system index organization, CPU 0 would have to fetch the list for a, while CPU 1 would fetch the c list. CPU 1 would send its list to CPU 0, which would then intersect the lists. i
- 2) *Disk Index Organization* With the disk index organization, the documents are logically partitioned into a number of sets, one for each disk. Each CPU then builds indices for those documents that are assigned to the disks under it. Hence this is an example of Local Indexing.

For the example mentioned above, assume document D0 is assigned to disk d0, D1 to d1, and so on. In each partition, inverted lists for the documents that reside there are built. To now answer the query “Find all documents with word b” we have to retrieve and merge 4 lists, one from each disk. (Since disk d1 contains no documents with word b, its b list is empty.)

- 3) Host Index Organization In the Host Index organization, documents are partitioned into separate groups, one for each CPU. This organization is an example of the Local Indexing scheme, since each processor builds indices for the documents stored by it.

In our example scenario, it is assumed that documents D0, D1 are assigned to CPU 0, and D2, D3 to CPU 1. Within each partition again inverted lists are built. The lists are then uniformly dispersed among the disk attached to the CPU. For example, for CPU 1, the list for a is on d2, the list for b is on d3, and so on.

A. System Index Organization

1) *Index Partitioning*: In the System Index Organization, the Global Index needs to be partitioned among the different processors/disks. There are a number of ways in which this can be done:

- Interval partitioning

In this scheme, the entire vocabulary is divided into a number of intervals. The index of each of these intervals is assigned to a different processor. The broker/host machine with which the query is actually submitted maintains information regarding the intervals and their assignment to processors. The drawback of this scheme is that it only takes into account the length of the intervals while partitioning (each interval is of approximately the same length), and not the size of the intervals in terms of the number of words that belong to the interval. This may lead to load imbalance due to large and sustained queries being routed to the same processor.

- Multiplexing Strategy

This scheme provides a solution to the load imbalance problem of the above scheme. In this scheme, the words of the vocabulary are multiplexed across the processors. For example, if array element i is stored in processor p , then elements $i + 1, i + 2, \dots$ are stored in processors $p + 1, p + 2, \dots$ respectively, in a circular manner. In this case, any binary search can start at any processor. Once a search has determined that the given term must be located between two consecutive entries k and $k + 1$ of the array in a processor, the search is continued in the next processor and so on, where at each processor it is only necessary to look at entry k of its own array.

For example, in Figure 3 a term located in the first interval, may be located either in processor 1 or 2. If it happens that a search for a term located at position 6 of the array starts in processor 1, then once it determines that the term is between positions 5 and 7, the search is continued in processor 2 by directly examining position 6.

In general, for large P , the inter-processors search can be done in at most $\log P$ additional supersteps by performing a binary search across processors.

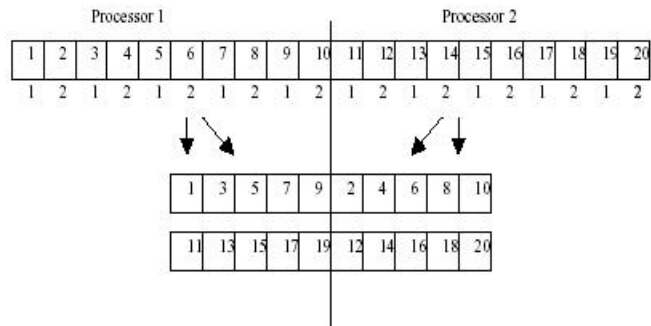


Fig. 3. Multiplexing the global index array entries

- Hybrid strategy

An intermediate strategy between Index Partitioning and Multiplexing can be obtained by considering the global array as distributed on $V = 2^k P$ virtual processors with $k > 0$ and that each of the V virtual processors is mapped circularly on the P real processors using $i \bmod P$ for $i = 0 \dots V$ with i being the i^{th} virtual processor. In this case, each real processor ends up with $\frac{V}{P}$ different intervals of $\frac{N}{V}$ elements of the global array. This tends to break apart the imbalance introduced by biased queries. Calculation of the array positions are trivial. The broker machine routes queries uniformly at random to the P real processors, and in every processor a $\log P$ ($\log V$) binary search is performed to determine to which processor to send a given query (this is done in order to avoid the broker to be a bottleneck). Once a query has been sent to its target processor it cannot migrate to other processors as in the case of the Multiplexing Strategy. That is, this strategy avoids the inter-processors $\log P$ binary search.

- 2) *Pros and Cons*:

- Disk I/O

With the system index organization, there are fewer I/Os: the *alist* is stored in a single place on disk. To read it, the CPU can initiate a single I/O, the disk head moves to the location, and the list is read in (this may involve the transfer of multiple blocks).

- Network resource consumption The system index organization may save disk resources, but it consumes more resources at the network level. In our example, the entire *clist* is transferred from CPU 1 to CPU 0, and we can expect these inverted lists to be much longer than the document lists exchanged under the other schemes. However, the long inverted list transfers do not occur in all cases. For example, the query “Find documents with a and b” does not involve any such transfers since all lists involved are within one computer. Also, it is possible to reduce the size of the transmitted inverted lists by moving the shortest list. In the “Find documents with a and c”

example, we can move the shorter list of a and c to the other host.

3) *Prefetching*: In order to address the problem of consumption of resources at network level, Prefetch I, II and III optimization techniques may be used.

In the Prefetch I algorithm, the broker machine determines the query keyword k that has the shortest inverted list. (We assume that hosts have information on keyword frequencies; if not, Prefetch I is not applicable.) In Phase 1, the broker machine sends a single sub-query containing k to the host that handles k . When the broker machine receives the partial answer set, it starts phase 2, which is the same as in the un-optimized algorithm, except that the partial answer set is attached to all sub-queries. Before a host returns its partial answer set, it intersects it with the partial answer set of the phase 1 sub-query. This significantly reduces the size of all partial answer sets that are returned in phase 2.

The Prefetch II algorithm is similar to Prefetch I, except that in phase 1 we send out the sub-query with the largest number of keywords. We expect that as the number of keywords in a sub-query increases, its partial answer set will decrease in size. Thus, the amount of data returned by the one host that processes the phase 1 sub-query should be small. If there is a tie (two or more sub-queries have the same number maximum of keywords), Prefetch II selects one of them at random.

Prefetch III combines the I and II optimizations. That is, the first sub-query contains the largest number of keywords, but if there is a tie, the sub-query with the shortest expected inverted lists is selected.

Intuitively, one would expect Prefetch III to perform the best. However, Prefetch I and III require statistical information on inverted list sizes.

To illustrate these optimizations, consider the query a and b and c and d in the example of Figure 2

With Prefetch I, the sub-query d would be sent to host CPU 1 in phase 1. (Of the four keywords, d occurs less frequently in the database, and it is stored in host CPU 1.) In phase 2, the sub-query a and b would be sent to CPU 0, together with the list for d from phase 1. CPU 1 would receive the query c together with the d list.

With Prefetch II, the first sub-query would be either a and b (to CPU 0) or c and d (to CPU 1), selected at random.

Prefetch III would select c and d as the first sub-query because it involves shorter lists.

Experimental results have indicated that prefetch II and III actually perform worse than prefetch I. Prefetch II and III would reduce the amount of data sent over the LAN. However prefetch II and III is performed sequentially with respect to the rest of the processing of the query, leading to longer response times. Thus, only in cases where the LAN is a bottleneck would prefetch II and III pay off.

B. Disk Index Organization

1) *Pros and Cons*: In the disk index organization, the list for a particular word in the vocabulary maybe stored across several disks. To read these list fragments, a number of I/Os

must be initiated, a number of disk seeks must happen, and a number of transfers must take place.

However, each of the transfers is roughly a fourth of the size, and all transfers may take place in parallel. So, even though we are consuming more resources (more CPU cycles to start more I/Os, and more disk seeks), the list may be read into memory faster.

C. Host Index Organization

1) *Pros and Cons*: For executing the query “Find all documents with word b ”, each CPU would find the matching documents within its partition. Thus, CPU 0 would get its a and c lists and intersect them. CPU 1 would do likewise. CPU 1 would sent its resulting document list to CPU 0, which would then merge the results. Hence the amount of network resources that are consumed is less as compared to System Index Organization (without Prefetch optimization). However, if disk access is a bottleneck, then this approach would perform poorly, since a and c lists need to be fetched from both the processors.

D. Comparison of the Index Organization schemes

- Sensitivity of response time to the value of uT where T is the size of the vocabulary and u is the fraction of the vocabulary which can appear in a query. Response times for each of the index organizations decreases as uT increases because the number of word occurrences in the database for an average query word decreases. That is, as uT decreases, the inverted lists that have to be read increase in size. The disk and I/O bus organizations are relatively insensitive to uT because they stripe lists across many disks, i.e., the list fragments that need to be read grow at a slower rate. The system organization is more sensitive to uT because inverted lists are read whole.
- Disk Seek Time The disk and I/O bus index organizations are most sensitive to the seek time of the storage device. This is because the disk index organization must retrieve for each query more inverted lists than any other organization. This same overhead is incurred by the I/O bus index organization to a lesser extent. The host index organization is very insensitive to seek time since only a few inverted lists must be retrieved per query.
- Effect of the load level on throughput As the load level rises, various bottlenecks in each index organization occur. Disk index organization has a disk utilization rate of 80.5% for a multiprogramming level of 1. The I/O bus index organization has a disk utilization of 58.7% for a multiprogramming level of 1 The host index organization has low disk and CPU utilization at a multiprogramming level of 1 (about 23.0% and 33.0% respectively) and thus has more spare resources to consume as the multiprogramming level rises. At a multiprogramming level of 32 (128 total simultaneous queries since there are 4 hosts) the disk utilization rises to over 74.3% and

CPU utilization to over 78.2% for this index organization. (Note that sufficient memory must be available to prevent excessive page faulting.)

- Effect of large partial answer sets on response time In some situations, the response time of a query decreases as the number of keywords in a query increases[3].

VI. MY THOUGHTS

Section V-A introduces the possible ways in which a Global Index may be distributed across the various processors in the network. In particular, the first technique, Index Partitioning, suffers from a load imbalance problem. It is in order to overcome this problem, that the other two techniques, in which the Index is multiplexed across processors, have been proposed. However, it may so happen that despite these load balancing strategies, the most frequently occurring words get assigned to the same processor, which may again lead to bottle necks. i

A slight variation of the techniques proposed in section V-A could be as follows. In case of a text based system, if a sufficient amount of domain knowledge is available, the Index structure could be sorted depending on the probabilities of occurrences of the word/suffix. Now, on this sorted array, the Multiplexing technique maybe applied. The drawback of this method is that a binary search cannot be applied in order to locate the processor. However, the problem maybe alleviated by caching all entries belonging to the group of words that are most frequently accessed. In practice this may or may not be possible considering the fact that there maybe a huge number of words that are accessed very frequently and it maybe infeasible to cache all of them. However, if the text database is such, that it has few words that are frequently accessed, then this technique maybe used.

The above technique maybe modified, to alleviate the cache requirement problem. This is done by keeping the usual lexicographic ordering of the words intact, but separating out only those words that occur most frequently. These frequent words maybe distributed across the various processors (using some technique like multiplexing or hybrid strategy) independent of their lexicographic ordering. Additional information about such words needs to be maintained by the broker machine, which maybe cached. The rest of the words are handled normally.

In another variation, if a load imbalance is observed, i.e. it is observed that most queries are routed to a specific processor, then the highly accessed words on that processor may be dynamically distributed among the other processors, which are less loaded. After this, the broker machine needs to update the information regarding the new location of the words. The locations of the re-allocated words need to be stored separately. However, this demands extra memory on the broker machine.

All the above variations would only be applicable if there are very few words that are accessed very frequently, and these words are known in advance. The above techniques, to the best of my knowledge have not been explored in detail.

VII. CONCLUSION

The paper starts out by describing the motivation for the use of Distributed/Parallel algorithms for Information Retrieval in Text Based systems. It provides a brief overview of some of the commonly used Index Structures for Text Based systems, and also summarizes Distributed Algorithms for generation of the same. It then goes on to describe the various options for physical design of a text document retrieval system. It provides information about the performance of several parallel query processing strategies, and the impact of the underlying technology. In particular, the choice of an Index organization depends heavily on the access time of the storage device and the bandwidth of interprocessor communication. The comparison of the three Index organization schemes reveals that Host based scheme is the best as compared to the others most of the times.

REFERENCES

- [1] Mauricio Marn Gonzalo Navarro *Distributed Query Processing using Suffix Arrays*
- [2] J. P. Kitajima, M. D. Resende, B. Ribeiro-Neto, N.Ziviani *Distribution of Parallel generation of Indices for very large text databases*
- [3] Anthony Tomasic, Hector Garcia-Molina *Performance of Inverted Indices in Shared-Nothing Distributed Text Document Information Retrieval Systems*
- [4] Berthier Ribeiro-Neto Edleno S. Moura Marden S. Neubert Nivio Ziviani *Efficient Distributed Algorithms to Build Inverted Files*
- [5] Mauricio Marin *Index Structures for Distributed Text Databases*
- [6] Mark G., Kevin L., Satish R. *Towards efficiency and portability: Programming with the BSP model*