CS 521 Parallel Algorithms and Architectures
Department of Computer Science
University of British Columbia

Inverted Index Used in Parallel Indexing and Query Processing

Charles Z. Chen

{*cszchen*}*@cs.ubc.ca*
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

Abstract

This paper proposes a way to do parallel query processing based on distributed inverted index. The conventional approach of search engines is to construct a complete inverted index and query on the inverted index in a high-computation ability machine. However, this approach suffers from two problems, one is the space limit and the other is the slow response time, particularly in a large scale. In this paper, we propose to divide the inverted index and distribute them to several nodes in a cluster, and then query on these nodes in a parallel way. In our model, there are two roles of processes, the farmer and the worker. The farmer process is mainly in charge of assigning tasks to workers and integrating partial results. The worker process is actually accessing the inverted index and calculating the results. Experimental results also show that querying in parallel outperforms the traditional approach by providing a better response time.

Keywords: parallel computing

1 Introduction

1.1 Search Engine Mechanism

Modern search engines basically consist of three parts: web crawler, indexer and a searcher. Web crawler is in charge of crawling new web pages on web and collecting them for indexing. As for the searcher, it receives queries from users and tries to find those documents with terms of the queries in indexer and rank them. In fact, we can see indexer has two functions. The first one is to analyze crawled documents and the second one is to fetch documents with given terms. Therefore, the efficiency to access the indexer is very important in a search engine, which directly decides the speed to answer a query.

1.2 Inverted Index

Inverted index data structure[1] is very commonly and efficiently used today in the indexer. The main idea of inverted index is to index all documents by all terms they include, which we call vocabulary. Each term should have a list of documents which contain the term and the corresponding weight of this document, for example the times the term has occurred in the document. When a user is delivering a query, he/she is going down to this inverted index and select out those documents that contain all of the terms in the query.

Generally the vocabulary and the according lists of documents, which are also called posting lists, are in memory while the files themselves are stored in hard disks. But sometimes the posting lists could be quite large and they would be put in hard disks too. The searcher works fine with this inverted index stored in just one computer in a small scale, which means the size of documents is small. But when the document size grows, the expense to read the whole posting lists of certain terms in one computer would be very high. On the other hand, what the users expect is fast response to their queries. In this paper, we are going to solve this problem by distributing the inverted index to a cluster of computers and deal with queries in a parallel way. Figure 1 shows an example of inverted index.

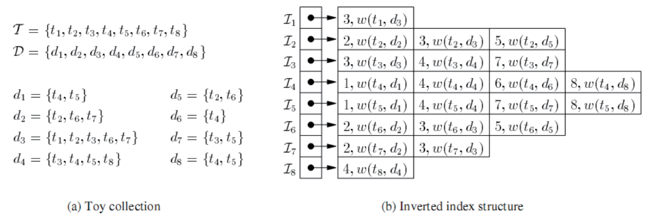

$\mathcal{T} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$
$\mathcal{D} = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8\}$
$d_1 = \{t_4, t_5\}$
$d_2 = \{t_2, t_6, t_7\}$
$d_3 = \{t_1, t_2, t_3, t_6, t_7\}$
$d_4 = \{t_3, t_4, t_5, t_8\}$
$d_5 = \{t_2, t_6\}$
$d_6 = \{t_4\}$
$d_7 = \{t_3, t_5\}$
$d_8 = \{t_4, t_5\}$
(a) Toy collection
$\mathcal{I}_1$: $3, w(t_1, d_3)$
$\mathcal{I}_2$: $2, w(t_2, d_2)$ | $3, w(t_2, d_3)$ | $5, w(t_2, d_5)$
$\mathcal{I}_3$: $3, w(t_3, d_3)$ | $4, w(t_3, d_4)$ | $7, w(t_3, d_7)$
$\mathcal{I}_4$: $1, w(t_4, d_1)$ | $4, w(t_4, d_4)$ | $6, w(t_4, d_6)$ | $8, w(t_4, d_8)$
$\mathcal{I}_5$: $1, w(t_5, d_1)$ | $4, w(t_5, d_4)$ | $7, w(t_5, d_7)$ | $8, w(t_5, d_8)$
$\mathcal{I}_6$: $2, w(t_6, d_2)$ | $3, w(t_6, d_3)$ | $5, w(t_6, d_5)$
$\mathcal{I}_7$: $2, w(t_7, d_2)$ | $3, w(t_7, d_3)$
$\mathcal{I}_8$: $4, w(t_8, d_4)$
(b) Inverted index structure

Figure 1: Inverted Index

In this figure, d_i represents the document with ID i, t_j means the term j. w is the weight of the document according to a term. In this example, there are totally 8 documents and 8 different terms. However, these 8 documents contain 21 terms all together, thus making the posting list size nearly 3 times the vocabulary size.

1.3 Parallel Query Processing Using Distributed Inverted Index

The parallel query processing based on distributed inverted index can be divided into two parts. The first part is to generate distributed inverted index from documents crawled. Basically documents are first kept on one computer using a Text Farmer process. Then Farmer process distributes documents to those Worker processes under the DocID-based rule. That is to say, we evenly divide documents by their ID to Worker processes. For example, if we have 1000 documents and two Worker processes, then the first 500 documents would be sent to the first Worker process while the latter 500 would be sent to the second Worker process. These two Worker processes keep an inverted index with partial posting lists in their computers. By now, we should have successfully distributed a large inverted index to a cluster of computers.

The second part is to do a parallel query processing. In this process, we need a central broker process which is in charge of directing queries to other worker processes. Once the worker processes receive the queries. They begin to read their partial posting lists, calculate them and return partial results to the broker. These partial results include weights of the resulting documents. Finally, the broker merges all the partial answers and ranks them with their weights, generating an optimized result back to the users.

With the parallel query processing, we can greatly shorten the search time by making use of a cluster of computers' computing ability. The distribution of inverted index allows each node in a cluster to generate partial results for the central computer, which is a great improvement in response time in query processing. In section 2, we will introduce our strategy with more detail. In section 3, we will present the details of the actual implementation. Performance test will be displayed in section 4 and section 5 will be the summary.

2 Strategy

As described in the previous section, we can see that to construct a distributed inverted index is actually a different procedure from query processing. It should be run in the back end and be prepared before the query processing can start. Ultimately, our goal is to enable processing queries in a parallel way to reduce the response time to find out those high ranked documents. An inverted index is a structure that first finds all the terms of the document set and then index all documents according to these terms.

2.1 Inverted Index Partitioning

Tomasic and Garcia-Molina[2] has examined two different techniques to partition an inverted index on a shared-nothing distributed system for different hardware configurations, including a term-based partitioning scheme and a document-based partitioning scheme. Term-based scheme states that each processor only deals with some specific terms, along with the documents attached while each processor is in charge of some selected documents in the document-based scheme. Cambazoglu[3] then has pointed out that document-based scheme would result a shorter response time. In Figure 2, both term-based partition and document-based partition schemes are shown based on the inverted index of Figure 1.

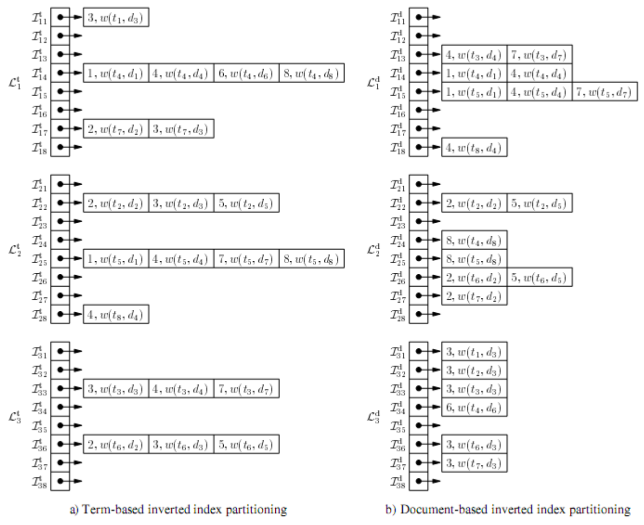

a) Term-based inverted index partitioning
b) Document-based inverted index partitioning

Figure 2: Two Different Partitioning Strategies

As we can see from the figure, in the document-based scheme, the posting list is shorter for each term in a node. The computation load for each query term has relieved by dividing the posting list to several nodes.

In our paper, we will adopt the document-based partitioning strategy. The most efficient way to implement this is to distribute documents to each processor and each processor will be in charge of generating its inverted index locally. However, there would be too many considerations in the process of generating an inverted index, for example the removal of stop words, the stemming of words ("ran" and "run" should be considered the same). Furthermore, this part is not the core objective in our paper. We decide to use an open source tool named WVTOOL[4], which is built in Java to generate the inverted index in one machine. Then we would use MPI to write a program to decompose this huge inverted index to different workers. What we would try to do is to evenly divide the inverted index as well as the documents and send them to different nodes. The most important criteria of this step is to distribute the inverted index and documents as soon as possible.

2.2 Parallel Query Processing

When a user delivers a query, a process would look into the inverted index to find out those documents with high rank. However, it takes a lot of time to search the whole posting lists. Now, since we have multiple machines, it would be good if we can make use of their computation abilities. To make the query processing efficient, the query search would contain two types of processes: a top process dealing with the clients and collating data and a bunch of leaf processes which are in charge of returning answers according to their inverted index table locally. When the top process receives a query, its job is to broadcast it to all other leaf processes. In MPI, the call function would be MPI_Bcast. And then the leaf processes would look into the inverted indexes and return the top N required documents back to the top process using a gathering mechanism, which maps to MPI_Gather in MPI. Actually only the document IDs would be returned to the top process to reduce the amount of message passing. The top process would keep a mapping function in memory of documents and their IDs. The actual documents would be sent to the top only if the user chooses to view the contents based on the heading.

2.3 Term Weighting Operations

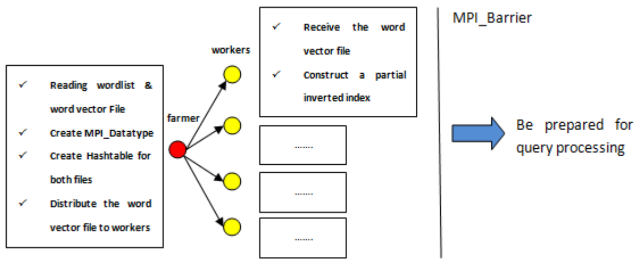
Considering that a user gives out a query with intersections, each term should not be viewed equally. For example, if a user gives out a query of "a very beautiful green car", in fact the term "very" is less important than "green" while "green" is also less important than "car". There is no reason that we find out those documents that contain the term of "very" many times. In this case, we need to use a method called TF-IDF[5], which considers the term frequency as well as the document frequency. We prefer those documents which the term occurs many times while the term does not appear in other documents very often. Therefore, the leaf processes are required to return the number of documents of a certain term first using the function MPI_Reduce. In this case, all leaf processes know the global information to do the TF-IDF weighing, so as to return those highly rank documents.

3 Details of the Implementation

In this project, we will divide the project into two parts. The first part is generally the preparation while the second part is the query processing. To implement the parallel programming, we will adopt MPICH[6], which is a high-performance and widely portable implementation of MPI.

3.1 Preparation

The preparation part will include the work of generating word list and word vector file, implementing hashtable, creating a specific MPI_Datatype and stemming the queries. By the end of the preparation part, the system should be able to execute queries and return the results. Figure 3 shows the main procedures in this preparation step.



✓ Reading wordlist & word vector File
✓ Create MPI_Datatype
✓ Create Hashtable for both files
✓ Distribute the word vector file to workers
farmer
workers
✓ Receive the word vector file
✓ Construct a partial inverted index
....
....
....
MPI_Barrier
Be prepared for query processing

Figure 3: Preparation for Query Processing

3.1.1 Word List & Word Vector

Before creating an inverted index, we need to parse the documents first and to generate a word list file and a word vector file. In a parallel way, when a document is crawled, it should be distributed to a worker and the worker is in charge of parsing it and add to the word list file and word vector file locally, and then notify the farmer process to update the global word list, which is stored in the farmer's machine. However, to generate a word list and a word vector file is too complicated, which itself is a big topic. We decide to adopt a tool WVTOOL to do this job for us. WVTOOL is written in Java and it can parse the documents using the techniques like stemming, statistics. Given a bunch of documents locally, WVTOOL can generate a word list file and a word vector file.

3.1.2 Hashtable

Hashtable is implemented in order to give a quick reference to a term or a document. In the real programming, we would not like to use a term or a document name directly because it takes up too much space. Instead, an integer which starts from 0, is assigned to each term or document to avoid the use of string directly. However, only assigning an integer is not enough. By storing them in an array, if the word list or document list contains too many entries, the sequential search on the array would be time-consuming. By adopting the hashtable, we can improve the efficiency of searching by quickly locating the position that holds the term name or document name.

3.1.3 MPI_Datatype

In the default data type of MPICH, only a few basic types like MPI_INT, MPI_CHAR are provided, which are not enough to represent the types needed to transfer between processes. In our project, we have a struct type defined in language C. Accordingly, it is necessary for us to create an MPI_Datatype to send the intact piece of information. Moreover, by defining a specific data type and patching data together can also avoid the overhead of frequent communication between processes.

3.1.4 Distributing Word Vector Data

Now we have already had the complete word vector file in the farmer process and have recorded all information needed in the farmer process. The next step is to divide the word vector file evenly with the document-based scheme and send them to different workers. In this step, MPI_Isend and MPI_Irecv are used to communicate. The reason we adopt the non-blocking message passing routines is that the partial word vector file could be very large, causing an undesirable delay by blocking other workers if some latency is high between a certain worker and the farmer, with blocking message passing routines. For the small scale communication, the blocking MPI_Send is used for convenience, for example the size of partial word vector file.

3.2 Query Processing Implementation

Now to this step, all the posting lists have been constructed and prepared. We would begin to implement the query processing phase. Like before, figure 4 has shown us a big picture of this query processing phase. The main jobs that we need to handle in query processing include dealing with query terms, the count of file number of a term, generating the top K results based on TF-IDF and the final result processing.

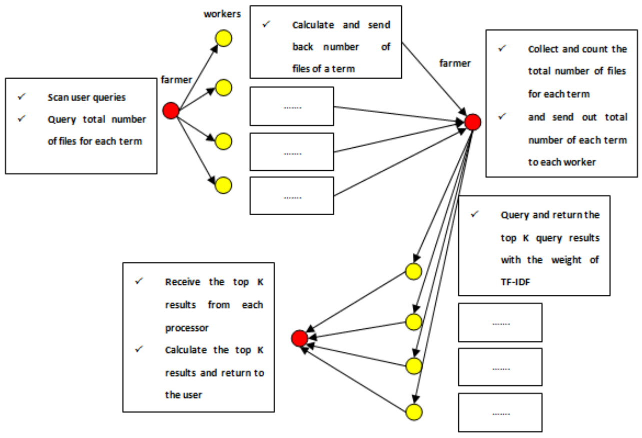

workers
✓ Calculate and send back number of files of a term
farmer
✓ Collect and count the total number of files for each term
✓ and send out total number of each term to each worker
farmer
✓ Scan user queries
✓ Query total number of files for each term
....
....
....
✓ Query and return the top K query results with the weight of TF-IDF
✓ Receive the top K results from each processor
✓ Calculate the top K results and return to the user
....
....
....

Figure 4: Parallel Query Processing Procedure

3.2.1 Stemming of the Queries

In the word list file generated by WVTOOL, words are actually stemmed using the Porter Stemmer Algorithm[7]. The Porter Stemmer Algorithm can identify a word with different formats. For example, *going*, *go* and *went* are all referring to the word *go*. On the other hand, some words also get trimmed to be a format without any meaning, like *whale* will turn to *whal* in Porter Stemmer Algorithm. Therefore, when given a query by the user, we need to get the stemmed format of the query term using the Porter Stemming Algorithm.

3.2.2 Implementing the Wight TF-IDF

The tfCidf weight (term frequencyCinverse document frequency) is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

In the definition of tf-idf, two kinds of measure are needed. One is the term frequency and the other is the inverse document frequency. In our implementation, term frequency is already calculated in the word vector file for each document. However, the inverse document frequency cannot be obtained unless the total number of documents for a term is known. In addition, this inverse document frequency is needed to calculate the top K results in a worker process locally. To solve this problem, we decide to make an additional communication between farmer process and worker processes. The goal of this communication is that the farmer adds up to all partial number to get the inverse document frequency for a term. And then the farmer sends back this number to all workers to local calculation. This process would be done with the MPI_Gather and MPI_Bcast functions.

3.2.3 Query Results with Posting Lists

When the query contains only one term, we just need to access the documents in the posting list of the term, sort them locally and then send them back to the farmer process. But when the query include multiple terms, this process would be a little bit complicated. We need to scan each posting list of different terms, find some common documents containing both terms and calculate the weight of tf-idf. Time complexity would be much bigger due to the mapping among posting lists and the construction of temporary posting list.

3.2.4 TOP K Results Optimization

In our query processing model, we make some optimization to relief the pressure from the farmer process from a practical view. Generally, take Google[8] for example, user only cares about the top K results from the result page. This observation means that workers do not need to return all the suitable documents to the farmer but only the top K ones with highest rank. In this case, the data needed to be transferred between processes is greatly reduced. In addition, it would save the farmer a lot of time to sort the results from multiple workers.

3.3 API Provided in the Project

This project consists of 5 parts logically, which are stemming, hashtable, MPI function, MPI preparation and File Accessing. They are all specified and defined in the header files separately. To make them reusable, we have packed them to a static library called libindex.a, which provides some APIs for users to call with the goal to do parallel query processing. In the following, we will explain the APIs provided in libindex. What worths to be mentioned is that, the APIs we provide is to meet the use of MPICH2 implementation, with the goal of simplicity. So we only show the APIs which directly concerns with MPICH programming, but not those APIs called inside another API. We will show the APIs in the following.

- ***MPI_Prepare(argc, argv);***
  - This API handles the MPI_Datatype, allocates some memory space for MPI communication and initiates some necessary MPI parameters
- ***void WordHash(char *wordlistname);***
  - This API reads in the word list file produced by WVTOOL and build a hash for the words
- ***void FileHash(char* wordvectorname);***
  - This API reads in the word vector file produced by WVTOOL and build a hash for the documents
- ***void DistributeIndex();***
  - This API divides the inverted index and distributes it to worker processes.
- ***int stem(char * p, int i, int j);***
  - This API deals with the query term and stem a term to a uniform form (e.g run, ran stemmed to run)
- ***void QueryWithIDF(char queries[MAXIMUMQUERYSIZE][100], int querysize);***
  - This API sends out the query terms, collects the number of documents reference to a term and sends out the total number of documents with a term back to workers
- ***void ReturnWithIDF();***
  - This API works in the worker process and sends back number of documents referred to a term to the farmer process
- ***struct ReturnType* ServerQuery(int* rsize);***
  - This API collects the partial results from the worker processes, resort these results and return the top K results to the user
- ***void ReturnQuery();***
  - This API works in the worker process side, calculate the results and return to the farmer process with the partial results.
- ***void FreeServer();***
  - This API frees the memory allocated in the farmer process
- ***void ReturnQuery();***
  - This API frees the memory allocated in the worker process

These APIs provide MPI programmers a convenient way to build a simple document search engine in parallel. In addition, programmers do not need to worry about the complex structure defined in the methods. Programmers can look up the usage of APIs in the head files.

4 Performance

In this section, we will conduct some experiments to test the performance of our parallel query processing. The tests are generally measured by time, we would see how faster it would be to proceed a query with multiple processes in a multi-core environment. Our experiment would run on the Selkirk server in UBC (selkirk.cs.ubc.ca), which is a 4-core machine with Linux. The CPU is Intel(R) Xeon(R) E5420, which can provide 8 processes running on it to maximally utilize the computation ability. But the architecture has limited this number to 7. Dataset preparation would be shown in section 4.1. Different experiments composed and their results would be presented in section 4.2. At last, section 4.3 will talk about some memory issues.

4.1 Dataset Preparation

Unlike the real search engine, which consists of a crawler to crawl web pages universally. In this experiments, we need to manually collect the dataset. We have collected a dataset of NSF Research Awards Abstract from 1990 to 1994, which has 51.798 documents out of 147MB. We further divide the dataset and construct five datasets for tests, which are documents for year 1990, documents for year 1990 and 1991, documents for year 1990, 1991 and 1992, documents for year 1990, 1991, 1992 and 1993 and the last one consists of all documents. By preparing the multiple size of dataset, we can test how our parallel query processing performs when the document size scales up.

For these five datasets, we use WVTOOL to generate the word list file and word vector file separately. For dataset 1, the word list file size is 288KB, the word vector file size is 24MB. The word list file size is 436 KB and the word vector file size is 49MB for dataset 2. For dataset 3, the word list size is 552KB and the word vector file size is 75MB. For dataset 4, they are 653KB and 101MB separately. At last, for dataset 5, these numbers are 741KB and 127MB. Table 1 shows the datasets we will use in testing.

Table 1: Datasets collected

Dataset	# of Docs	Size	Word List Size	Word Vector Size
1	10,140	27MB	288KB	24MB
2	20,976	55MB	436KB	49MB
3	31,756	85MB	552KB	75MB
4	41,939	115MB	653KB	101MB
5	51,978	147MB	741KB	127MB

4.2 Experiment Results

4.2.1 Varied Processes with Fixed Dataset and Query Term

At first, we adopt the non-parallel query processing as the baseline and test how our parallel query processing performs with different numbers of worker processes. When the number of processes equals to 1, it means the query processing is not done in a parallel way. When it equals to 2, two processes have been created with one as the farmer and the other as the worker. In this scenario, still only one worker is accessing the posting lists. However, when the number of processes increases to more than 2, the inverted index is split and we expect the query response time to be shorten. Because each query will only takes up a small time. We make an iteration of 1000 times on dataset 5 so as to make the response time more measurable. Figure 5 shows the test result on the query term "title".

As is shown in figure 5, we only did the experiment maximally to 7 processes on Selkirk. we can see that when the process number grows from 1 to 2, the process time increases by 10% due to the overhead of MPI communication. But we have two roles of processes right now. As the process number increases from 2 to 5, the response time is greatly reduced by 65%. But later there is some fluctuation. This is due to the additional MPI communication. In this scenario, the chance to have a higher latency is increased, which results in the performance degradation with the synchronized MPI calls like MPI_Gather. Another reason for this is that the server selkirk is shared by the whole department. The more processes we allocate, the easier some of them share the CPU with other intense jobs, which can halt other processes and influence the overall performance. Last but not least, the more processes would result in much sever unbalanced inverted index distribution. The load on all worker processes vary a lot.

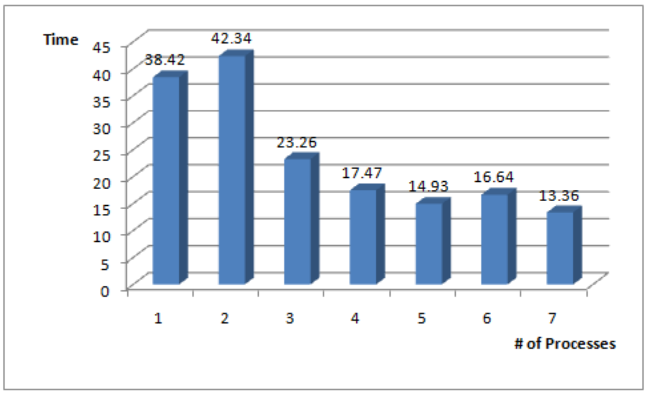


Figure 5: Parallel Query Processing with query term "title"

4.2.2 Varied Processes, Varied Query Term with Fixed Dataset

Based on the above experiment, we further extend our query terms and see whether all queries would get the same trend of performance when the processes increase. We choose some terms with decreasing popularity among all documents, which are "title", "grant", "university" and "August". We would like to see how different query terms would perform. Figure 6 will show the results based on dataset 5 with 1,000 iterations.

Apart from those mentioned in the analysis of figure 5, figure 6 has revealed that the longer the posting list is for a certain term, the more efficient it is with more processes. As a matter of fact, the experiment results show that when it takes more time to calculate the results, the synchronization time can be ignored in some way. Vice versa, when the calculation load is not high, the synchronization time becomes the main problem, even harms the performance as shown. Extra work needs to be done to make sure all workers return at around the same time. The strategy of evenly split

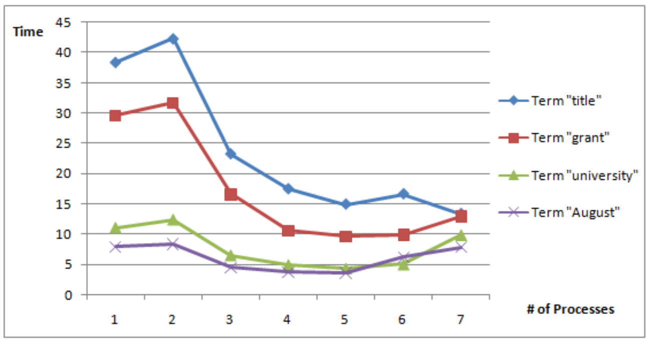


Figure 6: Parallel Query Processing with different query terms

of documents is not good enough.

4.2.3 Varied Process, Varied Dataset with Fixed Query Term

In this section, we will make a test on different datasets with the same query term. This step is to show the performance with different length of posting list for a specific term. This will simulate the situation that the crawler keeps crawled web sites and help decide when to have an additional process. Figure 7 shows the performance of term "title" for different size of datasets.

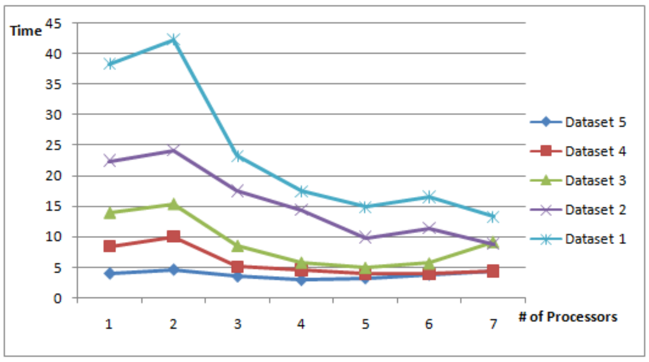


Figure 7: Parallel Query Processing with different datasets

As we can see from figure 7, for dataset 1 and 2, it is unnecessary to increase the process number when it reaches 3. However, when the dataset increases to the size of dataset 3, it would promote the performance to have 5 processes running. Of course, this is for a specific term "title". But we know the most efficient number of processes needed to deal with a certain length of posting list. In the same case, if we make some statistics about the average length of posting lists, we can have the optimal results on average.

4.3 Memory Issue

In our project, there is one more important but practical issues to mention, which is the memory issue. When there is only one query term, we just need to acquire one posting list and no additional memory is needed to find out the top rank results. However, when there are more than one query term, we would need to acquire according posting lists, which are often take up much memory space. In addition, to store the temporary lists produced, more space is needed. Under this condition, often one computer cannot provide enough memory to run the program. By dividing the posting lists to several nodes can not only provide faster response time but also solve this memory problem, even the inverted index is huge.

5 Conclusions

In this paper, we have implemented a simple parallel document search engine. This search engine differs from the traditional search engines in that it is proposed to run under a cluster of nodes or a multi-core computer to provide a better performance by utilizing the computation ability. This project mainly consists of two parts, which are the inverted index distribution and the parallel query processing. Inverted index distribution runs in the back end and aims to divide and distribute the inverted index to different processes. Parallel query processing handles the queries in different processes at the same time and return the user with the top K results.

We have made some experiments to test the query response time by creating different processes on an 8-processor machine. The results have shown that querying in a parallel way can greatly shorten the response time by at most 64% in our test case and provides a better user experience. Our experiments also show that we cannot increase the processes without limit. Communication cost and synchronization cost would become the major problem when the process number grows to a certain point.

But generally, with MPI, we can greatly promote the response time of query processing running under one process. And the project itself provides a better solution to search engines in a parallel way.

6 Future Work

In this project, we still leave some work to do. One thing of that is the construction of inverted index. Though it is not the focus of this project, our project just simulates this activity. In the real world, one computer cannot hold one complete inverted index in memory. The practical way to do this is to send the documents to a worker node, the worker constructs the partial inverted index locally. We leave this part of our library in the future work.

The other important thing is the distribution strategy. In this project, we adopt the simple even document-based strategy to divide an inverted index. But from the experiment, we can see this strategy would not improve the performance when the process number rises to a certain point. One reason of this is the problem of uneven work load among worker processes. In the future, we would try to propose a new split strategy of inverted index.

7 Acknowledgement

We will be thankful for those people who research on the parallel programming. The idea behind parallel programming is so powerful in the real world by promoting performance. To this specific paper, I would thank those researchers on parallel inverted index. The architecture of this project is from their ideas. Last but not least, the talk with Alan Wagner greatly clears my thought about this project.

References

[1] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full text databases. pages 352–362, 1992.

[2] Tomasic and Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval system. In *International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1992.

[3] B. Barla Cambazoglu and Cevdet Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems 67.

[4] Wvtool: http://nemoz.org/joomla/content/view/4-3/83.

[5] Wikipedia: http://en.wikipedia.org/wiki/tf-idf.

[6] Mpich2: http://www.mcs.anl.gov/research/projects/mpich2.

[7] Porter stemmer: http://tartarus.org/ martin/porterstemmer/.

[8] Google: http://www.google.com.