XSearch: Disk Data Pool

Krishna Bharadwaj
Illinois Institute of Technology
Department of Computer Science
Email: kbharadwaj@hawk.iit.edu

Animesh Patni
Illinois Institute of Technology
Department of Computer Science
Email: apatni1@hawk.iit.edu

Pooja Patel
Illinois Institute of Technology
Department of Computer Science
Email: ppatel115@hawk.iit.edu

***Abstract*—We live in a world now where the advances in computing and computer systems have led to a tremendous increase in the amount of data generated. Studies have shown that a major portion of the generated data is unstructured, and it is growing at 60% annually. With this upsurge of data, it is of paramount importance that we can retrieve the data and retrieve it as quickly as possible. This is especially true in the world of high performance computing(HPCs) where searches are performed over terabytes of files. In scenarios such as the one stated above, searching, which usually is a trivial task on a personal computer could get very tricky and end up consuming a lot of time. In this project, we look at some of the commonly used open source libraries such as Lucene, LucenePlusPlus and Xapian. The idea is to benchmark these libraries on a single node instance over text and metadata search. We observe the Indexing throughput, Query Throughput and the Index size and finally, identify potential bottlenecks in the system and perhaps come up with solutions to overcome them.**

I. Introduction

Information Retrieval is a technique of fetching relevant information from a large unstructured datasets. Searches can be based on text or content based indexing. Information Retrieval basically means searching for information with a text document, searching the document itself and also searching for metadata that describes the data itself. When we look at information inside our computers, they are in the form of directories and files. The outcome of these retrievals are basically unstructured on the other hand if the data is stored in a form of Relational Databases then it is structured data as all the data is in a tabular format. The modern day systems are powerful and load really large applications, they can even perform searches with high precisions with some really powerful search engines as well as data indexing approaches. But the question is what happens when the number of cores increases does this architecture still work? In case of Scientific Computers, the data is really huge and the files are distributed across many nodes which are accessed in parallel.

High Performance Computing Systems which came out from fast mainframes systems, they exploit parallelism on many levels and as well as there storage capacity is ever increasing. The storage structure of these High Performance Systems is arranged in such a way that it gives great performance on parallel distributed computing by splitting the data into small chunks and spreading it which gives high throughput. As we are migrating to cloud computing and using distributed systems in the form of Hadoop (HDFS), more efficient and fast information retrieval techniques have become a necessities.

This project in a way looks at this, and tries to compare different search libraries and how to perform on single thread as well as multi threading approach. This paper is organized in a particular manner that in each section we talk about how these approaches can be shaped for better performance on High Performance Systems for information retrieval. It also talks about what assumptions as well as the bottleneck we faced while running our experiments on different libraries.

II. Previous Work

As we delved deeper into the topics of information retrieval, we stumbled upon many studies focusing on performance evaluation of various information retrieval libraries. However, we notice that in many such studies, the emphasis was primarily on the accuracy of the search result. EVALUATION OF INFORMATION RETRIEVAL SYSTEMS by Keneilwe Zuva and Tranos Zuva talks about evaluating information retrieval systems based on precision and recall. However, our emphasis is firmly on query throughput, Index throughput and index size. To put this in simple terms, we look at how fast a search can be performed rather than how accurately the search is performed.The very common approach used is Elasticsearch, which helps indexing the documents that are text based in a distributed system and then on it the search applications is built using Apache's Lucene. But the indexing performed by Elasticsearch is not optimized for the kind of the text or data that is present in the parallel file systems.The other previous works related to this project is Robinhood (by Lustre's) which uses database in the current parallel file system's architecture.As this uses database hence it allows storing the metadata and also can query the metadata from other file systems that can be distributed file system but with this approach there still remains the requirement of managing the users to define the schemas and also there is one drawback that they cant be easily put into a file system model as they are for the structured queries than the free text.

III. QUANTITATIVE ANALYSIS

Before directly jumping into writing and calling different Search APIs, we looked at different aspects of our system as well as the data, Wikipedia Dumps and Meta Dumps to provide us some background of what exactly we are dealing with and what steps we need to take to achieve our goal

of comparing three different search libraries. We started by analyzing the instances(KVM as well as Bare-Metal). KVM instances proved to be good for small chunk of data and using RAM directory, while Bare-Metal was really strong in both small as well as large chunk of data as the disk access was really fast and accurate. We analyzed how many cores, buffer memory and disk IOs can be managed on both the instances and how we can leverage those to get closer to our goal. The specification for the KVM instance as well as the Bare-Metal instance is shown in the table below. Data Analysis was the second part, like how data is structured and how to pre-process it to get a better outcome. This helped us to decide which Analyzer to use for a better and accurate results. In case of Meta dumps, data analysis provided us the insight of how to deal with different tags and what approach to take if some of the tags were missing for some files. We not even looked at the index speed and the search speed we consistently printed the outcome to see if the search as well as the indexing is happening correctly or not. A storage instance to split huge dataset into smaller chunks as bare-metal cannot handle very large files. After analyzing the data the next step was to do some research on what exactly is the strength and weakness of Lucene, LucenePlusPlus and Xapian and how to overcome those to improve the search performance. Throughput for each libraries on small data set gave us a great insight of how each of them is performing in different circumstances. For a better analysis, wrote a script to extract 1000 random text as well as Meta keywords for better analysis. One of the most challenging phases of this project was to index meta data using multi-threading approach. The third library this project uses is Xapian which was a challenging task to implement as there was no clear documentation available.

IV. PROBLEM STATEMENT

The Problem we are addressing is from the information retrieval side of Computer Science. Information Retrieval is a process of obtaining relevant information based on the search criteria from the resources of information The search can be based on content as well as the metadata. In computer system the materials are represented as files while the collection of information resources is comprised of files system. To improve the performance and efficiency of search on text or metadata in the supercomputers is what we are looking at. Also this area is not explored or focused so much previously. The applications are used for a search of a work in computer and those application are called search engines. Future to verify the search results the performance evaluation metric are used, one of them is precision which gives the score of how many were the relevant search results among all the results thrown by a query. The other metrics that can be used for performance evaluation are recall which is also known as True Positive rate which is the ratio of all the correct search results returned by the actual number of search results that should have been returned. F1 score which is the harmonic mean of precision and recall. But on the other side the systems like cloud computing, grid computing, clusters and super computing the performance evaluation is done using different metrics which are latency , throughput , scalability etc. The problem given is dealing with the large amount of data then we need to use distributed system to solve the problem where we measure metrics like throughput, search latency, indexing latency etc. Index size , Index time and query size are also few of the concerns in such HPC (High Performance Computing) systems. The accuracy of the system can be computed by the above parameters.

V. APPROACHES, METHODS AND DESIGN

This section focuses on the approaches and discusses how the text and meta data search was performed on Lucene, LucenePlusPlus and Xapian.

A. VARIOUS MULTI-THREADED APPROACHES

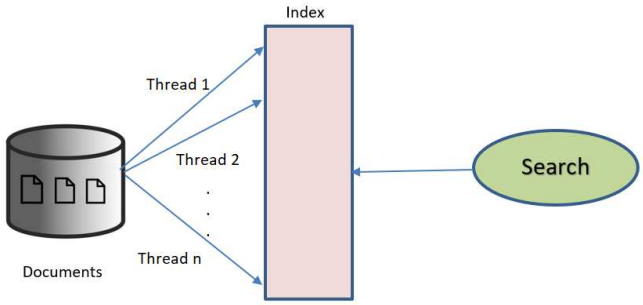

Fig. 1. Option A: Search and Index Approach

- **Option A:** In this option, we have a single global indexer and multiple threads access this indexer to index data. This provides a smooth implementation and a faster search, since we have only one index. However, since we have a single indexer, the problem now lies in the concurrency aspect. Only one thread can have access to the indexer. This problem can be mitigated to an extent by making use of the concurrent merge scheduler, however, we posit that even with the merge scheduler, this approach cannot be faster than an implementation where each thread runs independently with an indexer per thread. It is easy to visualize why this option is further discredited when we have to perform indexing in an environment with multiple nodes over a network. Here, the limiting factor will be the network speed and since each node will need to access a single indexer, this will firther slow down the indexing, with the obvious bottleneck being the network.

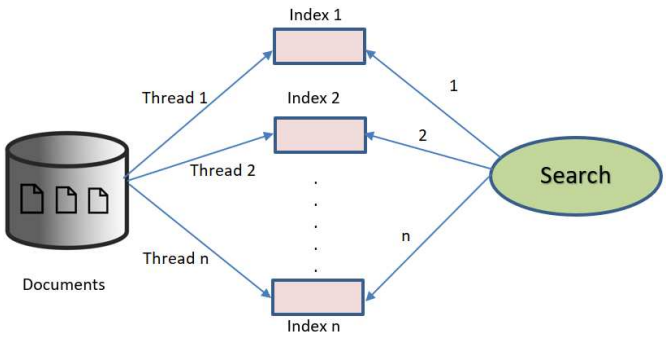

Fig. 2. Option B: Search and Index Approach

- **Option B:** Here, we have multiple indexers, one per thread. This would result in multiple indexes being created for a workload. Since there will not be any dependency between the threads, the overall indexing will go much faster than option A. However, once the index is completed, we will either have to merge the indexes into one single index, which will be time consuming, depending on the number of threads or we leave the indexes as is and perform search on multiple indexes, which will impact the search times. In our experiments, we gather that the search times did not impact by a huge margin when we tried to search all the indexes independently and merge the results. This search time could also be improved by reducing the number of threads so that the searcher will need to search lesser indices. Hence, this becomes an optimization problem. We can increase the number of threads to reduce the index time or decrease the number to improve on the search time. This approach also has merits when we try to index over a cluster of nodes since the network which was a bottleneck in our option A is no longer a bottleneck here since we no loger have a single indexer.

B. LUCENE

Lucene is an open source library originally written in Java and later on ported to different programming languages like Perl, Pascal, C# and others. Lucene library has grown a lot over the years, working on indexing large text file to metadata. Lucene is independent of file formats and can index text files with extention like PDF,DOCX amd many more. We worked on Lucene with text data as well as the metadata dump from High Performance System or Super Computer.

1) TEXT SEARCH: For Text data we indexed each document and then randomly picked 1000 words using Python script and ran our experiments. In the initial phase, we made our code multithreaded, so that we can index more quickly utalizing the full strength of our Instance. We faced issues as we were using RAM directory on KVM instancem that was not as efficient as FSD Directory which we went on to implement. The bottleneck for RAM Directory was the size of the buffer. Indexing and Searching speeds improved by switching to Simple Analyzer(which doesnot deal with Stop words or links), FSD directory and Multitreading. As Simple Analyzer indexes the whole document as it is, it has a drawback that the index size is large. We implmented Lucene on the baremetal instance with a large dataset keeping multithreading into account.We ran into randomn instability, leading to some exceptions or sometimes the it running into endless loop indexing the same file over and over again. We opted for the same approach that we used in LucenePlusPlus for indexing the data one was single global indexer and other was multiple indexer. The performace for option B was considerably faster, with one index per thread

2) METADATA SEARCH: As from using a small dataset to large and metadta coming into picture we moved on to Bare Metal instance which gave us more cores and memory to work with. Implementing search and indexing on meta dumps was a challenge as we have to take each line as a document and split them into to store field values as indexes. We started with small dataset implementing metadump split and then moved on to multithreaded implementation. We indexed documents on each thread, suppose for example if there are 20 threads running in parallel we indexed different documents containing 1000 lines each as a separate index and merge them together after the index process was complete. This considerably improved the performace as compared to our previous implementation for metadata search. The bottleneck for this appraoch was the disk size as we need to create a separate instance and split the huge dump file into smaller chunks to make it work on a bare-metal instance. We used IndexWriter(using skip list), IndexSearcher, QueryBuilder and QueryParser to build an index and perform search on it. We read a file which contained 1000 meta keywords generated randomly using a script to achieve the same.

The true strength of Lucene Library is you can easily index data using the schema and then can perform search with blazing speed. We can keep on improving the indexing speed(by implementing data strucutres that are more efficient and fast or by tweaking other parameters like Analyzers) which eventually increases the overall search speed of the system. It works really well with text based searches. The Lucene is very well documented. The only thing we dislike with Lucene is setting up vanila Lucene and indexing data parallely other than text. Setting up and getting to know how things are flowing takes some time. The performace analysis of Lucesne with comparisions with the other two libraries is shown in performance analysis part of the paper. We tweaked some other parameters like the Buffer size but with very little chane in the indexing time.

C. LUCENEPLUSPLUS

LucenePlusPlus as described in the github link is a C++ port of Apache Lucene. Completely written in C++ with the use of boost libraries, luceneplusplus pretty much functions similar to apache lucene, presenting the same interfaces and programing constructs. It does inherently support multithreading, however, it lacks the kind of autotuning of threads that can be found in Apache Lucene, leaving the tuning aspects to the programmer.

We also observed that with the increase in the number of threads beyond 8, we sometimes ran into random instability while running the program and at times, leading to segmentation faults and other memory exceptions. Hence, emphasis is on optimizing our multithreaded approaches. As stated in XSearch paper, some of the base classes had to be modified in order to optimize multithreading. The implementation as a whole was slow and had to follow in the footsteps of lucene, with a few modifications in order to suit the luceneplusplus constructs. The overall lack of documentation was frustrating.

1) TEXT SEARCH: Text search implementation in luceneplusplus was performed in parallel with the implementation of lucene. However, some precautions need to be taken, because, even though luceneplusplus is a port of lucene, there are a few constructs that differ, mainly due to lucene being in java and luceneplusplus in C++. One such instance was that we were not able to read more than 1000 files in luceneplusplus, whereas the same could be done on the lucene implementation. Upon further inspection, we found out that the files opened in luceneplusplus were not closed and we had to do so manually, however, java takes care of that inherently. Since, in this project, we focus solely on multithreading while indexing, we were able to devise two different implementations to do the task. Both of which have their pros and cons and it ultimately becomes a matter of choice.

We were able to implement both the options described above and we found significant gains by choosing option B, with an indexer per thread. Our experiments also showed that at index time peaked at 16 threads and any further increase in this number only served to increase the search time and not reduce the index time by a whole lot. We also performed experiments with the different analyzers such as the standard anlyzer and the simple analyzer. We found a small gain in performance while using the simple analyzer, however, tradeoff is the search time and the index size since this analyzer indexes the documents as is. We also tweaked various parameters such as MAX_BUFFER_SIZE and MAX_BUFFERED_DOCS and found negligible change in index time.

2) METADATA SEARCH: The core concepts of the metadata search for luceneplusplus are similar to the text search for lucene and luceneplusplus in terms of invoking the libraries to do the indexing. However, a significant difference lies in how we parse the the information and add to index. To elaborate upon this point further, let us look at our workload. We used a dump from a supercomputer, with each line pertaining to the metadata of a file. Each entry is separated by a space and all in all, we have 16 entries. These entries range from inode number, user ID, permissions, file path etc. For our experimentation, we only looked at the inode, user ID and the file path. We split the dump into smaller files of size ~240 KB, each containing 1000 lines, which pertain to 1000 files in the supercomputer. We began our experimentation with 1 GB of such data, holding ~5500 files and a total of ~5.5 million supercomputer files. All in all, we treat each entry as a file in the supercomputer and index them. Stemming cannot be done here since we may lose out on important information. We employed both the ways of multithreading discussed above and we have found out that *option B* comes out on top yet again.Our experiments also showed that at index time peaked at 16 threads and any further increase in this number only served to increase the search time and not reduce the index time by a whole lot.

D. XAPIAN

Xapian is an Open Source Search Engine Library written in C++, with binings to other languages such as perl, python, Java etc. Xapian was built as a system for efficiently implementing the probabilistic IR model (though this doesn't mean it is limited to only implementing this model - other models can be implemented providing they can be expressed in a suitable way). The model has two striking advantages -

- It leads to systems that give good retrieval performance. As the model has developed over the last 25 years, this has proved so consistently true that one is led to suspect that the probability theory model is, in some sense, the "correct" model for IR. The IR process would appear to function as the model suggests.
- As new problems come up in IR, the probabilistic model can usually suggest a solution. This makes it a very practical mental tool for cutting through the jungle of possibilities when designing IR software.

1) TEXT SEARCH: Indexes in Xapian are essentially databases that are stored on disk. We followed similar multithreading approaches as discussed for lucene and luceneplusplus. One interesting result we obtained was that the index time for xapian was about an order of magnitude higher than that of lucene and luceneplusplus for a 3 GB wikipedia dump. However, another interesting observation was that when we increased the size of our workload to 20 GB wiki dump, the advantage in performance for xapian reduced. This leads to the conclusion that as we increase the number of documents to be indexed, the performance of xapian reduces and we will see further that the performance worsens for metadata search. Coming back to text search, we implemented the second multithreaded option which was discussed previously, which features a dabase per thread. This significantly boosts the indexing throughput as we increased the threads from 1 to 48, peaking at 32 threads. However, a significant advantage in Xapian is that we were able to merge the databases significantly faster than Lucene, thereby reducing the search time, compared to lucene and luceneplusplus. It is worth noting that at this time, we have contacted the authors of Xapian and sent our implementation for verification. We have received no reply at the time of writiing this paper.

2) METADATA SEARCH: Metadata search on Xapian was performed once again on 1 GB of supercomputer metadata dump, with each file having 1000 entries with each entry corresponding to a file in the supercomputer. Since each line corresponds to a document in Xapian, the number of documents indexed in metadata indexing is exponentially higher than that of text indexing where each file is a single document. This change highly impacted performance of Xapian as we see further in the Analysis section of the paper.

VI. TEST BED

It is very important to know the system specifications in order to make comparison between two things. The following are the system configurations of the system on which we performed all the Lucene, LucenePlusPlus and Xapian text and metadata experiments and obtained the results. There also came a requirement where the KVM storage instances had to be launched due to the huge amount of the metadata dumps that we had to download and split it into smaller files.

A. Software Versions

Following are all the software and their versions which we used:

Software Name	Software Version
Lucene	7.1.0
LucenePlusPlus	3.0.7
Xapian	1.4.8

Fig. 3. Software Versions

B. Instances

KVM Instance system configurations:

Name	Value
RAM	8GB
VCPUs	4
Disk	80GB
Flavor ID	4
Flavor	m1.large

Fig. 4. KVM - System Configurations

KVM Storage Instance system configurations:

Name	Value
RAM	4GB
VCPUs	1
Disk	2048GB
Flavor ID	6
Flavor	storage.medium

Fig. 5. KVM Storage Instance - System Configurations

Baremetal Instance system configurations:

Name	Value
RAM	128 GB
Cores	48 cores
Threads per core	2
Cores per socket	12
Model	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
OS	Ubuntu 16.04

Fig. 6. Baremetal - System Configurations

C. Workloads

The below table shows the workloads of wikipedia dataset for text search and meta data search that were used and on which the experiments were conducted:

Search	Workload
Text Search	3 GB & 20 GB
Meta Data Search	1 GB & 7 GB

Fig. 7. Baremetal - System Configurations

VII. EXPERIMENTAL ANALYSIS

The experiment analysis was done using the libraries such as Lucene, LucenePlusPlus and Xapian. We have the results for text and also meta data search. The experiments were performed on 1 thread, 2 threads, 4 threads, 8 threads,16 threads, 32 threads and 48 threads. Emphasis was primarily on indexing with the use of different multithreaded approaches discussed in previous sections. The implementations index a chunk of files for each workload. The index time is printed to output once completed. Then the searcher searches for terms in the index. The terms are essentially a set of 1000 randomly generated words from the workload itself. These words were generated with the help of a python script. Experiments were run completely on the bare metal instance stated in section 6.2.

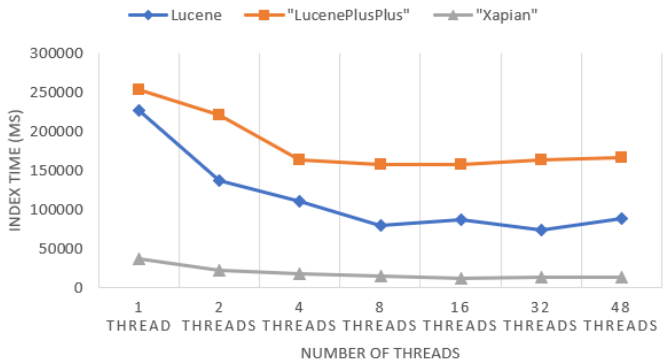

Fig. 8. Index time for 3 GB text search

The graph above depicts the index time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that index time decreases for all the libraries as the number of threads increases. Xapian out-performs the other two libraries in this case.

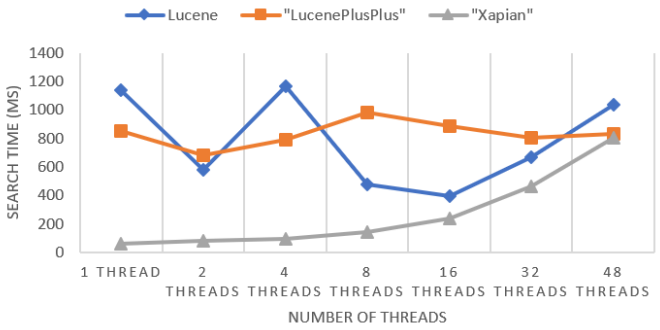

Fig. 9. Search time for 3 GB text search

The graph above depicts the search time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that search time is in-consistent as we increase the number of threads in case of Lucene as well as LucenePlusPlus while for Xapian it increases as the number of threads increases. Xapian out-performs the other two libraries in this case.

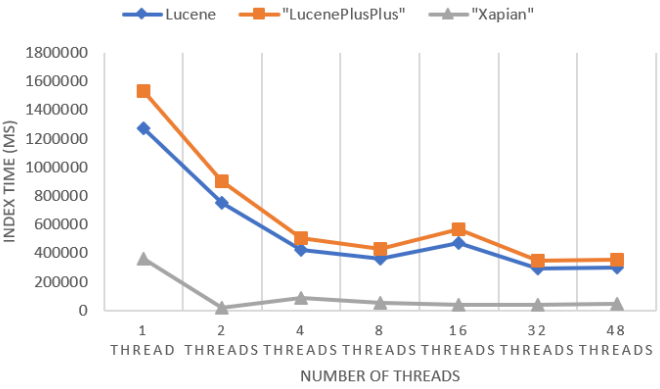

Fig. 10. Index time for 20 GB text search

The graph above depicts the index time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that index time decreases as the number of threads increases. Lucene and LucenePlusPlus have almost similar indexing time while Xapian takes the least amount of it.

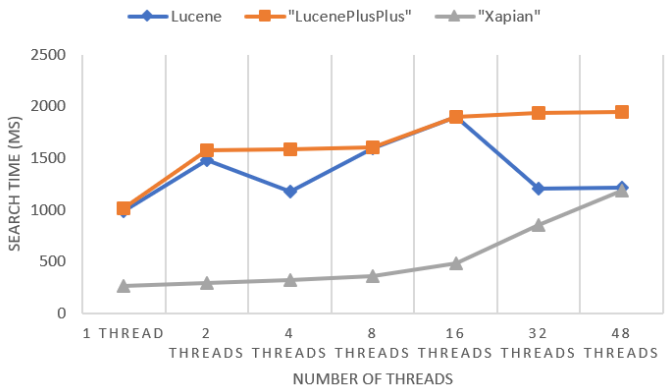

Fig. 11. Search time for 20 GB text search

The graph above depicts the index time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that search time for LucenePlusPlus and Xapian increases with the number of threads, in case of Lucene it is inconsistent. Xapian performs the best amongst the three.

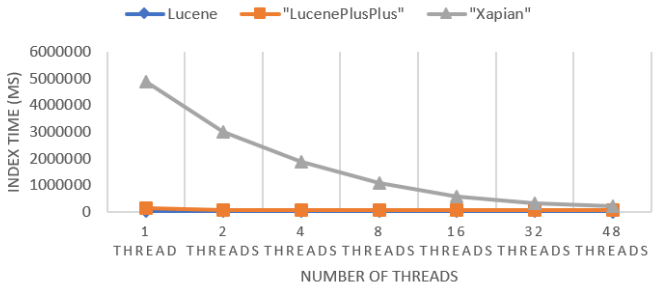

Fig. 12. Index time for 1 GB Meta data search

The graph above depicts the index time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that index time for LucenePlusPlus and Lucene is almost the same for all the threads, in case of Xapian the time increases with the number of threads

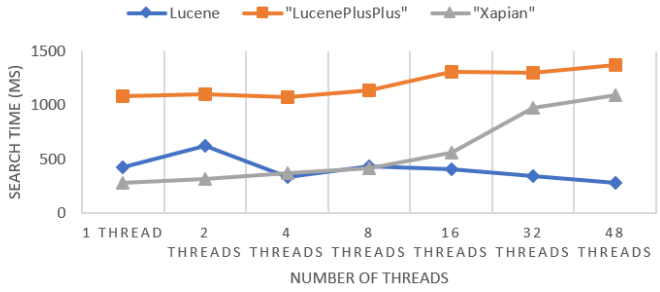

Fig. 13. Search time for 1 GB Meta data search

The graph above depicts the index time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that search time for LucenePlusPlus and Xapian increases with the number of threads. On the other hand the index time for Lucene decreases as the number of threads increases.

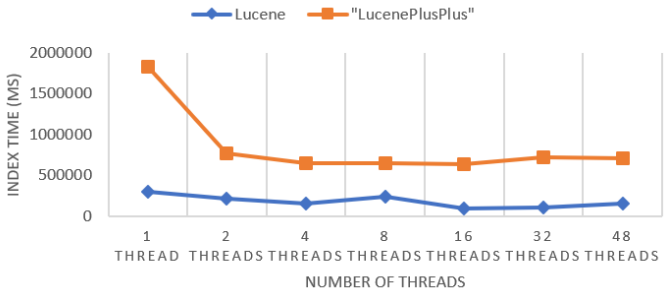

Fig. 14. Index time for 7 GB Meta data search

The graph above depicts the index time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that index time for LucenePlusPlus decreases as the number of threads increases. In case of Lucene the time taken is less as compared to LucenePlusPlus, the index time decreases as the number of threads increases.

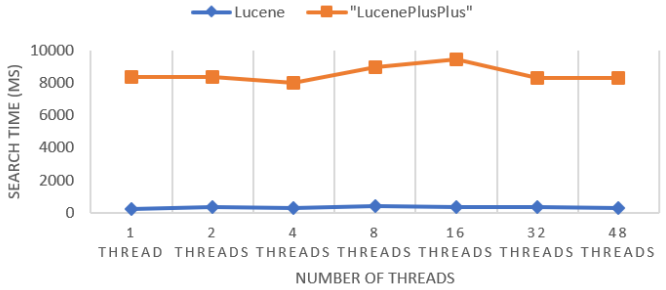

Fig. 15. Search time for 7 GB Meta data search

The graph above depicts the search time in milliseconds on the y-axis and number of threads on the x-axis. As it is visible that search time for LucenePlusPlus remains consistent as the number of threads increases. In case of Lucene the time taken is less as compared to LucenePlusPlus, the search time remains consistent as the number of threads increases

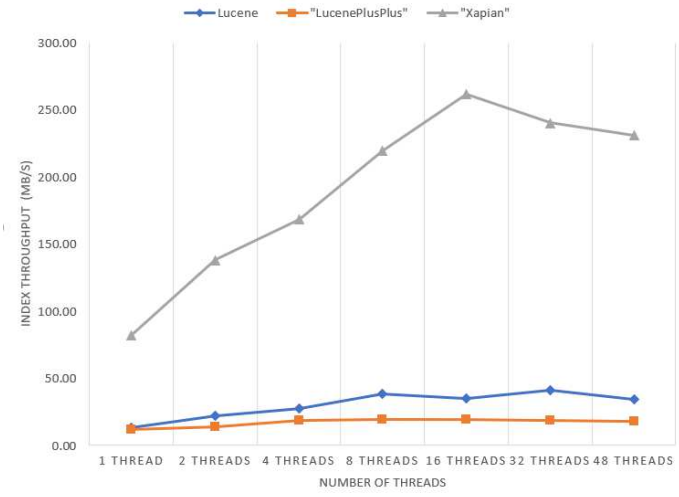

Fig. 16. Index Throughput for 3 GB Text search

As seen in the graph above, the index throughput for xapian outperforms lucene and luceneplusplus. The throughputs of Lucene and LucenePlusPlus are similar with Lucene performing a little better than the former

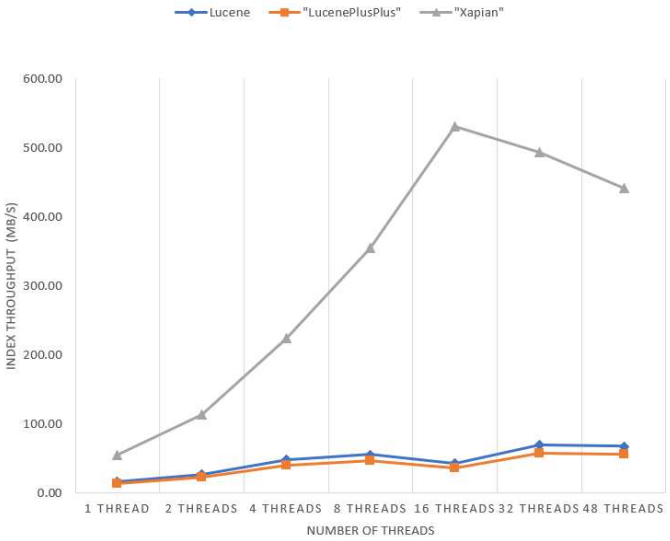

Fig. 17. Index Throughput for 20 GB Text search

We see similar results for the text search of 20GB data as we see for 3GB data. Xapian displays the best performance, followed by Lucene and we see that LucenePlusPlus has the least performance.

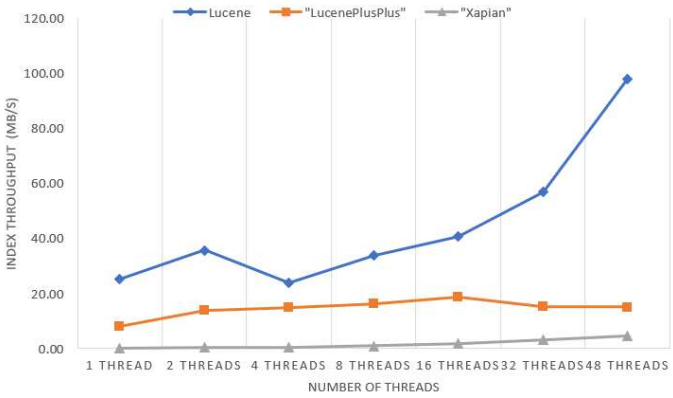

Fig. 18. Index Throughput for 1 GB Meta data search

The figure above depicts the index throughputs of each of the libraries in indexing 1GB of metadata file obtained from the supercomputer dump. We see a stark decline in performance of Xapian as compared to the text search and this is attributed to the fact that as the number of documents increases, the performance of Xapian reduces significantly. Lucene seems to perform best, followed by LucenePlusPlus.

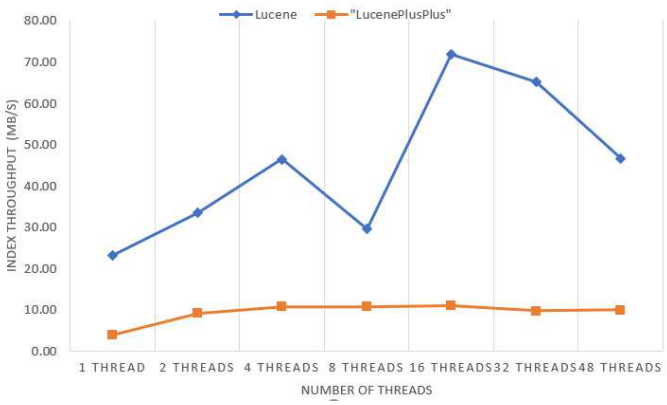

Fig. 19. Index Throughput for 7 GB Meta data search

The figure above depicts the index throughputs of each of the libraries in indexing 7GB of metadata file obtained from the supercomputer dump. Lucene seems to perform better as we increase the number of threads, which has better thread capabilities as compared to LucenePlusPlus.

A. KEY OBSERVATIONS

The following were some key observations we noted from the experiments conducted and visualization of the results obtained -

- In text search, Xapian far outperforms Lucene and LucenePlusPlus, however, this comes at the cost of a bigger index size, bigger than the original data
- Lucene and LucenePlusPlus recorded similar values in a single threaded environment. However, in with multi-threading, Lucene seems to perform better. This is mainly due to Lucene being able to autotune the threads whereas in LucenePlusPlus, the onus is on the developer to tune the threads.
- Lucene seems to be having the best performance when it comes to metadata search followed by LucenePlusPlus, however, only by a slight margin. This was to be expected since lucenePlusPlus is a direct port of lucene in C++
- However, Indexing time of Xapian was very high. We posit with this fact and the fact that Xapian performed very well in text searching that the number of documents plays a major role for indexing in Xapian. As an example, the 1GB metadata dump holds around 5500 files. Each file has 1000 lines where each line corresponds to metadata of a file in the supercomputer. Since we index each line as a document in Xapian, the total number of documents is close to 5.5 million. This seems to terribly affect the performance of Xapian indexer.

VIII. CONCLUSION

High Performance Computing Systems have huge amount of data and searching through that data is a very challenging task. Our work shows the comparison for three libraries and how they are performing on both Text Data as well as Meta Data. We observed that with the increase in threads the index time decreased but the latency for search kept on changing. In case of indexing on multi-threaded format we observed that it became both storage as well as computing problem when dataset kept on increasing. One key observation was that the performance was not increasing linearly with the number of threads. Rather, it increased to a certain point and it remained stagnant for most cases. This led us to believe that the processor cannot be the bottleneck. From our experimentation with Xapian, we can see that index time increased by a lot for metadata as compared to text. This was due to the number of writes into the database which was held on disk. Hence, the component that became the bottleneck for this project was disk. In the future we would like to continue working on these libraries, exploring them thoroughly and tweaking some of the parameters. We would work on implementing C-Trie Data Structure that is theoretically considered faster and more accurate. The impact of our work is we compared three libraries on the same test-bed and generated throughputs for the same. This project gave us a great insight of how different kind of search libraries are structured and how to perform search on Super Computer meta data.

ACKNOWLEDGMENT

Results in this project were obtained using Bare-Metal as well as KVM instances on Chameleon which is supported by National Science Foundation. Base of this project was provided by our mentor Alexandru Iulian who was very supportive and helpful. The work was done in the guidance of Ioan Raicu and we would like to thank him for giving us such an opportunity.

IX. REFERENCES

[1] Alexandru Iulian Orhean, Kyle Chard, Ioan Raicu.“XSearch: Distributed Information Retrieval in Large-Scale Storage Systems”

[2] Alexandru Iulian Orhean, Itua Ijagbone, Dongfang Zhao, Kyle Chard, Ioan Raicu. “Toward Scalable Indexing and Search on Distributed and Unstructured Data”, IEEE Big Data Congress 2017

[3] Itua Ijagbone, “Scalable indexing and searching on distributed file systems”. Master thesis. Illinois Institute of Technology, 2016

[4] keneilwe zuva and tranos zuva. “Evaluation of information retrieval Systems”

[5] Dongfang Zhao, Ning Liu, Dries Kimpe, Robert Ross, Xian-He Sun, and Ioan Raicu. “Towards Exploring Data-Intensive Scientific Applications at Extreme Scales through Systems and Simulations”

[6] T. Leibovici, “Taking back control of hpc file systems with robinhood policy engine,”

[7] S. Chafle, J. Wu, I. Raicu, and K. Chard, “Optimizing search in unsharded largescale distributed systems.”

[8] Patrick Glauner, Jan Iwaszkiewicz , Jean-Yves Le Meur and Tibor Simko, “Use of Solr and Xapian in the Invenio document repository software”

[9] LucenePlusPlus OpenHub : https://www.openhub.net/p/LucenePlusPlus

[10] http://wwwhome.cs.utwente.nl/~hiemstra/papers/IRModelsTutorial-draft.pdf

[11] http://lucene.apache.org/

[12] https://github.com/luceneplusplus/

[13] https://xapian.org/

[14] P. Schwan et al., “Lustre: Building a file system for 1000-node clusters,” in Proceedings of the 2003 Linux symposium, vol. 2003, 2003