Heterogeneous Parallel Architecture for Inverted Index Generation

Tiago B. Silveira, Felipe A. L. Soares, Wladmir C. Brandão, Henrique C. Freitas

Programa de Pós Graduação em Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Belo Horizonte – MG – Brazil

{tiago.batista, felipe.soares}@revex.digital,

{wladmir, cota}@pucminas.br

Abstract. *The amount of data generated on the Web has increased dramatically, as well as the need for computational power to prepare this information. In particular, indexers process these data to extract terms and their occurrences, storing them in an inverted file, a compact data structure that provides quick search. However, this task involves processing of a large amount of data, requiring high computational power. In this article, we present a heterogeneous parallel architecture that uses CPU and GPU in a cluster to accelerate inverted index generation. Experimental results show that the proposed architecture provides faster execution times, up to 60 times in classification and 23 times in the compression of 1 million elements.*

1. Introduction

The amount of information exchanged daily in the Web has dramatically increased [Chen and Zhang 2014]. This flow of information promotes the emergence of huge databases, demanding high processing power in order to prepare them for user consumption. The Information Retrieval (IR) research field investigates techniques to handle the increasing demand for information, addressing problems of information representation, storage, organization and location, with the main focus on retrieving relevant information to fulfill users' needs.

Indexing models, such as suffix arrays, signature files and inverted indexes, are processing methods used in IR to turn search queries more responsive. Inverted indexes stand out among the IR practices most used in commercial search engines. The approach used by an inverted index consists of data processing in order to store the terms, their occurrence in documents, and sometimes the source documents. The inverted index structure is essential to speed up the IR process. However, index generation demands high computational time in accordance with the amount of data. Due to this problem, it is paramount the development of efficient inverted index generation architectures to address the increasing demand for processing [Zhengyou and Tao 2009].

Therefore, advances in high-performance computing provide the infrastructure to handle large workloads [Kucukyilmaz et al. 2012]. For this reason, in this paper, we propose a heterogeneous parallel novel architecture for inverted index generation from the in-memory inversion algorithm. Particularly, our proposed architecture uses Central Processing Units (CPUs), Graphics Processing Units (GPUs) and a computer cluster

to improve scalability. The GPU node calculates index and sorts the *term-document-frequency* triples in memory using the Compute Unified Device Architecture (CUDA). The sorted dictionaries are stored in a First In First Out (FIFO) queue in a centralizing node, responsible for merging operations, and perform writing and reading (I/O) operations on disk. As a result, unnecessary concurrency and I/O operations are eliminated on the processing nodes. Experiments show improvements in speedup results compared to sequential approaches used to generate inverted indexes.

This article is organized as follows: Section 2 presents background concepts and Section 3 describes related work. Section 4 presents our proposed architecture for inverted index generation and Section 5 shows the experimental setup, results and evaluation. Finally, Section 6 presents our conclusions and future works.

2. Background

The IR research field deals with representation, storage, organization, and location issues, with the primary focus of retrieving relevant information for users. Given the vast amount of information contained in documents present in the Web, an IR system should collect, filter, store and manage documents, represent information within documents in a way to facilitate access, and retrieve information on documents that satisfy users information needs.

Figure 1 presents the structure of an inverted index, a structure composed of a dictionary of words in order to facilitate and make agile the response of search queries. It maps terms to their occurrences in a document or set of documents. In its simplest model, the index stores all unique terms in a database based on hash table or binary search tree, containing the index and the occurrence of the term in each document. This is a technique known as in-memory inversion [Moffat and Bell 1995].

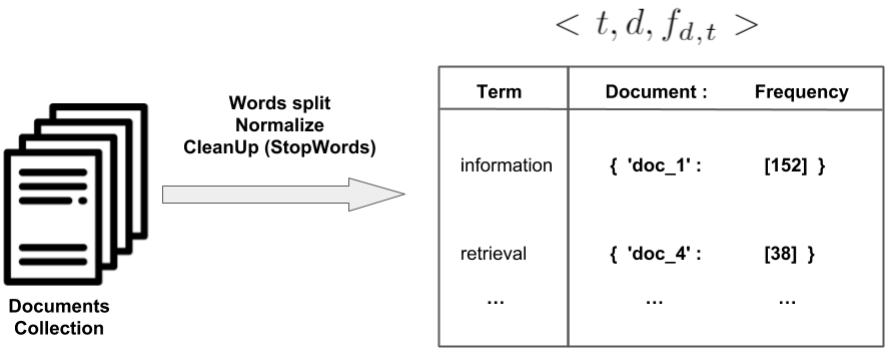

Figure 1. Invert index structure

Inverted indexes are the most popular data structure used in large-scale information retrieval systems, this aproach has some advantages, such as: speed of document search and simple implementation. The processing of large databases requires too much computational time due to the processing performed in the databases and operations performed in the index, requiring writing and reading operations on the disk due to limitations in the primary memory. According to Moffat and Bell (1995), the cost for creating an inverted index in memory in its simplest form is given by:

$$T = Bt_r + Ft_p + (t_d + t_r) \quad (1)$$

where T represents the inversion time, B, the size of the document on the disk, t_r, the transfer time, F, the number of words, t_p, the time required to perform parse, stem and look-up operations on the term and t_d, the compression and decompression time of the term.

In a context of high-performance computing, a parallel system is based on several and interconnected processing nodes (individual computers). Systems arranged in this way have the ability to divide tasks with high-processing consumption into available nodes. Thereby, it is possible to affirm that, when a task is divided into smaller pieces it can be executed in parallel, thus bringing significant gains in performance for the application.

In the same way, aiming to increase performance, current systems can take advantage of the characteristics that allow the parallelization of processes. A problem can be broken into smaller pieces and executed in parallel by multiple cores on a single computer. The need to perform tasks in parallel arises from limitations in the hardware, since processors are limited to their operation frequency and number of cores.

Following the line of reasoning, computation of complex problems can also take advantage of processing through graphics processing units (GPUs). This type of hardware have favorable characteristics, such as the potentially large number of instructions executed per clock cycle and high potential for multithreading execution, which make them faster for performing calculations. Thus, the development of APIs to extract the performance of GPUs has become necessary, such as the Compute Unified Device Architecture (CUDA), which is a platform for parallel computing that allows code parallelization in order to extract higher performance from applications.

3. Related Work

Addressing the theme of parallel web queries and inverted index, Pasari, Chaudhari, Borkar and Joshi (2016), Cho and Garcia-Molina (2002) and Wei and Jaja (2012) proposed methods to perform the web search in less time and with better performance.

Pasari, Chaudhari, Borkar and Joshi (2016) developed a parallel version of a vertical search algorithm. The goal was to bring more relevant and faster results to web-based queries. The parallel version was tested varying from 2 to 5 nodes and compared to the sequential one. The parallel proposal with 5 nodes is up to 6 times faster than the sequential one. Given the results, the creation of parallel algorithms for search engines has superior efficiency, and therefore, will completely replace the conventional versions.

Cho and Garcia-Molina (2002) present the creation of metrics to evaluate the viability of creating web crowlers in parallel. The authors goal is to propose a parallel algorithm to scan web pages faster in order to meet the large volume of data generated daily on the Internet.

Wei and Jaja (2012) present a new strategy that explores the current architectures of multicore processors, using pipeline and clustering. The tests were performed in a 32-node cluster, achieving a throughput of 280 MB/s on a single node and 6.12 GB/s in that

cluster, for a 10 Gb/s Ethernet interconnection. According to the authors, these results represent gains over the best known MapReduce algorithms.

In addition to the parallelization of web queries, research works have developed the parallelization of inverted index algorithms [Zhengyou and Tao 2009, Melink et al. 2001, Guana and Davidson 2012]. Melink, Raghavan, Yang and Garcia-Molina (2001) present a solution to reduce the time required for the generation of the inverted index in large collections of Internet pages. They propose the creation of a pipeline composed of three stages, Distributors, Indexers and Query servers to optimize the processing of the bases. The results prove that for large base sizes, the technique can accelerate the construction of the index in several hours.

Zhengyou and Tao (2009) proposed a new distributed parallel algorithm for constructing inverted web page indexes (called *P_Indexer*), applying a distributed parallel Middleware called ProActive. The algorithm was applied in a cluster computing system and the results showed that the algorithm has high efficiency and good scalability. The parallel algorithm with 4 processors was compared to the sequential one, increasing the collection size of web pages, and it is approximately 3 times faster than the sequential one.

Celikik at al. (2009) present in their work a method for the creation of a half inverted index. The approach reduces the document creation time by half when compared to state-of-the-art algorithms.

Guana and Davidson (2012) implemented and compared parallel data architectures for inverted index algorithms using the MapReduce programming framework in two different parallel programming systems: Hadoop and Message-Passing Interface (MPI). The results show that the use of a combiner optimizes the inverted index calculation in 56% and 10% in comparison to the original MPI and Hadoop strategies and in 50% to complete the map phase.

There are also research works that aimed parallel inverted index using GPU [Wei and JaJa 2012, Jung et al. 2017, Zhou et al. 2018]. Wei and Jaja (2012) presented a heterogeneous platform that combines multicore CPU and highly multithreaded GPU. The transfer rates of this approach are superior to the best-known algorithms, even when comparators are executed in large clusters.

Jung et al. (2017) developed a document inversion algorithm using CUDA, almost 3 times faster than a sequential system in two sets of tests performed. Zhou et al. (2018) developed GENIE, a generic inverted index structure for reduce the programming complexity for GPU in order to search for similarity of different data types. The tests in different databases show the efficiency of the proposed open source framework, reducing up to 3 times the execution time when comparing with your competitors.

Unlike the papers presented in this section, our proposal divides the database into several parts and distributes them on a computer cluster for inverted index creation. Each node has its code executed in parallel by CPU and the dictionary triples sorting by GPU. Thus, our proposal exploits three different parallel architectures in order to achieve high-performance computing. Table 1 shows a comparison between related works and our proposal.

Table 1. Overview of related work and our proposal

Related Work	CPU	Cluster	GPU
[Pasari et al. 2016]	X	-	-
[Cho and Garcia-Molina 2002]	X	-	-
[Celikik and Bast 2009]	X	-	-
[Melink et al. 2001]	X	-	-
[Zhengyou and Tao 2009]	-	X	-
[Guana and Davidson 2012]	-	X	-
[Wei and JaJa 2012]	X	X	-
[Zhou et al. 2018]	-	-	X
[Jung et al. 2017]	-	-	X
[Wei and JaJa 2012]	X	-	X
Our proposal	X	X	X

4. Proposed Architecture for Inverted Index

Our proposed architecture is based on a parallel system composed of multiple nodes illustrated in Figure 2. The flow is given as follows:

1. The data to be processed are divided into smaller packets by the centralizing node;
2. Each editing node receives a certain number of data packets, respecting the available memory limit. The number of nodes may vary depending on the amount of data being processed and the number of machines available;
3. Nodes use a hybrid approach to creating and sorting their dictionary in memory. They generate the inverted index in parallel via CPU according to the amount of cores available in the hardware. The inverted index, represented by triples $< t, d, f_{d,t} >$, is sorted in t by a sort algorithm, from a CUDA parallel approach. Where t represents the term, d the document and f the frequency of the term [Moffat and Bell 1995]. In addition, the information is processed from the Elias Gamma encoding in order to decrease the size of the dictionary;
4. The index is sent to a FIFO queue on the central node;
5. The central node reads the queue and does the merge operation of the current element with the dictionary contained on the disk.

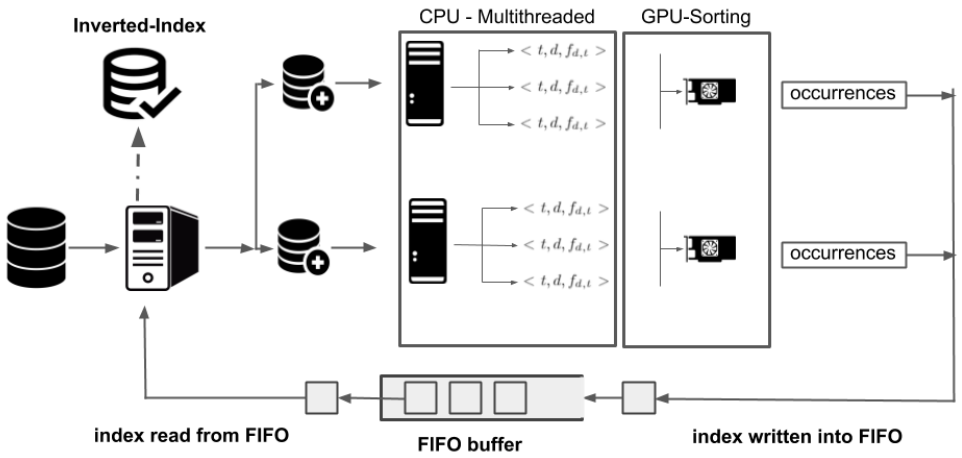

Figure 2. Architecture data flow

The architecture has three performance gain potentials: parallel cluster for index creation in multiple machines, parallel CPU for inverted index generation and parallel GPU for sorting the triples representing the inverted index. The processing nodes are only responsible for creating the index in memory and sending the results to a FIFO queue.

CPU parallelism tests were performed on a 4-core, 4-threaded Intel i5 6600K processor operating at a 4.5 GHz frequency. Code functions parallelized using the GPU were performed on a Nvidia GTX 1060 6Gb clocked at 1873 MHz with1820 CUDA cores.

The central node performs read and write operations to disk, centralizing the overhead generated by them, eliminating the impact of these operations on other nodes, however, adding network communication overhead. In addition, it performs the merge operation of the current queue element with the dictionary on disk. This node is responsible for performing the management of sending the instructions to the processing nodes, dividing the amount of operations into small pieces for parallel computing. The processing nodes are responsible for inverted index creation as presented by Moffat and Bell [Moffat and Bell 1995]. These nodes are responsible for reading the block sent by the central node, processing the data in memory and sending the packages containing the dictionaries to the FIFO queue.

The triples are previously sorted into t by the Quicksort algorithm via *CUDA JIT*, available from the Numba Python library. The functions were developed in python and then translated into PTX (Parallel Thread Execution) code and executed on CUDA hardware. Similarly, using GPU parallelism, the triple values are encoded in Elias Gamma, to decrease disk space for storage and the size of packets sent over the network. [Elias 1975]. The algorithm is executed sequentially and the result is compared to the proposed architecture. The parallelization criteria take into account the following aspects:

1. Stages with relevant amounts of writing, reading, and competition were streamlined by a FIFO queue and centered in one node;
2. Code parts with a large number of computations take advantage of GPU processing capabilities by a CUDA approach;
3. Chunks processed by In Memory Inversion approach was parallelized with multithread at each processing node.

5. Results

This section presents the results obtained through the analysis of experimental tests. The processing time is computed for each iteration and compared to other parallel variations. The database has a size variation to explore the Weak Scaling metrics as well as the number of threads executed on CPU to explore Strong Scaling.

The data used were samples extracted from the *WT10G*[1]. collection. Results were obtained from the average of 10 iterations. The portion of the algorithm responsible for reading the database and creating the dictionary was executed by a CPU with increasing number threads applied. The tests were performed from four databases of different sizes. The Table 2 illustrates the database sizes, the total number of words and the total number of unique words. The index sorting and Elias Gama encoding were processed by GPU for

[1]http://ir.dcs.gla.ac.uk. Access made on September 11, 2019

being CPU-Bound operations, taking advantage of hardware capabilities. The total size of the database for processing in GPU was 23.3 MB.

Table 2. Processed database information

Database size	Words	Unique words
15.0 MB	1,695,639	96,045
37.5 MB	3,750,514	185,274
43.1 MB	4,329,419	210,503
64.9 MB	6,376,600	291,359

5.1. Weak and Strong Scaling

By assigning a queue, a process is created for each core, where in each one the code is executed sequentially by a thread. In this way, the problem is divided into chunks, so each process can perform the execution of its workload independently. Figure 3 illustrates the performance gains when using 4 threads, were the workload ranges from 15 MB to 64.9 MB. The processing time of the sequential algorithm increases proportionately the workload variation. When dividing the workload (64.9 MB) into 4 threads running simultaneously, the performance is better, reaching a speedup by up to 2.5 times. This is due to the division of the workload per node and the division of the blocks sent to each node in smaller chunks for each available core on the system node.

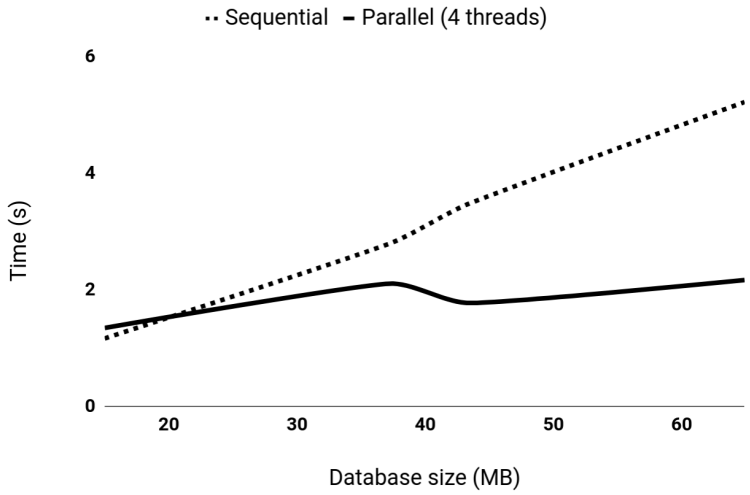

Figure 3. Application performance Weak Scaling.

Tests using the Strong Scaling concepts were performed on a 64.9 MB database described in the Table 2. The algorithm was executed sequentially and using the process queue, with nodes ranging from 2 to 6 cores.

The tests results are described in Figure 4. The sequential algorithm took 7.76 seconds to execute. With 2 nodes running the algorithm sequentially, the time dropped to 4.29 seconds, with 4 nodes to 2.01 seconds and 1.61 seconds with 6 nodes. By dividing processing into different machines running in parallel, the processing time is significantly reduced, with 6-node performance up to 4.8 times better than the sequential one. As the

number of nodes running in parallel increases, the workload is divided and the processing time decreases, showing that the system can scale as the number of nodes increases.

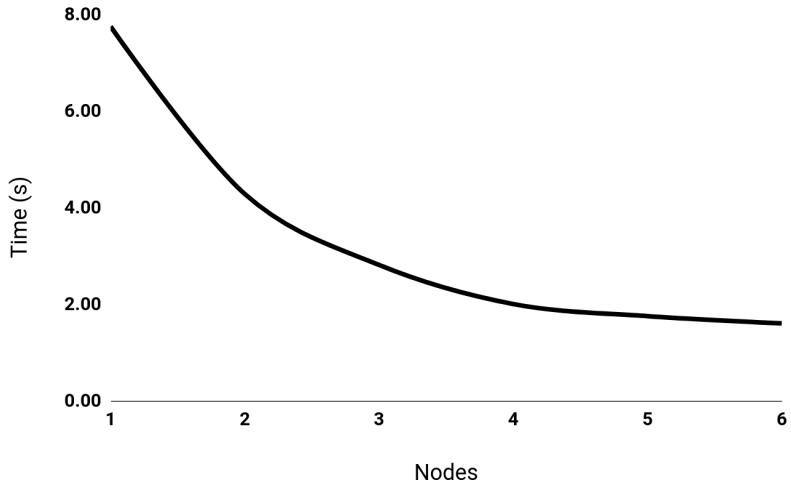

Figure 4. Application performance Strong Scaling.

Tests were performed taking into account only inverted index execution at the nodes, neglecting the data transfer time between the execution nodes and the central node. In extensive databases, the data transfer time represents a very low percentage of time compared to inverted index processing time at the nodes.

5.2. Total application performance

Figure 5 illustrates the algorithm performance in relation to the number of words processed per time interval. From the experiment, it was possible to note the superior performance of the parallel version using the process queue, since the workload is divided between nodes. The number of words processed per second increases proportionally to the number of nodes. It is also possible to note the reduction in the total execution time by the parallel approach. The number of words processed per second in the cluster can be calculated approximately by:

$$W = \sum_{i=1}^{q} (x_i * y_i) \tag{2}$$

given W as the total of words processed per second, q represents the total number of nodes in the cluster, x_i represents the number of cores that the node i has and y_i represents the number of words that each node i can process.

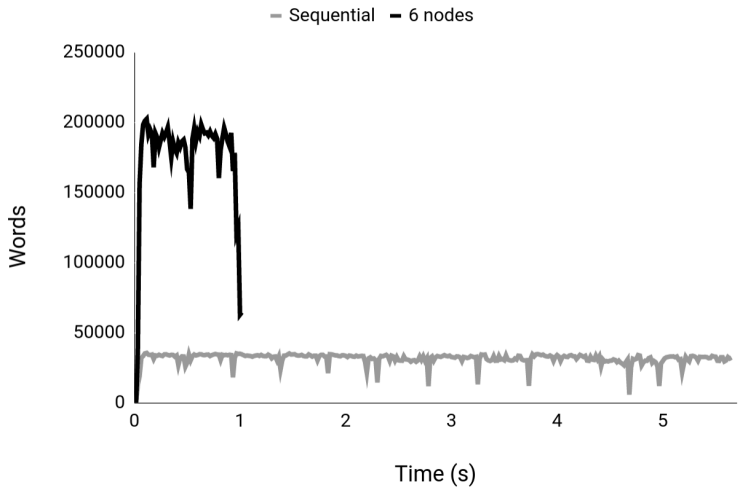

Figure 5. Words per second

Two measurement methods were performed:

1. The number of words processed per second has been counted. Disregarding the overhead of the measurement method.
2. The number of letters processed was counted and the value obtained was multiplied by 8 bits, which represents the space occupied by each letter within the document. This method is more accurate in performance evaluation. However, this method has a measurement overhead, considerably increasing the algorithm execution time, making this analysis unfeasible.

For the performance tests method 1 was used because it had lower overhead in the total execution time of the algorithm. Figure 5 shows the performance gain.

5.3. Heterogeneous Approach

CPU-bound operations require a large amount of processing, taking advantage of the potential for parallelism available in multi-core chips. In the algorithm analyzed, two operations demand a large amount of calculations and a relatively small amount of I/O operations: the code fragment responsible for performing the triples sorting and the part responsible for coding the triples of the dictionary in Elias Gamma.

Figure 6 illustrates the performance gain of sorting and encoding operations for the tests performed on a vector of 10 millions elements. These two steps have a lot of mathematical calculations, taking advantage of the parallelism offered by the GPU.

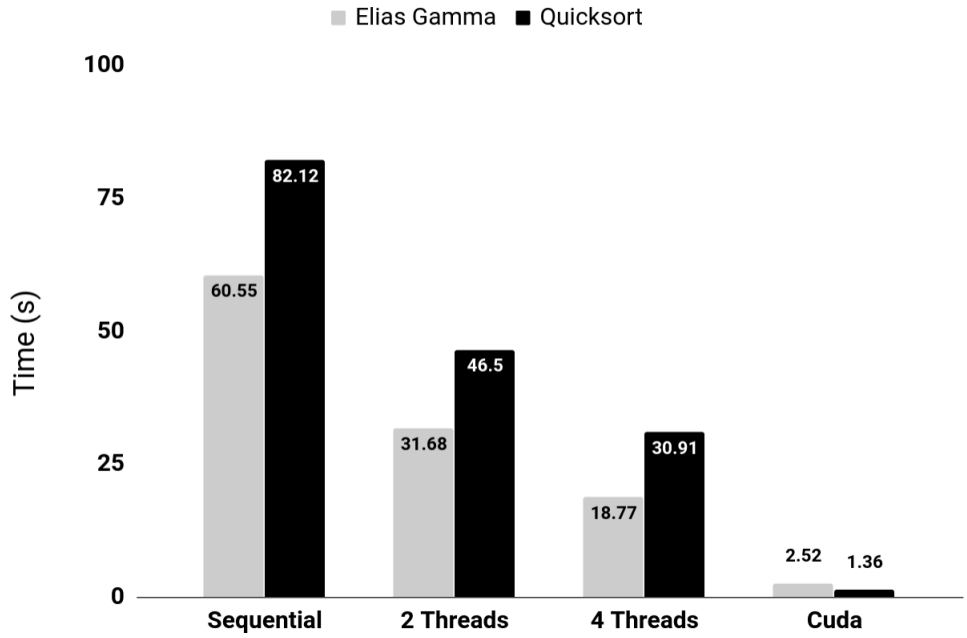

Figure 6. Elias Gamma and Quicksort - CPU and GPU (Cuda).

Figure 7 illustrates the comparison of CPU and GPU execution, applying the concepts of weak scaling by increasing the vector size from 1 million elements to 10 millions elements database. By increasing the database, there is a higher performance gain for GPU compared to CPU for both Elias Gamma and Quicksort.

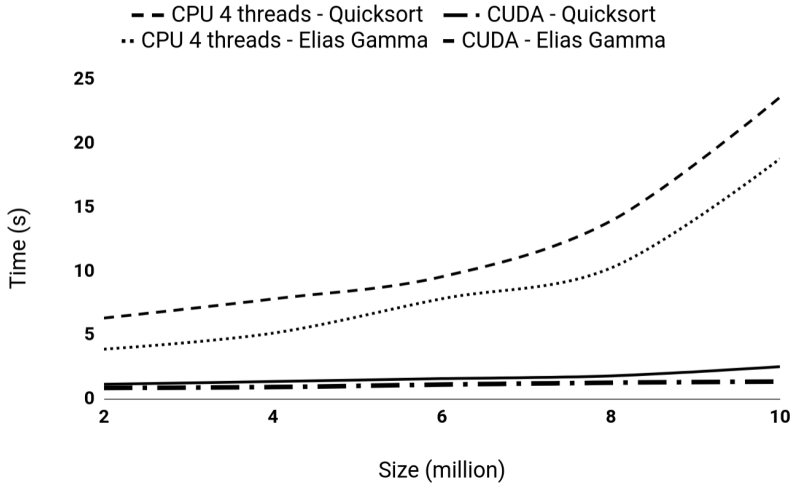

Figure 7. Comparison of performance running on CUDA and CPU.

Figure 8 shows the performance achieved by the parallel approach compared to the sequential version, achieving a speedup of approximately 60 times on a 10 millions elements database for Quicksort and more than 24 times on a 10 millions elements database for Elias Gamma encoding. It is possible to notice a growth curve, which indicates the superiority of the parallel version, which scales to larger bases.

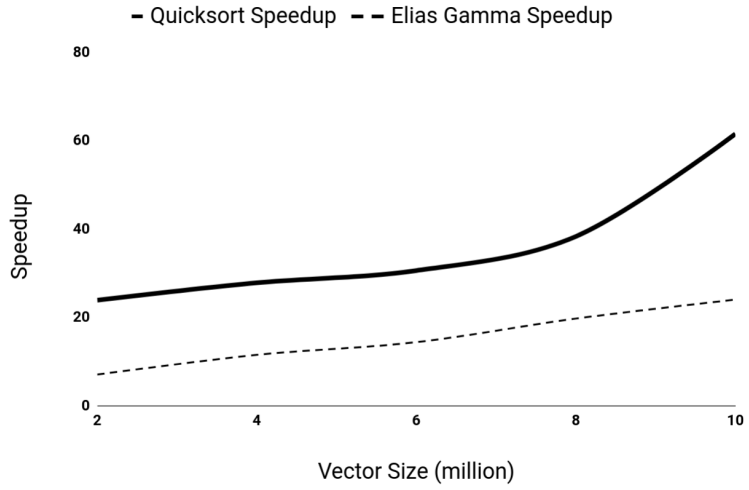

Figure 8. Speedup - *CUDA* vector sorting.

6. Conclusion

This paper presents a heterogeneous parallel architecture to inverted index creation. The tests show different approaches for inverted index parallelization. Thereby, it is possible to notice a significant gain in application performance. Thus, the proposed architecture can bring benefits in relation to the state-of-the-art, such as higher indexing speed, web page and document processing, and a scalable architecture.

It is possible to see that some types of parallel approaches do not have the capacity to withdraw the total expected execution performance. This phenomenon is due to different characteristics of the algorithms, such as I/O operations and the type of processing required by the algorithm. The proposed architecture circumvents this problem by centralizing operations on specific nodes. Thus, the processing nodes are free of competition, executing the algorithm version of the index inverted in memory, taking advantage of the speed that this method presents.

Another point observed in the tests and results is referring to heterogeneity, being a key piece computation in less time with better performance, since the use of accelerators, such as GPU, makes possible to reach a new level of speedup on calculations per second, as described in Section 5.

As future works, we intend to improve the performance by adding message-passing interface (MPI) to exploit a hybrid parallel programming based on C language also using CUDA and OpenMP for, e.g., inverted index processing in large databases in a supercomputer cluster, such as Santos Dumont at LNCC. For this, we intend to perform tests in a real database comparing the results with the state of the art, including evaluating the transfer and communication time between the systems.

Acknowledgment

The present work was carried out with the support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Financing Code 001. The authors thank CNPq, FAPEMIG, PUC Minas and REVEX for the partial support in the execution of this work.

References

Celikik, M. and Bast, H. (2009). Fast single-pass construction of a half-inverted index. In Karlgren, J., Tarhio, J., and Hyyrö, H., editors, *String Processing and Information Retrieval*, pages 194–205, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chen, C. P. and Zhang, C.-Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347.

Cho, J. and Garcia-Molina, H. (2002). Parallel crawlers. In *Proceedings of the 11th international conference on World Wide Web*, pages 124–135. ACM.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203.

Guana, V. and Davidson, J. (2012). On comparing inverted index parallel implementations using mapreduce. *University of Alberta*.

Jung, S., Chang, D.-J., and Park, J. W. (2017). Large scale document inversion using a multi-threaded computing system. *SIGAPP Appl. Comput. Rev.*, 17(2):27–35.

Kucukyilmaz, T., Turk, A., and Aykanat, C. (2012). A parallel framework for in-memory construction of term-partitioned inverted indexes. *The Computer Journal*, 55(11):1317–1330.

Melink, S., Raghavan, S., Yang, B., and Garcia-Molina, H. (2001). Building a distributed full-text index for the web. *ACM Transactions on Information Systems (TOIS)*, 19(3):217–241.

Moffat, A. and Bell, T. A. H. (1995). In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550.

Pasari, R., Chaudhari, V., Borkar, A., and Joshi, A. (2016). Parallelization of vertical search engine using hadoop and mapreduce. In *Proceedings of the International Conference on Advances in Information Communication Technology & Computing*, page 51. ACM.

Wei, Z. and JaJa, J. (2012). A fast algorithm for constructing inverted files on heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 72(5):728 – 738.

Wei, Z. and JaJa, J. (2012). An optimized high-throughput strategy for constructing inverted files. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2033–2044.

Zhengyou, L. and Tao, C. (2009). A distributed parallel algorithm for web page inverted indexes construction on the cluster computing systems. In *2009 International Forum on Information Technology and Applications*, volume 2, pages 33–36.

Zhou, J., Guo, Q., Jagadish, H. V., Krcal, L., Liu, S., Luan, W., Tung, A. K. H., Yang, Y., and Zheng, Y. (2018). A generic inverted index framework for similarity search on the gpu. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 893–904.