TuijinJishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 44 No. 4 (2023)

Analysis of Distributed Algorithms for Big-Data

Rajendra Purohit [1], K R Chowdhary [2], S D Purohit [3]
[1, 2] Dept. of Computer Science and Engineering, Jodhpur Institute of Engg. and Tech, Jodhpur
[3] Dept. of HEAS (Mathematics), Rajasthan Technical University, Kota
[3] Department of Computer Science and Mathematics, Lebanese American University, Lebanon

Abstract. The parallel and distributed processing are becoming de facto industry standard, and a large part of the current research is targeted on how to make computing scalable and distributed, dynamically, without allocating the resources on permanent basis. The present article focuses on the study and performance of distributed and parallel algorithms their file systems, to achieve scalability at local level (OpenMP platform), and at global level where computing and file systems are distributed. Various applications, algorithms, file systems have been used to demonstrate the areas and their performance studies have been presented. The systems and applications chosen here are of open-source nature, due to their wider applicability.

Keywords: Big-Data, Map-Reduce, OpenMP, parallel-processing, distributed processing.

1 Introduction

DUE to the fast growth of the Internet of (IoT), it is expected that by 2029, more than 20 billion items, from smartphones to wearable and simple monitors, will be connected to the Internet [2]. Distributed Ledger Technologies (DLTs), which are new and have been used to solve the problem of communication in distributed systems, link these devices together. Till recent, when dealing with issues that involve huge volumes of data, the only workable approach requires the computations to be split out over a number of different computers. In conventional parallel programming model (e.g., MPI), the developer is responsible for managing concurrency explicitly. The Map-Reduce solution is appealing because its functional abstraction helps in providing a paradigm that is easy to grasp, for designing of scalable and distributed algorithms [3]. The Map-Reduce algorithm is predicated upon the observation that many information processing jobs have a common underlying structure, such as the processing of web pages, wherein partial outputs are generated and subsequently combined. Map-Reduce stores data units on the local disks of computers within a cluster employing a distributed file system. Hadoop, an open-source implementation of the Map-Reduce programming standard, and the Hadoop Distributed File System (HDFS) are two components of the Hadoop ecosystem – an open-source distributed file system (DFS) that contributes the fundamental storage substrate, were used in the experiments conducted as part of this work.

2 Distributed Algorithms

From the beginning of distributed computing, the study of scheduling strategies for parallel processing has remained the most active areas of research. Utilizing scheduling heuristics which approximate an optimal schedule has remained a prevailing practice. However, it is impossible to compare the efficacy of various heuristics using analytical methods. One of options is to conduct back-to-back tests on actual platforms. The feasibility of this is evident in platforms with tightly coupled, however, it is not possible on modern distributed platforms, e.g., those which use Grid based processing, due to the labor-intensive nature of the process and the inability to replicate experiments. Thus, the solution is to opt for the simulation approach, which not only allows us to replicate the experiments but also permits us to investigate a vast array of platforms and applications. The works presented in this article make use of the Map-Reduce and Hadoop distributed framework, which facilitates the simulation of distributed computing applications utilizing numerous algorithms.

There are in fact many network simulators, like, NS, DaSSF, or OMNeT++, all of them concentrate on simulation of packets traveling in a network instead of on the network behavior as experienced by the application. In fact, none of existing simulation framework satisfy the requirements of distributed system other than Map-Reduce, which is open source, and allows simulation of random changes in performance, like those that happen when real resources are used in the background load.

3 Map-Reduce Architecture

Two systems became common in response to the data explosion, are as following, **a).** The Google file system (GFS), and **b).** Map-Reduce. The former was a pragmatic solution to managing Exabyte-scale data using commodity hardware, whereas the latter was an implementation of a time-tested design pattern applied to massively parallel processing on commodity machinery.

In the open source community, ***Hadoop*** subsequently consisted of Hadoop Distributed File System (HDFS) and the Hadoop Map-Reduce (HMR). However, it is a Map-Reduce system at its essence. The codes are converted into ***map*** and ***reduce*** tasks, which are executed by Hadoop.

The Map-Reduce architecture is based on a loosely coupled design. The pre-processing engine of Map-Reduce is not dependent on its storage architecture; due to this the processing and storage layers can be scaled independent of each other. Its storage system typically consists of a Distributed File System (DFS), like, the Google File System (GFS) [5] and that used in Hadoop Distributed File System (HDFS) [4]. The HDFS is a Java implementation of Google File System. Based on the DFS partitioning strategy, the data units are divided into equal-sized chunks and distributed across a cluster of processors. Each data segment is input to a ***mapper***. Thus, if data-set is divided into ***k*** pieces, the Map-Reduce will generate ***k*** number of mappers to process the data.

The Map-Reduce engine for processing comprises two types of nodes: 1. ***master nodes*** and, 2. ***worker nodes*** (Figure 1). The master node controls the execution flow of duties at the worker nodes through a scheduler module. Each worker node is in-charge of a map reduce operation. Each map worker node consists of a Input module, Map process, Map module, combine module, and partition module, whereas each Reduce worker node consists of the Reduce process, Group process, Reduce module, and Output Module.

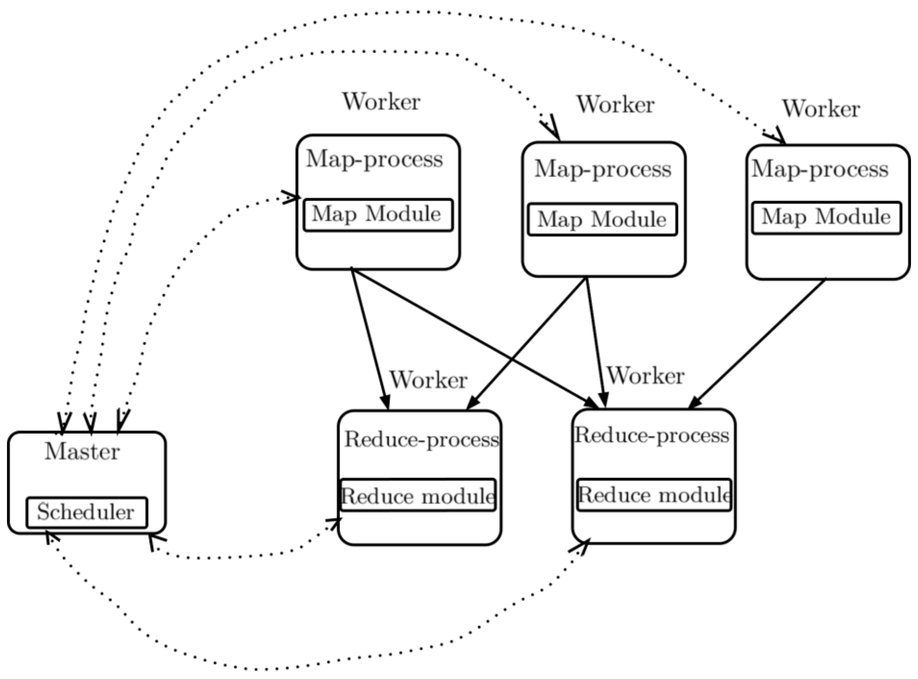

Figure 1: The Map-Reduce Architecture

The scheduler distributes map and reduce jobs to worker nodes in a cluster, based on the criteria of locality of the data, current state of the network, and other variables. The Map module will read through a data chunk and then calls the user-defined map function to handle the data that was input. Once the intermediate results are generated, which are a set of key/value pairs, it arranges the results according to the partition keys, sorts the tuples within each partition, and notifies the master about positions of the results.

Once notified by the master the Reduce module retrieves data from the mappers. After obtaining the intermediate regrouped. Each key/value pair is then subjected to the user-defined function, and the results are sent to output to HDFS.

Given its goal of scalability across a large number of working nodes, in which Map-Reduce system must efficiently support fault-tolerance. When any of the map or reduce task fails, a new task gets created on a different machine that will resume and rerun the unsuccessful task. Since the results are locally stored in the mapper, even a map task that has been accomplished must be executed again in the event of a failed node. However, since the reducer retains the results in DFS, it is not necessary to re-execute a completed reduce task when a node fails.

Map-Reduce Applications: One of the important application of Map-Reduce in the area of big-data is, indexing search-engine's index, to be later used in web search by Google and other search engines. The indexing system gets a lot of documents from the crawling system, which are kept as GFS files (Glarysoft Split File) files. The indexer takes input from these files and produces Google index in a distributed fashion. The salient features of this indexing are: simpler indexing code – due to the fact that distribution and parallelization is hidden within the Map-Reduce library. Apart from this, scalability is assured as new machines can be added dynamically, as part of the indexing cluster.

4 Map-Reduce Algorithm

Hadoop is a free, community-developed software framework for analyzing and processing large data-sets using the algorithm on a cluster of distributed computing nodes. By using the Map-Reduce programming model, we can distribute and process large data-sets in parallel. Before , the data set had to be partitioned, with each section being assigned to a different processor and the final results being integrated.

4.1. Traditional Method

Utilizing the conventional approach, the algorithm that will be employed is as follows: The text should be divided into blocks that are about equivalent in size to the number of available computers or nodes. Subsequently, each individual element is concurrently arranged, eliminating any duplicate words within each partition, followed by the merging of the outcomes, which are subsequently transmitted to the output (Figure 2).

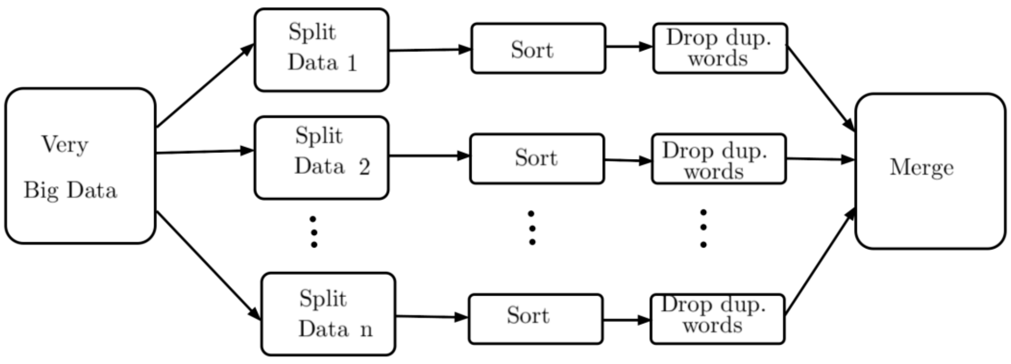

Figure 2: Traditional processing

However, it has following drawbacks:

- If any of the machine delays the job, the whole is delayed;

➢ If any of the machine fails, the whole work suffers (called reliability problem);

➢ How to equally split the data?

➢ Fault tolerance needed (if any machine fails);

➢ Mechanism needed to produce output.

4.2. Map-Reduce Algorithm

A diagrammatic representation of the algorithm shown in Figure 3 overcomes all these drawbacks.

The execution of parallel-distributed processing is facilitated by three distinct classes[1].

[1] The term Class originated from the Java programming language, where it is utilised for the implementation of Map-Reduce.

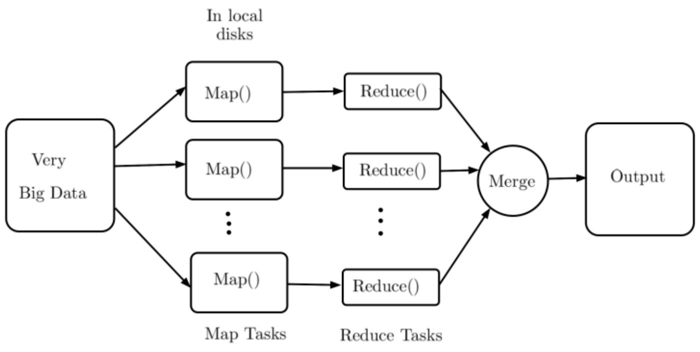

Figure 3: Map-Reduce Distributed Algorithm

Mapper Class: The system is comprised of two separate categories of jobs, namely ***Map and Reduce***. The reducer phase occurs subsequent to the completion of the map phase. The ***key-value*** pairs are produced as intermediate outcomes in the ***map ()*** operation by reading and processing blocks of data. The reducer blocks which are implemented using ***reduce ()*** function, receive key-value pairs from several ***map()*** jobs. The reducer function condenses the intermediate data tuples, to be specific, the key-value pairs which are obtained through subsequent map() operations (Figure 4) into a reduced assemblage of tuples or key-value pairs that becomes the ultimate result.

The mapper class comprises: 1) input split, each of which produces a single block of data which is assigned as a single map task in the Map-Reduce program, and 2) a record-reader which interacts with the input split task, and converts the obtained data in the form pf key-value pairs.

.

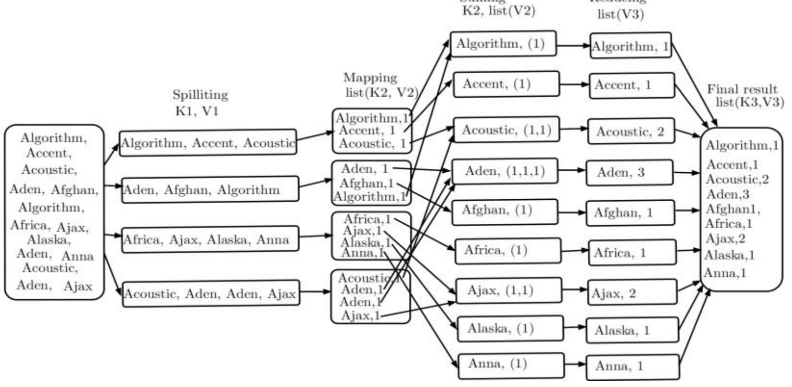
Spilliting K1, V1
Mapping list(K2, V2)
Suffling K2, list(V2)
Reducing list(V3)
Final result list(K3,V3)
Algorithm, Accent, Acoustic, Aden, Afghan, Algorithm, Africa, Ajax, Alaska, Aden, Anna Acoustic, Aden, Ajax
Algorithm, Accent, Acoustic
Aden, Afghan, Algorithm
Africa, Ajax, Alaska, Anna
Acoustic, Aden, Aden, Ajax
Algorithm,1 Accent, 1 Acoustic, 1
Aden, 1 Afghan,1 Algorithm,1
Africa,1 Ajax,1 Alaska,1 Anna,1
Acoustic,1 Aden,1 Aden,1 Ajax,1
Algorithm, (1)
Accent, (1)
Acoustic, (1,1)
Aden, (1,1,1)
Afghan, (1)
Africa, (1)
Ajax, (1,1)
Alaska, (1)
Anna, (1)
Algorithm, 1
Accent, 1
Acoustic, 2
Aden, 3
Afghan, 1
Africa, 1
Ajax, 2
Alaska, 1
Anna, 1
Algorithm,1 Accent,1 Acoustic,2 Aden,3 Afghan1, Africa,1 Ajax,2 Alaska,1 Anna,1

Figure 4: Word frequency counting through Map-Reduce

Reducer Class: Reducer classes are responsible for taking intermediate results from mapper classes and using them to create the final results, which are then written to the ***Hadoop Distributed File System (HDFS***).

Driver Class: The driver class assumes the responsibility of configuring and initiating a Map-Reduce job to execute within the Hadoop framework.

The advantage of environment is that it can utilize commodity computer clusters for massively parallel processing of data. A Map-Reduce cluster may scale to thousands of nodes while remaining fault-tolerant. One of the primary benefits of this framework is in the utilization of a straightforward and robust programming paradigm. Additionally, it frees the application developer from all the intricate complexities of managing a distributed program, including problems with data distribution, scheduling, and fault tolerance [6].

The distributed data processing system of Map-Reduce is based on the idea that parallel processing can be made easier by using a distributed computing platform. It has only following interfaces as: map and reduce. Typically, developers will create their individual map and reduce procedures, while the system is in charge of scheduling and coordinating the map and reduce tasks [1].

4.3. Map-Reduce Characteristics

The Map-Reduce approach may be utilised to address ”embarrassingly parallel” issues, which are those that take little or no effort to divide a work into a number of parallel but more manageable tasks. The Map-Reduce has applications in ***data mining, data analytics, and scientific computation***. The distinct qualities that it possesses are:

- **Flexible:** Without understanding how to run a Map-Reduce job in parallel, programmers can organise petabytes of data on hundreds of thousands of machines by writing simple maps and reducing functions.

- **Scalable:** One of the primary obstacles encountered in several contemporary applications is the ability to effectively handle and process the ever expanding quantities of data, which Map-Reduce can easily accommodate by allowing for data-parallel processing.

- **Efficient:** The Map-Reduce does not need loading of data into a database, hence saving the costs.

- **Fault tolerance:** It is the capacity of a system to continue working successfully despite the presence of faults or errors in the system. This capacity is referred to as fault tolerance. In the Map-Reduce, execution of each job involves the partitioning of tasks/jobs into numerous smaller units, which are subsequently distributed among multiple computers for processing. Failing of a task or machine receives compensation by allocating the task to a machine that is capable of managing the load. The job’s input is stored within a distributed file system, which maintains many replicas to provide a high level of availability. Therefore, the unsuccessful map work may be rectified by reloading the replica. The unsuccessful reduce job can be retried by retrieving the data from the finished map tasks again.

5 Programming Model for Map-Reduce

Map-Reduce use condensed parallel data processing approach, and runs on a cluster of computers. Its programming part comprises of two user defined functions: ***1. map, and 2. reduce***, as shown in Table I. The map function is provided with a collection of key/value pairs of data as its inputs. On submission of a job to the system, map tasks (called mappers) are initiated on the compute nodes. Every single map taskemploys the map function on each and every key-value pair that it processes ***(k1, v1)***. For a given input key/value combination it is possible to construct zero or more intermediate key/value pairs in the form of a list ***(k2, v2)***. These interim results are sorted by the keys and saved in the local file system in a distributed file system.

Table 1: The functions: Map-Reduce.

Function	Argument
map-function	(k1, v1) --> list(k2, v2)
reduce-function	(k2, list(v2)) --> list(v3)

On completion of all the map tasks, the engine informs about the reduce tasks to the reducer to start processing. The reducers are also processes, and They'll be picking up files generated by map tasks in concurrently, and then merges these files. This will be a merge-sort as the files are already sorted. The output of map tasks is used to combine the key/value pairs into a set of new key/value pairs ***(k2, list(v2))***. Here, the values that share the same key, denoted as ***k2***, are organised into a list and subsequently utilised as the input for the reduce function. The reduction function utilises a processing logic that is defined by the user to handle the data. The outcomes, often presented as a series of numerical numbers, are subsequently stored in the storage system.

Example of case study: Calculating total revenue for a website for each source IP. As an illustrative instance, considering the database UserVisits shown as database Table 2, a common task involves computing the aggregate sum of revenue, referred to as revnuSum, for each distinct source IP address that accesses the website. The approach is as follows: add up all of the money from advertising on all of the websites that are showing this product information. The website in question has a source IP and shows certain product information. These websites are scattered across different geographical locations.

The task is performed autonomously for all websites by employing parallel and distributed method.

Table 2: The user visit table.

ID->>	sourceIP	destIP	revnuSum
ID Type->	char(16)	char(32)	float
ID->	UserAgent	searchWord	duration
ID Type->	char(64)	char(32)	Int

A typical algorithms for its map and reduce functions are given as Algorithms 1 and 2, respectively.

Algorithm 1 Map-Function for Users Visits

```
1: input: charString key, value
2: charString[] array = value.split("j");
3: Emit Intermediate(array[0], ParseFloat (array [2]);
```

Algorithm 2 Reduce-Function for UserVisits

```
1: input: charString key, Iterator values
2: float revnue Sum = 0;
3: while values.Next()!= NULL do
```

```
4:      | revnue Sum += values.next();
5: end while
6: Emit(key, revnue Sum);
```

It is assumed that that the input data set is in text format, the tuples separated by lines, and columns are separated by the separator character “j”. Every mapper in the system parses the tuples that are assigned to it and produces a key/value pair in the form of (sourceIP, revnuSum). The outcomes are initially stored in the local disc of the mapper and subsequently transferred (called shuffling) to the reducers. During the reduction phase, the key/value pairs are organized into groups denoted as (sourceIP, (revnuSum1, revnuSum2, revnuSum3, ...)), where the grouping is determined by the keys, namely the sourceIP. In order to handle these pairs, each reducer summarises the revnuSum for a single sourceIP, and the resulting value (sourceIP, sum(revnuSum)) is created and returned.

6 Word Frequency Count Experiment

An experiment was conducted for ***word frequency count*** for a huge database (big-data), using the technique.

The obtained results were compared with alternative methodologies. Fig. 4 illustrates the detailed functioning of the system at a smaller scale. Considering that the following terms are present in the ***tokens.txt*** file: ***Algorithm, Accent, ., Ajax***. For the purpose of elucidation, we have taken a much-reduced form. The method uses the process depicted in Fig.4 to calculate the frequency of each word.

I. Work is first ***split*** into four parts so that it can be done on four different nodes.

II. Next, the ***mappers*** tokenize their share of the job in the form of ***list(K2; V 2)***, such that each word is a token with count 1.

III. Next, ***sorting*** and ***shuffling*** take place. The key ***K2*** is a token, and ***list(V2)*** is the combination of ***V2*** from the mapper for matched tokens. The key-value pairs ***(list(V 3))*** are sent to the ***reducer***,

IV. Lastly, the key/value pairs are gathered and written to the output, which is an HDFS file.

A. Big-data processing results

For this experiment, three files of 350 MB, 1 GB and as well as 2 GB all included token (keyword) sequences that had no particular order; this corresponds to the first box in Figure 4. The file tokens.txt was utilised for each size, but with varying sizes.

For Hadoop platform, a single java program was written to run on all the nodes, in the Hadoop distributed file system (dfs). A shell script “run.sh” initially builds a Hadoop distributed file system in a directory called “Count” to store the key-word counts, and the input database is tokens.txt. This shell script builds the Java program, runs it on Hadoop, and then shows the working time in seconds. This information is provided in second column of table III.

Table 2. The functions: Map-Reduce.

Platform –> File size ()	Hadoop (MR) (secs.)	Spark (secs.)	Hive (secs.)
ID Type->	char(16)	char(32)	float
ID->	UserAgent	searchWord	duration
ID Type->	char(64)	char(32)	Int

In addition, the Scala programming language was used, through its inherent capabilities, for the purpose of calculating the frequency of words. Scala utilizes a single node as opposed to a distributed file system such as Hadoop. The Scala program that was run on Spark machine (a variant of Unix/Linux), the times for three different big-data sizes are shown in the third column of table III, and the plots in Fig. 5 show the relative performance of data shown in the table.

The third result was achieved by performing sql queries on Hive platform, and the result in seconds are shown in the last column of table III.

Hadoop's use of the distributed file system (DFS) and ability to provide distributed processing, as opposed to the other two's usage of single-node programming, makes it significantly different from each of them. The increased duration of time required for executing Depth-First Search (DFS) might be caused due to delays in communications between Hadoop nodes.

B. OpenMP Results

The word-frequency count experiment was conducted out on the parallel processing platform of OpenMP for processor threads 1, 2, 3, and 4. Database file sizes of 350 MB, 1 GB, and 2 GB were tracked for each thread value. The computation time for word count for each pair of "thread-value X file-size" was determined. The times taken, for different database size as well as for different thread count, in seconds, for each category are presented in Table IV and in (Figure 6). The threads are effectively the processing elements, it shows an almost linear increase in performance as the number of threads increase, however, it slightly deviates from exact linear because the processor has devote itself to do other housekeeping jobs like memory management and processing switching.

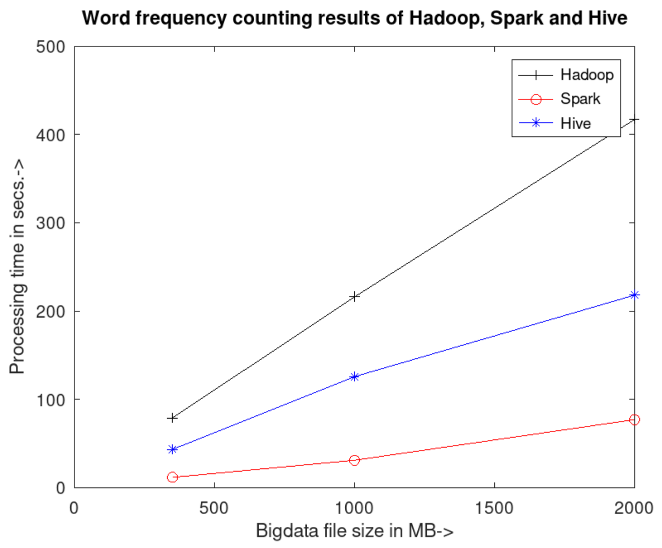

Figure 5: Word frequency counting result on Hadoop, Spark, and Hive

Table 3. The functions: Map-Reduce.

Threads –>	1 Thread	2 Thread	3 Thread	4 Thread
File size ()	(secs.)	(secs.)	(secs.)	(secs.)
Small (350 MB)	77.065	20.957	18.485	21.958
Medium (1 GB)	225.945	94.425	91.972	87.369
Large (2 GB)	469.34	201.428	156.469	147.031

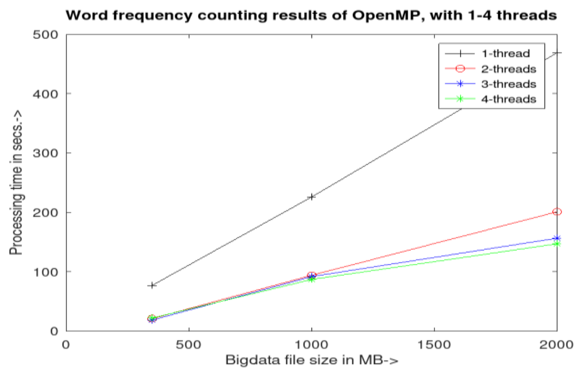

Figure 6: Word frequency counting on OpenMP 4-Core Machine

7 Conclusion

We have presented the application of distributed processing using algorithm, highlighting its simple structure, and have demonstrated its use for handling large (big) data. Its distributed working has been explained through a web application, and corresponding algorithms, where an advertisement agency bills from a product making company based how many times there has been mouse clicks on all the advertisements running geographically distributed manner on in distributed physical locations.

Apart from presenting its working in simple language, while comparing it with the traditional distributed processing and lattersits drawbacks, it has shown that this algorithm is scalable by increasing the size of big data from 350 MB to 2000 MB, with no degradation in performance. The truly distributed processing system Hadoop where is running, shows a almost 100 percent linear response as a function of size of big data.

References

[1] F. Li, B. C. Ooi, M. T. Ozsu, and S. Wu. Distributed data management using . ACM Computing Surveys (CSUR), 46(3):1-42, 2014.

[2] Qingyi Zhu, Seng W. Loke, Rolando Trujillo-Rasua, Frank Jiang, and Yong Xiang. 2019. Applications of Distributed Ledger Technologies to the Internet of Things: A Survey. ACM Comput. Surv. 52, 6, Article 120 (November 2020), 34 pages. https://doi.org/10.1145/3359982

[3] Jimmy Lin. 2009. Brute force and indexed approaches to pairwise document similarity comparisons with . In Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '09). Association for Computing Machinery, New York, NY, USA, 155162. https://doi.org/10.1145/1571941.1571970

[4] Shvachko K., Kuang H., Radia S. and Chansler R., "The Hadoop Distributed File System," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 2010, pp. 1-10, doi: 0.1109/MSST.2010.5496972.

[5] Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003) The Google File System. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 2943. https://doi.org/10.1145/945445.945450

[6] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. 2013. The family of and large-scale data processing systems, ACM Comput. Surv. 46, 1, Article 11 (October 2013), 44 pages.https://doi.org/10.1145/2522968.2522979

[7] Chowdhary, K.R., Purohit, R., Purohit, S.D. (2023). Diagnosing Distributed Systems Through Log Data Analysis. In: Bansal, J.C., Sharma, H., Chakravorty, A. (eds) Congress on Smart Computing Technologies. CSCT 2022. Smart Innovation, Systems and Technologies, vol 351. Springer, Singapore. https://doi.org/10.1007/978-981-99-2468-438