Multimed Tools Appl (2014) 69:513–537
DOI 10.1007/s11042-012-1283-x

Distributed media indexing based on MPI and MapReduce

Hisham Mohamed · Stéphane Marchand-Maillet

Published online: 15 November 2012

Abstract Web-scale digital assets comprise millions or billions of documents. Due to such increase, sequential algorithms cannot cope with this data, and parallel and distributed computing become the solution of choice. MapReduce is a programming model proposed by Google for scalable data processing. MapReduce is mainly applicable for data intensive algorithms. In contrast, the message passing interface (MPI) is suitable for high performance algorithms. This paper proposes an adapted structure of the MapReduce programming model using MPI for multimedia indexing. Experimental results are done on various multimedia applications to validate our model. The experiments indicate that our proposed model achieves good speedup compared to the original sequential versions, Hadoop and the earlier versions of MapReduce using MPI.

Keywords Distributed multimedia indexing · MPI · MapReduce · Distributed inverted indexing · Permutation-based indexes · Distributed approximate similarity search

H. Mohamed (✉) · S. Marchand-Maillet
Viper Group, Computer Vision and Multimedia Laboratory, University of Geneva, 7 Route de Drize, Geneva, Switzerland
e-mail: hisham.mohamed@unige.ch

S. Marchand-Maillet
e-mail: stephane.marchand-maillet@unige.ch

1 Introduction

Web-scale digital assets comprise millions or billions of documents. This huge amount of data needs to be indexed, stored, analyzed and visualized to allow easy access and extraction of information to benefit to a large public. Current sequential algorithms cannot handle such volume. Hence, the role of parallel and distributed computing become more important to help in assisting the management of large volumes of data.

Google is the leading search engine and uses a cluster of thousands of machines to handle their data in parallel. In 2004, Google proposed the MapReduce [5] programming model for large data processing in parallel. MapReduce is composed of only two functions, the first one is the *map* function which processes *(key,value)* pairs to generate intermediate *(key,value)* pairs. These intermediate pairs are grouped together with respect to their *key* to produce *(key,list(values))* tuples. Then, each tuple is passed to a *reduce* function, which does some analysis. There are many applications with a similar workflow including, inverted indexing [17], k-means [9], sorting and PageRanking [5].

Hence, MapReduce is suitable for data intensive applications; data is divided into groups and each machine processes independently its part. In contrast, MPI is a message passing library designed to function on parallel machines. MPI launches independent processes of an algorithm on each machine, in which the processes are connected using MPI by moving data from the address space of a certain process to another efficiently. MPI supports collective operations, remote memory access and parallel I/O. MPI is the abbreviation of Message Passing Interface and it is useful for high performance applications. It has many implementations like MPICH2 [19] and OpenMPI [8], but all of them follow MPI standards [18].

Most of the current applications depend on two types of parallelization: data parallelization and algorithmic parallelization [15, 23]. From there, it seems sensible to build a library supporting the two types. In the same library, we can combine the advantages of the MapReduce programming model with the efficient communication capabilities of MPI. We propose the MRO-MPI model (MapReduce overlapping using MPI) an idea to speed up the MapReduce model by avoiding its bottlenecks using MPI.

In the original implementation [5], the *reduce* function has to wait for the *map* function to finish before it can start processing. If the *map* function is slow for any reason, this will affect the whole running time as the reducers will wait. Our model is based on running the map and reduce functions concurrently in parallel by exchanging partial intermediate data between them in a pipeline fashion. Hence, the *map* and *reduce* functions works in parallel to achieve a good speedup. We have implemented our idea and applied it on three different applications. The first application is the *WordCount* example [5] to measure its effectiveness relative to the recent MapReduce implementations. The second application operates on a large number of text (XML) excerpts related to images from the ImageNet corpus. The third application for indexing high dimensional large scale multimedia data based on permutation indexes.

The rest of the paper is organized as follows. In the next section, we discuss the related work. A brief background about MPI and MapReduce is given in Section 3. In Section 4, we detail the main idea about our methodology. In Section 5, our results are presented for our three applications (Sections 5.1–5.3). Then, we conclude in Section 6.

2 Related work

2.1 MapReduce and MPI

The original MapReduce framework is detailed in [5]. Hadoop [26] is the most successful implementation of MapReduce. It is used in many companies like Amazon, Yahoo! and IBM. Hadoop is written in Java and supports parallel applications written in Java, Python and C. It also provides a distributed file system called (HDFS).

Hoefler et al. [12] made the first attempt to write MapReduce using MPI. They provided strategies for implementing MapReduce using blocking, non blocking and collective operations based on the original MapReduce model. They also built on MPI-2.2 and MPI-3 features to improve MapReduce. Plimpton and Devine [21] released the first public library for MapReduce using MPI. Their architecture exactly follows the original MapReduce model. Ahmad et al. [1] provide MaRCO, which is a plugin of Hadoop to provide communication overlap between the map, the shuffle, and the reduce phases to improve the performance. They were able to achieve good speed up comparing to Hadoop. Lu et al. [16] gave a good analysis on the Hadoop communication and implemented the original MapReduce model using MPI. The results showed that they obtained a good speed improvement comparing to Hadoop based on MPI communications. Jaliya et al. [7] proposed Twister, which is an iterative implementation of MapReduce. It is based on publish/subscribe messaging infrastructure for communication and data transfer. They achieved good performance comparing to Hadoop and other MapReduce implementations.

The main difference between our proposition and earlier work lies in the fact that we have modified the MapReduce model to achieve speed up using MPI.

By our idea, the map and reduce function works concurrently. The work done in [1, 7, 16] are similar to our model. However compared to [1], we emancipate from Hadoop as it has problems in connecting with C and C++ code. Authors in [16] provided an overlapping between the mapping and the communication phases, but without any details about the rate of sending the data, the ratio of mappers to reducers in case they are working together and the way of handling different applications. Concerning [7], we combined the main model with MPI, which is widely used in most of parallel algorithms now, and gives the programmer the ability to build an application that depends on data and algorithmic parallelization. In our model, we propose the idea of overlapping the map, the communication and the reduce phases with a more detailed policy for the communication and data exchange. We also compare our model to Hadoop and earlier implementations of MapReduce using MPI “MPI-MapReduce” [21], based on different applications.

2.2 Multimedia indexing

For experiments part, we use our model on different multimedia applications.

2.2.1 Inverted files

The inverted file is an indexing data structure for text collections [27]. It is composed of two elements: the dictionary and the inverted list. The dictionary is a lexicographically sorted list containing a list of unique terms appearing in the corpus. Every term is then associated with a list (posting list) containing the references to the documents where this term appears. Further, for each document, a weight is given that reflects the importance of the term into the document, in the context of the complete collection. The *TF-IDF* weighting scheme [27] is used in information retrieval and text-mining. This weight is a statistical measure used to evaluate how important a word is to a document in a corpus. Inverted files are also at the basis of a number of multimedia indexing strategies, aligned with the bag-of-keypoints representation model [4]. We are developing a large-scale interactive multimedia retrieval system based on learning user preferences [3, 28]. We wish to integrate an efficient distributed indexing strategy based on inverted files within our framework.

McCreadie et al. [17] show a good analysis and different ways of building inverted index using MapReduce and Hadoop. We will use one of their methodologies, but based on our proposed MRO-MPI model against Hadoop.

2.2.2 Approximate similarity search

The way of answering users' queries depends on the search scenario. The "exact match" scenario is commonly used, where the system retrieves all matches to a given query from the database. Nowadays, this way of answering the query is not the most useful for some applications such as text plagiarism to track the similarity between an article against a database of texts, multiple genome comparison to find all the similarities between one or more genes, and multimedia retrieval to find the most similar picture or video to a given example. The similarity search paradigm [14] is more applicable on these models. For a query q and a data collection D, similarity search sorts all the data items by similarity to the given query according to a given distance function $d : D \times D \longrightarrow \Re$. The most relevant objects to the query are the k-top ranked objects (k-NN query) or the objects located within a distance range ρ from the query (range query). Several techniques have been developed for improving the performance of the similarity search problem [29]. One of the research topics still attracting interest is the scalability of similarity search for high-dimensional data. Different approaches have been proposed to attack the curse of dimensionality [24]. One of the most promising routes is the approximate similarity search [13, 20]. It proposes solutions to improve the performance when handling high dimensional data at the price of effectiveness.

Permutation-based indexes are the most recent technique for approximate similarity search [2, 10]. Here, We propose three distributed implementations for indexing *permutation-based indexes* using inverted files. We describe our ideas and have tested them on high dimensional datasets, which consist of millions of objects based on our proposed MRO-MPI model against normal parallelization using MPI.

3 MapReduce and MPI

3.1 MapReduce

The execution of the MapReduce model is done through four stages, as follows (see also Fig. 1):

1. Mapping: A user-defined map function M starts simultaneously in parallel on different machines. Each Map function is responsible for a chunk of input data. The map functions process these input data and emits intermediate *(key, value)* pairs : $M : (K_r \times V_r) \mapsto (K_m \times V_m)$. These intermediate pairs are saved locally. A master machine monitors all the mapping functions. When it receives a termination signal from all the mappers, it starts to assign reducers tasks by defining a key, or range of keys for each reducer to work on them.
2. Shuffling: Reducers communicate with the mappers through remote procedure calls to gather all the intermediate (K_m, V_m) pairs based on the key, or within the range of keys assigned to them. This process requires all-to-all communication as the intermediate keys are distributed on different machines.
3. Merging: Each reducer then starts to merge the pairs, based on identical intermediate keys to produce $(K_m, list(V_m))$ tuples.
4. Reducing: The tuples are passed to a user-defined reducing function R one by one to emit the output O, which is distributed on different machines: $R : (K_m \times list(V_m)) \mapsto O$.

Thus, MapReduce brings a simple and powerful interface for data parallelization, by keeping the user away from the communications and exchange of data, as this is directly handled by the framework. The user only needs to write the map and reduce functions.

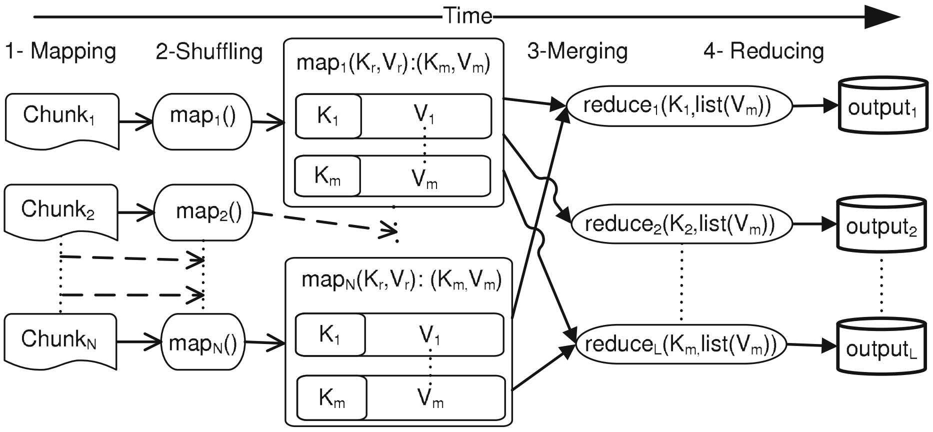

Time
1- Mapping
2-Shuffling
3-Merging
4- Reducing
Chunk$_1$
Chunk$_2$
Chunk$_N$
map$_1$()
map$_2$()
map$_N$()
map$_1$(K$_r$,V$_r$):(K$_m$,V$_m$)
map$_N$(K$_r$,V$_r$): (K$_m$,V$_m$)
K$_1$
V$_1$
K$_m$
V$_m$
reduce$_1$(K$_1$,list(V$_m$))
reduce$_2$(K$_2$,list(V$_m$))
reduce$_L$(K$_m$,list(V$_m$))
output$_1$
output$_2$
output$_L$

Fig. 1 MapReduce consists of four phases: *1* mapping, *2* shuffling, *3* merging, and *4* reducing

Table 1 Example of most common MPI functions

MPI Method	Description
MPI_Send()	Send data directly to a certain process
MPI_Recv()	Receive data from a certain process
MPI_Gather()	Gather data from different precesses
MPI_Scatter()	Scatter data on the processes
MPI_Bcast()	Broadcast data on all processes

3.2 MPI

MPI is a message passing standard for parallel programming [11]. A program written with MPI runs on all machines concurrently as a separate processes, where each process has a unique *rank*. Within the program, the programmer has to define the work done by each process and how the processes communicate with each other. MPI supports point-to-point, one-to-all, all-to-one and all-to-all communications. Table 1 shows examples of commonly used functions.

In general, MPI is suitable for solving dependent algorithms where machines need to exchange data during the execution. Full understanding of parallel programming and the library is required in order to use MPI. In contrast, MapReduce is suitable for independent algorithms where there is no need to exchange data between the machines during the execution. MapReduce is based on a number of sequential steps, but each step run over a number of machines in parallel to get maximum performance.

4 Map-Reduce overlapping using MPI (MRO-MPI)

4.1 MapReduce and MPI bottlenecks

The original model for MapReduce [5] has at least three bottlenecks:

Dependence: the reducers cannot start before the mappers are done, which affects the total performance of the system. In case one of the mapping functions is slower than the others, all the reducers have to wait.

Disk access: writing the intermediate (K_m, V_m) pairs during mapping, and reading these pairs again during reducing influences the performance especially if the emitting is done at a high rate.

All-to-All communication: excessive communication between all the machines needs to be done during the shuffling phase to create the $(K_m, list(V_m))$ tuples. The running time of this phase depends on the number of intermediate pairs.

At the same time, MPI is missing the simplicity; it is not easy to handle. The parallelization of a sequential application using MPI requires reconstruction of the algorithm. This means that the programmer has to specify the work done by each process and how these processes communicate among each other. That requires rearranging the functions and the values in order to get the best performance. Also, sending many small chunks affects the performance, which sometimes become slower than the sequential algorithm, because of the communication latency and the bandwidth limitations.

In our MRO-MPI model, we override the main bottlenecks from the original MapReduce model to get better performance, and at the same time we maintain the usability and the simplicity of MapReduce and keep the user away from MPI complex functions.

4.2 MRO-MPI model

The main idea behind our model is to overlap the mapping and the reducing phases by sending partial intermediate data in a pipeline fashion. Simply, when available, each mapper sends partial intermediate $(K_m, \mathit{Plist}(V_m))$ pairs to the responsible reducers. The reducer then works on this partial data and waits for more data, until all the mappers are done. With this model, we rule out the multiple read/write. As the mapper continuously sends the partial data directly to the reducers, there is no need to save intermediate data locally. Hence, the shuffling phase is merged with the mapping phase instead of doing it on separate step. The main gain is the simultaneous execution of the map and reduce functions. As opposed to the original model, the reducers do not wait until the mappers finish their work, which diminishes the running time and gives a good speed up as demonstrated in Section 5.1. Figure 2 illustrates our idea of overlapping.

4.3 Technical implementation

In MPI, the sender has to define the *rank* of the process that receives the data; which is a unique integer number assigned to each MPI running process for identification. In addition to that, the type of the sent data should be defined. At the same time, the receiver has to be ready and informed about the received data. Thus, this decreases the usability of MapReduce using MPI if we left it to the user. In our prototype, we keep the user away from the MPI complexities. Figure 3 shows the architecture of our prototype. The Map and Reduce sides are divided into two parts; user and system. The user side is the part where the programmer writes the mapping function. The system side is the part which is responsible to handle the communication and data merging. Our model based on three steps as follows:

Mapping and Shuffling: The mapping is exactly like the original one. The map function emits (K_m, V_m) pairs. The K_m of each pair is passed to a partitioning function:

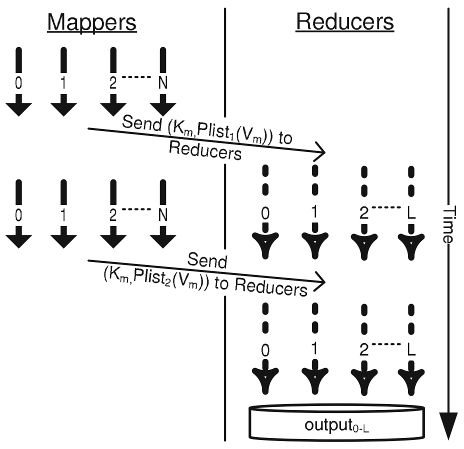

Mappers
Reducers
0
1
2
N
Send $(K_m, Plist_1(V_m))$ to Reducers
Send $(K_m, Plist_2(V_m))$ to Reducers
L
Time
output$_{0\text{-}L}$

Fig. 2 MRO-MPI: the mappers and reducers work in parallel and partial data is sent in a pipeline fashion

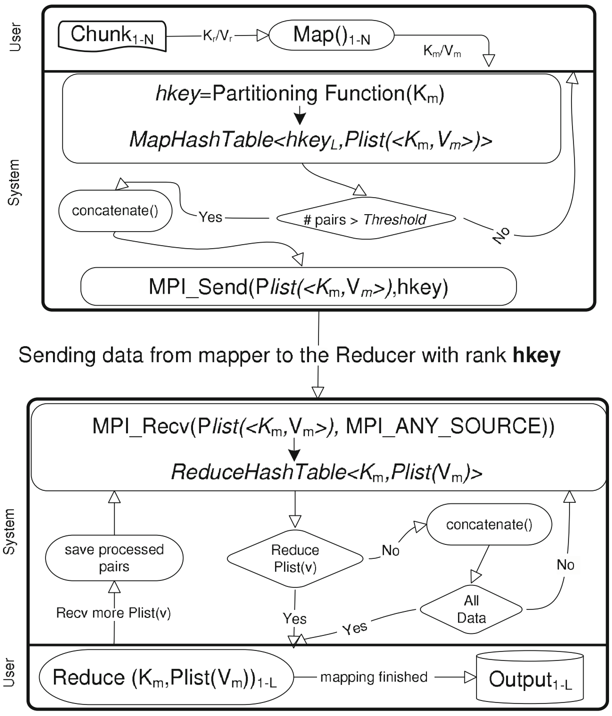

User
Chunk1-N
Kr/Vr
Map()1-N
Km/Vm
System
hkey=Partitioning Function(Km)
MapHashTable<hkeyL,Plist(<Km,Vm>)>
concatenate()
Yes
# pairs > Threshold
No
MPI_Send(Plist(<Km,Vm>),hkey)
Sending data from mapper to the Reducer with rank hkey
MPI_Recv(Plist(<Km,Vm>), MPI_ANY_SOURCE))
ReduceHashTable<Km,Plist(Vm)>
System
save processed pairs
Reduce Plist(v)
No
concatenate()
No
All Data
Yes
Yes
Recv more Plist(v)
User
Reduce (Km,Plist(Vm))1-L
mapping finished
Output1-L

Fig. 3 MRO-MPI: technical details

Partitioning : $(K_m) \mapsto (hkey_l)$. The partitioning function is used to distribute the keys on the reducers. The output range of the function is based on the number of the reducers; for L reducers the range is: $0 \rightarrow L$. The default function is a *"Hash"* function, which is similar to *hashPartitioner* in Hadoop, but also it can be replaced by user-defined hash function. For the main *"Hash"* function, hash collisions occurs as the number of keys is more than the number of the reducers, but it happens with equal distribution to roughly make equal load balancing of the keys on the reducers. The $(hkey_l, (K_m, V_m))$ pairs are saved in a local hash table for each mapper. Each $hkey_l$ is associated with a list of various (K_m, V_m) pairs:

$$MapHashtable : (hkey_l) \rightarrow ((K_i, V_1), (K_i, V_2), (K_{i+1}, V_1), \ldots (K_m, V_m)).$$

There are l counters C_{0-l}, each counter is assigned to a hash key $hkey_l$. The counters are used to count the size of pairs associated to each $hkey_l$. Every time a new pair is emitted, counter C_{hkey_l} is incremented by the size of the emitted *(key,value)* pairs and then the counter is checked if it is greater than a *threshold* value T, which is user-defined depends on the network connection. If so, the partial data is concatenated as one chunk and sent directly to the responsible reducer which has a rank $hkey_l$. Sending Data in MPI is based on the MPI Data types like MPI_INT, MPI_CHAR,..., etc. [11]. Additionally, MPI gives the user the ability to construct his own data types based on the original ones, which is called "derived data types" [11]. In our prototype, we have tested concatenating the pairs into one *CHAR* array against defining our MPI derived data type of a simple structure consists of a *CHAR* array as a word and an *INT* as a value. We found that good performance is achieved when the data is sent as a *CHAR* array (see Section 5.1). Hence, we combine all the data as one *CHAR* array and send it to the responsible reducer.

Receiving and Merging: All Reducers are actors, which means that they are ready to receive pairs from any mapper. The received data contains multiple intermediate various (K_m, V_m) pairs. These pairs are then organized with respect to their key using a hash table. The key of this hash table is the intermediate key and the value is a list of intermediate values received in this partial list and related to this key:

$$ReduceHashTable : K_m \rightarrow Plist(V_m).$$

Reducing: In reducing, we have two scenarios we enumerate them as partial and full reducing. In partial reducing, the reduce function can process partial intermediate data $(K_m, Plist(V_m))$(e.g. sum function). For full reducing, the reduce function needs the full intermediate data $(K_m, list(V_m))$ (e.g. max or average functions). Hence, we set another parameter to the user to define how the reduce function processes the intermediate pairs (e.g. full or partial).

For the two scenarios, after saving the data or the partial results, the system calls the receiving function again and this process continues until all the data is received from the mappers. When mapping is done and last partial data are reduced, output data are saved on the local hard disk of each reduce process.

Hence in our model, the three phases run in parallel on different machines and continue until the mapping is done. The user has to define the number of mappers, the number of reducers, the reducing type and the threshold value T. The ratio between the mappers and the reducers affects the performance of the model. A good ratio between the mappers and the reducers with analysis and the effect of changing the T value are given in the next section based on different applications.

5 Experimental results

We have conducted large-scale experiments to test the validity of our model. Our results section is divided into three subsections. In Section 5.1, We have implemented the *WordCount* example with our MRO-MPI model and compared it to Hadoop and MR-MPI [21], which is the only public implementation of MapReduce using MPI.

In Section 5.2, we used our MRO-MPI model to index 9,319,561 text (XML) excerpts related to 9,319,561 images from 12-million ImageNet corpus [6] and compared the running time with Hadoop. The XML files size is 36 GB.

In Section 5.3, we used our model to index 4,594,734 (84-dimensional) color features related to 4,594,734 images from the 12-million ImageNet corpus [6].

MPICH2 is installed on a Linux cluster of 20 DualCore computers (40 cores in total) holding each 8 GB of memory and 512 GB of local disk storage, led by a master 8-core computer holding 32 GB of memory and a TeraByte storage capacity. Hadoop is also installed on the same machines with HDFS block size of 128 MB.

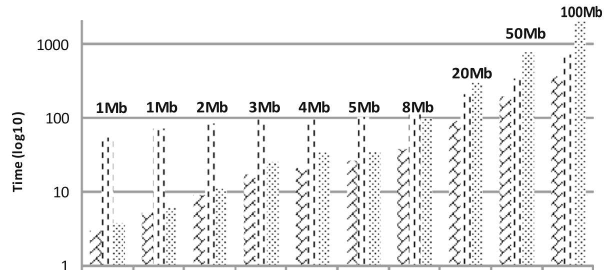
5.1 Word count

WordCount simply counts the occurrence of words in different documents. The map function emits (*word*, 1) pairs, where *word* is the key and 1 represent the value. The input data size varies from 0.2 to 53 GB from project Gutenberg [22]. For this example, we use 48 cores, 24 as mappers and 24 as reducers for our model. The threshold value T is empirically set to 10,000. For Plimpton and Devine implementation [21], the 48 cores are used as mappers then as reducers, as there is no overlapping like ours. A copy of the input data is located on all the nodes of the cluster. In our cluster, each node has two cores. Hence, each two cores have an access to the same copy. So, there is no network communication required to move the data from a certian node to another. The page size was the default value which is 64 MB. For Hadoop, we set the number of reducers to 48 and the number of mappers varies from 200 to 650 according to the number of partial input files. The data replication factor was 10.

Figures 4 and 5 show the $\log_{10}$ of the running time and the speedup for the three implementations respectively. Speed up is defined as:

$$Speedup = \frac{T_{\mathrm{H}}}{T_{\mathrm{MRO}}}, \tag{1}$$

where T_{H} is the Hadoop or MR-MPI execution time and T_{MRO} is our MRO-MPI execution time. As we can see from the figures, our MRO-MPI model and MR-MPI model [21] achieve high speedup compared to Hadoop when the data size is less than 5 GB. The reason for that is the time consumed by Hadoop to start and terminate the tasks. Also, Hadoop is mainly designed for large datasets. For data with size more than 5 GB, Hadoop becomes faster than MR-MPI, but not faster than our model. The reason is that both of Hadoop and MR-MPI follow the same model, but Hadoop has its own file system, which is based on moving the computation power instead of moving the data in contrast to MR-MPI. For Hadoop, this means that, during the computation, if the data is located in machine X and a reduce or a map function is running on machine Y, Hadoop terminates the task on Y and starts it on X instead of moving the data from X to Y. This is done based on some calculations defining the cost of moving the data. For our model, we achieve high speed up comparing to Hadoop and MR-MPI, because of the overlapping with the same number of machines but with less number of mappers and reducers. For example, for 27 GB our model is 1.7 times faster than Hadoop and 4.2 faster than MR-MPI.



Data size(Gb)	0.2	0.5	1	2	2.5	3	5	12	27	53
▨ MRO-MPI	3	5.2	9	17	21	26	37	90	189	369
¦ Hadoop	54	70	85	94	94	100	110	210	339	704
⁘ MR-MPI	3.8	6	11	25	33	35	96	306	805	1980

Fig. 4 WordCount example: the x-axis shows the data size in gigabytes. The y-axis shows the $\log_{10}$ of the running time. The value in the table under the figure shows the running time in seconds. The values above the columns shows the size of each partial file in the data set. As we can see, MRO-MPI outperforms MR-MPI [21] and Hadoop

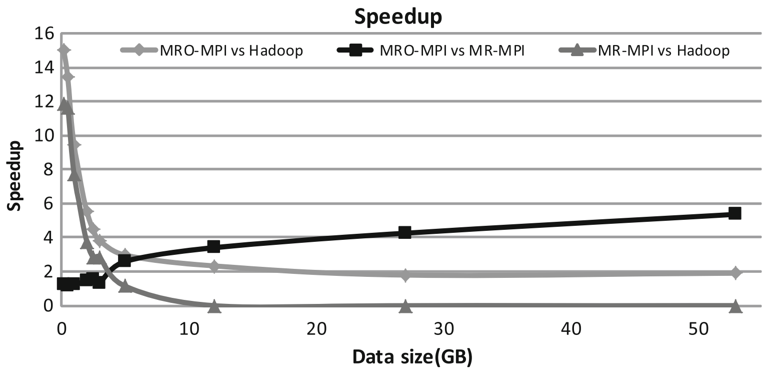


Fig. 5 Figure shows the speedup based on different data sizes. The x-axis shows the data size in GB. The y-axis shows the speedup

Another reason for this speed up is the partial reduction process. The size of the reduced data is smaller than the original data size. In our model, we save the reduced data and avoid the intermediate pairs. This means that we save memory and it is hard to have I/O access. Hadoop and MR-MPI save the pairs in the memory. The size of the pairs is larger than the original data size because each element is identified by a pair. This means that they need to access the hard-disk at some point.

For example, assume that the word "play" appeared three time in the corpus. For our MRO-MPI, the mappers produce three pairs <"*play*", 1> the key is the word and the value is 1. Assume also the size of each pair is about 8 bytes. During examining the rest of the corpus, these pairs are sent to the reducers and the reducer save them as one pair with value equals to 3 <"*play*", 3> so the new size after the partial reduction still the same 8 bytes. For Hadoop and MR-MPI, the three pairs are saved in the memory of the mappers until all the mapping is done. This means that they need extra memory at some point to save the pairs, and they have to access the hard-disk. For the previous example, the mappers need 24 bytes in the memory. Hence, with partial reduction we avoid the hard-disk access as much as possible.

Figure 6 shows the effect of changing the threshold value *T*. The *T* value is the message chunk size in bytes. Normally in MPI, sending small chunks of data between processes increase the running time, because of the communication latency. The same effect happens when we send large chunks through the network, because of the bandwidth. So, for different data sizes we have tested the effect of changing the *T* value. We found that the best time was achieved when the chunk type is *CHAR* array, with size in the range of 9.7 KB to 4 MB. Figure 7 shows the effect of choosing MPI_CHAR against a simple structure that consists of a *key* of type *CHAR* array of size 100 and a *value* of type *INT*. As we can see, the structure degrade the performance of the model for the same data sets with the same configurations.

5.2 Distributed inverted indexing

Our second application is the construction of distributed inverted files. The definition of an inverted file was presented in Section 2.2.1.

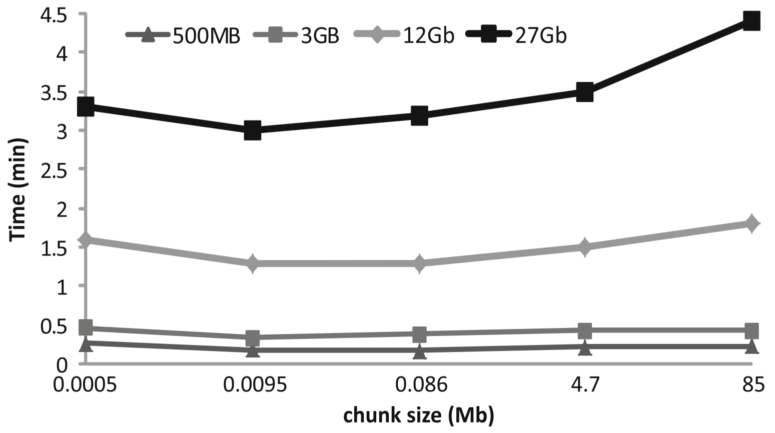


Fig. 6 Figure shows the effect of changing *T*. The x-axis shows the chunk size in megabytes. The y-axis shows the running time in minutes

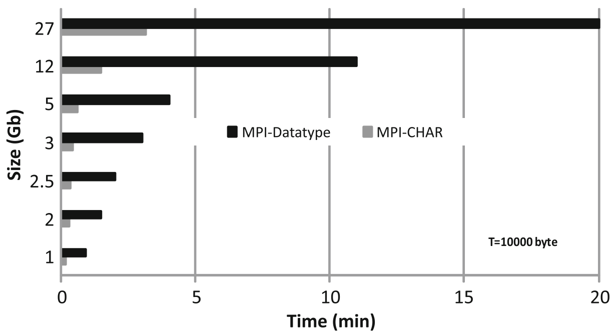
5.2.1 MapReduce for distributed inverted files

In order to preserve acceptable indexing time, the process should be done in parallel. McCreadie et al. [17] presented a good analysis about building an inverted index using MapReduce. The general idea is to build the inverted files on the fly in parallel without the need to access the hard-disk during processing. In our implementation, we apply one of their methods but based on our MRO-MPI model. The mappers are responsible to read and tokenize terms from every document. Mapper nodes emit *(key, value)* pairs. The key is the term extracted from the file and the value is the document name and the corresponding TF value for the term:

$$(K_m, V_m) = (term, (document\ name, tf)).$$

The *(key, value)* pairs are further sent to the reducing nodes. The *partitioning function* distributes the data based on their lexicographic order, each reducer being responsible for a certain range of terms. For example, if we have 26 reducers, reducer 1 is be responsible for all the terms start with character "a", reducer 2 for terms starts with "b", and so on. Each reducing node only receives the terms located within its lexicographic range. As the same terms from all documents are saved into the same

Fig. 7 Figure shows the effect of changing of choosing MPI_CHAR against MPI derived data types. The x-axis shows the running time in minutes and the y-axis shows the data size in gigabytes



database, reducer nodes can calculate the correct value of the IDF and then assign a weight to every term according to the TF-IDF scheme. In this example, the reducers do not work on the partial data as before. Instead, they concatenate this partial data until the mapper is done and then calculate the TF-IDF, which is not time consuming as the data is grouped and sorted.

5.2.2 Experiments

Figure 8 illustrates an example of the XML data used. Table 2 gives the running time in minutes. The first row shows the number of the reducers; 4, 13 and 26. The first column shows the number of mappers; 4, 10, 13, 20, 22, and 44. The value 0 for mappers and reducers corresponds to, the sequential time. The running time using Hadoop was 40 minutes using 93961 mappers, 26 reducers and replication factor 20 on the same cluster. Compared to our results which were obtained using 22 mappers and 26 reducers, our implementation was faster due to the partial copy, which is done during the mapping process.

MapReduce ratio From the reducer side, for 13 and 26 reducers and 4, 10, 13, 20 and 22 mappers, the running time decreases when the number of mappers increases. This is expected as the mappers do most of the work, and more mappers means that the workload is divided, which helps to improve the performance. But, when the number of mappers is high comparing to the number of reducers, the running time increases. The reason is that the reducers are not able to handle all the received data in an efficient way. For example, that happens for 4 reducers with respect to 13, 20, 22, and 44 mappers. The rate of emitting the (key, value) pairs is much higher than the rate of receiving, due to lower number of reducing nodes. From the mapper side, for 10, 13, 20 and 22 mappers with respect to 4, 13 and 26 reducers, the running time decreases when the number of reducers increases. More reducers help to decrease the communication load, which helps to improve the performance. This is not the case for 4 mappers, the running time increases when the number of reducers is larger than the double of mappers. The reason is the large number of reducers relative to the mappers. Mappers need to communicate with a high range of reducers with high load of data, which increases the communication time and affects the total performance.

For 44 mappers and 13 and 26 reducers, there is not such speedup, although the number of mappers and reducers are high. The reason is cluster overloading. Our cluster is composed of 48 cores and the number of processes are 57 and 70, which

```xml
<?xml version="1.0" encoding="utf-8"?>
<image name="n00015388_30042.JPEG" synsetid="n00015388" synsetnum="100015388">
            <url>http://farm4.static.flickr.com/3176/2433586904_180fae9f14.jpg</url>
            <description>animal % animate being % beast % brute % creature % fauna @ a living organism
characterized by voluntary movement</description>
            <keywords>animal % animate being % beast % brute % creature % fauna</keywords>
            <definition>a living organism characterized by voluntary movement</definition>
            <fulldescription>animal % animate being % beast % brute % creature % fauna @ a living organism
characterized by voluntary movement</fulldescription>
            <fullkeywords>animal % animate being % beast % brute % creature % fauna</fullkeywords>
            <fulldefinition>a living organism characterized by voluntary movement</fulldefinition>
</image>
```

Fig. 8 Example of XML data used in indexing

Table 2 The table shows the running time for indexing in minutes with respect to different number of mappers and reducers

Map/Reduce	0	4	13	26
0	81.6	–	–	–
4	–	39.2	46.6	51.7
10	–	41.5	39.8	39.2
13	–	66	28.8	24
20	–	111.9	19.6	18.5
22	–	113.9	16.45	14.2
44	–	118.5	125.2	45.8

For example, the value 14.2 is the running time for 22 mappers and 26 reducers. Sequential time is represented for 0 mappers and reducers

means that the numbers of processes running is much higher than the number of actual cores, this affects the performance of the system.

Based on our experiments, we found that when the number of mappers and reducers increase, the running time systematically decreases, with a non-linear decay, but we should care about the ratio between them and we should not overload the cores with more than one process. The best ratio between the mappers and reducers is found to be:

$$2M \geq R \geq M.$$

Where M is the number of mappers and R is the number of reducers. Figures 9 and 10 show the effect of changing the number of mappers for the reducers and vice versa. Sequential time (one node for all the processing) was obtained on the master node of our cluster, which holds 32 GB memory and 8 processors.

5.3 Distributed approximate similarity search

The third application is the permutation-based distributed indexing using MapReduce for parallel similarity search.

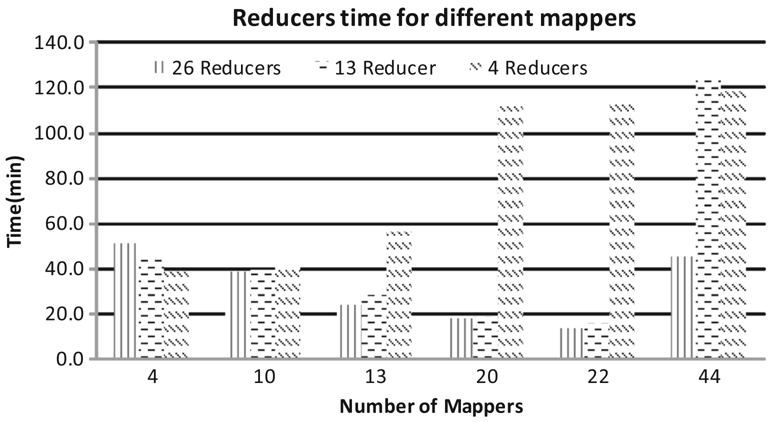


Fig. 9 Figure shows the running time of 4, 13 and 26 reducer for different number of mappers. As we can see, the best timing is achieved when $2M \geq R \geq M$ with no machine overloading

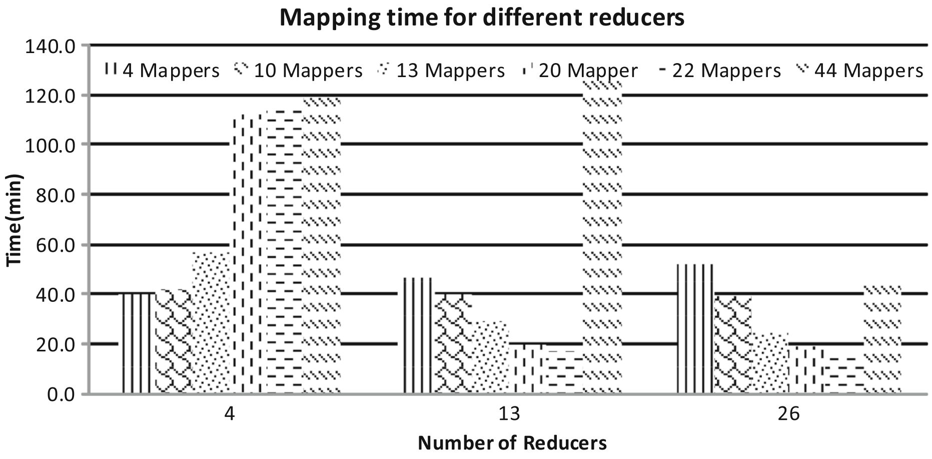


Fig. 10 Figure shows the running time of 4, 10, 13, 20, 22 and 44 mappers for different number of reducers. As we can see, the best timing is achieved when $2M \geq R \geq M$ with no machine overloading

5.3.1 Permutation-based indexes

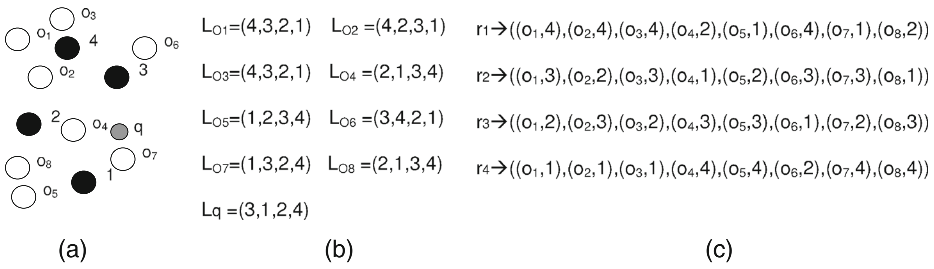
The intuition behind the *permutation-based indexes* is based on "predicting closeness between elements according to how they order their distances towards a distinguished set of anchor objects" [2, 10]. Given a collection of N objects o_i in a domain $D = \{o_1 \dots o_N\}$, and a distance function $d : D \times D \rightarrow \Re$ between the objects. We assume that the distance function $d(.,.)$ follows the metric space postulates [29] $\forall o_i, o_j, o_k \in D$:

- $o_i = o_j \Longleftrightarrow d(o_i, o_j) = 0$ *identity*,
- $d(o_i, o_j) \geq 0$ *non-negativity*,
- $d(o_i, o_j) = d(o_j, o_i)$ *symmetry* and
- $d(o_i, o_k) \leq d(o_i, o_j) + d(o_j, o_k)$ *triangle inequality*.

A set of n reference objects $R = \{r_1, r_2, \dots r_n\} \subset D$ is randomly selected from D. Each object $o_i \in D$ is represented by an ordered list L_{o_i}. The ordered list for each object contains the reference points set sorted by their distance d to the object o_i. More formally, L_{o_i} is the permutation of $(1, \dots, j, \dots, n)$ according to the distance function d. $P(L_{o_i}, r_j)$ returns the position of the reference object r_j within the ordered list L_{o_i} of object o_i. For example, $P(L_{o_i}, r_j) = 5$ means that r_j is the 5th nearest reference point to the object o_i. Figures 11a and b show a group of objects and their ordered list respectively.

The permutation lists for all object are saved in the main memory. For a given query q, an ordered list L_q is computed as for the database objects with respect to the same reference points. The similarity between the query and the database objects is measured by comparing the permutation lists using *Sperman Footrule Distance (SFD)* [29].

$$SFD(o_i, q) = \sum_{r \in R} |P(L_{o_i}, r) - P(L_q, r)| \tag{2}$$



$L_{O1}=(4,3,2,1)$ $L_{O2}=(4,2,3,1)$

$L_{O3}=(4,3,2,1)$ $L_{O4}=(2,1,3,4)$

$L_{O5}=(1,2,3,4)$ $L_{O6}=(3,4,2,1)$

$L_{O7}=(1,3,2,4)$ $L_{O8}=(2,1,3,4)$

$L_q=(3,1,2,4)$

$r_1 \rightarrow ((o_1,4),(o_2,4),(o_3,4),(o_4,2),(o_5,1),(o_6,4),(o_7,1),(o_8,2))$

$r_2 \rightarrow ((o_1,3),(o_2,2),(o_3,3),(o_4,1),(o_5,2),(o_6,3),(o_7,3),(o_8,1))$

$r_3 \rightarrow ((o_1,2),(o_2,3),(o_3,2),(o_4,3),(o_5,3),(o_6,1),(o_7,2),(o_8,3))$

$r_4 \rightarrow ((o_1,1),(o_2,1),(o_3,1),(o_4,4),(o_5,4),(o_6,2),(o_7,4),(o_8,4))$

Fig. 11 Example of Metric inverted files. **a** *Black circles* are reference objects; *white circles* are data objects; the *gray circle* is query object. **b** Ordered lists for all data objects o_i. **c** Inverted index; the vocabulary are the reference points and the posting lists are pairs of data objects and their positions

5.3.2 Metric inverted files

Amato and Savino [2] presented the metric inverted files (MIF) for indexing permutation-based indexes. Using the definition of the inverted files in Section 2.2.1, the dictionary in MIF is the set of reference points and the posting list for a reference point r_j contains a list of pairs $(o_i, P(L_{o_i}, r_j)) \forall o_i \in D$. Figure 11c shows an example of MIF. For Searching [2] a given query q, an accumulator is assigned to each object $o_i \in D$ and initialized to zero. The posting list for each reference point is accessed and the accumulator is updated by adding the difference between the position of the current reference object in the ordered list of the query and the position of current object, using (2). After checking the posting lists of all the reference points, the objects are sorted based on their accumulator value.

In [2], authors have improved the performance of the algorithm by indexing the objects with respect to some nearest reference objects only and perform the search using these nearest reference objects. They experimentally proved that the nearest reference objects are the most relevant ones. The complexity of this basic algorithm is $O(nN)$, where n is the number of reference objects and N is the number of objects. Algorithm 1 [2] explains the main idea for searching for a given query q.

Algorithm 1 Basic MIF searching algorithm

```
IN: Query: $q$,
   Reference Object list on $n$ elements: $R$,
   Posting lists assigned to each reference object for $m$ objects;
OUT: Sorted Objects list: $out$
1.   Create a list of accumulators $A[0 \ldots n]$
2.   Set accumulators values to 0
3.   For each $r \in R$
4.      Let $\Delta$ be the posting list for the reference object $r$
5.      Set $i \longleftarrow 0$
6.      For each pair $(o, P(L_o, r)) \in \Delta$
7.         Set $A[i] = A[i] + |P(L_o, r) - P(L_q, r)|$
8.         $i \longleftarrow i + 1$
9.   Sort(A)
10.  out $\leftarrow A$
```

5.3.3 Distributed metric inverted files indexing

In this work, we are more focusing on the indexing part. The direct way of building distributed inverted files is the posting list decomposition (PLD). The *PLD* was proposed in [25] for text indexing. We use the same technique, but for approximate similarity metric searching through three ways of distributed indexing based on MPI and MRO-MPI. The main idea of PLD is to divide the posting lists on number of processes and for searching each process accesses its own partial posting lists only. Then combine the results and send it back to the user. Figure 12 shows the posting list decomposition structure. For example, the posting list for reference point 1 in Fig. 11c is divided into four lists. Each list is processed by a different process. Using MapReduce, we can build these distributed inverted files in parallel. We propose two algorithms for indexing based on MapReduce, *pre-posting list* and *pre-ordered list*. An alternative is to implement directly using MPI our *Local indexing* algorithm. For the three algorithms, we assume that the reference points are available to all processes.

Pre-posting list The vector file is divided into small chunks. These vectors represent the objects. Each map function handles a chunk. The map functions read the vectors and emit a sequence of *(key, value)* pairs. The key is the reference-id I_{r_i}. The value is composed of the object-id I_{o_i} and the position of the reference point in the ordered list of this object $P(L_{o_i}, r_i)$:

$$(K_m, V_m) = (I_{r_i}, (I_{o_i}, P(L_{o_i}, r_i))).$$

Hence, all the work is done by the mappers and the reducers only receive the pairs to organize and save them. The partitioning function distribute the pair based on the object-id I_{o_i}. For example, if we have 1,000 objects and 5 reducers, then each reducer handles 200 objects. Reducer 0 handles objects with id from 0 to 199, reducer 1 handles objects with id from 199 to 399 and so on. Algorithms 2 and 3 show the pseudo-code of the mapping and reducing functions respectively.

Pre-ordered list In *pre-posting list* algorithm, all the work is done by the mappers and the reducers just organize the data. Hence, if we divided the work between the mappers and reducers we can achieve high performance. Similar to *pre-posting list*, the vectors file is divided into small chunks. Each map function handles a chunk. The map functions read the vectors and emit a sequence of *(key,value)* pairs. The key of

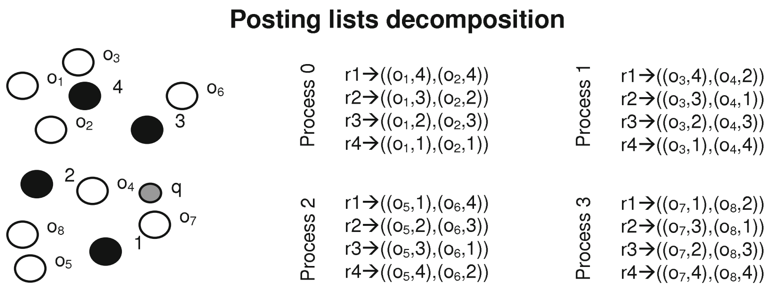


Fig. 12 Posting lists decomposition algorithm. The posting lists are divided on four processes

Algorithm 2 Pre-posting list mapping

```
IN:   Key: chunk name
      Value: vectors
OUT:  Key: Object ID $I_{r_i}$
      Value: Ordered list $(I_{o_i}, P(L_{o_i}, r_i))$
1.  Read the objects in $Objarr$ and the reference points in $Refarr$
2.  For each $o \in Objarr$
3.    Generate ordered list $L_{oi}$
4.    Sort the ordered list $L_{oi}$
5.    For each $r \in Refarr$
6.      Get the position of $r$ in $L_{o_i}$; $P(L_{o_i}, r_i)$
5.      emit($I_{r_i}$,($I_{o_i}$, $P(L_{o_i}, r_i)$))
```

the map function is the object-id I_{oi} and the value is the ordered list L_{o_i} related to this object:

$$(K_m, V_m) = (I_{o_i}, L_{o_i}).$$

Algorithm 4 shows the pseudo-code of the mapping function. The partitioning function is similar to the one used in the *pre-posting list* algorithm. When the reducers receive the data, they build their own posting lists by calculating the $P(L_{o_i}, r_i)$ for each reference in the ordered list of the received objects. Algorithm 5 shows the pseudo-code of the reducing function.

Local indexing Here, we use the basic MPI functions to index the objects without MapReduce. We divide the data domain D of N objects randomly into p sub-domains of equal sizes $D_0 \ldots D_p$, where p is the number of parallel processes. Each process then starts to build its own inverted file data structure based on the global reference points and the partial data it has access to. Accordingly, each process is responsible for all the references with partial posting list. Hence, there is no need to transfer data between the processes. Each process has its part and builds it independently.

Searching For the three algorithms, the inverted file is partitioned and distributed. Therefore, to answer a query q, all partial inverted files need to be scanned. A broker

Algorithm 3 Pre-posting list reducing

```
IN:   Key: Refrence Object id $I_{r_i}$
      Value: List of $< I_{o_i}, P(L_{o_i}, r_i)) >$
OUT:  Key: Object ID $I_{o_i}$
      Value: Position in the ordered list $P(L_{o_i})$
1.  For each $p \in < I_{o_i}, P(L_{o_i}, r_i)) >$
2.    Save it in the posting list of reference object $I_{r_i}$
3.  emit(posting list)
```

Algorithm 4 Pre-ordered lists mapping

```
IN:   Key: chunk name
      Value: vectors
OUT:  Key: object id $I_{o_i}$
      Value: ordered list $L_{o_i}$
1.  Read the objects in $Objarr$ and the reference points in $Refarr$
2.  For each $o \in Objarr$
3.      Generate ordered list $L_{oi}$
4.      Sort the ordered list $L_{oi}$
5.      emit($I_{o_i}$,$L_{o_i}$)
```

process accepts query requests. These queries are then broadcasted to all other processes. After receiving, each process starts to index the query and to apply the search on its local inverted file. Once done, every process sends its local accumulators to the broker process. The broker concatenates the accumulators and sorts the objects based on their accumulator values. More formally, in Algorithm 1, the **for** loop in lines 6–8 can run in parallel over the different partial domains $D_0, D_1, \ldots D_p$. Thus, theoretically the memory usage is reduced from $O(nN)$ to $O(n\frac{N}{p})$ and the complexity leads to $O(n\frac{N}{p}) + t_s$, where $n = \sum_{i=0}^{n} n_i$ and t_s is the time needed to receive the partial ranked objects. In this algorithm, all the reference points need to be checked and all the processes which have a partial posting lists have to participate to answer a query.

5.3.4 Experiments

We have compared the performance of the three algorithms, which we have already discussed. Our dataset consists of 4,594,734 (84-dimensions) objects. For the *pre-posting list* and the *pre-ordering list*, we performed two experiments. The first one uses 10 cores as mappers and 10 cores as reducers. The second experiment uses 20 cores as mappers and 20 cores as reducers. The threshold value T is set to 1.8 MB. For *local indexing*, we have indexed the data using 10 cores and then using 20 cores.

Algorithm 5 Pre-ordered lists reducing

```
IN:   Key: object id $I_{o_i}$
      Value: List of ordered lists $L_{o_i}$
OUT:  Key: Object ID $I_{o_i}$
      Value: Position of reference $r_i$ in the ordered list $P(L_{o_i})$
1.  For each $r \in L_{o_i}$
2.      Calculate $P(L_{o_i}, r_i)$
3.      Add ($I_{o_i}$, $P(L_{o_i}, r_i)$) to the posting list of $r$
4.  emit(posting list)
```

We have compared the performance of the three algorithms for different number of cores.

Indexing time Due to memory limitations, the sequential algorithm can not handle this data. We have 4,594,734 objects and each pair in the posting list needs about 8 bytes. For 1,000 reference objects we need about 34 GB of memory, which cannot be supported by any of our machines.

Figure 13 shows the indexing time for *pre-posting list*, *pre-ordering list* and *local indexing*. For the three algorithms, when the number of cores increases, the indexing time decreases. Also, when the number of reference objects increases the running time increases. As we can see from the figure the *pre-ordered list* algorithm is faster than *pre-posting list* algorithm. There are two reasons for that. The first reason is the rate of emitting the data. In *pre-posting list*, at each emitting the map function emits the reference-id, the object-id and the position of the reference point in the ordered list of the object. So, if we have 1,000 object and 10 reference points the mapping function emits 10,000 pairs. On the other hand, for the *pre-ordered list* algorithm the mapping function emits the object-id and the ordered list. So , if we have 1,000 object and 10 reference points the mapping function emits 1,000 pairs only. Figure 14 shows the average output of each mapping function in gigabytes for the two algorithms. As we can see, the average output of each mapping function for *pre-ordering list* is less than the average output of the *pre-posting list* algorithm. Also, for the two algorithms, when the number of nodes increase the average output decreases.

The second reason is the way of organizing the work between mappers and reducers. In *pre-posting list*, all the work is done by the mappers and the reducers only save the received pairs. On the other hand for the *pre-ordered list* algorithm the work is divided between them and that decreases the running time. Hence, the way of choosing the (*key*, *value*) pairs beside the way of dividing the work between the mappers and reducers are affecting the performance.

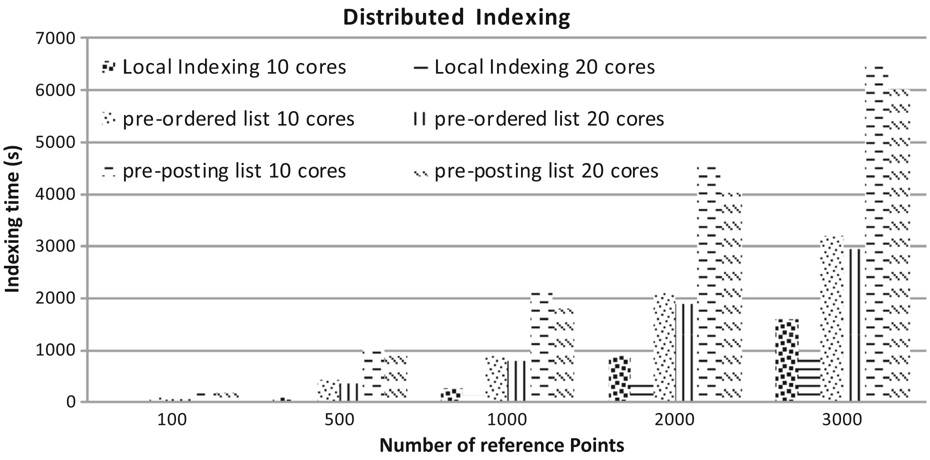
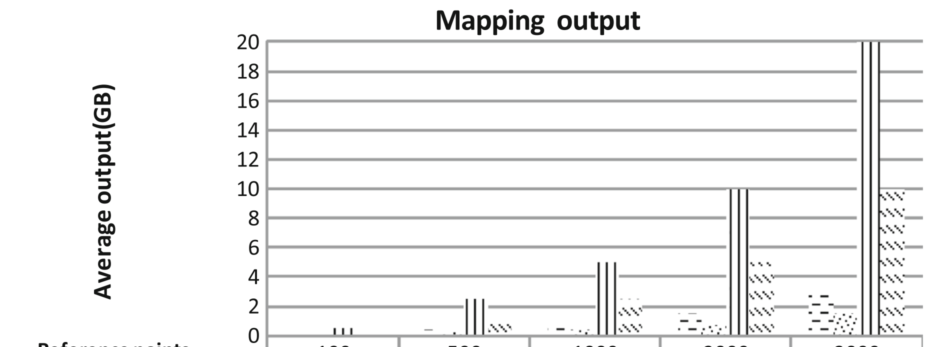


Fig. 13 Figure shows the indexing time for the three algorithms with respect to different number of cores. The x-axis shows the number of reference points used for indexing and the y-axis shows the running time in seconds



Reference points	100	500	1000	2000	3000
Pre-Ordered list 10 cores	0.086	0.4	0.8	1.5	2.81
Pre-Ordered list 20 cores	0.043	0.2	0.4	0.73	1.5
Pre-Posting list 10 cores	0.52	2.5	5	10	20
Pre-Posting list 20 cores	0.25	1.3	2.5	5	10

Fig. 14 Average output of each mapping function. The x-axis shows the number of reference points and the y-axis shows the data size in gigabytes

The *local indexing* algorithm is much faster than the two algorithms. The reason is that there is no data exchange between the cores, which improves the performance. At the same time, MPI misses the simplicity. More time and experience are required for coding the algorithm using MPI than using MRO-MPI. So, MapReduce is useful and makes the parallel programming an easy process, but is not suitable for all applications, as we can get better performance using normal MPI.

Searching time Figure 15 shows the average searching time for algorithms *PLD* based on 10 different queries from the datasets. The x-axis shows the number of

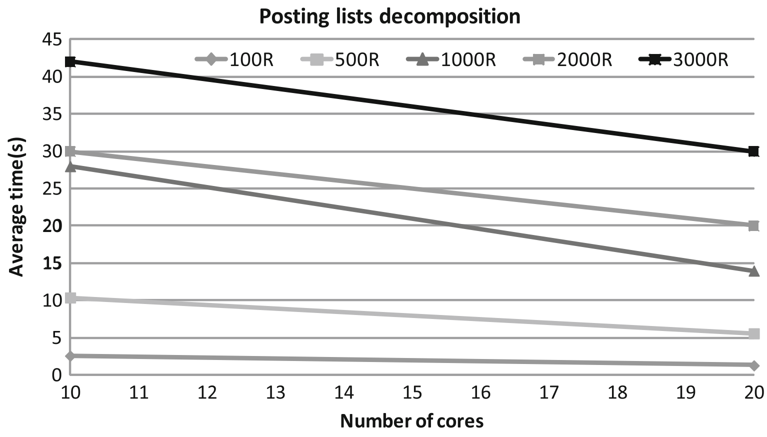


Fig. 15 Figure shows the average search time for *posting list decomposition* relative to 100, 500, 1,000, 2,000 and 3,000 reference points (R)

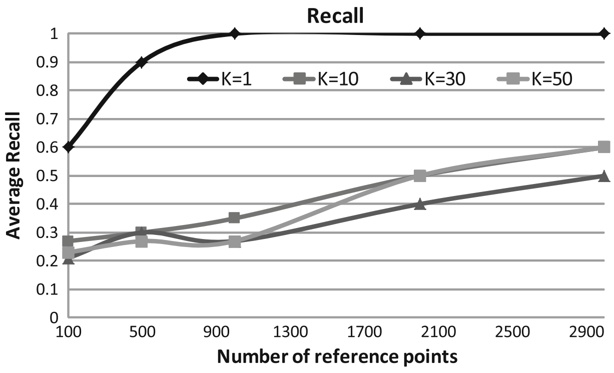


Fig. 16 Recall: The x-axis shows the number of reference points and the y-axis shows the average recall relative to $K =$ 1, 10, 30, 50 points

cores and the y-axis shows the running time in seconds. Similar to indexing when the number of cores increases the average response time decreases.

Recall and position error Here, we measure both the recall and the position error [29] for each algorithm. Given a query q the recall is defined as:

$$Recall = \frac{|S \bigcap S_A|}{|S|} \tag{3}$$

and the position error is defined as:

$$Position\ Error = \frac{\sum_{o \in S_A} |P(X, o) - P(S_A, o)|}{|S_A|.|D|} \tag{4}$$

where S and S_A are the ordering of K top ranked objects to q for exact similarity search and approximate similarity search respectively. X is the ordering of dataset D with respect to their distance from q and P is defined in Section 5.3.1.

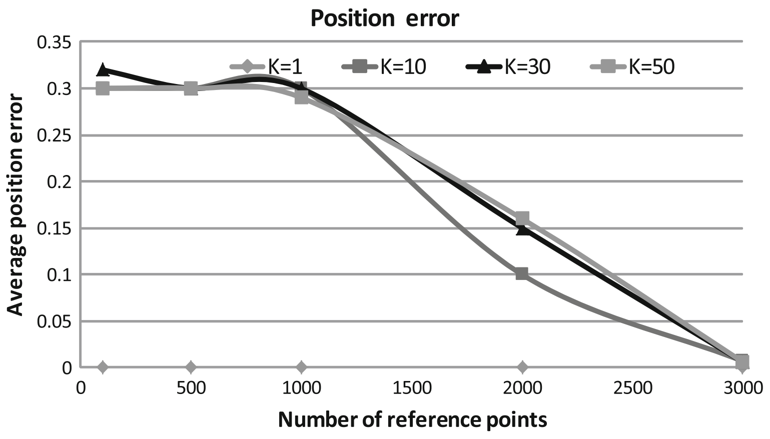


Fig. 17 Position error: The x-axis shows the number of reference points and the y-axis shows the average position error. Recall and position error measured relative to $K = 1, 10, 30, 50$ points

Figures 16 and 17 show the average recall and the average position error relative to 1, 10, 30, 50 K-top points based on 10 different queries from the datasets. The average recall and the average position error is similar to those obtained using the sequential implementation [2] with better computing performance as it is the same structure but it is accessed in parallel.

6 Conclusion and future work

This paper proposes a new way of handling the MapReduce programming model using MPI for fast large scale data processing. We have proposed the idea of the overlap between the map and reduce functions using MPI "MRO-MPI", which speeds up the process. The main advantages of our model are:

1. Maintain the simplicity of MapReduce.
2. Speed up: with the same number of nodes and less number of mappers and reducers we achieved high speedup comparing to other implementation of MapReduce.
3. No dependency: we removed the dependency between the two functions. The reducers do not have to wait until the mappers are done.

To evaluate our model, we have tested it on three different applications. The first one is the *WordCount* example. Using our model we achived a high speedup comparing to Hadoop and the earlier implementation of MapReduce-MPI. For example, for 53 GB our model is 2.8 times faster than Hadoop and 5.3 faster than MR-MPI. Also, we have discussed the threshold value T and the derivative data types against normal MPI data types and how these can affect the running time of our model.

In the second application, we have indexed text data using our model against Hadoop. From the results, our model is faster than Hadoop. Also, we have discussed the ratio of mappers to reducers and how this can affect the running time.

In the third application, we have indexed high dimensional multimedia data using our model against normal MPI. Results show that the way of arranging the mappers and reducers work and the rate of emitting can affect the running time. Also, we have shown that we can get better performance using MPI without MapReduce, but in trade of simplicity and coding time.

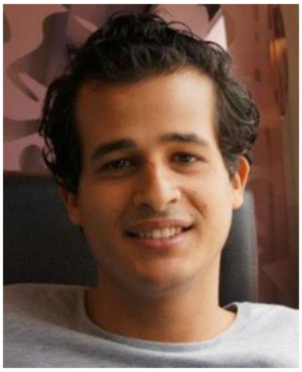
For future work, we will release a library for MapReduce overlapping based on MPI. We also are planning to make a deeper analysis on the communication time and use contiguous and noncontiguous MPI data types to see how this can affect the system performance.

Acknowledgements This work is jointly supported by the Swiss National Science Foundation (SNSF) via the Swiss National Center of Competence in Research (NCCR) on Interactive Multimodal Information Management (IM2) and the European COST Action on Multilingual and Multifaceted Interactive Information Access (MUMIA) via the Swiss State Secretariat for Education and Research (SER).

References

1. Ahmad F, Lee S, Thottethodi M, Vijaykumar TN (2007) Mapreduce with communication overlap. Technical report
2. Amato G, Savino P (2008) Approximate similarity search in metric spaces using inverted files. In: Proceedings of the 3rd international conference on scalable information systems, InfoScale '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ICST, Brussels, Belgium, pp 28:1–28:10. http://dl.acm.org/citation.cfm?id=1459693.1459731
3. Bruno E, Marchand-Maillet S (2009) Multimodal preference aggregation for multimedia information retrieval. J Multimedia 4(5):321–329
4. Csurka G, Dance CR, Fan L, Willamowski J, Bray C (2004) Visual categorization with bags of keypoints. In: Workshop on statistical learning in computer vision, ECCV, pp 1–22
5. Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on opearting systems design & implementation, vol 6. USENIX Association, Berkeley, p 10
6. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009) ImageNet: a large-scale hierarchical image database. In: CVPR '09
7. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G (2011) Twister: a runtime for iterative mapreduce. In: Proceedings of the 19th ACM international symposium on high performance distributed computing, HPDC '10. ACM, pp 810–818. doi:10.1145/1851476.1851593
8. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI users' group meeting, pp 97–104
9. Gillick D, Faria A, Denero J (2006) Mapreduce: distributed computing for machine learning
10. Gonzalez E, Figueroa K, Navarro G (2008) Effective proximity retrieval by ordering permutations. IEEE Trans Pattern Anal Mach Intell 30(9):1647–1658. doi:10.1109/TPAMI.2007.70815
11. Gropp W, Lusk E, Skjellum A (1994) Using MPI: portable parallel programming with the message-passing interface. MIT Press, Cambridge
12. Hoefler T, Lumsdaine A, Dongarra J (2009) Towards efficient mapreduce using mpi. In: Ropo M, Westerholm J, Dongarra J (eds) PVM/MPI, Lecture notes in computer science, vol 5759. Springer, pp 240–249
13. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the 30th annual ACM symposium on theory of computing, STOC '98. ACM, New York, pp 604–613. doi:10.1145/276698.276876
14. Jagadish HV, Mendelzon AO, Milo T (1995) Similarity-based queries. In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, PODS '95. ACM, New York, pp 36–45. doiL:10.1145/212433.212444
15. Kumar V (2002) Introduction to parallel computing, 2nd edn. Addison-Wesley Longman, Boston
16. Lu X, Wang B, Zha L, Xu Z (2011) Can mpi benefit hadoop and mapreduce applications? In: 40th international conference on parallel processing workshops (ICPPW), pp 371–379. doi:10.1109/ICPPW.2011.56
17. McCreadie R, Macdonald C, Ounis I (2011) Mapreduce indexing strategies: studying scalability and efficiency. Inf Process Manag. doi:10.1016/j.ipm.2010.12.003
18. Message passing interface. http://www.mpi-forum.org/
19. Mpich2. http://www.mcs.anl.gov/mpi/mpich2
20. Patella M, Ciaccia P (2009) Approximate similarity search: a multi-faceted problem. J Discrete Algorithms 7(1):36–48. doi:10.1016/j.jda.2008.09.014
21. Plimpton SJ, Devine KD (2011) Mapreduce in mpi for large-scale graph algorithms. Parallel Comput 37(9):610–632
22. Project gutenberg. http://www.gutenberg.org/
23. Rajasekaran R, Reif J (2007) Handbook of parallel computing: models, algorithms and applications. CRC Press
24. Samet H (2006) Foundations of multidimensional and metric data structures. In: The Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier/Morgan Kaufmann. http://books.google.ch/books?id=vO-NRRKHG84C
25. Stanfill C (1990) Partitioned posting files: a parallel inverted file structure for information retrieval. In: Proceedings of the 13th annual international ACM SIGIR conference on research and development in information retrieval, SIGIR '90. ACM, New York, pp 413–428. doi:10.1145/96749.98247

26. White T (2009) Hadoop: the definitive guide, 1st edn. O'Reilly
27. Witten IH, Moffat A, Bell TC (1999) Managing gigabytes: compressing and indexing documents and images, 2nd edn. Morgan Kaufmann, San Francisco
28. von Wyl M, Mohamed H, Bruno E, Marchand-Maillet S (2011) A parallel cross-modal search engine over large-scale multimedia collections with interactive relevance feedback. In: Proceedings of the 1st ACM international conference on multimedia retrieval, pp 73:1–73:2
29. Zezula P, Amato G, Dohnal V, Batko M (2006) Similarity search: the metric space approach, advances in database systems, vol 32. Springer



Hisham Mohamed received his B.S. in systems and biomedical engineering from Cairo University, Egypt in 2008 and his M.S. degree in software engineering from Nile University, Egypt in 2010. Since 2010, he is a PhD student and working at the Computer Vision and Multimedia Laboratory, University of Geneva, Switzerland, as a research assistant. His research interests focus on large scale multimedia information retrieval.

Stéphane Marchand-Maillet is associate professor in the Department of Computer Science at University of Geneva. He holds a PhD in applied mathematics from Imperial College UK ('97). He is the founding leader of the Viper Research Group on Information Retrieval and Machine Learning. He has authored a number of journal and conference contributions on Information Retrieval, Machine Learning, Data Mining and Multimedia processing. He was general chair of the ACM SIGIR 2010 in Geneva, CH and ACM CIVR 2009 in Santorini, GR. He is involved in a number national and international projects, in relation to Information Access. He participates in scientific committees, including as chair of the Technical Committee 12 of the International Association for Pattern Recognition ("Multimedia and Visual Information Systems").