J Supercomput (2009) 48: 286–318
DOI 10.1007/s11227-008-0225-x

Efficient parallel Text Retrieval techniques on Bulk Synchronous Parallel (BSP)/Coarse Grained Multicomputers (CGM)

Charalampos Konstantopoulos · Basilis Mamalis · Grammati Pantziou · Damianos Gavalas

Published online: 30 July 2008

Abstract In this paper, we present efficient, scalable, and portable parallel algorithms for the off-line clustering, the on-line retrieval and the update phases of the Text Retrieval (TR) problem based on the vector space model and using clustering to organize and handle a dynamic document collection. The algorithms are running on the *Coarse-Grained Multicomputer (CGM)* and/or the *Bulk Synchronous Parallel (BSP)* model which are two models that capture within a few parameters the characteristics of the parallel machine. To the best of our knowledge, our parallel retrieval algorithms are the first ones analyzed under these specific parallel models. For all the phases of the proposed algorithms, we analytically determine the relevant communication and computation cost thereby formally proving the efficiency of the proposed solutions. In addition, we prove that our technique for the on-line retrieval phase performs very well in comparison to other possible alternatives in the typical case of a multiuser information retrieval (IR) system where a number of user queries are concurrently submitted to an IR system. Finally, we discuss external memory issues and

C. Konstantopoulos (✉)
Department of Informatics, University of Piraeus and Research Academic Computer Technology Institute Patras, Piraeus, Greece
e-mail: konstant@cti.gr

B. Mamalis · G. Pantziou
Department of Informatics, Technological Educational Institution of Athens, Athens, Greece

B. Mamalis
e-mail: vmamalis@teiath.gr

G. Pantziou
e-mail: pantziou@teiath.gr

D. Gavalas
Department of Cultural Technology and Communication, University of the Aegean, Mytilene, Greece
e-mail: dgavalas@aegean.gr

show how our techniques can be adapted to the case when processors have limited main memory but sufficient disk capacity for holding their local data.

Keywords BSP model · CGM model · Parallel algorithms · Text retrieval · Document clustering · External memory

1 Introduction

The continuous growth of electronically available information, which is dispersed over a local or a wide area network or usually, over different internet locations, together with the increasing requirements for immediate processing and access to the information, make the use of parallel processing indispensable to the information retrieval field. Until today, many attempts have been made to apply parallel processing and build efficient and practical parallel text retrieval systems [8, 9, 19, 24, 29, 37–39, 44, 47], which achieve significant response-time improvements compared with serial text retrieval systems.

To the best of our knowledge, the attempts to adopt parallel processing techniques to large-scale text retrieval have been made either on the real environment of a supercomputer (e.g., connection machine [47], GCel3/512 Parsytec machine [19], APseries multicomputers [24], high performance transputer networks [9]) or of a PC cluster (e.g. [8, 29, 38, 44]), or by employing specific interconnection networks for connecting the processors of the parallel machine (e.g., fat-trees, hypercubes [39]). Many of the existing parallel text retrieval techniques designed on specific platforms provide very efficient code for these platforms but at the cost of portability, i.e., the code cannot be used on different platforms without rewriting it, as well as at the cost of feasibility, i.e., the techniques are implemented with a significant amount of effort. In addition, different text retrieval techniques cannot be compared with respect to a parallel model to determine which one is the best for a given criterion (e.g., the execution time).

In this work, we concentrate on the design and analysis of parallel text retrieval algorithms under specific parallel models, which capture within a few parameters the characteristics of the machine on which the algorithm is supposed to run, as well as the rules of execution of the program resulting from an implementation. Specifically, we present the first efficient, scalable and portable parallel algorithms written in the *Coarse-Grained Multicomputer (CGM)* and the *Bulk Synchronous Parallel (BSP)* models for the off-line clustering phase, the on-line retrieval phase and the update phase of the text retrieval problem based on the vector space retrieval model and using clustering to organize and handle a dynamic document collection.

We prove that the off-line clustering phase requires $O(p)$ computation and communication rounds/supersteps, where p is the number of processors of the parallel computing system. We also show that the local memory required in each processor for this phase is $O(\max\{\frac{R}{p}d_{\text{sim}}, \frac{n}{p}r_{\max}\})$ at most, where n is the number of documents, R is the total data volume of document collection while $r_{\max}$ and d_{sim} are two parameters, that depend on the maximum number of terms in each document and the "similarity degree" of the n documents, respectively. In addition, we show that the

total computation and communication cost is $O(\frac{nR}{p})$ and $O(g \max\{R, \frac{R}{p}d_{\text{sim}}\} + pL)$ respectively. Note that the computation cost is dominated by the cost of the sparse matrix multiplication required for finding document similarities. Also, the above amount of local computation and local memory per processor match in aggregate those ones of the corresponding serial information-retrieval algorithm, and thus can be considered optimal.

We also prove that for the on-line retrieval phase, $O(1)$ communication rounds are necessary with $O(g\frac{n}{p} + L)$ communication cost and $O(\max\{|C|, \frac{n}{p}\}r_q)$ local computation per round where r_q is the number of distinct terms of the submitted query. The local memory per processor required for this phase is $O(\frac{n}{p}r_{\max})$ at most. We also formally show that the document distribution resulting from the off-line clustering phase guarantees a very good system performance during the on-line retrieval phase specially, when a large number of queries are concurrently submitted to the IR system.

The update phase (document insertion/deletion) requires $O(gp + L)$ communication time at most. The local computation cost is $O(|C|r_{\max})$ for document insertion and $O(\max\{|C|, \log \frac{n}{p}\}r_{\max})$ for document deletion. The local memory per processor required for the update phase is $O(\frac{n}{p}r_{\max})$ at most.

It is important to observe that the number of communication rounds/supersteps for the off-line preprocessing/clustering phase is $O(p)$ while for the on-line retrieval and the update phases is $O(1)$, i.e., in all cases it is independent of the problem size. When the size of the problem grows, then the size of the messages grow but the total number of messages remains the same and, therefore, the overhead associated with the message transmissions is kept low.

Finally, we address the issue of external memory implementation of our techniques. Specifically, by making use of the general simulation techniques proposed in [12, 13, 15, 16], we present how our BSP/CGM algorithms can be adapted to the External Memory (EM) BSP model which is an enhancement of the classic BSP model and apart from communication and computation cost, it also captures the cost of data transfer between main and external memory.

A preliminary version of the paper can be found in [20]. That paper does not include the discussion about the external memory issues and the analysis of the off-line clustering and the on-line query phase is considerably shorter and less general than that described in this paper.

2 The BSP and CGM computing models

The *BSP* model was introduced by Valiant [48, 49] as a bridging model that tries to close the gap between the domains of decentralized (i.e., parallel or distributed [28]) architectures and parallel algorithms. Apart from the cost of the parallelism that is accounted by the traditional *PRAM* cost model, *BSP* also considers the communication and synchronization issues imposed by the realistic decentralized architectures. At the same time, *BSP* abstracts away from the algorithm designers, detailed architectural features, such as the topology of the interconnection network or the synchronization procedures and, therefore, it allows the design of parallel algorithms that

can be efficiently executed on a variety of decentralized architectures at a predictable cost expressed w.r.t. some architectural parameters that reflect the capabilities of the underlying platform.

A *BSP* computer is a collection of p processor/memory modules connected by a communication infrastructure that can deliver messages in a point to point fashion between the processors. A *BSP* computation is divided into a sequence of supersteps separated by barrier synchronizations. Each superstep is divided into a computation and a communication superstep. In a computation superstep, the processors perform computations on data that were present locally at the beginning of the superstep. In a communication superstep, data is exchanged among the processors via the communication network. The parameters of a *BSP* computer used to solve a problem are the problem size N, the number of processors p, the bandwidth parameter g, i.e., the ratio of overall system computational capacity (number of computation operation) per unit time divided by the overall system communication capacity (number of messages of unit size that can be delivered by the underlying communication network) per unit time, and the latency parameter L, which is the minimum time between two consecutive synchronization operations.

The cost of a *BSP* algorithm equals to the sum of the costs of its supersteps. The cost of a superstep is given by the following formula

$$T_{\text{superstep}} = \max\{L, T_{\text{comp}} + T_{\text{comm}}\},$$

where T_{comp} is the maximum (over all processors) cost for local computations and T_{comm} is the maximum time needed for transmitting all the outgoing messages to their target processors. If we consider that during the communication superstep each processor sends and receives at most h 1-word messages (i.e., an h-relation has to be implemented during the communication superstep) then $T_{\text{comm}} = g \cdot h$ [42]. In that case, we have

$$T_{\text{superstep}} = \max\{L, T_{\text{comp}} + g \cdot h\}.$$

The *Coarse-Grained Multicomputer (CGM)* [10, 11, 14, 17] model is a simplified version of the *BSP* model and its main purpose is the design of simple and practical yet theoretically optimal or efficient parallel algorithms for coarse-grained parallel systems. A *CGM(N,p)* computer consists of p processors with N/p local memory ($N/p \gg 1$) each connected by a router that can deliver messages in a point to point fashion. A *CGM* algorithm consists of an alternating sequence of *computation rounds* and *communication rounds*, which are separated by barrier synchronizations. A computation round is equivalent to a computation superstep in the *BSP* model [21, 22, 48, 49] and, therefore, the total computation cost is defined analogously. A communication round consists of a single h-relation with $h = O(N/p)$, i.e., each processor sends $O(N/p)$ data and receives $O(N/p)$ data. The main advantage of the *CGM* model is that it allows us to model the communication cost of a parallel algorithm by a single parameter, i.e., the number of communication rounds [17]. A practical assumption commonly met in most proposed CGM algorithms (e.g., see [17]) is that $N/p \geq p^{\epsilon}$ where ϵ is a constant greater than 1. The most important implication of this assumption is that the basic operation of sorting can be executed in a constant number of computation and communication rounds [6, 23].

Note that every *CGM* algorithm is also a *BSP* algorithm but not vice versa. Note also that a CGM algorithm with λ rounds and computation cost T_{comp} corresponds to a BSP algorithm with λ supersteps, communication cost $O(\lambda(g\frac{N}{p} + L))$ and the same computation cost.

3 The VSM-based text retrieval problem

The *Vector Space Model (VSM)* has been used as the basis for many ranking Information Retrieval (IR) systems [2]. The model assumes that we are given a document collection of size n and an array $(t_1, t_2, \ldots, t_m)$ of m different terms, which are word stems extracted out of all n documents. According to the model, each document D_i in the document collection is indexed by a so-called *document-term vector* d_i, after the application of suitable *stopping* and *stemming* procedures. Specifically, each document D_i is represented by the vector

$$d_i = (w_{i1}, w_{i2}, \ldots, w_{im}),$$

where w_{ij} is a weight value ranging from 0 to 1 (assuming normalization) and representing the significance of term t_j for the specific document D_i [45, 46]. Each weight value w_{ij} assigned to each document D_i is related to the frequency of occurrence of term t_j in both the specific document D_i and to the whole document collection. A weight value w_{ij} that is equal to 0 usually means that the term t_j is not appearing in document D_i. In a similar manner, the documents' term vectors may also be expanded by additional document identifiers (not only word stems) such as phrases, thesaurus classes, etc.

The model assumes that each user query Q (which actually is a set of words) is indexed in the same way, thus resulting to a *query-term vector* q of almost the same type as d_i. Specifically, each query Q is represented by the vector

$$q = (q_1, q_2, \ldots, q_m),$$

where q_j is usually set either to 1 or to 0 depending on the presence of term t_j in the query or not. However, a query-term vector may consist of values ranging from 0 to 1 (assuming normalization), in the case that the user assigns by himself weights on the query terms, depending on their significance for the search.

According to the vector space retrieval model, the query vector q is compared (applying a suitable similarity measure) to each one of the n document term vectors, yielding to a corresponding relevance value (score) $sim(q, D_j)$, for each document D_j, $j = 1, \ldots, n$. The documents are then ranked according to their scores and the top-scored ones are supposed to be the most relevant to the query.

For speeding up similarity calculations, an inverted index [2] is commonly used where for each term in the document collection, there is a posting list, that is a list of the ids of documents which contain that term. For each document in the posting list of a term, the corresponding weight of the term in that document is also stored. As a result, the similarity measure is calculated only for documents appearing in the

posting list of at least one of the terms of the query, yielding so a significant saving in the total number of required calculations.

A valuable extension [5, 40, 45], in order to further lower the total retrieval time, is to group the documents into a number of document clusters (cluster-based retrieval) where each cluster contains documents that are most similar to each other. Briefly, the worth of using document clustering in IR (especially for "cluster-based retrieval") is based on the well-known "cluster hypothesis" [50]: "the associations between documents convey information about the relevance of documents to requests." Since each cluster contains similar documents and if the cluster hypothesis holds for a document collection, then finding the most relevant clusters to a given query would automatically retrieve the majority of the relevant documents to the specific query.

One possible way of creating document clusters (followed in our work) is by computing the pair-wise similarities $sim(D_i, D_j)$ for all pairs of documents D_i, D_j $(i \neq j)$ of the document collection and then applying a suitable graph-based algorithm—i.e., connected components' estimation with a specific similarity threshold as the adjacency criterion—over the corresponding documents' similarity matrix. After the formation of the clusters, for each cluster C_i, one centroid-vector c_i is extracted of the following form $c_i = (wc_{i1}, wc_{i2}, \ldots, wc_{im})$ where $wc_{ij,}$ i.e., the weight of term j in cluster C_i, is computed as the average of the weights of term j in documents D_{ik} belonging to cluster C_i. Therefore, the corresponding retrieval task can now be performed via a suitable selective cluster-based searching algorithm (by first comparing q to the centroid-vectors c_i in order to find the most relevant clusters) and finally only a fraction of the whole set of documents in the collection (those belonging to the most relevant clusters) has to be considered. Can et al. in [5] show that by enhancing the posting lists in the classic inverted index with skip pointers so that only portions of posting lists corresponding to documents belonging to the relevant clusters will be accessed, cluster-based retrieval can significantly reduce the total number of posting elements processed and hence the total execution time.

With regard to the connected-components document clustering method, this method is widely used for flat/partitional clustering and it is based on the well known and more general *single-link* hierarchical clustering method. The *single-link* and *complete-link* methods are the most popular ones for hierarchical document clustering [26]. In the *single-link* hierarchical clustering method, the two clusters with the smallest minimum pairwise distance are merged at each step, whereas in the *complete-link* method the two clusters with the smallest maximum pairwise distance are merged, assuming that the smallest distance corresponds to the greatest similarity. Now, the connected-components clustering method can be shown as an instance of the *single-link* method or equivalently a break method for flat/partitional clustering out of a cluster hierarchy. Specifically, if $d(n)$ is the distance of the two clusters to be merged at step n of the *single-link* method, and $G(n)$ is the graph that links all data points within a distance of at most $d(n)$, then the clusters after step n are the connected components of $G(n)$.

Equivalently, the minimum spanning tree algorithm could be used with almost equivalent results both theoretically and practically. The above algorithms, *single-link* and *complete-link* as well as their variants for flat/partitional clustering, connected components, minimum spanning trees, maximal complete subgraphs, etc., are generally regarded as the most appropriate in the domain of document retrieval, although

they are computationally expensive since they require the computation of all the pairwise document similarities. Moreover, they have been proved to perform better than other less time-consuming heuristic/iterative partitional clustering methods such as squared error or k-means clustering methods. That is especially true when these methods are to be used for cluster-based document retrieval or automatic thesaurus construction etc. (see [43]). Also, the *single-link* hierarchical clustering method as well as the "connected components" method have been widely-used together with "centroid-based" mean-cluster vector representatives in classical well-known research IR systems such the SMART and SIRE systems [45].

Generally, (as stated in [5, 40, 45] too) cluster-based retrieval leads to relatively inexact search, however, the expected decrease in search quality is small (depending on the specific algorithms used for cluster formation and representation). The main need for using cluster-based retrieval despite of its qualitative tradeoffs/disadvantages lies on the need for faster response times in very large/huge collections, which is an important issue in IR nowadays. In addition, if the documents belonging to the same cluster and their meta-data (e.g., term-weights, etc.) are stored in contiguous positions in the system disk, the I/O time for cluster-based retrieval can be minimized [5]. Also, encouraging results about the effectiveness of the cluster-based retrieval (higher recall and precision) have recently been shown in [18, 34, 36] where either the basic framework of cluster-based retrieval is used in combination with a statistical language modeling approach for estimating the similarity of documents with a given query [34, 36] or clusters are built by following an information theoretic approach (distributional clustering) and then the clustering structure is used for searching as well as browsing the document collection [18].

4 The VSM-based text retrieval algorithm

In this section, we present the details of our BSP/GCM algorithms for the VSM text retrieval problem. First, we describe the parallel implementation of the off-line preprocessing phase and then we give the details of the on-line query processing phase. Next, we discuss how to handle the case when a number of queries is concurrently submitted to the system (bulk query processing). After that, we present techniques for dynamic cluster maintenance when documents are inserted or deleted from the collection. Finally, we discuss external memory issues related to the fact that in some cases the local memory of the processors is limited, and thus most of the data should be placed in local disks.

For convenience, most of the notation needed for the analysis of our techniques in the next two subsections has been placed in Table 1. Also, for keeping the notation simple and short, some of the symbols in Table 1 are reused with different meaning in the next subsections.

4.1 The off-line preprocessing/clustering phase

We assume that we are given p *BSP/CGM* processors and n documents $D_1, D_2, \ldots, D_n$. For each document D_i, $i = 1, \ldots, n$, its vector representation $d_i = (w_{i1}, w_{i2}, \ldots,$

Table 1 Basic notation of our algorithms

Basic notation for the VSM-based text retrieval algorithm	
n	the number of documents in the collection
p	the number of processors
d_i	the vector representation of document D_i
m	number of distinct terms in document collection
q	the vector representation of query Q
r_q	the maximum number of distinct terms in a query Q
c_i	the centroid vector of cluster C_i
C	the set of clusters in the document collection
d_{sim}	the maximum number of documents similar to a document
n_i	the number of documents stored in processor i when the data volume of document collection is evenly distributed to processors
r_i	the number of non-zero elements of document vector d_i
$r_{\max}$	$\max_{i=1\ldots n} r_i$
R	$\sum_{i=1}^{n} r_i$
$n_{\max}$	$\max_{i=1\ldots p} n_i$
$R_{\max}$	the maximum number of document vector elements stored in a processor when the documents of the collection are evenly distributed to processors

w_{im}) is given. Note that the vectors d_i are usually very sparse and let r_i be the number of nonzero elements of document vector d_i. Initially, each processor holds $O(\frac{R}{p})$ document vector elements where $R = \sum_{i=1}^{n} r_i$. We reasonably assume that $r_i \leq \frac{R}{p}$, and thus each processor will keep only entire document vectors.

The off-line clustering algorithm computes the similarities among the documents, forms the document clusters (centroids), computes the centroid vector of each cluster and evenly distributes the documents of each cluster to the processors. The basic steps of the algorithm are given as follows. A pseudo-code description is also provided in Algorithm 1.

Step 1. Consider the $n \times m$ matrix D, whose row i is the vector d_i, $i = 1, \ldots, n$. Notice that this is a sparse matrix. For each document D_i, $i = 1, \ldots, n$, use a sparse matrix-vector multiplication algorithm to "multiply" the sparse matrix D with the vector d_i, and, therefore, to compute the similarity $sim(D_i, D_j)$ of D_i with each other document D_j, $j = 1, \ldots, n$, $j \neq i$, using the well-known "cosine-similarity" function, i.e.,

$$sim(D_i, D_j) = \frac{\sum_{k=1}^{m} w_{ik} \cdot w_{jk}}{\sqrt{\sum_{k=1}^{m} w_{ik}^2} \cdot \sqrt{\sum_{k=1}^{m} w_{jk}^2}}. \tag{1}$$

Then use the similarity matrix to construct a graph $G = (V, E)$, such that there is a node in the graph for each document in the document collection, i.e., $V = \{D_1, D_2, \ldots, D_n\}$, while there is an edge between two nodes D_i and D_j, if and only if $sim(D_i, D_j)$ is greater than or equal to a threshold value th_1.

Algorithm 1 High level description of the off-line clustering phase

{Step 1}
for each pair of documents (D_i, D_j), $i \neq j$ **do**
  Compute the Similarity function $sim(D_i, D_j)$ from Eq (1)
end for
Define the graph $G = (V, E)$ with $V = \{D_1, D_2, \ldots, D_n\}$ and $e_{ij} \in E$ if $sim(D_i, D_j) \geq th_1$
{Step 2}
Find the clusters/connected components $C = \{C_1, C_2, \ldots, C_{|C|}\}$ of G
{Step 3}
for $i = 1$ to $|C|$ **do** {Compute the centroid vectors}
  $c_i = \frac{\sum_{D_i \in C_i} d_i}{|C_i|}$
end for
for each processor i **do**
  Build a local inverted index for the centroid vectors residing in processor i
end for
{Step 4}
for $i = 1$ to $|C|$ **do** {$\lceil |C_i|/p \rceil$ or $\lfloor |C_i|/p \rfloor$ document vectors per processor for Cluster i after the redistribution}
  Distribute the documents vectors of C_i evenly to processors
end for
{Step 5}
for each processor i **do**
  for $j = 1$ to $|C|$ **do**
    Build a local inverted index for the documents of C_j residing in the processor i
  end for
end for

Step 2. Apply a connected components algorithm to the graph $G = (V, E)$. The connected components of G comprise the set of clusters C of the document collection and, therefore, "similar" documents belong to the same cluster and we expect that relevant documents to each user-query will belong with high probability to some specific clusters.

Step 3. For each centroid (cluster) C_i in C, use the document vectors of its documents to compute the centroid vector c_i. Store the centroid vectors in each processor by using an inverted index structure. As mentioned in Sect. 3, the inverted index is the data structure of choice for almost all information retrieval systems since it helps execute faster the user queries [2, 54].

Step 4. For each centroid C_i in C, evenly distribute the vectors of the documents of the cluster to the p processors. Therefore, after the execution of this step, the document vectors of the collection are almost evenly shared among the processors, i.e., each processor holds approximately $\frac{n}{p}$ document vectors in its local memory.

Step 5. At each processor, build a separate inverted index for the documents of each cluster stored in the local memory of the processor. Now for each term appearing

Algorithm 2 Step 1—parallel implementation

1: **at** round i **do** $\{i = 1 \ldots p\}$
2: Processor i broadcasts its local document vectors d_l^i $(l = 1 \ldots n_i)$ to all processors
3: **for each** processor j **do**
4: **for each** pair of document vectors (d_k^j, d_l^i), $k = 1 \ldots n_j$, $l = 1 \ldots n_i$ **do**
5: Compute the value $S = sim(d_k^j, d_l^i)$
6: Store the value S only if $S \geq th_1$
7: **end for**
8: **end for**
9: **end at**
10: Redistribute the similarity values so that each processor holds the similarities of each of n/p documents to all other documents of the collection
11: **for each** processor i **do** {Define the similarity graph $G = (V, E)$}
12: **for each** local document D_j **do**
13: Define the corresponding vertex V_j of G
14: **end for**
15: **for each** local similarity value $sim(d_k, d_l)$ **do** {Only values $\geq th1$ have been kept (line 6)}
16: Define the corresponding edge e_{kl} of G
17: **end for**
18: **end for**

in the documents, there is a list of ids of the documents containing that term along with the weight of the term in each of these documents.

4.1.1 Detailed description and analysis of the off-line clustering algorithm

Now we will provide the implementation details for each of the basic steps of the algorithm. We also give a pseudocode description for each step.

Step 1. Each processor needs to compute the similarity $sim(D_i, D_j)$ of document D_j with each other document D_i in its local memory. Each processor i keeps $O(\frac{R}{p})$ document vector elements and let n_i be the number of documents corresponding to this amount of elements. Since the document vectors are sparse, we have that $n_i \neq n_j$ for $i \neq j$, in general. Now, each processor i should broadcast their own $O(\frac{R}{p})$ vector elements to all other processors. After receiving a new set of n_i vectors D_l^i $(l = 1 \ldots n_i)$, each processor j computes the similarity $sim(D_k^j, D_l^i)$ among its own set of vectors D_k^j $(k = 1 \ldots n_j)$ with the vectors D_l^i, i.e., $n_i n_j$ similarity values in total.

The whole computation is organized in p rounds (lines 1–9 in Algorithm 2). At round i, processor i broadcasts its $O(\frac{R}{p})$ vector elements to all other processors. By using a well-known technique [21, 27], broadcasting can be completed in $O(g\frac{R}{p} + L)$ time. The technique proceeds in two communication supersteps. In the first step,

Algorithm 3 Step 2—parallel implementation

1: Apply the connected components algorithm of Dehne et al. in [17] on G = (V, E)

processor i partitions the $O(\frac{R}{p})$ elements in p pieces of $O(\frac{R}{p^2})$ elements each, and send each piece to a different processor including itself. The cost of this transfer in the BSP model is $O(g\frac{R}{p}+L)$. Then each processor creates $p-1$ copies of the elements just received and sends these copies to the rest of the processors in $O(g\frac{R}{p}+L)$ time.

At each round after broadcasting, each processor j computes the $n_i n_j$ similarity values among its own vectors D_k^j and the received vectors D_l^i of processor i (lines 3–8 in Algorithm 2). Note that the number of multiplications required for this computation is $\sum_{w=1}^{m}|D^i|_w|D^j|_w$ where m is the total number of terms in document collection (the dimension of document vectors) and $|D^i|_w$, $|D^j|_w$ is the number of documents D_l^i and D_k^j, respectively, that contain the term w. Note also the number of summations for this computation is strictly smaller than the amount above.

Now, we can easily see that $\sum_{w=1}^{m}|D^i|_w|D^j|_w \leq \frac{R}{p}\min(n_i, n_j)$. Note that this bound is tight. Just consider the case that the documents D_l^i and D_k^j contain only the first $\lceil\frac{R}{pn_i}\rceil$ and $\lceil\frac{R}{pn_j}\rceil$, respectively terms of the document collection.

Thus, the total running time of round i is $O(\frac{R}{p}n_i + g\frac{R}{p}+L)$, and hence for the p rounds of the whole computation we need $O(\frac{R}{p}n + gR + pL)$ time in total.

It is worth mentioning that after each computation round, processors need to keep only those similarity values which are above the threshold value th_1. In this way, after the completion of the p rounds, the local memory requirement in each processor for storing these values is at most $O(n_{\max}d_{\text{sim}})$ where $n_{\max} = \max_{1=1\ldots p} n_i$ and d_{sim} is the maximum number of documents D_j, such that $sim(D_i, D_j) > th_1$, over all documents D_i. It is also reasonable to assume that parameter d_{sim} is $o(n)$, i.e. much smaller than n, the number of documents in the collection.

Next, we redistribute the $O(nd_{\text{sim}})$ similarity values so that each processor will store the similarity values for $\frac{n}{p}$ documents, that is $O(\frac{n}{p}d_{\text{sim}})$ similarity values at most (line 10 in Algorithm 2). Apparently, this step takes $O(gn_{\max}d_{\text{sim}})$ time at most.

With the local $O(\frac{n}{p}d_{\text{sim}})$ similarity values as input, each processor creates its own portion of the document similarity graph $G(V, E)$ (lines 11–18 in Algorithm 2). Specifically, graph G consists of n vertices which represent the n documents of the collection and there is an edge between two vertices V_i, V_j if the corresponding similarity value $sim(D_i, D_j)$ is above the threshold th_1. Clearly, each processor holds $\frac{n}{p}$ vertices and $O(\frac{n}{p}d_{\text{sim}})$ edges of the graph G.

Step 2. The *BSP* version of the *CGM* connected components algorithm of Dehne et al. [17] is used to compute the connected components of the graph G constructed in the previous step (Algorithm 3). Notice that G is a graph with n vertices whose number of edges is at most $m = O(nd_{\text{sim}})$. Therefore, the *CGM* connected components algorithm needs $O(\log p)$ communication rounds and $O(\frac{nd_{\text{sim}}}{p})$ local computation per round while the communication cost of the *BSP* connected components algorithm is $O(\log p(g\frac{nd_{\text{sim}}}{p}+L))$ and the computation cost is $O(\frac{nd_{\text{sim}}}{p}\log p)$.

Algorithm 4 Step 3—parallel implementation

1: Let $C = \{C_{i_1}, C_{i_2}, \ldots, C_{i_{|C|}}\}$ be the set of clusters/connected components from Step 2.
2: Sort the set C according to the indices of the clusters
3: Based on this order, renumber the clusters in the range $[1 \ldots |C|]$
4: Return the documents to their initial positions as they were before Step 1
5: **for each** processor j **do**
6: **for each** Cluster C_i **do**
7: $lc(i, j) = 0$ {$lc(i, j)$: the local centroid of C_i in processor j}
8: **for each** local document $D_k \in C_i$ **do**
9: $lc(i, j) = lc(i, j) + d_k$
10: **end for**
11: **end for**
12: **end for**
13: Let $b(i, j)$ be the number of non-zeroes in $lc(i, j)$
14: Compute $S(i, j)$ for $i = 1 \ldots |C|$, $j = 1 \ldots p$ using Algorithm 5 {$S(i, j)$ is given by Eq. (2)}
15: **for each** processor j **do**
16: **for** $i = 1 \ldots |C|$ **do**
17: Find k such that $(k-1)\frac{R}{p} < S(i, j) \leq k\frac{R}{p}$
18: Send $lc(i, j)$ to processor k
19: **end for**
20: **end for**
21: {Now, the p (at most) local centroids of each cluster are stored in consecutively numbered processors}
22: **for each** processor j **do**
23: **for each** Cluster i with local centroid vectors in processor j **do**
24: Add the local centroid vectors of Cluster i and store the result in $lc(i, j)$
25: **end for**
26: **end for**
27: {Here, for a cluster i totally contained in one processor j, the corresponding centroid vector has already been computed, i.e. $c_i = lc(i, j)$}
28: **for each** Cluster i spanning more than one processor **do** {in parallel}
29: Compute the Centroid vector of C_i using Algorithm 6
30: **end for**
31: **for each** processor i **do**
32: Organize the locally stored centroid vectors (or part of them) with an inverted index.
33: **end for**

Step 3. At the end of step 2, each document D_i knows the cluster it belongs to while the cluster-heads are distributed to the p processors. During step 3, the following substeps take place:

- Using sorting, give to the clusterheads consecutive numbering from C_1 to $C_{|C|}$, where $|C|$ is the number of clusters (lines 1–3 in Algorithm 4). Due to [6], sorting

Algorithm 5 Compute $S(i, j)$ for $i = 1, \ldots |C|$, $j = 1, \ldots, p$

1: **for each** processor j **do** $\{j = 1, \ldots, p\}$
2: Send the $\frac{|C|}{p}$ elements $b(k, j)$, $(k = (i-1)\frac{|C|}{p} + 1, \ldots i\frac{|C|}{p})$ to processor i, $(i = 1, \ldots, p)$
3: Compute the prefixes $SC_k(i) = \sum_{l=1}^{i} b(k, l)$, for $k = (j-1)\frac{|C|}{p} + 1, \ldots j\frac{|C|}{p}$ and $i = 1, \ldots, p$
4: Send the prefixes $SC_k(i)$ $(k = (j-1)\frac{|C|}{p} + 1, \ldots j\frac{|C|}{p})$ to processor i, $(i = 1, \ldots, p)$
5: **if** $j = p$ **then** {only processor p executes this part}
6: Compute the prefixes $GS(i) = \sum_{k=1}^{i} SC_k(p)$, $(i = 1, \ldots |C|)$
7: Broadcast the values $GS(i)$ to all processors.
8: **end if**
9: Compute $S(i, j) = \sum_{l=1}^{i-1} GS(l) + SC_i(j)$, $(i = 1, \ldots |C|)$
10: **end for**

can be completed in $O(\frac{n}{p} + g\frac{n}{p} + L)$ time with a constant number of communication and computation rounds.

- Return the documents to the their initial positions (line 4 in Algorithm 4). Specifically, return each document and its cluster identity to the processor that holds the corresponding document vector. This can be easily done by having remembered the position of each document before the redistribution substep at the end of step 1. This transfer takes $O(gn_{\max})$ time at most.
- By using its local document vectors, each processor j, $1 \leq j \leq p$ computes a "local" centroid vector $lc(i, j)$ for each centroid (cluster) C_i which has at least one document vector in the local memory of processor j (lines 5–12 in Algorithm 4). This requires $O(\frac{R}{p})$ local computation cost. Let $b(i, j)$ be the number of nonzero elements of local" centroid vector $lc(i, j)$. A crucial observation at this point is that the number of nonzeroes $b(i, j)$ in the "local" centroid vector $lc(i, j)$ is at most $O(\frac{R}{p})$. Thus, the computation of the centroid vectors of clusters C_i from the local centroid vectors $lc(i, j)$ should be done in such a way that no processor receive more than $O(\frac{R}{p})$ vector elements in the process. This can be achieved by the following three steps.
- For $i = 1, \ldots, |C|$, $j = 1, \ldots, p$, compute the expressions[1]

$$S(i, j) = \sum_{k=1}^{i-1} \sum_{l=1}^{p} b(k, l) + \sum_{l=1}^{j} b(i, l) \tag{2}$$

by adapting the standard prefix-sum technique [21] (lines 13–14 in Algorithm 4 and Algorithm 5). Clearly, each of the steps in Algorithm 5 involves either the execution of a $|C|$-relation or a computation step with $O(|C|)$ additions. Thus, the total running time for the calculation of $S(i, j)$ is $O(|C| + g|C| + L)$ at most.

- Every processor j, $1 \leq j \leq p$, for each $i = 1, \ldots, |C|$, sends its "local" centroid vector $lc(i, j)$, to processor k, $k = 1, \ldots, p$, if and only if $(k-1)\frac{R}{p} < S(i, j) \leq k\frac{R}{p}$

[1] For $i = 1$, the first sum is equal to 0.

Algorithm 6 Compute the Centroid vector of C_i

1: Let $j_i \ldots j_i + l - 1$ be the processors where the local centroids of C_i have been stored
2: Let $lc(i, j, k)$ be the weight of $lc(i, j)$ at index k ($k = 1 \ldots m$, $j = j_i \ldots j_i + l - 1$)
3: In parallel sort all $lc(i, j, k)$ values according to their index k
4: {Now all the weights of the same index are stored in the same or in consecutively numbered processors}
5: In parallel add the weights of the same index
6: {The centroid vector c_i may now be distributed in consecutively numbered processors}

(lines 15–20 in Algorithm 4). Every processor sends all its "local" centroid vectors in one (combined) h-relation. Since each processor sends and receives at most $O(\frac{R}{p})$ vector elements, only one $O(\frac{R}{p})$-relation is required for this step.

- Each processor j uses the "local" centroid vectors $lc(i, k)$, $k = 1, \ldots, p$ previously received to compute a new aggregated local centroid vector $lc(i, j)$, for each cluster C_i (lines 22–27 in Algorithm 4). This step requires at most $O(\frac{R}{p})$ local computation. Note that if all the local centroid vectors of a cluster C_i have been gathered from the previous step exclusively in processor j, then at this step the centroid vector c_i of cluster C_i has been computed, that is $c_i = lc(i, j)$. However, if a cluster C_i is relatively large, its "local" centroid vectors $lc(i, j)$ may be spread among a number of l processors, say processors j with $j = j_i, \ldots, j_i + l - 1$. It is possible that the final centroid vector c_i may not contain $O(\frac{R}{p})$ elements at most, and thus it cannot be stored in one processor. In this case, we keep the nonzeroes of centroid c_i distributed with each of the l processors holding $O(\frac{R}{p})$ nonzeroes at most. The distributed computation of the centroid vector c_i can be performed as follows (lines 28–33 in Algorithm 4 and Algorithm 6):
  - For each nonzero element of the "local" centroid vectors $lc(i, j)$, form the pair $(k, lc(i, j, k))$ where k is the position of the nonzero element inside vector $lc(i, j)$ and $lc(i, j, k)$ is the value of this element.
  - By using the algorithm in [23], sort pairs $(k, lc(i, j, k))$ by k in $O(1)$ rounds. The total computation and communication cost is $O(\frac{R \log R}{p})$, $O(g\frac{R}{p} + L)$, respectively. Now after this sorting step, all vector elements at the same position k have been placed consecutively.
  - Now, the centroid vector elements can be easily computed, by first adding the proper local elements in $O(\frac{R}{p})$ time and then adding the partial sums in one communication and computation round with a total $O(p)$ computation and $O(gp + L)$ communication cost.
  - The local centroid vector elements in each processor are organized by using an inverted index structure. This means that for each nonzero position k, there is a list of pairs (id, val) where id is the id of a cluster whose centroid vector contain an nonzero element at this position and val is the corresponding nonzero value. Building the inverted index involves a sorting step [54] of $O(\frac{R}{p})$ pairs (k, id, val) according to the non-zero position k. For this sorting step, we can use the merge-sort algorithm starting from $O(|C|)$ ordered lists each one corresponding to a

Algorithm 7 Step 4—parallel implementation

1: Redistribute the documents vectors of each cluster C_i evenly to processors $\{\lceil |C_i|/p \rceil$ or $\lfloor |C_i|/p \rfloor$ vectors per processor$\}$

Algorithm 8 Step 5—parallel implementation

1: **for each** each processor i **do**
2: **for** $j = 1$ to $|C|$ **do**
3: Build a local inverted index for the documents of cluster C_j residing in processor i
4: **end for**
5: **end for**

local centroid vector. They are $O(\log |C|)$ merge phases at most with $O(\frac{R}{p})$ cost per phase. Thus, the total cost of the merge sort algorithm is $O(\frac{R}{p} \log |C|)$ at most.

Clearly, the three steps above demand a constant number of communication and computation rounds and the total computation and communication cost is $O(\frac{R \log R}{p})$ and $O(g\frac{R}{p} + L)$, correspondingly.

Thus, with the completion of Step 3 the centroid vectors have been distributed among the processors and each processor has $O(\frac{R}{p})$ nonzero centroid vector elements at most in its local memory.

Step 4. This step requires a redistribution of document vectors so that the document vectors relevant to each cluster are evenly distributed among the p processors (Algorithm 7). By performing a prefix computation similar to that for the computations of vectors GS and S at Step 3 (see Algorithm 5), each processor can easily determine the processors which it should communicate with. Then the redistribution involves a $R_{\max}$-relation where $R_{\max}$ is the maximum number of document vector elements stored in any processor after this redistribution step. Thus, the total time for this step is $O(|C| + g \max\{|C|, R_{\max}\} + L)$.

Now, each processor contains $O(\frac{n}{p})$ document vectors. Since each document vector has a different number of nonzero elements, this document allocation may create an uneven load distribution over the processors. However, this allocation is essential for achieving fast execution in the on-line query processing phase as will be explained later.

Step 5. For the documents inside the local memory of each processor, build an inverted index structure (Algorithm 8). This means that for each term of the local document collection there is a list of local documents containing that term along with the weight of the term in each of these documents. We can also follow the technique of Can et al. [5] where along each posting list of the inverted index there are skip pointers so that the portions corresponding to a specific cluster can be fast accessed.

Similarly to local centroid inverted index, building this inverted index requires a sorting step of $O(R_{\max})$ pairs (t_i, d_j, w_{ij}, C_j) according first to the their terms t_i and

then to C_j where w_{ij} is the nonzero weight of term t_i in document d_j and the C_j is the cluster containing the document d_j. Again, we use the merge sorting algorithm starting now from $O(\frac{n}{p})$ ordered lists each one corresponding to a local document vector. The number of merge steps is $O(\log \frac{n}{p})$ at most and each step has $O(R_{\max})$ cost at most. Thus, the total cost of the sorting step is $O(R_{\max} \log \frac{n}{p})$ at most.

From the analysis above, we can easily prove the following theorem.

Theorem 1 *By reasonably assuming that* $|C| < \frac{R}{p}$, $r_{\max} \leq \frac{R}{p}$, $d_{\text{sim}} = o(n)$, $p^2 \leq n$, $R \geq n \log p$ *and* $n \geq \log n + \log r_{\max}$ *where* $r_{\max} = \max_{i=1\ldots n} r_i$, *the off-line clustering phase of the VSM-based text retrieval algorithm has a total of* $O(\frac{nR}{p})$ *computation and* $O(g \max\{R, \frac{R}{p} d_{\text{sim}}\} + pL)$ *communication cost*, *whereas the local memory requirement for each processor is* $O(\max\{\frac{R}{p} d_{\text{sim}}, \frac{n}{p} r_{\max}\})$ *at most*.

Proof By checking the complexity of each step, we can easily see that the total computation cost is

$$O\left(\max\left\{\frac{R}{p}n, \frac{nd_{\text{sim}}}{p}\log p, |C|, \frac{R\log R}{p}, \frac{R}{p}\log|C|, R_{\max}\log\frac{n}{p}\right\}\right)$$

at most. Note that $R \gg n$ and $n \gg d_{\text{sim}}$. So, we can reasonably assume that $R > d_{\text{sim}} \log p$. Thus, it holds that $\frac{R}{p}n > \frac{nd_{\text{sim}}}{p}\log p$. Also, we have that $R \geq R_{\max}$ and so $R\frac{n}{p} \geq R_{\max}\log\frac{n}{p}$. Moreover, if $n \geq \log n + \log r_{\max}$ then we also get that $\frac{R}{p}n > \frac{R\log R}{p}$. Finally, by assuming w.l.o.g that $|C| < \frac{R}{p}$, the total computation cost of the off-line clustering phase turns out to be $O(\frac{nR}{p})$ at most.

As for the communication cost of the off-line clustering phase, the total complexity is

$$O\left(g\max\left\{R, n_{\max}d_{\text{sim}}, \log p\frac{nd_{\text{sim}}}{p}, |C|, R_{\max}\right\}\right)$$

at most.

Again, we assume that $|C| < \frac{R}{p}$. Clearly also, it holds that $\frac{R}{p} > n_{\max}$, $R > R_{\max}$. Now, provided that $R \geq n \log p$, the total computation complexity is $O(g \max\{R, \frac{R}{p} d_{\text{sim}}\})$ at most.

Note that the condition $R \geq n \log p$ also covers the condition $R > d_{\text{sim}} \log p$ mentioned above for the computation cost.

Finally, the local memory requirement for this phase is

$$O\left(\max\left\{\frac{R}{p}, |C|, n_{\max}d_{\text{sim}}, R_{\max}\right\}\right)$$

at most.

Since, $\frac{R}{p} > n_{\max}$, $\frac{n}{p} r_{\max} > R_{\max}$ and $|C| < \frac{R}{p}$, it easily follows that local memory requirement at each processor is $O(\max\{\frac{R}{p} d_{\text{sim}}, \frac{n}{p} r_{\max}\})$ at most. □

Now, it would be interesting to compare the running time of our parallel algorithm with that of the corresponding sequential algorithm which implements the off-line

Algorithm 9 On line query processing—a high level description

```
1: q = the vector of the user query Q
2: for each Cluster C_k do {Step1}
3:    Compute the similarity value sim(q, c_k)
4:    if sim(q, c_k) > th_2 then {th_2 is a threshold}
5:       C_i is a qualifying cluster{The documents of C_i will be checked for similarity with q}
6:    end if
7: end for
8: for each Qualifying Cluster i do {Step2}
9:    for each document D_l in Cluster i do
10:      Compute the similarity value sim(q, d_l)
11:      if sim(q, d_l) > th_3 then {th_3 is a threshold}
12:         D_l is a qualifying document{It will be included in the retrieved documents}
13:      end if
14:   end for
15: end for
16: Sort the qualifying documents according to their similarity scores{Step 3}
```

clustering phase. First, the sequential sparse matrix multiplication needs $\Theta(nR)$ time at most. Then by repeated application of depth first search on the similarity graph of documents which is a graph with $N = n$ vertices and $M = nd_{\text{sim}}$ edges, we can find the connected components of the graph in $\Theta(N + M)$ or equivalently $\Theta(nd_{\text{sim}})$ time at most [32]. Finally, the computation of cluster centroid vectors takes $\Theta(R)$ time at most. So, the total time of sequential off-clustering algorithm is $\Theta(nR)$ at most. By comparing this complexity with those of Theorem 1, we can easily see that the speed-up of our technique is close to p for sufficiently large n and R.

4.2 The on-line query processing phase

Let Q be a user query whose query vector q contains r_q terms at most. Note that users most often submit queries with a small number of terms especially in web interfaces [40, p. 87]. Thus, we can assume that the parameter r_q is a relatively small integer constant most of time.

The on-line query processing algorithm proceeds as follows (see also Algorithm 9):

Step 1. Compute the "cosine-similarity function" of vector q with each of the centroid vectors c_k, $(k = 1 \ldots |C|)$. There are two possibilities for this computation. If all the elements of the centroid vector have been stored in a single processor, the computation is local and we can use the inverted index structure of local centroid vectors to complete this processing in $O(|C|r_q)$ time at most. Otherwise, if the centroid vector is distributed among a number of processors, the whole computation needs one communication and two computation rounds. The size of messages exchanged is $O(1)$ at most and the communication required is an $O(p)$-relation.

The computation rounds can be completed in $O(|C|r_q)$ time overall, by making use of the inverted index structure again.

Now, for each centroid C_k such that $sim(q, c_k)$ is greater than or equal to a threshold value th_2, consider it as a "qualifying cluster." Then broadcast the ids of these clusters to all other processors with $O(g|C| + L)$ communication cost at most.

Step 2. By using the "cosine-similarity function," each processor locally computes the similarity of query Q with each one of its local documents that belong to the "qualifying clusters" according to the previous step. Based on the inverted index structure of local documents, this local computation can be completed in $O(\frac{n}{p}r_q)$ time at most. Then each processor extracts the documents D_l such that $sim(q, d_l)$ is greater than or equal to a threshold value th_3. These documents are considered as "qualifying documents."

Step 3. Rank the qualifying documents according to their similarity scores with the query Q. For the similarity score values, we can safely assume finite precision, 3 or 4 decimal digits, for instance, since difference in similarity values smaller than this amount is immaterial and the corresponding documents can be considered equivalent in the final document ranking. Thus, this ranking can be easily done by employing integer sorting and specifically the algorithm in [6] with $O(\frac{n}{p})$ computation and $O(g\frac{n}{p} + L)$ communication cost.

Now, we can easily see that the following theorem holds.

Theorem 2 *The on-line query processing phase of the VSM-based text retrieval algorithm requires* $O(\max\{|C|, \frac{n}{p}\}r_q)$ *computation and* $O(g\frac{n}{p} + L)$ *communication time. The local memory in each processor required for this phase is* $O(\frac{n}{p}r_{\max})$ *at most.*

Recall again that the parameter r_q is a small integer, and thus the computation and communication cost of the on-line query phase increases linearly with the number of local documents stored in each processor. That was exactly the reason why we evenly redistributed the document vectors among the processors at the step 4 of the off-line clustering phase. In this way, we minimize the query execution time during this phase.

With regard to the speedup of our technique, we can easily check that the running time of the corresponding sequential algorithm is $\Theta((|C| + n)r_q)$ at most. So, the speedup offered by our technique is close to p for sufficiently large number of documents n and a relatively small number of clusters $|C|$.

4.3 On the bulk query processing

The previous section presented the details of the single query execution on the proposed IR system. A highly relevant and important issue is how the system processes a number of concurrently submitted queries. Today's commercial search engines accept thousand requests and they should respond to all these IR requests in a timely manner.

In this section, we show that our basic approach fares very well in comparison to other reasonable approaches of handling a number of concurrently submitted queries

to an IR system. For our analysis, we assume that q queries have been submitted to the IR system at the same time. Then the first step of our on-line query processing phase of the previous subsection is executed in order that the q queries can be classified according to their most relevant cluster. We assume that each query is relevant to only one of the $|C|$ clusters of the collection. For our purposes, this assumption is acceptable since we only intend to present an analysis that will be indicative of the real system performance. Our analysis is also valid in the following scenario common in major web search engines where users first browse the document collection using the existing clustering structure and then, after deciding which of the topics of the clusters is the most pertinent to their information need, they start searching among the documents of the most relevant cluster using the VSM model.

Now in our case, let q_i be the number of queries that are found relevant to cluster i $(i = 1 \ldots |C|)$. We also introduce the fractional parameter f_i such that $q_i = f_i q$ $(i = 1 \ldots |C|)$ i.e. f_i is the percent over all the queries that are relevant to cluster C_i, and $\sum_{i=1}^{|C|} f_i = 1$.

Then in our scheme of the previous subsection (*Single Query Processing (SQP) scheme*), the q queries are processed sequentially, one after another, by repeatedly executing the last two steps of the on-line query phase. Recall also that the documents of each of the $|C|$ clusters have been evenly distributed among the p processors of the system. Another reasonable approach is the parallel execution of queries concerning different clusters (*Multiple Query Processing (MQP) scheme*). More precisely, we dedicate a certain number of processors, p_i, to each cluster C_i and the documents belonging to this cluster are only distributed to the p_i specific processors. If cluster C_i contains n_i documents out of the n documents of the whole collection, each of the p_i processors of the cluster i holds $O(\frac{n_i}{p_i})$ documents at most. Then the q_i queries for cluster i are processed one after the other on the p_i processors by repeatedly following again the last two steps of the on-line query phase. In addition, the queries of each cluster i are executed in parallel with other queries concerning different clusters j $(i \neq j)$. Understandably, more processors should be devoted to clusters whose documents are in a great demand. So, it is normal to consider that $p_i = f_i p$ since f_i reflects the "popularity" of cluster i among the concurrently submitted queries. For convenience in the subsequent analysis, we assume that p_i for each i as defined above, is an integer.

In the following, we are going to show that the SQP scheme outperforms the MQP scheme with regard to two important metrics, namely the total execution time (makespan) and the average completion time. These two metrics have been widely used in the literature for evaluating the performance of scheduling algorithms [30].

In our case, if t_i is the completion time of query i $(i = 1 \ldots q)$, then the makespan (MS) and the average completion time (ACT) are defined as follows:

$$\mathrm{MS} = \max_{i=1\ldots q} t_i,$$

$$\mathrm{ACT} = \frac{\sum_{i=1}^{q} t_i}{q}.$$

From system's perspective, the makespan metric is very important because it shows how efficiently the proposed scheduling algorithm uses the CPU resources of

the system. However, from users' perspective, the average completion time is more important because this metric is indicative of the average delay users experience before getting the reply to their query.

In the following two theorems, we prove that the SQP scheme outperforms the MQP scheme with regard to the above mentioned metrics.

Theorem 3 *If the latency parameter L is negligible when compared with the communication cost due to message transfer and if also each one of the submitted queries is relevant to only one of the clusters, then it holds that*

$$\mathrm{MS}_{\mathrm{SQP}} \leq \mathrm{MS}_{\mathrm{MQP}},$$

where $\mathrm{MS}_{\mathrm{SQP}}$ *and* $\mathrm{MS}_{\mathrm{MQP}}$ *is the makespan of the* SQP *and the* MQP *Scheme, respectively.*

Proof In the SQP scheme, we first execute the q_1 queries of cluster 1, then the q_2 queries of cluster 2 and so on. Note also that the last two steps of the on-line query processing phase for a query relevant to cluster i takes

$$O\left(\frac{n_i}{p} r_q + g\frac{n_i}{p} + L\right)$$

time at most. The same time for the parallel scheme is

$$O\left(\frac{n_i}{p_i} r_q + g\frac{n_i}{p_i} + L\right).$$

In our analysis, it is reasonable to assume that gn_i/p as well as $gn_i/p_i \gg L$ since the number of documents n_i in each cluster i are so large that the terms gn_i/p and gn_i/p_i dominate in the communication complexity in comparison to the parameter L. For convenience also, we define the monotonically increasing function

$$A(x) = xr_q + gx.$$

So, eventually the above time complexities can be written as $O(\frac{A(n_i)}{p})$ and $O(\frac{A(n_i)}{p_i})$ for the SQP and MQP scheme respectively.

Now, we can easily see that the makespan (MS) of the SQP scheme is the sum of the execution time of the individual queries that is

$$\mathrm{MS}_{\mathrm{SQP}} = O\left(\sum_{i=1}^{|C|} q_i \frac{A(n_i)}{p}\right)$$

or equivalently

$$\mathrm{MS}_{\mathrm{SQP}} = O\left(\sum_{i=1}^{|C|} q_i \frac{A(n_i)}{p}\right) \overset{q_i = f_i q}{\Longrightarrow}$$

$$\mathrm{MS}_{\mathrm{SQP}} = O\left(\sum_{i=1}^{|C|} f_i q \frac{A(n_i)}{p}\right) \Longrightarrow$$

$$\mathrm{MS}_{\mathrm{SQP}} = O\left(q \sum_{i=1}^{|C|} f_i \frac{A(n_i)}{p}\right).$$

In the MQP scheme, the queries of each cluster are executed sequentially and at the same time in parallel with the queries belonging to different clusters. As the total execution of the queries of cluster i is $O(q_i \frac{A(n_i)}{p_i})$ or equivalently $O(q \frac{A(n_i)}{p})$ since $p_i = f_i p$ and $q_i = f_i q$, we can easily see that the makespan of the MQP scheme is as follows:

$$\mathrm{MS}_{\mathrm{MQP}} = O\left(\max_{i=1\ldots|C|} q \frac{A(n_i)}{p}\right).$$

From the expression above, we can easily see that the cluster with the largest number of documents dominates in the makespan complexity. So, if $i_{\max}$ is the index of that cluster ($n_{i_{\max}} \geq n_i$, $i = 1 \ldots |C|$), then the makespan for the MQP scheme can also be written as follows:

$$\mathrm{MS}_{\mathrm{MQP}} = O\left(q \frac{A(n_{i_{\max}})}{p}\right).$$

Note the bounds implied by the O-notation in the expressions of $\mathrm{MS}_{\mathrm{SQP}}$ and $\mathrm{MS}_{\mathrm{MQP}}$ are tight and we could used Θ-notation instead. So, we can safely remove the O-notation and in order to prove that $\mathrm{MS}_{\mathrm{SQP}} \leq \mathrm{MS}_{\mathrm{MQP}}$, we only need to prove that

$$q \sum_{i=1}^{|C|} f_i \frac{A(n_i)}{p} \leq q \frac{A(n_{i_{\max}})}{p}.$$

Indeed,

$$q \sum_{i=1}^{|C|} f_i \frac{A(n_i)}{p} \leq q \frac{A(n_{i_{\max}})}{p} \Longrightarrow$$

$$\sum_{\substack{i=1\ldots|C| \\ i \neq i_{\max}}} f_i A(n_i) \leq A(n_{i_{\max}})(1 - f_{i_{\max}}) \Longrightarrow$$

$$\sum_{\substack{i=1\ldots|C| \\ i \neq i_{\max}}} f_i (A(n_i) - A(n_{i_{\max}})) \leq 0.$$

The last inequality obviously holds because $A(n_i) \leq A(n_{i_{\max}})$ for $i = 1 \ldots |C|$. □

Now, we will show that the SQP scheme performs equally well with respect to the average completion time metric in the case that the queries are properly scheduled according to a predefined order. Specifically, we first execute the queries of the cluster

with the smallest number of documents, then the queries of the cluster with the second smallest number of documents and so on. So, we "visit" the clusters according to a nondecreasing order of their number of documents. Now, we can prove the following theorem.

Theorem 4 *If the latency parameter L is negligible compared with the communication cost due to message transfer and if also the queries are sequentially executed by "visiting" the clusters with an nondecreasing order in the number of documents per cluster and if, in addition, each one of the submitted queries is relevant to only one of the clusters, then it holds that*

$$\mathrm{ACT}_{\mathrm{SQP}} \leq \mathrm{ACT}_{\mathrm{MQP}},$$

where $\mathrm{ACT}_{\mathrm{SQP}}$ *and* $\mathrm{ACT}_{\mathrm{MQP}}$ *are the average completion time of the* SQP *and the* MQP *Scheme respectively.*

Proof For convenience, we renumber the $|C|$ clusters of the collection so that the sequence $\{n_1, n_2, \ldots, n_{|C|}\}$ forms an nondecreasing integer sequence.

Now, we can easily check that the completion time t_i^j of the ist query of cluster j ($i = 1 \ldots q_j$, $j = 1 \ldots |C|$) in the SQP scheme is given by the following expression:

$$t_i^j = O\left(\sum_{k=1}^{j-1} q_k \frac{A(n_k)}{p} + i\frac{A(n_j)}{p}\right).$$

The expression $q_k \frac{A(n_k)}{p}$ in the first sum is the total execution time of the q_k queries of cluster k, and thus the first sum is the delay due to the execution of queries of the previous $j-1$ clusters. The term $i\frac{A(n_j)}{p}$ in the expression above is the total time required for the first i queries of the cluster j.

Now the average completion time (ACT) for the SQP scheme will be

$$\mathrm{ACT}_{\mathrm{SQP}} = \frac{\sum_{j=1}^{|C|}\sum_{i=1}^{q_j} t_i^j}{q}.$$

By some algebraic manipulation and recalling that $q_i = f_i q$, we finally get that

$$\mathrm{ACT}_{\mathrm{SQP}} = O\left(q \sum_{1\leq i<j\leq |C|} f_i f_j \frac{A(n_i)}{p} + \sum_{i=1}^{|C|} \frac{f_i(f_i q+1)}{2}\frac{A(n_i)}{p}\right).$$

In the MQP scheme, we can easily check that the completion time t_i^j of the ist query of cluster j will be

$$t_i^j = i\frac{A(n_j)}{p_j}.$$

So, the average completion time for the MQP scheme will be

$$\mathrm{ACT}_{\mathrm{MQP}} = \frac{\sum_{j=1}^{|C|}\sum_{i=1}^{q_j} t_i^j}{q} \overset{\substack{p_j=f_j p,\\ q_j=f_j q}}{\Longrightarrow}$$

$$\mathrm{ACT}_{\mathrm{MQP}} = O\left(\sum_{i=1}^{|C|} \frac{(f_i q+1)}{2}\frac{A(n_i)}{p}\right).$$

Again, the bounds in the above O-notations are tight. Therefore, in order to prove that $\mathrm{ACT}_{\mathrm{SQP}} \le \mathrm{ACT}_{\mathrm{MQP}}$, we need only to show that

$$q\sum_{1\le i<j\le|C|} f_i f_j \frac{A(n_i)}{p} + \sum_{i=1}^{|C|}\frac{f_i(f_i q+1)}{2}\frac{A(n_i)}{p} \le \sum_{i=1}^{|C|}\frac{(f_i q+1)}{2}\frac{A(n_i)}{p}.$$

Notice that $f_i\frac{A(n_i)}{p} \le \frac{A(n_i)}{p}$ for $i = 1\ldots|C|$. So, we actually need to show that

$$q\sum_{1\le i<j\le|C|} f_i f_j \frac{A(n_i)}{p} + \sum_{i=1}^{|C|}\frac{f_i f_i q}{2}\frac{A(n_i)}{p} \le \sum_{i=1}^{|C|}\frac{f_i q}{2}\frac{A(n_i)}{p}.$$

Indeed,

$$q\sum_{1\le i<j\le|C|} f_i f_j \frac{A(n_i)}{p} + \sum_{i=1}^{|C|}\frac{f_i f_i q}{2}\frac{A(n_i)}{p} \le \sum_{i=1}^{|C|}\frac{f_i q}{2}\frac{A(n_i)}{p} \Longrightarrow$$

$$\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \le \sum_{i=1}^{|C|}\frac{f_i}{2}A(n_i) - \sum_{i=1}^{|C|}\frac{f_i f_i}{2}A(n_i) \Longrightarrow$$

$$\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \le \sum_{i=1}^{|C|}\frac{(1-f_i)f_i}{2}A(n_i) \Longrightarrow$$

$$\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \le \frac{1}{2}\sum_{i=1}^{|C|}\left(\sum_{\substack{j=1\ldots|C|,\\ j\ne i}} f_j\right) f_i A(n_i) \Longrightarrow$$

$$\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \le \frac{1}{2}\sum_{i=1}^{|C|}\left(\sum_{\substack{j=1\ldots|C|,\\ j<i}} f_j + \sum_{\substack{j=1\ldots|C|,\\ j>i}} f_j\right) f_i A(n_i) \Longrightarrow$$

$$\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \le \frac{1}{2}\sum_{i=1}^{|C|}\left(\sum_{\substack{j=1\ldots|C|,\\ j<i}} f_j\right) f_i A(n_i)$$

$$+\frac{1}{2}\sum_{i=1}^{|C|}\left(\sum_{\substack{j=1\ldots|C|,\\ j>i}} f_j\right) f_i A(n_i) \Longrightarrow$$

$$\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \le \frac{1}{2}\sum_{1\le i<j\le|C|} f_i f_j A(n_j) + \frac{1}{2}\sum_{1\le i<j\le|C|} f_i f_j A(n_i) \Longrightarrow$$

$$\frac{1}{2}\sum_{1\le i<j\le|C|} f_i f_j (A(n_i) - A(n_j)) \le 0.$$

Due to the particular scheduling order, we follow for executing the concurrently submitted queries, it holds that $A(n_i) \le A(n_j)$ for any $i < j$ and $i, j = 1 \ldots |C|$. So the left-hand expression is always a nonpositive number and, therefore, the inequality above holds. □

The previous two theorems clearly show that the even distribution of the documents of each cluster among all the processors is a well-justified approach that fares very well in comparison to other more involved alternatives. Note also, that SQP and MQP scheme have equivalent performance only in the rare case when each cluster C_i of the document collection contains the same number of documents, that is, $n_i = n/|C|$.

4.4 Dynamic cluster maintenance

With the advent of the world wide web, today's IR systems have become highly dynamic with frequent document insertions and deletions. For instance, a search engine should continually update its collection with documents that crawlers track down on the Internet. Further, a search engine should withdraw any document that has become obsolete and cannot be found on the Internet.

When clustering is used for organizing a dynamic document collection, a problem frequently arising is how to adapt the already built cluster structure so that it reflects the new content. Although complete reclustering is the obvious solution, it is generally agreed [1, 4, 7, 25] that this operation should be executed as rarely as possible, because it heavily penalizes the performance of a fully operational IR system. Thus, most of the proposed systems in the literature try to find a compromise between cluster adaptation cost and accuracy of the new clusters.

As a high-performance parallel technique, our VSM-based algorithm is mainly used for handling very large document collections. Thus, we can reasonably assume that the off-line indexing phase starts with a very large number of documents as input, and the clusters, the output of this phase, are relatively stable corresponding to well-separated topics of the document collection. Therefore, during the on-line operational

mode of the IR system, document insertions and deletions are not expected to change dramatically the existing clusters.

Based on this assumption, a new document inclusion can be done by simply finding the most similar cluster for the new document, namely the cluster whose centroid vector is the most similar to the new document vector. This is done as follows:

- By performing the first step of the on-line query processing phase, each processor can determine which centroid vector out of its local ones has the greatest similarity with the new document vector. By using the local centroid inverted index structure, this step can be completed in $O(|C|r_{\max})$ time at most.
- The identities of the most similar clusters along with the corresponding similarity values are sent to one processor, which then computes the globally maximum similarity value. If the global maximum is greater than a threshold th_4, then the new document is considered to belong to the cluster that corresponds to this maximum. The total cost of this step is $O(p + gp + L)$ at most.

The processor that will store the new document is selected in a round-robin fashion, i.e., each time the new document is stored in the next higher numbered processor starting from the first. This ensures that all processors take an equal share of the new documents. Also, the new document should be inserted in the local inverted index and specifically the id of this document should be placed at the end of $r_{\max}$ posting lists. So, this operation takes $O(r_{\max})$ time at most. Note also that by always giving to a newly inserted document an id higher than those of documents already in the collection, we can ensure that each affected posting list remains sorted after each insertion.

On the rare occasion that the new document D_e cannot be considered to belong to any of the existing clusters (the "cosine-similarity" function is less than th_4), this document is viewed as a new cluster C_e by itself, whose centroid vector equals to the document vector ($c_e = d_e$). In subsequent document inclusions, each time a new document happens to belong to the recently formed centroid C_e then the centroid vector c_e is updated accordingly. Again, the distribution of the new documents to the processors can be done by using the round-robin technique.

It can be easily seen that the inclusion of the new document requires $O(|C|r_{\max})$ local computation and $O(gp + L)$ communication time for finding the similarity of the centroid vectors with the new document vector and $O(p)$ computation and $O(gp + L)$ communication time so that the centroid vector which is the most similar to the new document vector is determined.

Note that if the number of documents inclusions exceeds a specific fraction of the total number of documents n, then the off-line clustering phase is ran so that new document clusters are formed (re-clustering) and new centroid vectors are computed.

Regarding document deletions, the deleted document is removed from the local inverted index of the processor where it has been stored. This step involves a searching through $r_{\max}$ posting lists, each corresponding to a term of the deleted document. Searching in each list can be carried out in $O(\log \frac{n}{p})$ time at most by using binary searching. So, the total cost of removing a document from the local inverted index is $O(r_{\max} \log \frac{n}{p})$ at most.

The following theorem summarizes the cost complexities of the update phase of the VSM-based text retrieval algorithm.

Theorem 5 *The update phase of the VSM-based text retrieval algorithm requires* $O(|C|r_{\max})$ *computation time for document insertion and* $O(\max\{|C|, \log \frac{n}{p}\}r_{\max})$ *computation time for document deletion. The communication cost for both insertion and deletion is* $O(gp + L)$ *at most. The local memory per processor required for this phase is* $O(\frac{n}{p}r_{\max})$ *at most.*

4.5 External memory based implementation

Information retrieval systems routinely handle a huge volume of data. This is especially true for commercial search engines which should store billions of web pages in order to be competitive in comparison to other search engines. So, it is necessary to consider how our parallel algorithms for the off-line clustering and the on-line query processing phases can be adapted to the case that the main memory of each processor is not enough to hold all the necessary data for processing. A possible solution would be to ignore this limitation and instead to rely on the virtual memory mechanism built in all modern operating systems to handle the data transfer between main memory and disk. However, practice has shown that this approach may seriously degrade the whole system performance due to frequent disk accesses. So, many researchers have deal with the problem of minimizing the number of disk accesses for various fundamental algorithms such as sorting, data permutation, etc. by exploiting the various locality data access patterns that these algorithms exhibit and also by taking into account the block structure of the disks (see [31, 51] for a survey of these techniques).

For the adaptation of our algorithms to the external memory equipped processors, we use the external memory BSP (EM-BSP) model proposed by Dehne et al. [12, 13, 15, 16] which abstracts away all implementation details and extends the BSP model. Specifically, in this model, each BSP processor has also a secondary memory consisting of D disks. Each disk can read or write a block of B items in a single I/O operation and hence a total of DB items can be transferred to or from the secondary memory in one operation if disks are accessed in parallel. It is also assumed that there is an upper-bound in the size of local main memory per processor, i.e., $O(M)$ main memory per processor.

Dehne et al. in [15, 16] have presented a technique for simulating a v-processor BSP algorithm on fewer, p, EM-BSP processors with each processor having local main memory at least as large as that of a simulated BSP processor. Essentially, each real processor simulates v/p virtual processors and stores the local memory of each of these processors in its D local disks. Now, the communication step of each superstep of the BSP algorithm is simulated in v/p rounds. At each round, a processor sends the messages of one of its v/p virtual processors and the messages arriving in a real processor during this round are stored in the secondary memory of the processor. After the v/p communication rounds, each real processor simulates the message reception by its virtual processors and then simulates the computation performed by its virtual processors in the current simulated BSP superstep. Specifically, for each virtual processor, the real processor reads from the disks the local memory of the virtual processor and the messages arrived for this processor in the previous BSP superstep and then executes the computations carried out in the current superstep by this

specific virtual processor. Finally, it sends the messages sent by this virtual processor in the current simulated BSP superstep to the real processors holding the virtual processors, destinations of these messages.

An important detail of the algorithm is the placement of the messages and local memories in the disks in such a way that it ensures full parallel I/O operation of the D local disks in each real processor. In summary, Dehne et al have proved the following theorem:

Theorem 6 [15, 16] *The simulation of a v processor BSP algorithm with λ supersteps, computation time β, communication time $g\lambda(\frac{N}{v}) + \lambda L$, and local memory size μ on p processors with D local disks and $\Theta(M)$ local memory per processor requires $\frac{v}{p}\beta + \frac{v}{p}O(\lambda\mu)$ computation time, $g\lambda O(\frac{N}{p}) + \frac{v}{p}\lambda L$ communication time and $G\lambda\frac{v}{p}O(\frac{\mu}{BD})$ I/O time for $M \geq \mu + BD$, $p \leq v$, $N \geq v^2B + \frac{v^2(v-1)}{2}$.*

The notation needs some explanation. Parameter G shows how many times an I/O operation is more expensive in comparison to an arithmetic operation. Also, N is the total problem size. Further, it holds that $\mu = \Omega(\frac{N}{v})$.

For our simulation, we assume that $M = \mu$. We also ignore the quantity BD as its size is usually very small when compared to μ. Also, the number v of virtual processors are as many as are needed so that their memories can hold the input of size N cumulatively, e.g., $v = \frac{N}{M}$. Replacing v with $\frac{N}{M}$ in the inequality $N \geq v^2B + \frac{v^2(v-1)}{2}$, we can find the following sufficient condition for the total input size in order that the previous inequality hold:

$$8B^3 \leq N \leq (M)^{3/2}. \tag{3}$$

Although it seems a heavy requirement for the size of the main memory at first glance, it is not that serious. Indeed, a simple calculation shows that a main memory of 4 GB in size is sufficient for all practical purposes. Specifically, according to [33], the Google search engine indexes about 20.5 billions pages currently (May 2008). Also, in [35], the average number of terms met in a typical web page is estimated at about 474 terms per page. So, the size R of input weight matrix of our techniques will contain $20.5 \times 10^9 \times 474 = 9.71 \times 10^{12}$ elements. Now, from (3), a main memory holding about $M = 4.56 \times 10^8$ elements is enough for handling $N = R$ input elements. Note that since the input matrix is a sparse matrix and only the nonzero elements are stored, for each element we may need to store the index of the document, the index of the term as well as the corresponding weight. For 20.5 billion pages, we can use a document index of 35 bits, for weight values we can allocate 10 bits and for term index, we can use 27 bits capable of indexing about 1.34×10^8 terms. So, in total, 9 bytes are required for each of the N input elements. So, the capacity of the main memory can be estimated at about $4.56 \times 10^8 \times 9/2^{30} \leq 4$ GB. Apparently, this size for the main memory is not a heavy demand for today server machines.

Alternatively, we can use the randomized simulation method of the same authors in [12, 13] whose complexity bounds hold with high probability. Specifically, the computation, communication and I/O cost of the simulation is $(1 + o(1))\frac{v}{p}\beta$, $O(g\frac{N}{p} + \lambda\log(M/B) + \frac{v}{p}\lambda L)$, $O(G(v/p)(\mu\lambda/BD))$ respectively, provided that

$M = \Omega(\mu)$, $v = \Omega(pD\log(M/B))$. These conditions can be easily satisfied in practice even when v is equal to $\frac{N}{M}$.

Now, we will briefly discuss how to apply the first deterministic simulation technique to our algorithms. Specifically, we discuss the basic issues by showing the simulation of the first step of the off-line clustering phase. Then the reader can easily fill the details for the simulation of the steps of the algorithm. Note that similar techniques are also followed in the case of the second randomized simulation method.

At each step, it is important to check the total size of data input or output by this step in the worst case. Then we assume as many virtual processors as are needed to cover that data size. So, at step 1 of the off-line clustering phase, the size of the aggregate input and output of the sparse matrix multiplication is $O(R)$ and $O(Rd_{\text{sim}})$, respectively. Therefore, we should use $v = O(\frac{Rd_{\text{sim}}}{M})$ virtual processors in order to cover this memory requirement. This also implies that initially each virtual processor holds $O(\frac{M}{d_{\text{sim}}})$ vector elements. Now, based on Theorem 6, we can show that the execution of sparse matrix multiplication on p processors with $\Theta(M)$ local memory and D local disks per processor requires $O(\frac{Rn}{p} + \frac{R^2 d_{\text{sim}}^2}{Mp})$ computation time, and $O(G\frac{R^2 d_{\text{sim}}^2}{MBDp})$ I/O time. Note that the communication complexity is lower than that predicted by Theorem 6. This is so, because at each round, a single virtual processor broadcasts $O(\frac{M}{d_{\text{sim}}})$ vector elements to the remaining virtual processor and so each real processor actually sends and receives only $O(\frac{M}{d_{\text{sim}}})$ data elements and not $O(\frac{R}{p})$ as implied by Theorem 6. Since they are v broadcast rounds, the total computation communication complexity is $O(gR + \frac{Rd_{\text{sim}}}{M}L)$ and not $O(g\frac{R^2 d_{\text{sim}}^2}{Mp} + \frac{R^2 d_{\text{sim}}^2}{M^2 p}L)$ according to Theorem 6.

After the sparse matrix multiplication, the similarity matrix whose total size is $O(nd_{\text{sim}})$ is redistributed so that each processor holds $O(\frac{n}{p}d_{\text{sim}})$ elements after redistribution. Since, before the redistribution, each processor contains $O(\frac{Rd_{\text{sim}}}{p})$ matrix elements at most, we should use $O(\frac{Rd_{\text{sim}}}{M})$ virtual processors for this simulation. Note that some of the virtual processors may be empty at the beginning. Also during the simulation of this step, we should be aware of the actual position of virtual processors with regard to the real processors so that the redistribution can be done correctly on the real processors. More precisely, since each processor simulates $O(\frac{Rd_{\text{sim}}}{Mp})$ virtual processors and after the redistribution step each processor should hold $O(\frac{nd_{\text{sim}}}{p})$ elements, only $O(\frac{nd_{\text{sim}}}{Mp})$ out of $O(\frac{Rd_{\text{sim}}}{Mp})$ virtual processors in each processor should be "filled" with data in the end. This different redistribution requires only a slight change to the original algorithm.

Now, again by Theorem 6, we can easily see that the simulation of the redistribution step has $O(\frac{Rd_{\text{sim}}}{p})$, $O(g\frac{Rd_{\text{sim}}}{p} + \frac{Rd_{\text{sim}}}{Mp}L)$ and $O(G\frac{Rd_{\text{sim}}}{pBD})$ computation, communication and I/O cost, respectively.

The rest of the off-line clustering phase as well as the on-line query processing phase can be simulated following the methodology above. A possible concern about this simulation is whether the assumptions made during the discussion of the off-line clustering phase actually hold when there is a large number of processors as is the

case when considering virtual processors. For instance, both sorting algorithms [6, 23] used in off-line clustering phase assume that $\frac{N}{p} \geq p^{\epsilon}$ where N is the input size, p the numbers of processors running the algorithm and ϵ some constant greater than 1. For $\frac{N}{M}$ virtual processors, the previous inequality gives that $M \geq N^{\frac{\epsilon}{\epsilon+1}}$. However, from (3), the last inequality is implied if we set ϵ a value equal to 2.

Also, some substeps such as the inverted index creation in each processor can apparently be done directly without using the simulation technique. There are many optimized algorithms for the basic operations of this important data structure which is widely used in information retrieval systems [2, 54].

Finally, the analysis in Sect. 4.3 is still valid, since after application of the simulation technique, the basic complexities have again the same form, $O(\frac{A(n)}{p})$, where $A(n)$ is monotonically increasing function of input size n.

5 Conclusions and future work

A notably efficient parallel algorithm for the VSM-based text retrieval problem, has been presented and analyzed by means of the *CGM* and *BSP* cost models. The total cost is kept considerably low, exploiting the inherent sparsity of the documents' and query term vectors, even in the case of the off-line clustering phase, which is the most intensive task in corresponding VSM-based text retrieval approaches with clustering enhancements. Note also that the above algorithm can be easily extended in order to cover (maintaining similar efficiency) additional "document identifiers" (not only word stems) for each document in the collection (i.e., phrases, thesaurus classes, etc.).

The *BSP* and *CGM* models were specifically chosen as the basic means for our work mainly because they are realistic enough to represent a wide diversity of existing platforms, thus allowing extensive experimental studies. Moreover, with regard specially to *CGM*, its inherent tendency of the model to have large local computation steps allows the use of good existing sequential algorithms during the local computations. It is also worth mentioning that all the cost complexities mentioned in the paper are worst-case complexities with low constants hidden in the O-notation. So, it is fair to assume that the average performance of our algorithms on real machines will be at least no worse or even better than that predicted by our analysis.

We also discussed the external memory implementation of our algorithms and showed that our techniques can be easily simulated on the EM-BSP model, an extension to the BSP model for modeling external memory related performance issues.

Finally, as a future work, we plan to investigate the parallel implementation of hierarchical clustering methods and the use of our techniques for supporting searching as well as browsing of document collection. Another interesting direction for future research is the study of our parallel text-retrieval algorithms on more recent parallel models that have been proposed as extensions to the classical *BSP* model such as the Decomposable *BSP* (*D-BSP*) model [3] which takes into account the fact that in a parallel architecture some processors may be nearer than others and the Heterogeneous *BSP* (*H-BSP*) model [52, 53] as well as the Dynamic *BSP* model [41] which have been proposed for modeling heterogeneous clusters and computational grids.

Acknowledgements This work is cofunded by 75% from EU and 25% from the Greek Government under the framework of the Education and Initial Vocational Training II, program "Archimedes."

References

1. Aslam J, Pelekhov E, Rus D (2004) The star clustering algorithm for static and dynamic information organization. J Graph Algorithms Appl 8:95–129
2. Baeza-Yates R, Ribeiro-Neto B (1999) Modern information retrieval. Addison-Wesley, New York
3. Bilardi G, Fantozzi C, Pietracaprina A, Pucci G (2001) On the effectiveness of D-BSP as a bridging model of parallel computation. In: Proc of the int conf on comput sci (ICCS 2001), pp 579–588
4. Can F (1993) Incremental clustering for dynamic information processing. ACM Trans Inf Process Syst 11:143–164
5. Can F, Altingovde I, Demir E (2004) Efficiency and effectiveness of query processing in cluster-based retrieval. Inf Syst 29:697–717
6. Chan A, Dehne F (1999) A note on coarse grained parallel integer sorting. Parallel Process Lett 9(4):533–538
7. Chaudhri B (1994) Dynamic clustering for time incremental data. Pattern Recognit Lett 13:27–34
8. Chung S, Kwon H, Ryu K, Chung Y, Jang H, Choi C (2001) Information retrieval on an SCI-based PC cluster. J Supercomput 19:251–265
9. Cringean J, England R, Manson G, Willett P (1990) Parallel text searching in serial files using a processor farm. In: Proc of ACM SIGIR, pp 429–452
10. Dehne F, Fabri A, Rau-Chaplin A (1993) Scalable parallel geometric algorithms for coarse grained multicomputer. In: Proc of the ACM 9th symp on comput geom, pp 298–307
11. Dehne F, Fabri A, Kenyon C (1994) Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In: Proc of the IEEE symp on parallel and distrib process, pp 586–593
12. Dehne F, Dittrich W, Hutchinson D (1997) Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In: Proc of the 9th annual ACM symp on parallel algorithms and archit (SPAA), pp 106–115
13. Dehne F, Dittrich W, Hutchinson D (2003) Efficient external memory algorithms by simulating coarse-grained parallel algorithms. Algorithmica 36:97–122
14. Dehne F, Deng X, Dymond P, Fabri A, Kokhar A (1995) A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In: Proc of the ACM symp on parallel algorithms and archit, pp 27–33
15. Dehne F, Hutchinson D, Maheshwari A, Dittrich W (1999) Reducing I/O complexity by simulating coarse grained parallel algorithms. In: Proc of the 13th Int and 10th symp on parallel and distrib process (IPPS/SPDP), pp 14–20
16. Dehne F, Dittrich W, Hutchinson D, Maheshwari A (2002) Bulk synchronous parallel algorithms for the external memory model. Theory of Comput Syst 35:567–597
17. Dehne F, Ferreira A, Caceres E, Wong S, Roncato A (2002) Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. Algorithmica 33:183–200
18. Dobrynin V, Patterson D, Galushka M, Rooney N (2005) SOPHIA: an interactive cluster-based retrieval system for the OHSUMED collection. IEEE Trans Inf Technol Biomed 9(2):256–265
19. Efraimidis P, Glymidakis C, Mamalis B, Spirakis P, Tampakas B (1995) Parallel text retrieval on a high performance supercomputer using the vector space model. In: Proc of ACM SIGIR, pp 58–66
20. Gavalas D, Konstantopoulos C, Mamalis B, Pantziou G (2005) Efficient BSP/CGM algorithms for text retrieval. In: Proc of the int conf on parallel and distrib comput syst (IASTED PDCS), pp 301–306
21. Gerbessiotis A, Siniolakis C (1996) Primitive operations on the BSP model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University
22. Gerbessiotis A, Valiant L (1994) Direct bulk synchronous parallel algorithms. J Parallel Distrib Comput 22:251–267
23. Goodrich M (1999) Communication-efficient parallel sorting. SIAM J Comput 29(2):416–432
24. Hawking D (1996) Document retrieval performance on parallel systems. In: Proc of the int conf on parallel and distrib process tech and appl (PDPTA '96), pp 1354–1365
25. Ishikawa Y, Chen Y, Kitagawa H (2001) An on-line document clustering method based on forgetting factors. In: Proc of the 5th Eur conf on res and adv technol for digit libr, pp 325–339

26. Jain A, Murty M, Flynn P (1999) Data clustering: a review. ACM Comput Surv 31(3):264–323
27. Juurlink B, Wijshoff H (1996) Communication primitives for BSP computers. Inf Process Lett 58:303–310
28. Kanellakis P, Shvartsman A (1997) Fault-tolerant parallel computation. Kluwer Academic, Norwell
29. Kang J, Ahn H, Jung S, Ryu K, Kwon H, Chung S (2004) Improving load balance and fault tolerance for PC cluster-based parallel information retrieval. In: Proc of the 5th int conf on parallel process and appl math (PPAM 2003), pp 682–687
30. Karger D, Stein C, Wein J (1997) Scheduling algorithms. In: Atallah M (ed) Algorithms and theory of computation handbook. CRC Press, New York, pp 35-1–35-33
31. Katriel I, Meyer U (2003) Elementary graph algorithms in external memory. In: Meyer U, Sanders P, Sibeyn J (eds) Algorithms for memory hierarchies. LNCS, vol 2625. Springer, Berlin, pp 62–84
32. Kleinberg J, Tardos E (2006) Algorithm design. Addison-Wesley, Boston
33. Kunder M (2008) The size of the World Wide Web: estimated size of Google's index. In: The size of the World Wide Web. Available via http://www.worldwidewebsize.com. Accessed 2 June 2008
34. Kurland O, Lee L (2004) Corpus structure, language models, and ad hoc information retrieval. In: Proc of SIGIR'04, pp 194–201
35. Levering R, Cutler M (2006) The portrait of a common HTML Web page. In: Proc of the 2006 ACM symp on doc eng, pp 198–204
36. Liu X, Croft W (2004) Cluster-based retrieval using language models. In: Proc of SIGIR'04, pp 186–193
37. Lu Z, McKinley K (2000) Partial replica selection based on relevance for information retrieval. In: Proc of the 22nd Int ACM SIGIR conf on res and develop in inf retr, pp 97–104
38. MacFarlane A, Robertson S, McCann J (2000) Pliers at TRECS. In: Proc of eighth text retr conf (TREC-8), pp 241–252
39. Mamalis B, Spirakis P, Tampakas B (2003) Parallel processing of multiple text queries on hypercube interconnection networks. Int J Comput Appl 10(1):115–132
40. Manning C, Raghavan P, Schutze H (2008) An introduction to information retrieval, preliminary draft. Cambridge University Press. Available via http://www.informationretrieval.org. Accessed 1 June 2008
41. Martin J, Tiskin A (2004) Dynamic BSP: towards a flexible approach to parallel computing over the grid. In: East I, Martin J, Welch P, Duce D, Green M (eds) Communicating process architectures. IOS Press, Amsterdam, pp 219–226
42. McColl W (1994) Scalable parallel computing: a grand unified theory and its practical development. In: Proc of IFIP world congr, pp 539–546
43. Rasmussen E (1992) Clustering algorithms. In: Frakes W, Baeza-Yates R (eds) Information retrieval: data structures and algorithms. Prentice-Hall, Englewood Cliffs, pp 419–442
44. Rungsawang A, Laohakanniyom A, Lertprasertkune M (2001) Low-cost parallel text retrieval using PC-cluster. In: Proc of Eur PVM/MPI, pp 419–426
45. Salton G, McGill M (1983) An introduction to modern information retrieval. McGraw-Hill, New York
46. Salton G, Wong A, Yang C (1975) A vector space model for automatic indexing. Commun ACM 18(11):613–620
47. Stanfill C (1989) Partitioned posting files: a parallel inverted file structure for information retrieval. In: Proc of ACM SIGIR, pp 413–428
48. Valiant L (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111
49. Valiant L (1991) General purpose parallel architectures. In: Leeuwen J (ed) Handbook of theoretical computer science, vol A. MIT Press, Cambridge, pp 943–971
50. van Rijsbergen C (1971) Information retrieval, 2nd edn. Butterworth-Heinemann, Newton
51. Vitter J (2001) External memory algorithms and data structures: dealing with massive data. ACM Comput Surv 33(2):209–271
52. Williams T (2000) A general-purpose model for heterogeneous computation. PhD Thesis, University of Central Florida, Orlando
53. Williams T, Parsons R (2000) The heterogeneous bulk synchronous parallel model. In: Proc of the IPDPS 2000 workshops on parallel and distrib process, pp 102–108
54. Witten I, Moffat A, Bell T (1999) Managing gigabytes: compressing and indexing documents and images, 2d edn. Morgan Kaufmann, San Francisco

Charalampos Konstantopoulos received his Diploma in computer engineering from the Department of Computer Engineering and Informatics at University of Patras, Greece (1993). He also received his Ph.D. degree in Computer Science from the same department in 2000. Currently, he is a Lecturer at the Department of Informatics, University of Piraeus, Greece. His research interests include parallel and distributed algorithms/architectures, mobile computing and multimedia systems.

Basilis Mamalis was born in 1970 and obtained his diploma (in Computer Engineering and Informatics) as well as his Ph.D. from the University of Patras (Greece), in 1993 and 1997 respectively. He has worked for almost ten years (in several projects and research programs) in Computer Technology Institute (C.T.I.) of Patras as well as an academic teacher in the Hellenic Open University. During the last three years he teaches as an Assistant Professor at the Technological Educational Institute (T.E.I.) of Athens, Dept. of Informatics. His research interests include information retrieval, document and knowledge management, parallel algorithms and architectures, distributed systems and databases, operating systems, mobile computing and sensor networks. He has about thirty publications in international journals and conferences in the above mentioned research areas and he was also a referee in several international conferences.

Grammati Pantziou received the Diploma in Mathematics and her Ph.D. Degree in Computer Science from the University of Patras, Greece, in 1984 and 1991, respectively. She was a Post-Doctoral Research and Teaching Fellow at the University of Patras (1991–1992), a Research Assistant Professor at Dartmouth College, Hanover, NH, USA (1992–1994), an Assistant Professor at the University of Central Florida, Orlando, FL, USA (1994–1995) and a Senior Researcher at the Computer Technology Institute, Patras (1995–1998). Since September 1998, she is a Professor at the Department of Informatics of the Technological Educational Institution of Athens, Greece. Her current research interests are in the areas of parallel computing, design and analysis of algorithms, distributed and mobile computing and multimedia systems.

Damianos Gavalas received his B.Sc. degree in Informatics from University of Athens, Greece, in 1995 and his M.Sc. and Ph.D. degree in electronic engineering from University of Essex, UK, in 1997 and 2001, respectively. He is currently Assistant Professor in the Department of Cultural Technology and Communication, University of the Aegean, Greece. His research interests include distributed computing, mobile code, network and systems management, network design, e-commerce, m-commerce, mobile ad-hoc and wireless sensor networks.