Advance Access publication on 5 January 2012
doi:10.1093/comjnl/bxr133

A Parallel Framework for In-Memory Construction of Term-Partitioned Inverted Indexes

Tayfun Kucukyilmaz, Ata Turk and Cevdet Aykanat*

Computer Engineering Department, Bilkent University, 06800 Ankara, Turkey
**Corresponding author: aykanat@cs.bilkent.edu.tr*

With the advances in cloud computing and huge RAMs provided by 64-bit architectures, it is possible to tackle large problems using memory-based solutions. Construction of term-based, partitioned, parallel inverted indexes is a communication intensive task and suitable for memory-based modeling. In this paper, we provide an efficient parallel framework for in-memory construction of term-based partitioned, inverted indexes. We show that, by utilizing an efficient bucketing scheme, we can eliminate the need for the generation of a global vocabulary. We propose and investigate assignment schemes that can reduce the communication overheads while minimizing the storage and final query processing imbalance. We also present a study on how communication among processors should be carried out with limited communication memory in order to reduce the total inversion time. We present several different communication-memory organizations and discuss their advantages and shortcomings. The conducted experiments indicate promising results.

Keywords: index inversion; term-based partitioning; parallel inversion; memory-based inversion

Received 30 July 2011; revised 29 October 2011
Handling editor: Yannis Manolopoulos

1. INTRODUCTION

The evolution of communication technologies in recent years gave rise to a rapid increase in the amount of textual digital information and the demand to search over this type of information. One of the largest industries of our era, the searching industry, has flourished around these demands.

Inverted indexes, due to their superior performance in answering phrase queries [1], are the most commonly used data structures in Web search systems. An inverted index consists of two parts: a *vocabulary* and *inverted lists*. The vocabulary contains the collection of distinct *terms*, which are composed of character strings (*words*) that occur in the documents of the collection. For each term in the vocabulary, there is an associated *inverted list*, or *posting list*. The inverted list for a term is a list of *postings*, where a posting contains an identifier for a document that contains that term. Depending on the granularity of information, the frequency and the exact term positions may also be stored in the postings.

Inverted index data structure is quite simple, yet Web-scale generation of a global inverted index is very costly due to the size, distributed nature and growth/change rate of the Web data [2]. Fast and efficient index construction schemes are required to provide fresh and up-to-date information to users. Furthermore, since the data to be indexed is crawled and stored by distributed or parallel systems (due to performance and scalability reasons), parallel index construction techniques are essential.

There are two major partitioning schemes used in distributing the inverted index on parallel systems: document-based and term-based partitioning. In document-based partitioning, the documents are assigned to index servers and all the postings related with the assigned documents are stored in a particular index server. In term-based partitioning, each term in the vocabulary and the related inverted lists are assigned to an index server.

Almost all of the major search engines use document-based partitioning due to the ease in parallel index construction of document-based, partitioned, inverted indexes. Term-based partitioning, on the other hand, has advantages that can be exploited for better query processing [3]. In this study, we propose an efficient parallel index construction framework that can be used for generating term-based partitioned, inverted indexes starting from a document-based partitioned collection most possibly generated via a parallel crawling of Web documents.

1.1. Related work

Early studies on index construction are focused around disk-based algorithms designed for sequential systems [4–6]. In [4], authors present a method that traverses the disk-based document collection twice; once for generating a term-based partition to divide the work into loads, and once for inverting the dataset iteratively for each pass defined in the previous pass. The emphasis is on using as little memory as possible. In [5], authors use a multi-way, in-place, external merge algorithm for inverted index construction with less primary memory. In [6], authors propose an in-memory index construction method for disk-based inverted indexes where the document set is divided into batches that are inverted in memory and then merged and written into disk. In their work, authors facilitate the use of compression in order to achieve a more effective inversion.

More recent works on sequential systems are mainly focused on on-line incremental updates over disk-based, inverted indexes [7, 8]. In [7], the authors propose a hybrid indexing technique. The proposed method merges small posting lists with the already existing index, while using posting list reallocation for large posting lists. The authors also propose two in-place merge techniques for updating long posting lists. In [8], the authors evaluate two index maintenance strategies and propose alternatives for improving these strategies. These improvements are based on over-allocation of posting lists and keeping incremental updates within vocabulary before index remerging.

The following studies on index construction [3, 9–14] extend disk-based techniques for parallel systems. In [9], a document-based allocation scheme for inverted indexes is presented. The authors emphasize both storage balance and inter-processor communication times and try to minimize both using genetic algorithms. In [10], the authors evaluate the effects of term- and document-based partitioning methods on a shared-everything architecture. They use query statistics to balance the required I/O times among processors on a disk-based architecture.

In [11, 12], the authors present a disk-based, parallel index construction algorithm, where initially the local document collections are inverted by all processors in parallel. The processors generate a global vocabulary on a host processor and the host processor divides the document collection among all processors in lexicographic order assuming global knowledge over the document collection. The authors also analyze the merging phase of the inverted lists in [12], presenting three algorithms. In their work, the authors mainly focus on the parallel generation of the distributed index and the communication costs are not taken into consideration.

In [13], the author describes an index inversion framework for distributed information retrieval systems. Although the method presented in [13] achieves storage balance among processors, it does not consider minimizing the communication loads of the processors. In [13], it is also assumed that it is possible for the inverted indexes to be incrementally updated over time, and specialized data structures for minimizing the index update times are proposed. The cost of the inversion process is also emphasized, and four different index inversion methods are presented. In [14], the authors again start from a document partitioned collection and use a software-pipelined architecture to invert document collections. The collection is divided into runs, and for each run, documents are parsed, inverted, sorted and flushed into disk in a pipelined fashion. In [3], the authors propose a load balancing strategy in a term-partitioned inverted index on a pipelined query processing architecture [15]. In [3], both replication of inverted lists and a query statistics-based assignment scheme is presented, yielding up to 30% net query throughput improvement.

1.2. Motivation and contributions

We would like to repeat a catchy phrase often credited to Jim Gray: 'Memory is the new disk, disk is the new tape'. With the advent of 64-bit architectures, huge memory spaces are available to single machines and even very large inverted indexes can fit into the total distributed memory of a cluster of such systems, enabling memory-based index construction. Furthermore, cloud computing systems such as Amazon EC2 are commercially available today. They offer leasing of virtual machines without owning and maintenance costs and thus ease the utilization and management of large cluster of servers. Thus, we believe that the benefits of parallel index construction is not limited to dividing and distributing the computational task to different processors. The current advances in network technologies, cloud computing and the high availability of low-cost memory provides an excellent medium for memory resident solutions for parallel index construction.

In this work, we extend our previously proposed in-memory parallel inverted index construction scheme [16] and compare the effects of different communication-memory organization schemes to the parallel inversion time. In our framework, we propose to avoid the communication costs associated with global vocabulary construction with a term-to-bucket assignment schema. This schema prevents term information to be sent to a host, where a reasonable term-to-processor assignment would be computed using the term distribution among processors, thus avoiding a possible bottleneck of communication. Furthermore, term-to-bucket partitioning allows the framework to completely avoid creating a global vocabulary, eliminating the need of a further communication phase.

We also investigate several assignment heuristics for improving the final storage balance, the final query processing loads and the communication costs of inverted index construction. Here, storage balance is important since we are trying to build a memory-based inverted index. Query processing load balance is important since the reason for building the inverted index is for faster query processing and this can be done better if the loads of the processors are balanced. Finally, the communication cost is important since it affects the running time of parallel inversion.

Furthermore, we investigate the effects of various communication-memory organization schemes. Since parallel inversion is a communication-bound process, we observe that the utilization of the communication memory and the network has significant effects on the overall inversion time. Our findings indicate that, dividing the communication memory into $2 \times K$ buffers, where K of which are used for sending messages and the remaining K are used for receiving messages, yields the best performance. This is due to the fact that this communication-memory organization scheme maximizes the communication/computation overlap.

Finally, we test the performance of the proposed schemes by performing both simulations and actual parallel inversion of a realistic Web dataset and report our observations. Our contributions in this work are prior to optimizations such as compression [17]. However, it is possible to apply data compression to the proposed model, making it possible to work with even larger data collections.

The organization of this paper is as follows: in Section 2, we introduce the memory-resident distributed index inversion problem and describe our framework. In Section 3, we provide our overall parallel inversion scheme. In Section 4, we describe the investigated assignment schemes in detail. In Section 5, we present several memory organization schemes in order to reduce the communication time and discuss their advantages and disadvantages. We provide our experiments, their analysis and extensive discussions on the results of our experiments in Section 6. Finally, in Section 7, we conclude and discuss some future work.

2. FRAMEWORK

Most of the largest text document collections that are actively in use today are Web-based. These repositories are mainly created and used by Web search engines. An important consideration in the design of parallel index construction systems should be their applicability to such real-life data collections. In this work, our efforts are based on presenting an efficient and scalable index construction framework specifically designed for Web-based document collections.

Parallel search engines collect Web pages to be indexed via distributed Web Crawlers [18]. In general, at the end of a crawling session, a document-based partition of the whole document collection is obtained, where each part is stored in a physically separate repository [18]. The state-of-the-art approach to distributing the crawling and storage tasks uses a site-hash-based assignment, where the site names of pages are hashed and documents are assigned to repositories according to those hash values [19–21].

The framework presented in this study has three assumptions on the initial data distribution. First, the initial document collection is assumed to be distributed among the processors of a parallel system; that is, each processor is assumed to have a portion of the crawled Web documents and maintain information about only its own local dataset. Thus, in this work, no processor contains a global view of the document collection. Secondly, each processor is assumed to contain a disjoint set of documents. This means that the overall system contains no replica of any document. Thirdly, the Web pages are assumed to be distributed among these processors using a site-based hashing; that is, all pages from a site are assigned to a single processor, and hence each site is assumed to be an atomic storage task. Consequently, the initial storage loads of the processors are not necessarily perfectly balanced. These three assumptions are in concordance with the output format of general purpose crawling systems.

In this framework, the objective of parallel index construction is to generate a final term-partitioned, parallel inverted index from a document-partitioned collection stored on a distributed shared-nothing architecture. The final term-partitioned inverted index will also be stored in a distributed fashion in order to allow both inter- and intra-query parallelism on query processing. In this context, our approach has similarities with parallel matrix transpose operations.

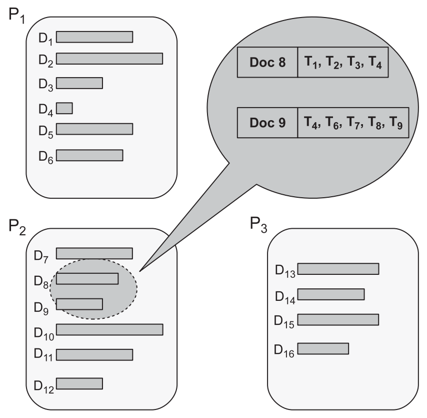
3. PARALLEL INVERSION

Our inversion scheme starts with a document-based, initial partition. Such an initial, document-based partition is depicted in Fig. 1a. Our overall parallel inversion scheme has the following phases:

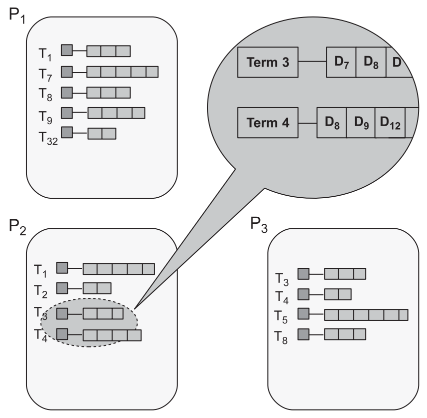
(i) *Local inverted index construction:* Each processor generates a local inverted index from its local document collection. This process is illustrated in Fig. 1b. Note that inverted lists for some terms can appear in multiple processors.

(ii) *TermBucket-to-processor assignment:* Each processor uses hashing to find a deterministic assignment of terms into a predetermined number buckets. Buckets are used to randomly group inverted lists so that the communication costs in the termBucket-to-processor assignment phase is reduced. All processors communicate the sizes of their term buckets to the host processor. The host processor generates a termBucket-to-processor mapping under the constraint that in the final assignment, the storage and query processing load balance is achieved and communication cost is minimized. This process is illustrated in Fig. 1c. Note that many buckets exist in multiple processors due to the initial document partitioning.

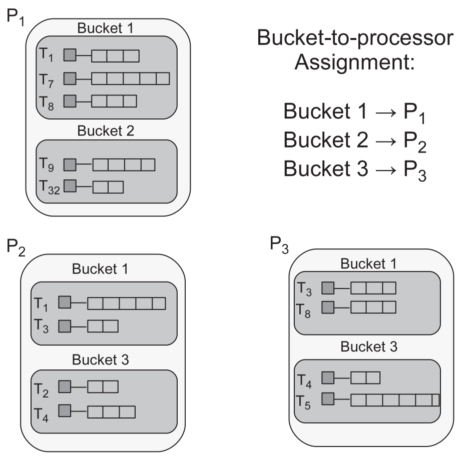
(iii) *Inverted list exchange-and-merge:* The processors communicate appropriate parts of their local inverted indexes in an all-to-all fashion. This process is illustrated in Fig. 1d. The remaining local inverted index portions are merged with the received portions and final inverted index is generated. The final term-partitioned inverted index of the initial document-partitioned index in Fig. 1b can be seen in Fig. 1e.


P_1
D_1 D_2 D_3 D_4 D_5 D_6
Doc 8 | T_1, T_2, T_3, T_4
Doc 9 | T_4, T_6, T_7, T_8, T_9
P_2
D_7 D_8 D_9 D_{10} D_{11} D_{12}
P_3
D_{13} D_{14} D_{15} D_{16}

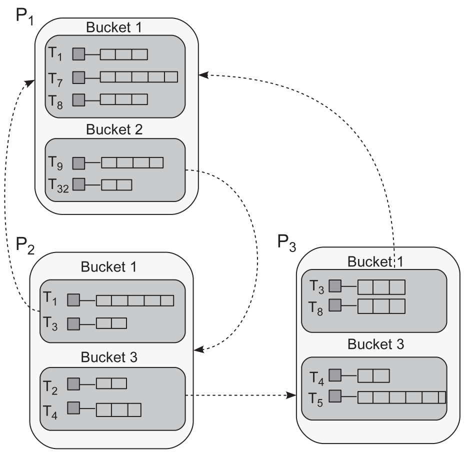
(a) Initial document partition.


P_1
T_1 T_7 T_8 T_9 T_{32}
Term 3 | D_7 | D_8 | D
Term 4 | D_8 | D_9 | D_{12}
P_2
T_1 T_2 T_3 T_4
P_3
T_3 T_4 T_5 T_8

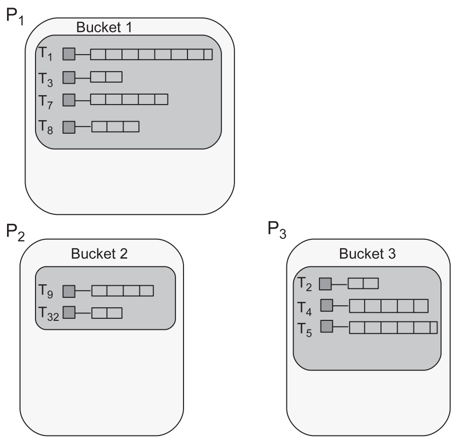
(b) Local inverted index construction.



(c) TermBucket-to-processor assignment.



(d) Inverted list exchange-and-merge.



(e) Final term-partitioned index

FIGURE 1. Phases of the index inversion process. (**a**) Initial document partition, (**b**) local inverted index construction, (**c**) TermBucket-to-processor assignment, (**d**) inverted list exchange-and-merge and (**e**) final term-partitioned index.

3.1. Local inverted index construction

In the local inverted index construction phase, each processor generates a local vocabulary and local inverted lists from its local document collection. Since each processor only contain a unique subset of documents, this operation can be achieved concurrently without any communication. In this phase, the local vocabularies and inverted list sizes are determined and each term is given a unique identifier.

3.2. TermBucket-to-processor assignment

After the local inversion phase, processors contain a document-based, partitioned inverted index. In this partition, processors contain different portions of inverted lists for each term. In order to create a term-based, partitioned inverted index, each inverted list, in its full form, should be accumulated in one of the processors. To this end, each term in the global vocabulary should be assigned to a particular processor.

This term-to-processor assignment depicts an inverted index partitioning problem. A suitable index partitioning can be defined by many different criteria. In this work, we set the following quality metrics for a 'good' term-to-processor assignment:

QM1: Balancing the 'expected' query processing loads of processors.
QM2: Balancing the storage loads of processors.
QM3: Reducing the communication overhead during the inversion process through minimizing:

(a) Total communication volume.
(b) Communication load of the maximally loaded processor.

The final query processing loads of processors indicate the amount of processing that a processor is expected to perform once the inversion is finished and the query processing begins. We can estimate this load utilizing previous query logs.

The storage balance of processors guarantees an even distribution of the final inverted index allowing larger indexes to fit in the same set of processors.

Since inversion is a communication-bound process, the minimization of the communication overhead ensures that the inverted list exchange phase of the parallel inversion process takes less time. In this work, minimization of the communication overhead is modeled as the minimization of the total communication volume while maintaining the balance on the communication loads of the processors. These are the two commonly used quality metrics that determine the communication performance of a task-to-processor assignment when the message latency overhead remains negligible compared with the message volume overhead [22, 23], which is the case for parallel index inversion.

To optimize the above-mentioned metrics, we investigate existing assignment schemes, comment on possible enhancements over these schemes and propose a novel assignment scheme that performs better than its counterparts. Our discussions about bucket-to-processor assignment schemes are explained in detail in Section 4.

For the purpose of finding a suitable term-to-processor assignment, the previous works in the literature either assume the existence of a global vocabulary or generate a global vocabulary from the local vocabularies. The global vocabulary can be created by sending each term string, in its word form, to a host processor, where they are assigned global term-ids, and these global term-ids are broadcasted to all processors. However, in such a scheme, a particular term would be sent to the host machine by all processors if all processors contain that specific term. Our observations indicate that the cost of such an expensive communication stage is proportional to the cost of inverted list exchange phase. Furthermore, since the host processor receives all the communication, it constitutes a serious bottleneck.

In this work, we propose a novel and intelligent scheme that enables us to avoid global vocabulary construction cost. We propose to group terms into buckets prior to the term-to-processor assignment. Using string hashing functions, each word in a local vocabulary is assigned to a bucket. Afterward, only the bucket size information is sent to the host processor. The host processor computes a termBucket-to-processor assignment, which induces a term-to-processor assignment, and broadcasts this information to the processors. The effect of the bucket processing order on the quality of the assignment is not investigated in this work and the same random bucket processing order is used in evaluating the assignment schemes. We should also note here that it is not necessary to build a global index at the host processor ever. It suffices for the host processor to store only a bucket-to-processor assignment array. Whenever the host processor receives a query term, all it has to do is to compute the hash of the term, find the bucket for that term and forward the term to the owner processor of the bucket.

3.3. Inverted list exchange-and-merge

At the end of the termBucket-to-processor assignment phase, all bucket-to-processor assignments are broadcast to the processors by the host processor, so that each processor is aware of the bucket-to-processor assignments. In order to create a term-partitioned inverted index, the document-based partitioned, local inverted list portions should be communicated between processors in such a way that the whole posting list of each term resides in one of the processors. To this end, all processors should exchange their inverted list portions in an all-to-all fashion. However, utilizing termBuckets instead of terms for assignment dictates a major change (and an additional cost) in the inverted list exchange-and-merge phase.

Since termBucket-to-Processor assignment prevents the need of creating a global vocabulary, when a processor receives a posting list portion of a term from another processor, it also requires additional information to identify the posting list it receives. To this end, upon sending the posting list portions, the processors should also send the associated term, in its word form, to the receiving processor. Due to this, the all-to-all inverted list exchange communication becomes slightly more costly. However, since the processor-to-host bottleneck due to global vocabulary construction is already avoided, the performance degradation in all-to-all inverted list exchange communication is more than compensated. Furthermore, this vocabulary exchange is distributed among all processors evenly, further reducing its overhead.

The inverted list exchange between processors is achieved in two steps. First, terms, in their word form, and their posting sizes are communicated. This is done by an all-to-all personalized communication phase, where each processor receives a single message from each other processor. At the end of this step, all processors obtain their final local vocabularies and can reserve space for their final local inverted index structures. Secondly, inverted list portions are exchanged in bucket id order, and within the buckets in alphabetical order. This step is again performed as an all-to-all personalized communication. However, since this step consumes significant amount of time, the inverted list portions are sent via multiple messages. Memory organization and communication scheme used in this phase is explained in detail in Section 5. At the end of inverted list exchange, the remaining inverted lists and obtained inverted lists for each term are merged and written into their reserved spaces in the memory.

4. TERM-TO-PROCESSOR ASSIGNMENT SCHEMES

In this section, we try to solve the termBucket-to-processor assignment problem with the objectives of minimizing the communication overhead during the inversion and maintaining a balance on the query processing and storage loads of processors after the inversion. We present adaptations of two previously proposed assignment algorithms [24] to the problem at hand, discuss the shortcomings of these algorithms and propose a novel assignment algorithm that provides superior parallel performance.

In the forthcoming discussions we use the following notations: The vocabulary of terms is indicated by $\mathcal{T}$. Due to the initial site-hash-based crawling assumption, the posting list of each term $t_j \in \mathcal{T}$ is distributed among the K processors. In this distribution, $w_k(t_j)$ denotes the size of the posting list portion of term t_j that resides in processor p_k at the beginning of the inversion, whereas $w_{\text{tot}}(t_j) = \sum_{k=1}^{K} w_k(t_j)$ denotes the total posting list size of term t_j.

We assume that prior to bucket-to-processor assignment, each processor has built its local inverted index $\mathcal{I}_k$ and partitioned the vocabulary $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ containing n terms into a predetermined number m of buckets. The number of buckets m is selected such that $m \ll n$ and $m \gg K$. Let

$$\mathcal{B} = \Pi(\mathcal{T}) = \{\mathcal{T}_1 = b_1, \mathcal{T}_2 = b_2, \ldots, \mathcal{T}_m = b_m\}. \quad (1)$$

denote a random term (RT)-to-bucket partition, where $\mathcal{T}_i$ denotes the set of terms that are assigned to bucket b_i. In this partition, $w_{\text{tot}}(b_i)$ denotes the total size of the posting lists of terms that belong to b_i and $w_k(b_i)$ denotes the total size of the posting list portions of terms that belong to b_i and that reside in processor p_k at the beginning of the inversion.

We also assume that we are given a query set $\mathcal{Q}$ where each query $q \in \mathcal{Q}$ is a subset of $\mathcal{T}$, i.e. $q \subset \mathcal{T}$. The number of queries that a term t_j is requested by is denoted with $f(t_j)$.

In an m-bucket and K-processor system, the bucket-to-processor assignment can be represented via a K-way partition

$$\Pi(\mathcal{B}) = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_k\}. \quad (2)$$

of the buckets among the processors. The quality of a bucket-to-processor assignment $\Pi(\mathcal{B})$ is measured in terms of three metrics: The query processing load balance (QM1), storage load balance (QM2) and the communication cost (QM3). The query processing load $\text{QP}(p_k)$ of a processor p_k induced by the assignment $\Pi(B)$ is defined as follows:

$$\text{QP}(p_k) = \sum_{b_i \in \mathcal{B}_k} \sum_{t_j \in b_i} w_{\text{tot}}(t_j) \times f(t_j). \quad (3)$$

The storage load $S(p_k)$ of a processor p_k induced by the assignment $\Pi(B)$ is defined as follows:

$$S(p_k) = \sum_{b_i \in \mathcal{B}_k} \sum_{t_j \in b_i} w_{\text{tot}}(t_j). \quad (4)$$

The communication cost of a processor p_k induced by the assignment $\Pi(B)$ has two components. Each processor must receive all portions of the buckets assigned to it from other processors. Thus total reception cost/volume of a processor p_k is

$$\text{Recv}(p_k) = \sum_{b_i \in \mathcal{B}_k} \sum_{t_j \in b_i} (w_{\text{tot}}(t_j) - w_k(t_j)). \quad (5)$$

Each processor must also send all postings that are not assigned to it to some other processor. The total transmission cost of p_k is represented by $\text{Send}(p_k)$ and is defined as

$$\text{Send}(p_k) = \sum_{b_i \notin \mathcal{B}_k} \sum_{t_j \in b_i} w_k(t_j). \quad (6)$$

The total communication cost of a processor is defined as

$$\text{Comm}(p_k) = \text{Send}(p_k) + \text{Recv}(p_k). \quad (7)$$

4.1. Minimum communication assignment

Minimum communication assignment (MCA) algorithm minimizes the total communication volume while ignoring storage and communication balancing [24]. The MCA scheme is based on the following simple observation. If a termBucket is assigned to the processor that contains the largest portion of the inverted lists of the terms belonging to that bucket, the total message volume incurred due to this assignment will be minimized. Thus, if we assign each termBucket $b_i \in \mathcal{B}$ to the processor p_k that has the largest $w_k(b_i)$ value, the total volume of communication for this term will be minimized. By assigning all terms using the above criteria, an assignment with global minimum total communication volume can be achieved.

4.2. Balanced-load MCA

The balanced-load (BLMCA) scheme is an effort to incorporate storage balancing to MCA [24]. In this scheme, termBuckets are iteratively assigned to processors. In BLMCA, for each termBucket, first the target processor that will incur the minimal total communication is determined using the criteria in the MCA scheme. If assignment of the particular termBucket to that processor does not make the storage loads of the processors more skewed (does not increase the maximum storage load of all processors) at that iteration, the assignment proceeds as in the MCA scheme. Otherwise, the termBucket is assigned to the minimally loaded processor.

4.3. Energy-based assignment

In BLMCA, two separate cost metrics are evaluated: The storage load balance and total communication cost. However, at each iteration, only one of these metrics is chosen to be optimized. Furthermore, both MCA and BLMCA model the communication cost as the total communication volume and disregard the maximum communication volume of a single processor. In order to minimize the maximum communication cost of a processor, we should consider both the reception cost of the assigned processor and the transmission costs of all other processors.

In the energy-based assignment (EA) scheme, we propose a model that prioritizes reducing the maximum communication cost of processors as well as maintaining storage and query processing load balance. To this end we define the energy E of an assignment $\Pi(B)$. This energy definition is based on the storage loads, query processing loads and communication costs of processors. Recall that $\text{Comm}(p_k)$ of a processor incorporates both reception and transmission costs of processor p_k. We define two different energy functions for a given termBucket-to-processor assignment $\Pi(B)$:

$$E^1(\Pi(B)) = \text{Max}\left\{ \underset{1\le k\le K}{\text{Max}}\{\text{Comm}(p_k)\}, \underset{1\le k\le K}{\text{Max}}\{S(p_k)\}, \underset{1\le k\le K}{\text{Max}}\{\text{QP}(p_k)\} \right\}, \quad (8)$$

$$E^2(\Pi(B)) = \sum_{1}^{K}(\text{Comm}(p_k))^2 + \sum_{1}^{K}(S(p_k))^2 + \sum_{1}^{K}(\text{QP}(p_k))^2. \quad (9)$$

Utilizing these two energy functions, we propose a constructive algorithm that assigns termBuckets to processors in a successive fashion. The termBuckets are processed in some order, and the energy increase in the system by K possible assignments of each bucket are considered. The assignment that incurs the minimum energy increase is performed; that is, for the assignment of a termBucket b_i in the given order, we select the assignment that minimizes

$$E(\Pi(B_{i-1} \cup \{b_i\})) - E(\Pi(B_{i-1})), \quad (10)$$

where B_{i-1} denotes the set of already assigned termBuckets.

We should note here that proposed EA schemes also have the nice property of being easily adaptable for incremental index updates. To enable this feature at the end of inversion process, it is sufficient to store the energy levels of each process. These values then can be used to perform (re)assignment of indexes in an incremental fashion. The minimization of inversion time feature of these schemes would be very helpful in minimizing the incremental update time as well. However, we should note that enabling incremental update in these schemes would necessitate the construction of a global vocabulary on the server node.

We consider both E^1 and E^2 energy definitions and report the results of both schemes in our experiments. We call the E^1-based assignment scheme as E^1A and the E^2-based assignment scheme as E^2A.

5. COMMUNICATION-MEMORY ORGANIZATION

In the final stage of the memory-based parallel inverted index construction, the portions of each posting list are communicated between processors to accumulate each posting list in one processor, where they would be merged in order to construct the final inverted index. This phase can be summarized as an all-to-all personalized communication phase with different number of messages and total message sizes. In this phase, each processor should identify local posting list portions to be sent to other processors, prepare message buffers to send them using the available memory for this communication and send them to the target processors. At the same time, each processor should retrieve posting list portions assigned to them from other processors and merge them in order to generate the final posting lists.

Posting list exchange operation requires intensive communication between processors and dominates the total time required to complete the index inversion. An important question when communicating the posting list portions is how to use/organize the available memory so that the communication phase takes the least possible time. In this work, we evaluate four different communication memory organization schemes and their impact on the total run time of index inversion. These schemes are:

(i) 1-Send 1-Receive buffer scheme (1s1r).
(ii) 1-Send $(K - 1)$-Receive buffer scheme (1sKr).
(iii) $(K - 1)$-Send 1-Receive buffer scheme (Ks1r).
(iv) $(K - 1)$-Send $(K - 1)$-Receive buffer scheme (KsKr).

In investigating different communication-memory organization schemes, we assume that the total memory spared for communication is fixed, say M. In 1s1r, the communication memory is split into one send and one receive buffer, each with size $M/2$. In 1sKr and Ks1r, the memory is split into K buffers each with size M/K. In 1sKr, one of these buffers is used as a send buffer and the remaining $K-1$ buffers are reserved for receiving messages from other processors. In Ks1r, each processor maintains one receive buffer and $K-1$ send buffers, which are reserved for sending messages to other processors. In KsKr, the memory is split into $(2 \times K) - 2$ buffers each with size $M/((2 \times K) - 2)$. $K - 1$ of these buffers are reserved as send buffers as in Ks1r, while the other $K - 1$ buffers are reserved as receive buffers as in 1sKr.

In all of these schemes, the communication commences through several stages. First, all processors issue non-blocking receives for each receive buffer. Then, each processor starts preparing the outgoing send buffer(s). During this preparation, the vocabulary of the local inverted index is traversed in order to copy the local posting list portions to the send buffer(s). Whenever a send buffer is full, the owner processor issues a blocking send operation. The blocking send operation stalls all computation on the sender-side until the send operation is successfully completed. Upon receiving a message, each processor starts emptying its respective receive buffer by copying the received posting list portions to the final inverted index, effectively finalizing the merge of posting list portions. After the merging phase is completed, processors issue a new non-blocking receive in order to receive any remaining messages from other processors, and restart filling their send buffers.

5.1. 1-Send (1s) versus $(K - 1)$-send (Ks) buffer schemes

In the 1s buffer schemes, in order to prepare messages to be sent to other processors, all posting list portions targeted to a specific processor should be put into the single send buffer prior to sending it. For a single target processor, in order to send all required posting list portions, the vocabularies of each local inverted index must be traversed once. As each processor probably requires to communicate with all other processors, preparation of the send buffers requires $K - 1$ traversals over the local inverted index.

On the other hand, in the Ks buffer schemes, in order to prepare outgoing messages, only one traversal of the local inverted index is sufficient. In this traversal, the processor would pick any outgoing posting list portion and place it into the appropriate send buffer. Once one of the send buffers is full, the communication can commence. However, using blocking sends ultimately results in stalling the process every time a send is issued, reducing the processor utilization.

5.2. 1-Receive (1r) versus $(K - 1)$-receive (Kr) buffer schemes

In 1r schemes, the communication memory is fairly utilized, whereas in Kr schemes, the utilization of the communication memory depends on the number of messages received by each processor and may be poor for most of the processors. In 1sKr, since there can be only K messages over the network at any time, only K of the $K \times (K - 1)$ receive buffers would be actively used. In this case, $K \times (K - 2)$ unused receive buffers are left idle, leaving the $(K - 2) \times M$ of the total $K \times M$ memory unused. In KsKr, since there are $K - 1$ send buffers, the processors can produce enough messages to actively use most of the $K \times (K - 1)$ receive buffers, resulting in a more utilized communication memory.

In Kr schemes, since each processor has a specific receive buffer for all other processors, cycles in the communication dependency graph do not cause deadlocks. However, in 1r schemes, depending on the communication order, cycles in the communication dependency graph may cause deadlocks. To avoid these deadlocks, we can utilize non-blocking sends instead of blocking sends. Non-blocking sends allow a processor to continue processing after a send is issued without the need of waiting for it to finalize, thus avoiding any possible deadlocks. However, the issued send still requires its particular send buffer to be intact. As a result, the processor should again be halted in case a local posting list is required to be written in that send buffer. For this reason, each send buffer is locked after a send, and all such buffers are probed after each messaging iteration. If a send buffer is released after a successful send, the lock is freed, allowing the processor to issue writes into that send buffer again.

In Ks1r, whenever a non-blocking send is issued, it is possible to fill other send buffers, allowing computation to overlap with communication. However, in 1s1r, deadlock avoidance via non-blocking sends may cause poor performance since there is only one send buffer and it is not possible to overwrite the contents of this buffer until the non-blocking send is completed, causing the computation to be stalled.

It is also possible to avoid deadlocks in the 1s1r scheme by employing a BSP-like [25] communication/computation pattern and by ensuring that no two processors send messages to the same processor in any given communication step. In 1s1r, since $K - 1$ traversals over the local inverted index is required for each processor, it is possible to divide the computation into $K - 1$ traversal steps and communicate at the end of each computation step. We can also freely choose the communication order in such a scheme. By exploiting this freedom, we can find a communication schedule that avoids deadlocks. Minimizing the number of communication steps induced by this schedule corresponds to minimizing the total inversion time of the proposed BSP-like scheme.

In this work, we show that the problem of finding a communication schedule with minimum number of steps can be reduced to the ‘Open Shop Scheduling Problem’ (OSP). In OSP, there are $|J|$ jobs and $|W|$ workstations. Each job $j_i \in J$ has to visit all workstations and perform a different task. There is an associated time $t(j_i, w_k)$ for finishing job j_i at workstation $w_k \in W$. No restrictions are placed on the execution order of jobs and it is given that no job can be carried out simultaneously on more than one workstation.

In [26], the authors proposed an optimal algorithm to find minimum finish time in an OSP. This is achieved by constructing a bipartite graph from the jobs and workstations, iteratively finding complete matchings over this graph, and modifying the graph by decreasing edge weights of edges in the discovered matching by the smallest edge weight until no more complete matchings can be found. Finding a complete matching ensures that no two jobs are assigned to the same workstation, while no two workstations are working on the same job at any time.

The posting list exchange and merge phase of the index inversion process can also be modeled using the above mentioned algorithm. In the parallel index inversion problem, each processor has to send inverted list portions to other processors. The send operation of inverted list portions corresponds to jobs in the scheduling problem. Also, each send should be received by a processor and merged into the final inverted lists. In that sense, each processor also functions as a workstation in the scheduling problem. There are two associated vertexes, one job vertex and one workstation vertex, for each processor in the bipartite graph. That is, the send responsibilities of processors constitute the jobs and the receive responsibilities of processors constitute the workstations. If a processor p_i has to send a message to processor p_j, there is an associated edge between the job vertex of p_i and workstation vertex of p_j and the number of messages to be sent from p_i to p_j is the weight of this edge. In this model, each match found on the constructed graph correspond to a schedule step, where finding an optimal finish time schedule defines an optimal communication schedule with the least possible number of communication steps.

6. EXPERIMENTS

6.1. Experimental framework

We conducted our experiments on a realistic dataset obtained by crawling educational sites across America. The raw size of the dataset is 30 GB and contains 1 883 037 pages from 18 997 different sites. The biggest site contains 10 352 pages while average number of pages per site is 99.1. The vocabulary of the dataset consists of 3 325 075 distinct terms. There are 787 221 668 words in the dataset. The size of the inverted index generated from the dataset is 2.8 GB. For query load balancing purposes, we used a synthetically generated query log of 1 000 000 distinct queries each of which contains 1 to 7 terms. In our experiments, we used a fixed number of buckets in termBucket-to-processor assignment and set the number of buckets to 10 000.

We tested the performance of the proposed assignment schemes in two different ways: First we report the relative performances of the assignment schemes in terms of the quality metrics described in Section 3.2 through simulations. In simulations we theoretically compute the assignment of terms to processors and compute the storage, query processing and communication costs of the assignment without performing actual parallel inversion. The simulation experiments are conducted for $K = \{4, 8, 16, 32, 64, 128\}$ values on a Sun AMD-Opteron machine with 128 GB of RAM.

Secondly, we provide a set of experiments using actual parallel inversion runs in order to show how improvements in quality metrics relate to parallel running times. For this purpose we developed an MPI-based parallel inversion code that can utilize each of the four communication-memory organization schemes described in Section 5 for a given termBucket-to-processor assignment. These second set of experiments are conducted on a 32-node PC-cluster, where each node is an Intel Pentium IV 3.0 GHz processor with 1 GB RAM connected via an interconnection network of 100 Mb/s fast Ethernet. The total communication-memory size M is set at 5 MB in these experiments.

6.2. Evaluation of the assignment schemes

As a baseline inversion method, we implemented a RT assignment algorithm. In the RT scheme, each term is assigned to a random processor without a term-to-bucket assignment. In this scheme the global vocabulary has to be created. In order to evaluate the viability of term-to-bucket assignment and as a baseline termBucket-to-processor assignment scheme, we also implemented a random assignment (RA) algorithm that assigns buckets to processors randomly. Note that RA requires the least possible time to compute a termBucket-to-processor assignment while avoiding the need for global vocabulary creation, and thus it can be used to compare/analyze the merits of the proposed bucketing scheme and the assignment schemes. The performance of the proposed assignment schemes are compared against the RT and RA schemes.

6.2.1. Simulation results

Tables 1–3 compare the performance of the assignment schemes in terms of the quality metrics described in Section 3.2.

Table 1 displays the performance of the proposed assignment schemes in optimizing the quality metric `QM1`. In the table, the query load imbalance percentages for different assignment schemes and different number of processors is presented. The query load imbalance values are calculated according to the following formula:

$$\left(\frac{\mathrm{Max}_{1\le k\le K}\{Q(p_k)\}}{(\sum_{k=1}^{K}(Q(p_k)))/K}-1\right)\times 100. \qquad (11)$$

Table 2 shows the performance of the proposed assignment schemes in optimizing quality metric QM2. In the table, the initial imbalances due to hash-based distribution and the final imbalances after applying the assignment schemes are presented. The storage imbalance values are computed according to the following formula:

$$\left(\frac{\mathrm{Max}_{1\le k\le K}\{S(p_k)\}}{(\sum_{k=1}^{K}(S(p_k)))/K}-1\right)\times 100. \qquad (12)$$

Table 3 compares the communication performance of the assignment schemes in terms of the average and the maximum message volume to be handled by a processor during parallel index inversion. The total volume of communication required by an assignment scheme can be computed from the table by multiplying the respective average message volume value of the assignment scheme with the respective K value. Thus, the 'Avg' columns of Table 3 indicate the performance of the assignment schemes in optimizing QM3(a). The 'Max' columns in Table 3 indicate the communication load of the maximally loaded processor and thus indicate the performance of the assignment scheme in optimizing QM3(b). The communication-load balancing performance of each assignment scheme can be evaluated by comparing the 'Avg' and 'Max' columns.

The comparison of RT and RA schemes relates to the effectiveness of the proposed term-to-bucket assignment. As shown in Table 2, RA performs slightly better than RT for $K \le 64$. Both Tables 1 and 3 display that RT and RA perform similarly in terms of query load balancing and communication volumes. Comparison of these two assignment schemes shows that term-to-bucket assignment prevents the global vocabulary construction without degrading much our quality metrics.

As seen in Table 1, MCA achieves significantly worse query load imbalance than all other assignment schemes. Similarly, Table 2 shows that MCA considerably degrades the initial storage balance. On the other hand, Table 3 reveals that MCA achieves the best average communication cost. These experimental findings are expected since MCA only considers the minimization of the total communication cost, disregarding storage and communication balancing.

As mentioned in Section 4.2, BLMCA is a modified version of MCA with added emphasis on storage balancing. As seen in Table 2, BLMCA achieves the best final storage balance in all instances. However, as seen in Table 3, the storage balance in BLMCA is achieved at the expense of increased

TABLE 1. Percent query processing load imbalance values.

K	RT	RA	MCA	BLMCA	E^1A	E^2A
4	30.5	54.3	91.4	51.6	47.8	19.6
8	55.5	86.3	115.0	78.2	74.1	24.1
16	100.4	102.8	352.1	92.4	90.1	44.8
32	319.3	233.8	457.3	167.1	123.4	61.7
40	437.2	284.9	755.8	225.6	189.3	79.8
64	602.5	503.7	1446.2	407.9	374.5	112.5
128	857.3	969.4	8456.8	821.7	682.1	216.4

TABLE 2. Percent storage load imbalance values.

		Final					
K	Initial	RT	RA	MCA	BLMCA	E^1A	E^2A
4	13.8	4.4	12.1	38.3	0.0	2.8	5.9
8	31.9	11.7	09.9	60.0	0.1	7.2	14.1
16	38.2	18.2	27.4	66.2	1.7	9.3	23.2
32	58.4	44.1	29.6	83.0	5.4	16.1	32.5
40	66.3	32.2	37.0	77.4	6.2	17.9	33.1
64	69.0	44.7	56.6	92.2	11.5	21.7	36.0
128	81.4	65.3	94.7	95.6	15.7	32.6	45.8

TABLE 3. Message volume (send + receive) handled per processor (in terms of $\times 10^6$ postings).

	RT		RA		MCA		BLMCA		E^1A		E^2A	
K	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
4	131.184	133.233	131.189	145.713	122.091	150.263	127.450	128.619	129.437	134.683	131.857	131.154
8	76.511	88.862	76.554	90.582	71.448	119.745	73.402	75.974	77.562	80.327	77.385	78.229
16	41.002	44.562	41.008	49.249	38.322	77.114	39.217	43.443	43.205	44.944	42.792	42.953
32	21.118	32.254	21.188	28.754	19.817	71.127	20.283	26.025	21.218	25.788	21.047	21.695
40	17.026	26.629	17.053	23.962	15.991	44.793	16.322	20.014	17.072	20.576	17.471	18.118
64	10.761	18.690	10.761	17.769	10.088	74.273	10.339	15.421	10.981	15.222	11.201	12.354
128	5.423	11.883	5.424	11.967	5.088	65.586	5.222	10.980	5.662	10.437	7.233	8.178

total communication volume compared with MCA. Table 1 also shows that especially with increasing K, BLMCA fails in balancing query processing loads.

Table 1 displays that, for all processor values, E^2A performs significantly better than all other assignment schemes in terms of query processing load balance. Additionally, in terms of query load imbalances, E^1A is the second best performer. As seen in Tables 2 and 3, although E^2A slightly degrades the storage balance, it performs better than the other schemes in terms of maximum communication volume handled by a processor for almost all K values (except for $K = 2$ and 4). Although E^1A produces better storage balance than E^2A, the communication volume handled by a processor incurred by E^1A is slightly worse than BLMCA. In terms of the maximum communication volume handled by a processor, E^2A achieves the best results for $K > 8$. Table 3 also indicates that the average and maximum communication volume values induced by E^2A are close, which shows that E^2A manages to distribute the communication load among processors evenly.

6.2.2. *Parallel inversion results*

Table 4 compares the running times of our parallel inversion code for different assignment schemes. Since the creation of the local inverted indexes from local document sets is an operation prior to our inversion schemes, it is assumed that the local inverted indexes are already created. Thus, the time for converting local document collection to local inverted indexes is not included in the inversion times displayed in Table 4.

We provide the RT scheme in order to present the benefits of using a term-to-bucket assignment. The RT scheme differs from other schemes in two ways. First, in the RT scheme termBucket-to-processor assignment is replaced with a term-to-processor assignment. Secondly, in the RT scheme there is an additional phase called global vocabulary construction phase. As seen in Table 4, RT performs significantly worse than other assignment schemes for all K values other than $K = 2$. This indicates that our bucketing scheme has a significant impact on performance.

As seen in Table 4, for $K = 2$, MCA achieves the lowest inversion time compared with the other schemes. This is because, for $K = 2$, minimizing the total communication volume also minimizes the maximum communication volume handled by a processor. However, for all K values greater than 2, MCA performs significantly worse since the maximum message volume handled by a processor for MCA is considerably higher than other assignment schemes. As seen in Table 4, E^2A performs considerably better than the other assignment schemes. For example for $K = 32$, E^2A performs up to 9% better than RA in terms of running time and achieves better final query load and storage balancing. The relative performance order of the assignment schemes in terms of actual inversion time values displayed in Table 4 are generally in concordance with the relative performance order of the assignment schemes in terms of the quality metrics displayed in Tables 2 and 3.

Figure 2 displays the dissection of parallel inversion time into: local inverted index construction, termBucket-to-Processor assignment and inverted list exchange-and-merge phases for different assignment and communication-memory organization schemes on $K = 8$ processors. For the sake of a better insight on the overall index inversion process, the inverted list exchange-and-merge phase is further divided into two components. The first component is called vocabulary communication, where processors send each other the terms, in their word form, and the associated posting list sizes in an all-to-all personalized fashion. The second component is called inverted list communication, where the posting list portions are communicated between processors.

Figure 2 shows that for the in-memory inversion task, the construction of a global vocabulary takes considerable time. For $K = 8$ processors, almost 35% of the total inversion time is spent on global vocabulary construction in the RT scheme.

As seen in Fig. 2, the local inverted index construction takes the same time in all schemes since local index inversion depends only on the initial data distribution. Figure 2 also shows that, as the complexity of the assignment schemes increases, the time required for termBucket-to-processor assignment also increases. The RA-based termBucket-to-processor assignment phase takes <1% of the total inversion time, whereas the E^2A-based termBucket-to-processor assignment phase takes more than 4% of the total inversion time. As the 'Max'

TABLE 4. Parallel inversion times (in seconds) including assignment and inverted list exchange times for different assignment and communication-memory organization schemes.

	K	RT	RA	MCA	BLMCA	E^1A	E^2A
	2	105.90	109.80	85.72	106.08	108.15	108.13
	4	71.60	69.19	81.34	68.63	69.34	68.49
1s1r	8	66.44	51.42	66.76	46.45	47.27	45.74
	16	63.00	35.89	60.82	33.04	33.65	32.48
	32	73.38	19.31	48.45	18.20	18.53	17.20
	2	105.66	109.82	86.04	105.87	107.36	107.97
	4	69.55	73.69	80.31	70.60	71.51	70.71
Ks1r	8	68.10	53.34	68.27	50.04	49.77	48.58
	16	62.59	36.66	60.30	34.32	34.70	32.91
	32	73.10	20.21	50.34	18.52	20.13	17.72
	2	105.17	109.60	86.06	106.31	108.11	108.91
	4	71.37	70.84	80.82	70.11	70.35	69.44
1sKr	8	69.60	58.13	64.97	45.78	47.63	45.22
	16	60.17	34.67	59.13	32.54	32.97	31.41
	32	73.31	20.24	48.36	18.59	19.02	18.07
	2	106.30	110.05	86.13	106.01	108.14	108.12
	4	67.25	66.79	71.89	64.50	64.08	62.51
KsKr	8	62.23	45.44	59.82	41.46	40.89	38.79
	16	57.92	31.28	54.56	29.36	28.78	26.00
	32	72.17	18.43	47.81	18.11	18.01	16.97

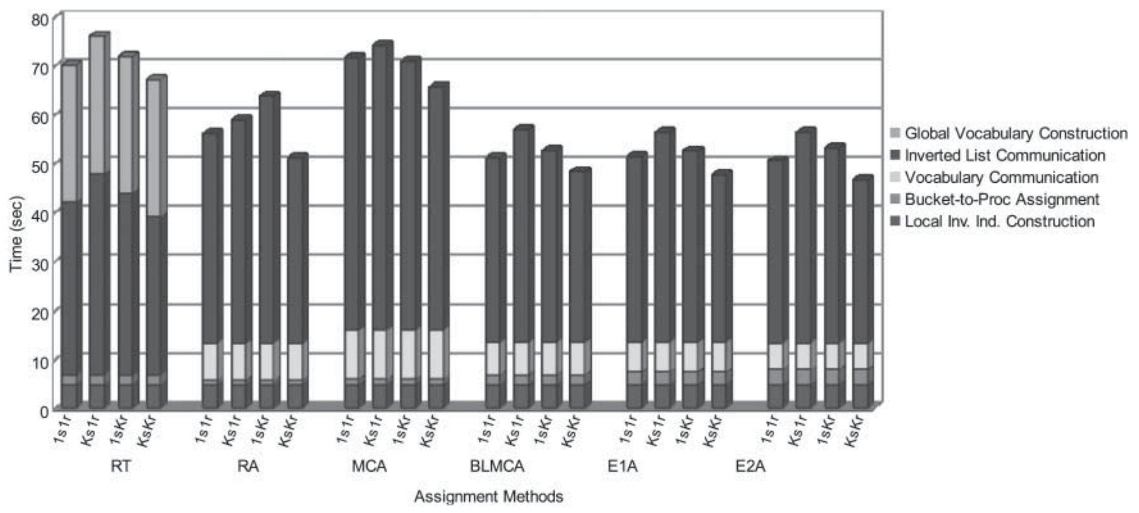


FIGURE 2. Times (seconds) of various phases of the parallel inversion algorithm for different assignment and communication-memory organization schemes on $K = 8$ processors.

columns of Table 3 suggest, the time spent on the vocabulary communication phase is minimum for E^2A and maximum for the MCA assignment scheme.

As seen in Fig. 2, the inverted list exchange-and-merge phase takes almost 85% of the total inversion time, thus confirming that parallel inversion is a communication-bound process. We compare and analyze the impact of different communication-memory organization schemes on this phase in the following subsection.

6.3. Evaluation of communication-memory organization schemes

Table 4 compares the running times of parallel inversion for different communication-memory organization schemes. Ks1r has the worst overall performance for all K values >2. Although Ks1r avoids redundant memory reads by doing only one traversal over the local inverted lists, the use of blocking sends causes stalls and prevents overlap between communication and computation.

Although 1sKr performs better than 1s1r for $K \leq 16$, its relative performance decreases when the number of processors increases. This is due to lower memory utilization of 1sKr on a higher number of processors since each processor must maintain $K - 1$ receive buffers. We theorize that for a higher number of processors, 1sKr would perform even worse.

For all K values >2, KsKr performs superior with respect to the other communication-memory organization schemes. As the number of processors increases, the performance gap between KsKr and the other schemes increases in favor of KsKr. This is because KsKr avoids redundant traversals during the preparation of send buffers and overlaps computation with communication. For this reason, we select KsKr as the de facto communication-memory organization scheme for the remaining experiment.

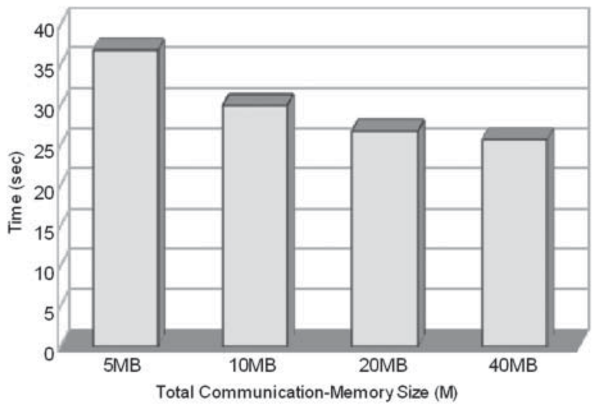


FIGURE 3. The effect of the available communication-memory size (M) on the inverted list exchange-and-merge phase of a $K = 8$ processor parallel inversion system utilizing E^2A and KsKr.

Figure 3 evaluates the effect of the available communication-memory size (M) on the running time of parallel inversion code utilizing the E^2A assignment scheme and the KsKr communication-memory organization scheme for $K = 8$ processors. As seen in Fig. 3, KsKr scales well with increasing communication-memory size. The ability to continue to process several send buffers without stalling allows KsKr to function relatively better with larger communication-memory sizes.

7. CONCLUSIONS

In this paper, a memory-based, term-partitioned, parallel inverted, index construction framework was examined. Several problems were identified and improvements were proposed for a parallel index inversion framework.

First, we proposed a termBucket-to-processor assignment scheme. This scheme minimizes the communication cost of local vocabularies among processors and distributes the final query processing and storage loads among all processors, allowing a finer grained parallelism. We also showed that, by using a termBucket-to-processor assignment scheme, the need to create a global vocabulary can be eliminated and all associated communications can be prevented.

Secondly, we developed and investigated several heuristics for generating a term-to-processor assignment. The results of our experiments show that, compared with a baseline RA scheme, our proposed methods improved the parallel inversion times significantly while providing reasonable final query processing and storage balances.

Thirdly, we presented and explored four different communication-memory organization schemes in order to reduce the communication time required. We also presented methods to avoid deadlocks and network congestion and commented on memory utilization of the overall system. Our results show that splitting the communication-memory in $2 \times (K - 1)$ parts yields the best results.

Simulations and actual parallel inversion times are presented in order to give insight on our improvements. According to the observed results, we recommend the use of the E^2A scheme for termBucket-to-processor assignment, and the $K\mathrm{s}K\mathrm{r}$ scheme for communication-memory organization.

This work can be extended in several ways. First, the framework used in this work does not consider the effect of the bucket processing order. For example, processing buckets in decreasing size order might present better results both in respect of final storage balance and communication costs. Secondly, the number of buckets is assumed to be fixed throughout this work. The scaling of our framework using different numbers of buckets can also be considered.

FUNDING

This work is partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project EEEAG-109E019.

REFERENCES

[1] Salton, G. (1989) *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley Longman Publishing, Boston, MA, USA.

[2] Cho, J. and Garcia-Molina, H. (2000) The Evolution of the Web and Implications for an Incremental Crawler. *Proc. 26th Int. Conf. VLDB*, Cairo, September 10–14, pp. 200–209. Morgan Kaufmann, Egypt.

[3] Moffat, A., Webber, W. and Zobel, J. (2006) Load Balancing for Term-Distributed Parallel Retrieval. *Proc. 29th Int. ACM SIGIR Conf. Research and Development in IR*, Seattle, WA, USA, August 6–11, pp. 348–355. USA.

[4] Witten, I.H., Moffat, A. and Bell, T.C. (1999) *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd edn). Von Nostrand Reinhold, San Francisco, CA, USA.

[5] Moffat, A. and Bell, T.A.H. (1995) In situ generation of compressed inverted files. *J. Am. Soc. Inf. Sci.*, **45**, 537–550.

[6] Heinz, S. and Zobel, J. (2003) Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, **54**, 713–729.

[7] Buttcher, S. and Clarke, C.L.A. (2006) A Hybrid Approach to Index Maintenance in Dynamic Text Retrieval Systems. *Advances in Information Retrieval 28th European Conf. IR Research*, London, April 10–12, pp. 229–240. UK.

[8] Lester, N., Zobel, J. and Williams, H. (2006) Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, **42**, 916–933.

[9] Freider, O. and Siegelmann, H.T. (1991) On the Allocation of Documents in Multiprocessor Information Systems. *Proc. 14th Int. ACM SIGIR Conf. Research and Development in Information Retrieval*, Illinois, October 13–16, pp 230–239. USA.

[10] Jeong, B.S. and Omiecinski, E. (1995) Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, **6**, 142–153.

[11] Ribeiro-Neto, B.A., Kitajima, J.P., Navarro, G., Sant'Ana, C.R.G. and Ziviani, N. (1998) Parallel Generation of Inverted Files for Distributed Text Collections. *Proc. 18th Int. Conf. Chilean Computer Science Society*, Antofagasta, November 12–14, pp. 149. Chile.

[12] Ribeiro-Neto, B.A., Moura, E.S., Neubert, M.S. and Ziviani, N. (1999) Efficient Distributed Algorithms to Build Inverted Files. *Proc. 22nd Int. ACM SIGIR Conf. Research and Development in IR*, Berkeley, CA, USA, August 15–19, pp. 105–112. USA.

[13] Sornil, O. (2001) Parallel inverted index for large scale dynamic digital libraries. Ph.D. Thesis, Virginial Polytechnic Institute and State University.

[14] Melink, S., Raghavan, S., Yang, B. and Garcia-Molina, H. (2001) Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, **19**, 217–241.

[15] Moffat, A., Webber, W., Zobel, J. and Baeza-Yates, R. (2005) A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, **10**, 205–231.

[16] Kucukyilmaz, T., Turk, A. and Aykanat, C. (2011) Memory Resident Parallel Inverted Index Construction. *Proc. 26th Int. Symp. Computer and Information Sciences (ISCIS2011)*, London, September 26–28, pp. 99–106. Springer, UK.

[17] Zobel, J., Moffat, A. and Ramamohanarao, K. (1998) Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, **23**, 453–490.

[18] Brin, S. and Page, L. (1998) The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Proc. 7th Int. Conf. World Wide Web*, Brisbane, April 14–18, pp. 107–117. Elsevier, Australia.

[19] Barroso, L., Dean, J. and Holzle, U. (2003) Web search for a planet: the Google cluster architecture. *Micro IEEE*, **23**, 22–28.

[20] Cho, J. and Garcia-Molina, H. (2002) Parallel Crawlers. *Proc. 11th Int. Conf. World Wide Web*, Honolulu, HI, USA, May 7–11, pp. 124–135. USA.

[21] Cevahir, A., Aykanat, C., Turk, A. and Cambazoglu, B.B. (2010) Site-based partitioning and repartitioning techniques for parallel pagerank computation. *IEEE Trans. Parallel Distrib. Syst.*, **22**, 786–802.

[22] Uçar, B. and Aykanat, C. (2004) Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix–vector multiplies. *SIAM J. Sci. Comput.*, **25**, 1837–1859.

[23] Bisseling, R.H. and Meesen, W. (2005) Communication balancing in parallel sparse matrix–vector multiplication. *Electron. Trans. Numer. Anal.* Special Issue on Combinatorial Scientific Computing, **21**, 47–65.

[24] Aykanat, C., Cambazoglu, B.B., Findik, F. and Kurc, T. (2007) Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *J. Parallel Distrib. Comput.*, **67**, 77–99.

[25] Valiant, L.G. (1990) A bridging model for parallel computation. *Commun. ACM*, **33**, 103–111.

[26] Gonzales, T. and Sahni, S. (1976) Open shop scheduling to minimize finish time. *J. ACM (JACM)*, **23**, 665–679.