Constructing Inverted Files: To MapReduce or Not Revisited

Zheng Wei, *Student Member, IEEE* and Joseph JaJa, *Fellow, IEEE*

Abstract— Current high-throughput algorithms for constructing inverted files all follow the MapReduce framework, which presents a high-level programming model that hides the complexities of parallel programming. In this paper, we take an alternative approach and develop a novel strategy that exploits the current and emerging architectures of multicore processors. Our algorithm is based on a high-throughput pipelined strategy that produces parallel parsed streams, which are immediately consumed at the same rate by parallel indexers. We have performed extensive tests of our algorithm on a cluster of 32 nodes, and were able to achieve a throughput close to the peak throughput of the I/O system: a throughput of 280 MB/s on a single node and a throughput that ranges between 5.15 GB/s (1 Gb/s Ethernet interconnect) and 6.12GB/s (10Gb/s InfiniBand interconnect) on a cluster with 32 nodes for processing the ClueWeb09 dataset. Such a performance represents a substantial gain over the best known MapReduce algorithms even when comparing the single node performance of our algorithm to MapReduce algorithms running on large clusters. Our results shed a light on the extent of the performance cost that may be incurred by using the simpler, higher-level MapReduce programming model for large scale applications.

Key Words— inverted files, MapReduce, multicore processors, cluster, I/O throughput, parallel algorithms, parallel parsing and indexing, pipeline.

— — — — — — — — — ◆ — — — — — — — — —

1 INTRODUCTION

THE main goal of this work is to develop optimized throughput strategies for constructing inverted files on a cluster of multicore processors, which exploit current and emerging architectures of multicore processors. At the same time, we compare the resulting performance to the best performance achieved by the much simpler and higher level MapReduce algorithms thereby shedding some light on the tradeoff between the programming simplicity of the MapReduce framework and the performance of carefully fine-tuned strategies to the underlying architectures. The current trend in CPU architectures increasingly includes more cores on a single chip, several levels of cache, and a large RAM. In particular, it is expected that the number of cores will double every 18 to 24 months, and such trend is likely to continue in the foreseeable future, and such a trend is likely to continue in the foreseeable future. These multicore processors offer opportunities for speeding up demanding computations if the available resources can be effectively used, which is in general very hard to accomplish for large complex computations such as the generation of inverted files.

The extraction of inverted files from a very large collection of documents forms a critical component of all information retrieval systems including web search engines. A considerable amount of research has been conducted to deal with various aspects related to inverted files. In this paper, we are primarily concerned with methods to generate the inverted files with the best pos-

sible throughput. All the recent fast indexers use the simple MapReduce framework on large clusters, which enables quick development of parallel algorithms dealing with internet scale datasets without having to deal with the complexities of low-level parallel programming. Such framework leaves the details of scheduling, processor allocation, and communication to the underlying run time system, and hence relieves programmers from all the extra work related to these details. However such an abstraction may come at a price in terms of performance, especially when using the emerging multicore processors. In this paper, we take the different approach that tries to exploit the common features present on current multicore processors to develop an optimized high-throughput algorithm and compare its performance to the best known MapReduce algorithms.

We conduct extensive tests of our algorithm on a cluster of 32 nodes, each node consisting of two Quad-core Intel Xeon X5560 processors with 24 GB of main memory and each quad-core shares an 8MB L3 cache. In our tests, either a 10Gb/s InfiniBand or a 1Gb/s Ethernet is used as the interconnect fabric in our cluster; moreover, the input collection of documents is either distributed among the disks attached to the nodes or stored on a separate storage pool connected to the cluster through a 4Gb/s pipe. Each node offers a multithreaded environment with a shared memory programming model and the nodes communicate with each other using the Message Passing Interface (MPI) framework.

The main contributions of this paper are:

- Development of an optimized high-throughput pipelined strategy for a cluster of multicore

---

• *Z. Wei and J. JaJa are with the Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. E-mail: {zwei, joseph} @umiacs.umd.edu.*

processors, under either the 1Gb/s Ethernet or the 10Gb/s interconnect, and under either the distributed storage model or the centralized storage pool model.

- Introduction of a number of new techniques to partition the indexing workload while minimizing the communication and ensuring load balancing in such a way that the parallel parsed streams are immediately consumed at the same fast rate by the distributed, parallel indexers.

- Generation of extensive experimental results illustrating scalability relative to the optimized single node algorithm. In particular, each node achieves a throughput of 280MB/s, leading to over 6GB/s for the 32-node cluster when the InfiniBand interconnect is used and over 5GB/s when the 1 Gb/s Ethernet is used. The performance results seem to be substantially better than the best previous published results that adopt the MapReduce framework.

The rest of the paper is organized as follows. In the next section, we provide a brief background about the typical strategy used to build inverted files and a summary of the previous work that is most related to our paper. Section 3 provides a description of an algorithm optimized for a single node with multicore processors, which was introduced in our earlier work [1]. Section 4 extends the algorithm to a cluster of multicore processors while Section 5 provides a summary of our test results on three very different, significant benchmarks. We conclude in Section 6.

2 BACKGROUND AND PREVIOUS RELATED WORK

Our overall process converts a collection of documents into inverted files consisting of a postings list for each of the terms appearing in the collection as follows. This well-known strategy starts by parsing each document into a "bag of words" of the form <term, document ID> tuples, followed by constructing a postings list for each term such that each posting contains the ID of the document containing the term, term frequency, and possibly other information. Parsing consists of a sequence of simple steps: *tokenization*, *stemming*, and *removal of stop words*. *Tokenization* splits a document into individual tokens; *stemming* converts different forms of a root term into a single common one (e.g. "parallelize", "parallelization", "parallelism" are all based on "parallel"); and *removal of stop words* consists of eliminating common terms, such as "the", "to", "and", etc. The overall parsing process is well understood, and follows more or less the same linguistic rules, even though there exist different stemming strategies.

The next phase consists of constructing the inverted index. All <term, document ID> tuples belonging to the same term are combined together to form the postings list of that term. During the construction, a dictionary is usu-ally built to maintain the location of the postings list of each term and to collect some related statistics. Postings on the same list are usually organized in a sorted order of document IDs for faster look up. Indexing is a relatively simple operation—group tuples for the same term together and then carry out sorting by document IDs—but it is always by far the most time consuming part given the typical size of the collection to be indexed.

Recent work includes the sort-based indexing [2] proposed by Moffat and Bell for limited memory. Their strategy builds temporary postings lists in memory until the memory space is exhausted, sorts them by term and document ID and then writes the result to disk for each run. When all runs are completed, it merges all these intermediate results into the final postings lists file. The dictionary is kept in memory; however as the size grows, there may be insufficient space for temporary postings lists. Heinz and Zobel [3] further improved this strategy to a single-pass in-memory indexing version by writing the temporary dictionary to disk as well at the end of each run. Dictionary is processed in lexicographical term order so adjacent terms are likely to share the same prefix and front-coding compression is employed to reduce the size.

We now turn to a review of the major parallel strategies that appeared in the literature. In [4], the indexing process is divided into loading, processing and flushing; these three stages are pipelined by software in such a way that loading and flushing are hidden by the processing stage. The Remote-Buffer and Remote-Lists algorithm in [5] is tailored for distributed systems. In the first run, the global dictionary is computed and distributed to each processor and in the following runs, once a <term, document ID> tuple is generated, it is sent to a pre-assigned processor where it is inserted into the destination sorted postings list. Today, MapReduce based algorithms are prevalent. First proposed in [6], the MapReduce paradigm provides a simplified programming model for distributed computing involving internet scale datasets on large clusters. The Map workers emit <key, value> pairs to Reduce workers defined by Master node, and the runtime would automatically group incoming <key, value> pairs received by a Reduce worker according to key field and pass <key, list of values associated with this key> to the Reduce function. A straightforward MapReduce algorithm for indexing is to use term as key and document ID as value, in which case the Reduce workers can directly receive unsorted postings lists. Since there is no mechanism for different Map workers to communicate with each other, creating a global dictionary is not possible. McCreadie et.al let Map worker emit <term, partial postings list> instead to reduce the number of emits and the resultant total transfer size between Map and Reduce since duplicate term fields are less frequently sent. Their strategy has achieved a good speedup relative to the number of nodes and cores [7, 8]. Around the same time, Lin et.al [9, 10] developed a scalable MapReduce Indexing algorithm by switching <term, posting{document ID, term frequency}> to <tuple{term, document ID}, term frequency>. By doing so, there is at most one value for each unique key, and moreover it is guaran-

teed by the MapReduce framework that postings arrive at Reduce worker in order. As a result, a posting can be immediately appended to the postings list without any post processing. Their algorithm seems to achieve the best known throughput rate for full text indexing.

We note that almost all the above strategies perform compression on the postings lists for otherwise the output file would be quite large. Because document IDs are stored in sorted order in each postings list, a basic idea used is to encode the gap between two neighbor document IDs instead of their absolute values combined with a compression strategy such as variable byte encoding, γ encoding and Golomb compression.

3 ALGORITHM ON A SINGLE MULTICORE NODE

The starting point of our cluster algorithm is the pipelined strategy on a single node with multicore processors presented in our earlier paper [1]. This section is devoted to an overview of this strategy.

3.1 Overall Approach

Briefly, a number of parsers run in parallel on the multicore CPU, where each parser reads a fixed size (typically, 1GB) block from the disk containing the documents, executes the parsing algorithm, and then writes the parsed results onto a buffer. A number of indexers pull parsed results from the buffer as soon as they are available and jointly construct the postings lists, which are written into a disk as soon as they are generated. The dictionary remains in main memory until the whole process is completed. Such a pipelined data flow avoids writing intermediate results onto disks unlike the Map workers used in the MapReduce framework, which typically transfer data to Reduce workers via disks [6-10].

There are many details that need to be carefully worked out for this approach to achieve optimal throughput. Here we summarize the key aspects used in the rest of the paper, starting with the dictionary data structure.

In [1], we introduce a hybrid data structure consisting of a trie at the top level and a B-Tree attached to each of the leaves of the trie as shown in Fig. 2. Essentially, terms are mapped into different groups, called *trie-collections*, each of which is then represented by a B-tree.
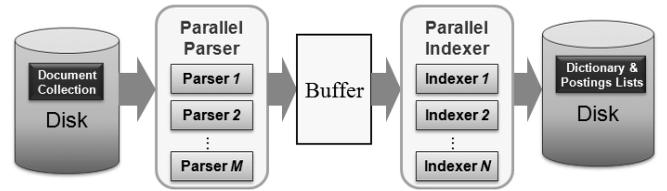

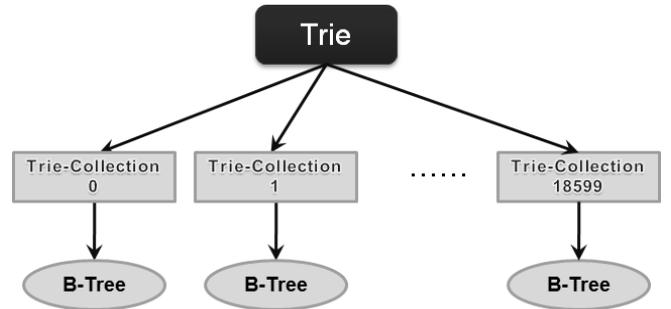Fig. 1. Pipelined and Parallel Parsing and Indexing on a Single Node

Fig. 2. Hybrid of Trie and B-Tree Structure of Dictionary

In our case, we fix the height of the trie to three, which implies that the first three letters in a term are used to determine the corresponding the index of the trie collection. We observe that there are still a significant number of terms with less than four letters or have at least one letter outside range [a-z] in the first three letters. To accommodate such terms, we create additional 1024 trie collections indexed 0-1023 and use a hash function for a balanced distribution.

In addition to allowing a high degree of parallelism through the independent B-trees, our hybrid data structure achieves two additional benefits. Since we replace a big B-tree by many small B-trees, the heights of the B-trees are smaller, implying that the time to search or insert a new term is reduced as well. Another advantage of the trie lies in the fact that terms belonging to the same trie index share the same prefix (except trie indices 0-1023) and hence we can eliminate such common prefix and save memory space for term strings and reduce string comparison time in B-tree operations. The average length of a stemmed token is 6.6 in the ClueWeb09 dataset and hence removing the first three letters results in almost doubling the string comparison speed. An alterna-

TABLE 1
TRIE-COLLECTION INDEX DEFINITION

Index		Term Category	Example
Terms not Falling in to the Next Categories (1024 entries)	0	Terms with less than four letters or contain one or more symbol outside [a-z] in the first three letters	"-80", "3d", "Česky" "01","0195" "9", "954" "a", "at", "act", "añonuevo" "z", "zoo", "zoé"
	1		
	…		
	1023		
Terms with >3 letters and no special letter in the first 3 letters (26*26*26=17576 entries)	1024	Terms with >3 letters and starting with 'aaa'	"aaat", "aaaé"
	1025	Terms with >3 letters and starting with 'aab'	"aabomycin"
	…	…	…
	18599	Terms with >3 letters and starting with 'zzz'	"zzzy"

TABLE 2
DATA STRUCTURE OF ONE B-TREE NODE

Field	Number	Data Size (Byte)
Valid term number	1	4
Pointer to term string	31	124
Leaf indicator	1	4
Pointer to postings lists	31	124
Pointer to children	32	128
4-Byte cache for term string	31	124
Padding	1	4
Total Size		512

Fig. 3. Data Flow of One Parser Thread

tive option to the trie is to simply use a hash function for all the terms, but this will still require comparisons and searches to be performed on whole strings and hence won't be as effective as the trie.

The structure of a B-tree node is illustrated in Table 2. The degree of B-tree is 16, that is, each node can hold up to 31 terms. Since the length of a term string is not fixed but varies over a wide range, it is impossible to store the strings within a fixed B-Tree node; instead, pointers are used to indicate the memory location of the actual strings. During a search or insert operation into one of the B-trees, strings are accessed through these pointers, and such operations can be quite expensive. To get around this problem, we include 31 four-byte caches in each node. These caches are used to store the first four bytes of the corresponding term strings.

3.2 Structure of Parallel Parsers

As shown in Fig. 1, we will have M parsers running in parallel on a single node. Each parser processes a segment of documents independently after reading the segment from disk as illustrated in Fig. 8. The number M of parsers depends on the number of CPU cores and overall resources available, to be discussed later.

Here we describe the sequence of operations executed by each parser, illustrated in Fig. 3. Each such sequence will be executed by a single CPU thread. The corresponding steps are briefly described next.

- **Step1** reads files from disk, decompresses them if necessary, assigns local document ID to each document, and builds a table containing <document ID, document location on disk> mapping.

- **Step2** performs tokenization, that is, parses each document into tokens and determines the trie index of each resulting term.

- **Step3** performs Porter stemmer.

- **Step4** removes stop words using a stop word list.

- **Step5** rearranges terms with the same trie index so that they are located contiguously. In addition, the prefix of each term captured by the trie index is removed.
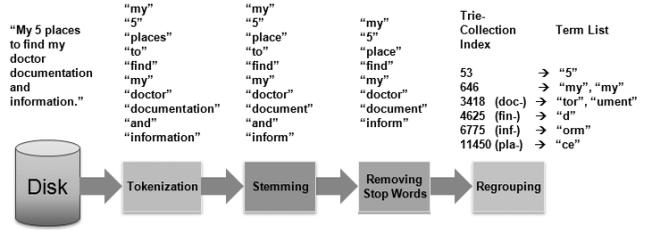
The first four steps are standard in most indexing sys-

tems. Step5 is special to our algorithm. Essentially, this step regroups the terms into a number of groups, a group for each trie collection index as defined by our dictionary data structure. We note that the overhead of this regrouping step is relatively small, about 5% of the total running time of the whole parsing process. This is due to the fact that tokenization scans input document character by character and hence a trie collection index can be calculated as a by-product using a minimal additional effort.

This regrouping is needed for our parallel indexing algorithm. More specifically, when indexing is carried out by a serial CPU thread, regrouping results in approximately 15-fold speedup based on our tests. The improved performance is due to improved cache performance caused by the additional temporal locality. Now we are processing a group of terms falling under the same trie collection index, which are inserted into the same small B-tree whose content stays in cache for a long time.

Therefore, after processing a number of documents (contained in a 1GB file in our case), the parsed results organized according to trie index values will be passed to the indexers. For each trie collection, the parsed results will look like:

Trie Collection: (Doc_ID1, term1, term2, …), (Doc_ID2, term1, term2, …), …

Doc_IDs on the lists are local IDs within each parser. A global document ID offset will be calculated by the indexer and then the global document ID can be obtained by adding *Doc_ID* and the global offset.

3.3 Structure of Parallel Indexers

The purpose of an indexer is to construct all the B-trees and the postings lists corresponding to each input term as shown in Fig. 4. To ensure load balancing, a CPU thread will take care of the B-trees of several trie collections as we explain later.

An indexer is executed by a single CPU thread, which follows the commonly used procedures for building the B-tree and the corresponding postings lists. The only difference is to make use of the fact that a cache is included within each B-tree node. Hence, when a new term is inserted into a B-tree, the first 4-bytes of the string are stored in the string cache field in the appropriate B-tree node. The remaining bytes, if any, are stored in another memory location, which can be reached via the string pointer for this term.

We observe that two tokens, appearing close to each other in a single document and belonging to the same trie
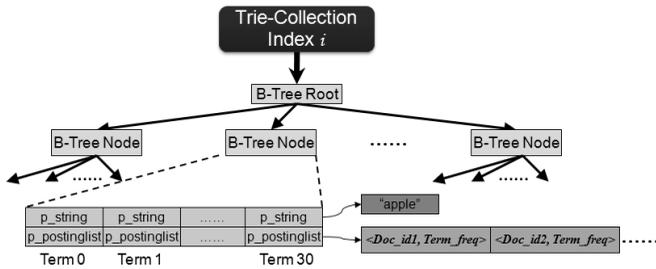
Fig. 4. A B-tree Corresponding to a Single Trie-Collection Index

Fig. 6. Data Flow of One Single Run on Parallel Indexers

collection, are likely to be the same term. For example, "that" is a commonly used term and hence the next term with prefix "tha-" is also likely to be "that"; on the other hand, an unusual term such as "zooblast" has the same implications since there are few terms with prefix "zoo-". We can mine such linguistic facts here because of the trie structure that groups terms with common prefix together. Therefore, we use a special cache to store the last term inserted into B-Tree and the location of its postings list. Then we compare the next term with the term stored in the cache and if they match we skip the B-tree operations and immediately update the corresponding postings list. We enable such cache only within a single document because different documents will behave differently in which case caching is ineffective in general.

We now address the issue of assigning the 18,600 trie collections among the parallel indexer threads so that the load will be distributed almost equally among the threads.

In [1], we argue that a sampling strategy is the most effective to allocate parsed streams to indexers. Sampling refers to extracting a sample from the document collection at the very beginning, for example a random 1MB out of every 1GB, and run several tests on the sample to determine the best partitioning strategy of the trie collections. In this case, once a trie collection is assigned to a certain indexer, it will always be processed by the same indexer throughout the lifetime of the algorithm, that is, there is a persistent binding between a trie collection and the indexer ID.

In addition to the main indexing step, pre-processing delivers input from buffer to multiple indexers and post-processing combines postings lists from all indexers, compresses them with variable byte encoding and then
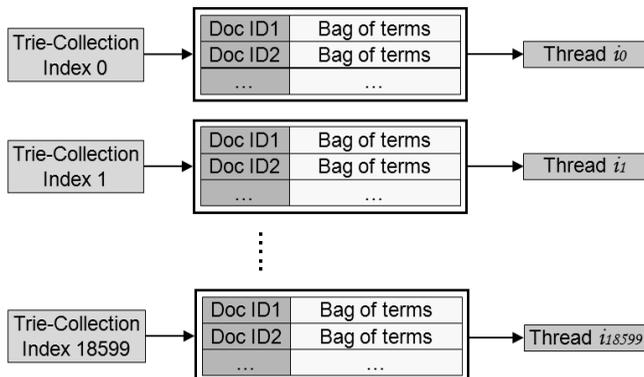
writes the compact results to disk. These two steps are serialized. Each iteration, beginning with the data in a parser buffer and ending in postings lists is referred to as a run illustrated in Fig. 6.

3.4 Overall Pipelined Data Flow

In our setting, the input document data collection is stored on a disk and is processed through our multicore CPU platform to generate the postings lists and store them on a disk. The dictionary is kept in main memory until the last batch of documents is processed, after which it is moved to the disk. The number of parsers and the number of indexers are determined depending on the physical resources available. In Section 5, we determine the best values of these parameters for our platform.

To avoid several parsers from trying to read from the same disk at the same time, a scheduler is used to organize the reads of the different parsers, one at a time. On the other hand, an output buffer is allocated to each parser to store the corresponding parsed results. The indexers in the next stage will read from these buffers in order, that is, (buffer of Parser 0, buffer of Parser 1, …, buffer of Parser M-1, buffer of Parser 0, …). Such read sequence is enforced to ensure that document first read from disk will also be indexed first so the postings lists are intrinsically in sorted order of assigned document IDs. A parser has to also wait until buffer is cleared to start the parsing of the next block of documents to ensure that it has the space to write the parsed results. When these constraints are ap-
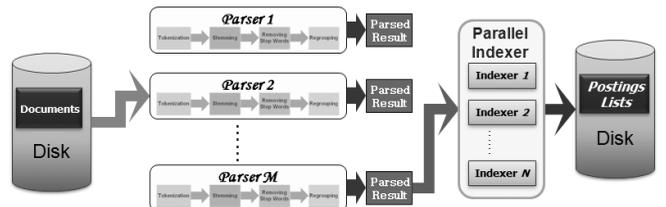

Fig. 7. Pipelined and Parallel Parsing and Indexing on a Single Node

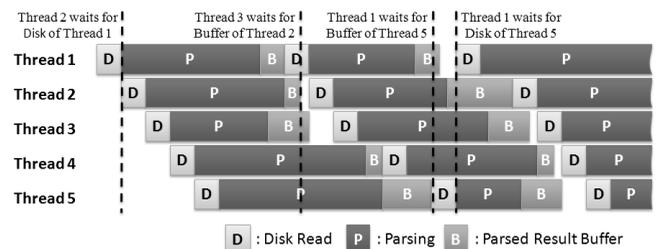Fig. 5. Work Assignments among Multiple Indexer Threads

Fig. 8. Timing Sequence of Parallel Parsers

plied, the timing sequence of parallel parsers looks like the example shown in Fig. 8.

We note that a separate output file is created for the postings lists generated during a single run, whose header contains a mapping table indicating the location and length of each postings list. This mapping table is indexed by the pointers to postings lists stored in the dictionary as shown in Table 1. To retrieve a postings list for a certain term string, we look it up in the dictionary and use the corresponding pointer to determine the location of the partial postings list in each of the output files. Additional benefits of this output format are described in [1]. If necessary, we can combine the partial postings lists of each term into a single list in a post-processing step, with an additional cost of less than 10% of the total running time.

4 ALGORITHM ON A CLUSTER

We now extend our single node strategy to a cluster of multicore processors. Our goal is to build a global dictionary and generate the postings lists stored on external storage with the maximum possible throughput. There are two possible strategies to extend the algorithm.

- **Divide-and-Merge.** *Each node processes an equal portion of the document collection following the single node algorithm, after which the local dictionaries and postings lists from all the nodes are merged. This method follows the standard divide-and-conquer strategy and hence its effectiveness depends on the merging phase.*

- **Partition-and-Index.** *At the end of each parsing stage, parsed streams are distributed among the cluster nodes in such a way that parallel indexers complete the indexing process with no need to communicate. This strategy includes a sampling preprocessing step that creates a persistent mapping between the trie collection indexes and the IDs of the indexers, which is used to distribute the parsed streams to the nodes.*

It is clear that the divide-and-merge strategy will achieve excellent performance during the first stage of parsing and indexing because every node will work independently on its portion of the document collections with no communication required between the nodes. However the merge stage is quite complex since all the different tries and their trie collections have to be combined into a single global indexing structure, a task that seems to require a substantial communication and coordination overhead.

On the other hand, the partition-and-index approach requires a careful fixed (regardless of the block of documents being processed) assignment of trie collections to indexer thread IDs so that the generated output (trie, B-trees, and postings listings) will always be distributed almost equally among the nodes. This strategy incurs some communication overhead up front immediately after a block of documents are parsed. However, at any time, our approach ensures that the dictionary is a coherent, global dictionary, stored on multiple nodes, and the postings lists will contain global document IDs. To handle
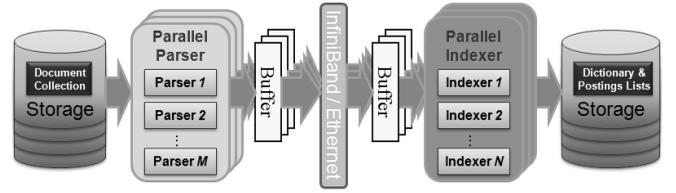

Fig. 9. Data Flow of Partition-and-Index Strategy

the interprocessor communication between the parsing and indexing phases of the pipelined algorithm, we insert a separate communication phase into the original pipeline. The latency of the pipeline increases but we will introduce techniques to ensure that the throughput will stay more or less the same.

The data flow of the partition-and-index approach is illustrated in Fig. 9. Unlike the case of a single node where all parsing or indexing threads share the same main memory, the parsers and indexers are now spread across the cluster and communicate through the interconnect fabric (10Gb/s InfiniBand or 1GB/s Ethernet in our case). This will be described in more details shortly.

4.1 Storage Model: Centralized Storage Pool Versus Distributed Storage

Every node of our cluster has two disks attached to it; in addition, the cluster also has a 4Gb/s link to a remote file server managing hundreds of terabytes of storage. Therefore two storage models for handling the input and output files are possible: (i) all files reside on the remote storage pool; or (ii) the files will be distributed to the disks attached to the nodes. The remote storage pool model seems more appealing for realistic scenarios since documents are usually deposited in a centralized storage pool, processed on a cluster, and then the inverted files are transferred to another cluster for search and retrieval. In our case, our storage pool model has a serious drawback, namely the 4Gb/s bandwidth that cannot keep up with the necessary throughput when we use more than 8 nodes on our cluster. The distributed storage model is similar to the storage model used in MapReduce since a distributed file system is used on the nodes of the cluster. Moreover, this model can provide scalable I/O bandwidth as a function of the nodes available. The output, including dictionary and postings lists, is stored on local disks. We will test our algorithm using both models.

4.2 Partitioning the Work among the Nodes

As in the case of the single node algorithm, the document collection is divided into fixed-sized segments (typically 1GB WARC files) which are assigned to parallel parsers. In both centralized and distributed storage models, read requests of parallel parsers from the same node are serialized to avoid contention on network interface or local disks. Note that, under the centralized storage model, read requests from different nodes have to compete for the 4Gb/s connection to the storage pool. In both cases, parallel parsers work independent of each other except when reading the data from external storage.

We now address the critical issue on how to assign the

workloads to the indexers. Prior to parsing, we collect a document sample (specifically, a random 1MB from each 1GB file) from the collection, parse it, and use the parsed stream to determine an almost equal-size partition of the trie collections into $k=N*P$ partitions, where N is the number of indexers per node and P is the number of nodes. We then use the k partitions to create a mapping between trie collections and indexers, which will create a binding that will persist throughput the processing of the document collection. As a result, the postings lists associated with a certain trie collection will all be written to the same local disk of the node where the corresponding indexer is running.

Another more elaborate strategy consists of a combination of sampling and dynamic round robin scheduling, where trie collections are first assigned to the nodes rather than indexers using the sampling method, followed by a dynamic round robin scheduling to allocate the work among the indexers on each node. This strategy achieves a better load balance than just sampling but the overall throughput is not as good, due to cache locality that is clearly enhanced when there is a persistent binding between trie collections and indexers.

Once the parsers on a node process their documents, the trie collections (each consisting of a document ID, followed by the corresponding bag of words, another document ID followed by its bag of words, and so on in sorted order by document IDs) will be distributed to the nodes according to the assignment determined by the sampling method. Indexers on a node will start indexing at the same time once the previous load is consumed and the next message load arrives. Note that indexers from different nodes will not necessarily start indexing at the same time because messages may reach their destinations at different times. Our main goal is to ensure that all parsers and indexers are kept busy so as to achieve the maximum possible throughput.

4.3 Communication Strategy between Parsers and Indexers

A straightforward way to manage the communication between parsers and indexers is to let each parser thread construct and send P MPI messages after each segment is parsed. This strategy does not work well when the number of nodes is large due to the presence of many very small messages as P increases. For example, consider the ClueWeb09 collection, for which a segment is of size 1GB and the corresponding parsed stream is of size 130MB. In this case, the size of a message is about 4MB when $P=32$,
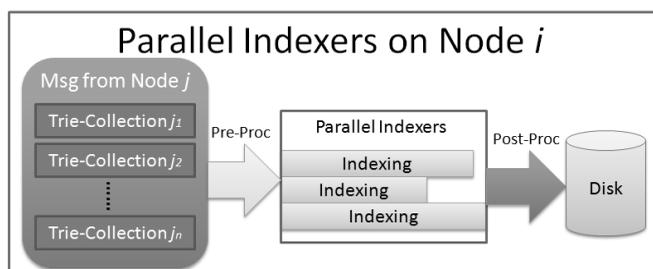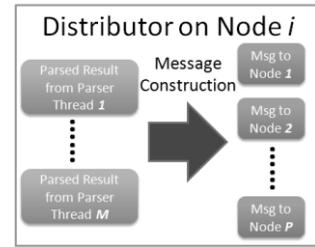

Fig. 11. Message Construction by Distributor

which only takes about 8 ms time to send it from one node to another using the 10Gb/s InfiniBand and about 60ms using the 1Gb/s Ethernet while the overhead to initialize such message is comparable to the transmission time. We can increase the collection segment size but we are limited by the memory size of each node as we have to be able to accommodate the segments for all the M parsers at the same time.

To address this problem, we introduce the notion of a distributor to manage communication in the pipeline. The job of a distributor on each node is to collect parsed results from the parsers running on the node over several segments, and then build the corresponding messages to the P nodes. The size of each parsed stream is much smaller than the original collection segment, and hence the memory can accommodate the parsed results of tens of segments. No changes are required for the parsers, except that the parsed results are now consumed by the distributor.

Another task of the distributor is to update the document IDs before the messages are constructed. Document IDs appearing in the parsed streams are local to each collection segment; therefore these need to be modified into DOC_IDs relative to the corresponding batch of parsed

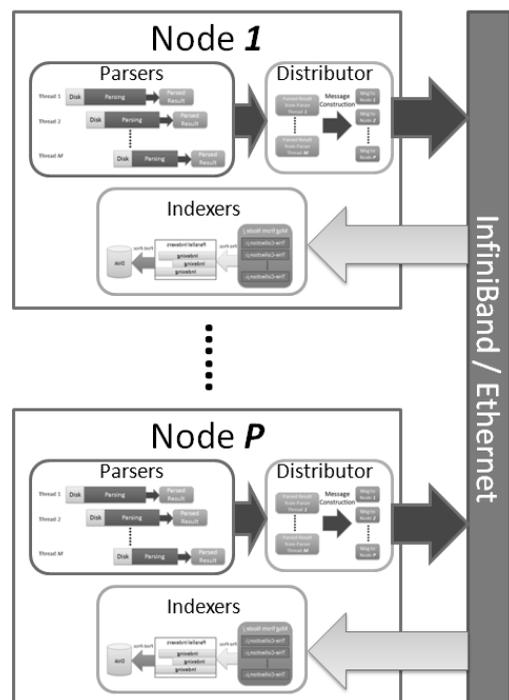Fig. 10. Data Flow of Parallel Indexers on One Node in the Cluster

Fig. 12. Overall Data Flow in the Cluster

results. The total number of documents is also included in the messages distributed to indexers so that indexers can calculate global offsets for *DOC_IDs* from the history of document numbers.

4.4 Overall Data Flow on the Cluster

Putting all the pieces together, we get the overall data flow shown in Fig. 12 for a cluster of multicore processors. Similar to the single node case, we use synchronous communication to enforce the sequence of messages processed by indexers, that is, each node sequentially receives messages from node 1 through node P.

The data sizes or segment numbers processed by different parsers in the cluster are not necessarily the same. For example, in the distributed storage model data are not split evenly among local disks on all nodes. In this scenario, some parsers exit earlier than others (i.e., when all assigned segments are processed), but all indexers stop when the last batch is completed.

5 EXPERIMENTAL RESULTS

We test the performance of our algorithm on a cluster with 32 nodes, each node holding two Intel Xeon X5560 Quad-core CPUs and 210 GB disk. We use three significant collections that exhibit different characteristics. We start with **ClueWeb09 English** collection, which has been heavily utilized as a benchmark by the information retrieval community. Crawled between January and February 2009 by Language Technologies Institute at Carnegie Mellon University, this data set includes 503,903,810 web pages packed into 13,217 files of total size 1.89TB compressed and 12.16 TB uncompressed. A subset of this collection, the first English segment, is used to tune parameters and compare results with previously published results. The second data set is the **Wikipedia01-07** data, which is derived from a publicly available XML dump of Wikipedia articles created on January 3th 2008 with 83 monthly snapshots between February 2001 and December 2007. The third collection is the **Congressional** data set from the Library of Congress, which includes weekly snapshots of selected news and government websites crawled between May 2004 and September 2005 by Internet Archive. The overall characteristics of the four benchmarks are given in Table 3. The number of terms and tokens may vary in different implementations due to the choice of tokenization and stemming procedures.

The generated output, postings lists and dictionary, are written to local disks. We report results that are averaged over three trials but we note that, in all our tests, the differences between the fastest and slowest execution times have been less than 5%. We first report the results on the cluster using the 10Gb/s InfiniBand interconnect, and later report the results for the case for the 1Gb/s Ethernet interconnect. Disk cache in memory is carefully cleared prior to every experiment. The throughput numbers correspond to the uncompressed collection size divided by the corresponding total running time.

Before proceeding, we examine the format of the input data to be processed by the parsers. A typical file of the ClueWeb09 data set is about 160MB compressed and 1GB uncompressed. On average, it takes about 1.6 seconds to read such a compressed file from either a local disk or the storage pool, and 3.2 seconds to decompress it. On the other hand, it takes about 10 seconds to read the uncompressed file. Therefore we load the compressed files and then decompress them in memory before parsing. There are two possible options to proceed: decompression can be folded into either the file read stage or can be performed as a separate step after reading. The advantage of the former is that decompression can be partially hidden by file reading time if decompression starts whenever partial data becomes available in memory, so the overall time for reading and decompressing a file takes 3.8 seconds on average, which translates into 263MB/s intake bandwidth. The disadvantage of this method is that the file access right cannot be released to another parser until reading and decompression are both completed. This causes a mismatch between the data generated by the parsers and the data consumed by the indexers. Hence we choose the second scheme in which decompression starts after the file is fully transferred to memory. In this case, the average time to read a compressed file is $(1.6+3.2/M)$ seconds where M is the number of parallel parsers. When $M=6$, the intake bandwidth reaches as high as 467MB/s.

In what follows, we start by determining the best values of the numbers of parsers and indexers for the single node algorithm (described in Section 3), which will be used as the basis for our scalability results. This will be followed by summarizing the performance of our single node algorithm on the three document collections. We then show that our cluster algorithm is scalable, relative to the optimized single node algorithm, up to the largest number of available nodes, using several scalability metrics. We end by comparing the performance of our algorithm to the best known results in the literature.

TABLE 3
STATISTICS OF DOCUMENT COLLECTIONS

	ClueWeb09 English	ClueWeb09 1st Eng Seg	Wikipedia 01-07	Library of Congress
Compressed Size	1,936GB	230GB	29GB	96GB
Uncompressed Size	12,453GB	1,422GB	79GB	507GB
Crawl Time	01/09 to 02/09	01/09 to 02/09	02/01 to 12/07	05/04 to 09/05
Document Number	503,903,810	50,220,423	16,618,497	29,177,074
Number of Terms	447,373,242	84,799,475	9,404,723	7,457,742
Number of Tokens	281,794,398,151	32,644,508,255	9,375,229,726	16,865,180,093

Fig. 13. Optimal Number of Parallel Parsers and Indexers on A Single Node

Fig. 14. Detailed Throughput of Parallel Indexers

5.1 Optimal Numbers of Parallel Parsers and Indexers on a Single Node

The performance of our single node algorithm on the ClueWeb09 first English segment as a function of the number M of parsers is shown in Fig. 13 under two scenarios: (1) M parsers and 8-M indexers; and (2) M parsers without any indexers. The value of M varies from 1 to 7 since there are only eight cores on each node. The second scenario illustrates the best possible throughput achieved by just parsing the document collection.

When the number of parsers is within the range 1 through 6, we observe similar performance in both scenarios, including an almost linear scalability as a function of the number of parsers. This indicates that the indexers are keeping up with the data generated by the parsers and hence, within this range, the parsers constitute the slow stage of the pipeline. The major limitations to speeding up the parsers include the sequential access to the single disk and the contention on cache and memory resources. Beyond 6 parsers, when the number of indexers decreases, the indexing pipeline stage is not able to catch up with the parsing stage, indicating that a ratio of 6:2 between parsers and indexers is the best possible on our single 8-core CPU.

5.2 Indexing Throughput and Dictionary Growth

Given that we have already determined that the best overall performance on a single node is achieved by using six parsers, we now take a closer look at the indexing throughput of parallel indexers, not including the pre-processing and the post-processing steps. We track the time of the parallel indexers spent on each file in the ClueWeb09 first English segment and compute the throughput for each file as shown in Fig. 14. Note that starting with file index 1,201, we can see a significant drop in performance. This can be explained by the fact that the files with indices from 1,201 to 1,492 all belong to Wikipedia.org, and hence they exhibit a totally different behavior than the earlier documents. This portion of the Wikipedia files is relatively small within the ClueWeb09 first English segment, and hence the parameters determined by the sampling process do not effectively reflect the characteristics of this small subset.

The overall slope consists of a sharp decrease near the beginning followed by a trend that approaches a horizontal line. This pattern correlates well with the inverse of the depth of B-tree because as the B-trees grow deeper, it takes more time to perform insert or search operations.
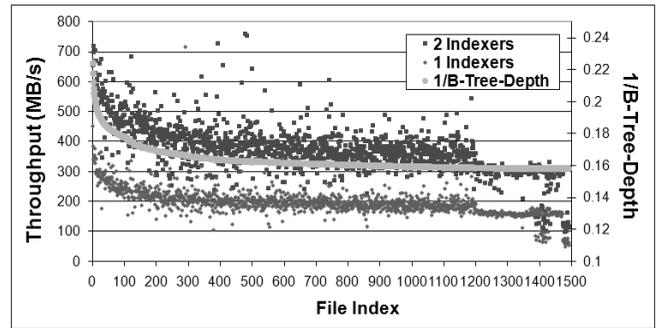
TABLE 4
PERFORMANCE COMPARISON ON DIFFERENT DOCUMENT COLLECTIONS

	ClueWeb09 1st Eng Seg	Wikipedia 01-07	Library of Congress
Throughput (MB/s)	280.12	78.29	223.76

5.3 Performance of our Algorithm on Different Document Collections

We show in Table 4 the overall throughput of our algorithm on our three document collections. For all tests, six parsers and two indexers are used to achieve the best performance. The throughput achieved on the ClueWeb09 and Library of Congress datasets is within the same ballpark. For the Wikipedia01-07 collection, the HTML tags were removed, and the remainder is just pure text. As we can see from Table 3, the uncompressed size is only 1/18th of ClueWeb09 first English segment, yet the numbers of documents and tokens are about a third compared to those of the ClueWeb09 first English segment. Hence the slower than 100MB/s throughput achieved on Wikipedia01-07 actually amounts to a very high processing speed given the large numbers of documents and tokens.

5.4 Scalability of the Cluster Algorithm Relative to the Optimized Single Node Algorithm

We use three metrics to evaluate the scalability of our cluster algorithm on a cluster with 32 nodes with multi-core processors—we measure throughput scalability by (1) increasing the number of nodes with the same overall input data; (2) increasing the number of nodes while keeping the data size fixed per node; and (3) increasing the size of the data on 32 nodes.

5.4.1 Scalability Relative to the Number of Nodes over the same Document Collection

Due to the limited size of local disks on each node, we can't store the first English segment of ClueWeb09 locally on less than four nodes for the distributed storage model and hence we measure performance on four or more nodes. In this case, Table 5 shows the overall throughput and speedup calculated relative to the best performance of the single node algorithm, with six parsers and two indexers on each node. Notice that the cluster implementation of our algorithm running on a single node has almost the same performance as the version tailored for a

single node on the storage pool model. When the number of nodes is less than or equal to eight, we achieve almost linear scalability in both storage models. With more than eight nodes, there is limited improvement under the storage pool model since a large number of nodes have to compete for the 4Gb/s external link to data server; however, the throughput of the distributed storage model continues to improve up to the maximum number of nodes available to us. In particular, the throughput on 32 nodes increases by a factor over 22 relative to the throughput of the best single node algorithm; this translates into 6.12GB/s throughput over 32 nodes.

We now take a closer look at the performance of our algorithm when the centralized storage model is used. We conduct tests that simulate the I/O behaviors of the storage using 1 to 32 nodes, and compare the execution times with those obtained by running our algorithm on the same document collection (first English segment of ClueWeb09). Two concurrent threads, one for input and the other for output, are used. Since there is a scheduler in our case to ensure that at any time at most one parser thread is reading from the disk, only a single input thread is included in the tests to just read the same document collection. After this thread reads a segment (the same 1GB as in our algorithm), the output thread will write to disk certain data of the same size as that of the postings lists produced by our algorithm. Reading and writing may occur at the same time, and hence such tests reflect the I/O pattern of our algorithm and as a result they are able to capture the peak I/O throughput of the underlying file system.

The numbers in Table 6 show that in the centralized storage model our algorithm is processing the input at almost the same rate at which the input can be read when using 8, 16 and 32 nodes. This confirms the fact that the

TABLE 5
SCALABILITY OVER THE NUMBER OF NODES WITH SAME INPUT DATA

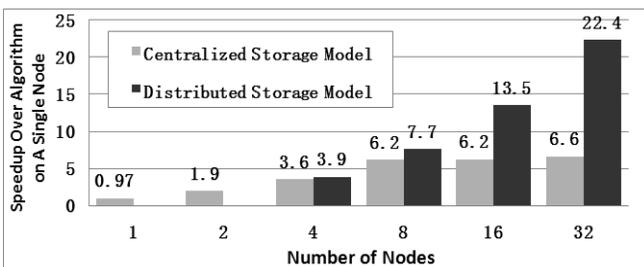| Number of Nodes	Distributed Storage Model		Centralized Storage Model	
	Throughput (GB/s)	Speedup	Throughput (GB/s)	Speedup
1	N/A	N/A	0.27	0.97
2	N/A	N/A	0.53	1.92
4	1.06	3.87	0.98	3.56
8	2.10	7.66	1.69	6.17
16	3.69	13.49	1.70	6.22
32	6.12	22.38	1.82	6.64

Fig. 15. Scalability over the Number of Nodes with Same Input Data

TABLE 6
RATIO OF THE THROUGHPUT OF OUR ALGORITHM AND PEAK I/O THROUGHPUT

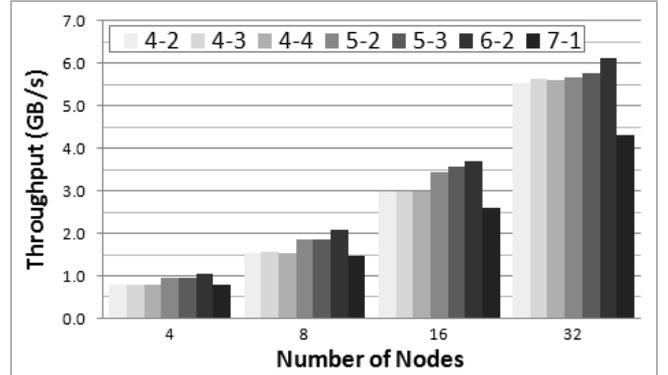| Number of Nodes	Distributed Storage Model	Centralized Storage Model
1	—	0.46
2	—	0.43
4	0.56	0.74
8	0.60	0.96
16	0.52	0.97
32	0.43	0.95

Fig. 16. Optimal Number of Parsers and Indexers on Cluster in Distributed Model

throughput of our algorithm on the storage pool model is limited by the link bandwidth when using more than 8 nodes. Note that the peak reading throughput is 350MB/s (or 2.8 GB/s), which achieves near 70% of peak performance of the 4Gb/s pipe.

On the other hand, the throughput for reading from the local disks scales linearly under the distributed storage model. However, the throughput of our algorithm is able to catch up with at least 43% of the reading throughput. Note that in our algorithm the pipeline may stall as illustrated in Fig. 8, and there exist additional costs such as sampling time, and therefore it would be difficult to achieve better ratios.

In some cases, web crawling and indexing processes may run concurrently in a streamed fashion, where a crawled document collection is expected to be immediately processed by parsers and indexers. In this streamed model, our algorithm has a clear advantage over MapReduce because in both centralized and distributed storage models, the throughput of our algorithm is close to the peak I/O bandwidth and hence document collections can be processed as fast as they are streamed.

We next examine the best combination of the number of parsers and the number of indexers for our cluster algorithm. Note that we have earlier found that 6 parsers and 2 indexers achieve the best performance on a single multi-core node. Fig. 16 shows the overall throughput of seven potential combinations of (Number of Parsers, Number of Indexers) using 4, 8, 16 and 32 nodes in distributed storage model. It is clear that the combination 6:2 achieves the best performance in all cases, and the streams are consumed at the same rate as they are produced in this case.

We now shed additional light on the extent of load balancing by comparing the relative numbers of inverted files generated on each of the 32 nodes of our cluster. On the first English segment of ClueWeb09 processed by 32 nodes, we set the average size of inverted files on a node to 1. Then the maximum size of inverted files on any node is 1.128 and the minimum is 0.834 with a standard deviation of 0.0678. This indicates a very good load balance between the 32 nodes.

5.4.2 Scalability Relative to the Number of Nodes with Fixed Data Size per Node

After placing 45GB of uncompressed document collection (part of the first English segment of ClueWeb09) on each node, we examine the scalability of our algorithm as the number of nodes increases from 1 to 32. The performance results are listed in Table 7. The execution time degrades slightly as the number of nodes increases. This degradation is to be expected since the size of document collection grows linearly with the number of nodes, and hence the dictionary becomes much larger when $P=32$ compared to the case when $P=1$.

5.4.3 Scalability over Data Size

Fig. 17 shows the scalability as a function of the input size with the algorithm running on 32 nodes. We start with the first English segment of ClueWeb09, then add the second English segment, and continue until all the ten English segments are there. The running time is a linear function of the input size with a variance of $R^2=0.9985$. This implies that our algorithm has stable throughput regardless of the collection size. Since we transfer postings lists to disks after each single run and the buffer size required by parsers is fixed, the only growing part of our pipelined algorithm is the dictionary size. As long as each local part of the dictionary can fit in the node's main memory, our algorithm is linear as a function of the input size since the

dictionary size grows very slowly after the first few runs.

5.5 Performance Results under the 1Gb/s Ethernet Interconnect

So far we have determined how to achieve balanced performance between parsers and indexers using the 10Gb/s InfiniBand as the interconnection fabric. However, such expensive network interface is not used on MapReduce clusters. To conduct a fair comparison, we perform experiments using the 1Gb/s Ethernet interconnect on our cluster.

Let's first take a look at the impact of network speed in an ideal parsing pipeline where each pipeline stage takes constant time with no idle time for each parsing thread. In our implementation we enforce that the buffer containing the parsed result must be cleared by the distributor before the parsing thread could start processing the next segment. After collecting all the parsed results from parser 1 to parser M, the distributor will send the parsed data to the appropriate destination nodes. Before this type of communication is executed, the distributor cannot collect any parsed results and hence if at this time a parser finishes the next parsing round, it has to wait until the distributor has completed its data exchange task. An example is shown in Fig. 18, where parser 1 becomes idle since it finishes its second parsing round before the end of the data exchange of the previous round.

To prevent such stalls in the pipeline, the following equation must be satisfied:

$$T_P - (M - 2)T_D \geqslant T_N$$

where T_P is the parsing time, M is the number of parsers on one node, T_D is the time to read the compressed segment from disk, and T_N is the time to distribute the parsed results over the network. On average, the processing of 1GB uncompressed ClueWeb09 data, we have $T_D = 1.6$ seconds, $T_P = 16$ seconds, and $T_N = 1.9$ seconds when the Ethernet interconnect is used, or $T_N = 0.26$ se-

TABLE 7
SCALABILITY OVER THE NUMBER OF NODES WITH FIXED
DATA SIZE PER NODE

Number of Nodes	Time (second)
1	177.40
2	183.22
4	186.19
8	195.95
16	203.90
32	232.2

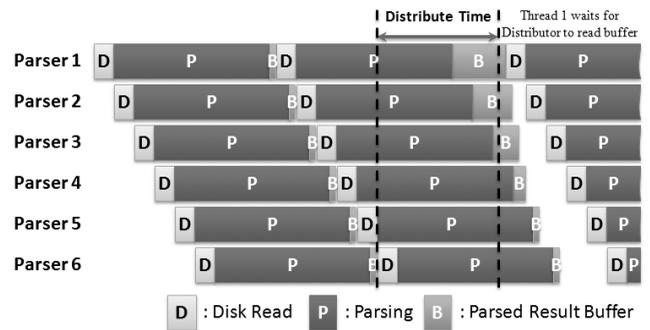Fig. 17. Scalability over the Size of Input Documents

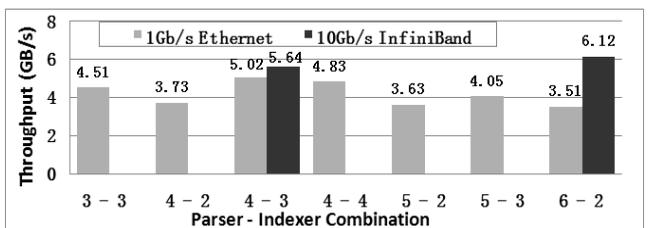Fig. 18. Impact of Distribute Time in the Ideal Pipeline

Fig. 19. Performance on 32 nodes using the 1Gb/s Ethernet

cond when the InfiniBand is used. As a result, we obtain that the number of parsers on each node has the following upper bound: $M \leq 5$ with Ethernet and $M \leq 10$ with InfiniBand. This argument presents an analysis of the impact of the network characteristics assuming that we have to achieve an ideal pipeline.

From the experimental results with 32 nodes shown in Fig. 19, we have the following results:

- The optimal combination when using Ethernet is four parsers and three indexers, which is very close to the bound $M \leq 5$ limit we calculated above;

- When the number of parsers varies from four to six, better throughput is obtained from fewer parsers when the number of indexers is fixed to either two or three;

- With four or five parsers, increasing the number of indexers from two to three leads to higher throughput because more indexers can consume data streams faster and therefore the indexing stage is less likely to impede the speed of the pipeline;

- There is no benefit from using more than three indexers and in fact it is better to let the remaining CPU cores serve the operating system and network processes rather than trying to consume non-existent parsed data.

- The optimal throughput with Ethernet is about 89% of that achieved with InfiniBand assuming the same parameter configuration and 82% of the best throughput possible with InfiniBand.

5.6 Comparison with Fastest Known MapReduce Indexers

In this section, we compare the performance of our algorithm with the best known MapReduce algorithms that appeared in the literature, namely Ivory MapReduce [9, 10] and Terrier MapReduce [8] on exactly same ClueWeb09 1st English segment data set. Both of these algorithms are implemented using the MapReduce framework, and hence the comparison is somewhat unreasonable since these are high level algorithms that do not exploit the underlying architectures. The Ivory MapReduce tests are conducted on exactly the same ClueWeb09 collection as ours using a cluster of either 99 or 280 nodes, each node having two cores. Positional postings lists are generated by the Ivory MapReduce algorithm, which will add an extra overhead. For a better comparison, we also modified our software to include positional information. In our experiments, our algorithm on single node is 7% slower and about 10% slower on the cluster, while the resulting postings lists are about 1.6 times larger. The slowdown is slightly higher on the cluster because we have to transfer more intermediate results over the cluster interconnect. According to [11], their Ivory MapReduce implementation with positional indexes is about 1.2 times slower compared to non-positional indexes, which is very close to the 1.6 times increase in postings lists size. We believe that this result is due to the fact that under the MapReduce framework, intermediate results are written to disks and shuffled in between Map and Reduce, a process that is more sensitive to the increase in data size. In our algorithm, dictionary lookup or B-tree search operation consumes the majority of CPU cycles and as a result structural changes in postings lists should not introduce a significant overhead. Parsing and indexing times are reported separately in [10], 54.3 minutes for parsing and 29.6 minutes for indexing with 280 nodes, and the throughput is calculated by dividing uncompressed size by the sum of the these two numbers. On the other hand, the Terrier MapReduce algorithm uses a cluster of 30 nodes with a total of 240 cores on the same ClueWeb09 collection. Originally while computing the throughput they used compressed data size and we've translated that into our metric using uncompressed data size. The main features of the platforms are captured in

TABLE 8
PLATFORM CONFIGURATION AND PERFORMANCE COMPARISON

		This Paper	Ivory MapReduce	Terrier MapReduce
System Details	**Processors per Node**	Two Intel Xeon 2.8GHz Quad-core CPUs	Two Intel Single-core 2.8GHz CPUs	Two AMD Quad-core Opteron CPUs
	Memory per Node	24GB	4GB	16GB
	File System	File System via 4Gb/s Ethernet or local disks	Hadoop Distributed File System	Hadoop Distributed File System
Throughput (MB/s)	**1 Node (3 cores)**	—	—	—
	1 Node (8 cores)	280	—	33
	8 Nodes (24 cores)	—	—	—
	8 nodes (64 cores)	2148 (InfiniBand) 1616 (Ethernet)	—	—
	30 nodes (240 cores)	—	—	460
	32 nodes (256 cores)	6271 (InfiniBand) 5145 (Ethernet)	—	—
	99 Nodes (198 cores)	—	167	—
	280 Nodes (560 cores)	—	289	—

Table 8.

It is clear that the throughput of our pipelined and parallel indexing algorithm using the 1Gb/s Ethernet is substantially higher even when compared to the two algorithms running on larger clusters. We note that this comparison has its significant shortcomings, but it still provides a clear indication of the effectiveness of the approach described in this paper.

The scalability of Ivory MapReduce does not seem to be linear since the throughput only increases by 73% when the number of nodes is tripled, but the improvement is still significant given the fact that hundreds of nodes are involved. On the other hand, Terrier MapReduce scales almost linearly within the range of 30 nodes.

6 CONCLUSION

We introduced a new pipelined strategy for constructing inverted files on a cluster of multicore processors, which can process documents near the peak I/O rate of the cluster. Several key elements were developed to achieve the optimized throughput, including:

- Combined pipelining and parallelism that match maximum possible parsing throughput with parallel indexing on available resources;

- A hybrid trie and B-tree dictionary data structure, in which the logical trie is implemented as a table for fast look-up and each B-Tree includes character caches to expedite term string comparisons;

- Assignment of parsed sub-streams to indexers using a random sampling preprocessing step;

- Development of a fully parallelized scheme that makes efficient use of available cores on a single node as well as across the cluster;

- Careful management of communication resulting in hiding the inter-processor communication overhead.

Our strategy significantly outperforms the best known MapReduce algorithms in the literature and achieves a throughput that is close to the peak I/O of the underlying system. This work sheds some light on the potential performance cost incurred in using the higher-level MapReduce programming model.

REFERENCES

[1] Z. Wei and J. JaJa, "A Fast Algorithm for Constructing Inverted Files on Heterogeneous Platforms", International Parallel and Distributed Processing Symposium (IPDPS2011), Anchorage, AK, May 2011.

[2] A. Moffat and T. A. H. Bell, "In situ generation of compressed inverted files", Journal of the American Society of Information Science 46(7), pp. 537-550, Aug. 1995.

[3] S. Heinz and J. Zobel, "Efficient single-pass index construction for text databases", Journal of the American Society for Information Science and Technology, vol. 54(8), pp. 713-729, June 2003.

[4] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina, "Building a distributed full-text index for the Web", ACM Transactions on Information Systems, Vol. 19(3), pp. 217-241, July 2001.

[5] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani, "Efficient distributed algorithms to build inverted files", SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, pp. 105-112, 1999.

[6] J. Dean and S. Ghemawat. "Mapreduce: Simplified data processing on large clusters". In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Dec. 2004.

[7] R. McCreadie, C. Mcdonald, and I. Ounis, "Comparing Distributed Indexing: To MapReduce or Not?", 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009.

[8] R. McCreadie, C. Mcdonald, and I. Ounis, "MapReduce indexing strategies: Studying scalability and efficiency", Information Processing and Management, 2011.

[9] J. Lin, D. Metzler, T. Elsayed, and L. Wang. "Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search". Proceedings of the Eighteenth Text REtrieval Conference (TREC 2009), November 2009, Gaithersburg, Maryland.

[10] T. Elsayed, F. Ture, and J. Lin, "Brute-Force Approaches to Batch Retrieval: Scalable Indexing with MapReduce, or Why Bother?", Technical Report HCIL-2010-23, University of Maryland, College Park, October 2010.

[11] J. Lin, personal communication, April 2011.