Pipeline and Data Parallel Hybrid Indexing Algorithm for Multi-core Platform

Suqing Zhang and Jirui Li

Information Engineer Department,Henan Vocational and Technical Institute, Zhengzhou, China
280946342@qq.com, ljrokyes@163.com

Abstract. The scale and growth rate of today's text collection bring new challenges for index construction. To tackle this problem, Pipeline and Data Parallel Hybrid Algorithm (PDPH), is proposed to improve the indexing performance for multi-core platform. Compared to existing sequential indexing algorithms, Pipeline and data parallelism are introduced by the PDPH to improve the algorithm flexibility and scale the performance with more cores. Evaluations showed this algorithm can improve index construction speed for multi-core platform.

Keywords: Multi-core platform, Indexing Algorithms, Data Parallel, PHPD.

1 Introduction

The world's data is increasing at an astonishing rate. The scale and growth rate of text collection bring new challenges for index construction. Building an index for a large text collection may involve parsing billions of documents, handling millions of distinct words, and processing billions of occurrences of words in the text. Text collections have become so large and are growing so rapidly that traditional indexing schemes become unmanageable, requiring huge resources and taking days to complete. More powerful computing resources and more efficient algorithms are needed to tackle this problem. T

With the growth of data, the computing power also increases according to Moore's law. Computing power is increasing because of the multi-core technology. Multi-core technology packs two or more execution cores into a single processor so a single chip can provide multiple execution resources. Multi-core architecture is in essence a divide-and-conquer strategy. By divvying up computational work and then spreading it over multiple execution cores, a multi-core processor can perform more work in a given clock cycle than a traditional single core processor. Multi-core processors provide thread-level parallelism. However, to make full use of thread-level parallelism, an application should be threaded so that it can spread its workload across multiple execution cores.

In a multi-core system, there are usually several execution cores, shared memory and disks. However, those existing sequential indexing algorithms, which are mostly

K. Liu et al. (Eds.): ICISO 2014, IFIP AICT 426, pp. 272–280, 2014.

single-threaded, treat the multi-core system the same as the traditional single-core system and cannot make use of the multiple execution cores. Also, semiconductor manufacturers of multi-core processors choose to scale back the clock speed so that the chips run cooler, so the performance of single-threaded sequential indexing algorithms in a multi-core system will decrease a little compared with the traditional single-core environment.

We present an efficient indexing algorithm that can be deployed on multi-core systems. It is the Pipeline and Data Parallel Hybrid (PDPH) algorithm. In addition to employing a pipeline, the PDPH algorithm also introduces data parallelism into the indexing process. This algorithm has good performance. It is also more scalable than the existing sequential algorithms.

2 Sequential Indexing Algorithms

There are several proposed algorithms to construct index files(inverted files). The Simple In-Memory algorithm keeps all index data in the main memory. It is only suitable for small text collections. The Disk-Based algorithm makes use of temporal files in order to reduce the need for the main memory. However, because there are many random disk accesses, the Disk-Based approach is too slow for large collections. The Two-Pass In-Memory approach introduces compression to limit temporal disk space usage, but it uses two passes over the collections that means it needs to traverse the text collection twice. For large text collections, for example a terabyte scale collection, just traversing the whole collection will take a long time. Both of the Sort-Based algorithm and the Single-Pass algorithm are scalable methods. They can be used for text collections with any size and can work with limited main memory. However, because the Single-Pass algorithm stores compressed index data in memory, it makes better use of memory than the Sort-Based algorithm. Because of this, the Single-Pass algorithm is the most efficient sequential indexing algorithm.

The Single-Pass algorithm travels only one pass over the collection. The Single-Pass algorithm maintains a lexicon for distinct terms of the collection in memory first. Each term in the lexicon is assigned a dynamic in-memory bit-vector for its inverted list. The bit-vector is used to accumulate a term's corresponding postings in a compressed format. Each document is read into the main memory and then parsed into postings. For each posting delivered from the parsing stream, a lookup for its corresponding term in the lexicon is made. If the term does not exist in the lexicon, the term is inserted into the lexicon and the corresponding bit-vector is allocated and initialized. The posting is inserted into the bit-vector and compressed on the fly. The process is repeated as long as the main memory is available. When the main memory is used up, the terms and their inverted lists in the lexicon are written to a temporary disk index in lexicographical term order. The allocated space for terms is freed and the process repeats until all the documents in the collection are processed. When all the documents are processed, there could be several temporary disk indices. These indices should be merged into a single inverted index for fast query. Suppose the number of indices is N; then an N-way merge requires only one merging pass over the

N indices. To avoid excessive disk costs, an in-memory input buffer is assigned to each of the N indices. During merging, inverted lists are processed in lexicographical order. Inverted lists are decompressed, re-compressed, and merged into the final inverted list.

The merged lists can be written to a new file directly. In this case, the Single-Pass algorithm requires temporary disk space more than twice the size of the final inverted file. To save disk space, the Single-Pass algorithm can also write merged lists back in place into the temporary disk index. However, the temporary disk space is saved at the cost of indexing time. The Single-Pass algorithm is shown in Fig. 1.

Fig. 1. Structure of the Single-Pass algorithm

3 Pipeline and Data Parallel Hybrid algorithm

3.1 Algorithm Design

The PDPH algorithm is a pipeline algorithm. The indexing process is divided into four stages: the loading stage, the processing stage ,the flushing stage and the merging stage. Loading, processing and flushing are executed in order so they form a pipeline. The loading stage loads documents into the document buffer from the disk or network. In the processing stage, we launch multiple threads for processing. The works of these processing threads are the same: The segmentation stage is designed especially for Chinese, Japanese and Korean text collections. During this stage, sentences are segmented into words. This stage is useful only when the text collection contains Chinese, Japanese or Korean text because only sentences in these three languages do not have any delimiters between words. The parsing stage parses documents and tokenizes documents into terms. During the parsing stage, postings are extracted and fed into the following stage - the compression stage. The compressing stage processes the posting stream, accumulating lists in the main memory in a compressed format. Each thread accumulates the compressed inverted lists in the memory individually. We refer to the compressed inverted lists maintained by a thread as a memory index. When the memory is exhausted, the memory indices of all processing threads are written onto the disk as a temporal disk index during the flushing stage. During merging stage, those temporal disk indices are merged into a single memory index and then written onto the disk. The PDPH algorithm is shown in Fig. 2.

Among these stages, the loading and the flushing stages are I/O-intensive. the disk conflict between these two stages is not serious. The segmentation stage, the parsing stage and the compression stage are all CPU-intensive.

All processing threads do the same work. If they are fed with the same amount of data, their run times should be approximate to each other. In a multi-core system with n execution cores, we can launch n processing threads and assign each processing

thread to a core. When the core number increases, the PDPH algorithm still can make use of the multiple computing resources by increasing the processing thread number. It is also suitable for optimization. Optimized parsing or compression can improve the performance of the algorithm.

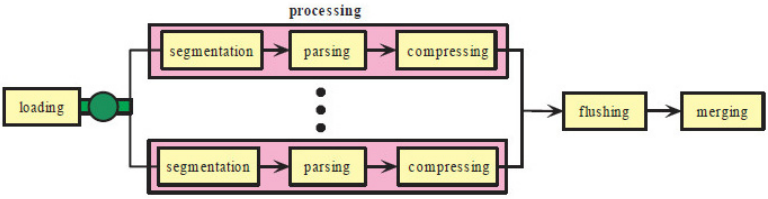

Fig. 2. Structure of the PDPH algorithm

For a threaded program, communication between threads is a critical issue for the performance of the program. Main memory buffer is used for thread communication. Generally speaking, the size of the memory buffer is important to the communication efficiency. If the buffer size is too small, it will decrease the communication efficiency. If the buffer size is too big, the buffer will take up too many memory resources. So the goal of the buffer design is to maximum the performance with a minimum buffer size. In the evaluation section we will show that we can get good performance in the cost of a moderate size buffer.

Synchronization is another critical issue for a threaded program. Since the pipeline stages are in sequence, we only have to handle synchronization for adjacent stages. Lock is a common mechanism for thread synchronization. Lock mechanism should be carefully designed because it not only affects the performance of the program, but more important, it affects the correction of the program.

Careless lock design may cause a very common problem in multi thread program - dead lock. Since adjacent stages communicate by the memory buffer, we can handle thread synchronization in the memory buffer, for example, making the buffer operation thread-safe. When a thread is operating in a buffer, other threads which want to access the buffer at the same time will be blocked.The granularity of the lock is important to performance. If the granularity is too big, other threads will wait a long time to grab a lock. If the granularity is too small, threads will grab and release lock more frequently and introduce much overhead. For simplicity, in our implementation, the granularity is a document. That means to put a document in the buffer or get a document from the buffer, a thread has to grab and release the lock one time. Of course some other sophisticated lock mechanisms will provide better lock performance, but we can see that even with this simple lock design, the PDPH algorithm will outperform the Single-Pass algorithm a lot.

3.2 Experimental Evaluations

We used three text collections to test our algorithm. The statistics of these collections were shown in Table 1. These three collections are drawn from the Terabyte track in

the TREC 2011. The Terabyte track consists of a collection of Web data crawled from Web sites in the .gov domain during early 2011. This collection ("GOV2") contains a large proportion of the crawlable pages in .gov, including HTML and text, plus the extracted text of PDF, Word, and Postscript files. The GOV2 collection is 426GB in size and contains about 25 million documents. Collection 1, Collection 2 and Collection 3 were disjointed subsets of the GOV2 collection. Since the test collections are English text collection, so we omit the segmentation stage in our indexing process.

Table 1. Collection description

	Collection 1	Collection 2	Collection 3
Size	5.5GB	22GB	40GB
Documents	349900	13,43,092	2,214,327
Distinct terms	3,448,052	10,016,184	13,686,308
Term occurrences	261,373,711	1,086,543,215	2,112,867,468
Average file size	17KB	17KB	19KB

The test machine had two Intel Woodcrest 2.66GHz CPUs. Each CPU had four cores, so there were eight cores in the system. However, one of the eight cores had a defect, so we only use the other seven cores in the system. There was 2GB memory in the system and we used 1.5GB memory for constructing the inverted files. The disk was an Ultra320 SCSI disk. And the text collection and the inverted files were placed on the same disk. The operating system running on the test machine is a Linux operating system with kernel 2.4.22.

We measured the indexing performance when different numbers of processing threads were launched for a certain number of cores. Each processing thread maintained a memory index. When the main memory was used up, all of the memory indices were merged together and then flushed to the disk as a temporal disk index. For comparison purposes, we also measured the performance of the Single-Pass algorithm. The result of test is shown in Table 2.

The Single-Pass program is a modification of indri 2.8, which is an efficient indexing and searching engine. The Single-Pass program is composed of three threads. The first thread each time loads a document into the memory, parsing it into postings and compressing the postings. The first thread keeps running until there is no free memory space. Then the first thread is paused and the flushing thread is activated. The flushing thread writes compressed inverted lists in memory onto the disk as a temporal disk index and then it frees the memory. When there is available memory space, the first thread is awakened up and continues to run. When all documents in the text collection are processed, the merging thread merges all temporal disk indices into a final inverted index. The indexing process in the Single-Pass program has a slight difference from the standard Single-Pass algorithm. In the standard Single-Pass algorithm, Golomb codes and Elias codes are used to compress

Table 2. Elapsed time in seconds to construct inverted files with the PDPH algorithm

Case	Algorithm \ Cores	1	2	3	4	5	6	7
Collection 1	Single-Pass	400						
	PDPH1	296	259	260				
	PDPH2	397	261	210	224			
	PDPH4	408	266	242	236	224	220	
	PDPH6	401	264	239	229	233	228	221
	PDPH8	418	264	237	236	233	233	236
Collection 2	Single-Pass	1680						
	PDPH1	1285	1090	1060				
	PDPH2	1743	1130	951	913			
	PDPH4	1720	1136	1035	969	944	937	
	PDPH6	1706	1125	1027	968	947	943	933
	PDPH8	1723	1146	1026	989	971	963	952
Collection 3	Single-Pass	3112						
	PDPH1	2299	1981	1979				
	PDPH2	3186	2048	1736	1692			
	PDPH4	3163	2044	1895	1806	1707	1670	
	PDPH6	3171	2108	1896	1797	1783	1748	1725
	PDPH8	3150	2073	1854	1773	1732	1749	1740

postings. However, use of byte-aligned codes or word-aligned codes can reduce the query evaluation time compared to the Golomb or Elias codes. The overhead of byte-aligned codes is only a modest amount of temporal disk space. Since the word-aligned codes are more complex but have similar performance with byte-aligned codes, for the simplicity, the byte-aligned code is adopted to compress postings instead of the Golomb and Elias codes in the Single-Pass program.

The PDPH algorithm needs two buffers in the memory: the original document buffer and the the parsed document buffer. The loading thread loads documents into the original document buffer. The parsing thread fetches documents from the original document buffer, parsing these documents and filling the parsed documents into the parsed document buffer.Since the average file size of these three collections is less than 20KB, we test buffer size 512KB, 1MB and 10MB and find that they all have similar performances. Besides, we also generate some bigger documents by aggregating some small documents. The average file size of these bigger documents is 17MB. Then we test buffer size 20M, 50M, 100M and 200M and find that they also have similar performances. So we can achieve good performance in the cost of a little memory for buffering.

In Table 2, PDPHn (n = 1...8) means n processing threads were launched for the processing stage. For example, PDPH1 means there was only one processing thread. When only one core was available, PDPH1 outperformed the Single-Pass algorithm by 24%. For PDPHn (n>1), their indexing times were close to the Single-Pass algorithm. The reason is that the benefit of the pipeline was offset by the overhead introduced by the context switching and thread synchronization.

When there were two cores available, the indexing times for PDPH1 and PDPHn were very close. The performance improvement was about 32%. When three cores

were used, the PDPH1 had no further performance improvement. For PDPH2, the performance improvement compared to the Single-Pass algorithm was about 44%. Fig.3 shows the running time of each stage in PDPH2 when three cores were used to construct the inverted files for collection 3. Processing1 and processing2 were the two processing threads. The running times of these two threads were almost the same. The loading stage was the stage with the longest running time, so the pipelining time was approximate to the loading time.

For PDPHn (n>2), when the number of cores was increased from two to three, their performances was also improved, but the improvement was not as significant as the improvement of PDPH2. This was because the number of threads in PDPH2, which require large amount of processor resources, matched the number of cores in the system. In PDPH2, in addition to the two processing threads, the loading thread also required many processor cycles, so there were three threads that had a large amount of computing work. There were exactly three cores in the system, so each thread could be served by an individual core. In PDPHn (n>2), there were at least four threads which represented heavy computing work. Context switching brought some overhead, so PDPH2 outperformed PDPHn (n>2) when only three cores were available. More generally, we also can conclude that if there are n cores in the system, PDPH (n-1) will outperform PDPHk (k > n).

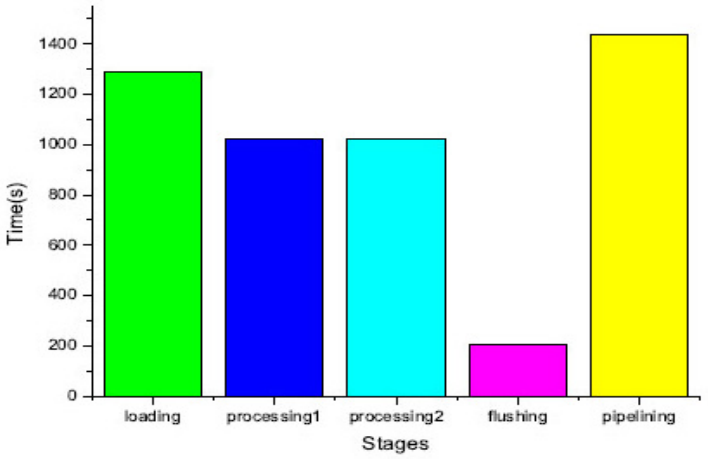

Fig. 3. Running time of each stage in PDPH2 when three cores are used

When the number of cores is increased from 4 to 7, the performance of PDPHn (n=1...8) did not change much. This was also the result of the pipeline. Fig.3 shows that the loading stage is the most time-consuming stage. When the number of cores was increased, the processing times decreased, but the loading time was left unchanged. The loading time hid the processing times and the pipelining time was approximate to the loading time, so even if the processing times were totally eliminated, the pipelining time would not change much. In order to improve the scalability of the PDPH algorithm, I/O optimization is a critical issue.

We compared the Single-Pass algorithm, and PDPH algorithm in Fig.4. As can be seen, the FDPH algorithm can greatly save time than the serial algorithm in multi-core environment.

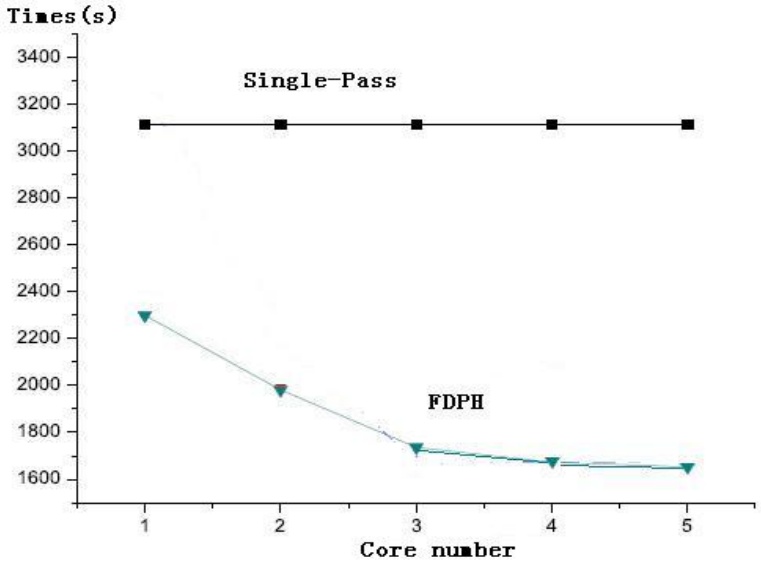

Fig. 4. Performance and scalability of the four algorithms

4 Conclusions

In multi-core environments, traditional sequential indexing algorithms cannot make use of all the cores in a system. They are also not scalable when the number of cores increases. In this paper, we present an efficient indexing algorithm for multi-core systems: the PDPH algorithm. The PDPH algorithm divides the indexing process into pipeline stages. However, it does not divide the computing work into several stages. Instead, the computing work is kept in one single stage but data parallel is introduced so this computing stage will run in parallel on several execution cores. The I/O operation is kept in one stage and it also can overlap with the computing stage. The PDPH algorithm can achieve good performance. When one, two, three or four cores are used, the performance improvement is 26%, 36%, 44% or 46%.

References

1. Anh, V.N., Moffat, A.: Inverted Index Compression Using Word-Aligned Binary Codes. Information Retrieval, 151–166 (2005)
2. Anh, V.N., Moffat, A.: Improved Word-Aligned Binary Compression for Text Indexing. IEEE Transactions on Knowledge and Data Engineering 18, 857–861 (2006)

3. Trotman, A.: Compressing Inverted Files. Information Retrieval, 5–19 (2003)
4. Heinz, S., Zobel, J.: Efficient single-pass index construction for text databases. Journal of the American Society for Information Science and Technology, 713–729 (2003)
5. Yue, M., Li, W.: Dynamic indexing for large-scale collections. Journal of Beijing Normal University (Natural Science), 134–137 (2009)
6. Ling, S., Xue-jun, Y., Lan, M.: Research on Data Organization and Index of EMMDB. Journal of Frontiers of Computer Science & Technology, 742–748 (2010)
7. Dejiao, N., Tao, C., Yong-zhao, Z., Shiguang, J.: Hierarchical metadata indexing algorithm of mass storage system. Application Research of Computers, 510–513 (2010)
8. Feng, W.: Study of XML Search Engines Based on Document Type Definition. Journal of Anhui Science and Technology University, 35–39 (2010)
9. Yue, Z., Hao-min, Y., Qi, Z., Xuan-jing, H.: Distributed Index for Near Duplicate Detection. Journal of Chinese Information Processing, 91–97 (2011)