

## ABSTRACT

Title of Document: HIGH-PERFORMANCE COMPUTING  
ALGORITHMS FOR CONSTRUCTING  
INVERTED FILES ON EMERGING  
MULTICORE PROCESSORS

Zheng Wei, Doctor of Philosophy, 2012

Directed By: Professor Joseph F. JaJa, Department of  
Electrical and Computer Engineering

Current trends in processor architectures increasingly include more cores on a single chip and more complex memory hierarchies, and such a trend is likely to continue in the foreseeable future. These processors offer unprecedented opportunities for speeding up demanding computations if the available resources can be effectively utilized. Simultaneously, parallel programming languages such as OpenMP and MPI have been commonly used on clusters of multicore CPUs while newer programming languages such as OpenCL and CUDA have been widely adopted on recent heterogeneous systems and GPUs respectively. The main goal of this dissertation is to develop techniques and methodologies for exploiting these emerging parallel architectures and parallel programming languages to solve large scale irregular applications such as the construction of inverted files.

The extraction of inverted files from large collections of documents forms a

critical component of all information retrieval systems including web search engines. In this problem, the disk I/O throughput is the major performance bottleneck especially when intermediate results are written onto disks. In addition to the I/O bottleneck, a number of synchronization and consistency issues must be resolved in order to build the dictionary and postings lists efficiently. To address these issues, we introduce a dictionary data structure using a hybrid of trie and B-trees and a high-throughput pipeline strategy that completely avoids the use of disks as temporary storage for intermediate results, while ensuring the consumption of the input data at a high rate. The high-throughput pipelined strategy produces parallel parsed streams that are consumed at the same rate by parallel indexers.

The pipelined strategy is implemented on a single multicore CPU as well as on a cluster of such nodes. We were able to achieve a throughput of more than 262MB/s on the ClueWeb09 dataset on a single node. On a cluster of 32 nodes, our experimental results show scalable performance using different metrics, significantly improving on prior published results.

On the other hand, we develop a new approach for handling time-evolving documents using additional small temporal indexing structures. The lifetime of the collection is partitioned into multiple time windows, which guarantees a very fast temporal query response time at a small space overhead relative to the non-temporal case. Extensive experimental results indicate that the overhead in both indexing and querying is small in this more complicated case, and the query performance can indeed be improved using finer temporal partitioning of the collection.

Finally, we employ GPUs to accelerate the indexing process for building inverted

files and to develop a very fast algorithm for the highly irregular list ranking problem. For the indexing problem, the workload is split between CPUs and GPUs in such a way that the strengths of both architectures are exploited. For the list ranking problem involved in the decompression of inverted files, an optimized GPU algorithm is introduced by reducing the problem to a large number of fine grain computations in such a way that the processing cost per element is shown to be close to the best possible.

HIGH-PERFORMANCE COMPUTING ALGORITHMS FOR CONSTRUCTING  
INVERTED FILES ON EMERGING MULTICORE PROCESSORS

By

Zheng Wei

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park, in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2012

Advisory Committee:  
Professor Joseph F. JaJa, Chair  
Professor Rajeev Barua  
Professor Shuvra S. Bhattacharyya  
Professor Ankur Srivastava  
Professor Jimmy Lin

© Copyright by  
Zheng Wei  
2012

## Dedication

*To Lu, Elise and my parents.*

## Acknowledgements

I would like to express my greatest gratitude to my advisor, Dr. Joseph F. JaJa, for his guidance, patience and support during my four years' graduate study in University of Maryland. I felt incredibly lucky to be part of his several interesting and productive projects. His broad perspective and valuable experience have escorted me through many difficult times and trained me to look at problems at a high level, maintain the highest standard in research and future professional careers. I have benefited a lot from discussion with him on research problems and greatly improved my skills on critical thinking as well as paper organizing and writing.

I would also like to thank my dissertation examining committee members, Dr. Jimmy Lin, Dr. Shuvra S. Bhattacharyya, Dr. Ankur Srivastava, Dr. Gang Qu and Dr. Rajeev Barua, for their time and effort to serve on the committee and valuable questions and comments to improve the quality of the dissertation.

I'm grateful that I have wonderful group members. Dr. SangChul Song gave me insights to the Information Retrieval area, great software programs with ready-to-use data sets, and amazing materials to prepare job interviews without which I might not get the internship opportunity and then the current job offer. Mike Smorul has been always there to address my problems with the clusters, storage, Linux system and other special requirements. Chi Cui offered very useful advice on CUDA GPU programming as well as the Ph.D. study here in University of Maryland. I especially thank Jing Wu, both as an office mate and a good friend, for her precise help throughout my four years here on my work and my family. And the life in Maryland won't be cheerful without so many fellow students: Zheng Zhou, Hsiang-Huang Wu,

Bing Shi and Wenjun Lu.

Finally, I give my special thanks to my family. My wife, Lu Chen, has been so supportive to whatever I decided to do and even if it meant a lot sacrifice at her side. I believe that she would have become a great journalist, or would have finished her Ph.D. much earlier, but she selflessly put my situation in the first place without any complaint. Her love, encouragement and endurance are critical to the completion of my dissertation and degree. My parents and parents-in-law are always there for us with deep love and support. In the past year we have been very stressful with our new family member, cute little Elise. I could not have devoted so much time on my dissertation and finish it in time if our parents did not come to help us with the house work and taking care of baby. Elise, my hope and love, brings us priceless joy and wonderful parenting experience with her smiles, first uttering of “papa” and “mama”, and every new progress.

# Table of Contents

Dedication.....	ii
Acknowledgements.....	iii
Table of Contents.....	v
List of Tables.....	vii
List of Figures.....	viii
Chapter 1 Introduction.....	1
1.1 Overall Background.....	1
1.2 Web Search.....	3
1.3 Indexing Strategies and MapReduce.....	6
1.4 Temporally Anchored Web Contents and Inverted Files.....	8
1.5 Major Contributions of this Thesis.....	9
Chapter 2 Fast Construction of Inverted Files on a Single Node.....	13
2.1 Overview.....	13
2.2 Related Work.....	14
2.3 Description of Our Algorithm.....	17
2.3.1 Overall Approach.....	18
2.3.2 Dictionary Data Structure.....	19
2.3.3 Structure of Parallel Parsers.....	23
2.3.4 Structure of Parallel Indexers.....	25
2.3.5 Overall Pipelined Data Flow.....	28
2.4 Experimental Results.....	30
2.4.1 Optimal Number of Parallel Parsers and Indexers.....	32
2.4.2 Indexing Throughput and Dictionary Growth.....	34
2.4.3 Performance of our Algorithm on Different Document Collections.....	35
2.4.4 Comparison with Fastest Known Algorithms.....	36
2.5 Conclusion.....	38
Chapter 3 Fast Construction of Inverted Files on a Cluster.....	39
3.1 Overall Approach on a Cluster.....	39
3.2 Our Extended Approach.....	41
3.2.1 Storage Model.....	41
3.2.2 Load Balancing among the Nodes.....	42
3.2.3 Communication between Parsers and Indexers.....	43
3.2.4 Overall Data Flow.....	45
3.3 Experimental Results.....	46
3.3.1 Scalability of the Cluster Algorithm.....	47
3.3.2 Performance with Different Types of Interconnect.....	53
3.3.3 Comparison with Fastest Known Algorithms.....	55
3.4 Conclusion.....	57
Chapter 4 Construction of Temporal Inverted Files.....	59
4.1 Overview of Temporal Inverted Files.....	59
4.2 Related Work.....	61
4.3 Problem Definition and Temporal Strategy.....	63
4.3.1 Problem Definition.....	63

4.3.2 Overall Strategy for Temporal Indexing and Searching .....	64
4.3.3 Query Strategy .....	69
4.4 Experimental Evaluation.....	71
4.4.1 URI Hash Table and VID Table: Implementation Details.....	72
4.4.2 Growth of URI Hash Table and VID Table.....	75
4.4.3 Relationship between Query Performance and the number of Time Windows .....	76
4.4.4 Relationship between Query Performance and Different Time Window Strategies.....	80
4.5 Parallel Algorithms for Constructing the Inverted Files in the Temporal Case	82
4.5.1 Algorithm on a Single Multicore Node .....	82
4.5.2 Algorithm on a Cluster.....	86
4.6 Experimental Results of Our Algorithm .....	91
4.6.1 Number of Parallel Parsers, Indexers and Temporal Mergers on a Single Node.....	91
4.6.2 Throughput on the Cluster .....	93
4.7 Conclusion .....	99
Chapter 5 Applications Accelerated by GPU .....	101
5.1 GPU and CUDA Overview.....	101
5.2 GPU Indexers on a Single Node.....	106
5.2.1 Heterogeneous Implementation of Indexing.....	106
5.2.2 Experimental Results .....	111
5.2.3 Conclusion .....	117
5.3 Optimized List Ranking Algorithm on GPUs.....	118
5.3.1 Overview.....	118
5.3.2 Problem Definition and Related Work .....	119
5.3.3 CUDA Implementation of Prefix Sum Computation.....	120
5.3.4 Experimental Results .....	126
5.3.5 Conclusion .....	141
5.4 Conclusion .....	142
Chapter 6 Concluding Remarks and Future Perspectives.....	143
Bibliography .....	147

## List of Tables

Table 2.1: Trie-Collection Index Definition .....	21
Table 2.2: Data Structure of One B-Tree Node .....	23
Table 2.3: Statistics of Document Collections.....	31
Table 2.4: Performance Comparison on Different Document Collection .....	36
Table 2.5: Platform Configuration Comparison .....	37
Table 3.1: Statistics of the ClueWeb09 English Document Collections .....	46
Table 3.2: Scalability over the Number of Nodes with Same Input Data.....	48
Table 3.3: Ratio of the Throughput of Our Algorithm and Peak I/O Throughput.....	49
Table 3.4: Scalability over the Number of Nodes with Fixed Data Size per Node ....	52
Table 3.5 Platform Configuration and Performance Comparison .....	56
Table 4.1: Statistics of Document Collections.....	72
Table 4.2: Throughput and Output Size over the number of Time Windows .....	77
Table 4.3: Query Process Time in Step 2 .....	78
Table 4.4: Query Performance as a Function of the Number of Time Windows .....	79
Table 4.5: Size of Output Size with Different Time Window Splitting Strategies.....	81
Table 4.6: Query Performance with Different Time Window Splitting Strategies ....	82
Table 4.7: Throughput Over the Number of Nodes .....	94
Table 4.8: Speedup in the Merging Stage.....	96
Table 4.9: Throughput over the Number of Time Windows .....	97
Table 4.10: Platform Configuration and Performance Comparison .....	98
Table 4.11: Throughput on Different Document Version Collections .....	99
Table 5.1: Overall memory organization of the Tesla C1060. ....	103
Table 5.2: Comparison between GT200 and Fermi.....	104
Table 5.3: Performance Comparison on Different Indexers with Six CPU Parsers .	112
Table 5.4: Work Load between CPU and GPU .....	113
Table 5.5: Comparison between Five Different GPUs on Processing Same Data ...	115
Table 5.6: Normalized Throughput with Single Multiprocessor .....	116
Table 5.7: Detailed Running Times of Our Algorithm as a Function of the List Size .....	132
Table 5.8: Time per element comparison between prefix sum and global memory testing program on the Tesla C1060 (Stride = 1001).....	135
Table 5.9: Performance comparison of cost per element between prefix sum and global memory testing program using extremely large strides on the Tesla C1060.	136
Table 5.10: Configuration comparison of Tesla C1060, Tesla C2050 and GTX480	137
Table 5.11: Performance comparison of Three Different GPUs on 64M Random List .....	138

## List of Figures

Figure 2.1: Dataflow of Pipelined and Parallel Indexing .....	19
Figure 2.2: Hybrid of Trie and B-Tree Structure of Dictionary .....	20
Figure 2.3: Data Flow of One Parser Thread .....	24
Figure 2.4: A B-tree Corresponding to a Single Trie Collection Index .....	26
Figure 2.5: Work Assignments among Multiple Threads .....	27
Figure 2.6: Data Flow of One Single Run on Parallel Indexers .....	28
Figure 2.7: Pipelined Data Flow of Overall Indexing System on a Single Node .....	28
Figure 2.8: Timing Sequence of Parallel Parsers .....	29
Figure 2.9: Optimal Number of Parallel Parsers and Indexers .....	33
Figure 2.10: Scalability of Parallel Indexers .....	35
Figure 3.1: Data Flow of Partition-and-Index Strategy .....	40
Figure 3.2: Data Flow of Parallel Indexers on One Node in the Cluster .....	43
Figure 3.3: Message Construction by Distributor .....	44
Figure 3.4: Overall Data Flow in the Cluster .....	45
Figure 3.5: Scalability over the Number of Nodes with Same Input Data .....	48
Figure 3.6: Optimal Parameters on the Cluster in Distributed Model .....	51
Figure 3.7: Scalability over the Size of Input Documents .....	52
Figure 3.8: Impact of Distribute Time in the Ideal Pipeline .....	53
Figure 3.9: Performance on 32 nodes using the 1Gb/s Ethernet .....	54
Figure 4.1: An Example of Temporal Indexing .....	67
Figure 4.2: Example of the VID Table Data Structure .....	74
Figure 4.3: Growth of URI Hash Table and VID Table .....	75
Figure 4.4: Impact of Time Window Numbers on Index Size and Query Performance .....	80
Figure 4.5: Overall Data Flow on a Single Node .....	84
Figure 4.6: Data Flow in the Temporal Merging Stage .....	84
Figure 4.7: Data Flow of Partition-by-Trie-Collection-and-URI Strategy .....	88
Figure 4.8: Throughput on A Single Node .....	92
Figure 4.9: Detailed Timeline in the Pipeline with Three Time Windows .....	93
Figure 4.10: Scalability over the Number of Nodes .....	95
Figure 4.11: Detailed Timeline on the Cluster .....	96
Figure 5.1: Architecture of the Tesla C1060 .....	102
Figure 5.2: Architecture of the TPC in the Tesla C1060 .....	103
Figure 5.3: Architecture of one SM in Fermi GPU .....	104
Figure 5.4: String Representation: Term Length in the First Byte .....	107
Figure 5.5: Parallel Comparison in One GPU Thread Block .....	108
Figure 5.6: Data Flow of One Single Run on Parallel CPU and GPU Indexers .....	110
Figure 5.7: Optimal Number of Parallel Parsers and Indexers .....	111
Figure 5.8: Input array and the corresponding output for list ranking .....	122
Figure 5.9: Performance on different types of lists with 64M nodes on the Tesla C1060 .....	128
Figure 5.10: Performance of Step 3 and overall algorithm as a function of $s$ on 64M stride list (stride = 1001) on the Tesla C1060. Upper curve represents the maximum load on an SM as a function of $s$ .....	129

Figure 5.11: Performance of our algorithm on a 64M stride list (stride = 1001) as a function of the number of blocks on the Tesla C1060 .....	131
Figure 5.12: Scalability to list sizes on Stride lists (stride = 1001) on the Tesla C1060 .....	133
Figure 5.13: Time per element comparison between prefix sum and global memory testing program on the Tesla C1060 (Stride = 1001).....	135
Figure 5.14: Time per element comparison between prefix sum and global memory testing program with extremely large strides on the Tesla C1060.....	136
Figure 5.15: Performance comparison on Tesla C1060, Tesla C2050 and GTX480	137
Figure 5.16: Comparison of different platforms on Random List with 8M nodes ...	139
Figure 5.17: Performance of the scan operation and our prefix sum algorithm on 64M ordered list .....	140

# Chapter 1 Introduction

The continuously growing computational powers of multicore CPUs and manycore graphic processing units (GPUs) offer unprecedented opportunities for speeding up demanding large scale applications. To make use of the powerful underlying architectures, novel algorithms are often required to exploit the substantial available resources while avoiding the overhead due to communication, synchronization, and data transfers. In this dissertation, we consider a number of platforms: a single node multicore CPU; a heterogeneous node consisting of a multicore CPU and multiple GPUs; and a cluster of such nodes, and develop techniques for mapping irregular applications on such platforms.

## 1.1 Overall Background

Challenges in power and thermal constraints and in increasing the throughput of the traditional microprocessor architectures prohibit higher performance from single-core processors. Hence current high-end processor designs have switched to integrating more cores on a single chip [21]. Meanwhile more advanced features such as shared L2 or L3 caches between different cores and integrated memory controllers, have been introduced to increase the high memory throughput. While the architectures of current and emerging multiprocessors vary significantly, they all include several levels of memory hierarchy, SIMD or vector type operations, and multithreading. Such multicore processors offer unprecedented opportunities for speeding up demanding computations even on a single machine if the available resources can be effectively used.

Simultaneously, GPUs have substantially gained in popularity as accelerators for large scale data parallel computations. First appeared in the early and middle 1990s to relieve CPUs from heavy duty 2D/3D graphics workloads, GPUs have been pursuing highly parallel architectures consisting of many simple cores designed to operate in a streaming processing style. Modern GPUs are orders of magnitude faster than CPUs on a variety of applications [20, 34, 62, 82] due to the integrated hundreds or thousands of cores running in parallel and due to high memory bandwidth. With the introduction of the CUDA programming language, GPUs can now be programmed to handle any application. A number of works [18, 67, 81] have been published for handling the inverted files on GPUs, which are related to this thesis.

On the other hand, parallel programming models and languages have also been evolving to take full advantage of the underlying parallel hardware components as well as to relieve programmers from low level details thereby simplifying software development and debugging [35, 65]. Within a single multicore CPU in which the main memory is shared among the cores, programming languages such as POSIX Threads (Pthreads) and OpenMP [16] have been commonly used where concurrent processes and threads communicate by accessing the shared memory [1]. OpenMP provides a simple programming model for parallelization at a coarse-grained level while Pthreads support multithreading through a number of integrated primitives [15]. On a cluster where the nodes are interconnected via an interconnection network, the Message Passing Interface (MPI) is widely used as an effective communication protocol to move data between the memories on different machines. The situation is more complicated when it comes to programming GPUs. Compute Unified Device

Architecture (CUDA) is the graphics processing programming model that is tailored for the NVIDIA GPUs. Low level explicit management such as deciding on the numbers of blocks and threads, conflict-free shared memory accesses as needed, and coalesced on-board memory accesses require knowledge of relatively low-level details of the GPUs. The parallel programming languages mentioned above may need to be combined together to handle applications running on clusters of heterogeneous multicore nodes. For example, three of the top five most powerful supercomputers as of November 2011 [66] exploit large clusters of heterogeneous platforms including both multicore CPUs and GPUs.

## 1.2 Web Search

Today we find local restaurants, sports game schedules, news feeds, current weather, and stock prices by simply typing the right terms on a device followed by launching a web search engine, which collects and organizes information over the internet to answer our queries in real time. Web searches are indeed becoming an integral part of our daily lives, and it was reported that over 570 million queries were carried out per day in April 2012 [69].

In the early 1990s, search tools emerged to provide catalogues of web pages [60] and after nearly 20 years of evolution, the modern web search engine consists of three major components: the web crawler, the indexer and the query processor.

The web crawler downloads web pages from the internet and stores copies on some large scale repositories. It normally starts with a manually selected set of uniform resource locators (URLs), fetching the pages that correspond to these URLs,

extracting the URLs that appear on these web pages which are then added to the existing URL set. In such an iterative way, web crawlers are expected to capture a good fraction (up to 70% in 2005 [22]) of the web. For example, starting with 18,000 URLs from over 60 countries, the Heritrix web crawler completed a unique global snapshot of the web with two billion pages [29]. However, a 100% coverage of the entire web is impractical since some websites are not publicly available and some web pages are fairly isolated. In May 2012, Google captured about 50 billion web pages in its index and Bing crawled 16 billion ones in its search engine [75].

The naïve way to process a query is to scan all the crawled web pages to determine the matching ones, but such an approach is clearly impractical, especially when considering the billions of documents and Petabytes of data involved for today's major web search engines. The inverted files, introduced much earlier for document retrieval, constitute an appropriate data structure to speedup web search. The indexer is responsible for constructing the inverted files from the collections of crawled web data. The documents (web pages in our case) processed by the indexer are in the form of {Document Doc\_ID: a sequence of tokens}, where an integer Doc\_ID is assigned here to identify the document. These documents are then transposed into the form of {Term T: a list of Doc\_IDs that contain T}. Such list is called the postings list of a term T. Web pages also incorporate elements like HTML tags, which are intended to display the pages properly but these elements will not be inserted into the inverted files. In order to filter out such useless elements, extract only textual contents and find word boundaries, a process called parsing is performed just before indexing. The indexers build, in general, a dictionary to quickly locate the

postings list of a term. Incremental updating is a highly desirable feature of indexers [13, 14, 36], especially for the document collections archived from the fast changing web. Creation, modification or deletion of web pages all can happen many times every second on the internet, especially those involving news or social networking. To provide up to date information, newly crawled web pages or their most recent versions are added into the inverted files as soon as possible so that they can become searchable immediately.

In addition to the steps described above, web search engines have to quickly serve queries. A typical query consists of several terms connected by Boolean operations [41, 64, 84]. The inverted files generated from the indexers help search engines find the accurate search results by merging the postings lists corresponding to all the query terms. The average query length in the publicly available AOL query log is 2.34 terms [5] and such a few query terms are expected to appear in millions of web pages. As a result, it is highly desirable that searching results be somewhat ranked in order of importance to the users. This is accomplished through a scoring function that attempts to capture the relevance between a query and a document. One of the popular metrics is the TF-IDF weighing function, where both the term frequency (TF) and inverse document frequency (IDF) are taken into consideration based on the observation that a document is highly relevant for a certain term if this term occurs many times in the document (large TF) but rarely shows up in others (large IDF). Such weights (TF and IDF) are usually accommodated within the postings lists or dictionary and thus can directly be used during query processing.

In this dissertation, we mainly focus on the inverted files construction problem

from a given collection of documents and develop extremely high-throughput algorithms that are tested and evaluated on well-known benchmarks.

### 1.3 Indexing Strategies and MapReduce

The indexing process can also be viewed as a sparse matrix transposition problem if the document collection is represented by a document-term matrix as it is commonly done in the field of natural language processing. In this scenario, the postings list of each term is easily obtained by traversing each column of the matrix. However, at the internet scale the numbers of web pages and terms easily reach tens of billions and hence such a matrix is extremely hard to create or update efficiently. Sort based indexing is an approach to get around this problem. Such a strategy consists of generating  $\langle \text{term}, \text{Doc\_ID} \rangle$  tuples during the parsing stage followed by sorting all these tuples by the term field, after which all tuples corresponding to the same term are grouped together to represent the postings list of this term. Normally in the postings lists, Doc\_IDs are also in sorted order to achieve quick seeking by binary search and good compression ratio. A dictionary is also constructed in such a way that, given a term, we can quickly find a pointer to its postings list as well as collection statistics such as the inverse document frequency of a term. Since the size of  $\langle \text{term}, \text{Doc\_ID} \rangle$  tuples is very close to the size of the original document collection, it is impossible to fit these tuples into memory and hence an external sorting algorithm is required. As a consequence, the complexity of this problem is mainly disk I/O bound.

A significant number of parallel and distributing computing strategies have been

used to generate the inverted files. The current prevalent programming model is MapReduce introduced by Google in 2004 [17]. MapReduce is designed for large scale data processing and is scalable on large clusters of commodity computers. More specifically, the *Mappers* generate <key, value> pairs from the input, then the runtime system automatically sort these pairs to aggregate all values for the same key and the corresponding partitioned results are then handled by different *Reducers* for further processing. Many large scale problems like sorting, machine learning and graph computations can be expressed in this model and hence solved by MapReduce on large clusters. To apply MapReduce on the construction of inverted files, the sorting tasks are automatically handled by the underlying MapReduce framework. Therefore in a simple case Mappers will carry out the document parsing and yield <term, Doc\_ID> pairs and Reducers are guaranteed to directly receive the postings lists of terms, sort them by Doc\_ID if necessary and write them onto disks.

The parallel and shared-nothing architecture of MapReduce hides the complexities of low-level parallel programming details and hence reduces software development time. However, this comes at some performance cost. Also, the applications of MapReduce are somewhat limited by the lack of global synchronization or communication within and between different Mappers and Reducers [37]. What's more, modern heterogeneous platforms combining both computing powers from CPUs and GPUs are not fully utilized under the current MapReduce framework.

## 1.4 Temporally Anchored Web Contents and Inverted Files

The web is gradually capturing almost every facet of human activities. From a certain perspective, the web captures a detailed picture of today's world, including what people are thinking (blog), chatting (social network) and what's happening in various countries (news sites). However, the web has been evolving rapidly since its advent and many old web pages are replaced by new versions. Such digital contents over time constitute an important part of our cultural heritage and hence a critical part of this content need to be preserved [2, 7]. There exist many efforts to archive different types of web contents. The most well known related project is the Wayback machine [72], which was created and maintained by the Internet Archive. Since 1996, the Wayback machine has archived over 150 billion web pages and, given one URL, it is able to retrieve all the versions of this URL which have been captured and preserved in its storage. From this type of records, answers can be provided for questions such as what statements showed up on a particular web page at a specific time in the past and how this web page has changed over time. Such temporal archives not only provide evidence for legal issues [31] but also trigger new research in fields such as sociology and linguistics.

However, full text search is currently missing on web archives including the Wayback machine. As in the case of patent examination, locating the prior art with its time stamp is only possible if the URL is already given. However, in most cases these URLs are unknown in advance and are in fact what the users are trying to find in the first place. On the other hand, leading commercial search engines display highly relevant results based on a few keywords, but the postings lists are built based on the

most up to date (last time crawled) versions of the web pages.

The temporal inverted files which insert the time dimension into the indexing structure can handle temporal queries with both query terms and time constraints. A number of web pages crawled from the same URL at different times will all be included in the temporal inverted files. For example, matching results for the query “Iraq War in 2002” are the web pages that contain terms “Iraq” and “war” which were alive during year 2002, but not other pages that still contain Iraq and war. Adding the temporal dimension to web search significantly complicates the problem of building the appropriate inverted files as well as temporally anchored scores to determine an appropriate ranking of the matched pages.

## 1.5 Major Contributions of this Thesis

In this dissertation, we focus on deriving parallel and high performance algorithms for large scale irregular computations such as the construction of inverted files. More specifically, we start by tackling the general inverted files construction problem on a single node with multicore processors, then discuss a fast solution on a cluster of multicore processors. A new approach to handle the temporally archived document collections is developed. We also explore the use of GPUs to accelerate the indexing process as well as to solve the list ranking problem.

Our inverted files construction algorithm for a single node is based on the shared memory model. A number of novel techniques are introduced to implement a high-throughput pipelined strategy; including a hybrid trie and B-tree dictionary data structure, and dynamic work allocation to parallel CPU threads. Given a collection of

documents residing on a disk, the new algorithm is developed for processing these documents and building the inverted files extremely fast. The high throughput pipeline strategy produces parallel parsed streams, which are immediately consumed at the same rate by parallel indexers. We have performed several tests of our algorithm on a single node (two Intel Xeon X5560 quad-core CPUs), and were able to achieve a throughput of more than 280MB/s on the ClueWeb09 dataset. Similar results were obtained for widely different datasets. The throughput of our algorithm is comparable to the best known algorithms reported in the literature running on large clusters.

Based on the algorithm on a single node, the high-throughput pipelined strategy is modified and optimized for a cluster of multicore 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. Besides, a number of new techniques are introduced to partition the indexing workload while minimizing the communication overhead and ensuring load balancing. We have generated extensive experimental results to illustrate the scalability of our strategy relative to the optimized single node algorithm. In particular, even with the MPI communication overhead, each node still achieves a throughput of 272MB/s, leading to over 6GB/s for the 32-node cluster when the 10Gb/s InfiniBand interconnect is used and over 5GB/s when the 1Gb/s Ethernet is used. The performance results seem to be substantially better than the best previous published results that adopt the MapReduce framework.

We develop a new approach to deal with the problem of temporally-anchored search of web archives. Our approach guarantees that the dictionary and postings lists

remain exactly the same as before. Additional temporal indexes are built to capture the lifespan of each document version with little overhead. The lifespan of the entire collection is partitioned into multiple time segments in such a way that each query is localized into one time segment so that the query response time is fast. Such partitioning is conducted by separate threads in a new pipelined strategy and is well hidden by the existing parsing and indexing stages. Several experiments have been performed with real world data at significant scale, showing that the new temporal algorithm still maintains the high and scalable throughput on the cluster which is over 3.7GB/s. What's more, the test results indicate that with our collection partitioning strategy the query performance is not traded in for high indexing throughput and is quite stable independent of the query time constraint.

Finally, we explore the possibility of solving the indexing problem and the list ranking problem fast on GPUs. The parallel indexing approach is tailored for a heterogeneous platform consisting of both multicore CPUs and high multithreaded GPUs. Particular features of CPUs and GPUs are taken into consideration to split workloads between these two types of processing units. By doing so, they are both assigned computations that better fit into their architecture and the combined throughput is even better than the simple sum of their individual ones. The list ranking primitive can be used to decompress the inverted files and we introduce an optimized multithreaded GPU algorithm through a randomization process that reduces this problem to a large number of fine-grain tasks. These small tasks are then mapped onto the multithreaded GPU in such a way that the processing cost per element is shown to be close to the best possible. The experimental results show

scalability for a wide range of list sizes and significant improvement on the previously published parallel implementations. The performance also compares favorably to the one of the best known CUDA algorithm for the scan operation on the same GPUs.

## Chapter 2 Fast Construction of Inverted Files on a Single Node

In this chapter, we address the traditional inverted files construction problem on a single multicore node. Our approach involves a hybrid dictionary data structure consisting of a trie and B-trees, and a pipelined strategy that includes parsing and indexing stages, both of which are implemented using parallel threads.

### 2.1 Overview

A critical component of all information retrieval systems including web search engines is the set of inverted files generated typically from a very large collection of documents. A considerable amount of research has been conducted to deal with various aspects related to inverted files. In this chapter, we are primarily concerned with methods to generate the inverted files as quickly as possible. 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 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 comes at a significant price in terms of performance, especially when using the emerging multicore processors. In this chapter, we take the different approach that exploits common features present on current multicore processors to obtain a very fast algorithm for generating the inverted files.

Our testbed consists of two Intel Xeon X5560 quad-core CPUs with 24GB of

main memory and each quad-core shares an 8MB L3 cache. The multicore processor offers a multithreaded environment with a shared memory programming model. In this model, communication is carried out through the shared memory, and hence a careful management of the shared memory and the workloads among the cores is critical to achieve good performance.

The rest of the chapter 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 work that is most related to ours. Section 2.3 provides a detailed description of our algorithm, including our new dictionary data structure and the organization of the parallel parsers and parallel indexers. Section 2.4 provides a summary of our test results on three very different benchmarks, and we conclude in Section 2.5.

## 2.2 Related Work

We start by giving a brief overview of the process of building inverted files given a collection of documents residing on a disk. The overview will be followed by a summary of previous work on parallel and distributed implementations of this strategy.

The overall process essentially converts a collection of documents into inverted files consisting of a postings list for each of the terms appearing in the collection as follows. The strategy starts by parsing each document into a “bag of words” of the form  $\langle \text{term}, \text{document ID} \rangle$  tuples, followed by constructing a postings list for each term such that each postings 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 usually 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.

In [23], postings lists are written as singly linked lists to disk and the dictionary containing the locations of the linked lists remains in main memory; however, another run is required as post-processing to traverse all these linked lists to get the final contiguous postings lists for all terms. Moffat and Bell proposed sort-based indexing in [47] 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 [28] 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 [45], 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 [58] 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 [17], 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 the 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  $\langle \text{term, partial postings list} \rangle$  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 [42, 43]. Their algorithm seems to achieve the best known throughput rate for full text indexing. Around the same time, Lin et.al [19, 39] developed a scalable MapReduce Indexing algorithm by switching  $\langle \text{term, posting}\{\text{document ID, term frequency}\} \rangle$  to  $\langle \text{tuple}\{\text{term, document ID}\}, \text{term frequency} \rangle$ . By doing so, there is at most one value for each unique key, and moreover it is guaranteed 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.

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,  $\gamma$  encoding and Golomb compression.

## 2.3 Description of Our Algorithm

Our main goal in this chapter is to present a very fast indexing algorithm on multicore processors. To evaluate the actual performance of our algorithm, we use the Intel Processor Xeon X5560 consisting of two quad-core CPUs. These eight cores are

divided into two groups dedicated for parsing threads and indexing threads respectively. These two types of threads are running concurrently. In our implementations, we use Pthreads and their programming environment.

### 2.3.1 Overall Approach

Briefly, a number of parsers run in parallel on the multicore CPU, where each parser reads a fixed size (typically 1GB WARC files [71]) 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 [17, 19, 39, 42, 43].

There are many details that need to be carefully worked out for this approach to achieve optimal throughput. Before providing details about the parsing and indexing tasks and how they are allocated to the available cores, we describe the dictionary data structure used since it plays a central role through which the various tasks coordinate their work. This will be followed by a description of the parsing and indexing tasks allocated to the various cores available on our parallel platform.

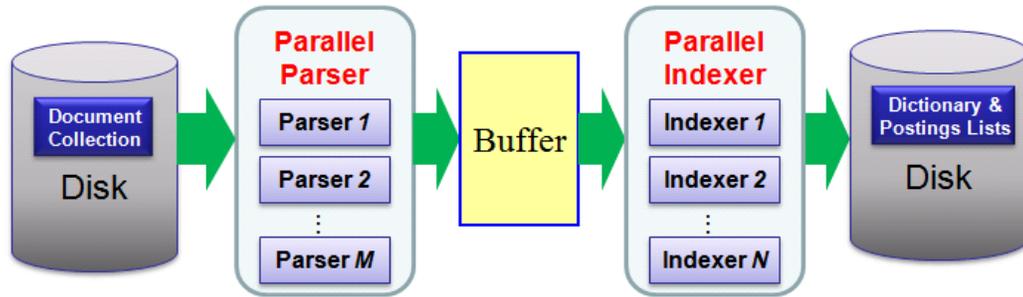


Figure 2.1: Dataflow of Pipelined and Parallel Indexing

### 2.3.2 Dictionary Data Structure

The structure of the dictionary plays a critical role in the performance of our indexing algorithm since multiple concurrent threads have to access the dictionary and hence conflicts among the corresponding parallel threads must be properly resolved in such a way to ensure correctness and achieve high performance. The B-tree is the typical data structure used in many information retrieval systems due to its balanced structure and small height. In particular, the height of any  $n$ -key B-Tree is at most  $\log_t \frac{n+1}{2}$  where  $t$  is the degree of the tree. Such a structure is not in general suited for operations such as multiple threads attempting to insert a new term into the same node or any other operations with similar conflicts. Locks can be used to prevent such hazards but the overhead is extremely high since many threads may have to wait until a thread completes its modification of the B-Tree. In our implementation, we introduce a combination of a trie at the top level and a B-Tree attached to each of the leaves of the trie. A similar data structure was used in [27] to achieve compact size and fast search; however in our case we will exploit this hybrid data structure to achieve a high degree of parallelism and load balancing among the multiple cores.

### 2.3.2.1 A Hybrid Data Structure

Our hybrid data structure for the dictionary is shown in Figure 2.2. Essentially, terms are mapped into different groups, called trie collections, followed by building a B-tree for each trie collection.

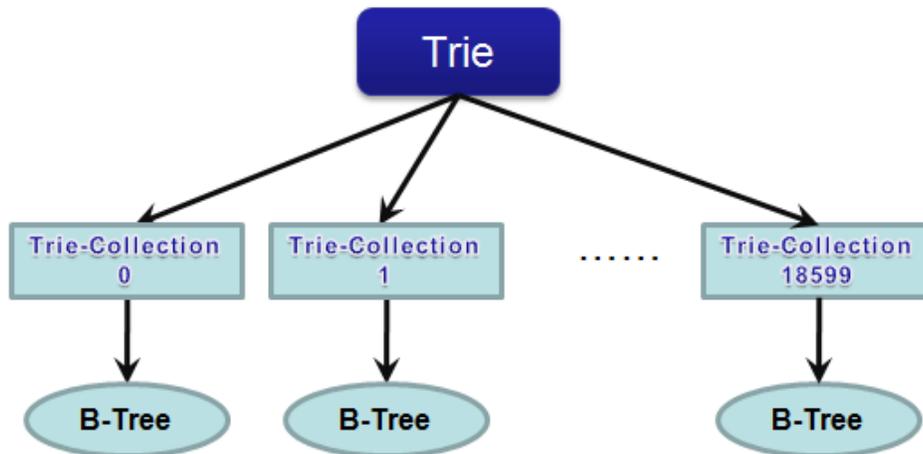


Figure 2.2: Hybrid of Trie and B-Tree Structure of Dictionary

The main reason we use a trie at the top level is to generate many independent B-trees instead of a single B-tree. Each B-tree is then handled by a single thread, independently of the other B-trees. In our case, we fix the height of the trie to three, which means 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 fact, the index category of each leaf of the trie is specified in Table 2.1. Clearly, the number of terms belonging to different trie collections varies significant; for example, there are many words with prefix “the” and hardly any terms

with prefix “zzz”. The height of three for the trie seems to work best since a smaller height will lead to a wide variety of trie collections, some very large and some very small, which will be hard to allocate to the different core processors in such a way as to achieve a good load balance. A larger value for the trie height will generate many small trie collections, which will be again hard to manage. Since the trie height is constant here, we don’t need to actually build the trie structure but we use a table to map a trie index directly into the root location of the corresponding B-tree.

Table 2.1: Trie-Collection Index Definition

Index	Term Category	Example
<b>Terms not Falling in to the Next Categories (1024 entries)</b>	0	Terms with less than four letters or contain one or more symbol outside [a-z] in the first three letters
	1	
	...	
	1023	
<b>Terms with &gt;3 letters and no special letter in the first 3 letters (26*26*26=17576 entries)</b>	1024	Terms with >3 letters and starting with ‘aaa’
	1025	Terms with >3 letters and starting with ‘aab’
	...	...
	18599	Terms with >3 letters and starting with ‘zzz’

In addition to allowing a high degree of parallelism through the independent B-trees, our hybrid data structure achieves two more 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, save memory space for term strings and reduce string comparison time in B-tree operations. The average length of a stemmed token is between 6.6 and 6.7 in all the

datasets used in this dissertation and hence removing the first three letters results in almost doubling the string comparison speed. An alternative option to the trie is to use a hash function, but a hash function will still require comparisons and searches on full strings and hence won't be as effective as the trie.

### 2.3.2.2 Special Node Structure in B-tree

The structure of a B-tree node is illustrated in Table 2.2. The degree of B-tree is 16, that is, each node can hold up to 31 terms, and this number is selected from experiments as well as to match the CUDA warp size which will be later described in Chapter 5. 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. Since the average length of a stemmed token is between 6.6 and 6.7 and the trie at the top level takes care of the first three bytes, four bytes will be sufficient to hold the remaining bytes in most cases. These caches are used to store the first four bytes of the corresponding term strings. Consider for example the term “application”—the first 3-byte “app” is not needed since it is already captured by the trie, so we only have to store the term string “lication” into the B-Tree, and hence “lica” is stored in the cache, and the remaining string is stored in another memory location indicated by the term string pointer.

Table 2.2: Data Structure of One B-Tree Node

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

Occasionally some memory space will be wasted when caches are not fully occupied. However the advantages of our scheme are substantial because:

- *Short strings can be fully stored within the B-tree node;*
- *For long strings, even though only the first four bytes are stored, it is highly likely that the required comparison between two term strings can be done with only these four bytes since it is a rare case that two arbitrary terms share the same long prefix.*

A padding space of 4-byte is also added such that the total size of a B-tree node is exactly 512 bytes. In addition, the order of the digits is organized in such a way that a B-tree node is partitioned into four 128-byte chunks so as to match the CPU cache line or the GPU shared memory width.

### 2.3.3 Structure of Parallel Parsers

As shown in Figure 2.1, 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 later in Figure 2.8. 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 Figure 2.3. Each such sequence will be executed by a single CPU thread. The corresponding steps are briefly described next.

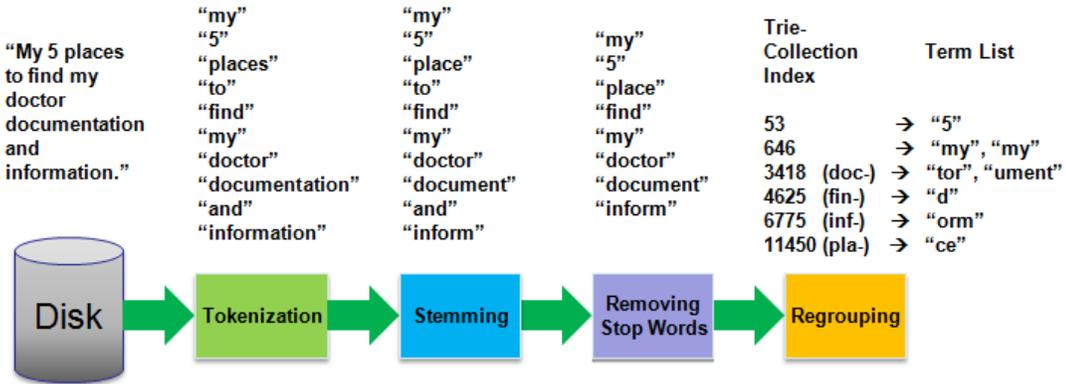


Figure 2.3: Data Flow of One Parser Thread

- **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 systems. 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 corresponding to index i: (Doc\_ID1, term1, term2, ...), (Doc\_ID2, term1, term2, ...), .....*

*Doc\_IDs* in the lists are local ones within this parser. A global document ID offset will be calculated by the indexer; thus the global document ID can be obtained by adding *Doc\_ID* and the global offset.

#### 2.3.4 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 Figure 2.4. To ensure load balancing, a CPU thread will take care of the B-trees of several trie collections as we explain later.

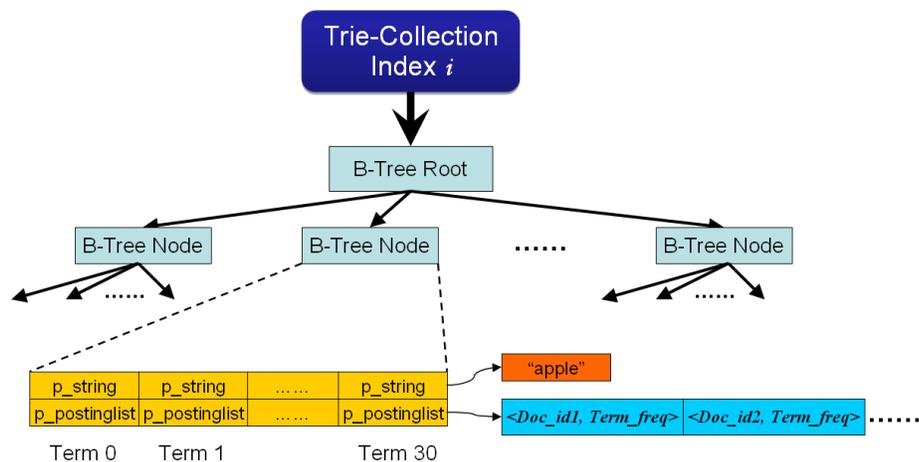


Figure 2.4: A B-tree Corresponding to a Single Trie Collection Index

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 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.

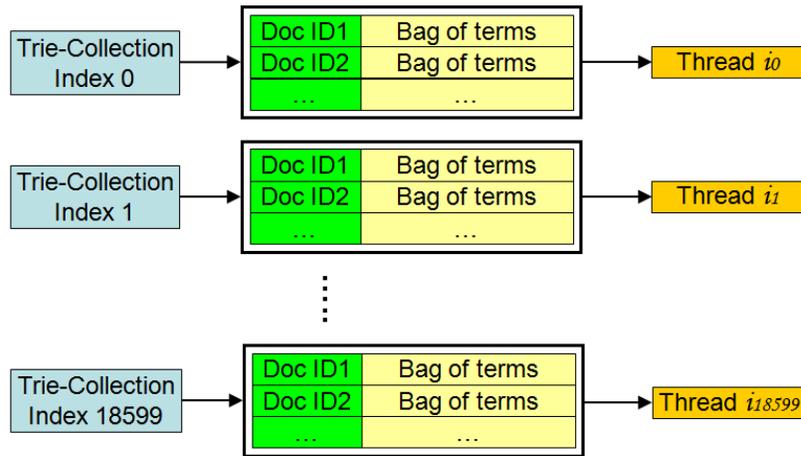


Figure 2.5: Work Assignments among Multiple Threads

A sampling strategy is used 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 thread 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 Figure 2.6.

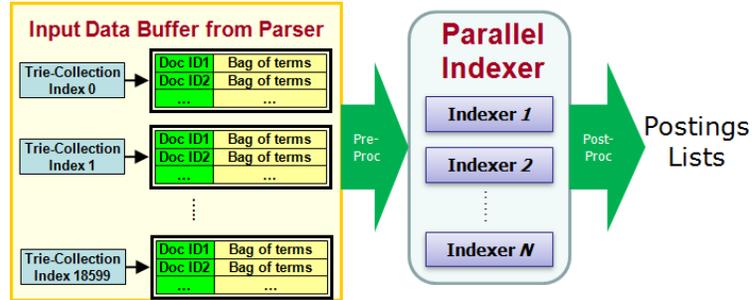


Figure 2.6: Data Flow of One Single Run on Parallel Indexers

### 2.3.5 Overall Pipelined Data Flow

In our setting, the input document data collection is stored on a centralized storage connect to our machine via a 1Gb/s Ethernet and is processed through our multicore CPU platform to generate the postings lists and store them on either a local disk or the centralized storage. 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 2.4.1, we determine the best values of these parameters for our platform.

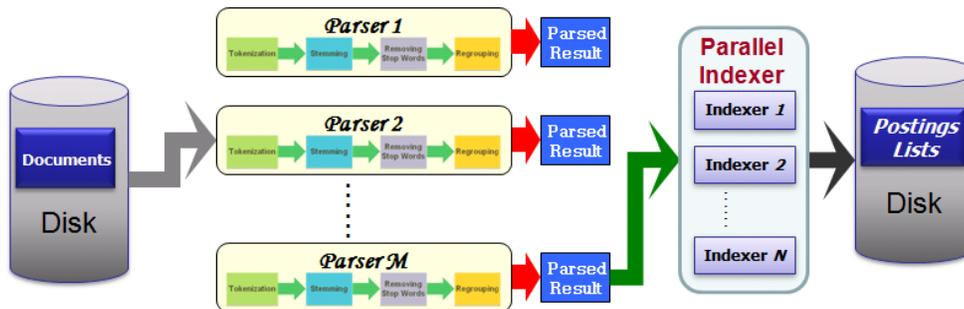


Figure 2.7: Pipelined Data Flow of Overall Indexing System on a Single Node

To avoid several parsers from trying to read from the same storage 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 applied, the timing sequence of parallel parsers looks like the example shown in Figure 2.8.

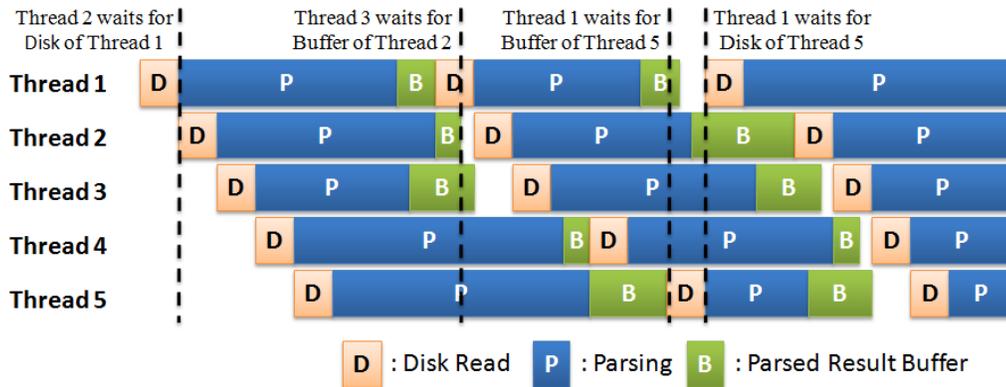


Figure 2.8: Timing Sequence of Parallel Parsers

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 2.1. 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.

This output format has some additional benefits including:

- *faster search when narrowed down to a range of document IDs since we can fetch only those partial postings lists that overlap with this range. This is possible since we include an auxiliary file containing the mapping of document IDs to output file names; and*
- *the possibility of parallel reading of the postings lists because the output files can be written onto multiple disks.*

Although a postings list is divided into partial lists and stored in separate files, the index is still monolithic for the entire document collection. We can also see later that such output format is critical in the temporal case in Chapter 4. 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.

## 2.4 Experimental Results

Our parallel and pipelined indexing system is tested on a single machine that holds two Intel Xeon X5560 quad-core CPUs. We use three document collections to test the performance of our algorithm. We start with the first English segment of the **ClueWeb09** collection, which has been heavily utilized by the information retrieval community. Crawled between January and February 2009 by Language Technologies Institute at Carnegie Mellon University, this dataset includes 50,220,423 web pages packed into 1,492 files with a total size of 230GB compressed and 1.389 TB uncompressed. The second dataset is the **Wikipedia01-07** data, which is derived from

a publicly available XML dump of Wikipedia articles [74] created on January 3th 2008 with 83 monthly snapshots between February 2001 and December 2007. The third is the dataset from the **Library of Congress**, which includes weekly snapshots of selected news and government websites crawled between May 2004 and September 2005 by the Internet Archive. Overall statistics about the three are given in Table 2.3. The number of terms and tokens may vary with different implementations due to the choice of tokenization and stemming procedures. These document collections are stored on a disk connected to our platform via a 1Gb/s Ethernet. The generated output, postings lists and dictionary, are written to either a remote disk (with 1Gb/s connection) or to a local disk (resulting performance differences are very small and insignificant). We report results averaged over three trials and in all our tests the differences between the fastest and slowest execution times have been less than 5%.

Table 2.3: Statistics of Document Collections

	<b>ClueWeb09 1<sup>st</sup> Eng Seg</b>	<b>Wikipedia 01- 07</b>	<b>Library of Congress</b>
Compressed Size	230GB	29GB	96GB
Uncompressed Size	1422GB	79GB	507GB
Crawl Time	01/09 to 02/09	02/01 to 12/07	05/04 to 09/05
Document Number	50,220,423	16,618,497	29,177,074
Number of Terms	84,799,475	9,404,723	7,457,742
Number of Tokens	32,644,508,255	9,375,229,726	16,865,180,093

In what follows, we start by determining the best values on our platform for the following parameters: the number of parallel parsers and the number of parallel indexers. Then we look at the scalability of parallel indexers and dictionary growth. This will be followed by summarizing the overall performance of our algorithm on

the three document collections. We end by comparing our performance to the best reported results in the literature.

#### 2.4.1 Optimal Number of Parallel Parsers and Indexers

In this section we focus on determining the best number of parallel parsers and indexers. Note that our goal is not only to speed up the parsing of the documents but also to match it with the speed at which indexers are able to consume the parsed data. 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 Figure 2.9 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 8-core CPUs. When all eight cores are executing the parsing program, the overall throughput is 341MB/s, which indicates the maximum possible throughput of the parsing stage on a single

node. Our algorithm has reached 82% of this maximum throughput with 6 parsers and 2 indexers. On the other hand, over 47% of the disk bandwidth has been consumed during the parsing and indexing process, which means that the throughput of our algorithm is near the peak I/O throughput of the underlying hardware.

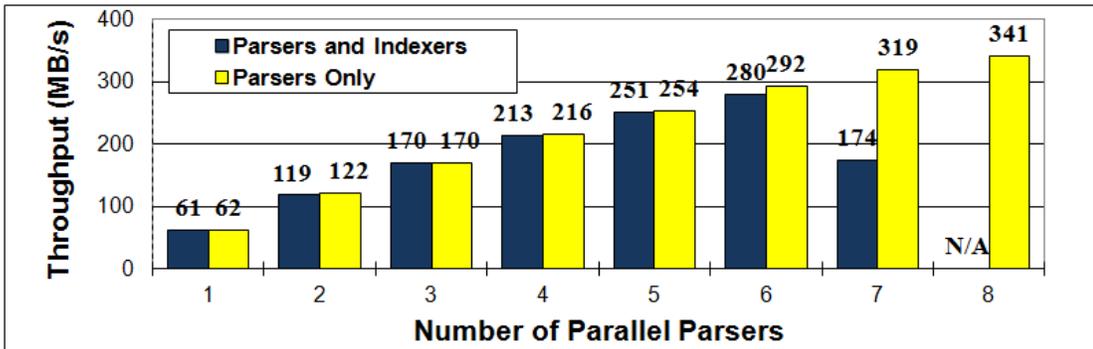


Figure 2.9: Optimal Number of Parallel Parsers and Indexers

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.

#### 2.4.2 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 Figure 2.10. 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.

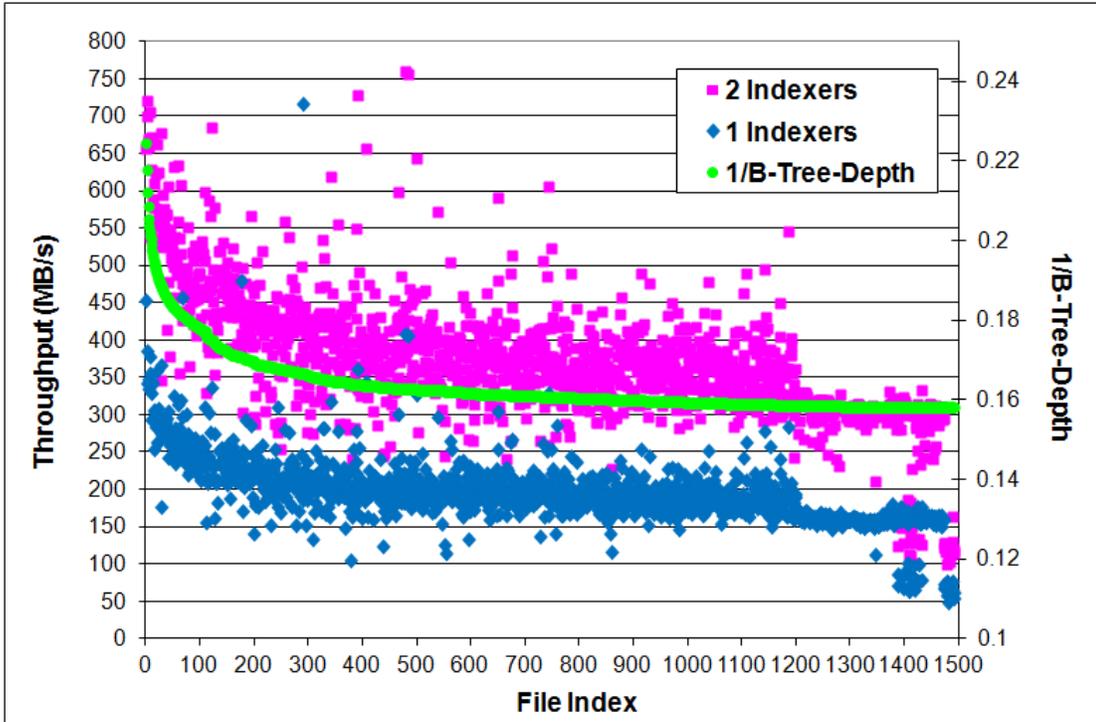


Figure 2.10: Scalability of Parallel Indexers

### 2.4.3 Performance of our Algorithm on Different Document Collections

We show in Table 2.4 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 2.4, 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.

Table 2.4: Performance Comparison on Different Document Collection

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

#### 2.4.4 Comparison with Fastest Known Algorithms

In this section, we compare the performance of our algorithm with the best known MapReduce algorithms that appeared in the literature, namely Ivory MapReduce [19, 39] and Terrier MapReduce [43] on exactly same ClueWeb09 first 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 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 and in this case our algorithm is 7% slower while the resulting postings lists are about 1.6 times larger. According to [38], their Ivory MapReduce implementation with positional indices is about 1.2 times slower compared to non-positional indices, 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 [19],

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 2.5.

Table 2.5: Platform Configuration Comparison

		<b>This Thesis</b>	<b>Ivory MapReduce</b>	<b>Terrier MapReduce</b>
<b>System Details</b>	<b>Processors per Node</b>	Two Intel Xeon 2.8GHz Quad-core CPUs	Two Intel Single-core 2.8GHz CPUs	Two AMD Quad-core Opteron CPUs
	<b>Memory per Node</b>	24GB	4GB	16GB
	<b>File System</b>	File System via 4Gb/s Ethernet or local disks	Hadoop Distributed File System	Hadoop Distributed File System
<b>Throughput (MB/s)</b>	<b>1 Node (8 cores)</b>	280	—	33
	<b>30 nodes (240 cores)</b>	—	—	460
	<b>99 Nodes (198 cores)</b>	—	167	—
	<b>280 Nodes (560 cores)</b>	—	289	—

It is clear that our pipelined and parallel indexing algorithm implemented on a single node achieves the performance comparable to the ones from MapReduce on larger clusters. A number of factors contribute to the superior performance of our algorithm including:

- *The pipelined and parallel strategy that matches maximum possible parsing throughput with parallel indexing on available resources.*
- *The 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;*
- *Parallel parsers that scale well with the number of threads, in addition to the fact that the file reading and decompression are carefully optimized to boost the I/O bandwidth;*
- *The regrouping operation that is integrated into the parsing stage with little overhead but that noticeably increases CPU cache performance;*

## 2.5 Conclusion

In this chapter, we presented a high performance pipelined and parallel indexing system based on a single node of multicore CPUs. We have shown how to optimize the performance of the pipeline by using parallel parsers and indexers in such a way that the streams produced by the parsers are consumed by the indexers at the same rate, and that rate is optimized. Several new techniques were introduced including a hybrid trie and B-trees data structure. The experimental tests reveal that our implementation on a single multicore processor shows scalable parallel performance in terms of the number of cores with a resulting throughput comparable to the most recent published algorithms on large clusters.

## Chapter 3 Fast Construction of Inverted Files on a Cluster

The starting point of our cluster algorithm is the pipelined strategy on a single node with multicore processors presented in the last chapter. We generalize this strategy to a cluster environment, under either the distributed storage model or the centralized storage pool model, to develop a high-throughput, scalable algorithm that ensures that all the parallel parsed streams on all the nodes are immediately consumed at the same fast rate by the distributed, parallel indexers. As for our communication model, we use the standard MPI primitives to manage the communications between the different nodes.

### 3.1 Overall Approach on a Cluster

We start by exploring different ways to map our pipelined algorithm onto a cluster of multicore processors. Our main goal is to be able to build a global dictionary and generate the postings lists stored on external storage with the maximum possible throughput. There are two reasonable approaches 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 indices and the IDs of the indexers, which is used to distribute the parsed streams to the nodes.*

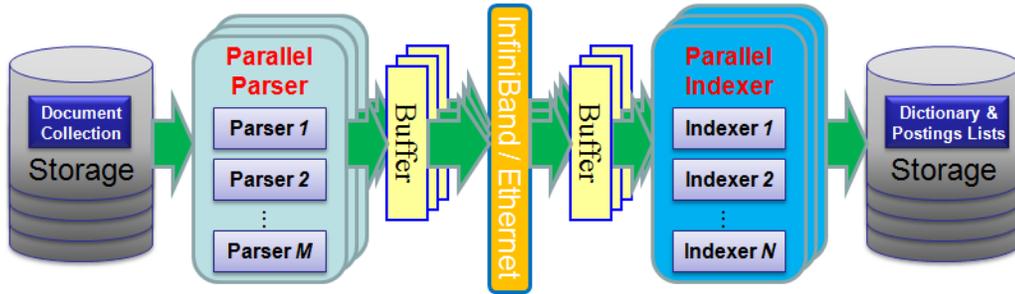


Figure 3.1: Data Flow of Partition-and-Index Strategy

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 amount of 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 the inter processor 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 Figure 3.1. 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 (in our case, either 10Gb/s InfiniBand or 1Gb/s Ethernet). This will be described in more details shortly.

## 3.2 Our Extended Approach

### 3.2.1 Storage Model

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.

### 3.2.2 Load Balancing 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 [71]) 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 throughout 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.

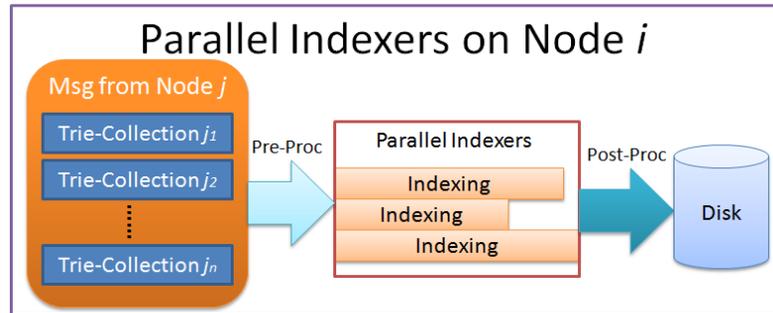


Figure 3.2: Data Flow of Parallel Indexers on One Node in the Cluster

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.

### 3.2.3 Communication 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  $P=32$ , which only takes about 8 ms time to send it from one node to another using the 10Gb/s InfiniBand and about 60 ms 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.

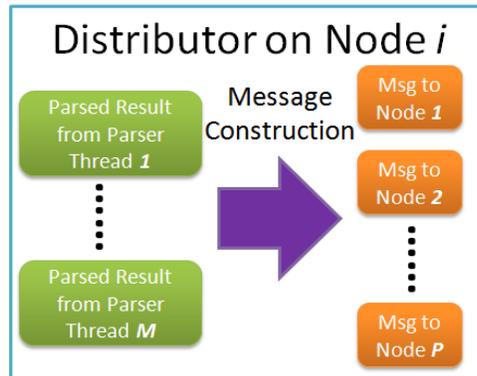


Figure 3.3: Message Construction by Distributor

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 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.

### 3.2.4 Overall Data Flow

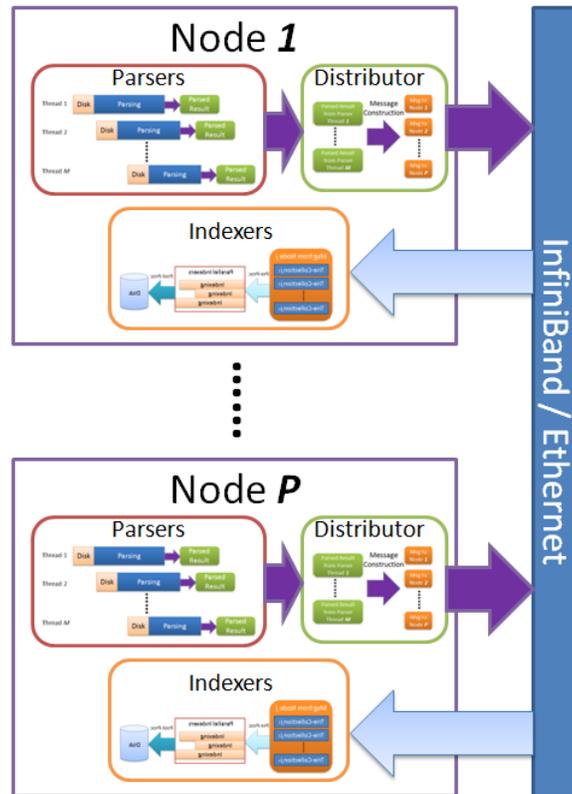


Figure 3.4: Overall Data Flow in the Cluster

Putting all the pieces together, we get the overall data flow shown in Figure 3.4 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.

### 3.3 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 210GB disk. We use the same three collections in Chapter 2 plus the entire **ClueWeb09 English** collection.

Table 3.1: Statistics of the ClueWeb09 English Document Collections

	<b>ClueWeb09 English</b>
<b>Compressed Size</b>	1,936GB
<b>Uncompressed Size</b>	12,453GB
<b>Crawl Time</b>	01/09 to 02/09
<b>Document Number</b>	503,903,810
<b>Number of Terms</b>	447,373,242
<b>Number of Tokens</b>	281,794,398,151

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.

In what follows, we start by showing 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. This will be followed by examining performance with two types of interconnect. We end by comparing the performance of our algorithm to the best known results in the literature.

### 3.3.1 Scalability of the Cluster Algorithm

We use three metrics to evaluate the scalability of our cluster algorithm on a cluster with 32 nodes with multicore 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.

#### 3.3.1.1 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 3.2 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 in Chapter 2. 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.

Table 3.2: 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

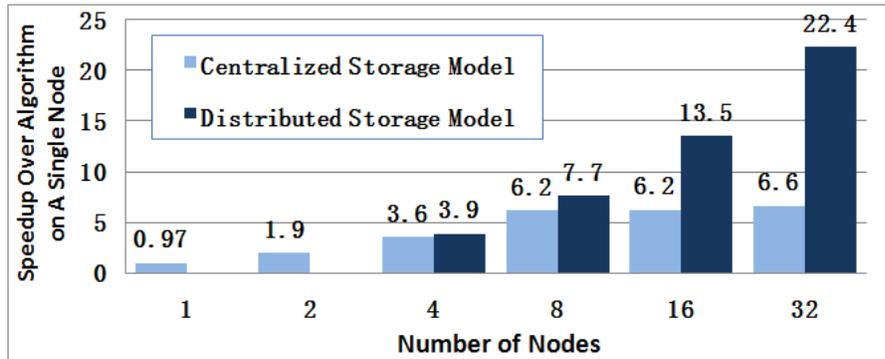


Figure 3.5: Scalability over the Number of Nodes with Same Input Data

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.

Table 3.3: Ratio of the Throughput of Our Algorithm and Peak I/O Throughput

<b>Number of Nodes</b>	<b>Distributed Storage Model</b>	<b>Centralized Storage Model</b>
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

The numbers in Table 3.3 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 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 2.8GB/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 Figure 2.8, 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 or incremental updating 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 found that 6 parsers and 2 indexers achieve the best performance on a single multi-core node in Section 2.4.1. Figure 3.6 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.

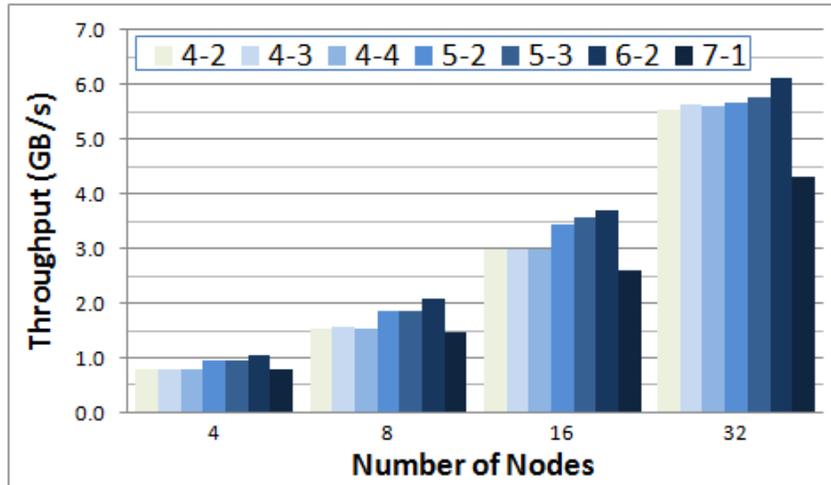


Figure 3.6: Optimal Parameters on the Cluster in Distributed Model

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.

### 3.3.1.2 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 3.4. 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  $P=1$ .

Table 3.4: 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

### 3.3.1.3 Scalability over Data Size

Figure 3.7 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.

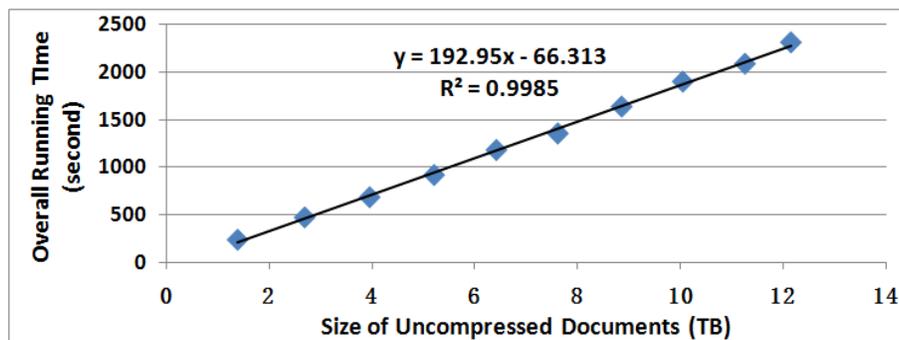


Figure 3.7: Scalability over the Size of Input Documents

### 3.3.2 Performance with Different Types of 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.

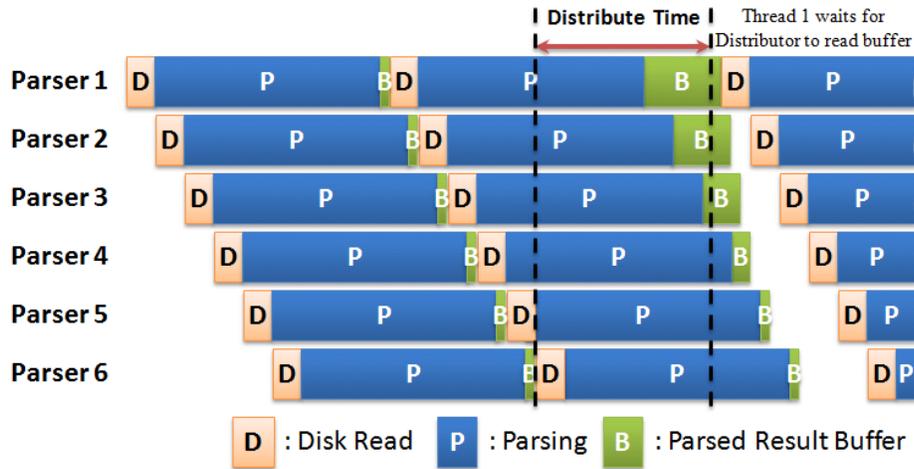


Figure 3.8: Impact of Distribute Time in the Ideal Pipeline

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 Figure

3.8, 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:

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

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  $T_N = 0.26$  second 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.

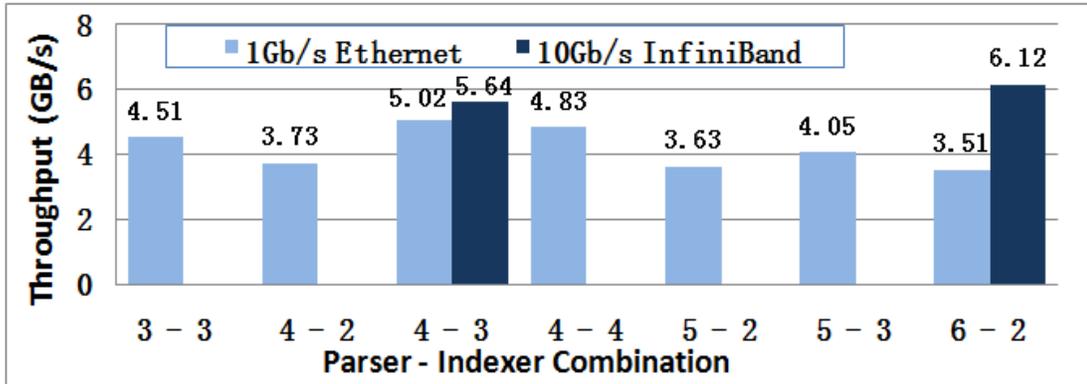


Figure 3.9: Performance on 32 nodes using the 1Gb/s Ethernet

From the experimental results with 32 nodes shown in Figure 3.9, 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.*

### 3.3.3 Comparison with Fastest Known Algorithms

In this section, we compare the performance of our algorithm again with the Ivory MapReduce [19, 39] and Terrier MapReduce [43] on the same ClueWeb09 first English segment dataset. As mentioned earlier, the Ivory MapReduce algorithm employs positional postings lists and our algorithm about 10% slower on the cluster with the same output format. The slowdown here is slightly higher on the cluster than the 7% on a single node in Section 2.4.4 because we have to transfer more intermediate results over the cluster interconnect.

Table 3.5 Platform Configuration and Performance Comparison

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

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. 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.

We note that this comparison has its shortcomings. First, the overall performance may be affected by details such as different compression and stemming schemes. Second, the goal of MapReduce is for simplified processing on large data stored on the disks of the nodes of a cluster with the programming efforts focused on determining the operations of the Map and Reduce workers and their numbers

without taking an in-depth look at the underlying hardware architecture. The same MapReduce program is also portable to other parallel platforms with little effort. On the other hand, our goal in this dissertation is to exploit the potential powers from the emerging multicore processors with reasonable memory size and interconnect network. We optimize our algorithms by carefully exploiting the hardware details of our current cluster; however, such algorithm can also be adapted to different hardware configurations. For example, we can change the combination of parsers and indexers to 12-4 or 13-3 with sixteen CPU cores per node, and re-define the size of the document segment in each single run if the size of the main memory changes. As a conclusion here, such comparison still provides a clear indication of the effectiveness of the approach described in this thesis.

### 3.4 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:

- *Assignment of parsed sub-streams to distributed indexers using a random sampling preprocessing step;*
- *Development of a fully parallelized scheme that makes efficient use of available cores 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.

## Chapter 4 Construction of Temporal Inverted Files

We consider in this chapter the information retrieval problem over a collection of time-evolving documents such that the search has to be carried out within a temporal context. A solution to this problem is critical for a number of emerging large scale applications such as web archiving and the exploration of time-stamped web objects generated through information feeds.

### 4.1 Overview of Temporal Inverted Files

A number of critical emerging applications require the organization and management of an extremely large number of time-evolving objects, such that these objects can be interactively explored through content based search within a temporal context. Such applications include web archiving and time-stamped web objects generated through information feeds. Our main goal in this chapter is to develop an effective approach for tackling such applications using and extending information retrieval technologies. More specifically, we develop an efficient strategy for building the inverted files, supplemented by new indexing structures, such that a temporal-based search can be carried out as efficiently as in the traditional non-temporal case. The overhead required for building the extra indexing structures is shown to be minimal relative to the traditional approach of processing the documents without taking the time stamps into consideration.

Information retrieval technologies do not typically include the temporal dimension. For example, major web search engines such as Google and Bing support queries on the web pages that are currently available on the web. Their basic

approach is not tailored to handle temporally-anchored searches such as locating the most important news about Barack Obama before year 2005. While the internet is by far the largest library that ever existed, its contents are always changing, and hence old valuable contents can be lost forever if not archived and preserved properly. Major library and archiving organizations are constantly crawling the web to capture and archive particular contents of interest. Such contents constitute a collection of time-evolving documents; each document corresponds to a specific uniform resource identifier (URI) with its corresponding document versions time stamped by the crawl times. Such organizations include the Internet Archive [32], the Library of Congress [46], UK Web Archiving Consortium [68], the National Library of Australia [55] and the California Digital Library [73]. In particular, over 5.8 petabytes of web data have been captured by the Internet Archive Petabox project as of December 2010 [56], which continues to grow at a rate of more than 100 terabytes per month since 2009 [44]. The Internet Archive uses its WayBack [72] machine to search such contents where the user supplies a URL and the engine responds with a chronological list of the dates when the corresponding page was crawled. Our work includes the temporal dimension as an integral component of the strategy to build the inverted files so that queries containing text and a time specification can be handled efficiently.

The rest of the chapter is organized as follows. In the next section, we provide a brief background about a summary of the previous work on temporally anchored information retrieval that is most related to this thesis. In section 4.3 we formally define the problem and introduce the new indexing structures needed for managing the temporal dimension. In Section 4.4, the query performance is examined with our

new indexing structures. The parallel algorithms for constructing the temporal inverted files are developed in Section 4.5 and the experimental results are presenting in Section 4.6. We conclude in Section 4.7.

## 4.2 Related Work

We assume that each document, which refers to a time-evolving object in this chapter, is uniquely identified by a global identifier such as a URI. In the temporal case, a document may have many versions over time, each of which includes a time stamp that indicates the start time of this document version. Only one version of a document can exist at any time. The start time could be the time when the document version was created or the time when the document was first seen. For example, in web archiving, a time stamp is assigned whenever the corresponding web page is crawled, and the versions correspond to the crawled distinct pages all of which belong to the same URL, which is the URI in the case of web content. Our problem is more general than the standard web search problem discussed in Chapters 2 and 3. In the following we summarize prior work on construction of inverted files for time-evolving documents.

In the temporal case, previous work has been limited. A temporal document database first developed in [51] introduced a solution with a hybrid of document name index, version database and text index. The document name index maps a URI to its list of global Version IDs since one document may have several versions. The version database assigns global Version ID to versions of documents according to the start time, but two contiguous global Version IDs may refer to two versions of

different documents. The list of document versions that a term appears in can be found in the text index (or the inverted files in our definition). In [50], the global Version ID is replaced by <Document ID, Document Version ID> tuple, thereby if a term appears in several consecutive versions of a document its postings could be shrunk into <Document ID, Document Version ID range> tuples for space efficiency. Time Index+ is then incorporated for frequent terms for large document databases [53]. However, similar to [26, 70] their main target is a database for versioned documents instead of an information retrieval system. As a result, temporal query has very high cost with the text index since the whole postings list has to be scanned and critical elements such as term frequency or term positions in a document are not available for scoring [52]. Berberich et. al. [9, 10] extended the content of traditional postings to <Document ID, score, time\_start, time\_end>. Postings with the same Document ID and similar scores are merged to reduce the space requirement as a lossy compression strategy. To reduce search cost, postings lists are split into sublists by valid time intervals so only the sublists valid in the time range in the query will be traversed. Based on this, Anand et. al. [4] further proposed the sharding approach for efficient temporal inverted files in both space and query processing time. In [63] an analytical model is proposed to effectively choose the time span of sublists to optimize the search cost and the model is shown to be very close to the actual experimental results.

## 4.3 Problem Definition and Temporal Strategy

### 4.3.1 Problem Definition

Following standard information retrieval terminology, we refer to our objects (typically, web pages) as documents, each document is uniquely identified by a URI. In our case, each document evolves over time and hence many versions of the document may exist. A document version  $Doc_{i,j}$  is the  $j^{th}$  version of the document identified by  $URI_i$ , which was first created or detected at time  $t_{i,j}$  and hence can be defined by:

$$\{URI_i, t_{i,j}, < T_1^{ij}, T_2^{ij}, \dots, T_{N_{ij}}^{ij} >\}$$

where  $T^{ij}$  is a term that occurs in  $Doc_{i,j}$  and  $N_{ij}$  is the total number of terms of  $Doc_{i,j}$ . By definition, the start time of a document version is equal to  $t_{i,j}$ , but the end time is not defined until a new version  $Doc_{i,j+1}$  occurs in which case it is equal to  $t_{i,j+1}$ . The lifetime of the document version  $Doc_{i,j}$  is then defined as  $[t_{i,j}, t_{i,j+1})$ .

A query in the temporal case contains a set of query terms  $\{Q_k\}$ , possibly connected by Boolean operators, and a query time  $t_Q$ . A document version  $Doc_{i,j}$  matches the query if it contains some or all the terms in  $\{Q_k\}$  (depending on the query Boolean operator) and is alive at  $t_Q$ , that is,  $t_Q \geq t_{i,j}$  and, if  $t_{i,j+1}$  exists,  $t_Q \leq t_{i,j+1}$ . The result of the search is a ranked set of document versions based on a scoring function used to determine similarity scores between the query and the document versions. Note that the computation of the scoring function typically requires some global statistics at  $t_Q$  such as the number and average length of live document versions and the number of live document versions containing  $Q_k$  (i.e. document frequency of  $Q_k$ ).

As in the non-temporal case, we assign a monotonically increasing global Version ID (VID) to each document version and use this VID in the postings lists to identify the appropriate document version in our collection. From now on, we use the unique identifier  $VID_{i,j}$  to represent the document version  $Doc_{i,j}$ . Note that we are implicitly assuming that our collection consists of document versions over discrete time steps called the *elementary time steps*. An elementary time step captures the time granularity of the document collection as well as the granularity at which queries can be answered. For example, the Wikipedia collections used in our tests are monthly snapshots of Wikipedia generated at the end of a month. Hence in this case, an elementary time step is a month which constrains  $t_Q$  to be within a month such as February 2004. Within an elementary time step, there is at most one document version per URI and the global statistics remain the same. All time values defining start and end times correspond to elementary time steps. As a result, if snapshots of document versions are read by the chronological order of increasing elementary time step, it is guaranteed by our VID assignment strategy that the start time of a higher VID is later than or equal to the start time of a lower VID. Given that in our algorithm the postings lists are in sorted order of VIDs as described in Section 2.3.5, the postings are also in sorted order of document version start time in a postings list.

#### 4.3.2 Overall Strategy for Temporal Indexing and Searching

Our overall strategy is similar to the traditional strategy of using postings lists, each posting consists of the VID and the term frequency within that document version, sorted according to the VIDs on the list. However to carry out temporal indexing and searching effectively, we introduce two new indexing structures, in

addition to the usual global dictionary indexing structure. The new data structures are two temporal tables defined as follows:

- **VID table**, which maps a VID to its unique URI, start and end times, document length and possibly other document version information.
- **URI hash table**, which maps each URI in our collection to all the VIDs corresponding to this URI. Note that, in addition to its role in building the VID table, this data structure will enable a user to quickly receive, if desired, all the document versions of a particular document once a version satisfying the temporal query is found.

Accordingly the format of a posting changes from the traditional <Document ID, term\_freq> to <VID, term\_freq>. Note that in the work reported in [4, 9], each postings includes the life time of a document such as <Document ID, time\_start, time\_end, term\_freq>. Such approach returns the lifetime of a document version directly from the postings list at a very significant space cost since the same start and end times have to be duplicated in the postings lists for all the terms that appear in this document version. Moreover, with our approach previous well-known schemes can be extended to generate the new postings lists such that the postings are sorted in ascending order of VID, and hence can be easily compressed.

Let us now consider how a query with a query time  $t_Q$  can be handled using our indexing structures and the postings lists. A straightforward approach would be to start by retrieving all the matching VIDs from the postings lists of query terms and then determine the temporal validity of each VID by looking up its lifetime using the VID table. This strategy can be easily implemented as effectively as in the traditional

case. However, as mentioned in [9, 63], the postings lists can grow very large, which can introduce a very high overhead when determining the valid VIDs matching the time constraint  $t_Q$  of the query. To tackle this problem, the authors of [9, 63] propose an approach based on splitting each postings list into many segments after the postings lists are constructed. Our approach will be very different. As we are building the temporal inverted files, we will partition each postings list into different segments on the fly such that each segment belongs to a certain time span determined dynamically and all the postings lists will have the same temporal partition. We refer to the time span covered by a segment as a *time window*, which is somewhat similar to the time window notion introduced in [63]. A time window  $[TW\_start, TW\_end)$  consists of one or more elementary time steps such that each valid elementary time step belongs to a time window and no two time windows overlap. A document version  $VID_{ij}$  is considered alive in this time window if there is an overlap between  $[t_{ij}, t_{i,j+1})$  and  $[TW\_start, TW\_end]$ , i.e.  $VID_{ij}$  is alive for some time within this time window.

In our approach, each time window has its separate postings lists (segments) and VID table corresponding to all the document versions that are alive within this time window. Query can now be localized to a single time window, which substantially reduces the search time of the retrieval algorithm. However, this comes at the cost of storing some duplicate postings in consecutive time windows due to the fact that some document versions will be alive across a time boundary. This implies that the postings of a document version will be replicated through all the time windows in which this document version stays alive. Therefore a long-live document version may

appear in many time windows and hence will generate multiple copies of the same postings in the output. However as we will see later, this overhead is relatively small overall while, on the other hand, the search time is dramatically improved.

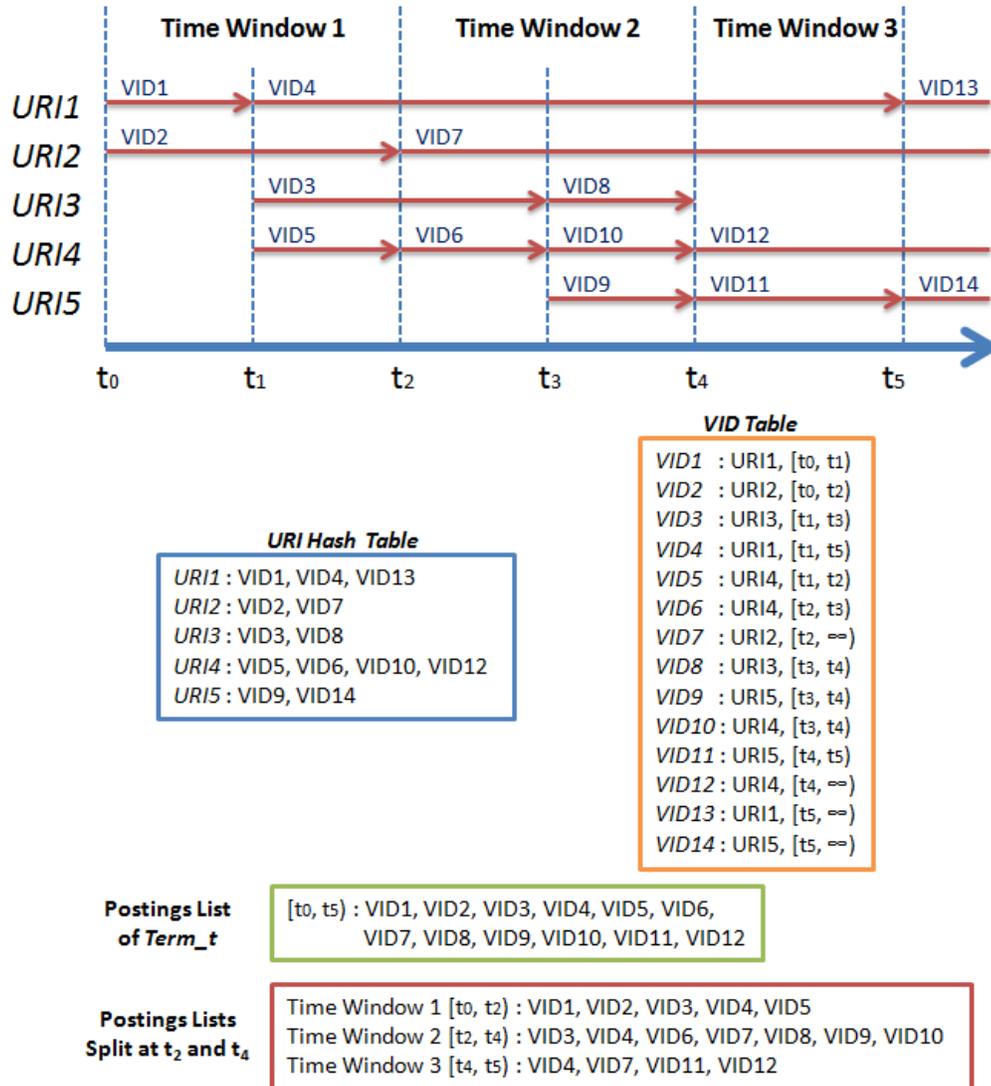


Figure 4.1: An Example of Temporal Indexing

Let us consider a small example to fix the ideas. A collection with five documents is illustrated in Figure 4.1, where all documents contain term  $Term_t$  and time stamp  $\infty$  means it is still alive. Time  $t_2$  and  $t_4$  are the starting points of new time window and document version  $VID4$  is alive in all time windows, so it has footprints in the

postings lists in all three segments of postings lists.

At one extreme, we can set each time window to be the same as an elementary time step, and hence a search will be immediately localized to the corresponding postings lists and the VID table at an elementary time step, which can be carried out quite fast (best possible search time). However this will come at an extremely large space cost for the output files. The other extreme is to have a single time window covering all the elementary time steps, which will correspond to the smallest possible output file, but with generally unacceptable search time through very large postings lists. Therefore we need to strike a compromise between these two extremes and try to reach a balance between the query response time and output inverted index size.

We consider three possible strategies for determining the appropriate window size.

- ***Even-Time***, which splits the document collection into several time windows such that every time window covers the same length of time period;
- ***Even-Size***, which monitors the total uncompressed size of input files such that a new time window starts when the collection size managed under the current time window exceeds a threshold;
- ***Output-Oriented***, which forces an upper bound on the size of the output postings lists.

The first strategy was used in [63]. For example, the 36-month Congressional document collection (to be introduced in Table 4.1) is divided into three 12-month time windows. The second strategy attempts to ensure that the collection size processed during each time window is about the same, and hence the output postings

lists from different time windows are expected to be of similar size. Note, however, that the final postings lists for a time window also include those inherited from earlier time windows and hence even with the second strategy, the total output size for a time window is only weakly controlled. The third strategy presents the best option, which is similar to the performance-guaranteed and space-bound approach derived in [9]. In practice, this strategy incurs the largest overhead and its implementation is not straightforward.

### 4.3.3 Query Strategy

In this section, we take a look at the temporal query strategy details. Given all the necessary temporal inverted files residing on disks, we can handle a temporal query with query terms  $\{Q_k\}$  and query time  $t_Q$  through the execution of the following steps:

- **Step 1:** *Initialization of Search engine*
- **Step 2:** *For each temporal query  $\langle Q_k, t_Q \rangle$ , do:*
  - **Step 2.1:** *Determine the corresponding time window  $TW$  and retrieve the average document length at  $t_Q$ ;*
  - **Step 2.2:** *Parse the query string into individual query terms;*
  - **Step 2.3:** *Search all the query terms in the dictionary to locate their postings lists;*
  - **Step 2.4:** *Fetch data of all query terms from disks:*
    - **Step 2.4.1:** *Fetch the document frequency of each query term at  $t_Q$ ;*
    - **Step 2.4.2:** *Fetch and decompress the postings list of each query term in  $TW$ ;*
  - **Step 2.5:** *Merge the postings lists to find all matching documents and*

*compute scores;*

- **Step 2.5.1:** *Merge the postings lists of all query terms to find out common VIDs;*
  - **Step 2.5.2:** *Check the validity of VIDs at  $t_Q$ ;*
  - **Step 2.5.3:** *Fetch the document lengths of the valid VIDs from the VID table;*
  - **Step 2.5.4:** *Compute scores for the valid VIDs;*
- **Step 2.6:** *Find the top  $K$  document versions with highest scores and display their URIs;*

In Step 1, we load the dictionary from disk into memory and the time span of each time window and elementary time step (ETS) such that given a query time  $t_Q$ , we can immediately find which time window and ETS this  $t_Q$  falls into. We also assume that the VID table is pre-loaded into memory in this step for a fair comparison with the query performance in non-temporal case. It is because in order to retrieve the document length in Step 2.5.3 and the document title, URI or file location of the searching results in Step 2.6, a similar data structure is accessed in non-temporal case as well. As a result, we apply the setting that in either temporal or non-temporal cases such data structure (VID table in our temporal case) resides in the main memory, just as the dictionary.

Steps 2.1, 2.4.1 and 2.5.2 are unique to the temporal query scenario. Step 2.1 is trivial in general because the total number of elementary time steps is typically not large, for example, 1M elementary time steps means over 20 years of hourly snapshots. Steps 2.5.2 and 2.5.3 involve accessing the VID table but as we explained

in the last paragraph that the document length and file location still has to be obtained from some data structure in the non-temporal case and hence we consider these two steps as not introducing any extra cost. The overhead of Step 2.4.1 will be fully analyzed later.

We use the well-known Okapi BM25 function [59] to compute the scores in Step 2.5.4:

$$\text{score}(D, Q) = \sum_{i=1}^n \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})},$$

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

where  $N$  is the total number of documents,  $n(q_i)$  is the number of documents with term  $q_i$ ,  $f(q_i, D)$  is the term frequency in  $D$ ,  $|D|$  is the length of  $D$ ,  $\text{avgdl}$  is the average document length in the collection,  $k_1$  and  $b$  are two parameters and we use  $k_1 = 2.0$  and  $b = 0.75$ .

No approximation or lossy compression is used in our algorithm, and hence the scores and ranking results of the temporal query reflect the exact scores given the collection of documents that are live at time  $t_Q$ .

## 4.4 Experimental Evaluation

We use two significant collections that exhibit different characteristics to evaluate our strategy. The first collection is the Congressional dataset from the Library of Congress, which includes 36 monthly snapshots of selected news and government websites crawled between January 2004 and December 2006. The second dataset is the Wikipedia01-07 data we already used in Chapter 2. The overall characteristics of the two benchmarks are given in Table 4.1.

The generated output, dictionary, two temporal tables and postings lists, are written onto local disks.

Table 4.1: Statistics of Document Collections

	<b>Congressional</b>	<b>Wikipedia 01-07</b>
<b>Compressed Size</b>	314GB	29GB
<b>Uncompressed Size</b>	1,672GB	79GB
<b>Crawl Time</b>	01/04 to 12/06	02/01 to 12/07
<b>Document Number</b>	28,731,237	2,077,745
<b>Document Version Number</b>	80,092,737	16,618,497
<b>Average Versions per Document</b>	2.788	7.998
<b>Number of Terms</b>	12,229,793	9,404,723
<b>Number of Tokens</b>	41,356,310,380	9,375,229,726

To evaluate our approach, we estimate the space overhead due to the introduction of the new indexing structures, the URI hash table and the VID table, and then query response time relative to the time windowing strategy used. We note that we will show later in Section 4.6 that the processing time to create the inverted files with the extra indexing structures is very close to the traditional strategy for processing the same collection while ignoring the temporal information.

We start by taking a close look at our implementation of the new indexing structures.

#### 4.4.1 URI Hash Table and VID Table: Implementation Details

At any time during the initial processing of the document collection, the URI hash table is used to map a URI into a list of VIDs representing all the versions encountered thus far of the specific document identified by the unique URI. If multiple URIs hash to the same value, then a singly linked list is created to store these URIs in a sorted order for fast insertion of new URI since we can determine if a URI

exists in the linked list or not before traversing the entire list.

On the other hand, the VID table stores, for each valid VID, specific information pertaining to that document version such as: start time  $t_{start}$ , end time  $t_{end}$ , document length, pointer to the corresponding URI string, and the location of this document version within the underlying file system. In the non-temporal case, a similar data structure is typically constructed as well to fetch this type of information pertaining to a document except for the temporal information. A significant difference is that during the indexing process an entry of the VID table can be written twice since the end time of a VID has to be updated in the future and be read multiple times whenever we need to check the validity of the VIDs. On the other hand in the non-temporal case, such entry is completely built during its creation and will not be read or written any more during the indexing process.

When a document version is processed given its  $\{URI_i, VID_{ij}, t_{ij}, Doc\_Length\}$ , the URI hash table is first visited to fetch most recent VID corresponding to this URI, that is,  $VID_{i,j-1}$  which belongs to the latest document version processed just before this newer version. In the VID table, the  $t_{end}$  of  $VID_{i,j-1}$  can now be set to  $t_{ij}$ . Then,  $VID_{ij}$  is inserted at the end of the list in the URI hash table indicating that this is now the most recent version. The VID table also stores the information regarding  $VID_{ij}$  except  $t_{end}$  which cannot be determined until a newer one  $VID_{i,j+1}$  is encountered. In the temporal inverted files construction process, the URI hash table assists to find the  $VID_{i,j-1}$  for the  $VID_{ij}$ ; during query processing, the URI hash table helps us find all the versions for a specific document. In practice, we only store the last and most recent VID of the list in main memory and flush older ones onto disks because only

this one is accessed during the indexing process.

As time moves forward, we will only be interested in those document versions that are still alive, i.e. the VIDs whose  $t\_end$  fields are blank (set to  $\infty$ ) in the VID table. With a simple table consisting of consecutive VIDs, it is expected to see that most of the lower VIDs are not alive while most of the most recently assigned higher VIDs are still alive. To keep the size of the VID table small, residing in memory, we split the table into two parts, where **Part I** contains the consecutive VIDs for higher VIDs and **Part II** contains nonconsecutive VIDs for lower VIDs.

The Part I can be indexed directly by the VID so that the lookup time is very small, but it will contain more and more VIDs that will never be accessed as time moves forward. On the other hand, Part II is stored in a more compact fashion, in such a way that binary search can be used to look up an entry corresponding to a VID. In our current setting, if the size of Part I exceeds 8M at regular check points (for every 100GB input data or at the end of a time window in our case), only the highest (also the most recent) 8M VID entries stay in Part I and the remaining entries are migrated to Part II followed by a scan on Part II to remove the useless VIDs which are written onto disks (and hence the size of Part II shrinks considerably).

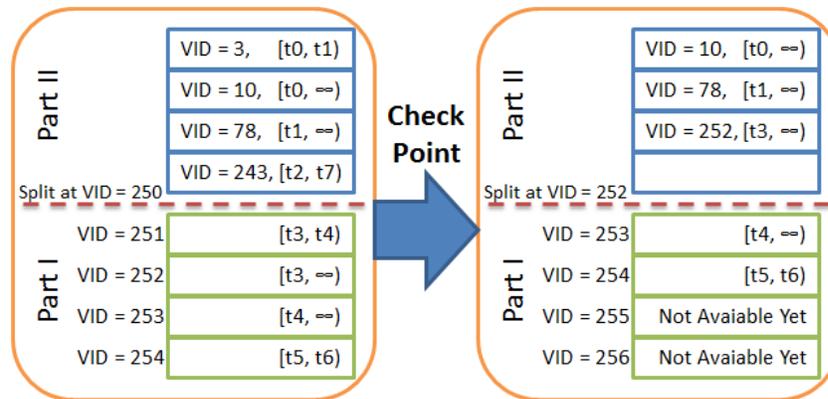


Figure 4.2: Example of the VID Table Data Structure

An example on the data structure of the VID table is shown in Figure 4.2. In Part I, VIDs are implicit and can be calculated by the sum of an offset and table index; on the other hand, Part II stores the VIDs inside the table memory space explicitly. After the check point mentioned above, both Part I and Part II are reorganized in order to achieve a balance between memory space and table search time.

#### 4.4.2 Growth of URI Hash Table and VID Table

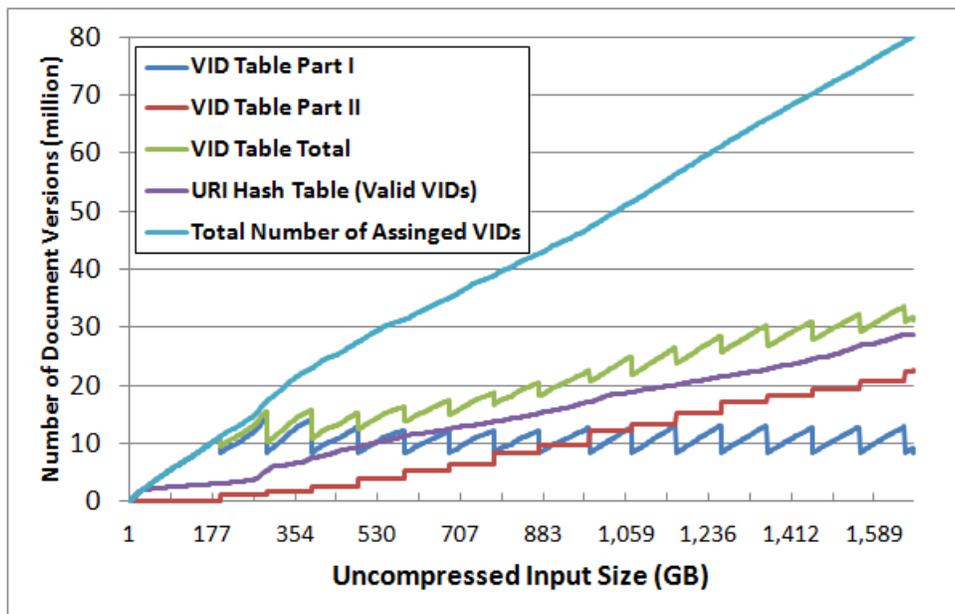


Figure 4.3: Growth of URI Hash Table and VID Table

We now take a closer look at how the dynamic data structure of the VID table contributes to saving space. The VID tables are checked periodically in such a way that invalid VIDs are written onto disks and some VIDs are migrated from the VID table Part I to Part II. As a result, after each checkpoint the size of the VID table Part II increases and the size of the VID table Part I shrinks back to 8M entries as shown in Figure 4.3. On the other hand, the number of VIDs stored in the URI hash table grows steadily over time. The total number of VIDs remaining in the VID table is

always larger than the number of valid VIDs, which is due to some invalid VIDs that are kept in Part I. The top line in Figure 4.3 shows the total number of assigned VIDs, which also reflects the size of the entire VID table in the temporal case and the size of corresponding data structure of document information in the non-temporal case. The difference between this top line and the second top line indicates the memory space saved by our VID table data structure.

### 4.4.3 Relationship between Query Performance and the number of Time Windows

In this section, we explore the effects of different strategies for selecting time windows on the sizes of the postings lists and query performance. We will also compare these strategies to the traditional strategy when the temporal component of the document collection is ignored. We use the Even-Size strategy to evaluate the impact of the number of time windows in this section.

#### 4.4.3.1 Size of Postings Lists Relative to the Number of Time Windows

We start by considering the compressed output size as a function of the number of windows for the Library of Congress Congressional dataset. For the extreme case of a single time window, the generated postings lists are exactly the same as those that would be generated had we treated the document collection without paying attention to the temporal information. The output size is minimal in this case since no duplicate documents have to be considered. As the number of time windows increases, we expect the output size to increase since the number of duplicate document versions across the time windows is expected to increase. This is illustrated in the next table.

Note that since we only have 36 monthly snapshots, i.e. 36 elementary time steps, with 36 time windows we actually have one time window for each elementary time step, the same as with the Even-Time strategy.

Table 4.2: Throughput and Output Size over the number of Time Windows

<b>Number of Time Window</b>	<b>Compressed Output PL Size (GB)</b>
1	33.7
3	42.0
9	88.6
18	136.6
36	248.1

#### 4.4.3.2 Detailed Query Processing Time

We now take a close look at the detailed query processing time. To conduct our tests, a thousand queries are tested, each query having a random query time between 01/2004 and 12/2006, with three terms chosen randomly from the dictionary. The search program runs on a CPU with a single thread such that the one thousand queries are processed sequentially. We filter out the 10% slowest query batches and calculate the average of the remaining 90% data. Such filtering is carried out due to two reasons: (1) queries that consist of very popular terms map onto very long postings lists resulting in millions of matching results, which will significantly slow down the speed; (2) the first few queries are generally slower because of the empty caches (including CPU cache, virtual memory and disk cache) are still empty but will later be filled up to accelerate memory and disk accesses. The same approach is used to represent the query processing time in this chapter.

We use the Even-Size strategy to split the entire collection into three time

windows. Table 4.3 lists the average processing times from Step 2.1 to Step 2.6 as well as the overall time cost in Step 2 of the search algorithm presented in Section 4.3.3. We note that the most time consuming part is Step 2.4, in which the postings lists and the document frequencies of the query terms are read into memory. As we have mentioned in Section 4.3.3, the extra cost to processing a temporal query lies in the execution of Step 2.4.1, which is about 105ms for three query terms or 35ms for one query term, which is close to the 25ms random disk seek time from our experiments. Over 98% of the time is spent in executing Steps 2.4 and 2.5, and hence in the sections below we focus on these two steps.

Table 4.3: Query Process Time in Step 2

<b>Step</b>	<b>Time (millisecond)</b>
<b>Step 2.1 Find ETS and TW</b>	0.018
<b>Step 2.2 Query Parse</b>	0.010
<b>Step 2.3 Search Dictionary</b>	0.073
<b>Step 2.4 Fetch Doc_Freq and PL</b>	648.052
<b>Step 2.4.1 Fetch Term Doc_Freq</b>	105.905
<b>Step 2.4.2 Fetch PL</b>	540.517
<b>Step 2.5 Merge and Score</b>	80.539
<b>Step 2.6 Display Top 100 Results</b>	10.036
<b>Total</b>	741.894

#### 4.4.3.3 Query Performance Relative to the Number of Time Windows

The main reason for introducing multiple time windows is to reduce query response time by localizing the search into a small time frame that generates relatively short postings lists. We expect the number of time windows to primarily influence the execution of Steps 2.4 and 2.5 since these steps process the postings lists corresponding to the query terms. The size of the VID table decreases as well

with the number of time windows, which makes possible for the VID table to fit into the CPU cache and thereby accelerate the execution of Steps 2.5.2 and 2.5.3.

Table 4.4: Query Performance as a Function of the Number of Time Windows

<b>Number of Time Windows</b>	<b>Step 2.4 Fetch Doc_Freq and PL (millisecond)</b>	<b>Step 2.5 Merge and Score (millisecond)</b>	<b>Average Size of Compressed PLs for One Time Window (GB)</b>
1	2383.81	126.91	33.7
3	648.05	80.54	14.0
9	372.66	28.07	9.8
18	310.21	21.79	7.6
36	300.64	21.35	6.9

Table 4.4 shows the execution times of Steps 2.4 and 2.5. When the number of time windows is larger than 9, very limited performance gain is obtained because the average size of postings lists for a time window decreases very slowly beyond. The size of postings lists from newly arrived document versions is roughly proportional to the length of the current time window so when new time windows are created frequently resulting in many short time windows, the new postings generated within a time window are relatively few and the dominating factor will be the postings lists passed from the previous time window. Consequently, the average size of postings lists in a time window decreases slowly with a large number of time windows. On the other hand, the total size of postings lists from all time windows grows quickly with a large number of time windows (from 9 to 36) with limited improvement on query performance. Therefore, based on these experimental results, choosing three or nine time windows achieves a balance between space and performance for this Congressional dataset.

#### 4.4.3.4 Summary

Figure 4.4 shows the relationship between the overall query response time, the sizes of postings lists, and the size of the additional temporal indexes as a function of the number of time windows, assuming the Even-Size strategy. Comparing the non-temporal case and the temporal case with a single time window, we can see that only a small increase in query response time is introduced the temporal case, and hence the query performance does not suffer due to the additional temporal indexing structures even though it can now handle the temporal queries. In fact it can be improved using finer temporal partitioning of the collection.

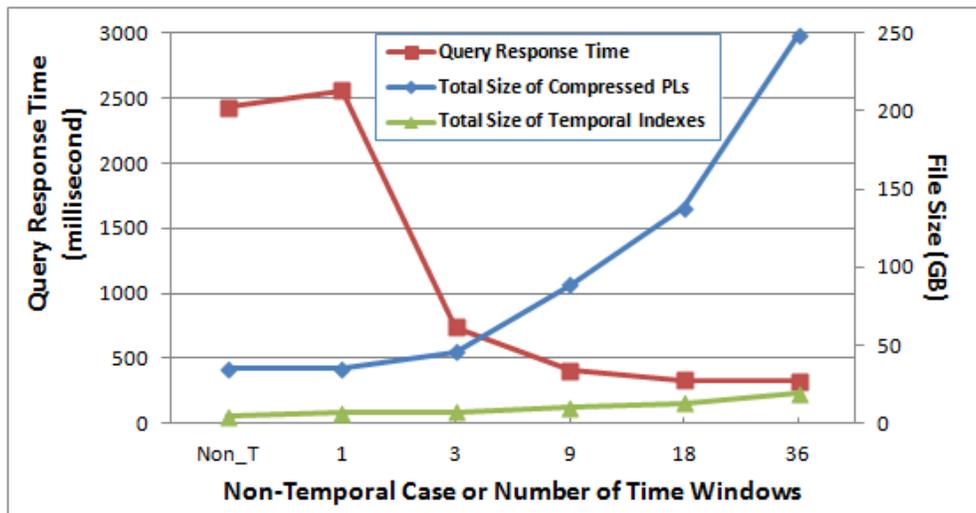


Figure 4.4: Impact of Time Window Numbers on Index Size and Query Performance

#### 4.4.4 Relationship between Query Performance and Different Time Window Strategies

In this section, we compare the query performance under the two distinct strategies introduced earlier, the Even-Size strategy and the Even-Time strategy. By the Even-Size strategy, the resultant three time windows for the Congressional dataset

cover respectively 19 months (01/2004 to 07/2005), 10 months (08/2005 to 05/2006) and 7 months (06/2006 to 12/2006), and each time window consists of about 560GB of input. On the other hand, the Even-Time strategy splits the 36 monthly snapshots into three time windows, each of which covers 12 months. Table 4.5 shows the sizes of the generated postings lists after compression. Under the Even-Size strategy, the number grows slowly because of the increasing size of postings lists inherited from previous time window is increasing relatively slowly. Since the web is steadily increasing in size, it is natural to find that each monthly snapshot in 2006 is much larger than the one in 2004 and hence the Even-Time strategy results in fast increasing size of output postings lists.

Table 4.5: Size of Output Size with Different Time Window Splitting Strategies

Time Window	Compressed Size of Output PLs (GB)	
	Even-Size	Even-Time
1	12.6	5.4
2	13.9	13.2
3	15.5	22.3

As for the query performance, the average query response time remains almost the same in either strategy if we randomly choose a query time between 01/2004 and 12/2006. However, if we look deeper and capture the average time required to process a query during Steps 2.4 and 2.5 with query time in year 2004, 2005 or 2006 separately, some interesting but expected results emerge as shown in Table 4.6. Since the size of output postings lists grows quickly from 2004 to 2006 with the Even-Time strategy, the resultant execution time of Step 2.4 and Step 2.5 also increase at a similar rate. This leads to a query response time that fluctuates drastically. In practice, this trend is likely to get worse if we assume that users will be more interested in

recent document versions and more temporal queries fall into year 2006. On the other hand, the Even-Size strategy yields much more stable results for both Steps 2.4 and 2.5. The time cost is still growing as the size of output postings lists is increasing as indicated in Table 4.5.

Table 4.6: Query Performance with Different Time Window Splitting Strategies

Query Time Range	Step 2.4 Fetch Doc_Freq and PL (millisecond)		Step 2.5 Merge and Score (millisecond)	
	Even-Size	Even-Time	Even-Size	Even-Time
2004	580.59	336.12	76.92	37.37
2005	614.21	655.24	81.39	76.02
2006	748.95	1124.66	84.30	99.78

## 4.5 Parallel Algorithms for Constructing the Inverted Files in the Temporal Case

Our goal is to build the dictionary, the URI hash table and the VID table, as well as the postings lists stored on external storage with the maximum possible throughput without sacrificing latency and throughput of query processing. What’s more, the algorithm should have the property of incremental updating, which means that more data are available for newer time windows as time goes by and our algorithm is able to update the existing temporal inverted files with minimal overhead.

### 4.5.1 Algorithm on a Single Multicore Node

The starting point of our algorithm on a single node of multicore processors is an extension to the algorithm presented in Chapter 2. To accommodate temporal information from the archived data collection, some new components and techniques are added into the pipeline described in Chapter 2.

#### 4.5.1.1 Overall Approach

In the temporal case, parsers also pass the URI and crawl date of each VID to the indexers to build the temporal inverted files. In the indexing stage, the main thread pulls parsed results from the buffer as soon as they are available, builds the URI hash table and the VID table before spawn a number of indexers which jointly construct the dictionary and postings lists. The extra workload here is trivial compared to the one of non-temporal indexers since for one document version the above two temporal tables are updated only once but there are hundreds of tokens need to be inserted into the dictionary and postings lists. Therefore the main indexing thread carries out this task instead of creating a separate thread.

Similar to the dictionary, the URI hash table and the VID table remain in main memory until the entire data collection has been processed; however the two temporal tables are checked periodically such that only a small portion of VID entries are kept in main memory as described in Section 4.4.1; the postings lists are kept in memory first and written into a disk as temporary files before the memory is full. When the current time window is closed, a number of parallel temporal mergers read these temporary files as well as the postings lists inherited from previous time window from the disks, and generate the postings lists for both current time window and next time window. The number of parsers, indexers and temporal mergers are determined depending on the physical resources available later in Section 4.6.1.

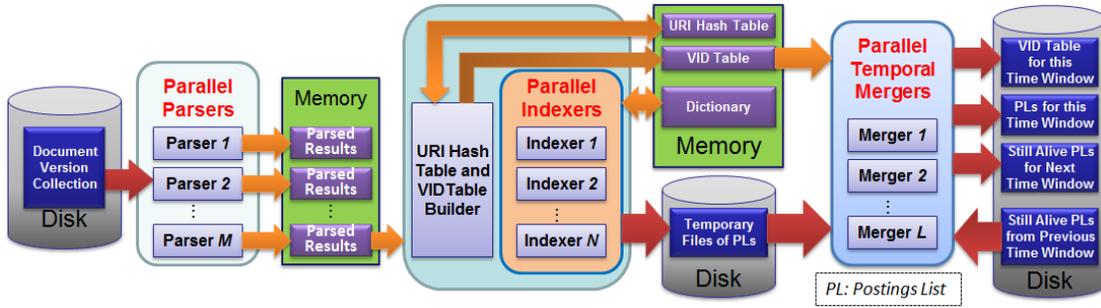


Figure 4.5: Overall Data Flow on a Single Node

#### 4.5.1.2 Temporal Merging Stage

The temporal merging stage is invoked only at the time window boundary, where the current time window closes and a new one starts.

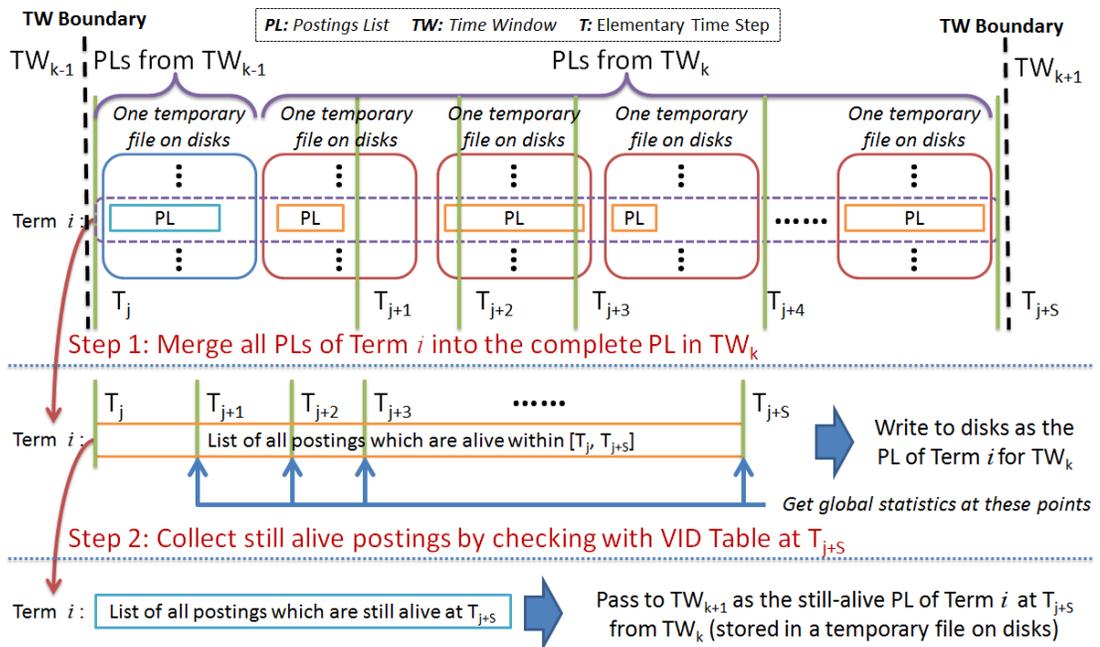


Figure 4.6: Data Flow in the Temporal Merging Stage

The input data to temporal mergers are one temporary file passed from time window  $TW_{k-1}$  and several temporary files generated in this time window  $TW_k$ . A temporal merger processes one term at a time such that the postings list of this single term can fit into memory. It starts by reading out all the partial postings lists from the

above temporary files. These postings lists are then combined into the complete postings list of this term in  $TW_k$ , which contains all postings that are alive within time  $[T_j, T_{j+S})$ . From this complete list, we can get useful global statistics at each elementary time step for this term. For example, the document frequency of this term at  $T_{j+2}$  can be obtained by counting the number of postings that are alive at  $T_{j+2}$ . These postings lists of all terms are written to disks which can be used to handle the queries within  $TW_k$ . For one elementary time step there is only one value of average document length but the document frequency is per term and thereby requires certain amount of space on disks. In Section 4.4.3 we already saw the cost in the querying process from accessing the document frequencies of query terms.

In the next step, we check the postings in this complete list of this term to find out all postings which are still alive at the end of  $TW_k$ , that is, time  $T_{j+S}$ . The temporary file containing such shortened lists of all terms is therefore passed to the next time window  $TW_{k+1}$ , just as the one passed from  $TW_{k-1}$  to construct the postings lists in  $TW_k$ .

Even though temporal mergers are reading and writing multiple files at the same time, each file is accessed contiguously. It is because when the postings list of Term  $i$  is finished the temporal merger moves onto Term  $i+1$  whose file locations are in neighboring spaces of Term  $i$  in both input and output files. Awareness of this point, we use buffered file read and write (a typical buffer size is 10MB) to alleviate disk read/write conflicts.

When the entire data collection is ingested and the last time window is completed by temporal mergers, similar to [51] we will have two types of postings lists

generated:

- ***postings lists for each time window:*** *postings lists containing all the document versions that are alive in this time window, which are enough to serve the query that falls into this time window.*
- ***postings lists of still alive VIDs:*** *the temporary file of postings lists containing still alive postings, as the byproduct from the last time window, is also useful to search the most up to date version of a document if it still exists. Such results are the ones used by today's search engines which are only interested in web pages from valid URIs.*

The first ones are unique to temporal inverted files. The second type of postings lists is the one used by today's major search engines and critical component of versioned documents database. In the scenario of incremental updating, when the current time window is still open to take in more elementary time steps, the still alive or most recent document versions may appear in both the temporary file passed from last time window or the ones generated from elementary time steps within current time window. Therefore there exists some overhead to process the queries for still alive document versions but such extra costs are well controlled once we close the current time window.

#### 4.5.2 Algorithm on a Cluster

We now extend our single node strategy to a cluster of multicore processors, similar to the approach in Chapter 3. There are several possible strategies for such extension:

- ***Partition-by-Time-Window.*** *Each node fetches the input files corresponding to*

*one time window, builds the local dictionaries and postings lists for this time window. The merging phase is not needed for temporal queries because we only search within one time window for a query.*

- ***Partition-by-URI.*** *At the end of each parsing stage, each indexer processes an equal portion of the documents by hashing on their URIs following the single node algorithm, after which the local dictionaries, local URI hash table, local VID table, and local 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-by-Trie-Collection.*** *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. On the other hand, temporal information for building the URI hash table and the VID table is distributed to indexers by hashing on the URIs and each indexer builds local URI hash table and local VID table on its assigned portion of URIs. A synchronization of VID tables is needed to obtain the global VID table when a time window is closed.*

It is clear that the first strategy will achieve excellent performance during the each batch of one time window because every node will work independently on its portion of the time windows with no communication required between the nodes. However the temporal mergers can't finalize the current time window until they receive the still alive postings from previous time window, which are required to build the complete postings lists of the current time window. As a result, the lower bound of the running

time is  $\sum(\text{time in the merging stage in each time window})$  which is not inversely related to the cluster size. Such temporal dependency stops us from getting speedups on the cluster.

The bottleneck of the second strategy lies in the global merging stage. Such merging is required otherwise for a given query term we have to search the same term in all the local dictionaries and the local postings lists to find all matching results. And if newer data are ingested in the future, we have to carry out such merging over and over again, which doesn't meet our requirements.

The last approach is extended from the partition-and-index approach described in Section 3.1 with the advantage of a coherent, global dictionary and the global VID tables stored in the postings lists on all nodes. The global synchronization of VID tables will bring in some overhead but it is well hidden in our pipeline design. As a result, we choose this one to move forward.

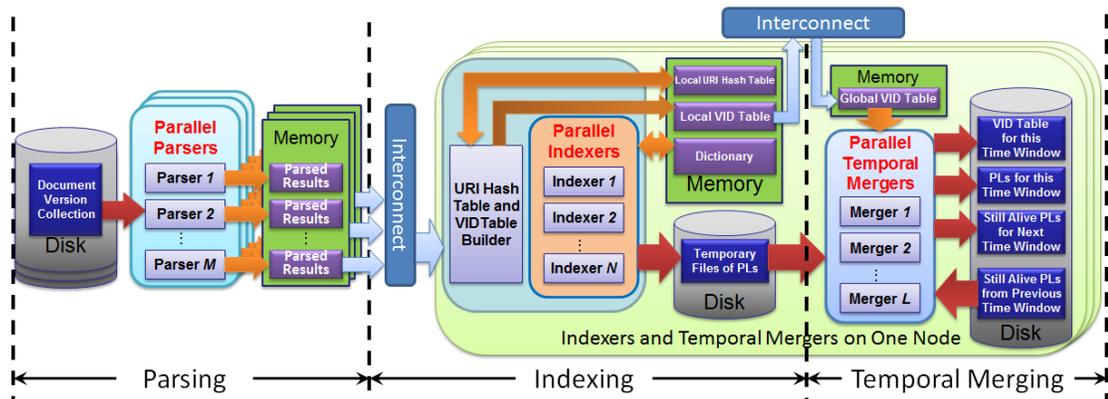


Figure 4.7: Data Flow of Partition-by-Trie-Collection-and-URI Strategy

The data flow of the third approach is illustrated in Figure 4.7. The indexers and temporal mergers still share the same disks so the temporary files in Figure 4.6 will be written and read locally; however, at time window boundaries the temporal

mergers on different nodes have to exchange their local VID tables to obtain the global VID table.

Our algorithm on the cluster distributes the building of URI hash table and VID table evenly among the nodes by hashing on the URIs. This is applicable because only VIDs crawled from the same URI will have temporal relations. However, when a time window closes these distributed VID tables of all nodes are synchronized to reflect the complete and global view of all VIDs. It is a required step because the postings lists generated by the indexers on different nodes are not partitioned by URIs. Consequently, the postings handled by a temporal merger on one node may contain any VIDs occurred on any node, not necessarily restricted to the VIDs assigned to this node by the URI hashing function. On the other hand, the local URI hash tables are not synchronized since they are not used in the temporal merging stage due to the fact that they contain neither the start time nor the end time of VIDs which are required by temporal mergers to check if a posting is still alive or not.

One alternative implementation is to let all nodes build their own copies of the global URI hash table and VID table from the parsed results. Such approach eliminates synchronization by duplicating computation on every node. If the average time to build these two temporal tables is  $T_1$  and the average indexing time is  $T_2$  for each batch, then the processing time of the indexers with  $P$  nodes becomes  $T_1 + T_2/P$ . When  $P$  is over eight in our experiments,  $T_1$  will be dominating so the indexing speed hardly increases with larger  $P$ . In our algorithm, the processing time is  $(T_1 + T_2)/P$ , which is scalable. The temporal mergers are responsible for this synchronization so parsers and indexers will not be stalled due to this communication operation.

Putting all the pieces together, we get the overall data flow shown in Figure 4.7 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  $P$  to guarantee the order of VIDs in the postings lists. Indexers on different nodes start indexing once the parsed results are received via the interconnect and they don't necessarily communicate with other indexers; on the other hand, all the temporal mergers will begin at the same time once the VID table synchronization is done.

We also believe that our extension from non-temporal indexing case to temporal indexing case can be applied to other indexers running on the clusters with minor adjustment since both the dictionary and the postings lists remain the same. The two MapReduce implementations in [43] and [19, 39] also distribute different terms to different Reduce workers, similar to the Partition-by-Trie-Collection approach in this chapter. The URI hash table and VID table can be directly constructed by the MapReduce framework by letting each Map worker send out  $\langle \{URI, t\_start\}, VID \rangle$  tuples to every node. Because the MapReduce runtime will sort all these tuples first by URI and then by  $t\_start$ , Reduce workers will directly receive  $[\{t\_start_1, VID_1\}, \{t\_start_2, VID_2\} \dots]$  for each URI and  $t\_start_2$  is actually the end time of  $VID_1$ . Thereby the complete VID table is obtained by simple computation. Then the Reduce workers can partition the postings list of a term into several time windows as our algorithm by referring to the VID table.

## 4.6 Experimental Results of Our Algorithm

In what follows, we start by exploring the values of the numbers of parsers, indexers and temporal mergers for the single node algorithm (described in Section 4.5.1). We then show that our cluster algorithm is scalable relative to the optimized single node algorithm, up to the largest number of available nodes, as well as the performance of our single node algorithm on the two document collections in Table 4.1.

### 4.6.1 Number of Parallel Parsers, Indexers and Temporal Mergers on a Single Node

For simplicity we use the triplet  $\langle M, N, L \rangle$  to represent the combination of  $M$  parsers,  $N$  indexers and  $L$  temporal mergers in the temporal case and the tuple  $\langle M, N \rangle$  for the combination of  $M$  parsers and  $N$  indexers in the non-temporal case. To determine this parameter with the new temporal mergers, we conduct tests based on the following rules:

- *1) the total number of parsers and indexers is at most eight beyond which these parsers and indexers will compete for the available cores and hence hurt the overall performance instead;*
- *2) the tasks of temporal mergers are mainly disk I/Os, so we let indexers and temporal mergers share the same cores if the total number of running threads exceed eight;*
- *3) two indexers are fast enough to keep up with up to six parsers but one indexer will play as the bottleneck as shown in Section 2.4.1, so we fix the*

number of indexers to two;

- 4) in the indexing stage, each indexer generates its own portion of postings lists and it is very hard to have multiple temporal mergers process one indexer's output, thus the postings lists from one indexer will only be processed by one temporal merger, which means  $L$  must be less than  $N$ .

Then we have five reasonable combinations of parsers, indexers and temporal mergers:  $\langle 6, 2, 2 \rangle$ ,  $\langle 6, 2, 1 \rangle$ ,  $\langle 5, 2, 2 \rangle$ ,  $\langle 5, 2, 1 \rangle$ ,  $\langle 4, 2, 2 \rangle$ . The Congressional dataset is used and three time windows are created under the Even-Size strategy to split the inverted files. To show the extra cost from temporal processing, the throughput in the non-temporal case using  $\langle 6, 2 \rangle$  is also included, where we use the algorithms in Chapter 2 and treat every document version in the collection as a new and independent document. Figure 4.8 illustrates that with  $\langle 6, 2, 2 \rangle$  our temporal algorithm reaches over 60% of the throughput from non-temporal algorithm with  $\langle 6, 2 \rangle$ . This means that we have developed the temporal algorithm with reasonable overhead and this efficiency mainly attributes to the careful design of the three-stage pipeline and two effective temporal tables.

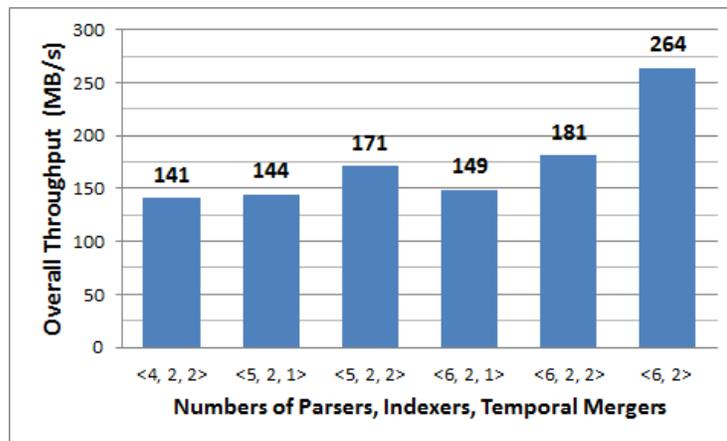


Figure 4.8: Throughput on A Single Node

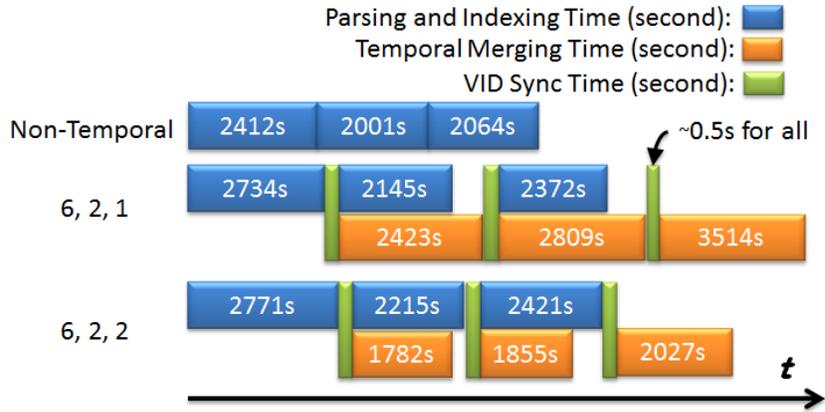


Figure 4.9: Detailed Timeline in the Pipeline with Three Time Windows

Now let's take a deeper look at what happens in the pipeline in Figure 4.9. In the non-temporal case, the parsing and indexing speed is the fastest one in all the three time windows, followed by the temporal case using  $\langle 6, 2, 1 \rangle$ , which is due to the extra work of building two temporal tables and the impact of the new temporal merging thread. The parsing and indexing speed is slowest when using  $\langle 6, 2, 2 \rangle$ , however, the overall throughput is higher than the one by using  $\langle 6, 2, 1 \rangle$  where the much slower temporal merging stage plays as the bottleneck such that parsers and indexers have to wait for temporal mergers to catch up.

#### 4.6.2 Throughput on the Cluster

Now we switch to the algorithm on the cluster and calculate the scalability relative to the number of nodes. The speedup is calculated based on the algorithm on a single node, so the speedup from one node with algorithm on the cluster is less than 1 due to the overhead like MPI communication operations. Please note here the optimal combination of parsers, indexers and temporal mergers is not  $\langle 6, 2, 2 \rangle$  in all scenarios. For example the combination of  $\langle 5, 2, 2 \rangle$  yields a throughput 6% better

than the combination of  $\langle 6, 2, 2 \rangle$  on 32 nodes. Such optimal choice also changes with different time window numbers and splitting methods, but in general the result from  $\langle 6, 2, 2 \rangle$  is very close to the best possible (within 10%). Our main focus is to study the extra cost of temporal processing and for a straightforward comparison with the non-temporal case (where  $\langle 6, 2 \rangle$  is applied) we only choose  $\langle 6, 2, 2 \rangle$  in this section. Again, three time windows are generated under the Even-Size strategy.

#### 4.6.2.1 Scalability over the Number of Nodes

From Table 4.7 and Figure 4.10, we can see that both of our temporal and non-temporal algorithms scale almost linearly with the number of nodes involved. The speedup values in the temporal and non-temporal cases are relatively close and the ratio is around 0.70. In Chapter 3 we have already shown the balanced load between the 32 nodes with the non-temporal indexing algorithm and the stable ratio here indicate the good load balancing with our temporal indexing algorithm.

Table 4.7: Throughput Over the Number of Nodes

Number of Nodes	Temporal (6, 2, 2)		Non-Temporal (6, 2)		Throughput Ratio
	Throughput (MB/s)	Speedup	Throughput (MB/s)	Speedup	
1	172	0.95	249	0.94	0.69
2	312	1.72	480	1.81	0.65
4	619	3.41	935	3.54	0.66
8	1226	6.76	1717	6.50	0.71
16	2382	13.14	3176	12.02	0.75
32	3796	20.93	5401	20.44	0.70

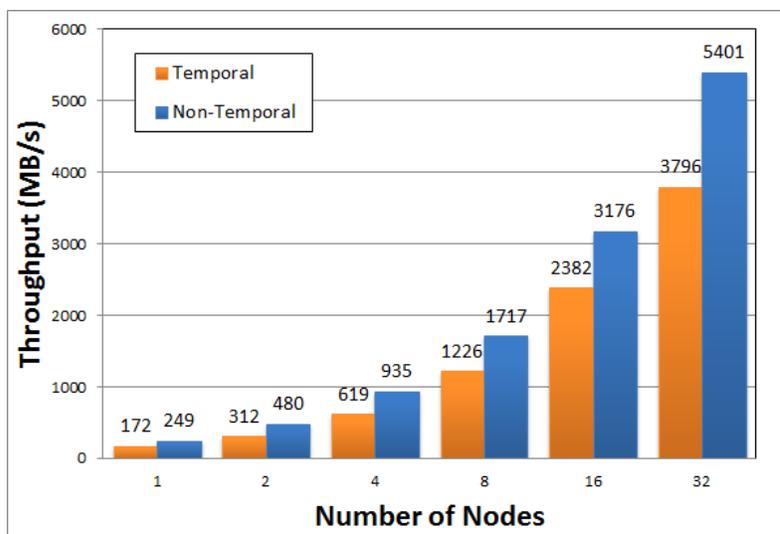


Figure 4.10: Scalability over the Number of Nodes

#### 4.6.2.2 Detailed Running Time of the Pipeline

Figure 4.11 shows the running time for each time window. Similar to Figure 4.9, it takes longer to parsing and indexing the document versions in each time window when indexers and temporal mergers share the same physical CPU core. Compared to the running time on a single node, the VID table synchronization time is also significantly larger because of the all-to-all communication between 32 nodes. Another interesting fact is the temporal merging time, which is over 40 times faster than the one with a single node. This is because that with  $P$  nodes, the sizes of input and output files in Figure 4.6 are only  $1/P$  of the ones with single node, in which case the file accessing caches better assist the file reading and writing throughput to a great extent and disk access conflicts are less likely to happen. In Table 4.8, we list the speedup over the number of nodes by summing up the total running time spent in the temporal merging stage. It's obvious that the super-scalability doesn't show up until the number of nodes reaches eight in which case the size of a temporary file in Figure

4.6 is as low as about tens of MBs, which is close to the 10MB memory buffer size we mentioned in Section 4.5.1.2 for each temporary file.

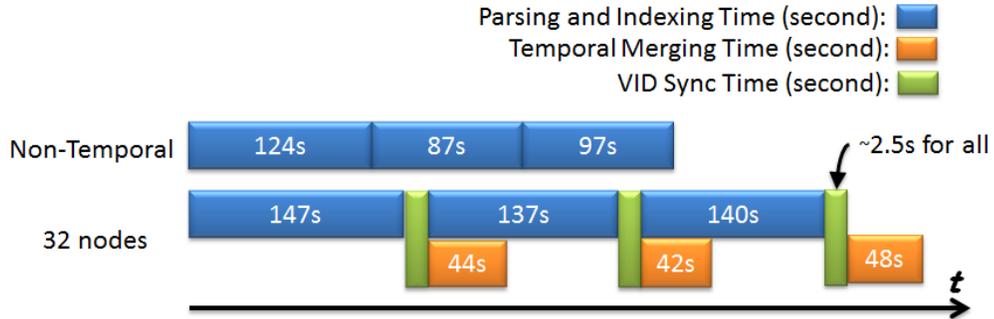


Figure 4.11: Detailed Timeline on the Cluster

Table 4.8: Speedup in the Merging Stage

Number of Nodes	Speedup of Temporal Merging Stage
1	1.00
2	1.63
4	3.75
8	9.41
16	20.29
32	43.76

#### 4.6.2.3 Throughput over the Number of Time Windows

In this section we test the overall running time and throughput with different number of time windows on the cluster with 32 nodes. With only one time window, the process of temporal merging won't start until all parsing and indexing tasks have finished and hence the temporal merging time is not hidden by the parsing and indexing time at all. When the number of time windows increases, the time cost by the last batch of temporal merging (the right most temporal merging time in Figure 4.11) is also reduced, that's why the running time decreases when the number of time window grows from 1 to 9. Beyond that, VIDs that are alive across time window

boundaries are more likely to be duplicated in the postings lists of several time windows, which can be verified by the drastically inflated size of output postings lists. Writing such duplicates onto disks consumes a significant amount of CPU cycles and as a result the overall running time climbs back to even a higher value when there are 18 or 36 time windows.

Table 4.9: Throughput over the Number of Time Windows

<b>Number of Time Window</b>	<b>Running Time (second)</b>	<b>Throughput (MB/s)</b>	<b>Compressed Output PL Size (GB)</b>
1	538	3185	33.7
3	451	3796	42.0
9	408	4196	88.6
18	550	3110	136.6
36	612	2797	248.1

#### 4.6.2.4 Performance Comparison with Fastest Known MapReduce Non-Temporal Indexers

The best way to examine the efficiency of our algorithm on construction of the temporal inverted index is the comparison with other temporal indexers on the same document collection. However, to the best of our knowledge such literature is not available and we instead compare our performance with fastest known non-temporal indexers. The two MapReduce implementations in [19, 43] use the first English Section of ClueWeb09 data collection on which our non-temporal algorithm has a throughput of 6,271MB/s with 32 nodes. Such throughput is close to the 5,401MB/s on the Congressional data collection with the same algorithm. As a result, these two different data collections are considered to have similar statistics and hence it is still

fair to perform the comparison as shown in Table 4.10.

Table 4.10: Platform Configuration and Performance Comparison

		<b>Our Temporal Approach</b>	<b>Ivory MapReduce</b>	<b>Terrier MapReduce</b>
<b>System Details</b>	<b>Processors per Node</b>	Two Intel Xeon 2.8GHz Quad-core CPUs	Two Intel Single-core 2.8GHz CPUs	Two AMD Quad-core Opteron CPUs
	<b>Memory per Node</b>	24GB	4GB	16GB
	<b>File System</b>	Local disks	Hadoop Distributed File System	Hadoop Distributed File System
	<b>Data Collection</b>	Congressional	ClueWeb09 first English	ClueWeb09 first English
<b>Throughput (MB/s)</b>	<b>1 node (8 cores)</b>	181	—	33
	<b>30 nodes (240 cores)</b>	—	—	460
	<b>32 nodes (256 cores)</b>	3796	—	—
	<b>280 Nodes (560 cores)</b>	—	289	—

It's clear that even though our algorithm carries out extra steps and generates much larger and complicated inverted indexes, the throughput is still superior to the ones from MapReduce non-temporal indexers.

#### 4.6.2.5 Performance of our Algorithm on Different Document Collections

We show in Table 4.10 the overall throughput of our algorithm on our two document collections with the parameter  $\langle 6, 2, 2 \rangle$ . For the Wikipedia01-07 collection, the HTML tags were removed, and the remainder is just pure text. As we can see from Table 4.1, the uncompressed size is only 1/21th of the Congressional

collection, yet the number of tokens is about a fourth compared to the one of the Congressional collection. Hence the 50MB/s throughput achieved on Wikipedia01-07 actually amounts to a very high processing speed given the large numbers of tokens. Since the size of Wikipedia01-07 is too small, we can't get any meaningful speedup beyond four nodes and hence those results are not included in Table 4.11. The scalability from single to four nodes is as good as the one on Congressional data and we believe that should we have much larger Wikipedia document version collection, similar speedups as in Table 4.7 can be achieved from 8 to 32 nodes.

Table 4.11: Throughput on Different Document Version Collections

	<b>Throughput (MB/s)</b>			
	<b>Congressional</b>		<b>Wikipedia 01-07</b>	
	Temporal	Non-Temporal	Temporal	Non-Temporal
<b>Single Node</b>	181.35	264.26	51.33	78.29
<b>Four Nodes</b>	618.54	935.08	192.12	242.05

## 4.7 Conclusion

In this chapter, we introduced a new parallel and pipelined strategy for constructing temporal inverted files on a cluster of multicore processors, which is extended from our previous algorithm in the non-temporal case. New temporal indexing structures are developed in such a way that the previous dictionary, postings lists and parsing and indexing strategies all remain the same. Low level parallelism in the underlying hardware is carefully studied to achieve load balancing within parallel thread in each pipeline stage as well as among three pipeline stages. The overall throughputs on two document version collections reach around 70% of the ones with our non-temporal algorithm and are also substantially faster than best know

MapReduce non-temporal implementations. Meanwhile, our query processing time is carefully examined with the more complicated temporal indexing algorithm and time window partitioning approach, showing that our high-throughput indexing algorithm does not come at a cost of slower query performance but actually improve it by finer temporal partitioning of the collection.

## Chapter 5 Applications Accelerated by GPU

Compared to CPU, GPU exhibits different architecture characteristics that include hundreds to thousands of much simpler cores that execute in a SIMD fashion, extremely high memory bandwidth between the cores and the GPU global memory, and a small shared memory on each streaming multiprocessor for fast inter-processor communication. The peak computing powers of GPUs are very impressive but in general the algorithms need to be redesigned to make good use of the available resources. In this chapter, we start by briefly introducing the NVIDIA CUDA GPU, and then develop fast GPU algorithms for the inverted files indexing and list ranking problems.

### 5.1 GPU and CUDA Overview

A surprising facet of the recent evolution of multicore processors is the continued appearance of extremely powerful GPUs that have much better performance to power ratio than the standard multicore CPUs, and that offer increasingly more flexible general purpose programming environments. Examples of such co-processors include the IBM Cell Broadband Engine (Cell BE), the NVIDIA GT200 and Fermi series, and the Intel Larrabee. These multicore processors provide flexible general-purpose programming environments with impressive peak performance. In this chapter, we are primarily concerned about optimization techniques for the NVIDIA CUDA programming model [48] and hence we devote the rest of this section to briefly summarize that programming model and related features.

The basic architecture of the NVIDIA GT200 series consists of a set of Streaming

Multiprocessors (SMs), each of which containing eight Scalar Processors (SPs or CUDA cores) executing in a SIMD fashion, 16,384 registers, and a 16KB of shared memory. The 16KB shared memory is organized into 16 banks. Threads running on the same SM can share data and synchronize limited by the available resources on the SM. Each SM has small constant and texture caches. All the SMs have access to a very high bandwidth global memory; such a bandwidth is achieved only when simultaneous accesses are coalesced into contiguous 16-word lines. However the latency to access the global memory is quite high and is around 400-800 cycles [48].

In our work, we have used the NVIDIA Tesla C1060 that has 30 SMs coupled to a 4GB global memory with a peak bandwidth of 102GB/s. Figure 5.1 and Figure 5.2 illustrate the overall architecture of the Tesla C1060 [40].

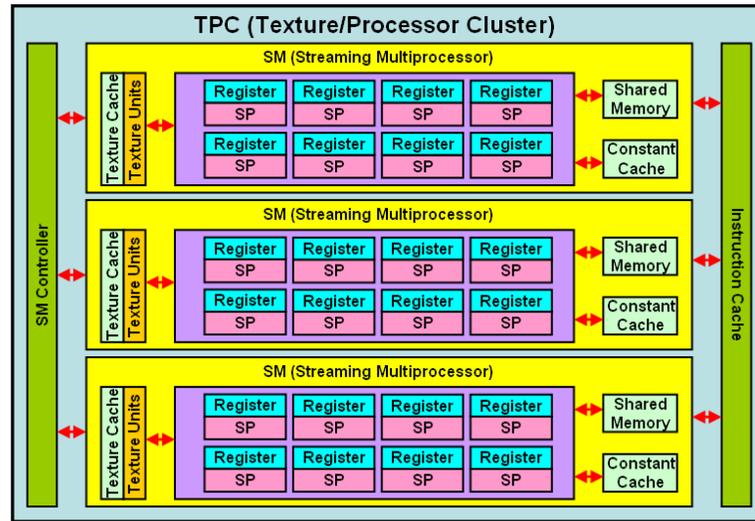


Figure 5.1: Architecture of the Tesla C1060

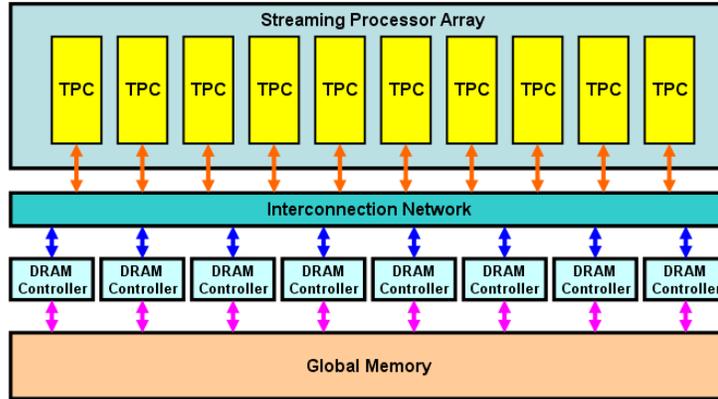


Figure 5.2: Architecture of the TPC in the Tesla C1060

The characteristics of the memory organization of the Tesla C1060 are summarized in Table 5.1.

Table 5.1: Overall memory organization of the Tesla C1060.

	<b>Global Memory</b>	<b>Shared Memory</b>	<b>Constant Cache</b>	<b>Texture Cache</b>
<b>Size</b>	4GB	16KB per SM	8KB per SM	6-8KB per SM
<b>Accessibility</b>	All threads	All threads in the same block	All threads	All threads
<b>Other Properties</b>	High latency	Low latency	Only readable by thread	Only readable by thread

For the newly released Fermi series, several improvements of the GPU architecture have been introduced to provide superior performance. Table 5.2 lists the major changes that are relevant to the works reported in this chapter. The on-chip memory size per SM is increased to 64KB and can be configured as 16KB L1 cache and 48KB shared memory or 48KB L1 cache and 16KB shared memory. The architecture of a single SM in a Fermi GPU can be found in Figure 5.3 [49].

Table 5.2: Comparison between GT200 and Fermi

	<b>GT200</b>	<b>Fermi</b>
<b>Number of SPs per SM</b>	8	32
<b>Warp Scheduler per SM</b>	1	2
<b>Shared Memory per SM</b>	16KB	16KB/48KB
<b>Shared Memory Banks</b>	16	32
<b>L1 cache</b>	None	16KB/48KB
<b>Global Memory L2 Cache</b>	None	768KB
<b>Global Memory Coalescing Size</b>	16-word	32-word

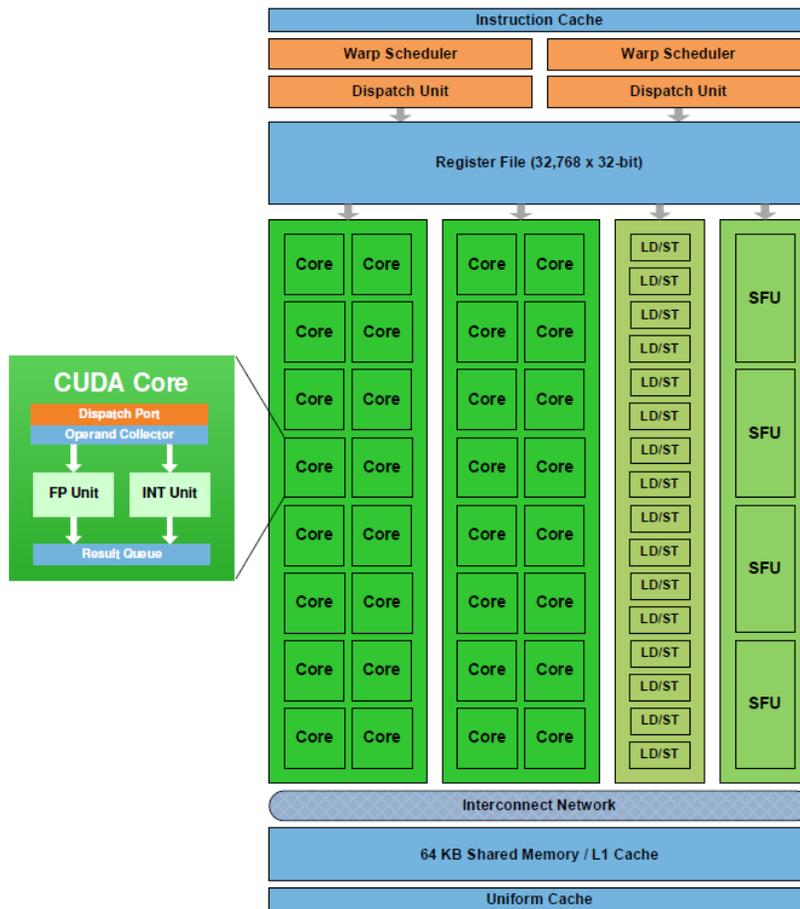


Figure 5.3: Architecture of one SM in Fermi GPU

The CUDA programming model envisions phases of computations running on a host CPU and a massively data parallel GPU acting as a co-processor. The GPU executes data parallel functions called *kernels* using thousands of threads. Each GPU phase is defined by a *grid* consisting of all the threads that execute some kernel

function. Each grid consists of a number of *thread blocks* such that all the threads in a thread block are assigned to the same SM. Several thread blocks can be executed on the same SM, but this will limit the number of threads per thread block since they all have to compete for the resources (registers and shared memory) available on the SM. Programmers need to optimize the use of shared memory and registers among the thread blocks executing on the same SM.

Each SM schedules the execution of its threads into warps, each of which consists of 32 parallel threads. In GT200 and earlier series, half-warp (16 threads), either the first or second half of a warp, is introduced to match the 16 banks of shared memory. When all the warp's operands are available in the shared memory, the SM issues a single instruction for the active threads. All cores in a SM will be fully utilized as long as operands in the shared memory reside in different banks of the shared memory (or access the same location from a bank). If a warp stalls, the SM switches to another warp resident on the same SM.

Optimizing performance of multithreaded computations on CUDA requires careful consideration of global memory accesses (as few as possible and should be coalesced into multiple of contiguous 16-word lines on the GT200 series and 32-word lines on the Fermi series); shared memory accesses (threads in a warp should access different banks); and partitioning of thread blocks among SMs; in addition to carefully designing highly data parallel implementations for all the kernels involved in the computation.

## 5.2 GPU Indexers on a Single Node

In this section, we explore the possibility of GPU acceleration of the algorithm presented in Chapter 2. The parsing process involves mainly finding word boundaries and shortening terms to their root forms, both of which involve complicated sets of branch instructions and hence they are very hard to map onto the GPU. As a result, we only include GPUs in the indexing stage and optimize the throughput by carefully considering hardware features of both CPUs and GPUs.

### 5.2.1 Heterogeneous Implementation of Indexing

The cores on the CPU offer opportunities for a limited amount of parallelism on highly irregular computations. On the other hand, the streaming cores on the GPU are ideally suited for a very high number of fine grain data parallel computations each of which follow the same execution paths. Our approach attempts to exploit both capabilities simultaneously, carefully orchestrating which tasks are assigned to the CPU cores and which are assigned to the GPU streaming cores. More specifically, besides the two Intel Processor Xeon X5560 quad-core CPUs we also use two NVIDIA Tesla GPUs each consisting of 240 streaming cores and 4GB of device memory. The data structure remains the same as the one described in Chapter 2 but the indexing approach is modified and optimized according to the CUDA GPU programming guidelines. Our heterogeneous algorithm can easily be adapted to any other such heterogeneous configuration.

#### 5.2.1.1 GPU Indexer

In the CUDA architecture, 32 threads form a warp and they are executed in a

SIMD fashion. Note that some of the processor cores have to idle if the corresponding threads follow different execution paths. Therefore, we allocate the work to build a single B-tree and the related postings lists corresponding to a single trie collection to a thread block consisting of one warp and coordinate these 32 concurrent threads to jointly build the B-tree.

At this stage, we assume that our term strings are already moved into the device memory and are represented as indicated in Figure 5.4. Without loss of generality, we also assume that no term is longer than 255 bytes and hence one byte will be sufficient to hold the length of the corresponding string. We read these term strings in contiguous chunks (512 bytes) and store them into the shared memory corresponding to the thread block handling this particular trie collection. Hence we are making use of coalesced memory accesses to move the data into the streaming multiprocessor shared memory. The GPU threads will then access the shared memory to process the corresponding terms instead of accessing the device memory.

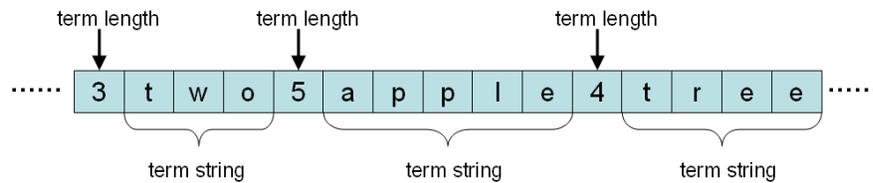


Figure 5.4: String Representation: Term Length in the First Byte

Each term is now inserted into the B-tree using the 32 threads as follows. Starting from the root and as we go down the B-tree, we move the next B-tree node to be examined into the shared memory using coalesced memory access. We use the available threads to perform a comparison between the term to be inserted and each of the terms stored in the node in parallel. This parallel comparison operation followed

by a parallel reduction step [25] will enable us to identify the location of the new term as indicated in Figure 5.5. If this term needs to be inserted in the current position of term  $(i+1)$ , then term  $(i+1)$  up to the current last term in the node must be shifted to the right, which is achieved by a number of parallel threads.

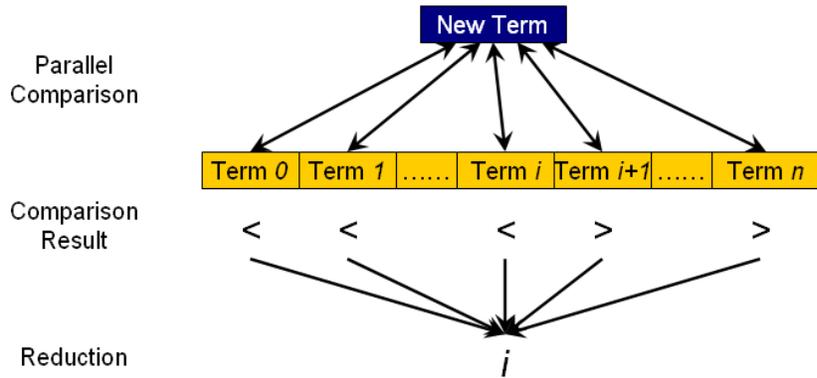


Figure 5.5: Parallel Comparison in One GPU Thread Block

During B-tree insertion, three major operations inside a B-Tree node can take place and they are all carried out in parallel using coalesced device memory accesses.

- **Searching:** *the algorithm compares the new term with existing terms inside the current B-Tree node and then do one of the following: (1) if this term is already present, we update the postings lists; (2) if this term is not there and this node is not a leaf, we proceed to the corresponding child node for searching; (3) if this term is not there and this node is a leaf, we insert this term into this node.*
- **Inserting:** *in order to insert a new term, we must first shift those existing terms which are “larger” than the new term so that a blank location is created to accommodate this new term.*
- **Splitting:** *before accessing a B-Tree node, we check to determine whether this node is full or not and if yes, the node will be split into two nodes.*

We now address the issue of how the trie collections assigned to the GPU will be handled. Since the trie collections are of different sizes and depend on the input documents, any static allocation of these collections to the available thread blocks is likely to incur a serious load imbalance. In our algorithm we use a dynamic round-robin scheduling strategy such as whenever a thread block completes the processing of a particular trie collection, it starts processing the next available trie collection.

### 5.2.1.2 Load Balancing between the CPU and GPU Indexers

In deciding how to allocate the trie collections among the CPU and GPU, we exploit the strength of each architecture—a large cache on the CPU and a high degree of data parallelism on the GPU. We divide the trie collections into two major groups. The first group, to be called *popular* trie collections, consists of the trie collections containing the most frequently occurring terms. In this group, a few common terms dominate the entries in each corresponding trie collection (by Zipf’s law [83]). In this case, the B-tree nodes on the path from the root to these common terms will be accessed frequently and hence it makes sense to store such paths in cache, which would indeed happen if we process such collections on the CPU.

The second group consists of the remaining collections. Unlike the popular trie collections, this group contains primarily infrequent terms and, again according to Zipf’s law, the differences in their frequencies are relatively very small. This means that every time we perform a B-tree operation for a new term, the path taken is likely to be very different than the previous one, and hence caching won’t be so useful. However exploiting data parallelism in processing each node (to perform all the comparisons in parallel) speeds up the computation significantly, and this is exactly

what we do using CUDA thread blocks.

Therefore, we assign the popular trie collections to a number of CPU indexers and unpopular ones to the GPU indexers. To determine which collections belong to which group, we extract a sample from the document collection, e.g. 1MB out of every 1GB, and run several tests on the sample to determine membership. Since there are many trie collections in this group and we have multiple GPUs, say  $N$  ( $N=2$  on our platform), we use a simple method of splitting the unpopular trie collections among the  $N_2$  GPUs by assigning the trie collection  $TC_i$  with index  $i$  to the GPU whose index is given by  $i \bmod N_2$ . For example, if unpopular trie collections have indices (0, 13, 27, 175, 384, 5810, 10041, 17316) and there are two GPU, then (0, 384, 5810, 17316) are assigned to GPU indexer 0 and (13, 27, 175, 10041) to GPU indexer 1. The overall data flow of the indexing stage is shown in Figure 5.6.

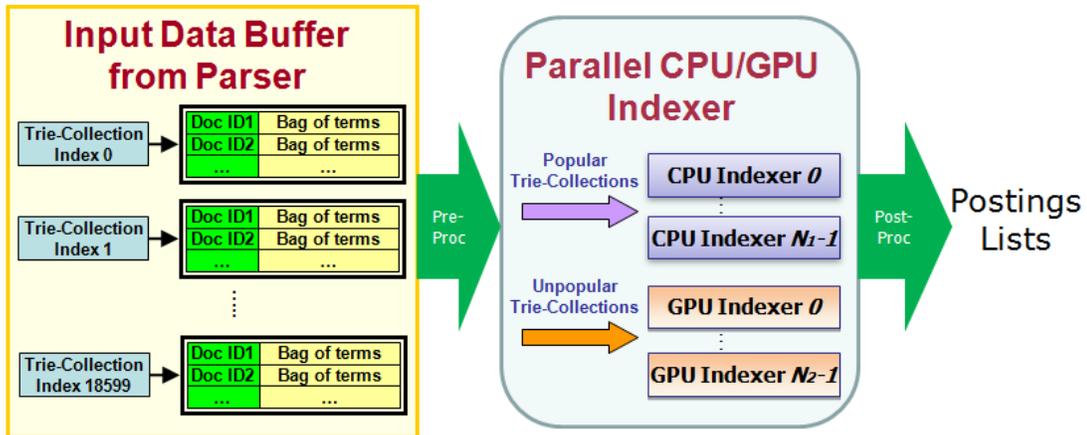


Figure 5.6: Data Flow of One Single Run on Parallel CPU and GPU Indexers

However there are relatively very few popular trie collections (around one hundred), and hence we split these trie collections into  $N_1$  independent sets such that each contains almost the same number of tokens, where  $N_1$  is the number of CPU threads used.

## 5.2.2 Experimental Results

Our new heterogeneous algorithm is tested on a single machine that holds two Intel Xeon X5560 quad-core CPUs and two NVIDIA Tesla C1060 GPUs each with a 4GB device memory. The first English segment of the ClueWeb09 collection is used and the details can be found in Table 2.3. In what follows, we first examine possible acceleration from GPUs and later compare the performance on five different GPUs.

### 5.2.2.1 GPU Accelerated Indexing Throughput

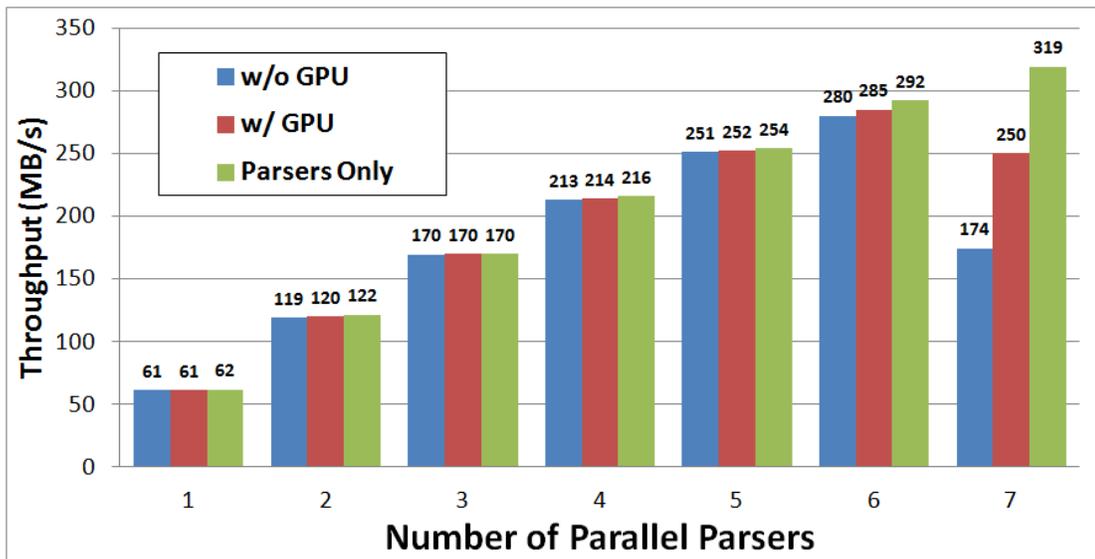


Figure 5.7: Optimal Number of Parallel Parsers and Indexers

We first focus on determining the best throughput generated by using GPUs. Figure 5.7 illustrates the performance of our algorithm on the ClueWeb09 dataset as a function of the number of parsers under three scenarios: (1)  $M$  parsers and  $8-M$  CPU indexers without GPU indexers; (2)  $M$  parsers and  $8-M$  CPU indexers with 2 GPU indexers; and (3)  $M$  parsers without any CPU or GPU indexers. The value of  $M$  varies from 1 to 7 since there are only eight physical cores available.

From Figure 5.7 we observe that the best overall performance is achieved by using six parsers, and we now proceed to examine the scalability of CPU and GPU indexers in combination with the six parser threads. We test the performance of our algorithm using three configurations: (i) no CPU indexer and two GPUs; (ii) two CPU indexers and no GPUs; and (iii) two CPU indexers and two GPUs. However we start by discussing the best possible strategy for indexing on a single GPU (Tesla C1060). As mentioned earlier we use thread blocks each with 32 threads to match the number of keys in each B-tree node and load each such node into shared memory using the 32 threads to achieve coalesced access. After extensive testing using a wide range of values for the number of thread blocks, it turns out that the best performance is achieved by using 480 thread blocks per GPU. From now on, whenever we refer to a GPU indexer we mean 480 thread blocks are running on a single GPU, each with 32 threads.

Table 5.3: Performance Comparison on Different Indexers with Six CPU Parsers

<b>Type of Indexers</b>	<b><i>GPU Only</i></b>	<b><i>CPU Only</i></b>	<b><i>CPU+GPU</i></b>
<b>Sampling Time (second)</b>	0	67	60
<b>Parallel Parsers (second)</b>	4990	5101	4998
<b>Parallel Indexers (second)</b>	14185	4017	2884
<b>Indexing Throughput (MB/s)</b>	103	362	505
<b>Dictionary Combine (second)</b>	3	3	3
<b>Dictionary Write (second)</b>	18	18	19
<b>Overall Time (second)</b>	14227	5195	5103
<b>Overall Throughput (MB/s)</b>	102	280	285

Detailed running times of various steps are shown in Table 5.3. The major part of the overall running time is  $\text{MAX}\{\text{parsing time, indexing time}\}$  plus some overhead like sampling time at beginning and dictionary handling time at the end. The indexing

time includes the pre-processing time, post-processing time and the time to transfer input and output data between main memory and GPU memories.

We now focus on the indexing throughput using the four configurations mentioned above. First, we notice an extra 39.5% performance gain is achieved through the acceleration from the two GPU indexers. Second, we notice that the throughput from CPU+GPU combination in scenario (iii) is higher than the throughput of CPU in scenario (ii) plus the throughput of GPU in scenario (i) separately. This super-linear scalability comes from our specific task partitioning strategy between CPU and GPU so both processors are doing computation that they are good at. Note that the performance of multiple GPU indexers is limited by the time it takes to transfer the parsed input to the GPU device memory and the time it takes to move the output postings lists back to main memory at the end of each single run. Another issue is the possibility of load imbalance among the CUDA threads, which is hard to fully control.

Now let's take a look at the contents processed by the CPU and the GPUs during indexing when the configuration consists of two CPU indexers and two GPU indexers. The GPU indexers process roughly the same number of tokens compared to those processed by the CPU, while the numbers of terms and characters are respectively 4.17 times and 3.52 times as those performed by the CPU. This shows that the effectiveness of the way we split the work load between CPU and GPU.

Table 5.4: Work Load between CPU and GPU

	<b>CPU Indexers</b>	<b>GPU Indexers</b>
Token Number	15,823,992,395	16,820,515,860
Term Number	16,408,505	68,390,970
Character Number	130,607,685	437,891,240

### 5.2.3.2 Shedding Light on the Performance of the GPU Indexers

The indexing problem by its nature consists of a large numbers of irregular memory accesses with relatively few complex operations. The 400-800 cycles due to the memory access latency are an integral part of the GPUs, and hence we can only resort to memory coalescing or caching to improve performance. To get deeper insights into the performance of the GPU indexers, we compare the throughput of five different GPUs on indexing 1GB segment of ClueWeb09 data. For a clean comparison, we set the throughput achieved on the Tesla C1060 to one and normalize the throughputs of the other GPUs relative to it.

Let's first look at the GT200 series GPUs, where the GTX260, the GTX280 and the Tesla C1060 are selected. Generally speaking, these three GPUs generate almost the same throughput as shown in Table 5.5. With the highest global memory bandwidth, the GTX280 achieves slightly higher performance. On the other hand, the GTX260 has many fewer CUDA cores but its performance is very close to the GTX280 or the Tesla C1060. The conclusion here is that the overall indexing throughput is not related to the number of CUDA cores. For the indexing algorithm, most of the memory accesses on the GT200 series GPUs are not coalesced and hence the memory bandwidth utilization rate is very low. It follows that higher memory bandwidth does not help a lot. The numbers of global memory reads and writes are obtained from the CUDA Profiler.

Table 5.5: Comparison between Five Different GPUs on Processing Same Data

<b>GPU Series</b>	<b>GT200</b>			<b>Fermi</b>	
<b>GPU Model</b>	GTX260	GTX280	Tesla C1060	Tesla C2050	GTX480
<b>SM Number</b>	27	30	30	14	15
<b>CUDA Core Number</b>	216	240	240	448	480
<b>Memory Bandwidth (GB/s)</b>	111.9	141.7	102	144	177.4
<b>Number of Global Memory Reads (million)</b>	284.21	290.4	290.97	27.18	27.31
<b>Number of Global Memory Writes (million)</b>	54.23	56.71	54.52	66.92	66.74
<b>L1 Miss Rate</b>	N/A	N/A	N/A	0.35	0.33
<b>L2 Miss Rate</b>	N/A	N/A	N/A	0.25	0.25
<b>Normalized Throughput</b>	1.02	1.03	1	1.3	1.64

When the newest Fermi series GPUs are considered, we observe a significant drop (over 90%) in the number of global memory reads as shown in Table 5.5 because of the incorporation of L1/L2 caches. However, the number of global memory writes increases by about 20% when compared to the GT200 series GPUs. The additional writes should come from the underlying caching policy because we observe that the number of write requests from our program remains the same (about 52 million) as in the cases of all three GT200 series GPUs. Caching has very limited effect on memory writes because most of writes come from updating the postings lists of terms where hardly any locality can be exploited.

We also see that the performance of the GTX480 is 26.5% faster than that of the Tesla C2050. To verify that such performance gain comes from the 40.7% higher memory bandwidth rather than one more Streaming Multiprocessor, we also include the performance of a single SM on all the GPUs in Table 5.6, where the throughput of

the Tesla C1060 is set to one and the remaining GPUs are normalized correspondingly. With a single SM, the GTX480 is still 22% faster than the Tesla C2050, for which the only reason should be the difference in memory bandwidth. So in this scenario with L1/L2 caches, higher memory bandwidth in fact results in significant improvement on the overall throughput. On the other hand, for the GT200 series GPUs the throughput with a single SM is not very sensitive to the memory bandwidth and all three GPUs, regardless of the number of SMs, achieve almost the same speedup when all available SMs are used.

Table 5.6: Normalized Throughput with Single Multiprocessor

<b>GPU Series</b>	<b>GPU Model</b>	<b>SM Number</b>	<b>CUDA Core Number</b>	<b>Memory Bandwidth (GB/s)</b>	<b>Normalized Throughput w/ Single SM</b>	<b>Speedup w/ All SMs Over Single SM</b>
<b>GT200</b>	GTX260	27	216	111.9	1.01	60.40
	GTX280	30	240	141.7	1.04	59.32
	Tesla C1060	30	240	102	1.00	59.84
<b>Fermi</b>	Tesla C2050	14	448	126 (ECC on)	1.62	47.85
	GTX480	15	480	177.4	1.98	49.58

It's hard to justify the performance difference between the GT200 and the Fermi GPUs because they also have different CUDA cores on each SM. However, using comparisons between the GT200 GPUs and between the Fermi GPUs, neither the compute power nor the number of CUDA cores is closely related to the overall throughput. Therefore we believe that the Fermi GPUs perform better than the GT200 GPUs because of small but very effective L1/L2 caches.

As a result, the major bottleneck of GPU indexers is in the overhead introduced

by global memory accesses. A small amount of cache will significantly help by decreasing the number of global memory accesses; moreover, high memory bandwidth is only useful when L1/L2 caches are present. However, the Tesla C1060 GPU has the largest global memory which is large enough to process the entire ClueWeb09 first English segment, and hence we only use this GPU for the other tests in previous sections.

### 5.2.3 Conclusion

It is clear that our heterogeneous indexing approach achieves a higher indexing throughput than just using a multicore CPU even though the process of building inverted files involve irregular computations. A number of factors contribute to the superior performance of our algorithm including:

- *The allocation of unpopular trie collections to the GPU where cache sensitive computations remain on the CPU; and*
- *The careful organization of memory accesses on the GPU in such a way as to exploit coalesced memory accesses and shared memory.*

The architecture features which are critical to the throughput are well studied by comparing the detailed performance with five different GPUs. The small caches in Fermi GPUs are proved to be very effective in reducing global memory accesses and higher global memory bandwidth is only helpful when such caches exist in the indexing problem. However, the speedup in the indexing stage doesn't contribute significantly to the overall throughput since the bottleneck is the parsing stage and as a result GPUs are not included in the cluster indexing and temporal indexing algorithms in Chapter 3 and Chapter 4.

## 5.3 Optimized List Ranking Algorithm on GPUs

### 5.3.1 Overview

It has been widely recognized that the *scan operation* on an array of elements plays a fundamental role in parallel processing [11, 12, 61]. This operation amounts to computing all the prefix sums of the elements stored in an array, and was recognized early on to be solvable by a *fast* and *work-optimal* data parallel algorithm. In this case, a fast parallel algorithm refers to  $O(\log n)$  parallel time assuming an unlimited number of processing elements. By work optimal, we refer to the fact that the data parallel algorithm asymptotically employs the same total number of operations as the best sequential algorithm. A fast and work-efficient implementation of the scan operation on the NVIDIA GPU was recently reported in [24]. The computation of prefix sums on the elements contained in a linked list also plays an important role in parallel processing of irregular applications involving linked structures such as trees and graphs [33] and postings list decompression during the construction of temporal inverted files or query processing [3, 18]. Given that the successor of each node of a linked list can appear anywhere in the memory, this computation can be dominated by fine-grain irregular memory accesses and as such it represents a challenging problem for parallel computing. It is worth noting that the *list ranking* problem is a special case in which all the values stored in the linked list are equal to the identity element. In such a case, the prefix sum of a node (called its *rank*) is equal to the distance of the node from the head of the list. In this section, we will tackle the general prefix sums problem but sometimes refer to it as list ranking.

### 5.3.2 Problem Definition and Related Work

Let  $L$  be a singly linked list in which each node contains two fields, one containing a data element and the second containing a pointer to the successor. The last node on the list is distinguished by a negative value in the successor field. The computation of prefix sums on  $L$  amounts to updating the value in the data field of a node by the sum of the values stored in the data fields of all the predecessor nodes, including the node itself. In other words, it is identical to the scan operation when it is carried out on an array  $A$  such that  $A[i]$  holds the value of the node of rank  $i$ . We assume that our list  $L$  is represented by an array  $X$  such that  $X[i].prefix$  and  $X[i].succ$  represent respectively the data and successor fields. Note that  $X[i].succ$  could be any arbitrary index in the array  $X$ . In our case, we assume that the head of  $L$  is not known, and hence has to be identified by the algorithm before prefix computations can take place. The main reason for making this assumption is to explore the impact of the presence of significant caches (as in CPUs) since the initial step that determines the head of the list will fill the cache with some of the input data thereby rendering the execution of later steps faster on such processors.

The list ranking problem admits of a simple and effective sequential algorithm involving two passes through the array  $X$ . The first pass identifies the head of the list, and the second pass traverses the list, starting from the head, and following the successor field while accumulating the prefix sums in the traversal order. This sequential algorithm performs extremely well in practice, especially if the processor contains a large enough cache, and hence it is somewhat of a challenge to develop a work-optimal, data parallel version that scales linearly.

The development of parallel algorithms for prefix sum has received significant attention in the literature dating back to the work of Wyllie [80] in which he introduced the pointer jumping technique for the PRAM model. More recently, the emergence of many internet applications that involve extremely large amounts of data with linked structures has rekindled interest in list ranking. Recent work on parallel algorithms for list ranking is reported in [6, 57], which will later be compared to the approach developed in this section.

We now quickly review the best known CUDA implementation of the scan operation on an array  $X$  in [24], a considerably simpler problem. The array is partitioned into subarrays, each group of whose prefix sums is computed separately by a block of threads using a data parallel algorithm that is a slight variation of Blelloch's [11]. The prefix sums of the last element of each subarray are then written into another array  $Y$ , followed by applying the same scan procedure on  $Y$ . Each prefix sum of  $Y$  is then added to each element of the corresponding block of  $X$ . This strategy can be viewed as a divide-and-conquer strategy in which the conquer steps involve the use of data parallel algorithms. As is, this strategy will not be appropriate for the prefix sum problem. In particular, each subarray resulting from the initial partition may contain many sublist fragments of the initial list, whose head nodes have to be identified before any useful work can be done. Such a process seems to require almost as much work as the initial prefix sum problem.

### 5.3.3 CUDA Implementation of Prefix Sum Computation

Our approach for prefix sum is also based on a divide-and-conquer strategy but the initial partitioning is randomized. We *randomly* select  $s$  nodes (called *splitters*)

from  $L$  and use these nodes to decompose the list into *sublist fragments* (or just sublists), each of which begins with a random node and consists of the successor nodes until we reach a node whose successor is a splitter. As we will see later, the choice of  $s$  is critical for optimal performance; the larger  $s$  the better the load balance among the streaming processors but the more the overhead will be incurred in combining the partial results. We divide these fragments equally between CUDA grid blocks, and use highly data parallel algorithms to process the fragments within each block. This is followed by a prefix sum algorithm applied on the list  $L'$  consisting of these  $s$  random nodes, where the successor of a random node  $Z$  is the first random node encountered upon the traversal of  $L$  starting from  $Z$ . The last merge step is similar to that of the scan operation and involves adding each prefix sum of  $L'$  to each element of the corresponding sublist fragment.

### 5.3.3.1 Detailed Description of Prefix Sums Algorithm

The prefix sums problem is formally defined as follows. We are given a singly linked list  $L$  of  $n$  elements stored in an array  $X$  such that  $X[i]$  contains two fields,  $X[i].prefix$  holding a data value and  $X[i].succ$  holding the array index of its successor. The successor of the last element of  $L$  contains a negative integer indicating the end of the list. We make the assumption that the index of the head of the list is not known. The prefix sum of the  $i^{\text{th}}$  node (assuming not the head of the list) is defined by:

$$X[i].prefix = X[i].prefix \otimes X[pre].prefix,$$

where  $pre$  is the index of the predecessor of  $X[i]$  and  $\otimes$  can be any binary associative operator. For the head of the list, the corresponding prefix sum is equal to the value of the data stored there. Figure 5.8 illustrates a simple example of a list and

the corresponding prefix sums.

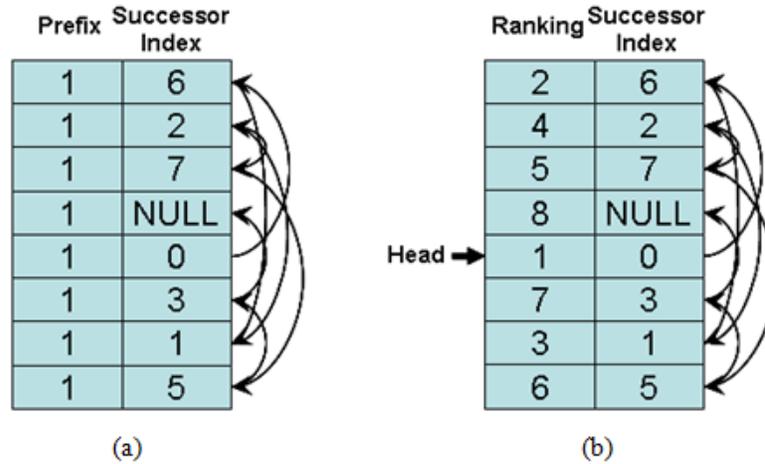


Figure 5.8: Input array and the corresponding output for list ranking

Our algorithm will follow the strategy outlined in the previous section using in particular many of the details developed in [30]. More specifically, our algorithm consists of the following five steps:

- **Step 1:** Compute the location of the head of the list.
- **Step 2:** Select  $s$  random locations of  $X$  to split the list into  $s$  random sublists, where each random location provides a pointer to the head of a sublist. As we will see, the value of  $s$  will be selected to guarantee load balancing with high probability.
- **Step 3:** Using the standard sequential algorithm, compute the prefix sums of each sublist separately. All the sublists can be handled in parallel.
- **Step 4:** Compute the prefix sums of the list consisting exclusively of the splitters, where the successor of a splitter is the next splitter encountered when traversing the initial list.
- **Step 5:** Update the prefix value of each element of the array  $X$  by using the

*prefix sum values computed in Step 4.*

### 5.3.3.2 Implementation Details and Analysis

The list ranking algorithm in [30] was designed for symmetric multiprocessors with caches. In our heterogeneous CPU and GPU platform, we partition the work so that few specialized tasks that can substantially benefit from caches are allocated to the CPU. The remaining tasks are handled by thousands of threads on the GPU in such a way as to hide memory access latency and make effective use of the shared memory. The implementation details of each step are described next.

**Step 1** can be implemented as follows. Since each of the indices between 0 and  $n-1$ , except for the index of the head node, must occur exactly once in the successor fields, the index of the head node can be computed by the formula:

$$HEAD = \frac{n(n-1)}{2} - SUM\_SUCC$$

where  $SUM\_SUCC$  is the sum of the all indices in the successor fields, except for the negative index. Hence this step amounts to a sum operation, which can be performed by a data parallel CUDA algorithm built around a balanced binary tree [25]. The resulting algorithm is work optimal and makes effective use of global and shared memory accesses. The details can be found in [25].

**Step 2** is implemented as follows. For every subarray of  $X$  of size  $n/s$ , we select a random location as a splitter, and record it in an array  $Sublist\_head$ . Then the successor field of splitter  $i$  in the original list is first copied into array  $Sublist\_scratch[i]$  and then changed into  $-i$  in  $X$  to indicate the fact that it is now the head of sublist  $i$ . Hence Step 2 involves highly data parallel computations that are

independent of each other.

**Step 3**, the most computationally demanding step of our algorithm, is implemented as follows. The  $s$  sublists are allocated equally among the CUDA blocks, which in turn are allocated equally among the threads of each block. Each thread will then compute the prefix sums of each of its sublists and copy the prefix value of the last element of sublist  $i$  into  $Sublist\_prefix[i]$ . Note that the head location of sublist  $i$  can be found in  $Sublist\_head[i]$ , and its successor index in  $Sublist\_scratch[i]$ . The end of a sublist can be identified by the negative successor index, indicating the head of another sublist. Also, once a node is traversed, its successor is changed into  $-j$  if it belongs to sublist  $j$ .

Using Lemma 1 of [30], the total number of nodes handled by a thread is about the same as any other thread with high probability if the number of sublists is at least  $p \ln n$  and the number of processors  $p$  is  $\sqrt{\frac{n}{\ln n}}$ , where  $n$  is the total number of nodes in the original list. We will later determine the best combination of the values of the number of blocks, the number of threads per block, and the number of sublists per thread, which will clearly satisfy these conditions.

**Step 4** uses the standard sequential algorithm to carry out the prefix computations on the list of splitters. In our case, this step is executed on the CPU since the amount of data involved is quite small. As we will see, the amount of time taken by this step (including the time to move the list of splitters from the GPU global memory to the main memory and back into the GPU global memory) is less than 3% of the total execution time of the algorithm.

**Step 5** updates the values of the prefix sums computed in Step 3 using the

splitters prefix sums of Step 4. Hence we have to identify the sublist of  $X[i]$  for each  $i$ . This can easily be done by checking the value of  $X[i].successor$ , which was set to the negative index of the corresponding sublist. Therefore, all the elements of  $X$  can be updated as follows:

*for all  $i$  between 1 and  $n$  do in parallel*

$$X[i].prefix = X[i].prefix \otimes Sublist\_prefix[-X[i].successor]$$

We perform this operation using coalesced memory accesses to the array  $X$ . We store  $Sublist\_prefix$  into the constant memory of the GPUs and then each SM can use its constant memory cache to load  $Sublist\_prefix$  instead of always fetching it from global memory.

### 5.3.3.3 Comparison with Recent Parallel Prefix Sum Algorithms

While both [6] and [57] use the algorithm of [30] as the basis of their prefix sum algorithms for CUDA and the Cell Processor respectively, there are a significant number of differences between their implementations and ours. We focus here on the main differences with [57] and ours since they use the same CUDA programming model. The main differences are:

- *The algorithm in [57] uses  $n/\log n$  or  $n/(2\log^2 n)$  as the number of splitters, which tends to generate too many sublists for very large  $n$ . As a result, handling the list of splitters and combining the results incur a very significant overhead. In our implementation we strike an optimal balance between the desirability of a large number of sublists (for fine-grain data parallel computations and load balancing) and the splitting/merging costs.*
- *The algorithm of [57] recursively computes the prefix sum of  $Sublist\_prefix$ ,*

*which will incur a very significant overhead. We simply perform this step using a sequential algorithm on the CPU. In our case, the execution time of this step is at most 3% of the total time.*

- *It is not clear how the splitters in [57] are selected. Unless they are selected carefully, the longest sublist can be very long in which case no load balancing will be possible. In our algorithm, we select the splitters in such a way as to guarantee load balancing with high probability.*
- *Another difference is the fact that [57] assume that they already know the head of the list, which makes their problem slightly easier than ours.*

### 5.3.4 Experimental Results

Our prefix sums algorithm is primarily tested on the NVIDIA Tesla C1060 graphic card as a co-processor to two Intel Xeon X5560 quad-core processors, because it has the largest global memory thereby accommodating lists with up to 256M nodes. In Section 5.3.4.5, we illustrate similar (improved) performance results achieved on the NVIDIA Tesla C2050 and the GTX480 graphic cards. Each of the input arrays consists of 64-bit entries such that 32 bits are reserved for the data field and the remaining 32 bits are reserved for the successor field. We select addition as our associative  $\otimes$  operation. As in [30], we consider three main types of input: (i) random, in which the successor is selected randomly from among the indices of the array; (ii) ordered, in which the successor node is the next consecutive node in the array; and (iii) stride of size  $d$ , in which the successor of each node is  $d$  indices away from the node, with wrapping around as necessary, and where  $d$  is some integer constant. We typically run each test hundreds of times, and report the average times

over all these runs.

#### 5.3.4.1 Performance as a Function of Various Parameters

We conduct extensive tests to determine the performance of our algorithm as a function of: (i) type of list (random, ordered, stride); (ii) number of random splitters (i.e. number of sublists). In the rest of this section, we give a summary of the results of these tests.

##### *1) Performance as a Function of the types of lists*

Typical execution times of our algorithm on ordered, random, and stride (with different stride values) lists with 64M ( $1M = 2^{20}$ ) nodes on the Tesla C1060 are shown in Figure 5.9. Tests run using different list sizes show a very similar pattern. Parameters (number of sublists, number of blocks, number of threads per block, and number of sublists per thread) are selected to yield the best performance for each type. The values of these parameters turn out to be the same for all list types. The results show a slightly faster performance on ordered lists, and show performance on stride lists which ranges from that of ordered lists to that of random lists depending on the size of the stride. However, overall the difference in execution times is at most 10%.

Given the overall memory architecture of the Tesla C1060, these results are not surprising. Note that since parallel accesses to contiguous locations can be coalesced, we see a slightly improved performance on ordered lists, or stride lists with small strides. However memory coalescing is limited once parallel threads process different sublists that are relatively far apart. An important observation is that when the stride value is large enough ( $>100$ ), the performance on stride lists is almost identical or

worse to that on random lists. Given this fact, we use stride lists with different stride values (typically stride value = 1,001) to report on the performance of our algorithm since it is much easier to generate such lists.



Figure 5.9: Performance on different types of lists with 64M nodes on the Tesla C1060

## 2) Performance as a Function of the number of sublists

We focus our attention here in determining the best value of  $s$ , the number of sublists. This issue is somewhat intertwined with the choices of other parameters such as number of blocks, number of threads per block, and number of sublists per thread.

Since the head of each sublist is randomly selected, the length of each sublist may vary over a large range. To achieve the best performance, we must try to balance the work load (number of nodes processed) among all the SMs and the SPs, which is in particular critical for the most computationally demanding Step 3 of our algorithm. In fact, the running time of Step 3 is dominated by the execution time of the SM that has to process the maximum number of nodes.

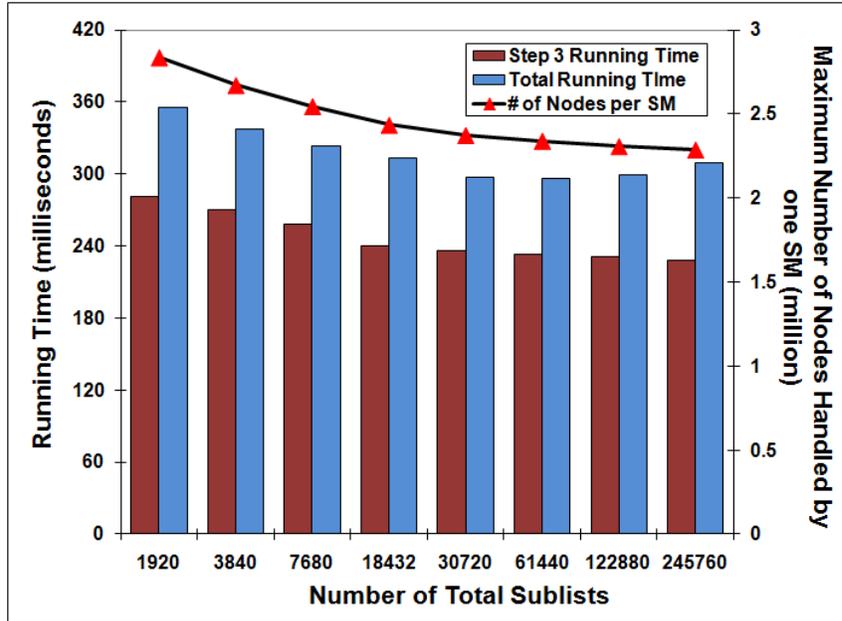


Figure 5.10: Performance of Step 3 and overall algorithm as a function of  $s$  on 64M stride list (stride = 1001) on the Tesla C1060. Upper curve represents the maximum load on an SM as a function of  $s$

For a fixed value of  $s$ , it is clear from the randomization performed in Step 2 that the more sublists are assigned to each thread, the more load balanced the work of the threads will be. This is intuitively clear and also follows from the proof of the main lemma in [30]. Hence this implies that a smaller number of threads will yield better performance. We can make the same argument regarding the number of blocks. For our Tesla C1060 processor, the number of blocks has to be a multiple of 30 (including 30 itself) and the number of threads per block has to be a multiple of 64. It turns that, for a fixed size list, the best performance is achieved when the number of blocks is equal to 30 and the number of threads per block is set equal to 64. Fixing these two parameters, we now take a close look at the execution time of Step 3 and overall execution time of the algorithm as a function of the number of sublists for a list of size 64M in Figure 5.10

As expected, the execution time of Step 3 decreases as the number  $s$  of sublists increases. However the overhead to create these lists (Step 2) and to combine the partial results (Step 4) will eliminate any gain beyond the value of  $s=61,440$ , resulting from the combination of 30 blocks, 64 threads per block, and 32 sublists per thread. It turns out that these values are consistent for all lists of sizes larger than 16M (up to 256M in our tests). Also, note that in the same graph we show that the maximum number of nodes handled by any SM, which decreases as we increase the number of splitters.

### *3) Performance as a Function of the number of blocks*

In this section, we report on the performance of our algorithm as we vary the number of blocks from 1 to 120. The NVIDIA scheduler allocates each block to an SM, and may allocate up to eight blocks to a single SM as necessary. Figure 5.11 shows that the performance of our algorithm is almost linear in the number of blocks whenever the number is less than or equal to 30. In our tests, we have used a stride list of 64M nodes with stride value 1001, 64 threads per block, and 61,440 sublists. This combination yields the best performance for any number of blocks. The performance stays about the same as we increase the number of blocks from 30 to 120, indicating that load balancing and memory accesses do not improve as we increase the number of blocks beyond 30.

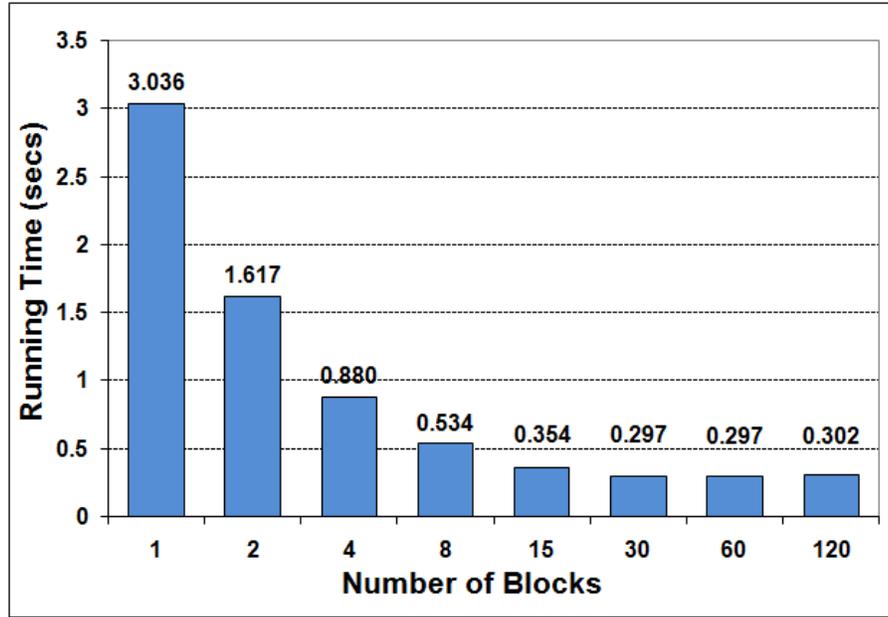


Figure 5.11: Performance of our algorithm on a 64M stride list (stride = 1001) as a function of the number of blocks on the Tesla C1060

#### 5.3.4.2 Migration of the computations in Step 4 to CPU

In Step 4, the sequential prefix sums algorithm is executed on the splitters, a process that can be carried out on the CPU or the GPU. If this task is migrated to the CPU, we have to pay the additional cost of moving the data from the GPU global memory to main memory and then copying the results back into the GPU to complete the execution of the algorithm. A reason to do this is that all the splitters can fit into CPU cache, which can speed up the computation. In our tests on 64M lists with 61,440 splitters, the GPU sequential algorithm takes 56.72 milliseconds to finish this task, which is about 16.57% of the total running time. On the other hand, this step only takes 2.016 milliseconds if migrated to the CPU: 0.335 milliseconds to copy these splitters from the GPU memory to main memory, 1.348 milliseconds to perform the computation, and 0.337 milliseconds to copy the results back to the GPU. This amounts to 0.70% of the total running time and the percentage is even lower for

larger lists.

### 5.3.4.3 Overall Performance of our Algorithm and its Scalability

We now report on the overall performance of our algorithm as a function of the list size using stride lists whose sizes range from 1M to 256M. Table 5.7 shows the detailed execution times of each of the steps for different list sizes. We have used the number of blocks to be 30 and the number of threads per block to be 64 based on the discussion above. The number of sublists per thread increases linearly with list size until it reaches 32, after which the running time spent on dividing and conquering increases faster than the time saved in fine tuning the loads among the threads. The overall running time scales linearly with the size of the list as illustrated by the graph in Figure 5.12.

Table 5.7: Detailed Running Times of Our Algorithm as a Function of the List Size

<b>List Size</b>	<b>Sublists per Thread</b>	<b>Step 1 (secs)</b>	<b>Step 2 (secs)</b>	<b>Step 3 (secs)</b>	<b>Step 4 (secs)</b>	<b>Step 5 (secs)</b>	<b>Total (secs)</b>
<b>1M</b>	30*64*2	0.000220	0.000342	0.003812	0.000108	0.000754	0.005511
<b>2M</b>	30*64*4	0.000381	0.000737	0.007046	0.000455	0.001680	0.010214
<b>4M</b>	30*64*8	0.000657	0.000641	0.014247	0.000462	0.003137	0.019945
<b>8M</b>	30*64*16	0.001244	0.001121	0.028098	0.000960	0.005967	0.037931
<b>16M</b>	30*64*32	0.002437	0.001908	0.056801	0.001697	0.011157	0.075175
<b>32M</b>	30*64*32	0.004850	0.002126	0.110164	0.001895	0.023580	0.146189
<b>64M</b>	30*64*32	0.011498	0.003519	0.221761	0.002016	0.048346	0.296590
<b>128M</b>	30*64*32	0.019204	0.002129	0.443050	0.001997	0.097861	0.574119
<b>256M</b>	30*64*32	0.038354	0.002654	0.930460	0.002003	0.199643	1.175801

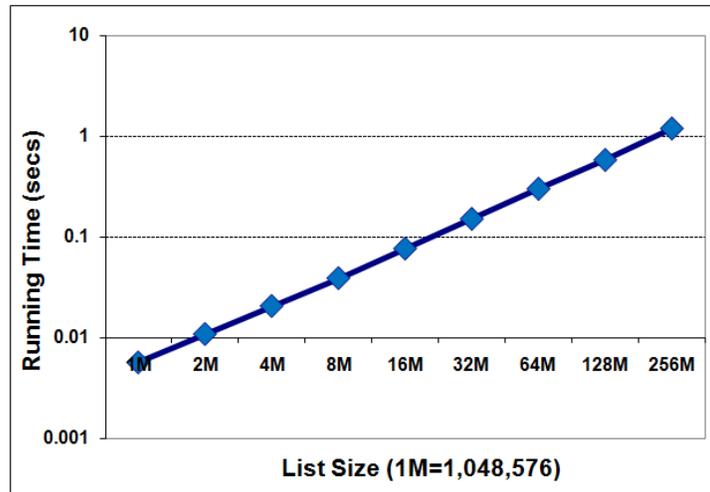


Figure 5.12: Scalability to list sizes on Stride lists (stride = 1001) on the Tesla C1060

#### 5.3.4.4 How much Can our Algorithm be Improved?

We now take a close look at the processing time per node achieved by our algorithm as a function of the list size ranging from 1M to 256M. Based on our extensive tests, the processing cost per node ranges between 4.28ns to 5.26ns, as long as the stride value is not extremely large. However, and surprisingly, the processing cost per node seems to increase significantly when the stride value is extremely large. To better understand this unexpected phenomenon and to determine whether our cost per node can theoretically be improved, we develop an optimized CUDA program whose main goal is to only access the device global memory at a very fine granularity, and compare its execution time per element to that of our prefix sums algorithm. It turns out that the performance of our algorithm is very close to the optimized global memory testing algorithm, both for extremely large strides and for moderate/small strides. We provide the details next.

Let  $Y$  be an array such that each entry consists of 64 bits. We develop a trivial CUDA algorithm such that each of its threads reads a location of  $Y$  (dependent on the

thread ID), and then successively reads  $N$  locations which are at a certain stride  $d$  from the initial location. Our goal is to develop such a CUDA algorithm that results in the best possible performance per single access. Let  $B$  be the number of blocks and  $Th$  be the number of threads of our program. We conduct tests based on the following 224 possible combinations of the values of  $B$ ,  $Th$ , and  $N$ :

$$B = \{ 8, 15, 16, 30, 32, 60, 64, 128 \};$$

$$Th = \{ 8, 16, 32, 64, 128, 256, 512 \};$$

$$N = \{ 1024, 2048, 4096, 8192 \}.$$

The ranges above seem to cover all reasonable values to achieve the best possible access cost per element. After exploring all the combinations, the values achieving the best access time per element turn out to be:  $B = 16$ ,  $Th = 64$  and  $N = 4096$ . Some other combinations came close but not as good as this combination. The typical time per access is 4.04ns for many of the tested sizes for the array  $Y$  (varied from 1M to 256M entries – that is, actual size of array  $Y$  varies from 8MB to 2GB).

We now compare the performance of the global memory testing algorithm with that of our prefix sums algorithm focusing on stride lists, starting with the case of stride value  $d=1001$ . A summary of the results is shown in Table 5.8, which is also illustrated in the graph shown in Figure 5.13. These results clearly demonstrate that the performance of our prefix sum program is very close to the performance achieved by the optimized global memory testing program.

Table 5.8: Time per element comparison between prefix sum and global memory testing program on the Tesla C1060 (Stride = 1001)

List Size (1M = 1,048,576)	1M	2M	4M	8M	16M	32M	64M	128M	256M
File Size (MB)	8	16	32	64	128	256	512	1024	2048
Test Program (per 64-bit integer) (ns)	3.70	3.84	3.93	3.97	4.00	4.04	4.06	4.13	4.19
Prefix Sum (per 64-bit node) (ns)	5.26	4.87	4.76	4.52	4.48	4.36	4.41	4.28	4.38

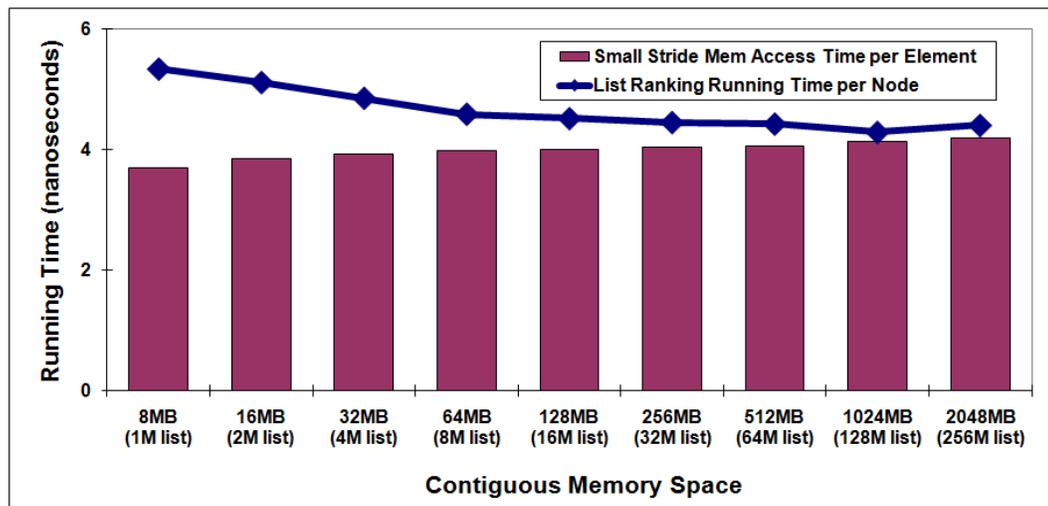


Figure 5.13: Time per element comparison between prefix sum and global memory testing program on the Tesla C1060 (Stride = 1001)

We now consider the case of extremely large stride values which are close to the size of lists. In this case, the performance degrades substantially when the stride value is 64M or larger (and hence list sizes are larger than 64M nodes) both by the global memory testing program as well as our prefix sums algorithm. However the performance per element still matches for both algorithms. Table 5.9 and Figure 5.14 illustrate the performance of both algorithms for extremely large stride values. However, this problem does not seem to show up on the Fermi series GPUs. On both Tesla C2050 and GTX480, we have run the largest possible stride lists with stride value larger than 64M as constrained by the global memory size and could not

observe this kind of degraded performance with large strides.

Table 5.9: Performance comparison of cost per element between prefix sum and global memory testing program using extremely large strides on the Tesla C1060

List Size (1M = 1,048,576)	1M	2M	4M	8M	16M	32M	64M	128M	256M
File Size (MB)	8	16	32	64	128	256	512	1024	2048
Test Program (per 64-bit integer) (ns)	3.74	3.86	3.95	3.99	4.03	4.06	4.07	8.70	13.40
Prefix Sum (per 64-bit node) (ns)	5.32	5.14	4.82	4.54	4.49	4.42	4.40	9.27	14.02

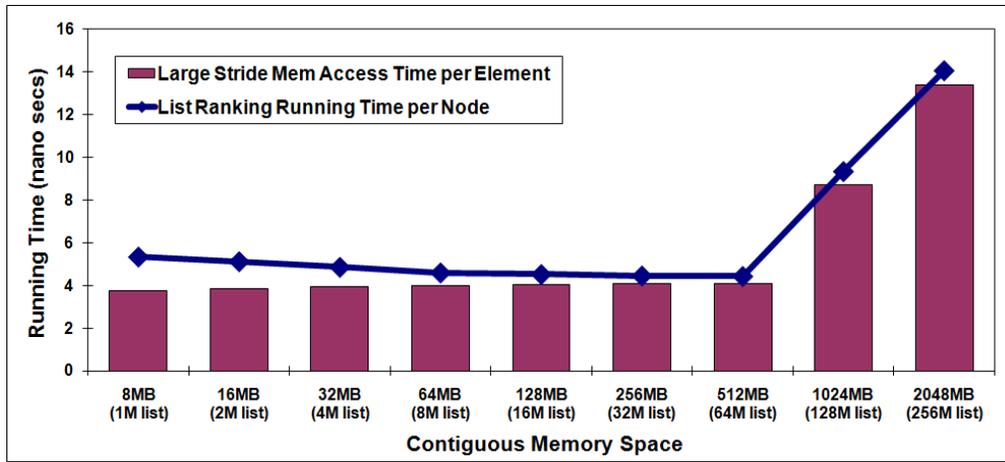


Figure 5.14: Time per element comparison between prefix sum and global memory testing program with extremely large strides on the Tesla C1060

### 5.3.4.5 Performance Comparison on the Newer GPUs

The new Fermi architecture introduces significant changes and adds a number of attractive features such as L1/L2 caches. In this section, we compare the performance of our prefix sum algorithm on three different NVIDIA GPUs, Tesla C1060, Tesla C2050 and GTX480 using the three types of lists with 64M nodes (largest possible list on our GTX 480). The hardware configurations of the above three GPUs are illustrated in Table 5.10.

Table 5.10: Configuration comparison of Tesla C1060, Tesla C2050 and GTX480

	<b>Tesla C1060</b>	<b>Tesla C2050 (with ECC)</b>	<b>GTX 480</b>
<b>Clock Speed (GHz)</b>	1.296	1.147	1.401
<b>Number of SMs</b>	15	14	15
<b>Total Number of SPs</b>	240	448	480
<b>Memory Size (GB)</b>	4	2.6	1.5
<b>Peak Memory Bandwidth (GB/s)</b>	102	126	177

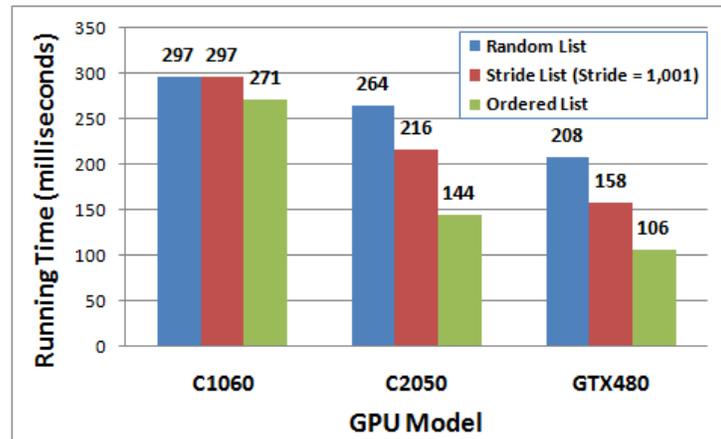


Figure 5.15: Performance comparison on Tesla C1060, Tesla C2050 and GTX480

Figure 5.15 shows the running times of our algorithm on ordered, stride and random lists with 64M nodes on the three GPUs. In each case, we optimized the values of the parameters such as number of blocks, number of threads per block, and number of sublists per thread so as to achieve the best possible performance. As mentioned in Section 5.3.4.3, the optimal parameters for Tesla C1060 on 64M list are 30 blocks, 64 threads per block and 32 sublists per thread. Due to the particular features on Fermi, we make the following changes that can be shown to yield the optimal performance: (1) change the number of blocks to 30 for the GTX480 and 28 for the Tesla C2050, which are equal to twice the number of SMs respectively because two warps can be scheduled concurrently onto SMs on Fermi; (2) increase the number of threads per block to 256 since the number of SPs per SM is

quadrupled; (3) reduce the number of sublists per thread to 4 in order to retain the same total number of sublists as suggested in Figure 5.10.

Table 5.11: Performance comparison of Three Different GPUs on 64M Random List

	<b>C1060</b>	<b>C2050</b>	<b>GTX480</b>
<b>Memory Bandwidth (GB/s)</b>	102	126	177
<b>Random List Throughput (GB/s)</b>	1.68	1.89	2.40
<b>Ratio of the above Two</b>	0.0165	0.0150	0.0136

Let’s first look at the performance comparison on 64M random list. In Figure 5.15, GTX480 performs best, followed by Tesla C2050 and then Tesla C1060. To find out the underlying reason, we compute in Table 5.11 the throughput by dividing list size by the running time to approximate its ratio to peak memory bandwidth. The ratios almost remain the same for three GPUs, a fact that shows that the performance improvement actually comes from the increase in global memory bandwidth. We note that this ratio number is not the global memory bandwidth utilization because we have several rounds of reading and writing over the input, output and temporal data in global memory. This is to be expected because the major bottleneck in prefix sum computation on GPUs is its highly irregular memory access pattern and hence more cores won’t contribute much. The resulting superior performance also illustrates that our algorithm can exploit any improvement in the global memory bandwidth.

The effect of caching available on the Fermi cards is significant for stride and ordered lists as shown in Figure 5.15. The L1 and L2 cache sizes are small on the Fermi GPUs, but they still have a large impact when processing regular data such as ordered lists. While the running time decreases slightly from random list to ordered list on the Tesla C1060, it reduces by almost 50% on either Tesla C2050 or GTX480.

We use NVIDIA Visual Profiler to monitor the number of global memory accesses occurred and the results indicate half the number of global memory accesses on ordered lists when compared to random lists on Fermi cards. On Tesla C1060, the number of global memory accesses remains almost the same.

### 5.3.4.6 Performance Comparison on Different Machines

In this section, we compare the performance of our prefix sums algorithm on the Tesla C1060 GPUs and the two Intel quad-core X5560 CPUs, as well as, to the performance of other recently reported prefix sum algorithms on the Cell Processor, the MTA-8 (eight 220MHz Cray MTA-2 processors) [6], and the NVIDIA GTX280 [57].

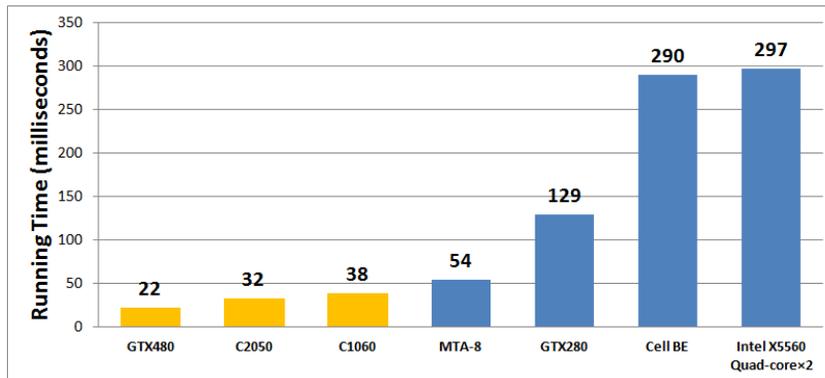


Figure 5.16: Comparison of different platforms on Random List with 8M nodes

Figure 5.16 provides a summary of the overall performance of our prefix sums algorithm on the Tesla C1060, Tesla C2050, GTX480 and two Intel X5560 quad-core CPUs, and the best performance numbers reported on other machines for the prefix sum problem on random list with 8M nodes. It is clear that our algorithm achieves the best raw performance even when compared to the Cray MTA-8 machine. The high performance of our algorithm can be attributed to the following facts:

- *Our implementation uses thousands of active threads to make effective use of the hundreds of processors available and to hide memory access latency;*
- *Cache sensitive computations are migrated to the CPU while the Cell has a small local cache and the MTA-8 has no local cache;*
- *The tradeoff between balanced workload for each processor and a small overhead of merging parallel results is well addressed and as a result the power of the underlying GPU hardware is fully utilized;*
- *Shared memory with one clock cycle access time among eight SPs from the same SM is used whenever applicable to reduce the inter-processor synchronization and communication overhead of parallelism; on the other hand, processors on either the Cell or the MTA-8 are interconnected by networks with relatively high latency;*
- *Global memory accesses are coalesced to exploit the high bandwidth of the GPU whenever possible.*

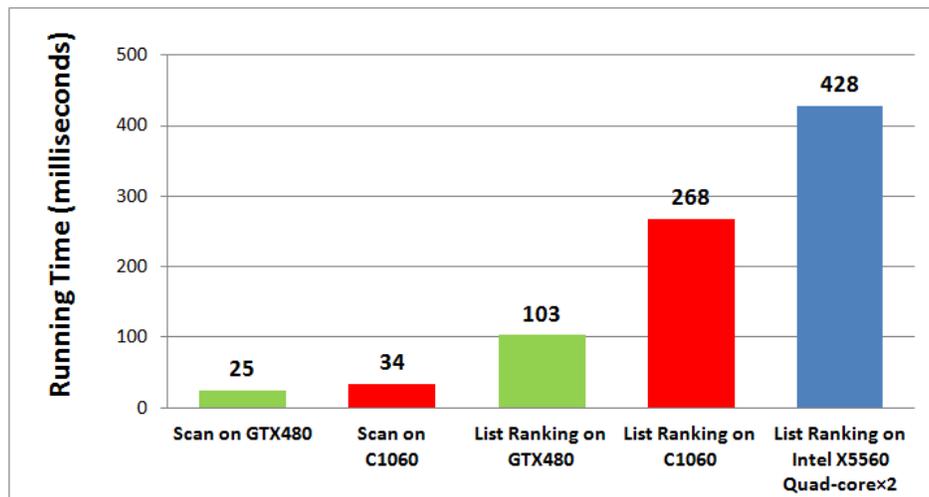


Figure 5.17: Performance of the scan operation and our prefix sum algorithm on 64M ordered list

We now compare the performance of our algorithm to that of the scan operation on the Tesla C1060 and GTX480, as well as to that of the CPUs on ordered lists. Clearly we expect the CPU performance to improve significantly on ordered lists because of the large L2 cache. On the other hand, the scan operation is specifically designed for sequential lists, and hence optimized coalescing of global memory accesses and optimized access to the shared memory (that is, avoiding bank conflicts) are reported in the CUDA implementation of [24]. A summary of the results is illustrated in Figure 5.17 for the case of 64M nodes. When both the scan algorithm and our algorithm are run on the Tesla C1060, the performance ratio 7-8 of our algorithm compared to the scan operation seems to hold independent of the list size. Such ratio reduces to 4 in the case of the GTX480 because of the improved performance on ordered lists due to L1/L2 caches, but the ratio still remains the same regardless of the list size.

### 5.3.5 Conclusion

In this section, we presented an optimized CUDA algorithm for performing prefix sums on linked lists. Such computation amounts to highly irregular, fine-grain global memory accesses, and hence is usually considered to be unsuitable for the stream based, data parallel CUDA programming model. The algorithm creates randomized sublists that are handled by parallel threads in such a way that the merging operation is organized to exploit the CUDA architecture. We have conducted extensive tests of our algorithm on the Tesla C1060, Tesla C2050 and GTX480 using different types of linked lists of sizes ranging from 1M nodes to 256M nodes. The results show scalable performance with the cost of processing a node close to the best possible for the Tesla

C1060 processor. Our algorithm also ports very well onto the newest Fermi GPUs, with significant improvement on stride and ordered lists due to the L1 and L2 caches available on the Fermi GPUs. A byproduct of our tests is the discovery of significant memory performance degradation whenever the consecutive locations accessed by a thread become far apart on the GT200 GPU. A recent work adopted our randomization strategy and applied a slightly modified algorithm of ours on a heterogeneous platform with both CPUs and GPUs and achieved better performance than just using GPUs [8].

## 5.4 Conclusion

In this chapter, we explored a number of techniques to accelerate highly irregular computations using GPUs. For the indexing problem, we develop an effective strategy to partition the tasks between the CPU and the GPU in such a way that each is focusing on the computations that are best suited for. For the list ranking problem, we partition the task into a large number of fine-grain computations through a randomization process and achieve scalable performance that is comparable to the scan operations on the same GPU. In both applications, several GPUs of different series are used to illustrate the effectiveness of small cache on newer Fermi GPUs.

## Chapter 6 Concluding Remarks and Future Perspectives

In this dissertation, we developed techniques and methodologies for exploiting the emerging parallel architectures of modern processors to solve large scale irregular applications including the construction of inverted files and computation of prefix sums on linked lists.

First, we introduced a novel pipelined strategy with a new dictionary data structure to build the inverted files on a single node of multicore processors. The dictionary data structure is built using a hybrid of trie and B-trees in such a way that parallel threads can read and write the dictionary simultaneously without conflicts. In addition, a high-throughput pipeline is implemented that avoids using disks as temporary storage for intermediate results and ensures that the generated parsed stream are immediately consumed at the same high rate by parallel indexers. Our resulting algorithm achieves a throughput of more than 280 MB/s on the ClueWeb09 dataset when tested on a node consisting of two Intel Xeon X5560 Quad-core CPUs and a centralized storage pool via a 1Gb/s Ethernet connection. Note that such performance is comparable to the currently best known algorithms on large clusters.

Second, we developed the approach to generalize the single-node inverted files construction algorithm to a cluster of multicore processors. The pipeline is carefully modified taking into consideration the communication overhead, where nodes communicate through an interconnection network, and load balancing between cores distributed across the cluster. We have generated extensive experimental results to illustrate the scalability of our strategy relative to the optimized single node algorithm. The experimental results show scalable performance using different

metrics on a cluster of 32 nodes with either 10 Gb/s InfiniBand or 1 Gb/s Ethernet. Overall, our throughput improves significantly on the previous best known algorithms on large clusters.

Third, we extended existing traditional information retrieval algorithms to develop new methodologies for searching collections of time stamped documents, such that the new inverted files include now the necessary extra temporal indexing structures. Our strategy ensures that the postings lists and the dictionary remain the same as in the non-temporal case but with the addition of relatively small temporal indexing structures. To guarantee a fast query response time, the lifetime of the collection is split into multiple time windows and an extra pipeline stage is inserted to take care of the additional work. Our strategy is well suited for incremental temporal updating of the document collection. Extensive experimental results indicate that the throughput in this more complicated case includes only a small overhead relative to treating the collection without considering the temporal dimension. Query performance is also verified to be very competitive to the non-temporal case, and can indeed be improved using finer temporal partitioning of the collection.

Last, we show how to make use of the latest NVIDIA GPUs to accelerate computations related to two highly irregular applications. For the indexing problem, a sampling strategy is employed to partition terms into popular and unpopular groups and deliver them to CPUs and GPUs separately such that the strengths of both architectures are taken advantage of. For the list ranking problem, an optimized multithreaded GPU algorithm is introduced through a randomization process that reduces the problem to a large number of fine grain computations in such a way that

the processing cost per element is shown to be close to the best possible. Our overall performance results are significantly superior to those reported in papers published prior to our work.

A number of additional research questions are worth pursuing.

In Chapters 2 and 3, we looked at the basic form of postings consisting of document ID and term frequency and briefly mentioned positional postings. A more practical implementation is to include other document information that is critical for ranking search results. For example, a term appears in the title should receive more weight. By analyzing URLs and building the connectivity graph between web pages, the additional information can be used to improve the relevance of the returned results, and the most famous technique to accomplish this is the PageRank first introduced by Google [54]. Such additional information will require new algorithms to maintain the high throughput of our pipelined strategy.

The construction of temporal inverted files was addressed in Chapter 4 and the experimental tests conducted on the hardware platform available to us. The major bottleneck of the newly added temporal merging stage is the disk I/O latency and bandwidth. However, with the widely used solid-state drives (SSDs) used as another layer in the memory hierarchy between main memory and traditional magnetic disks, the random access time and throughput can be both significantly improved. Should we have used SSDs on all nodes of the cluster, we believe that we would be able to access intermediate results much faster leading to a significantly reduced overhead due to the processing of temporal time windows. To better illustrate the strength of our approach, one might build the extra temporal indexing structures as well as the

postings lists using the MapReduce, and conduct experiments to determine the overhead with this totally different framework relative to treating the collection without considering the temporal dimension.

GPUs have shown their effectiveness in Chapter 5; however, their contribution to the overall throughput in real-world applications such as constructions of inverted files is not quite satisfactory. Newer GPU architecture designs and programming models are evolving fast to enable more flexibility. Meanwhile, parallel algorithms must be developed to make full use of hundreds or even thousands of cores inside one chip and of special memory resources such as the small shared memory and L1/L2 caches.

## Bibliography

- [1] S.V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial", *Computer*, Vol. 29(12), pp. 66-76, 1996.
- [2] O. Alonso, M. Gertz, and R. Baeza-Yates, "On the Value of Temporal Information in Information Retrieval", *ACM SIGIR Forum*, Vol. 41(2), 2007.
- [3] N. Ao, F. Zhang, D.S. Stones, G. Wang, X. Liu, J. Liu and S. Lin, "Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units", *Proceedings of the VLDB Endowment*, Vol. 4(8), pp. 470-481, 2011.
- [4] A. Anand, S. Bedathur, K. Berberich, and R. Schenkel, "Temporal Index Sharding for Space-Time Efficiency in Archive Search", *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, New York, USA, pp. 545-554, 2011.
- [5] A. Arampatzis and J. Kamps, "A Study of Query Length", *SIGIR*, pp. 811-812, July 2008.
- [6] D. A. Bader, V. Agarwal and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking", *IEEE International Parallel and Distributed Processing Symposium*, 2007.
- [7] A. Ball, "Web Archiving", Edinburgh, UK: Digital Curation Centre, 2010.
- [8] D.S. Banerjee and K. Kothapalli, "Hybrid Algorithms for List Ranking and Graph Connected Components", *Proceeding of 18th Annual International Conference on High Performance Computing (HiPC)*, Bangalore, India, 2011.

- [9] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum, "A time machine for text search", Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, Amsterdam, The Netherlands, 2007.
- [10] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum, "FluxCapacitor: Efficient TimeTravel Text Search", VLDB, Vienna, Austria., September, 2007.
- [11] G. E. Blelloch, "Prefix sums and their applications", Chapter 1 in Synthesis of Parallel Algorithms by J. H. Reif, Morgan Kaufmann Publishers Inc., San Mateo, California, 1993.
- [12] G. E. Blelloch, "Vector Models for Data-Parallel Computing", MIT Press, Cambridge, 1990.
- [13] S. Buttcher, C. Clarke, "Indexing Time vs. Query Time Tradeoffs in Dynamic Information Retrieval Systems", ACM Intl. Conf. on Information and Knowledge Management, pp. 317-318, 2005.
- [14] T. Chiueh and L. Huang, "Efficient Real-Time Index Updates in Text Retrieval Systems", Technical Report, Stony Brook University, New York, 1999.
- [15] Chossing Between OpenMP and Explicit Threading Methods,  
<http://software.intel.com/en-us/articles/choosing-between-openmp-and-explicit-threading-methods>. Accessed date: 05/29/2012.

- [16] L. Dagun and R. Menon, "OpenMP: an industry standard API for shared-memory programming", IEEE Computational Science & Engineering, Vol. 5(1), pp. 46-55, 1998.
- [17] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters", In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, Dec. 2004.
- [18] S. Ding, J. He, H. Yan, and T. Suel, "Using graphics processors for high-performance IR query processing", WWW '08, pp. 1213-1214, NY, USA, 2008.
- [19] 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.
- [20] J.F. Ferreira, J. Lobo, and J. Dias, "Bayesian real-time perception algorithms on GPU Real-time implementation of Bayesian models for multimodal perception using CUDA", Journal of Real-Time Image Processing, Volume 6, Number 3, pp. 171-186, 2011.
- [21] D. Geer, "Chip makers turn to multicore processors", Computer, Vol. 38(5), pp. 11-13, 2005.
- [22] A. Gulli, A. Signorini, "The Indexable Web is More than 11.5 Billion Pages", WWW 2005, May 10 –14, pp. 902-903, Chiba, Japan, 2005.
- [23] D. Harman, G. Candela, "Retrieving records from a gigabyte of text on a minicomputer using statistical ranking", Journal of the American Society for Information Science, vol. 41(8), pp. 581-589, Dec. 1990

- [24] M. Harris, "Parallel Prefix Sum (Scan) with CUDA", available at [http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/scan/doc/scan.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf). Access date: 05/29/2012.
- [25] M. Harris, "Optimizing Parallel Reduction in CUDA", available at [http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/reduction/doc/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf). Access date: 05/29/2012.
- [26] J. He, H. Yan, and T. Suel, "Compact Full-Text Indexing of Versioned Document Collections", ACM Intl. Conf. on Information and Knowledge Management, Hong Kong, China, November, 2009.
- [27] S. Heinz, J. Zobel, and H.E. Williams, "Burst Tries: A Fast, Efficient Data Structure for String Keys", ACM Transactions on Information Systems, Vol. 20, pp. 192-223, 2002.
- [28] 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.
- [29] Heritrix-Web-Crawler, <http://wa.archive.org/aroundtheworld/>. Accessed date: 05/29/2012.
- [30] D.R. Helman and J. JaJa, "Prefix Computations on symmetric multiprocessors", Journal of Parallel and Distributed Computing, Vol. 62(2), pp. 265-278, 2001.
- [31] B.A. Howel, "Proving Web History: How to Use the Internet Archive", Journal of Internet Law, pp. 3-9, Feb. 2006.

- [32] Internet-Archive, <http://www.archive.org/>. Access date: 05/29/2012.
- [33] J. JaJa, "An Introduction to Parallel Algorithms", Addison-Wesley, New York, 1992.
- [34] A. Khajeh-Saeed and J.B. Perot, "Computational Fluid Dynamics Simulations Using Many Graphics Processors", *Computing in Science & Engineering*, pp. 10-19, May/June, 2012.
- [35] E.A. Lee, "The Problem with Threads", *Computer*, Vol. 39(5), pp. 33-42, 2006.
- [36] N. Lester, J. Zobel, and H. E. Williams, "In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems", In *CRPIT '06: Proceedings of the 27th Conference on Australasian Computer Science*, pp. 15-23, Darlinghurst, Australia, 2004.
- [37] J. Lin and C. Dyer, "Data-Intensive Text Processing with MapReduce", Morgan & Claypool Publishers, 2010.
- [38] J. Lin, personal communication, April 2011.
- [39] 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.
- [40] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA TESLA: A Unified Graphics and Computing Architecture", *IEEE Micro*, Vol. 28(2), p. 39-55, 2008.

- [41] C.D. Manning, P.Raghavan, and H.Schütze, "An Introduction to Information Retrieval", Cambridge University Press, 2009.
- [42] 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.
- [43] R. McCreadie, C. McDonald, and I. Ounis, "MapReduce indexing strategies: Studying scalability and efficiency", Information Processing and Management, 2011.
- [44] L. Mearian, "The Internet Archive's Wayback Machine gets a new data center", Computer- world.com, Mar. 2009,  
[http://www.computerworld.com/s/article/9130499/The\\_Internet\\_Archive\\_s\\_Wayback\\_Machine\\_gets\\_a\\_new\\_data\\_center](http://www.computerworld.com/s/article/9130499/The_Internet_Archive_s_Wayback_Machine_gets_a_new_data_center). Accessed: Oct. 3, 2011.
- [45] 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.
- [46] Minerva, <http://lcweb2.loc.gov/diglib/lcwa/html/lcwa-home.html>. Accessed date: 05/29/2012.
- [47] 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.
- [48] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide Version 4.2", April 2012.

- [49] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi Version 1.1", 2009.
- [50] K. Nørnvåg, "Space-Efficient Support for Temporal Text Indexing in a Document Archive Context", Proceedings of the seventh European Conference on Digital Libraries, Trondheim, Norway: Springer Verlag, pp. 511-522, 2003.
- [51] K. Nørnvåg, "V2: a database approach to temporal document management," Proceedings of the seventh Database Engineering and Applications Symposium (IDEAS 2003), Hong Kong, China, pp. 212-221, 2003.
- [52] K. Nørnvåg, "Supporting temporal text-containment queries in temporal document databases", Data & Knowledge Engineering, vol. 49, pp. 105-125, 2004.
- [53] K. Nørnvåg and A.O. Nybø, "DyST: Dynamic and Scalable Temporal Text Indexing", Proceedings of the Thirteenth International Symposium on Temporal Representation and Reasoning, pp. 204-211, June, 2006.
- [54] L. Page, S. Brin, R. Motwani, and T. Wingograd, "The PageRank Citation Ranking: Bringing Order to the Web", Technical Report, Stanford InfoLab, 1999.
- [55] Pandora, <http://pandora.nla.gov.au>. Accessed date: 05/29/2012.
- [56] PetaBox, <http://www.archive.org/web/petabox.php>. Accessed date: 05/29/2012.

- [57] M. S. Rehman, K. Kothapalli and P. J. Narayanan, "Fast and Scalable List Ranking on the GPU", 23rd International Conference on Supercomputing (ICS), New York, USA, 2009.
- [58] 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.
- [59] S. Robertson, S. Walker, M. Hancock-Beaulieu, M. Gatford, and A. Payne, "Okapi at TREC-4", Proceedings of the Fourth Text REtrieval Conference (TREC-4), Gaithersburg, Maryland, pp. 73 –96, 1995.
- [60] Search-Engine-History, <http://www.searchenginehistory.com/>. Accessed date: 05/29/2012.
- [61] S. Sengupta, M. Harris, Y. Zhang and J. D. Owens, "Scan primitives for GPU computing", Proc. of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 97-106, 2007.
- [62] A. Shiraki, N. Takada, M. Niwa, Y. Ichihashi, T. Shimobaba, N. Masuda, and T. Ito, "Simplified electroholographic color reconstruction system using graphics processing unit and liquid crystal display projector", Optics Express, Vol. 17, Issue 18, pp. 16038-16045, 2009.
- [63] S. Song and J. JaJa, "Effective Strategies for Temporally Anchored Information Retrieval", UMIACS Technical Report, 2010.

- [64] A. Spink, D. Wolfram, B. Jansen, B.J. Jansen, and T. Saracevic, "Searching The Web: The Public and Their Queries", *Journal Of The American Society For Information Science And Technology*, Vol. 52(3), pp. 226:234, 2001.
- [65] H. Sutter and J. Larus, "Software and the Concurrency Revolution", *Queue-Multiprocessors*, Vol. 3(7), pp. 54-62, 2005.
- [66] Top500, <http://www.top500.org>. Accessed date: 05/29/2012.
- [67] M. Ujaldon, J. Saltz, "The GPU as an indirection engine for a fast information retrieval", *International Journal of Electronic Business*, Vol. 3. pp. 316-327, 2005.
- [68] UK Web Archiving Consortium, <http://www.webarchive.org.uk>. Accessed date: 05/29/2012.
- [69] U.S. Search Engine Rankings,  
[http://www.comscore.com/Press\\_Events/Press\\_Releases/2012/5/comScore\\_Releases\\_April\\_2012\\_U.S.\\_Search\\_Engine\\_Rankings](http://www.comscore.com/Press_Events/Press_Releases/2012/5/comScore_Releases_April_2012_U.S._Search_Engine_Rankings). Accessed date: 05/29/2012.
- [70] P.J. Varman and R.M. Verma, "An Efficient Multiversion Access Structure", *IEEE Transactions on Knowledge Data Engineering*, Vol. 9(3), pp. 391-409, May/June 1997.
- [71] WARC: Web ARChive file format,  
[http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=44717](http://www.iso.org/iso/catalogue_detail.htm?csnumber=44717). Accessed date: 05/29/2012.
- [72] Wayback, <http://archive-access.sourceforge.net/projects/wayback/>. Accessed date: 05/29/2012.

- [73] Web-at-Risk, <https://wiki.cdlib.org/WebAtRisk/tiki-index.php>. Accessed date: 05/29/2012.
- [74] Wikipedia, <http://en.wikipedia.org/wiki/Wiki>. Accessed date: 05/29/2012.
- [75] WWW-Size, <http://www.worldwidewebsite.com/>. Accessed date: 05/29/2012.
- [76] Z. Wei and J. JaJa, "Optimization of Linked List Prefix Computations on Multithreaded GPUs Using CUDA", International Parallel and Distributed Processing Symposium (IPDPS2010), Atlanta, GA, April 2010.
- [77] 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.
- [78] Z. Wei and J. JaJa, "A Fast Algorithm for Constructing Inverted Files on Heterogeneous Platforms", Journal of Parallel and Distributed Computing, Vol. 72(5), 2012.
- [79] Z. Wei and J. JaJa, "An Optimized High-Throughput Strategy for Constructing Inverted Files", to be published on IEEE Transactions on Parallel and Distributed Systems.
- [80] J. C. Wyllie, "The Complexity of Parallel Computations", PhD Thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.
- [81] Y. Zhang, F. Mueller, X. Cui, and T. Potok, "GPU-Accelerated Text Mining", EPHAM'09, Seattle, Washington, March, 2009.
- [82] Y. Zhao, X. Chen, C. Sham, W.M. Tam, and F.C.M. Lau, "Efficient Decoding of QC-LDPC Codes Using GPUs", Lecture Notes in Computer Science, 2011, Volume 7016, 294-305, 2011.

- [83] G. K. Zipf, "Human Behavior and the Principle of Least Effort : An Introduction to Human Ecology", Addison Wesley, Cambridge, Mass., 1949.
- [84] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines", ACM Computing Surveys, Vol. 38(2), July 2006.