Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A fast algorithm for constructing inverted files on heterogeneous platforms

Zheng Wei[*], Joseph JaJa

Department of Electrical and Computer Engineering, Institute for Advanced Computer Studies, University of Maryland, A. V. Williams Building, College Park, MD 20742, USA

ARTICLE INFO

Article history:
Received 28 September 2011
Received in revised form
6 February 2012
Accepted 8 February 2012
Available online 16 February 2012

Keywords:
Indexer
Inverted files
Multicore
GPU
Pipelined and parallel parsing and indexing

ABSTRACT

Given a collection of documents residing on a disk, we develop a new strategy for processing these documents and building the inverted files extremely quickly. Our approach is tailored for a heterogeneous platform consisting of multicore CPUs and highly multithreaded GPUs. Our algorithm is based on a number of novel techniques, including a high-throughput pipelined strategy, a hybrid trie and B-tree dictionary data structure, dynamic work allocation to CPU and GPU threads, and optimized CUDA indexer implementation. We have performed extensive tests of our algorithm on a single node (two Intel Xeon X5560 Quad-core CPUs) with two NVIDIA Tesla C1060 GPUs attached to it, and were able to achieve a throughput of more than 262 MB/s on the ClueWeb09 dataset. Similar results were obtained for widely different datasets. The throughput of our algorithm is superior to the best known algorithms reported in the literature even when compared to those run on large clusters.

1. Introduction

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 paper, 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 a framework leaves the details of scheduling, processor allocation, and communication to the underlying runtime 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 paper, we take the different approach that tries to exploit the common features present on current processors, both general CPUs and GPUs, to obtain a very fast algorithm for generating the inverted files.

The current trend in CPUs or GPUs increasingly includes more core processors on a single chip. It is expected that the number of cores will double every 18 to 24 months, and such a trend is likely to continue in the foreseeable future. While the architectures of current and emerging multiprocessors vary significantly, they all include several levels of memory hierarchy, Single Instruction Multiple Data (SIMD) or vector type operations, and multithreaded cores. These processors offer unprecedented opportunities for speeding up demanding computations on a single processor if the available resources can be effectively used.

In this paper, we consider a heterogeneous processor consisting of a CPU and two GPUs. The CPU consists of two Quad-core Intel Xeon X5560 with 24 GB of main memory, and each quad core shares an 8 MB L3 cache. Two NVIDIA Tesla C1060 GPUs are attached to our processor. The CPU 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 coupled with load balancing among the cores is critical to achieve good performance.

On the other hand, the basic architecture of our GPU consists of a set of Streaming Multiprocessors (SMs), each of which containing eight Streaming Processors (SPs or cores) executing in an SIMD fashion, 16,384 registers, and a 16 KB of shared memory. All the SMs have access to a very high bandwidth device memory; such a bandwidth is achieved only when simultaneous accesses are coalesced into contiguous 16-word lines. However, the latency to access the device memory is quite high, and is around 400–600 cycles. In our work, we have used the NVIDIA Tesla C1060 that has 30 SMs coupled to a 4 GB device memory with a peak bandwidth of 102 GB/s.

The Compute Unified Device Architecture (CUDA) programming model [1] of the NVIDIA GPUs 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,

[*] Correspondence to: 2464 A.V. Williams Building, University of Maryland, College Park, MD 20742, USA.

E-mail addresses: zwei@umiacs.umd.edu (Z. Wei), joseph@umiacs.umd.edu (J. JaJa).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.02.005

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. 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 16 threads in a half-warp. The eight cores 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 in the same SM. Optimizing the performance of multithreaded computations on CUDA requires careful consideration of global memory accesses (as few as possible, and they should be coalesced into multiples of contiguous 16-word lines), 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.

The rest of the paper is organized as follows. In the next section, we provide a brief background about the typical strategy used to build inverted files and a summary of the work that is most related to our paper. Section 3 provides a detailed description of our algorithm, including our new dictionary data structure, the organization of the parallel parsers and parallel indexers, and a high-performance CUDA implementation of the indexer. Section 4 provides a summary of our test results on three very different benchmarks, and we conclude in Section 5.

2. Background and previous 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 ⟨term, document ID⟩ 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, and 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 [3], 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 [10] 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 [5] 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. The 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 have appeared in the literature. In [9], 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 [11] is tailored for distributed systems. In the first run, the global vocabulary 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.

Nowadays, MapReduce-based algorithms are prevalent. First proposed in [2], the MapReduce paradigm provides a simplified programming model for distributed computing involving Internet-scale datasets on large clusters. The Map workers emit ⟨key, value⟩ pairs to Reduce workers defined by Master node, and the runtime would automatically group incoming ⟨key, value⟩ pairs received by a Reduce worker according to key field and pass ⟨key, list of values associated with this key⟩ to the Reduce function. A straightforward MapReduce algorithm for indexing is to use term as key and document ID as value, in which case the Reduce workers can directly receive unsorted postings lists. Since there is no mechanism for different Map workers to communicate with each other, creating a global dictionary is not possible. McCreadie et al. let Map worker emit ⟨term, partial postings list⟩ instead to reduce the number of emits and the resultant total transfer size between Map and Reduce since duplicate term fields are less frequently sent. Their strategy has achieved a good speedup relative to the number of processors and cores [8]. Around the same time, Lin et al. [7] developed a scalable MapReduce Indexing algorithm by switching ⟨term, posting{document ID, term frequency}⟩ to ⟨tuple{term, document ID}, term frequency⟩. By doing so, there is at most one value for each unique key, and moreover it is guaranteed by the MapReduce framework that postings arrive at the Reduce worker in order. As a result, a posting can be immediately appended to the postings list without any post processing. Their algorithm seems to achieve the best known throughput rate for full text indexing.

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