VIRTUAL MEMORY STREAMING AND SORTING IN MAPREDUCE APPLICATIONS

An Undergraduate Research Scholars Thesis

by

YUAN YAO

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Dmitri Loguinov

May 2018

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Virtual Memory Streaming in MapReduce Applications

Yuan Yao
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Dmitri Loguinov
Department of Computer Science and Engineering
Texas A&M University

In the age of fast growing technology, massive storage, and cluster computing, efficient big-data processing algorithms are in high demand. MapReduce is one of the programming models that enables massive-scale cluster technology around the world. Despite significant public efforts, the open-source implementation of MapReduce – Apache Hadoop – is cumbersome, complex, and inefficient. The purpose of this research is to improve the performance of Hadoop, specifically its sorting component, by developing a single-pass, stream-based multithreaded bucket sort. Our new set of algorithms has the potential to influence the future of data-centric computing.

ACKNOWLEDGEMENTS

I would like to thank my research advisor and mentor, Professor Dmitri Loguinov for his teaching and guidance. I would not be able to learn this much without his support.

Thank also goes to the department faculties and staffs for providing me with precious resources to study, learn and improve myself.

NOMENCLATURE

AWE	Address Windowing Extensions
API	Application Programming Interface
MSDN	Microsoft Developer Network
OS	Operating System
PG	Physical Pages
PFE	Page Fault Exception
RAM	Random Access Memory
VMS	Virtual Memory Space

CHAPTER I

INTRODUCTION

Background

The twenty-first century has been the golden age of computer technology. With the increasing usage of personal devices and the Internet, the knowledge base of the society is rapidly shifting from the traditional paper and books to hard drives, where large-scale server platforms, i.e. the cloud, are expected to provide all storage-related computation in the future. According to a report cited by Forbes, global spending on Infrastructure-as-a-Service (IaaS) was projected to grow 32.8% from 2014 to 2015 [1]. With the vast growth of cloud services, the need for fast scalable cluster computing is greater than ever. MapReduce is one of the most important frameworks for data analysis. It is a distributed and parallel programming model designed to quickly process large data sets [2]. One of its best-known open-source implementations is Apache Hadoop [3]. Despite its popularity, Hadoop suffers from performance bottlenecking in its sorting algorithms. Our goal is to improve the speed of this core function and provide novel research results that can benefit millions of servers by saving them significant amounts of computation time and hardware energy.

Although much effort has been made to enhance MapReduce applications, such as optimization of schedulers, improvement of robustness of failsafe systems, etc., most stayed high-level on current MapReduce frameworks. Little has been seen to enhance one of the most fundamental infrastructures, larger-than-memory file handling. In this paper, we start by introducing popular existing methodologies frequently used for streaming applications, and analyze their drawbacks to show why our proposed method prevails.

Related Work

Bucket Sort

Bucket Sort is known to be one of the fastest sorting algorithms that rivals Quick Sort in cases. Nevertheless, its great performance in speed comes at a cost of memory usage. One major drawback of Bucket Sort is that it's a out-of-place algorithm. With n being the size of an arbitrary input, an out-of-place algorithm uses double amount of memory as its output co-exists with the input data: $2n + c$, where c standards for additional constant memory usage not proportional to n. Whereas an in-place algorithm uses $n + c$ amount of memory. In large-scale MapReduce applications, memory is especially restricted as input sizes are generally enormous, which prevents out-of-place algorithms like classic Bucket Sort to be efficiently applied.

Given the disadvantages of Bucket Sort, our motivation is clear – introduce data streaming methods to reduce memory usage of Bucket Sort. The following two sections will list and discuss examples of currently used streaming solutions.

Shadow Buffer

When a program reads files of large volume that exceeds the total RAM supported in the system, a tool/technique called the *Shadow Buffer* is used to help segmented process of the files. In most cases, an algorithm processing files requires the continuity of the file to be maintained to ensure correctness of result. Shadow Buffers are very helpful under such circumstances. Given a large file segmented into n chunks to be each read into memory. In turn, there are $n - 1$ shadow buffers each allocated with r memory. Therefore, the memory overhead of such algorithm is: $Overhead = r \times (n - 1)$.

The implementation of Shadow Buffers is very complex and problem-specific. A Shadow Buffer developed for one algorithm may or may not be able to work on another algorithm. This is one of the worst drawbacks, not reusable.

Memory-Mapped File

Similar to Shadow Buffer, Memory-Mapped File (MMF) is a tool used to help process files of large volumes. Primarily, MMF maps virtual memory to an on-disk file directly, byte by byte. Its design helps programmers to treat files on disk as a piece of memory directly accessible. While portions of the file are processed, MMF prefetches the next portions so that they'll be ready whenever needed. MMF releases programmers from the burden of worrying about disk I/O.

Simplicity does not always pay off. Unlike the versatility of Shadow Buffer, when dealing with files larger than memory, MMF lacks memory recycling as it can hardly obtain information of if a processed portion will be needed in the future. Additionally, many implementations of MMF cannot provide optimal speed for the user.

CHAPTER II

METHODS

Dynamic Virtual Memory Allocation

Memory is a precious resource in computer programs, even with rapid improvement on memory capacity. What is often called memory generally refers to the physical memory of a device, representing the actual capacity of the device. For security reasons, physical memory addresses of a system are normally inaccessible from user-level programs, hiding sensitive information such as passwords, info of another program, system state, etc. In return, user-level programs are given “fake” address spaces called the virtual address space. The operating system then translates each virtual address internally to match actual physical memory. Each program has the same set of virtual memory space. Additionally, operating systems divide virtual memory into segments called pages so that mapping of virtual memory to physical memory is more manageable. This mapping of pages is referred to as committing memory; whereas un-mapping is called de-committing memory.

Dynamic virtual memory allocation utilizes the feature that virtual memory can create an illusion of “infinite” memory – virtual address can grow up to 24 TB (Windows Server 2016 64-bit). Most modern operating systems such as Windows and various Linux distributions support this feature automatically and implicitly. For example, when a program allocates a large set of virtual memory, a typical OS such as Windows does not immediately commit physical memory. The commitment of memory only happens when the memory is mapped Although physical memory is the ultimate limitation, by deallocating used virtual memory, physical memory can be freed and recycled. Such is the core concept behind dynamic virtual memory allocation.

Page Fault Exception Handling

Page fault exception (PFE) is a common exception supported by most operating systems.

Virtual Data Streaming

Despite the excellent performance and relative ease of programming compared to Shadow Buffer, the implementation of dynamic virtual memory allocation is still far too verbose and requires numerous micro management to allocated memory region. In fact, the user has to maintain the allocation process of the buffer for proper usage. To eliminate this inconveniency, we take advantage of the page fault exceptions introduced in the previous section.

AWE-based Data Streaming

Address Windowing Extensions (AWE) is a set of API's developed by Microsoft on Windows operating systems. AWE is originally used to extend the memory capabilities of a 32-bit software application by allowing the program to access physical memory greater than 4GB [4]. AWE introduces a process called "physical mapping". In this process, two separate blocks of memory are allocated, a block of continuous virtual memory space (VMS) and a block of physical pages (PG). VMS is the memory address that'll accessible in both reading and writing by the user. In the contrary, PG is an inaccessible memory block used by the kernel to store information regarding the actual content user stored in VMS. Each byte of PG represents a page p bytes in memory, which is typically 4096 KB. Therefore, to use n bytes of memory, $\frac{n/1024}{4096}$ bytes of PG have to be allocated. In order to access virtual address, PG blocks must be mapped to the designated location within VMS. VMS will not occupy any memory unless mapped to PG.

Although the original intention of AWE was to help 32-bit applications access more memory, we took advantage of the features and applied to our 64-bit application. Due to the memory consumption of BucketSort, applying AWE to our program creates an illusion to the

user – our BucketSort algorithm that memory space is continuous while avoiding the complex implementation of Shadow Buffers.

We developed two primary models for the data streaming, producer-consumer (PC) based model and single-buffer based. Both of them apply the Vectored Exception Handling (VEH) API's introduced by MSDN [5] as well as a customizable page fault trigger, meaning that instead of triggering a page fault on VMS in every p bytes of memory, the program can be configured to trigger a page fault in every $i \times p$ bytes, where i is a positive integer. We call this newly defined page a *block*. This means that the amount of time in throwing and handling page fault exceptions can be greatly reduced by a factor of i.

Producer-Consumer Dual-buffer Stream Model

The producer-consumer model utilizes two separate VMS to maximize the reading and writing efficiency of the data stream. One VMS has read-only access and the other one has write-only access. Such implementation is a great demonstration of the advantages brought by using AWE. With PG carrying the actual content of the buffer, it is irrelevant which VMS is operated on, so long as it is mapped with a PG block. Through the application of Vectored Exception Handling (VEH), an exception handling routine that functionalize exception handling code [5], the algorithm model was able to encapsulate the mapping action, creating an appearance that VMS is continuous. The producer-consumer model is enforced by the use of semaphores such that there are always a fixed number of PG blocks mapped to the VMS.

Producer-Consumer Single Buffer Stream Model

Similar to the producer-consumer model, this approach also uses page fault exceptions as triggers to handle the mapping and un-mapping process. However, the difference is the complexity and buffer count. As the name of this model states, the algorithm only uses one VMS

buffer. This method greatly reduces the complexity of mapping and un-mapping procedure. At the initialization stage, user defines the peak memory usage m B. In turn, $\frac{m}{p}$ number of physical pages allocated, all of which are then pushed and stored in a queue. Upon each read page fault exception, the least recently used PG block is unmapped from VMS and pushed back into the queue for next use, and vice versa for write page fault.

ADS-improved Bucket Sort

Based on previously developed methods and models, we introduce our first adaptive application-usage of the stream models.

CHAPTER III

EXPERIMENTS AND RESULTS

Test Environment

We conduct our experiments on three setups of computers with different specifications. As shown in Table 1 below:

Table 1: Specifications of Test Machines

	c1	c2	c3
i7 CPU	3930K	4930K	7820X
Platform	Sandy Bridge-E	Ivy Bridge-E	Skylake-X
Cores	6	6	8
Turbo clock	3.8 GHz	3.9GHz	4.7 GHz
RAM	32 GB	32 GB	32 GB
RAM type	DDR3-2400	DDR3-2400	DDR4-3200
Test disk	24-disk RAID	24-disk RAID	M.2 SSD
Primary OS	Server 2008 R2	Server 2008 R2	Server 2016

File I/O

In this experiment, we compare file read and write speeds of Virtual Data Stream Model, Producer-Consumer Dual-buffer Stream Model, Producer-Consumer Single Buffer Stream Model, Memory Mapped Files, and C++'s ifstream.

Producer-Consumer

Memory Usage

Given virtual memory of λ bytes, the block size of each mapping to be b bytes, the maximum number of blocks to be n, and a page size of p bytes. Our results have proven that the amount of virtual memory allocated does not greatly affect the run-time memory usage. Both

producer-consumer stream and single buffer stream exhibits similar theoretical peak RAM usage is:

$$peak\ memory = nb + \frac{nb}{p} \cdot sizeof(ULONG\_PTR) \qquad (1)$$

In formula (1), the trailing *ULONG_PTR* indicates the number of bytes used in a 64-bit application to represent one memory address, which is 8 bytes.

Performance

The performance of our models were measured on a single desktop with the specifications of 6-core 2.80 GHz AMD Phantom II X6 1055T processor with 16 GB of RAM. Stream model results are compared with the non-stream based, regularly allocated heap memory. The experiment was designed to set up a fixed number of page faults monitoring the defects of physical page mapping/unmapping API's.

Table 2: Stream Performance Benchmark Results

Model	Preset page fault count	memset (ms)	read by + 8 (ms)	combined (ms)	Total Executin time (ms)	MapUserPhysicalPages %	MapUserPhysicalPages time (ms)
Single buffer stream (ms)	1088	318	253	564	845	7.06%	59.657
Regular buffer	1088	307	232	539	787	0.00%	0

Note that MapUserPhysicalPages is the mapping/unmapping API used. The results from Table 2 proves our Stream model to have very good performance, as little as 7.06% execution time spent. This number may be further deducted when performing more complex memory manipulation algorithms.

CHAPTER IV

CONCLUSION

Based on our experimental results, AWE-based Data Streaming and Bucket Sort show great results in both performance and usability. Different models of ADS have much faster read and write performance than all pre-existing I/O solutions that we've tested, while maintaining a programmer-friendly interface – memory allocated by ADS is almost the same as a normal region of memory allocated by normal means. It is our hope that Operating System makers in the future would consider ADS as a feature built into the OS. In that case, the simplicity of programming and performance could be further improved.

With the success in virtual memory streaming, our ADS Improved Bucket Sort benefits with the opportunity to reduce memory usage of a Bucket Sort while preserving the performance of a classic Bucket Sort algorithm. However, MapReduce is generally run clusters of computers. In other words, sorting algorithm for MapReduce are generally multi-threaded to support large scale sorting. Therefore, despite the success in ADS Improved Bucket Sort, our future work will focus on parallelizing Bucket Sort.

REFERENCES

[1] L. Columbus, “Roundup of Cloud Computing Forecasts And Market Estimates Q3 Update”, http://www.forbes.com/sites/louiscolumbus/2015/09/27/roundup-of-cloud-computing-forecasts-and-market-estimates-q3-update-2015/#7a3a80246c7a, 2015.

[2] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, *USENIX OSDI*, October 2004.

[3] The Apache Software Foundation, “Hadoop”, http://hadoop.apache.org/.

[4] Microsoft Corporation, “Address Windowing Extensions”, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366528(v=vs.85).aspx.

[5] Microsoft Corporation, “Vectored Exception Handling”, https://msdn.microsoft.com/en-us/library/windows/desktop/ms681420(v=vs.85).aspx