Enhance Data De-Duplication Performance With Multi-Thread Chunking Algorithm

This paper is submitted in partial fulfillment of the requirements for Operating System Class (COEN 283)

Santa Clara University

Computer Engineering Department

December 9, 2014

Xinran Jiang, Jia Zhao, Jie Zheng

Abstract

In this work, we develop a multi-threaded Frequency-Based Chunking method (FBC), which exploits the multicore architecture of the modern microprocessors. The legacy single threaded Frequency-Based Chunking method makes much improvement in reducing the total chunk size and the metadata overhead, compared to the popular Content-Defined Chunking algorithms (BSW, TTTD, etc.). Yet it also leaves much to be explored in terms of faster and more efficient file de-duplication and backup. Our work is aimed at a significant increase in the chunking time performance, by applying concurrent frequency-based data chunking algorithm.

Keywords

Frequency-based Chunking; Content-defined Chunking; Bloom Filter; Multi-thread; Basic Sliding Window; Two Thresholds Two Divisors

Table of Contents

List of Figures

1. Introduction

Data de-duplication is a file system feature that only saves unique data segments to save space [1]. It has been most popular and successful for secondary storage systems (backup and archival). The most sophisticate systems using inline de-dupe, meaning the data is de-duplicated as it's been written to the file system. Simply speaking, when a new file arrives, the de-duplication system uses a chunking algorithm to break the entire file into many small blocks known as chunks. Then the system uses a hash algorithm to generate unique signatures for the chunks. After that, the system can accurately compare those signatures with the ones in its database to identify these chunks as new data or redundant data. If the system detects a matching signature, it creates a logical reference to the duplicate data since its identical copy is already stored in the database. Otherwise, the system treats these chunks as new data and stores them in its database.

The Basic Sliding Window (BSW) algorithm is the first prototype of a content-based chunking algorithm that can handle most types of data. The Two Thresholds Two Divisors (TTTD) algorithm was proposed to improve the BSW algorithm by controlling the chunk-size variations. Recently a modified version of TTTD algorithm, the TTTD algorithm with a new switch parameter, is brought up by Moh et al [3]. It shows a slight improvement on time performance, reducing 6% of the running time and a solid progress on chunk size variations, reducing 50% of the large-sized chunks. Our concern is how to further enhance the time performance of the TTTD algorithm.

Another improvement that draws our attention is Frequency Based Chunking (FBC) algorithm, by Lu and Du [2]. Unlike the most popular Content-Defined Chunking

(CDC) algorithm which divides the data stream randomly according to the content, FBC explicitly utilizes the chunk frequency information in the data stream to enhance the data deduplication gain especially when the metadata overhead is taken into consideration. The benefits of FBC include achieving a better dedup gain (metadata taken into account) and producing much less number of chunks. However, the time performance of the algorithm has been overlooked and we would like to improve on this aspect.

Generally speaking, the objective of our studies is to improve the time performance of two state-of-the-art chunking algorithms, the TTTD and FBC. We propose the use of multi-threading in a multi-core environment. We would implement both the single-stream and multi-thread versions of the chunking algorithms. By comparison of the time performance, we would conclude the improvement in efficiency of multi-thread chunking over single-stream, particularly for the Frequency Based Chunking algorithm.

Previous studies have shown that application of multi-threading can greatly improve the time performance of Content-Defined Chunking algorithms [4] [5] [6]. Won, Lim and Min implemented the multi-threaded BSW chunking algorithm in "MUCH: Multithreaded Content-Based File Chunking". They successfully addressed the performance issues of file chunking which is one of the performance bottlenecks in modern deduplication systems by parallelizing the content-defined file chunking operation while guaranteeing Chunking Invariability. We would like to extend the studies from Content-Defined Chunking to Frequency Based Chunking.

2.Theoretical bases and literature review

The vital phase of data de-duplication is "chunking", which divides a file into unique data chunks. The algorithm has to be stable for local modifications and is very CPU intensive. To gain better performance, our approach is using multi-thread to enhance the chunking algorithm.

In our project, we focus on improving the hybrid FBC(Frequency Based Chunking) algorithm's performance by using multi-thread computing. The chunking steps of hybrid FBC algorithm consists of a coarse-grained content defined chunking(CDC) and a second-level fine-grained frequency based chunking(FBC). During coarse-grained chunking, we chop the byte stream into large-size data chunks. During the fine-grained chunking, FBC examines each coarse-grained chunk to search for the frequent fix-size chunk candidates (based on the chunk frequency estimation results) and uses these frequent candidates to further break the CDC chunks.

In the CDC stage, we will use TTTD (Two thresholds Two Divisors) algorithm. The TTTD algorithm is hard to implement as multithread because of data dependency. In original TTTD algorithm, the next chunk's starting position depends on the previous chunk's ending position. In this paper we will only use single-thread for TTTD and focus on using multi-threading in the second stage of FBC. Using multi-threading in the first stage can be studied in the future.

Based on the primitive TTTD chunking, we will implement FBC to do further chunking in multi thread. Each FBC thread will have its own buffer, but share the same set of bloom filters and frequency counting table.

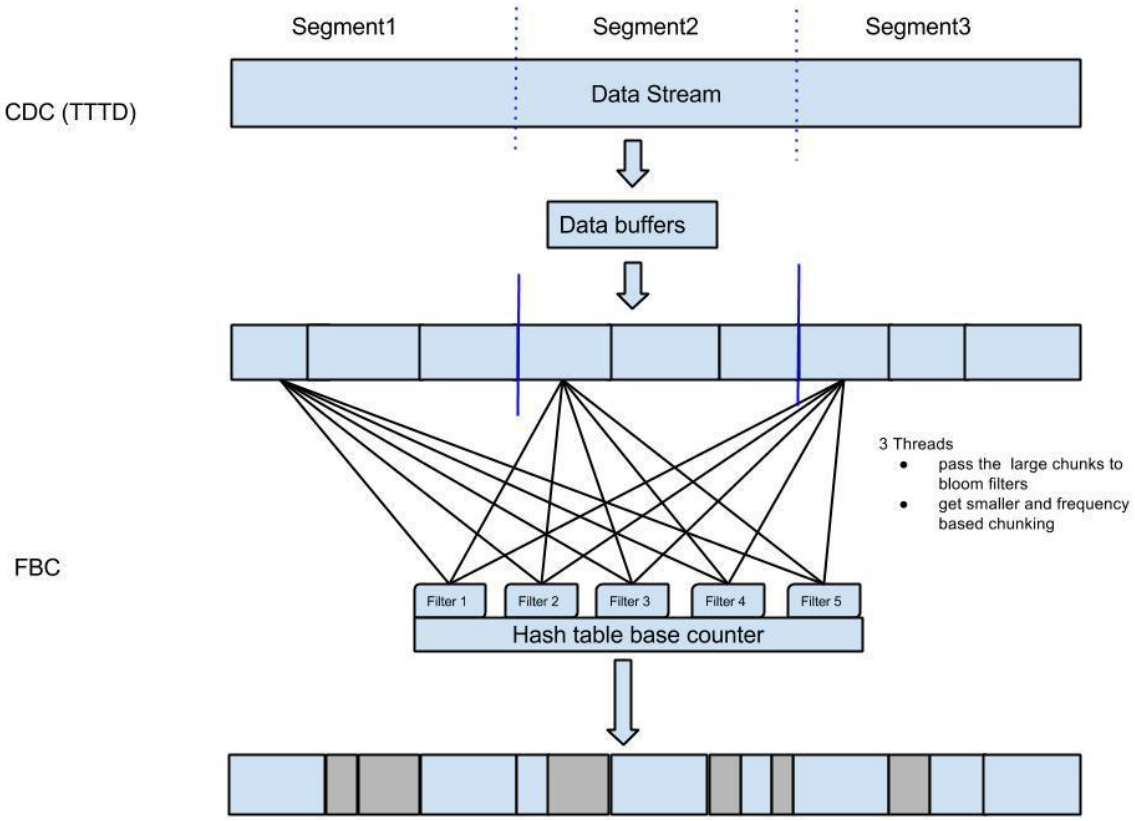

Figure 1 Theoretical Design

One critical implementation issue is keeping the two stages relatively same speed. If the CDC step is very fast, having more FBC threads will help overall performance; if the FBC step is too fast, having more FBC threads will not help since most of them will simply wait for more input from CDC. Therefore, the number of threads assigned to the CDC stage and the number of threads assigned to the FBC stage need to be tuned. We will adjust the parameters in the test phase.

The most significant innovation of our approach is combining FBC and multi-threading. In this approach we can gain better performance in terms of both data eliminating rate and running time.

In all experiments of the FBC paper[2], the FBC algorithm persistently outperforms the pure CDC algorithm in terms of achieving a better dedup gain or producing much less number of chunks(less metadata). However, the single stream FBC didn't achieve good performance in terms of running time. Since the phase of chunking is very CPU intensive, using multi-thread will make up for this performance issue.

Please note that FBC as described in the paper is not an inline de-dupe algorithm. For actual chunking to happen, the data stream needs to pass through the FBC algorithm twice: the first time to populate the frequency table, and the second time to generate the chunks based on the frequency table. Our focus is in the first pass (so we only generate the frequency table in the end, not actual chunks), but our technique can easily be applied to the second pass.

3. Hypothesis

FBC is a two-stage chunking algorithm. While the first stage of coarse-grained chunking can be done with multithreading as the MUCH paper demonstrated, the second stage is really ideal for parallelism since each segment can be processed by a different thread concurrently. We expect the performance improvement in the second phase to scale very well with number of CPU cores.

At this point, our focus is to apply multithreading to the second stage, but we keep our option open to use it in the first stage as well. The MUCH paper describes how such stage can be performed in multithreading. The overall chunking performance

for multi-threading depends on the relative cost of the two stages and how many threads we use for each stage. For example, if we use only one thread for the first stage, the following would be the ideal case:

- If each stage costs the same, and we use one thread for the second stage, the overall performance will be nearly doubled
- If the second stage is twice as slow as the first, and we use two threads for the second stage, the overall performance will be tripled
- If the first stage is twice as slow as the second, and we use one thread for the second stage, the overall performance will see a 1.5x boost (and the thread for the second stage will only be 50% busy)

Ideally, regardless of threading, the chunking result should be deterministic (Chunking Invariability). For FBC, the final chunking result with multithreading might be slightly different from single thread if consecutive segments have identical chunks. For example, if chunk X appears in both segment 1 and 2, in single-thread case chunk X might be identified as high-frequency chunk in segment 2, but not in multithread. This is because in single-thread case, the processing of segment 2 always happens after segment 1 which has processed chunk X. In multithreading case, due to parallelism and timing difference, during segment 2's processing chunk X may or may not be identified as high frequency chunk, therefore affecting the chunking result. This is only an issue if during processing of segment 1 chunk X happens to just cross the FBC high frequency threshold. We expect this to be very rare in practice (and the higher the threshold is the rarer this problem is). Furthermore, FBC itself already has some degree of randomness (e.g., populating the bloom filter).

4. Methodology

To scale chunking performance with CPU cores, we need to be able to distribute the chunking work across multiple cores and minimize locking overhead, and keep all threads streamlined with minimal stalling.

For implementation, we'll use C++ and pthread on a Linux host. We will write a program that takes a list of files for chunking, and outputs resulting chunks (fingerprint+size). We will use the same parameters as the FBC paper. The program will operate in either single thread or multi thread mode.

In multithread mode, there is a master thread and a pool of worker threads. When the data is coming in, the master thread performs the first stage of FBC chunking using TTTD, and dispatches each segment to a worker thread to be chunked into smaller sizes. The exact number of worker threads will need to be tuned to achieve the best performance.

In each worker thread, the second stage of FBC is performed. Accesses to the bloom filters and counter table are synchronized through read-write locks if necessary.

If the maximum chunk size for the first stage is M, and the number of worker threads is N, the program allocates N buffers each with size M, so each thread (including the master thread) can work on one buffer. The master manages the buffers by giving a buffer to a worker thread after completing the first stage and reclaiming a buffer from a worker thread after it completes the second stage. The master also generates the final chunks by aggregating information from all worker

threads.

Since our goal is to show improvement in chunking CPU performance (vs generating better chunks), we can use any data sets that generate reasonable chunks. We'll use multiple versions of the Linux kernel source tarball file as the input, as used in the FBC paper. For BSW (Basic Sliding Window) based algorithms, each file's chunking is completely independent of one another, so we could just use one single tarball; for FBC (Frequency Based Chunking), however, we need multiple tarballs to populate the bloom filters and the counting table to emulate a more realistic data flow.

To verify our hypothesis, we measure the CPU time of running the program with the same input data in both single thread and multi thread mode. We expect the second stage of FBC scales very well with number of threads.

We also expect very minor difference in generated chunks between single and multi threading. The difference if there are any can be measured by calculating the total size of unique chunks.

5. Implementation

A. Basic design

The FBC does chunking in two stages. The first is a coarse CDC chunking. The second is sub-chunking by using a frequency counting table. The basic data flow shows in figure 2. In fact, FBC processes the entire byte stream twice: the first to establish the frequency counting table, and the second does the actual chunking,

but here we are only focusing on the first pass that calculates the frequency counting table.

There are two queues and N buffers (each buffer with the maximum possible chunk size). There are N + 1 threads: N threads each doing the second stage FBC sub-chunking, while the main thread does the coarse chunking using the well-known TTTD algorithm.

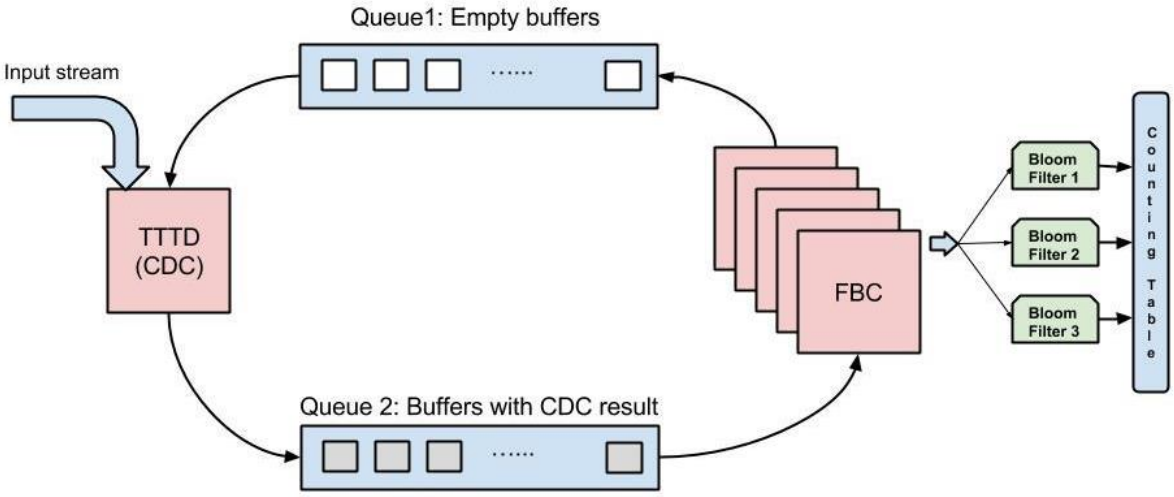

Figure 2 Basic Design and Data Flow

Initially, all N buffers are in queue 1. The main thread dequeues from queue 1, and fills each buffer with a coarse chunk, and enqueues into queue 2. The N FBC threads dequeue buffers from queue 2 and process them to populate the frequency counting table. After that the buffers are enqueued back into queue 1, and rinse and repeat.

B. Major data structures and algorithms

Figure 3 shows the major classes we implement and the relations between them.

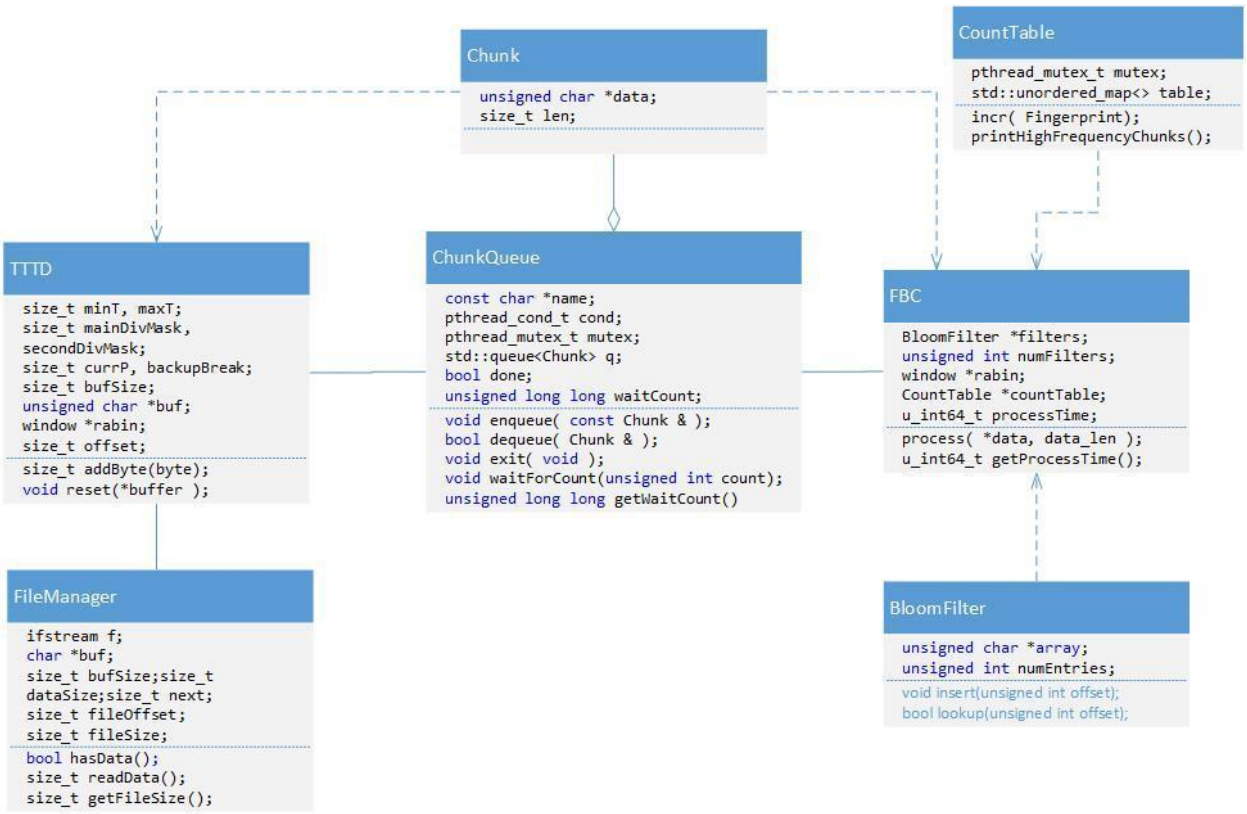

Figure 3 UML class diagram

- ChunkQueue: queue for chunk (addr+len) information. There are 2 of them, protected by mutexes and conditional variables.
- TTTD: The coarse CDC chunker. It takes a byte stream and outputs chunks. There is one instance of it run in the main thread.
- FBC: The Frequency Based Chunker. There are N instances of it, each run inside its own thread.
- BloomFilter: There are 3 bloom filters used by all FBC instances. It has no locking since collision is very unlikely, and even when it happens the effect is minimal. We use the Murmur3-128 algorithm to generate 4 32-bit hashes in one shot (although we only use 3). Code is from ://code.google.com/p/smhasher/wiki/MurmurHash3.
- Frequency Counting Table: There is one frequency counting table shared by all FBC instances. It's protected by a mutex.

- Rabin hasher: rabinpoly.cpp|h and msb.cpp|h are downloaded from ttp://www.cs.cmu.edu/~hakim/software.
- FileManager: Just a wrapper around ifstream to improve performance.

C. Memory usage

Three bloom filters consume 2.4MB. With the input the frequency counting table grows to 1.4MB entries, which amounts to approximately 45MB (20-byte key + 4-byte value + 8-byte pointer overhead -> 32-byte per entry). The chunk and rabin buffers are negligible. The total memory usage is around 48MB.

D. Usage

To build, just run "make" after unpacking.

Type ./fbc to see help. It provides many knobs but the only ones we use in the final report are -t to tune threads and -b to tune number of buffers.

The default number of buffers is the max of 2 times the number of FBC threads and 6. See the data analysis part for more discussion.

6.Data Analysis and Discussion

A. Parameters

The parameters are picked according to the FBC paper.

* CDC/TTTD parameters
  - minimum chunk size: 1B
  - maximum chunk size: 100KB
  - main divisor: 8KB
  - secondary divisor: 4KB
  - sliding window size: 64B

* FBC (second stage) parameters
  - prefiltering sample density rate: 32
  - bloom filters: 3
  - bloom filter size: 800KB (per filter)
  - counting bias: 5
  - frequency threshold: 5
  - frequent chunk size: 512B (1/16 of TTTD average chunk size ~ its main divisor)

B. Data:

We used emacs source code tar file downloaded from http://ftp.gnu.org/emacs from version 21.4a to 22.4. We feed the unzipped tar files into the fbc program to emulate the data stream.

C. Result Analysis:

Our program is run on the SCU linux cluster, using CPU: 3.7G HZ Intel Xeon 8-core and Kernel/Linux: 2.6.32 x86_64.

Our results show that threading improves the time performance of FBC chunking with a factor of 75%, for peak time cases. This finding confirms our hypothesis that multi-threaded FBC gives a significant improvement on the time performance compared to the single-stream FBC. Figure 3 illustrates this idea.

Peak performance is at 16 seconds with threading, vs 28 seconds without. This is a 75% performance improvement.

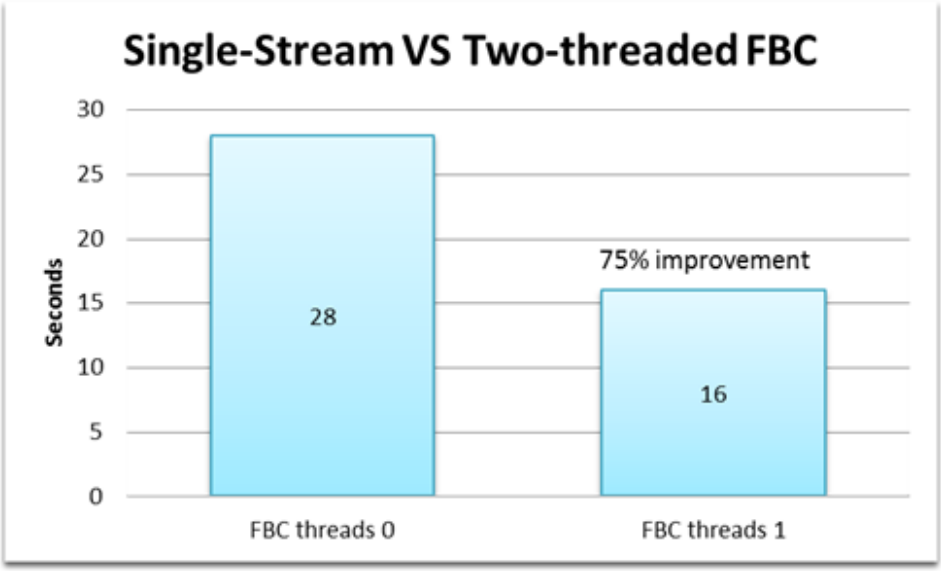

Figure 4 Single-Stream VS Two-Threaded FBC

For 1 thread in particular, number of buffers is interesting.

2 buffers: 70 seconds

3 buffers: 60 seconds

4 buffers: 31 seconds

5 buffers: 20 seconds

6 buffers: 17 seconds

7 buffers: 17 seconds

Low buffer count is pretty awful - it actually is much worse even than single threaded case. With more buffers the chance of parallel running of the two FBC stages is much improved. 6-buffer makes it decent enough.

With more than one thread, the default number of buffers (doubling number of FBC worker threads) is good enough which is almost as good as 2-thread with 4 buffers.

Adding more buffers to cases with 2 and more FBC threads doesn't help much.

Our finding is that a minimum of 6 buffers allows for the peak performance of two-threaded FBC. The full potential of the two threads working at the second is exploited when the buffers are no less than 6. This is shown in Figure 4.

In fact, the number of buffers for an N-threaded FBC depends on the relative performance of CDC and FBC threads. Also, the performance is not constant - depending on the data stream the relative performance varies. So even though in average the two phases cost the same, at various times one might be faster than the other and more buffers help deal with these cases.

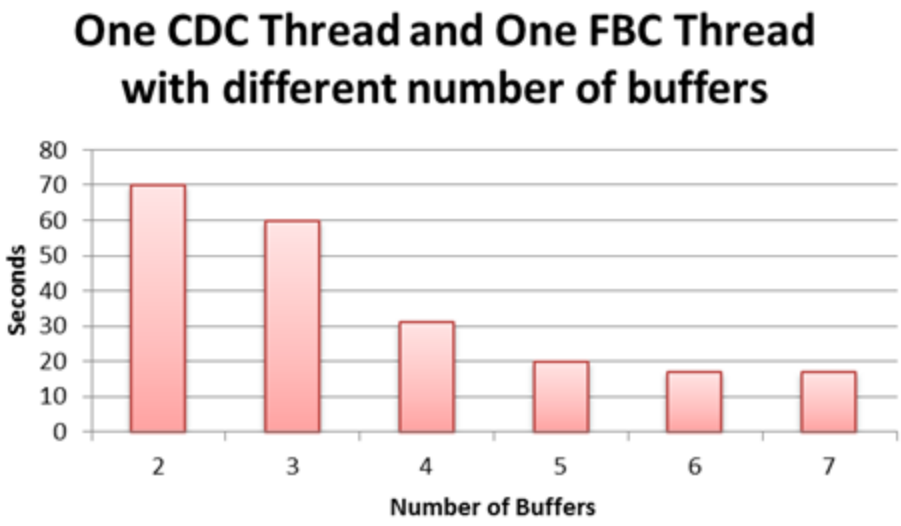

Figure 5 One CDC Thread and One FBC Thread with Different Number of Buffers

Another finding is how much multi-threading at the second stage can help improve the overall performance of FBC chunking.

No threading: 28 seconds
1 thread (6 buffers): 17 seconds
2 thread (6 buffers): 16 seconds
3 thread (6 buffers): 16 seconds
4 thread (8 buffers): 16 seconds

So it is saturated after 2 threads. Afterwards the main thread is the bottleneck and more FBC worker threads won't improve performance anymore.

It’s shown in Figure 5 that overall performance reaches the glass ceiling with two threads at FBC stage. From here on, the first stage, the CDC, constitutes the

bottleneck for performance. So the increase in threads at second stage will not improve the overall time consumed.

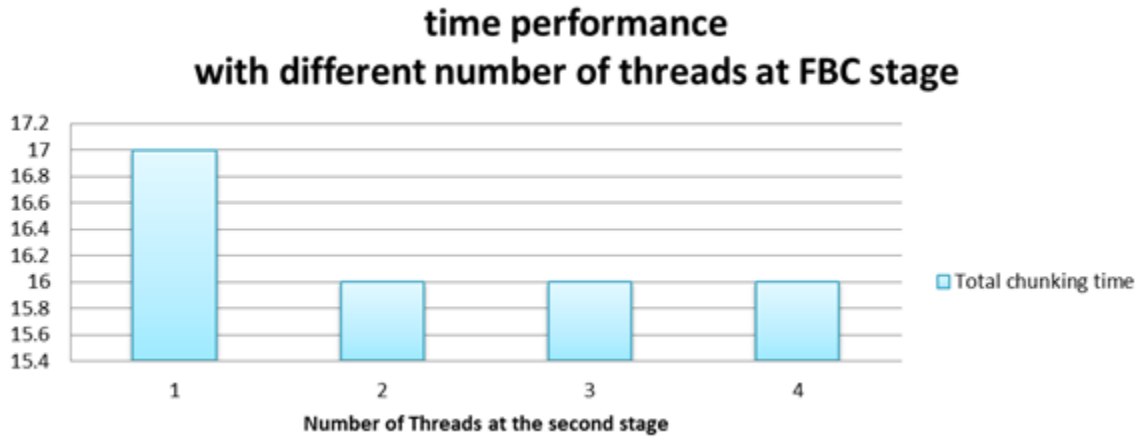

Figure 6 Time Performance with Different Number of Threads at FBC Stage

Another fact that justifies our bottleneck findings is the wait counts of the two queues. Each time the queue's request for a chunk is not satisfied, the wait count is increased by one. As is shown in Figure 6, the wait count is increased dramatically with the two-thread case. It's an indication that the threads at stage two spends more and more time sitting idle, thirsty for the first stage to provide coarse chunks. Therefore stage one, single-stream coarse chunking, is dragging the overall performance slow, when second stage reaches two threads.

The explanation for this result is that the two chunking stages have similar performance costs, so more than one FBC thread is not helping much. With only one FBC thread, the two threads run at roughly the same speed, and wait on each other depending on data pattern, so more buffers makes waiting less likely, improving performance. With more than one FBC threads the main thread is almost always the bottleneck and the one to wait, so more buffers do not help much.

Indeed, with "fbc -t 1 -b 6", the CPU usage is 180%, very close to optimal. With more threads, the CPU tops at 200%.

The results can be reproduced by running the 'runTest.py' file with the emacs source tar files.

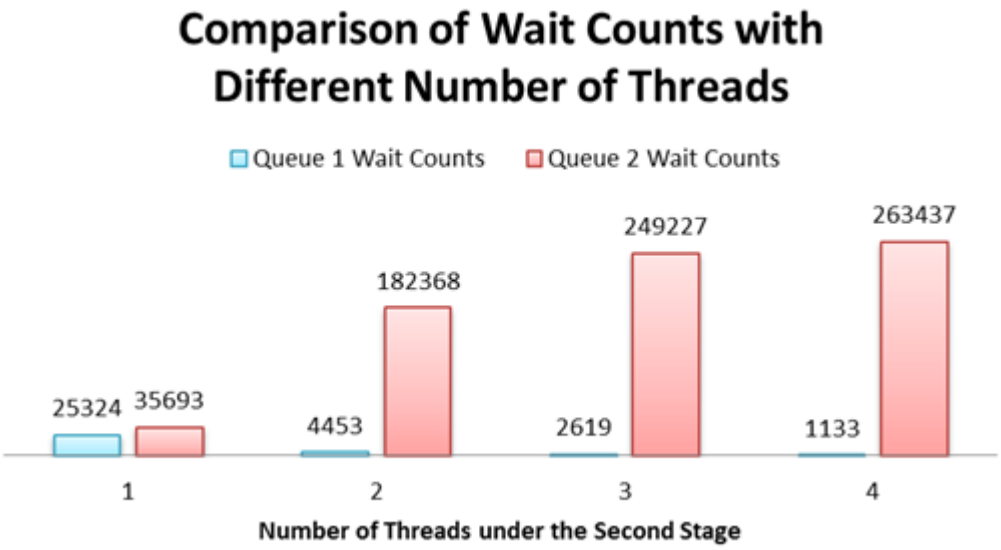

Figure 7 Comparison of Wait Counts with Different Number of Threads

Last, a figure of CPU ticks comparison between CDC and FBC threads (Figure 7) helps us summarize our multi-threaded algorithm. The parallel threads at FBC stage share the CPU time almost at the same rate. Because from time to time FBC threads give up CPU and wait for CDC thread to feed them, the total CPU time of FBC threads is close to that of the single CDC thread. This mechanism is accomplished by setting up two queues with chunk buffers rotating between them.

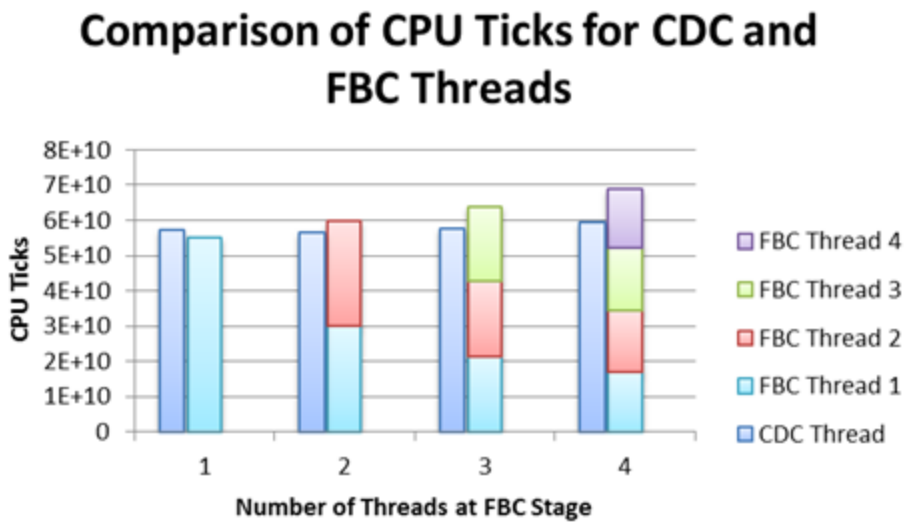

Figure 8 Comparison of CPU Ticks for CDC and FBC Threads

7.Conclusions and Recommendations

A. Summary and conclusions

Our experiment demonstrated that using multithreading can greatly improve FBC performance. With enough buffers (6), we see 75% performance improvement over single-thread implementation with just one dedicated thread for FBC second stage. Our test shows the two stages of FBC algorithm have similar performance, so we do not see greater gain with more than two threads. In order to achieve better performance, we'd have to apply multi-threading to the first stage as well.

The number of buffers also impact performance, especially with fewer threads. On the particular system we tested, 6 buffers are good enough.

B. Recommendations for future studies

From the data analysis, we can see that CDC chunking is the bottleneck when we have more than one FBC thread. We can combine the multithreading CDC chunking technique described in the "MUCH"[6] paper with ours to further improve the overall performance.

We can also avoid tuning the number of buffers by dynamically allocating buffers when a thread needs it. The additional memory requirement is minimal compared to other data structures.

Bibliography

[1] He, Li, et Zhang (2010). "Data Deduplication Techniques". 2010 International Conference on Future Information Technology and Management Engineering
[2] Lu, Jin, Du (2010). "Frequency Based Chunking for Data De-Duplication". 2010 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems
[3] Moh, Chang (2011). "A Running Time Improvement for the Two Thresholds Two Divisors Algorithm". ACM SE '10 Proceedings of the 48th Annual Southeast Regional Conference
[4] Tang, Won (2011). "Multithread Content Based File Chunking System in CPU-GPGPU Heterogeneous Architecture." 2011 First International Conference on Data Compression, Communications and Processing
[5] Won, Kim, et al. (2008). "PRUN : Eliminating Information Redundancy for Large Scale Data Backup System". International Conference on Computational Sciences and Its Applications ICCSA 2008

[6] Won, Lim and Min (2014). “MUCH: Multithreaded Content-Based File Chunking”. IEEE Transactions on Computers, Volume:PP , Issue: 99