Parallel Data Mining

JASON MA and KANISHK TANTIA
Harvey Mudd College

1. ABSTRACT

There is currently vast interest in both, High Performance Parallel Computing and in Deduplication efforts to save space and reduce memory usage. While deduplication methodology is still in it's infancy, a major target for improvement are the "Chunking" algorithms that take in file or block level traces and identify candidate blocks that may be similar to each other across the trace. These chunking algorithms are primarily serial in nature, due to the serial, interconnected nature of block and file traces. In this paper we propose a parallelized version of a commonly used Chunking algorithm based on the Rabin method, dubbed "Rabin Context Defined Chunking" or RabinCDC. The key idea behind Parallel RabinCDC is to apply data parallelism to the trace to apply the cyclic nature of RabinCDC to different sections of the trace without introducing large differences in breakpoints across trace data. We use multiple processes to carry out a serialized version of RabinCDC on multiple sections of the trace which are broken apart in a context defined manner. We further compare parallel RabinCDC to FastCDC, an industry standard serial chunking algorithm, to Serial RabinCDC, and to AECDC, a now defunct chunking algorithm. Our comparisons show large speed and efficiency improvements over the serial CDC algorithms over large traces with multiple processes running. Parallel RabinCDC also had few differences in the identified break points, which ensures that the deduplication ration remains nearly identical.

2. INTRODUCTION

Data storage needs have grown exponentially as more data is increasingly digitized. IBM estimates that 2.5 Quintillion bytes of data are created daily. IDC research estimates that digital data growth will grow at a compound growth rate of 11.7% through 2020. As of 2011, the demand for more storage has outpaced the growth of digital storage means, and by 2020, it is estimated that the demand will outpace the growth of digital storage by over 15,000 Exabytes of data per year.

Data Deduplication is an efficient manner in which data usage can be reduced, and has slowly gained traction over the last decade. Assuming that a large amount of the data being stored is redundant or "junk" data, data storage needs can be met by simply never storing multiple copies of data and instead just providing the same set of bytes when the data is required. Data deduplication therefore removes redundant data at a file or chunk level, and uses a hash code to identify similar blocks, which are then compared in order to eliminate redundancies while salvaging important data.

A critical requirement of data deduplication is efficient chunking, which involves looking at traces of data and identifying blocks of data that may be similar to each other. The research standard for chunking is at the block level, as this allows for the data stream to be fine tuned and allows for deduplication to proceed at a much finer granularity. Chunk level deduplication has been explored by various authors. The simplest method, which is not considered in this paper due to it's lack of practical application, is Fixed Size Chunking [Quinlan et. al]. Since FSC is unable to actually account for slight shifts in data, and changes every subsequent breakpoint after a small change in files due to a cascading effect, it has been considered defunct for a while. This is an example of a *boundary shift* problem [Muthitacharoen], which is an unwanted shift in data that has not been modified through either an insertion or deletion.
Context Defined Chunking, (CDC), is a solution to the *boundary shift* problem. CDC methods declare chunk boundaries on the basis of only the data contained within each block, and therefore performs some preliminary block analysis before declaring a breakpoint. This has the drawback of a higher CPU overhead, but the advantage of avoiding cascading shifts due to small modifications. Moreover, the preliminary analysis also allows for a higher deduplication ration, as the fast analysis allowed for by CDC methods ensure that blocks need to meet certain criteria based on their content.

The more complex CDC methods considered are RabinCDC [Rabin], FastCDC, [Wen et al] and AECDC [Zhang, Yucheng at. al]. These methods are currently industry standard, except for AECDC, and are all highly effective at detecting context defined breakpoints. Each of these methods is considered to be accurate at identifying candidate blocks for deduplication, but, with the exception of FastCDC, each has a high CPU Overhead. FastCDC is a sequential algorithm that was specifically designed to have a low CPU Overhead, but has a worse deduplication ration than RabinCDC and AECDC.

CDC methods are sequential in nature, since they run through the trace and identify breakpoints based on each line of the trace. FastCDC specifically is extremely trace dependent, and is impossible to parallelize without causing a significant cascade effect in the breakpoints. However, RabinCDC is cyclic in nature, and judges each block of data by hashing it and comparing it to stored hash values, using a cyclic and rolling hash algorithm.

In general, we observe that none of the current CDC methods are parallelized. While algorithms like FSC do not require parallelization, as they are simplistic enough to perform well without any added parallelization, it is certainly necessary for CDC methods, which judge traces based on block data, and therefore have an overall O(n) time at best. Given that trace data routinely reaches very large sizes, parallelization could certainly improve the Big O time of these algorithms, while changing the output very little.Motivated by this observation, we proposed Parallel RabinCDC, which has the following characteristics:

—**Task Parallelism**: Parallel RabinCDC is a wrapper that runs the sequential RabinCDC algorithm on multiple processes, thus reducing Big O time to $O(\frac{n}{p})$

—**Efficient Hashing**: Parallel RabinCDC uses a simple cyclic hash, similar to the FastCDC hash methodology, in order to increase hashing time on each parallel processor

We see that there is a linear improvement in processing speed under Parallel RabinCDC vs number of Processors, and that the strong and weak scaling are both linear in nature. We further observe that the Parallel RabinCDC algorithm is modular in nature, and would be trivial to implement in research involving the current sequential version assuming that Parallel Processing capabilities are enabled.

The rest of this paper is laid out as follows. Section 3 presents background information that is necessary for this research. Section 4 discusses the design of Parallel RabinCDC algorithm. Section 5 discusses testing Methodology and trace data. Section 6 presents an evaluation of the experimental work. Section 7 presents results on practical and synthetic trace data. Section 8 discusses future work and conclusions.

3. BACKGROUND

3.1 General Chunking

Chunking, as mentioned, is the first step in data deduplication, and primarily breaks large traces into smaller blocks that can then be compared in order to identify duplicates. Fixed Size Chunking [Quinlan] has a low CPU overhead and is extremely fast, but also inaccurate. It is unable to handle *boundary shift* problems, so that any insertions or deletions in a file cause a cascading effect that disrupt every breakpoint after the modification.

3.2 FastCDC

FastCDC was first proposed in 2016 [Wen et al]. It involves a simplistic hash judgement mechanism and an efficient bit-shifting hashing algorithm that increases hashing speed. FastCDC is currently considered to be an industry standard chunking algorithm.

FastCDC first generates a random 1:1 mapping of every 8 bit number to a random value in a hash table. It then runs through the trace, and accounts for each block number in a running total, with upper and lower bounds on block size. The block number first undergoes a bit shift, and then the bit-shifted value is looked up in the hash table. The looked up value is added to a running total, and once the running total is divisible by a given value, a breakpoint is designated [Wen et al.]

Unfortunately, FastCDC is very difficult to parallelize as the granularity of the tasks that we can run in parallel are too small to see much benefit overall compared to running it serially. However, it is a good upper bound or "golden standard" for algorithmic runtime.

3.3 RabinCDC

RabinCDC involves a cyclic, rolling hash. Proposed by Rabin as a simple algorithm in 1981 for the identification of similar substrings, it has been modified to work for block addresses. Rabin is a variable or Dynamic chunking algorithm, which has a *window* of certain size. Every time the contents of the window match a predefine criteria, a breakpoint is introduced. If this fails to happen, the window moves forward by one byte and the process repeats.

The Rabin Rolling Hash we designed works in the following way. The window of size n is summed, and bit shifted by 1 to the left. The bit-shifted sum is then modded against a prime number generated by random seed, and if the resulting value is 0, a breakpoint is introduced. This is a modified version of the original Rabin Fingerprint, which was found to produce similar results to the original Rabin Algorithm, as per the PCompress Project.

Unlike FastCDC, RabinCDC is a slower algorithm. However, it is extremely parallelizable, because it guards against insertions and deletions due to the cyclic property of the hashing algorithm. As the hashing algorithm doesn't depend on past block addresses, but simply on the current n valid bytes, splitting up the trace means that the breakpoints are largely unaffected. Considering the likelihood of a cascading effect is extremely low, RabinCDC is perfect for parallelization.

The sequential version of RabinCDC is used to provide a primary lower bound for the parallel version, and provides information on the improvement from control that is visible post-parallelization.

3.4 AECDC

Asymmetric Extremum Chunking, or AE CDC was introduced in 2015 [Zhang et al.] AE CDC is fast, but is still slower than RabinCDC and FastCDC, while being upto 3x more accurate.

AE CDC depends on extreme values and two candidate windows. The first window of size n contains bytes behind a candidate block address. The second window of size m contains the bytes immediately ahead of a candidate block address. Each new block address is considered a viable breakpoint, and is eliminated as a breakpoint if it does not meet both the following criteria:

—The block address must have a value greater than any block address seen in the *tail* window.
—The block address must be greater than or equal to any block address in the *head* window.

AE CDC is a slightly slower algorithm than RabinCDC or FastCDC. Thus, it is apparent that this algorithm will provide a useful lower bound of how slow our parallelization will affect Rabin CDC.

4. DATA

4.1 Practical Trace Data

Practical Trace Data was obtained through the SNIA IOTTA Trace Repository, which stores traces from various research endeavours across the world. Specifically, Block I/O Traces from the Florida International University, collected on the home directories of multiple different end users were used for practical data. Practical Trace Data numbered upto 3.5 Million lines on the largest traces. We used `blkparse` to read in the trace data and used a python script to extract block number information from it.

For overall testing, we used specifically the *homes1, homes2, homes3* and *homes4* traces. These were chosen because they belong to 4 different end users and record block access over a period of two weeks, which allows for a fair amount of statistical variability while also allowing for patterns in end-user usage to be apparent [Verma, Koller et al.]. Moreover, for the generation of synthetic traces (See Section 4.2), we needed practical trace data with easily identifiable patterns so that we could use a simple Markov Chain Generator to synthetically increase trace sizes.

4.2 Synthetic Trace Data

In order to gain a more complete understanding of parallelization improvements, we synthesized trace data using a simple Markov Chain on the practical data. We generate this new file using an aggregated file of our practical trace data as a source. The size of the chunks with which we sampled the source files ranged from 1 line to 20 lines. This method allows us to maintain patterns from the data it is sampled from while allowing tests at large volumes. Thus, we can more carefully tune our tests to our specific requirements, and test conditions outside of what the practical trace data covers. Additionally, the randomness of the index from which we sample and the size of the chunk we sample out allows us to maintain randomness in our testing data to mimic real world situations.

For our testing, we created files that are 1 million to 200 million lines. We increment the file sizes by 10 million lines from 1 million to 100 million, and by 50 million between 100 million and 200 million lines. These file sizes will be adequate in covering the full behaviour our parallelized RabinCDC at small file sizes, and its behaviour as the file sizes approach infinity, which let's us analyze the O(n) behaviour. Additionally, the choices for which sizes we used were chosen, so that the weak scaling efficiency could also be easily tested.

5. EXPERIMENTS

We tested our algorithms in a 64-bit Python3 environment, using the `multiprocessing` module to implement parallelization. We will note that this only creates multiple processes, and allows the CPU's scheduler to handle the parallelization of these processes, but these results are still valid in that they show how one can easily parallelize the RabinCDC algorithm for significant runtime improvement.

We shall further note that Python3 was chosen as the testing environment primarily due to the ease of integration of the python `multiprocessing` environment, as well as the significantly low overhead of said environment in Python3 [Fowler]. Moreover, the Python3 dev environment was already set up on Comet and on our test machine. Further, we realized that the total communication time and startup time between processes could be significantly lowered due to the `pool` utility. Finally, we also found that Python3 does it's own optimization of the interpretation process. Due to the significant performance improvement found in the implementations of Python3 `multiprocessing` over our original implementation of C++ `OpenMP`, we decided to implement the development in Python3.

5.1 Practical Trace Data Tests

With the practical trace data, we timed the run time over 10 tests using from 1 process to 5 processes. Given the uneven size of these files, it is only feasible to test the strong scaling efficiency.

5.2 Synthetic Trace Data Tests

With data that we can generate, we gain much better control over the types of tests that we can run. Thus, using synthetic trace data, we can specifically target the strong and weak scaling efficiency of a parallel Rabin CDC algorithm. To generate data, we run the tests 10 times and average the time across all the runs to produce an average run time.

To test the strong scaling of our parallelization, we ran our algorithm on the test data serially, and then increased the number of processors up to 5. We time how long it takes in each case, starting from when the algorithm is passed the data in the serial case and starting from when we create the new processes in the parallel case. Our tests ranged from files that were 1 to 200 million lines long. Using these results, we can uncover how well parallelization affects Rabin CDC at small to large file sizes.

To test the weak scaling efficiency, we also created file sizes ranging from 10 to 50 million lines long. We then ran our CDC algorithms on these files keeping a constant ratio of 10 million lines to one process.

To specifically target our parallel speed up at large file sizes, we created a test file about 200 million lines long, and ran it using one to ten processes.

Given the larger file sizes, it is apparent that we would need to use better computers to test the effectiveness of using more processes. Thus, for the synthetic data, we used a computer with 4 cores, 8 logical processors and a CPU name.

5.3 Baselines

In addition to RabinCDC, we also run AECDC and FastCDC on each file to get an idea of how well our parallelization compares to well known context defined chunking algorithms. We run the algorithms on the exact same files, and average times across 10 runs.

6. RESULTS

6.1 Practical Trace Data

When running tests on small data files (number of lines less than 1 million), there were no significant improvements in the runtimes. In fact for the very small files, there were adverse effects on the run time. We believe these results are due to the fact that at these small file sizes, the benefit of using multiple processes is outweighed by the cost of spawning said processes. This may indicate that parallelizing RabinCDC for small file sizes is not optimal, and may in fact incur a cost.

In contrast, there seem to be a clear benefit to parallelizing RabinCDC at large file sizes. Particularly for homes3, and homes1212, which had 2.5 million lines and 3.5 million lines respectively, there were great improvements. We see on the system we tested on that we were able to reduce the run time of RabinCDC to about 50% of the serial run time. This shows great improvement since parallelizing with the `multiprocessing` module was relatively simple. This implies that any code can be parallelized with ease and great benefit. As we can see on these relevant figures, the run time noticeably decreases when increasing the number of processes. Interestingly, when we compare the strong scaling factor of the most noticeable improvements, we see that the scaling efficiency actually decreases as we increase the number of processes.

Our weak scaling efficiency shows a moderate decline as we increase the number of processes, which indicates there must be some

extra factor slowing down the CDC algorithm besides the computation of the break indices.

7. CONCLUSIONS

We were able to obtain significant improvement of the run time in our context defined chunking algorithms by parallelizing them using the Python3 `multiprocessing` module. At large file sizes, we saw speedups of up to 50% reduction in the amount of time required to process the file when using 5 processes. This may imply that the benefits of parallelization quickly reach their limit on personal systems such as the laptop we tested on. While supercomputing centers like the one at San Diego will probably see benefits including many more processes, this is still an important result as it shows that any individual can potentially introduce significant run time improvements to their code using the `multiprocessing` module in Python.

When we graph the strong scaling efficiency vs number of processes in fig 3, we see that there is a larger decrease in strong scaling efficiency for the larger file sizes. Since we are using the `multiprocessing` module in Python, the module just spawns new processes with specific sections of the data to work on. This result would be obvious when considering the purpose of the Python module we used. Since it just spawns new processes, eventually the results showcase more the effectiveness of the system's scheduler than the benefits of parallelization. As a result, it is clear that there are diminishing returns for spawning new processes, which impacts the possible benefits of parallelization.

As for our weak scaling efficiency, we do not notice that much of an improvement over time. In fact the run time is much worse than just one processing element distributed to one process. We believe that this discrepancy is due to the more costly data transfer costs incurred when allocating the large file into its separate processes. This highlights the costly nature of communication and data synchronization when parallelizing algorithms.

Unfortunately, the Big O runtime of our algorithms were still O(n). Though this makes sense when analyzing the purpose of parallelization, which only splits up the work among many processes, so increasing the workload still increases the processing time linearly, just by a lesser amount than if we ran the code in serial.

REFERENCES

[1] Wen et al. FastCDC: a Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. USENIX Annual Technical Conference (USENIC ATC 16), Jan. 2016, www.usenix.org/system/files/conference/atc16/atc16-paper-xia.pdf.

[2] Muthitacharoen, Athicha et al. A Low-Bandwidth Network File System. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles - SOSP '01, 2001, doi:10.1145/502051.502052.

[3] Quinlan, Sean, and Sean Dorward. Venti: a New Approach to Archival Storage. Proceedings of USENIX Conference on File and Storage Technologies (FAST02), Jan. 2002.

[4] Zhang, Yucheng et al. AE: An Asymmetric Extremum Content Defined Chunking Algorithm for Fast and Bandwidth-Efficient Data Deduplication. 2015 IEEE Conference on Computer Communications (INFOCOM), 2015, doi:10.1109/infocom.2015.7218510.