Parallel Data Mining

JASON MA and KANISHK TANTIA
Harvey Mudd College

In this project, Kanishk and I procrastinated until the end to start, and now we are absolutely fucked

1. 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.

2. DATA

2.1 Practical Trace Data

2.2 Synthetic Trace Data

In order to gain a more complete understanding of parallelization improvements, we synthesized 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 behavior our parallelized RabinCDC at small file sizes, and its behavior as the file sizes approach infinity, which let's us analyze the O(n) behavior. Additionally, the choices for which sizes we used were chosen, so that the weak scaling efficiency could also be easily tested.

3. ALGORITHMS TESTED

3.1 FastCDC

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.

3.2 RabinCDC

Unlike FastCDC, RabinCDC is a slower algorithm, but it is very easy to parallelize, which makes it a great candidate to attempt to parallelize.

3.3 AECDC

AE CDC is a much 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. EXPERIMENTS

We tested our algorithms in a 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 any algorithm for significant runtime improvement.

4.1 Practical Trace Data Tests

We used practical trace data from the work of Kanishk and Keller from the summer of 2016. Tests were run using a Surface Pro 3 with 2 cores.

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.

4.2 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 the 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.

4.3 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.

5. RESULTS

5.1 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.

6. 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.