Task 3
GitHub

1 Introduction

The topic of the third assignment is performance benchmarking. The motivation behind this task is that performance matters in every part of the system. To understand the different types of performance bottlenecks in the system we perform various benchmarks. There are four parts of this assignment. You can reproduce each of them separately by visiting our GitHub repository and following the instructions in the readme files in the corresponding folders.

2 Key-Value store benchmark

The motivation behind the first part is that KV stores are gaining popularity and importance in the industry. In order to choose an implementation for a specific use case we need to be able to compare them.

2.1 RocksDB and memcached

In this part, we measure performance of a persistent Key Value store `rocksdb` and compare it with [7] `memcached`, an in-memory implementation. The latter is used by many companies for caching on top of persistent databases [10]. For benchmarking we use the [4] `Yahoo! Cloud Serving Benchmark (YCSB)` tool.

To set up `rocksdb`, we clone and compile the [6] source code. The build output is a C++ library which we use in `initRocksDB.cc` to create an instance of the database. For `memcached`, as well as for many other tools we specify the package in `shell.nix` and run `nix-shell` to open a shell with the specified packages.

2.2 Benchmarking

The YCSB program offers a user-friendly interface for KV store performance evaluation. For `rocksdb` testing we use the `ycsb-0.17.0` release. The tool offers six core workloads: update heavy, read mostly, read only, read latest, short ranges and read-modify-write. We also added a custom, write only workload `g` to benchmark the databases for with write operations with large values. You can find a more detailed description in the [8] documentation. Running the YCSB benchmark consists of two phases. In the load phase, YCSB generates the data and populates the database. In the run phase, the tool performs the operations specified in the workload file. Although YCSB provides metrics for both steps, we are only interested in the run phase.

The independent parameter is `-target`. It describes how many operations per second the tool should request. To evaluate the performance and scalability we increase this parameter exponentially and monitor the latency. We set the upper limit for target throughput to 2^{21} as none of the databases exceeds 2^{20} ops/sec. We keep the records count constant at 5mm records and the number of threads for YCSB is always 64 (utilize all cores).

For some reason YCSB reports errors for a part of `UPDATE` and `READ` operations when benchmarking `memcached`. Nevertheless, `memcached` logs retrieved with `-vvv` verbosity mode did not contain any failure messages. All operations appeared successful: Each `ADD` call is followed by a `STORED` response and after each `GET` there is an `END`.

2.2.1 Workload A: 50/50 Read/Write

Figure 1 shows the plots of latency against throughput. You can see that the latency is high for small throughput, then falls sharply and grows steadily again for higher throughput. This holds for most workloads, not only for A.

One phenomena specific to write-heavy workloads is that, unlike for read-heavy workloads, the latency variance is high. Researchers from the Parallel Data Labaratory at Carnegie Mellon observed the same behavior when performing a multiphase test [9]. We also observe that `memcached` (right plot) latency remains lower and that it reaches slightly higher throughput than `rocksdb` (left plot).

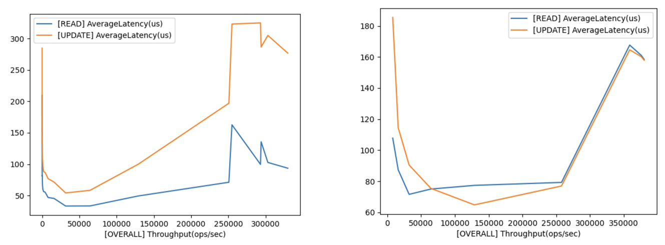

Figure 1: Latency vs. Throughput
50/50 Read/Write workload

2.2.2 Workloads B,C: Read-Heavy

Workloads B (95/5 Read/Write) and C (Read-Only) are read-heavy. Unlike with workload A, the latency and variance are small. The dependant variable is also growing slower and reaches less than 100 microseconds for 1.6mm ops/sec. Figure 2 shows the plot for `rocksdb`. The plot for `memcached` is similar, but the latency is around 2 times higher.

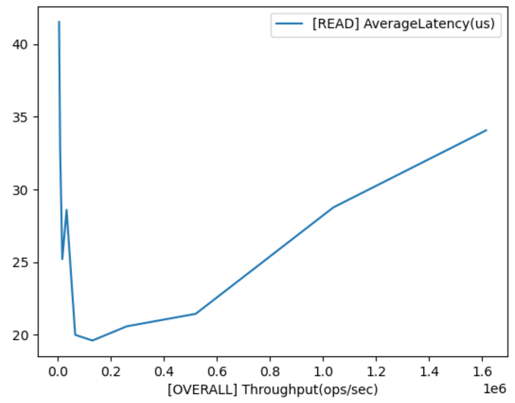

Figure 2: Latency vs. Throughput
Real Only workload

2.2.3 Workload D: Read Latest

D is an interesting workload. As described in the documentation [8] ‘In this workload, new records are inserted, and the most recently inserted records are the most popular’. This is a a common scenario for many modern applications. The author mentions user status updates in social networks as an example. The notable point is that on `memcached` the latency curve is very flat and almost not increases for increasing throughput. For `rocksdb` the result is similar to a read heavy scenario.

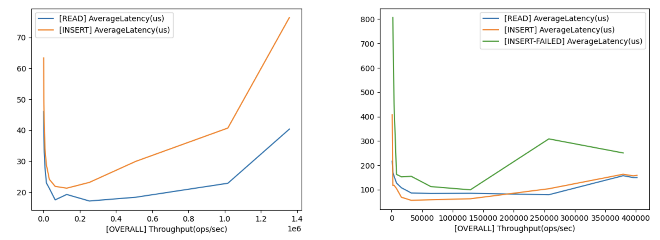

Figure 3: Latency vs. Throughput
Read Latest workload

2.2.4 Custom Workload: Write Only with large files

Our custom workload is a write only workload. The value size is 10kB. Although this may not seem a lot, it is 100 times larger than in other workloads. Figure 4 shows the resulting latency/throughput graph. `memcached` on the left, `rocksdb` on the right. For the first, latency for `UPDATE` increased 10-20 fold compared to workload A. The maximum throughput also degraded 6 fold. The throughput for the latter reached only 3000 ops/sec with a latency of 14000 microseconds. The big difference in both metrics comes from the fact that `memcached` is an in-memory implementation, whereas `rocksdb` has to access lower level memory. We suppose that for smaller values the difference is not as significant because `rocksdb` uses caching and accesses the persistent memory less often if the data size is smaller.

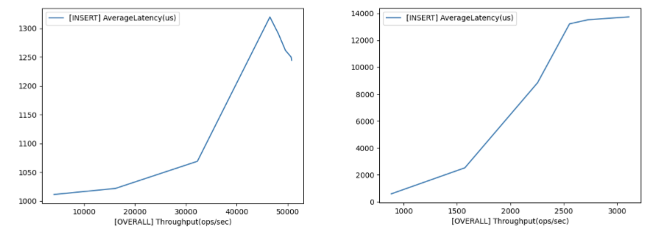

Figure 4: Latency vs. Throughput
Write Only workload

2.3 TPC-C Benchmark on RocksDB

The Transaction Processing Performance Council Benchmark C [3] (TPC-C) is the industry standard to measure the performance of online transaction processing (OLTP) systems. The benchmark simulates online transaction transactions, such as creating new orders, payment, order status checks, and warehouse stock transactions.

For this benchmark we use Docker with `MariaDB:10.3` image to set up MyRocks. MyRocks uses the rocksdb engine and MySQL syntax. For MySQL to use the rocksdb engine, we specify the engine in the config file `config-file.cnf`. The

TPC-C benchmark implemention that we use is implemented by [2] `percona` and uses [1] `sysbench`.

Like in the simple benchmark, we measure latency against throughput. The plot shows that we could reach a maximum throughput of around 6 thousand transactions per second. The latency went from 2.5 ms at under 1000 transactions per second to 10 ms at the maximum throughput. This is a hundred-fold increase from the read-only YCSB workload and almost a ten-fold increase from our custom write only workload.

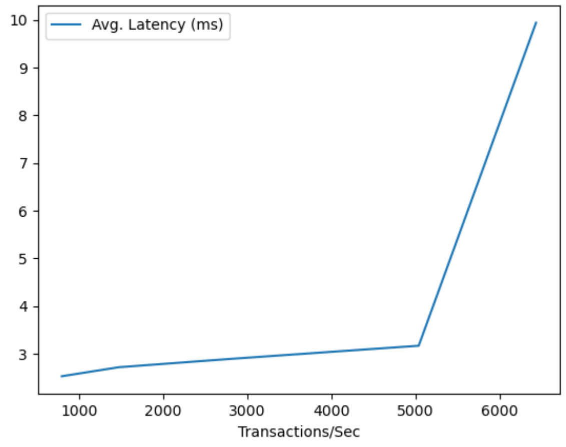

Figure 5: Latency vs. Throughput
TPC-C on MyRocks

3 Filesystems benchmark

3.1 Description

Ext4, *fourth extended filesystem*, created as extension for Ext3 mainly to overcome storage and performance limits, has been the defacto standard for Linux setups. Nonetheless, is currently the default file system for both *Debian* and *Ubuntu*. Other filesystems, including Btrfs, *B-Tree file system*, are in active development and they implements different set of features in order to give end user the vastest choice on file storage solutions.

3.2 Ext4 vs. Btrfs

For the extend of the paper and benchmarks, an introduction to these two filesystems main characteristics and diversities needs to be introduced. Ext4 is namely a Journaled Filesystem, in which a "journal" is kept in order to list file positions and changes. Btrfs uses a B-trees to store internal file structures, achieving searches, sequential access, insertions, and deletions in logarithmic time [11]. BTRFS main features that ext4 lacks are CoW (Copy on Write) policy, in which changes to files are mainly made on free available spots on the drive, instead of replacing actual already existing data, support for 16 Exbibytes partitions, native support for CRC32C checksum for data ingerity and data snapshots for easy-to-use rollback to previous file versions [12].

3.3 Setup and Methodology

In this paper we will discuss throughput and latency performance of both Ext4 and Btrfs filesystems, created on the same physical drive mounted on the same environment, using `FIO`, *Flexible I/O tester* [15], as benchmark tool.

`FIO` offers a very light framework to simulate particular type of I/O action using threads or processes, as specified by the user. `FIO` jobs could be declared using proper formatted *job-files* or passing job descriptions as *cli* arguments, and options includes output format, multiple working IO thread, percentage of read/write mixing, IO library to use, and so on.

3.4 Basic task: FIO Benchmarks

In our analysis, we will evaluate random read bandwidth, stressing the system using different settings of block sizes and performing measurements using both `Cached IO` and `Direct IO`

3.4.1 Workload A: Bandwidth Random Read w/ different block-sizes and IO types

In this first benchmark, we will test both file systems with 6 different settings, using `64K, 128K, 256K block-sizes`, all tested using both `Direct IO and Buffered IO`. We're currently focusing on average read bandwidth using one measured job performing random `30GB` read on the choosen file system. Other job specifications have been selected:

- gtod_reduce=1, using this option we achieve to use the gettimeofday(2) syscall just about 0.4 compared to non-optimized routines;
- iodepth=8 * –numjobs=4, evaluation and comparison has been made in prior to find 32 with a iodepth*numjobs combination, as sweet-spot between accuracy and velocity, also, it hasn't been found as proper bottleneck or data-biasing in .json reports.
- –ioengine=io_uring, for this benchmark, I chose to use io_uring as IO engine as it's linux native, fast, async and supports both direct and buffered IO.

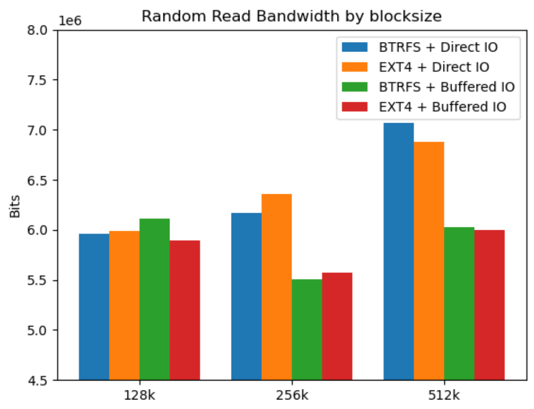

Figure 6: Random read Bandwidth for block sizes

While buffered `ext4` and `btrfs` are mostlikely on the same trend, we do have a a noticeable thoughtput enhancement on the 512k blocksize benchmark, where the `btrfs` partition looks to have a better response to the workload. Another really interesting data this benchmark shows is that with <128k block size settings, we do not really have wide differences using a Buffered or a Direct IO for performing writes.

3.5 Exploration task: Phoronix benchmark

`Phoronix` [13] is an extensive test suite for performance benchmarking. It supports a various set of OS, including Windows, Mac, and Linux. Nonetheless, the remarkable integration OpenBenchmarking.org, makes this tool useful for any needs, supported by community, with 450 test profiles and over 100 test suites. using `phoronix-test-suite list-tests` let the end user visualize which test could be supported and installed natively. We aimed to perform benchmarks using `FS_Mark`, as it is purely designed to test a system's file-system performance [14]

3.5.1 Phoronix x FS Mark benchmark

FS Mark allow us to use four different write settings between:

1. 1000 Files, 1MB Size
2. 1000 Files, 1MB Size, No Sync/FSync
3. 5000 Files, 1MB Size, 4 Threads
4. 4000 Files, 32 Sub Dirs, 1MB Size

We'll focus on a comparison between both File Systems using 32 subdirectories and how it can affect performances [17] [18].

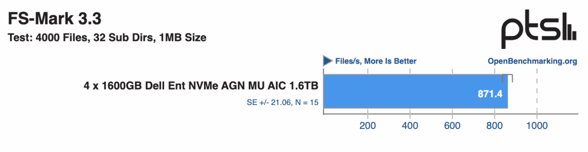

Figure 7: EXT4 FS Mark benchmark

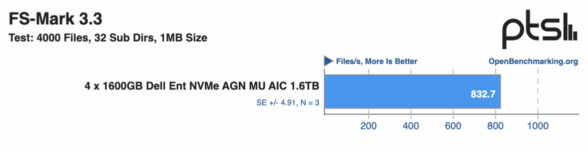

Figure 8: BTRFS FS Mark benchmark

As we can see,performances are slightly degraded in heavy thoughtput phases in which many files and folders are created.Due to the different write policy, `ext4` journaling looks more performant than the `btrfs` Copy-on-Write paradigm.

3.6 Conclusion

New standards and file systems implementation are becoming more and more concrete and solid, supporting, as the btrfs case, new features like native checksums, rollbacks and extended partition sizes. Nonetheless, Ext4 has been proved to be the way to go for an out-of-the-box implementation, in terms of write-read performances and overall integration with Linux kernel environment.

4 Networking

The aim of this task was to analyze various network stats using POSIX sockets, and more precisely, measuring the effect of different connection modes on TCP/UDP bandwidth, packet loss and average packet jitter. Packet jitter is the average sender delay between sending two consecutive requests.

4.0.1 Iperf

Iperf was initially released about 20 years ago, and has been completely rewritten from scratch as iperf3 and released on 2014. Iperf is extremely customizable, letting users test out individually tailored scenarios, such as number of parallel connections, TCP window size, payload size, duration of test, file upload instead of static content, reverse, bidirectional communication, syntax of output and more.

4.1 Basic task: analyze network performance with Iperf

4.1.1 Setup and Methodology

The different connection modes which we considered were parallel connection (and the effect of increasing the number of parallel connections on speed, packet jitter and loss), zerocopy (where instead of the normal `socket.send()`, `socket.sendfile()` is used, reverse (where the server and client switches roles, the server sends and the client receives), and bidirectional (server and client both send and receive). We also varied TCP window size, payload size, and number of connections, and measured their impact on the important stats.

4.1.2 Result

Firstly we measured performance changes with respect to changing TCP window size, and saw that it obviously didn't have an effect on UDP speed, as we expected. However, as TCP window size increases, interestingly enough we recorded a decrease in parallel connection mode speed, an increase in average sender jitter, and no effect on packet loss.

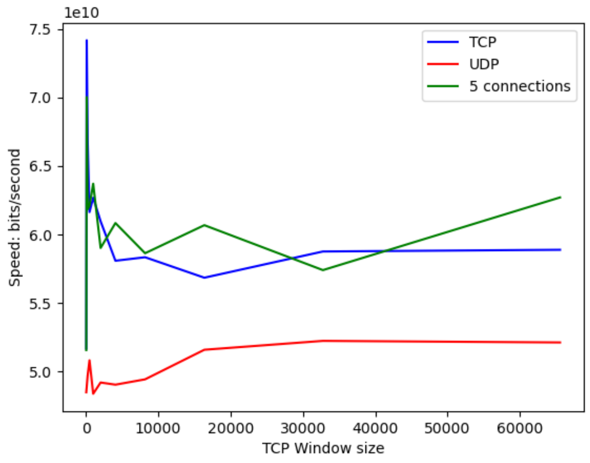

Figure 9: Effect of window size on performance

We then proceeded to measure the performance changes when varying the payload length, and surprisingly it had a much stronger effect on TCP than UDP when it comes down to speed. Increasing payload length also increased average jitter, again had no effect on packet loss.

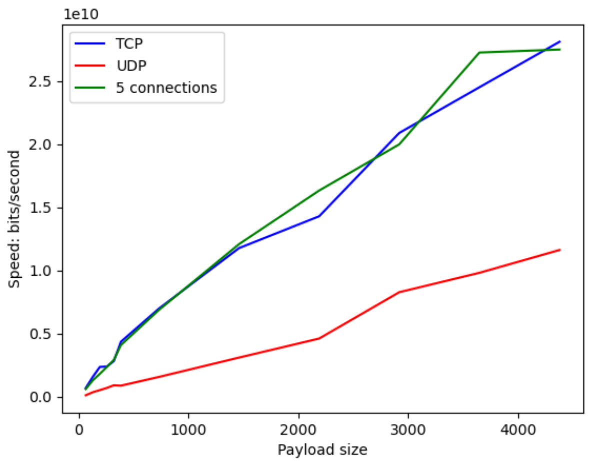

Figure 10: Effect of payload size on performance

Afterwards we measured the difference on performance when individually enabling zerocopy, reverse and bidirectional communincation in comparison to the default states. Enabling bidirectional communication actually reduced the speed, same as reverse mode. Using zerocopy actually increased the speed by a factor of about 1/7.

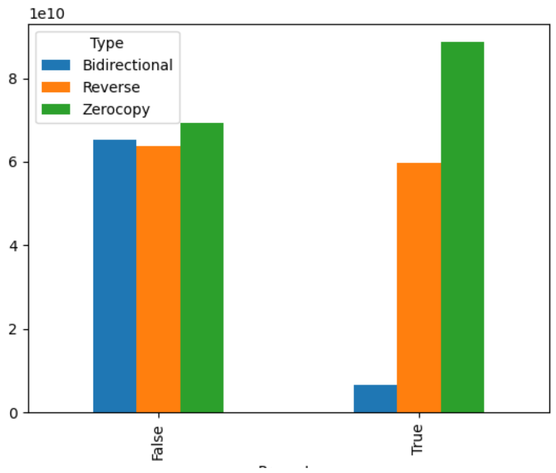

Figure 11: Effect of different connection modes on performance

At last we measured the effect on performance which the number of parallel connections has. We measured TCP and UDP performance on 1-100 connections, and benchmarked the results for every number of connections in that range. We found out that usually, the more active connections there are, the higher the jitter gets, packet loss wasn't meaningful, and interestingly enough on most cases the performance gap in speed between TCP and UDP gets bigger, with TCP being faster.

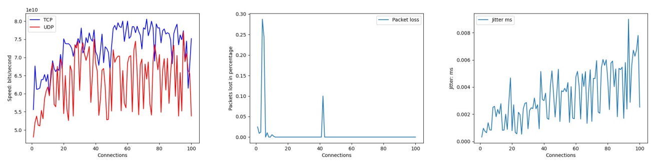

Figure 12: Effect of parallelism on performance

4.2 Explore task: develop a network performance analysis tool

In the exploration task we were tasked with building our own network performance measurement tool, a "dummy Iperf" basically, using traditional socket programming. The tool reports server client bandwidth, packet loss and jitter, same as Iperf.

4.2.1 Setup and Methodology

Our tool was developed in Python, and uses a couple of command line arguments to achieve customizability. We support parallel connection mode, test duration, payload length, reverse communication mode, use zerocopy, limit max MSS, set TCP window size, switching between string and JSON output, and sending output to stdout vs a log file. As it turns out setting the socket maximum MSS doesn't work, and hasn't worked in Iperf as well, judging by an open issue on their Github page [17] and on the freebsd forums [16].

4.2.2 Result

We decided to examine the TCP speed changes when we vary the number of simultaneous connections, and the results can be seen in the following graph. In the graph it is evident that initially the server and client speeds are about equal (with exactly one connections, thus no parallelism), then the client speed increases considerately faster than the server's, and then until around 10 connections both of them increase, probably due to server oversaturation.

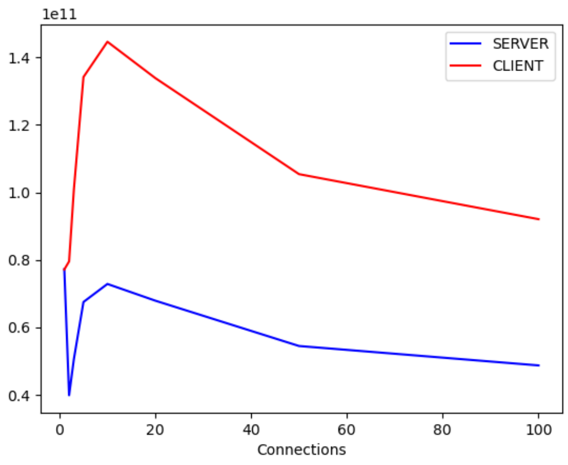

Figure 13: Effect of number of parallel connections on speed, measured in bits/second

5 Multicore benchmark

5.1 Description

Multicore architectures are ubiquitous nowadays. Parallel programming has become a necessity for a better performance. `Phoenix` and `PARSEC` frameworks which are originally developed at Stanford and Princeton universities are two software suits to evaluate emerging multicore processors. They cover different workloads in areas like finance, data-analytics, simulations, etc. And our target is to compare the software performance with different configurations.

5.2 Setup and Methodology

In this paper we use `speedup` as performance evaluation metrics. It is the gain in speed made by parallel execution compared to sequential execution [19]:

$$Sp = \frac{T_1}{T_p}$$

We benchmark the programs from two aspects by tuning the number of CPU cores used and the size of the data set.

5.3 Basic task: Phoenix

Phoenix is a shared-memory implementation of Google's MapReduce model for data-intensive processing tasks. It can be used to program multi-core chips as well as shared-memory multiprocessors (SMPs and ccNUMAs). [20]
The initial version of Phoenix was released in the April of 2007. `Phoenix 2` has been significantly updated to improve scalability and portability. `Phoenix++` is a C++ reimplementation of `Phoenix 2` which can reach speedup of

about 4.7x over `Phoenix 2`. [22] So here we benchmark with `Phoenix++`.

5.3.1 Applications and Workload

The following table shows the the applications and their corresponding dataset sizes used in this study.

Programs	Description	Data Sets
Word Count	Determine frequency of words in a file	S:10MB, M:50MB, L:100MB
Matrix Multiply	Dense integer matrix multiplication	S:100x100, M:800x800, L:1000x1000
Kmeans	Iterative clustering algo--rithm to classify 3D data points into groups	S:10K, M:50K, L:100K
String Match	Search file with keys for an encrypted word	S:50MB, M:100MB, L:500MB
PCA	Principal components analysis on a matrix	S:300x300, M:500x500, L:1000x1000
Histogram	Determine frequency of each RGB component in a set of images	S:100MB, M:400MB, L:1.4GB
Linear Regression	Compute the best fit line for a set of points	S:50M, M:100M, L:500M

The application `Reverse Index` is ignored as its phoenix++ version is not implemented.

5.3.2 Evaluation

The result plots of the benchmark is as follows and can be found at the directory `./demo` in the code repository. The text result is also included and the file name is of the pattern `result_t*_d*` where `t*` means how many threads are used and `d*` indicates the used data sets. The sequential execution result is located in the file with the name `result_seq_d*`.

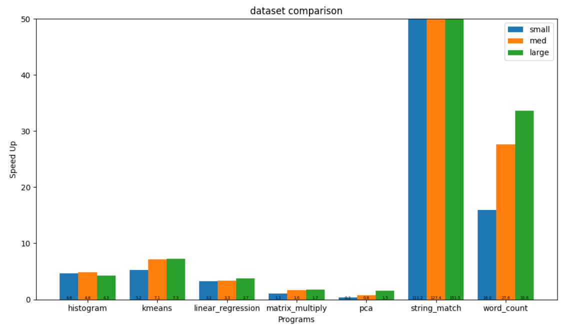

Figure 14: Speedup with 8 cores as we vary the dataset size

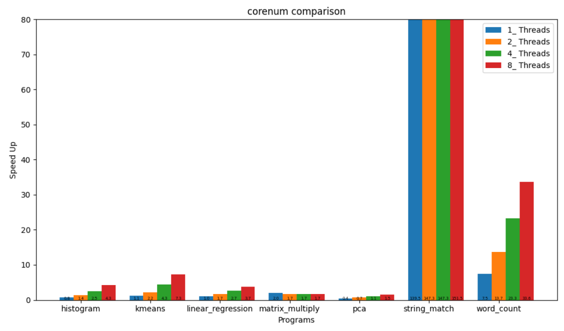

Figure 15: Speedup for the large datasets as we scale the number of processors

The analysis of the benchmark result will be in the conclusion section, and compared with Parsec.

5.4 Exploration task: Parsec benchmark

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [23] is a benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors. Here, as we mainly focus on the multithreading performance, we will not compare different memory protection methods(Intel MPX, AddressSanitizer, SoftBound and SAFECode) described in the project [25].

5.4.1 Applications and Workload

PARSEC 3.0 includes 13 workloads from different application domains and supports multiple parallelization models(Pthreads, OpenMP, Intel TBB). We select 7 of them, namely blackscholes, bodytrack, ferret, Fluidanimate, freqmine, swaptions and vips. When compiling with `gcc` configuration, the parallel model is automatically selected. All tasks use the pthread model except for freqmine, which uses openmp. [24]

Programs	Descriptions	Domain
Blackscholes	Calculate the prices analytically with the Black-Scholes PDE	Finance
Bodytrack	Track a human body with multiple cameras through an image sequence	Computer Vision
Ferret	Content-based similarity search	Search Engine
Fluidanimate	Simulate an incomp- -ressible fluid for interactive animation purposes	Physics Simulations
Freqmine	Array-based version of the FP-growth method for Frequent Itemset Mining	Data Mining
Swaptions	Price a portfolio of swaptions with Monte Carlo simulation to compute the prices	Finance
Vips	VASARI Image Processing System for fundamental image operations	Image

5.4.2 Evaluation

As in the basic task of phoenix++ benchmark, the result plots of the parsec benchmark can be found at the directory `./demo` in the code repository as well. The text result is also included and the file name is of the pattern `result_t*_d*` where `t*` means how many threads are used and `d*` indicates the used data sets. The sequential execution result is located in the file with the name `result_seq_d*`.

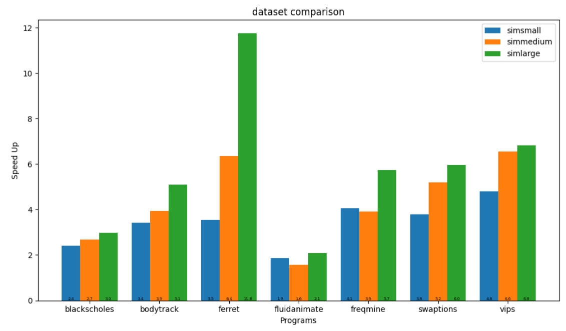

Figure 16: Speedup with 8 cores as we vary the dataset size

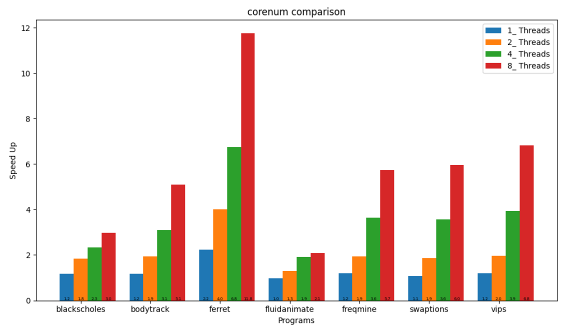

Figure 17: Speedup for the large datasets as we scale the number of processors

5.5 Conclusion

With different applications and settings, we can conclude that in most cases with increasing number of cores the performance of the application is also growing. It is also interesting to find that the larger the dataset, the higer performance improvement of the multi-core applications.

In the phoenix benchmark framework, the key-based structure that MapReduce uses fits well the algorithm of WordCount, StringMatch. (Note: The `Phoenix++` version of StringMatch is quite fast compared to its sequential baseline. So the structure of plots is affected.) Hence, these applications achieve significant speedups across all system sizes which is also observed by the researcher from the Computer Systems Laboratory at Stanford University [21]. On the other hand, the key-based approach is not the natural choice for PCA. It has even shown an apparent overhead caused by the introducing of MapReduce structure. But all of them benefit from the increasing CPU resources. In benchmark Parsec, the speedup performances are quite similar. Most of them adhere to the work law $T_1/T_p \leq p$ and doesn't exhibit a superlinear speedup.

Besides, it is also clear that increasing the dataset leads to higher speedups over the sequential execution for most applications. This is due to two reasons. First, a larger dataset allows the library runtime like libphoenix to better amortize its overheads for task management, buffer allocation, data spliting and sorting. Such overheads are not dominant if the application is truly data intensive. [21] Second, caching effects are more significant when processing large datasets and load imbalance is more rare. It is also the case for benchmark Parsec.

References

[1] https://github.com/akopytov/sysbench

[2] https://www.percona.com/

[3] http://tpc.org/tpcc/detail5.asp

[4] https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/

[5] https://github.com/brianfrankcooper/YCSB

[6] https://github.com/facebook/rocksdb/

[7] https://memcached.org/

[8] https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads

[9] http://www.cs.cmu.edu/~wtantisi/files/tablebenchmark-pdl11-talk.pdf

[10] https://anthonyaje.github.io/file/An_empirical_evaluation_of_Memcached_Redis_and_Aerospike_kvstore_Anthony_Eswar.pdf

[11] Douglas Comer (1979) *T*he ubiquitous B-tree, Douglas Comer, ACM Computing Surveys.

[12] https://btrfs.wiki.kernel.org/index.php/Btrfs_design

[13] https://www.phoronix-test-suite.com/

[14] https://openbenchmarking.org/test/pts/fs-mark

[15] https://fio.readthedocs.io/en/latest/fio_doc.html

[16] https://bugs.freebsd.org/bugzilla/show$_bug.cgi?id = 144000https://github.com/esnet/iperf/issues/$779

[17] https://openbenchmarking.org/result/2112273-TJ-TEAMAEXT489

[18] https://openbenchmarking.org/result/2112271-TJ-FSMARKBTR36

[19] https://en.wikipedia.org/wiki/Analysis_of_parallel_algorithms

[20] https://github.com/kozyraki/phoenix

[21] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007, pp. 13-24, doi: 10.1109/HPCA.2007.346181.

[22] Talbot, J., Yoo, R.M., Kozyrakis, C. (2011). Phoenix++: modular MapReduce for shared-memory systems. MapReduce '11.

[23] https://parsec.cs.princeton.edu

[24] http://wiki.cs.princeton.edu/index.php/PARSEC

[25] https://intel-mpx.github.io/performance/