# CS553 Homework #4

## Benchmarking Memory & Storage

***Instructions:***

- *Assigned date: Thursday March 7th, 2024*
- *Due date: 11:59PM on Friday March 22nd, 2024*
- *Maximum Points: 100% (20% extra credit available)*
- *This homework must be done individually*
- *Please post your questions to BB*
- *Only a softcopy submission is required*
- *Late submission will be penalized at 20% per day*

### 1 Your Assignment

This project aims to teach you about file I/O and benchmarking. You can use any of the following languages to implement this assignment: Python, Java, C, C++, Rust; doing your implementation in C, C++, or Rust will get you up to 10% extra credit points for efficient implementations. You can use any libraries (e.g. STL, Boost, PThread, OMP, BLAKE3, etc). You must use Linux system for your development. The performance evaluation should be done on Chameleon in a Linux environment.

You will make use of a hashing algorithm:

- Blake3:
  - Repo: https://github.com/BLAKE3-team/BLAKE3
  - Paper: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf

Your benchmark will use a 6-byte NONCE to generate 2^26 (SMALL) or 2^32 (LARGE) BLAKE3 hashes of 10-bytes long each and store them in a file on disk in sorted order (sorted by 12-byte hash). A record could be defined in C as follows:

```
#define NONCE_SIZE 6
#define HASH_SIZE 10

// Structure to hold a 16-byte record
typedef struct {
     uint8_t hash[HASH_SIZE]; // hash value as byte array
     uint8_t nonce[NONCE_SIZE]; // Nonce value as byte array
} Record;
```

Your file should be 1GB (SMALL) or 64GB (LARGE) in size when your benchmark completes (64GB = 2^32*(12B+4B)) – your file should be written in binary to ensure efficient write and read to this data; do not write the data in ASCII. You are to parallelize your various stages (hash generation, sort, and disk write) with a pool of threads per stage, that can be controlled separately.

You will have several command line arguments that you will explore from a performance point of view for the SMALL 1GB workload. There are 27=1x3x3x3 experiments to run, so make sure to automate the execution of these runs in a bash script (e.g. 3 nested loops in bash should work).

1. Maximum memory allowed to use (MB): 128
2. Number of hash threads: 1, 4, 16
3. Number of sort threads: 1, 4, 16
4. Number of write threads: 1, 4, 16

Each experiment should take about a minute or less, depending on the hardware, processor type, core counts, and hard drive technology. These experiments should take less than an hour to run in all.

Plot the results of these 27 experiments and identify the best combination of command line arguments. Explain why you believe the best and worst configurations make sense.

Here are the command line arguments your program should have:

```
iraicu@athena:/hw4$ ./hashgen -h
Help:
  -f <filename>: Specify the filename
  -p <num_records>: Specify the number of records to print from head
  -r <num_records>: Specify the number of records to print from tail
  -d <bool>: turns on debug mode with true, off with false
  -v <bool>: verify hashes sort order from file, off with false, on with true
  -b <num_records>: verify hashes as correct BLAKE3 hashes
  -h: Display this help message
```

Here is a sample invocation:

```
iraicu@athena:/hw4$ ./hashgen -t 16 -o 1 -i 1 -f data.bin -m 16 -s 1024 -d true
NUM_THREADS_HASH=16
NUM_THREADS_SORT=1
NUM_THREADS_WRITE=1
FILENAME=data.bin
MEMORY_SIZE=16MB
FILESIZE=1024MB
RECORD_SIZE=16B
HASH_SIZE=10B
NONCE_SIZE=6B
[1][HASHGEN]: 19.06% completed, ETA 4.3 seconds, 12232/65536 flushes, 195.2 MB/sec
[2][HASHGEN]: 38.00% completed, ETA 3.3 seconds, 24546/65536 flushes, 193.9 MB/sec
[3][HASHGEN]: 57.12% completed, ETA 2.3 seconds, 36865/65536 flushes, 195.8 MB/sec
[4][HASHGEN]: 76.45% completed, ETA 1.2 seconds, 49452/65536 flushes, 197.9 MB/sec
[5][HASHGEN]: 95.32% completed, ETA 0.2 seconds, 61853/65536 flushes, 193.2 MB/sec
sort started, expecting 64 flushes for 1024 buckets...
[6][SORT]: 2.64% completed, ETA 38.2 seconds, 27/1024 flushes, 26.1 MB/sec
[7][SORT]: 5.37% completed, ETA 36.4 seconds, 55/1024 flushes, 27.1 MB/sec
…
```

```
[41][SORT]: 94.82% completed, ETA 2.0 seconds, 971/1024 flushes, 27.0 MB/sec
[42][SORT]: 97.56% completed, ETA 0.9 seconds, 999/1024 flushes, 27.2 MB/sec
Completed 1024 MB file data.bin in 43.22 seconds : 1.55 MH/s 23.69 MB/s
```

The same invocation with debug off would produce the following output:

```
iraicu@athena:/hw4$ ./hashgen -t 16 -o 1 -i 1 -f data.bin -m 16 -s 1024 -d false
hashgen t16 o1 i1 m16 s1024 43.22 1.55 23.69
```

A tool to help you verify that the data you are writing is correct can be found in hashverify.

```
iraicu@athena:/hw4$ ./hashverify -h
Help:
  -f <filename>: Specify the filename
  -p <num_records>: Specify the number of records to print from head
  -r <num_records>: Specify the number of records to print from tail
  -v <bool> verify hashes from file, off with false, on with true
  -h, --help: Display this help message
iraicu@athena:/vault$ ./hashverify -d true -f data.bin -v true -d true
DEBUG=true
FILENAME=data.bin
VERIFY_RECORDS=true
RECORD_SIZE=16B
HASH_SIZE=10B
NONCE_SIZE=6B
[1][VERIFY]: 70.40% completed, ETA 0.4 seconds, 704 read, 790.0 MB/sec
[1.335][VERIFY]: 100.00% completed, ETA 0.0 seconds, 1024 read, 767.1 MB/sec
Read 1073741824 bytes and found all records are sorted.
```

Similarly you can turn off debug logs:

```
iraicu@athena:/vault$ ./hashverify -d false -f data.bin -v true
Read 1073741824 bytes and found all records are sorted.
```

A tool to help you print your data to the screen is shown below:

```
iraicu@athena:/hw4$ ./hashverify -f data.bin -p 8
Printing first 8 of file data.bin'...
[0] Hash: 000000744acc5940a01b : d92c93010000 : 26422489
[16] Hash: 0000009fcad95785e04e : a40913010000 : 18024868
[32] Hash: 000001012778f96f23bd : 4deb68030000 : 57207629
[48] Hash: 000001bb43255f863484 : 077914000000 : 1341703
[64] Hash: 0000023640c2b8cc714d : eb4df6030000 : 66473451
[80] Hash: 000002475cea698b6761 : 66d84d020000 : 38656102
[96] Hash: 0000030bc90ed2b07331 : 987112000000 : 1208728
[112] Hash: 0000033687c51d72ee2d : 0f903f000000 : 4165647
```

Once you have everything done, you should double check that the hashes you are generating are valid BLAKE3 hashes. You can do that with the following hashverify command:

```
iraicu@athena:/hw4$ ./hashverify -f data.bin -b 10
verifying random records against BLAKE3 hashes
Number of total verifications: 10
Number of verifications successful: 10
Number of verifications failed: 0
Time taken: 0.05 ms/verification
Throughput verifications/sec: 21691.97
```

You will have several command line arguments that you will explore from a performance point of view for the LARGE 64GB workload. You have 27 experiments in total, where each one will take on average under 1 hour to complete (even with a HDD). These runs should take less than a day.

1. Maximum memory allowed to use (MB): 1024
2. Number of hash threads: 1, 4, 16
3. Number of sort threads: 1, 4, 16
4. Number of write threads: 1, 4, 16

Some sample commands:

With HDD storage:

```
iraicu@athena:/hw4$ ./hashgen -t 16 -o 1 -i 1 -f data.bin -m 1024 -s 65536 -d false
hashgen t16 o1 i1 m1024 s65536 4037.50 1.06 16.23
```

With a NVMe storage:

```
iraicu@MacBook-Pro /hw4$ ./hashgen -t 16 -o 1 -i 1 -f data.bin -m 1024 -s 65536 -d
false
hashgen t16 o1 i1 m1024 s65536 550.15 7.81 119.12
```

Once you have identified the best combination of thread counts for 1024MB memory case, let's try different amount of memory for a LARGE 64GB workload: 1024, 8192, 32768

You are to create a makefile (or CMake, ANT, Maven, etc) that helps build your benchmark, as well as run it through the benchmark bash script. You will be given a binary tester that will verify if your file is indeed sorted.

Other requirements:

- You must write all benchmarks from scratch. Do not use code you find online, as you will get 0 credit for this assignment. This is also an individual assignment, make sure you do this assignment by yourself.
- All of the benchmarks will have to evaluate concurrency performance; concurrency can be achieved using processes or threads.
- Not all timing functions have the same accuracy; you must find one that has at least 1ms accuracy or better, assuming you are running the benchmarks for at least seconds at a time.
- Since there are many experiments to run, you must use a bash script to automate the performance evaluation.
- You must use binary data when writing in this benchmark.
- No GUIs are required. Simple command line interfaces are required.

## 2 Extra Credit

**C/C++/Rust (up to 10%):** If you decide to implement your assignment in C, C++, or Rust will get you up to 10% extra credit points depending on the efficiency of your implementation.

**Leaderboard (up to 10%):** If you will publish your best results of the 64GB file with 1GB of memory, you will get up to 10% extra credit points depending on the ranking in the leaderboard. The #1 spot on March 22nd at 11:59pm will receive 10% extra credit (with proper verification). The #2 spot will receive 9%. #3 will get 8%, and so on. The top 9 spots will receive between 2% and 10% extra credit. Everyone else will receive 1% extra credit for submitting. The submission must be for 64GB file with 1GB RAM, and the file must pass the hashverify test. Leaderboard submission form:

https://docs.google.com/forms/d/e/1FAIpQLSfWsZ9NWWUwdm_KmfCAxCHAa9Bhcez08__zTEKhC2QxG3Ulkw/viewform

## 3 What you will submit

When you have finished implementing the complete assignment as described above, you should submit your solution to your private git repository. Each program must work correctly and be detailed in-line documented. You should hand in:

1. **Source code and compilation (60%):** All of the source code, as well as bash scripts; in order to get full credit for the source code, your code must have in-line documents, must compile (with a build file), and must be able to run a variety of benchmarks through command line arguments as outlined.
2. **Report / Performance (40%):** A separate (typed) design document (named hw4-report.pdf) describing the results in a table format. You must summarize your findings, in terms of scalability, concurrency, and performance. For your particular setup (everyone will be different since you are running on different hardware), what seems to be the best configuration and worst configuration for each workload, in terms of concurrency level and record size? Highlight the best one in green, and worst one in red for the SMALL workload, and the LARGE workload. Can you explain in words why they are the best, or the worst, and why the data you have makes sense.

You will have to submit your solution to a private git repository created for you on GitHub. The repository is created through GitHub Classroom and you will need to accept the assignment before you can clone it, by accessing this invitation link: https://classroom.github.com/a/C5s9grq-. The first time you access an invitation link the system will ask you to identify yourself. Please select the identifier that contains your school email address. After that you can clone the repository. Then you will have to add or update your source code, documentation, and report. If you cannot access your repository, contact the TAs. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/

**Grades for late programs will be lowered 20% per day late.**