Final Project

CS-GY 6233

Vineet Bhardwaj

VB2182@NYU.EDU

Tests have been performed on SSD. Used Ubuntu VM via Oracle VM VirtualBox.

Grant the execute permission for 'build' and 'run' files to the user.

1. Navigate to the code subfolder with the "**build**" file
2. Give execute permission to it and to the "**run"** file by typing the following

chmod u+x build
chmod u+x run

Q1:
Instructions for code:

(a) In the code subfolder, run the build file to compile everything in the root of the repo.

(b) Then test the code.c file by either running it or by using the run utility as follows:
./run <filename> [-r|-w] <block_size> <block_count>

Write: 50 bytes per block X 5 million blocks written at 9.25 MiB/s

```
abc@abc-VirtualBox:~/Desktop/Test$ ./run vineet1.txt -w 50 5000000

File Created:
        File name    : vineet1.txt
        # of blocks : 5000000
        Blocks size : 50 Bytes
        Total size   : 250000000 B (238.42 MB)
        Write time   : 25.77 seconds
        Write speed : 9701280.50 B/s (9.25 MiB/s)
```

Write: 5 million bytes per block X 50 blocks written at 276.66 MiB/s

```
abc@abc-VirtualBox:~/Desktop/Test$ ./run vineet2.txt -w 5000000 50

File Created:
        File name    : vineet2.txt
        # of blocks : 50
        Blocks size : 5000000 Bytes
        Total size   : 250000000 B (238.42 MB)
        Write time   : 0.86 seconds
        Write speed : 290098587.10 B/s (276.66 MiB/s)
```

Clearly, bigger block writing is quicker.

Read mode: In the read mode, it will read a file with the specified name and provided block size and number of blocks. It returns XOR of all 4-byte integers in the file — treating the file as an array of integers.

Read: Upon reading the ubuntu-21.04-desktop-amd64.iso (2.8 GB) file with 512-byte block at a time for 10 blocks, it yields a performance of 366.91 MiB/s. XOR is a7eeb2d9.

```
abc@abc-VirtualBox:~/Desktop/Test$ ./run ubuntu-21.04-desktop-amd64.iso -r 512 10
Read through-put with blocks-size 512 was: 384736508.95 B/s (366.91 MiB/s)
The XOR of all 4-byte integers in the file was a7eeb2d9
```

Reading a 100 times bigger block than before, yields a performance of about 1.76 times as before. XOR value of the same file stays the same, irrespective of the block size.

```
abc@abc-VirtualBox:~/Desktop/Test$ ./run ubuntu-21.04-desktop-amd64.iso -r 51200 10
Read through-put with blocks-size 51200 was: 680256367.77 B/s (648.74 MiB/s)
The XOR of all 4-byte integers in the file was a7eeb2d9
```

Conclusion: Bigger block reading /writing are quicker which follows intuitively also.

Q2:

The program starts by creating 100 MB file and checking the read time using given block size. The file size is doubled at each iteration until reasonable time is achieved. Reasonable time is defined to be between 5s and 15s. Read of a file is as specified above i.e. read one block at a time with user defined block size and create an integer array by performing a 4-byte XOR operation.

Reasonable time is computed by running
./reasonable <BLOCK_SIZE>

Three consecutive runs of the script with 5-byte, 100-byte, and 1000-byte block sizes as inputs, yield 0.1 GB, 0.78 GB, and 1.56 GB as sizes of the files that can be read in reasonable times of 5.67 seconds, 5.64 seconds, and 8.60 seconds respectively. Different XORs are because of generation of different temp files for this program.

```
abc@abc-VirtualBox:~/Desktop/Test$ ./reasonable 5

Success !!!
Reasonable time of 5.675080 seconds achieved with
        file size of 104857600 Bytes (0.10 GB)


The XOR of all 4-byte integers in the file was fec90bc1

abc@abc-VirtualBox:~/Desktop/Test$ ./reasonable 100

Success !!!
Reasonable time of 5.636964 seconds achieved with
        file size of 838860800 Bytes (0.78 GB)


The XOR of all 4-byte integers in the file was 27a3e4a6

abc@abc-VirtualBox:~/Desktop/Test$ ./reasonable 1000

Success !!!
Reasonable time of 8.603624 seconds achieved with
        file size of 1677721600 Bytes (1.56 GB)


The XOR of all 4-byte integers in the file was 15f91361
```

Note: I didn't put **"&& runtime <=15"** in line 86 of reasonable.c, because I wanted to accommodate exceptional cases.

Extra Credit:

dd:

dd is a command-line utility for Unix and Unix-like Operating Systems where the primary purpose is to copy and convert a file during the process. We can actually make use of this Linux command by opening a sub-process and running *dd* on that sub-process shell to create the file.

Compared it with my code run in write mode vs dd in write as follows:

File Size	DD shell time (seconds)	C-program time (seconds)
100 MB	0.539	0.843
500 MB	2.821	2.817
1 GB	5.409	5.263
2 GB	10.597	10.662
5 GB	27.140	26.166
10 GB	54.531	52.470

Conclusion: dd is generally faster for smaller file size, but my code is faster for larger file size in the data sample above.

Google Benchmark:

Install google benchmark library to be able to use it. Need to ensure that appropriate version of cmake / gcc / g++ is installed.

Follow the steps below for configuration:

(a) Install Dependencies:
   (i) sudo apt install gcc g++ cmake
   (ii) sudo apt install git

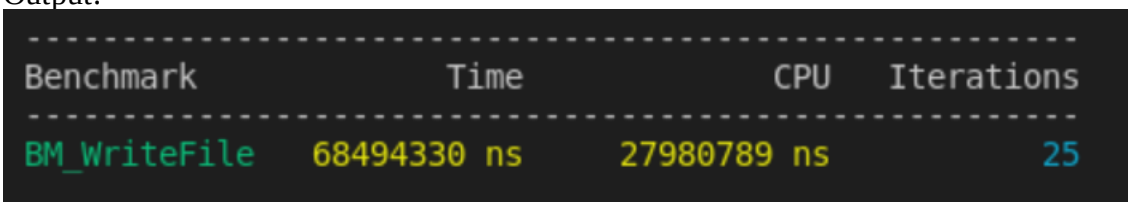
(b) Clone Google benchmark and build it:
   (iii) git clone https://github.com/google/benchmark.git && cd benchmark
   (iv) cmake -E make_directory "build"
   (v) cmake -E chdir "build" cmake -DBENCHMARK_DOWNLOAD_DEPENDENCIES=on -DCMAKE_BUILD_TYPE=Release ../
   (vi) cmake --build "build" --config Release

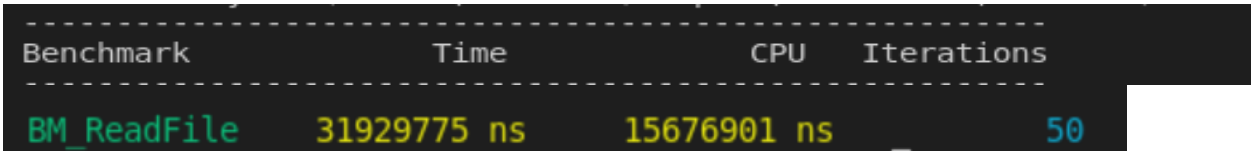
(c) Test the Installation:
   (vii) cmake -E chdir "build" ctest --build-config Release

(d) Install the library globally:
   (viii) sudo cmake --build "build" --config Release --target install

(e) To run the benchmark file compile and run the code using:
   (ix) Move the mybenchmark.cc file to the benchmark folder.
   (x) g++ mybenchmark.cc -std=c++11 -isystem benchmark/include -Lbenchmark/build/src -lbenchmark -lpthread -o mybenchmark
   Then run ./mybenchmark

Output:





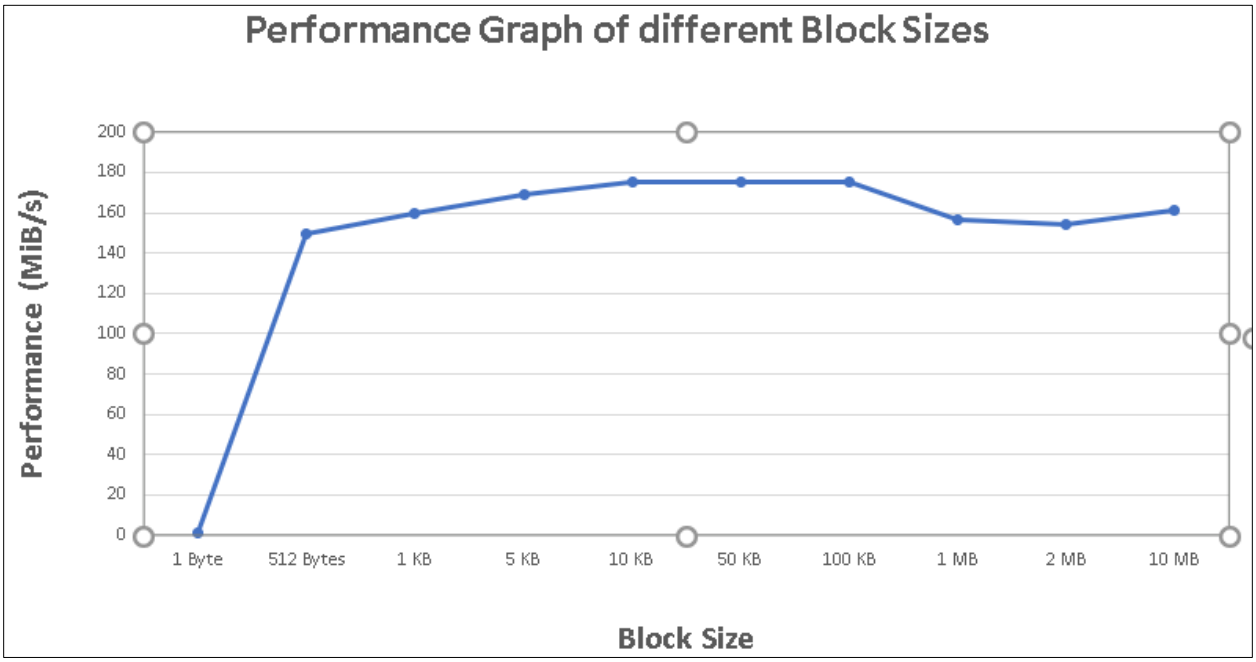
Benchmark, explanation:

The idea is to create functions that need to be benchmarked and then run them via the benchmark templates. Read and write Benchmark functions are created in lines 100 and 114 of the mybenchmark.cc file, and are then called. I registered the benchmark functions by using BENCHMARK () and passing the function names to the calls.
Finally, I ran the benchmark by adding BECHMARK_MAIN (), which handles everything else.

Q3:

Code.c file is configured to output the performance in B/s and MiB/s

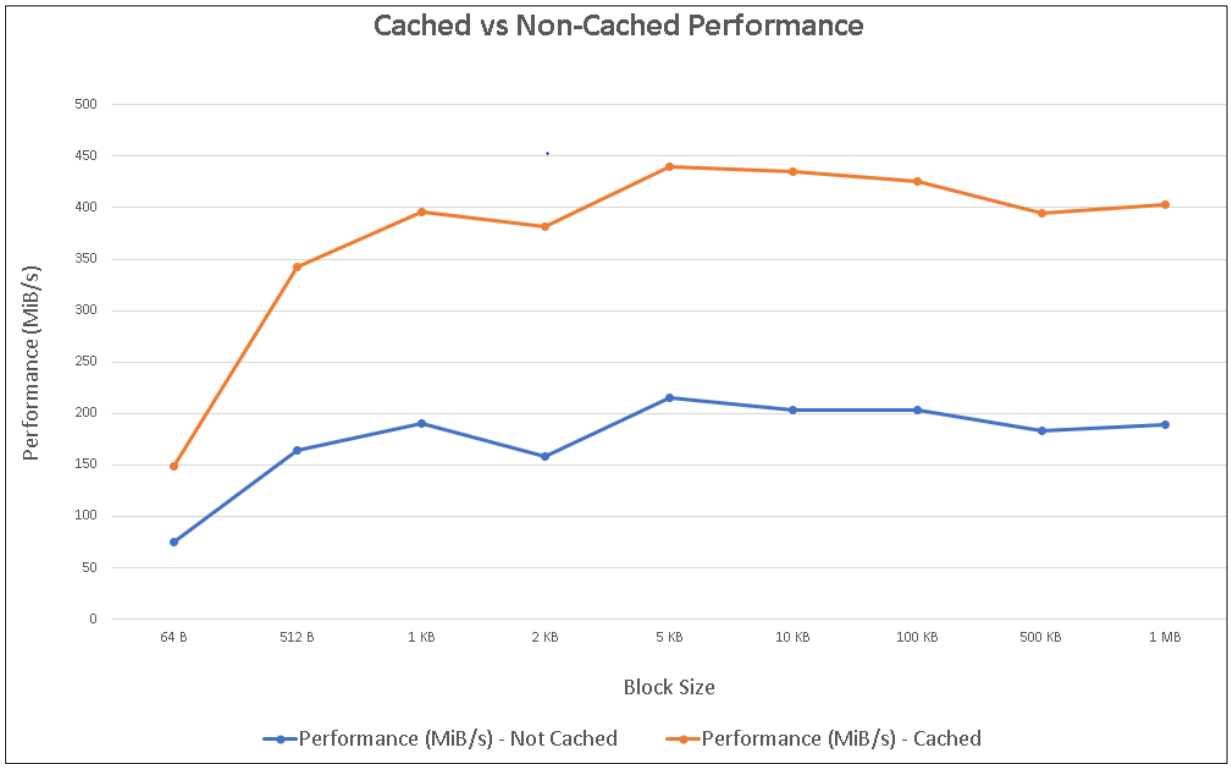
Graphs and data are in the Excel file “Data and Graphs.xlsx”



Steps followed:
a. Run Reasonable.c with some arbitrary block size.
b. It will display the optimal file size. Note that down.
c. Run the code.c file with -w flag and provide it the block size and # of blocks.
Computation can be done using the formula: filesize = block-size * # of blocks.
d. Now run code.c file multiple times with -r flag with some block sizes below the optimal one and some above it to test.

Q4:

Followed the instructions on the project page to successfully observe the effect of caching. Experimented with clearing the caches. Graph for cached vs non-cached entries:



Extra credit:

The Linux filesystem cache is used to make I/O operations faster. Linux provides a way to flush or clear the RAM caches in case any program is consuming the available memory. In order to clear the caches, I need to pass a flag to /proc/sys/vm/drop_caches file. There are 3 different options available:

echo 1 > /proc/sys/vm/drop_caches -> Clear PageCache only.
echo 2 > /proc/sys/vm/drop_caches -> Clear dentries and inodes.
echo 3 > /proc/sys/vm/drop_caches -> Clear pagecache, dentries, and inodes.

I used the third option to clear all pagecache, dentries and inodes because I wanted to get rid of all disk cachces.

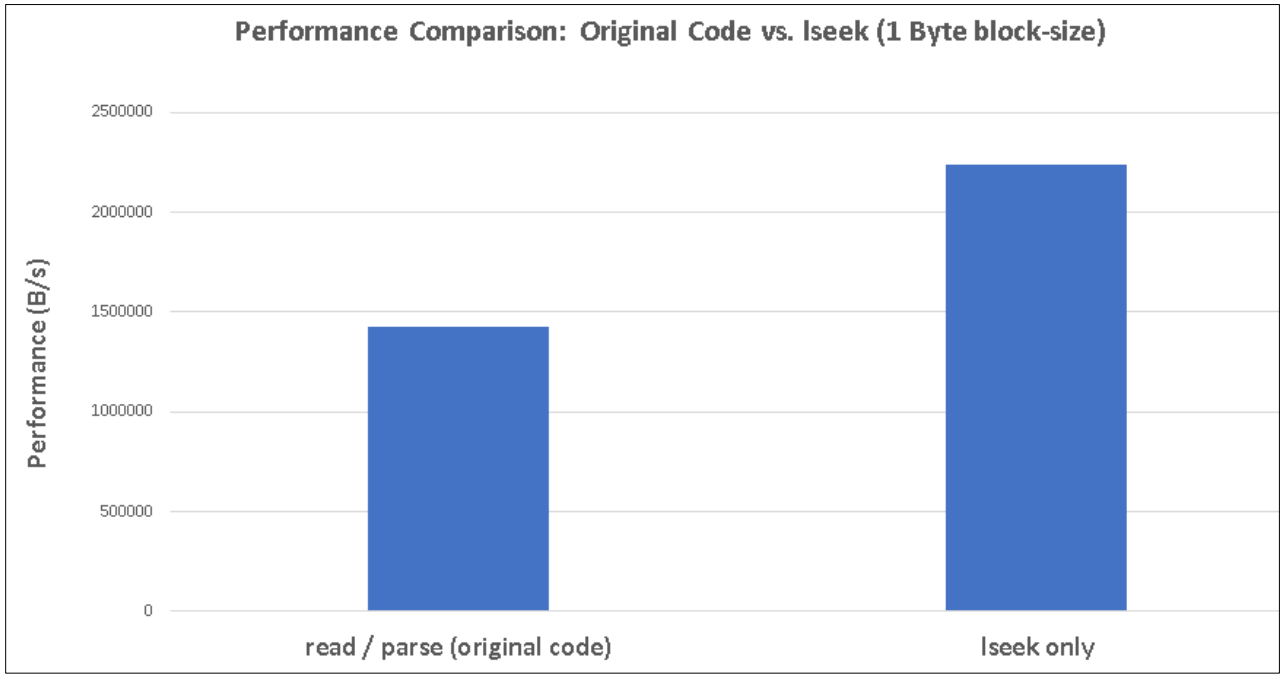
Q5:

After compiling, run task5.c as follows:

./task5 [FILE_NAME] <BLOCK_SIZE>

This uses the read part of code.c file and makes an implementation to use less intensive system calls like lseek. This file performs read operation so the file to be used should already be present on the disk.
I used the iso file that Professor mentioned for this.

a. Run code.c in read mode.
b. Run task5.c for the same and notice the difference in performance.
c. Lseek is a single operation used to set / change current pointer.



Q6:
Once compiled run the file “code.c” or “run” in write mode using either of the below:
./code [FILE_NAME] -w <BLOCK_SIZE> <BLOCK_COUNT>
./run [FILE_NAME] -w <BLOCK_SIZE> <BLOCK_COUNT>

Once sufficiently large file is created run the "fast" program to read that file in parallel using:

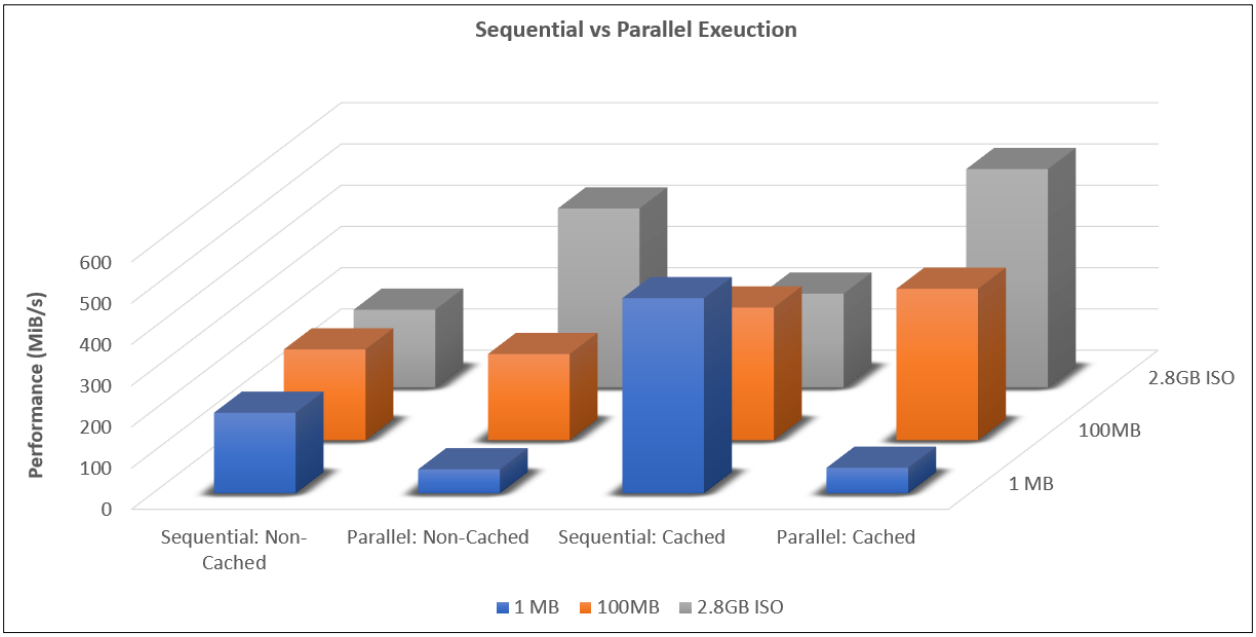
./fast [FILE_NAME]

Here, I am using the ISO (2.8GB), a 1 MB file and a 100MB file for comparisons.

This code uses the read part of code.c file and makes a parallel implementation using pthreads library. This file is configured to perform read operation so make sure the file to be read is already present.

One of the fastest performances achieved for my **fast** program for the ISO file is 528.935MiB/s.

```
abc@abc-VirtualBox:~/Desktop/Test$ ./fast ubuntu-21.04-desktop-amd64.iso
Read through-put with blocks-size 352342272 was: 554629416.664158 B/s (528.935830 MiB/s)
The XOR of all 4-byte integers in the file was a7eeb2d9
```



The pattern observed for sequential is similar to the graph for Block size vs. Performance in Q3.
It improved as I went from lower block size to higher until it reached a peak and then became linear and eventually declined.

I noticed that threads throughput the higher outputs for larger block sizes but perform worse than sequential code for smaller blocks. This is because larger blocks give large workload to threads fully utilizing parallelization. Using smaller block sizes adds additional overheads of having to create threads frequently while their workload is quite small.

Optimal block size depends on many factors like storage medium, read / write speeds, cache sizes, I/O buffer sizes etc.

From my trials, I conclude that using larger block sizes, coupled with caching, and multiple threads yields optimal performance.