Contents

1. CPU Benchmarking

a. Design Process

1. FLOPS and IOPS computations:

32 Floating point operations (8 different parallelizable addition and assignment operations; 8 different parallelizable multiplication and assignment operation) were done in a loop.

The loop is repeated 1000*USHRT_MAX times after starting the watch. After stopping watch, we have number of operations done and time taken for that. So GFLOPS are calculated using following formula.

$$GFLOPS = \frac{Number\ of\ operations\ done\ (1000 * USHRT\_MAX * 35)}{Total\ time\ taken * 1{,}000{,}000{,}000}$$

In this experiment counter increment after loop and condition check is also considered and hence the number 35 (32+2+1). In case of multi-threaded test, total operations done by all the threads is considered for calculation. Average of the time taken by each thread is considered for calculation of GFLOPS. Time taken for performing $(1000 * USHRT\_MAX * 35)$ computation is stored in an array with array number as index. Here's the formula for GFLOPS calculation with multiple threads.

$$GFLOPS = \frac{4(N1 + N2 + N3 + N4)}{(T1 + T2 + T3 + T4) * 1{,}000{,}000{,}000}$$

Nx = Operations done by xth thread, Tx is time taken by xth thread. Nx is same for all x's in this case. Similar process is repeated for IOPS.

2. Sampling FLOPS & IOPS over an interval:

A timer thread is launched along with number of threads for doing FLOPS computations in a loop. Timer thread sleeps for a second and after waking up increments a global variable called second. A two dimensional array with Thread-Id as one dimension and second as another is created before launching the experiment. Computing block perform 350 operations before incrementing number of operations performed. This value is stored in two dimensional array.

At the end of experiment, we will get an array for each thread with 600 samples each. We combine operations done by all threads and average the time taken to compute GFLOPs for that particular second.

$$\frac{Total\ operations\ done\ during\ sampling\ interval\ by\ all\ 4\ threads}{Average\ of\ exact\ sampling\ time\ (including\ the\ noise)\ for\ all\ 4\ threads}$$

b. Tradeoffs

- Approximations in case of sampling as the number of computations will be stored in the next second even though the calculation were done in last second. This is to avoid the overhead of storing and checking condition after every computation.
- A function is used for returning the current time which will add overhead of context switching.

c. Future Improvements

- Time can be calculated inline instead of asking a function to return it.
- Using inline assembly for the instructions to be executed. This will minimize the compiler influence on the performance of the code.

2. Memory Benchmarking

a. Design Process

Input is taken for the test case to be ran. Memory equal to 100 times the block size is allocated and initialized with random characters as source and destination blocks.

$$Total\ Size\ ALlocated = 100 * BlockSize$$

Block Sizes of 1B/KB/MB are considered for performing this experiment. Threads are created with a function in which the allocated and initialized memory is traversed for source block and copied to destination block a B/KB/MB at a time. This procedure is repeated 1000 times and thus total memory transferred per second is

$$Throughput = \frac{1000 * 100 * BlockSize}{total\ time\ taken}\ Bytes/sec$$

Similar loop is executed for setting memory with a char which for this experiment's sake is 'a'. Memset is performed over Total size allocated with B/KB/MB data in each operation. This procedure is repeated for 1000 times and thus latency is calculated as

$$Latency = \frac{total\ time\ taken}{1000 * 100}\ Seconds$$

Similarly, for random access, a random number between 1 to 100 is added every time to the starting position of the memory allocation and a block size of data is copied to that location.

b. Tradeoffs

- Memory allocation was only 100 times the block size to keep the time taken to run experiment shorter.
- Memory allocation is also constrained by total available memory in the system which can be compensated by having number of iterations and having new allocation in each iteration. But this puts overhead in time taken to allocate and free memory.

c. Future Improvements

- Minimizing cache usage so that the operations will be done completely using memory
- Having bigger allocation size
- Creating random number on the go for random access instead of asking a function for it

3. Disk Benchmarking

a. Design Process

C's file read/write APIs are used for benchmarking disk. The design of this benchmarking is somewhat similar to memory benchmarking. A file is created of size 10 MB and filled with random characters and then all the experiments are performed on this file.
In each experiment file is opened in read/write+ mode and block of data of size block-size (B/KB/MB) is read/write to this file block by block for 100 iterations. As fread/fwrite functions return the number of bytes read/written it can be used to increment the value of bytes transferred which can be used to calculated throughput. Therefore, total data read/written would be

$$Throughput = \frac{Total\ Bytes\ Read/Written}{total\ time\ taken}\ Bytes/sec$$

Similarly, for latency the time to read/write the BlockSize of data is measured and then latency can be calculated as:

$$Latency = Time\ taken\ to\ read/write\ a\ BlockSize\ of\ Data$$

BlockSize of B/KB/MB are read/written number of times and then average latency is calculated by dividing total time by number of iterations.

- Sequential Read/Write: Similar to memory approach a file pointer is moved by BlockSize after copying BlockSize of data. This process is repeated 100 times and then total bytes transferred are considered for calculating throughput
- Random Read/Write: File pointer is set to initial position after every read/write operation and a random offset is added to it before starting new fread/fwrite operation.

The process is followed by number of threads in multi-threaded application.

b. Tradeoffs made

- No provision to detect and avoid cache usage.
- Fixed file size
- Smaller file size (Time and disk constraints)

c. Possible Improvements

- Mechanism to avoid cache usage so that actual disk operation happens instead of reading it from memory. (A file of size > memory should be considered for avoiding cache hit)
- Low level APIs can be used to avoid the overhead for read/write/seek operations.