Report
Martin Rosenfeld Student Id: A20350531

1 Introduction

The following is the report of the experiments performed for benchmarking. The components chosen to benchmark were CPU, Memory and Disk.

I will start describing general considerations and design decisions made for all the benchmarks. Afterwards, for each of the benchmarking programs, I will show the results and other details.

2 General Considerations

The three benchmarks were developed in *C*, and compiled with *gcc*. To manage threads, I used *pthread*. Plots were generated using *gnuplot* library.

All the experiments were run in AWS t2.micro instances.

To perform the experiments, I ran a bash script which compiles (using a provided Makefile) and executes the benchmark program. The program saves the results in a csv file and also shows them interactively in the console. The bash script finally runs gnuplot, which creates the graphs loading the data from the csv file generated by the program.

The three programs allow the user to provide parameters for the number of iterations to perform each experiment, and for the time (in seconds) to run each iteration. If more than one iteration is specified, the results of all of them are averaged. The last parameter allows each experiment to run the same amount of time.

To measure time elapsed in the experiments and get the results, I analyzed many timing alternatives available in *C*. I finally decided to use *gettimeofday()* function, which ensures microsecond precision and uses wall clock time. This function is also used in other existing benchmarks, e.g., *Stream*.

3 Disk Benchmark

3.1 Decisions

The disk benchmark program creates a big file of 300MB to work with, in order to avoid the usage of cache which would distort the performance results.

For each experiment the main process creates the corresponding threads, and assigns each of them an equal part of the file to work with. In this way, collisions while performing the file operations are avoided. Once the specified time has passed, the main process joins the threads, and collects and aggregates the results.

I analyzed different file access alternatives and libraries. To get more realistic performance results, I decided not to use *stdio* library (fread, fseek and fwrite functions) because it

performs buffered operations. Alternatively, I used the read() and write() functions from *unistd*, which provides access to the POSIX operating system API.

To perform random access to the file, each time the programs does an operation it randomly chooses the block. The overhead of getting a random number is low enough to not degrade the performance and distort the results.

3.2 Results

I first show a table with the results of the 24 experiments. For each of them, the throughput and latency obtained are shown. To make the results more accurate, I ran each of the experiments 3 times, each of them for 10 seconds, and then averaged them.

Block Size(Bytes)	Threads	Operation	Strategy	Throughput (MB/sec)	Latency (ms)
1	1	Read	Seq	4.808732	0.000205
1	1	Read	Random	2.360730	0.000399
1	1	Write	Seq	1.645652	0.000623
1	1	Write	Random	0.040848	0.024494
1	2	Read	Seq	4.799993	0.000203
1	2	Read	Random	2.374215	0.000406
1	2	Write	Seq	1.648724	0.000587
1	2	Write	Random	0.040701	0.024160
1024	1	Read	Seq	3725.179918	0.000265
1024	1	Read	Random	1747.696741	0.000558
1024	1	Write	Seq	74.106658	0.013357
1024	1	Write	Random	38.454555	0.025506
1024	2	Read	Seq	3647.265978	0.000280
1024	2	Read	Random	1751.463945	0.000553
1024	2	Write	Seq	84.415552	0.011092
1024	2	Write	Random	40.196733	0.024519
1048576	1	Read	Seq	8008.557579	0.128452
1048576	1	Read	Random	8312.043066	0.120446
1048576	1	Write	Seq	77.130793	12.964969
1048576	1	Write	Random	146.082539	6.839305
1048576	2	Read	Seq	7704.034701	0.130219
1048576	2	Read	Random	7927.100666	0.125591
1048576	2	Write	Seq	92.237726	10.821826
1048576	2	Write	Random	144.532051	7.000564

As a general comment, it can be easily observed that the bigger the block size, the higher the throughput. Also, there is no considerable performance improvement when increasing the number of threads.

To analyze the results in more detail I will show a series of plots.

The figures 1 and 2 show the effect of the block size in the performance of the operations.

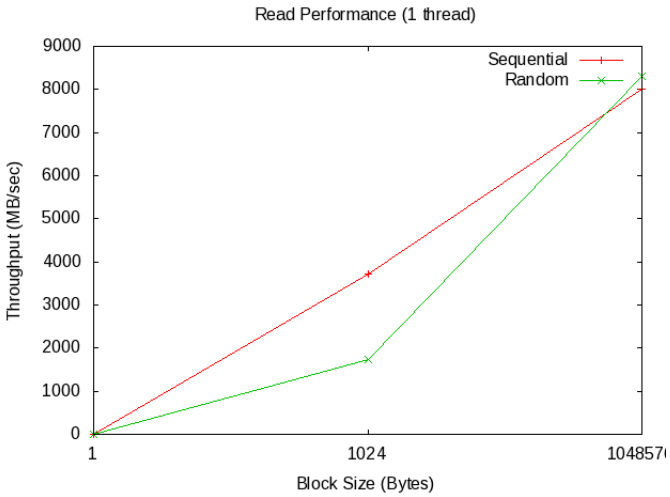

Figure 1: Block Size effect in Read Performance

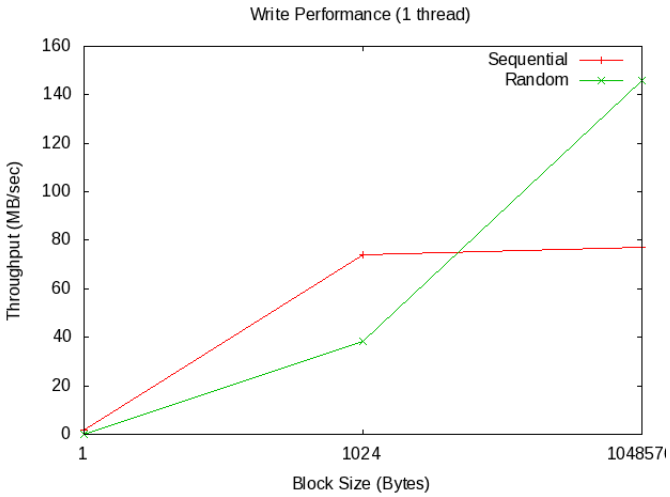

Figure 2: Block Size effect in Write Performance

It is clear that both read and write operations performance improve as bigger block sizes are used. This results are expected, as with bigger block sizes, more data is utilized at a time, and better usage of hardware is achieved. Sequential access performs better for 1KB operations. However, it is not the case for 1MB blocks. This is because the seek time plays a smaller role in the total operation time when the block size is bigger.

Also, the range of the y-axis of the two plots shows that write operations throughput is very low compared to read operations. This is because write operations are more expensive than read operations.

The figures 3 and 4 show the effect of the number of threads used. There is not a clear effect that the number of threads improves the performance. In fact, the plots suggest a slight degradation in the performance when using 2 threads instead of 1. This is because Amazon WS t2.micro instances have only 1 hardware thread. Thus, using more threads produces only an overhead to manage the threads which slightly degrades performance.

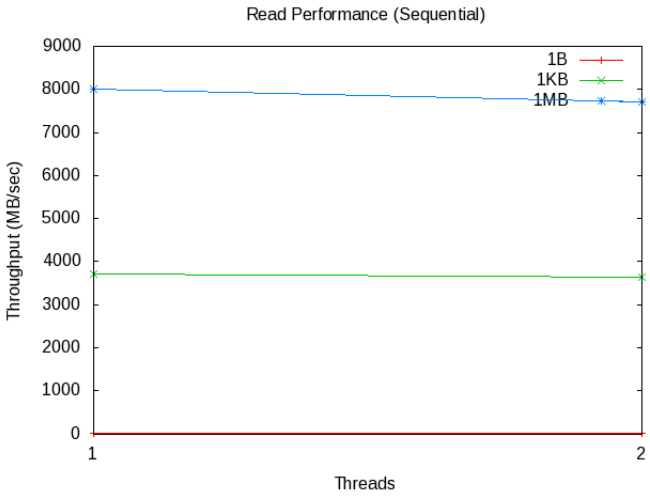

Figure 3: Number of threads effect in Read Performance

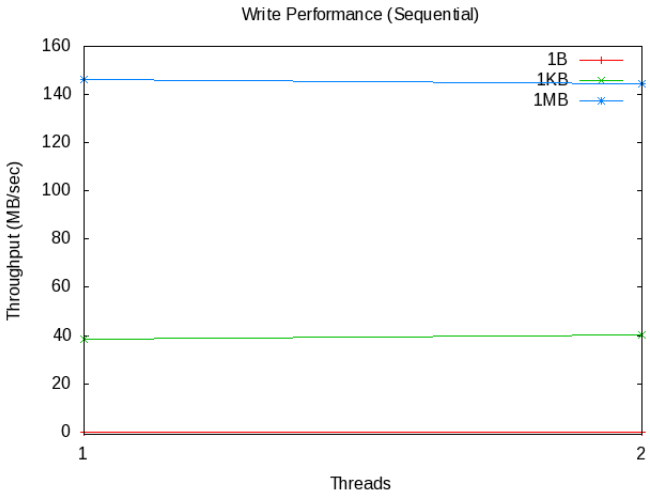

Figure 4: Number of threads effect in Write Performance

3.3 Iozone

I ran Iozone benchmark and compared the results obtained. Because the minimum block size supported by Iozone is 1KB, I could only obtained results for 1KB and 1MB block sizes. Also, I could only compare the experiment with 1 thread. This table summarizes the comparison:

Block Size (Bytes)	Operation	Strategy	My Benchmark Throuhput (MB/sec)	Iozone Throughput (MB/sec)
1024	Read	Seq	3725.17	3960.9
1024	Read	Random	1747.6	2119.5
1024	Write	Seq	74.10	670.1
1024	Write	Random	38.45	972.6
1048576	Read	Seq	8008.5	7719.6
1048576	Read	Random	8312	8000.4
1048576	Write	Seq	77.13	1868.1
1048576	Write	Random	146.08	3228.5

The results obtained by my benchmark program are very similar for the read operations of any size. However, write operations performance obtained by my benchmark look very low compared to Iozone.

3.4 Theoretical Performance

AWS does not provide data of the specific disk it is being used, which would allow to look for the theoretical peak performance from the vendor.

However, there are resources, discussions and forums in the web which can give a hint of the theoretical performance. The white paper **Amazon Web Services AWS Storage Services Overview** (`http://d0.awsstatic.com/whitepapers/AWS%20Storage%20Services%20Whitepaper-v9.pdf`) specifies that AWS Instance Storage offers "up to 3.5 GB/second read performance and 3.1 GB/second write performance with a 2 MB block size". This numbers are lower than the ones obtained by my benchmark and also Iozone. Others resources in the web from people performing benchmarks, such as `https://gist.github.com/ktheory/3c3616fca42a3716346b` and `http://atodorov.org/blog/2013/02/26/performance-test-amazon-ebs-vs-instance-storage-pt1/` give expected AWS local storage performance similar to the ones I got.

3.5 Possible Extensions

As future work for the disk benchmark, it would be desirable to compare different C libraries under the same experiments. It is expected that different libraries provide different performance results.

4 Memory Benchmark

4.1 Decisions

The memory benchmark program allocates a space of memory bigger than the cache in order to avoid its usage, which would distort the performance results. The program uses *malloc* function to allocate memory for both the source and target spaces of memory.

For each experiment the main process creates the corresponding threads, and assigns each of them an equal part of the memory to work with. In this way, collisions while

performing the operations are avoided. Once the indicated time has passed, the main process collects and aggregates the results.

The program uses *memcpy* C function to perform the operations. *memcpy* performs a real copy of characters between two areas of memory.

For block size of 1B, where the latency is very low, the overhead of generating a random number to choose a random block to read seems to be significant. To make the comparison between Sequential and Random access fair, I generate the random number even if the experiment is being performed for the sequential access.

4.2 Results

I first show a table with the results of the 12 experiments. For each of them, the throughput and latency obtained are shown. To make the results more accurate, I ran each of the experiments 3 times, each of them for 10 seconds, and then averaged them.

Block Size(Bytes)	Threads	Strategy	Throughput (MB/sec)	Latency (ms)
1	1	Seq	36.338134	0.000026
1	1	Random	7.669430	0.000120
1	2	Seq	36.416175	0.000026
1	2	Random	8.375060	0.000113
1024	1	Seq	5492.350636	0.000177
1024	1	Random	4180.293573	0.000227
1024	2	Seq	5490.603771	0.000177
1024	2	Random	4403.956328	0.000220
1048576	1	Seq	9023.974046	0.110799
1048576	1	Random	10448.436944	0.093725
1048576	2	Seq	8990.523552	0.111780
1048576	2	Random	10802.612433	0.089035

Figure 5 shows the effect of the block size in the performance. It can be observed that as the block size increases, the throughput also increases for both random and sequential strategies. It is explained by the fact that using a bigger block size permits to copy more data in a single operation, improving the throughput. As expected, also the latency (seconds to perform a single operation) increases when the block size does. It is because each operation is more expensive. See figure 6 to appreciate how the latency evolves with different block sizes.

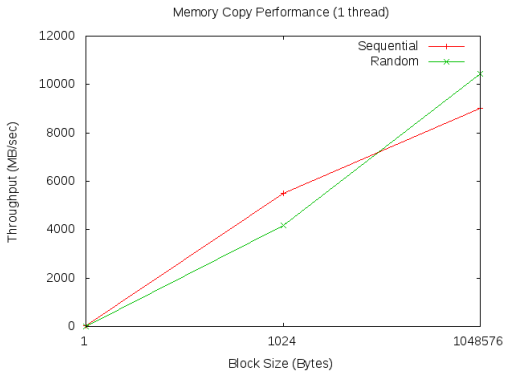

Figure 5: Block Size effect in Memory Throughput

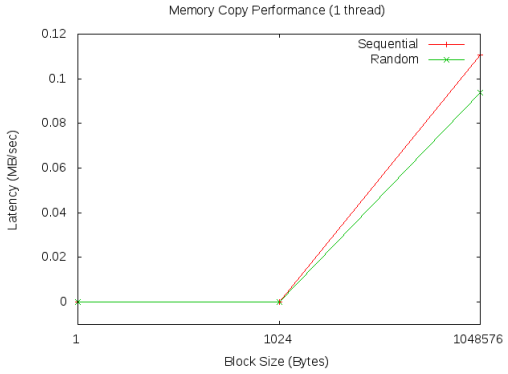

Figure 6: Block size effect in Memory Latency

The number of threads does not seem to improve the throughput for the memory operations. For different block sizes the throughput remains nearly constant by switching 1 thread to 2 threads.

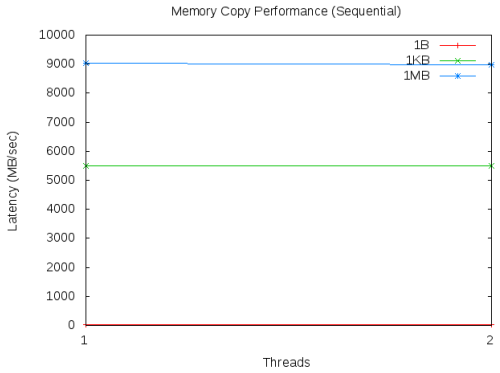

Figure 7: Number of threads effect in Memory Throughput

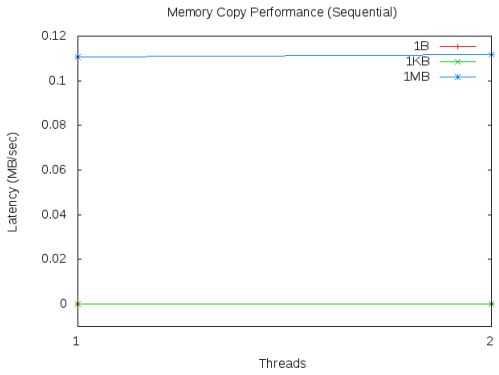

Figure 8: Number of threads effect in Memory Latency

4.3 Stream

I ran *Stream* benchmark and compared the results obtained. Stream does not offer flexibility to run experiments with different parameters. Based on the documentation, it performs operations of 8 bytes. The result obtained from Stream is **6054.2 MB/s**, which is comparable with the result obtained by my benchmark for blocks of 1KB.

It must be mentioned that Stream measures *copy* performance by doing the operation a[i] = b[i], where a and b are arrays of elements of 8 bytes.

4.4 Theoretical Performance

From the Intel specifications of the processor, I found that the maximum Memory Bandwitdth is 59.7 GB/s.

See the specifications in the following link: `http://ark.intel.com/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz`

This number is much higher than the one I obtained. However, it must be said that this is a *maximum* bandwidth mentioned by the vendor. Also, AWS instances are virtual machines and the hardware performance is not always the one given to each of the VMs.

4.5 Possible Extensions

As explained before, the overhead for generating a random number is significant when the block size is small. Other methods that perform better for generating random numbers could be analyzed and tried.

Because Stream benchmark is open source, I could modify the code to make it work with different block sizes. By doing this, I would be able to compare all the experiment cases.

5 CPU

5.1 Decisions

The cpu benchmark program performs integer and floating point operations. The four operations performed are +, -, * and /.

All the operation types contribute the same to the total operations, i.e. there is no distinction between each of the four operation types.

The code runs a loop. In each iteration, multiple operations are performed. It allows the processor take advantage of all its power, and not be idle at any time.

For each experiment the main process creates the corresponding threads. Once the indicated time has passed, the main process collects and aggregates the results.

To avoid compiler optimizations that could improve the performance, but distort the real result, I compile the code with the option -O0, which disables optimizations.

5.2 Results

The following table shows a summary of the results obtained for the six cpu experiments. To make the results more accurate, I ran each of the experiments 3 times, each of them for 20 seconds, and then averaged them.

Threads	Operation	GOPS
1	Int	1.282303
1	FP	0.425912
2	Int	1.287938
2	FP	0.426902
4	Int	1.287737
4	FP	0.425931

As can be seen in figures 9 and 10, there is a slight effect of the number of threads in the cpu performance. However, the range in the y axis shows that the improvement is not significant, and can be justified by random processor circumstances.

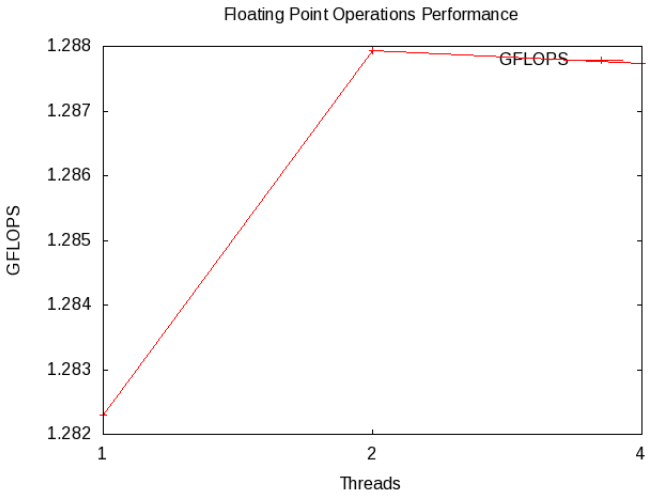

Figure 9: Number of threads effect in Floting Point Operations

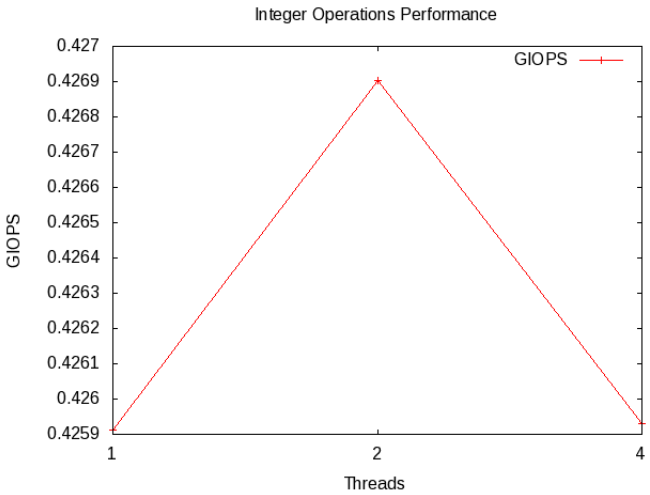

Figure 10: Number of threads effect Integer Operations

To see the behavior of the cpu in a long run of time, I ran my benchmark on floating point and integer instructions and 4 threads for a 10-minute period for each one. Figures 11 and 12 show the results obtained.

During a 10-minute period, most of the time the performance is very similar. However, every some amount of seconds the graphics show a degradation in the performance, which only lasts for 1 second. Both figures show a similar pattern. The guess to explain this is that the AWS VMM does not offer the VMs the full processor the whole time. During 10 minutes, there are times when it does, but others when the given processor performance given is lower.

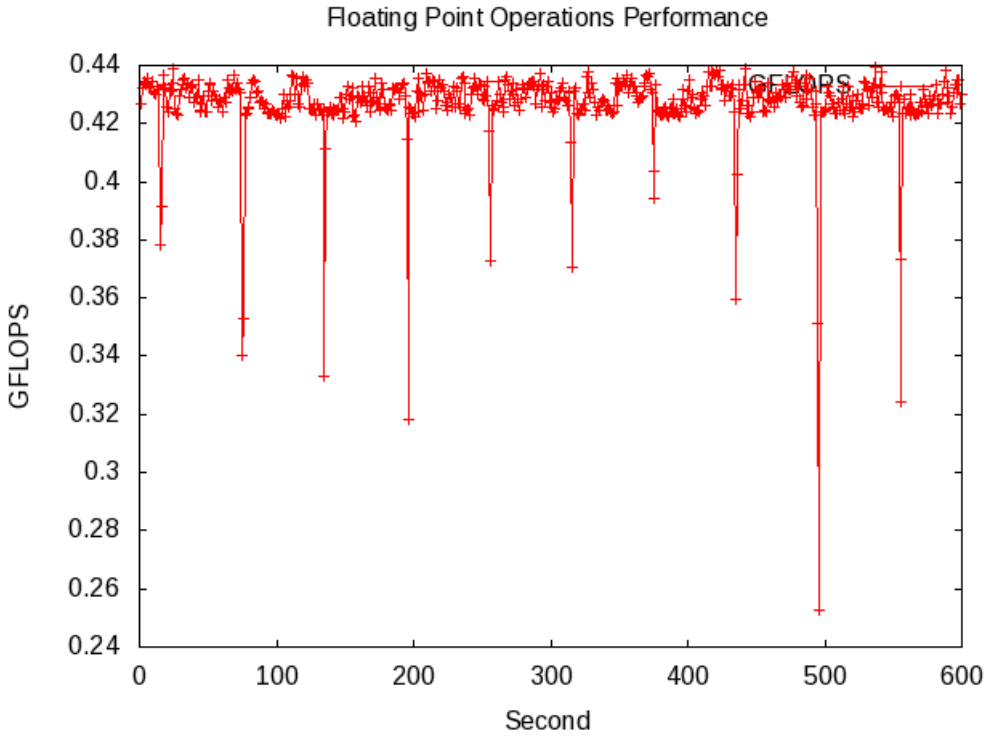

Figure 11: 10-minute run Floating Point Operations

Figure 12: 10-minute run Integer Operations

5.3 Linpack

I ran *Linpack* benchmark and compared the results obtained. The results for GFLOPS were 17.5, 18.9, 20.1 and 20.5 for 1000, 2000, 5000 and 10000 problem sizes. Linpack results are much better than my benchmark, and are more similar to the theoretical performance. Linpack performs matrix operations to run the experiments and might be achieving a better usage of the cpu.

5.4 Theoretical Performance

The CPU theoretical performance is defined as:

$$GHz * IPC * Cores$$

I obtained the necessary data from */proc/cpuinfo*. AWS t2.micro instances have 2.50GHz, 1 CPU core and 4 IPC. Thus, the theoretical peak performance is 10 GFLOPS.

6 Conclusions

During this assignment I could run experiments for Disk, Memory and CPU.

By offering a bash script which compiles the code, run the experiments and creates the plots, the benchmarks are easy to run. Many experiments with different parameters can be performed with low effort.

For each of the benchmarks, alternatives for the C libraries functions were analyzed and the ones with would offer more realistic results were chosen.

Results obtained for the benchmarks are similar but slightly lower than the theoretical performances and other benchmarks.