CS 553 - Cloud Computing, Spring 2018

PA 1 - Design Document

Benchmarking

Saumil Pareshbhai Ajmera | A20397303 | sajmera4@hawk.iit.edu

1. Processor Benchmarking

This benchmark measures the GigaOPS of a processor for combinations of data types and concurrency for 1 trillion arithmetic operations done in C language.

- I have used FMA instructions for single and double data type experiments and AVX2 instructions for alf and quarter precision data type experiments using _m256d and _m256i bit vectors for storing values.
- The code is being implemented in multithreaded manner with the strong scaling concept which keeps same amount of work per thread, running simultaneously and main thread awaits for all threads to complete.
- The *Main()* method reads the input .DAT file passed as an first argument and calls *executeCPU()* method where threads are created according to each data type and concurrency mentioned in the DAT file.
- *QFlps(), HFlps(), SFlps()* and *DFlps()* are the individual methods which runs for their corresponding data types. It computes 1 trillion arithmetic operation across multiple threads.
- Loop unrolling is achieved by dividing no of operations by instructions getting executed in one cycle as - Double - two 4-wide FMA instructions, Single - two 8-wide FMA instructions, Half - 64 AVX2 instructions and Quarter - 128 AVX2 instructions
- The output contains theoretical value, practical value and its efficiency which are stored in the output file which is passed as an second argument to the code.

2. Memory Benchmarking

This benchmark measures the throughput in (GB/sec) and latency in (us) for random and sequential read+write operations on the memory for multiple block size and different concurrency done in C language.

- Here 1KB, 1MB and 100 MB block sizes are used to access 1GB of memory over 100 times in 1,2 and 4 threads in sequential and random manner for calculating throughput.
- The code is being implemented in multithreaded manner with the strong scaling concept which keeps same amount of work per thread, running simultaneously and main thread awaits for all threads to complete.
- Here in order to access separate block per each thread, I have divided entire 1GB of memory (src pointer) into blocks per thread. I have used a struct data type named thread_info_struct to store startIndex, endIndex, blockLength and no_of_ops for each thread and passed it as an argument while creating thread.
- The *Main()* method reads the input .DAT file passed as an first argument and calls *ExecuteRam()* method where threads are created according to operation and concurrency mentioned in the DAT file.

- ➢ *seq_access()* and *rand_access()* are the methods which does read (src pointer)+write(dest pointer) operation by *memcpy* command in a sequential and randomly respectively for 100 times for given block size.
- ➢ *latseq_access()* and *latrand_access()* are the methods which does read (src pointer)+write (dest pointer) operation over 1 byte of data for in a random order for calculating latency.
- ➢ The output contains theoretical and practical throughput values and its efficiency which are stored in the output file which is passed as an second argument to the code.

3. Disk Benchmarking

This benchmark measures the throughput in (MB/sec), IO operations in (IOPS) and latency in (ms) for combination of read, write along with random and sequential patterns on the memory for multiple block size and different concurrency done in C language.

- ➢ Here 1MB, 10MB and 100MB block sizes are used to access(reading and writing) 10GB of file created over a disk in /tmp directory of compute node in which job is running with 1,2 and 4 threads in sequential and random manner for calculating throughput.
- ➢ The code is being implemented in multithreaded manner with the strong scaling concept which keeps same amount of work per thread, running simultaneously and main thread awaits for all threads to complete.
- ➢ Here in order to access separate block per each thread, I have divided entire 10GB of file into blocks per thread. I have used a struct data type named thread_info_struct to store startIndex, endIndex and blockLength for each thread and passed it as an argument while creating thread.
- ➢ The *Main()* method reads the input .DAT file passed as an first argument and calls *ExecuteDisk()* method where threads are created according to operation and concurrency mentioned in the DAT file.
- ➢ *sampleWrite()* method prepares 10 GB and 1GB of file with 10 parallel threads for reading purpose in /tmp directory
- ➢ *randRead_access()* and *seqRead_access()* are the methods which does read operation by *fread* command in a sequential and randomly respectively for given block size.
- ➢ *seqWrite_access()* and *randWrite_access()* are the methods which does write operation over *fwrite* command in a sequential and randomly respectively for given block size.
- ➢ The output contains theoretical and practical throughput values and its efficiency which are stored in the output file which is passed as an second argument to the code.

4. Network Benchmarking

This benchmark measures the throughput in (Mb/sec) and latency in (ms) for network data transmission of 1GB, using TCP and UDP protocols over multiple block size and different concurrency for 100 times done in C language.

- Here 1KB and 32KB block sizes are used to transfer 1GB of data from memory of one compute node to memory of another compute node in sequential manner for calculating throughput.
- The code is being implemented in multithreaded manner for TCP and UDP protocols with the strong scaling concept which keeps same amount of work per thread, running simultaneously and main thread awaits for all threads to complete.
- Here in order to access separate block per each thread, I have divided entire 10GB of data into blocks per thread. I have used a struct data type named thread_info_struct to store startIndex, endIndex, ops and blockLength for each thread and passed it as an argument while creating thread.

TCP Code:

- The *Main()* method reads the input .DAT file passed as an first argument and calls *TCPClient() or TCPServer()* method where sockets are created and then threads are created according to operation and concurrency mentioned in the DAT file.
- Here depending upon thread multiple socket will be created for a single client and server will also accept multiple client socket connection.
- *TCPServerData()* and *TCPClientData()* are the methods which performs data receive and send using *recv and send* command in for a given block size for 100 times.
- *TCPServerPing*() and *TCPClientPing*() are the methods which performs round trip time (latency) of 1B of data using *recv* and *send* command for for 1 million operations.
- The output contains theoretical and practical throughput values and its efficiency which are stored in the output file which is passed as an second argument to the code.

UDP Code:

- The *Main()* method reads the input .DAT file passed as an first argument and calls *UDPClient() or UDPServer()* method where sockets are created and then threads are created according to operation and concurrency mentioned in the DAT file.
- Here depending upon thread multiple socket will be created for a single client and server will also accept multiple client socket connection.
- *UDPServerData()* and *UDPClientData()* are the methods which performs data receive and send using *recvfrom and sendto* command in for a given block size for 100 times.
- *UDPServerPing*() and *UDPClientPing*() are the methods which performs round trip time (latency) of 1B of data using *recvfrom* and *sendto* command for 1 million operations.
- The output contains theoretical and practical throughput values and its efficiency which are stored in the output file which is passed as an second argument to the code.