README

By: Anmol Agarwal (2018CS10327)

Part A

How to run:

If only input file is provided then all policies are run, and graph is plotted in output directory.

make ARGS=”./src/input.txt FF LR LR SC”

Implementation Details

Created struct element which stores page_number, reference_bit, counter and reference_array of 8 integers. And main memory is stored as an array of data of type struct element.

Initialize_memory(): this function sets the value of each member of the array to default values, i.e., page number is -1, reference_bit=0, counter=0, and reference_array (which would be needed to approximate LRU using 8 bits) is initialized with all values to 0.

This function would be called every time new policy is implemented to refresh the memory to default initial values.

The logic behind each policy is as follows:

1. **FIFO:** memory array was treated like a queue by making use of pointer variable, which essentially stored the reference to the block which needs to replaced during page fault. When a given page (maintaining FIFO) is replaced by a new page, we increment the pointer to reference to next block (because the next block now becomes the earlier comer to the memory array).
   But there few corner cases to consider when memory had empty space, in this case, instead of replacing page, we needed to add new page.

2. **LFU:** This policy made use of counter variable in element struct which stored the number of times the given page was referenced. In case of page fault, if memory had empty space new page was added else find_min_index_LFU() function was called which found the index in memory array with least counter value. And then that index was replaced with new page and counter initialized back to 0.
   Note: In case of multiple elements with same counter, the element with earliest index in memory array was replaced.
3. **LRU:** This made use to reference_array[] which simulated the role of a time stamp. Since using time stamp in real architectures is very costly, one simple way to is to use an array of 8 bits for each page.
   In each page access, the array was shifted to right by 1 bit, and in case of page hit, that page's reference array's most significant bit was changed to 1 indicating that page was reference at that time.
   Note: In case of multiple elements with same counter, the element with earliest index in memory array was replaced.
4. **Clock:** This again made use of pointer which now acted as a handle of a clock and also used reference_bit variable of element struct. In case of page miss, when the handle pointed to page with reference_bit=1, then that page's reference_bit was turned to 0, indicating that it was given a chance to be referenced again until next time the clock handler comes back to same back. And if the reference_bit was 0 then that page was replaced.
   Finally, in case of page hit, that page's reference_bit was set to 1.
5. **N-Chance:** This was same as clock (also known as 2-Chance) but now in case of page miss, page with reference_bit=0 was allowed to be pointed to by clock handler N-1 times before

being removed. So in total, page was given N chances before finally being replaced.

Part B:

Description of Experiment:

1. Ubuntu 20.04 was the native ubuntu used.
2. Docker with Dockerfile (to install dependencies and transfer Assignment 2 and y-cruncher folder from Base OS to Docker).
3. QEMU (6 GB RAM, 4 cores, Image of size 30 GB, and virtio enabled for better graphics)

Command: `qemu-system-x86_64 -enable-kvm -boot menu=on -drive file=Image.img -m 6G -cpu host -smp 4 -vga virtio`

y-cruncher: Benchmark was done on calculating value of pi upto 25M, 50M, 100M, 250M and 500M on all three platforms. Both single threaded and multi-threaded performance was measured.

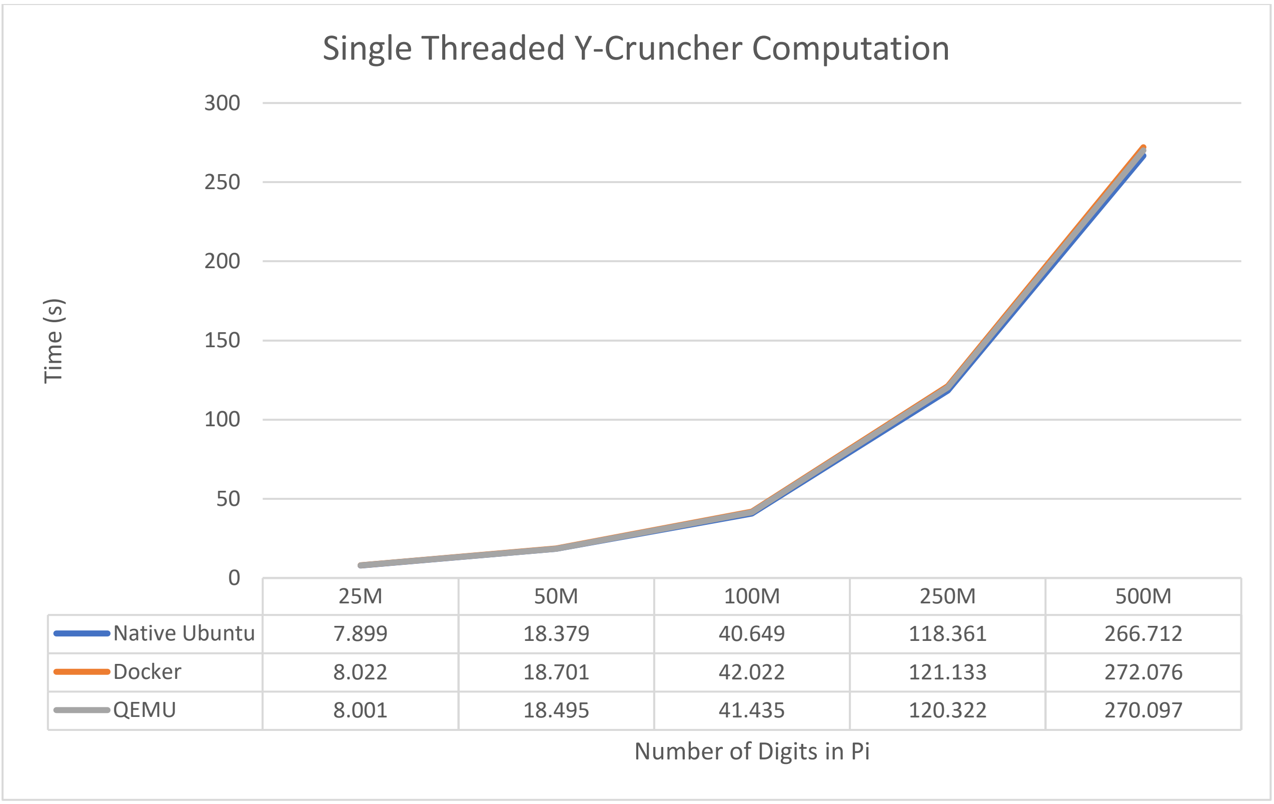
In multithreaded case, docker used 24 threads in Native Ubuntu and Docker and 4 threads in QEMU.

Assignment 2: Benchmark was done on both naïve and load-balanced approach (1,2,4,8 threads) on input size of 10^5,10^6,10^7 (for all) and 10^8 (for only 4,8 number of threads).

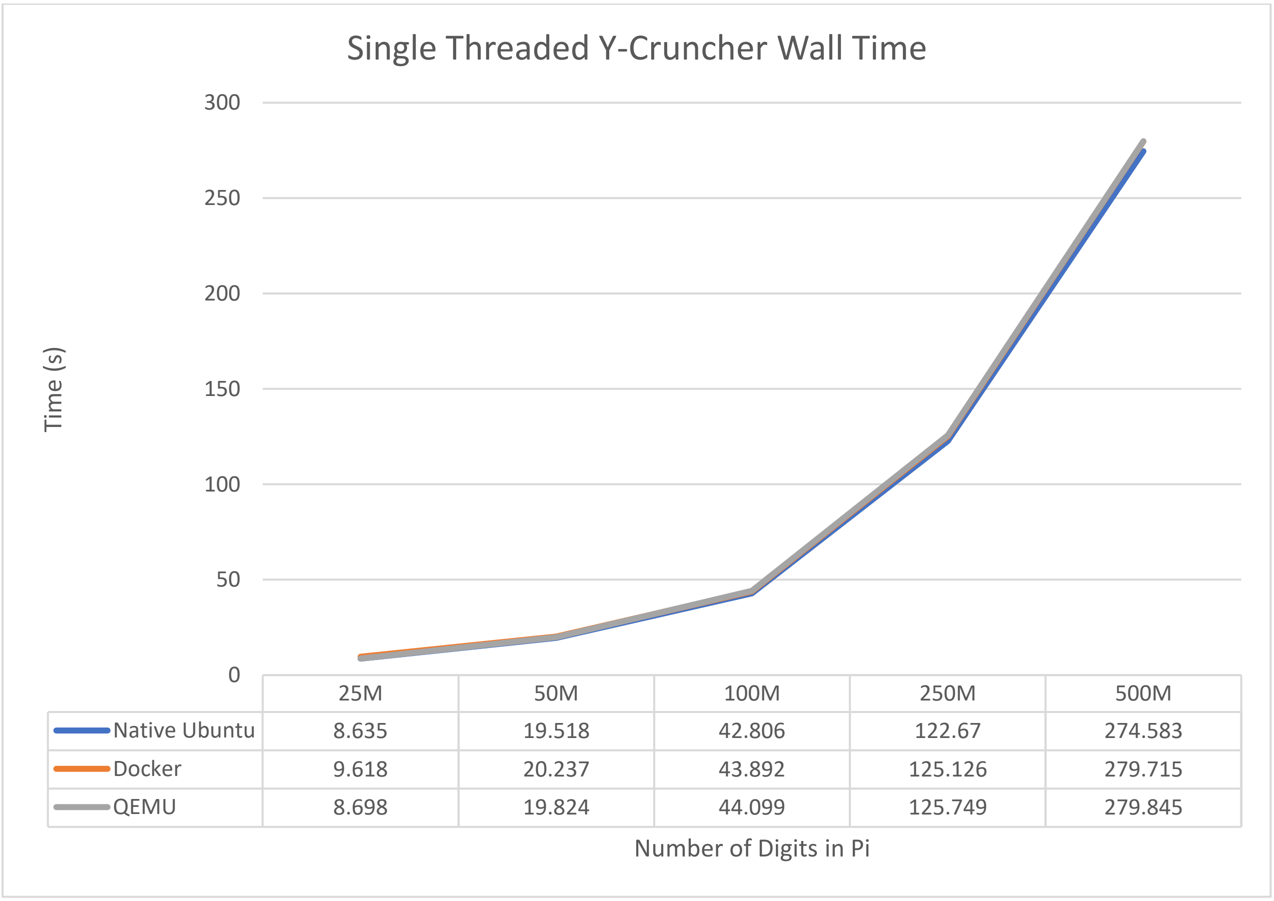
Performance was measured and graph was plotted.

The graphs for each benchmark are as follows:

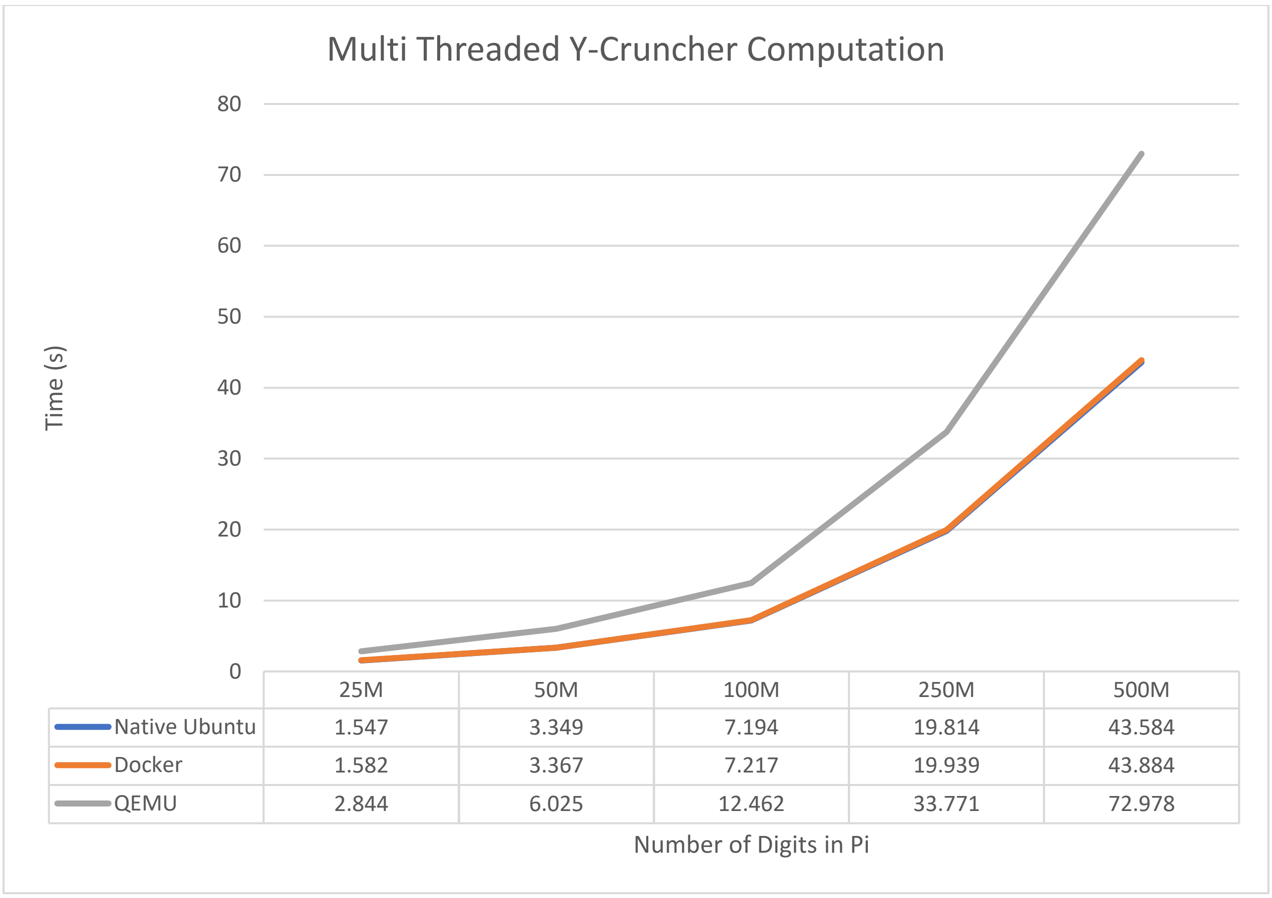
Y-CRUNCHER



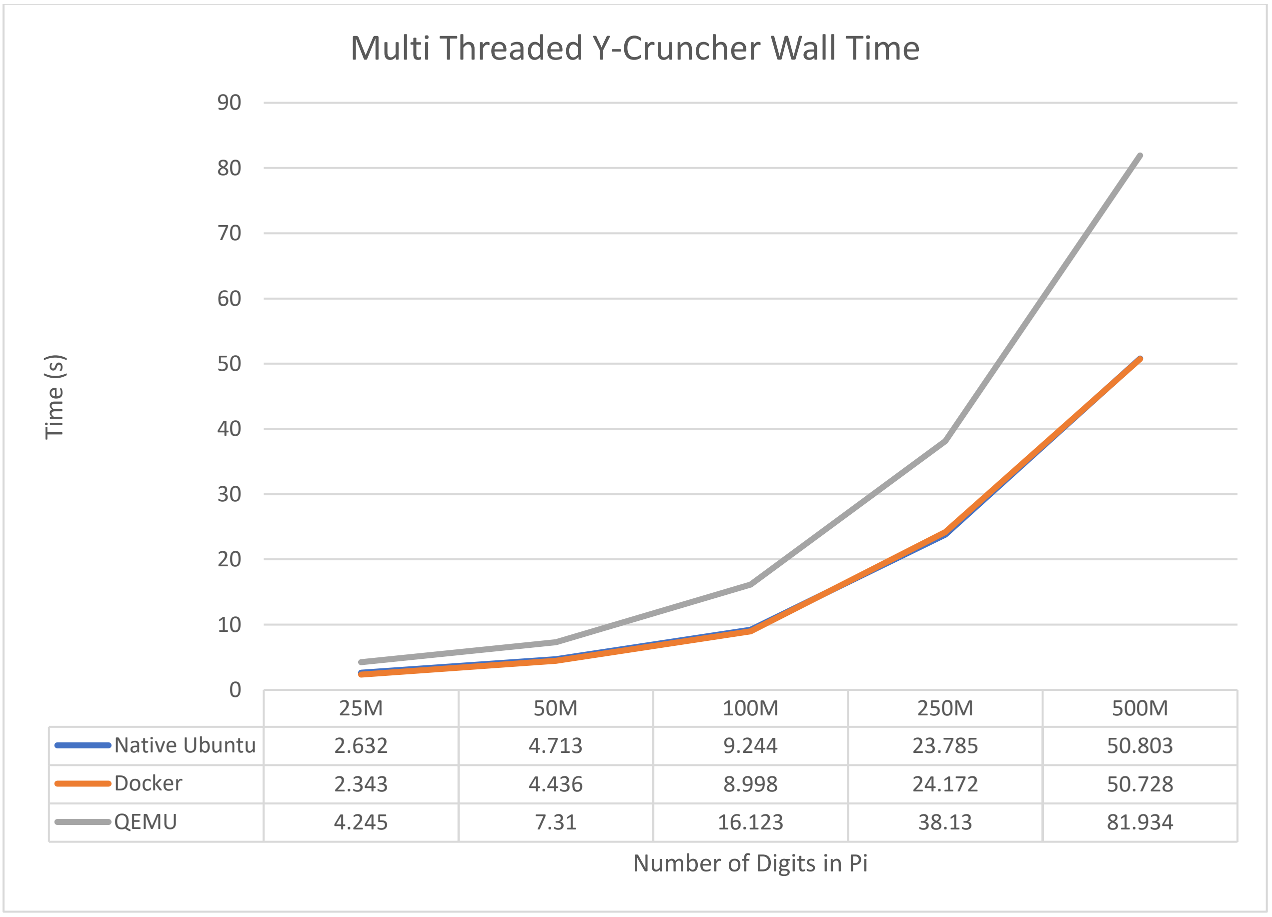
	25M	50M	100M	250M	500M
Native Ubuntu	7.899	18.379	40.649	118.361	266.712
Docker	8.022	18.701	42.022	121.133	272.076
QEMU	8.001	18.495	41.435	120.322	270.097



	25M	50M	100M	250M	500M
Native Ubuntu	8.635	19.518	42.806	122.67	274.583
Docker	9.618	20.237	43.892	125.126	279.715
QEMU	8.698	19.824	44.099	125.749	279.845

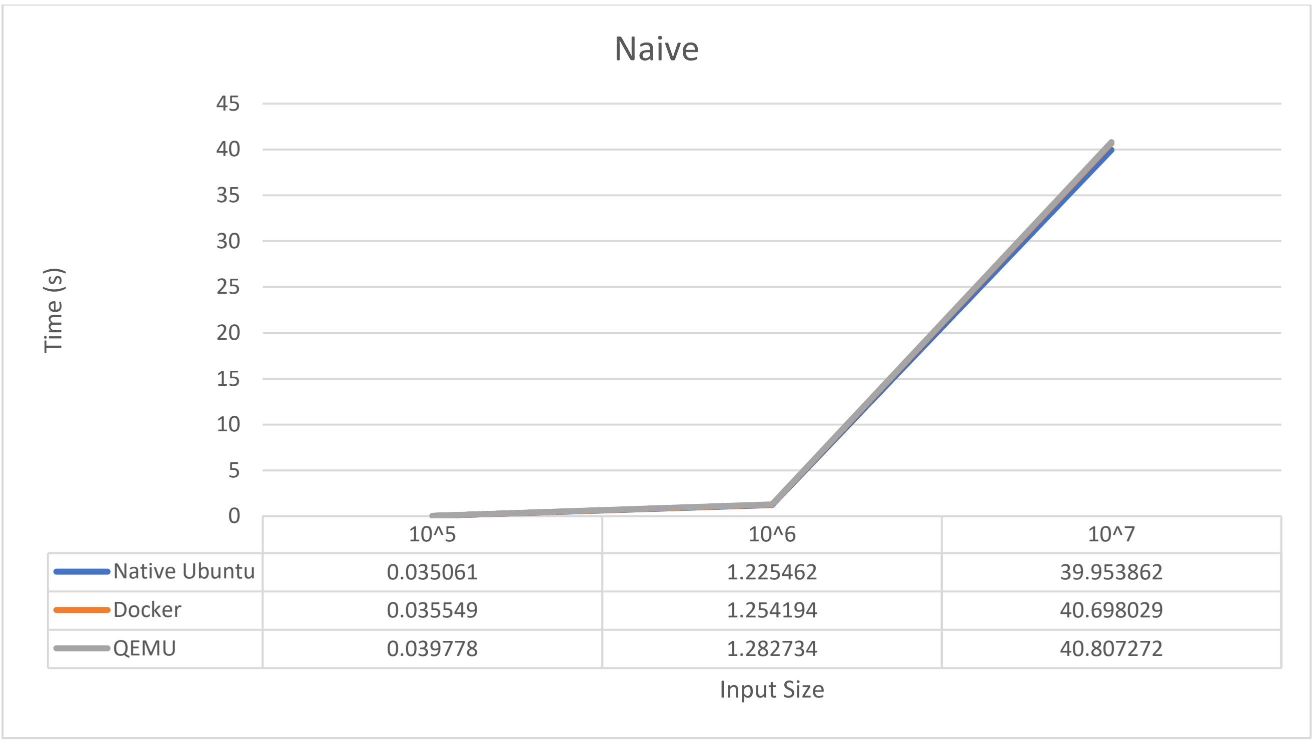


	25M	50M	100M	250M	500M
Native Ubuntu	1.547	3.349	7.194	19.814	43.584
Docker	1.582	3.367	7.217	19.939	43.884
QEMU	2.844	6.025	12.462	33.771	72.978

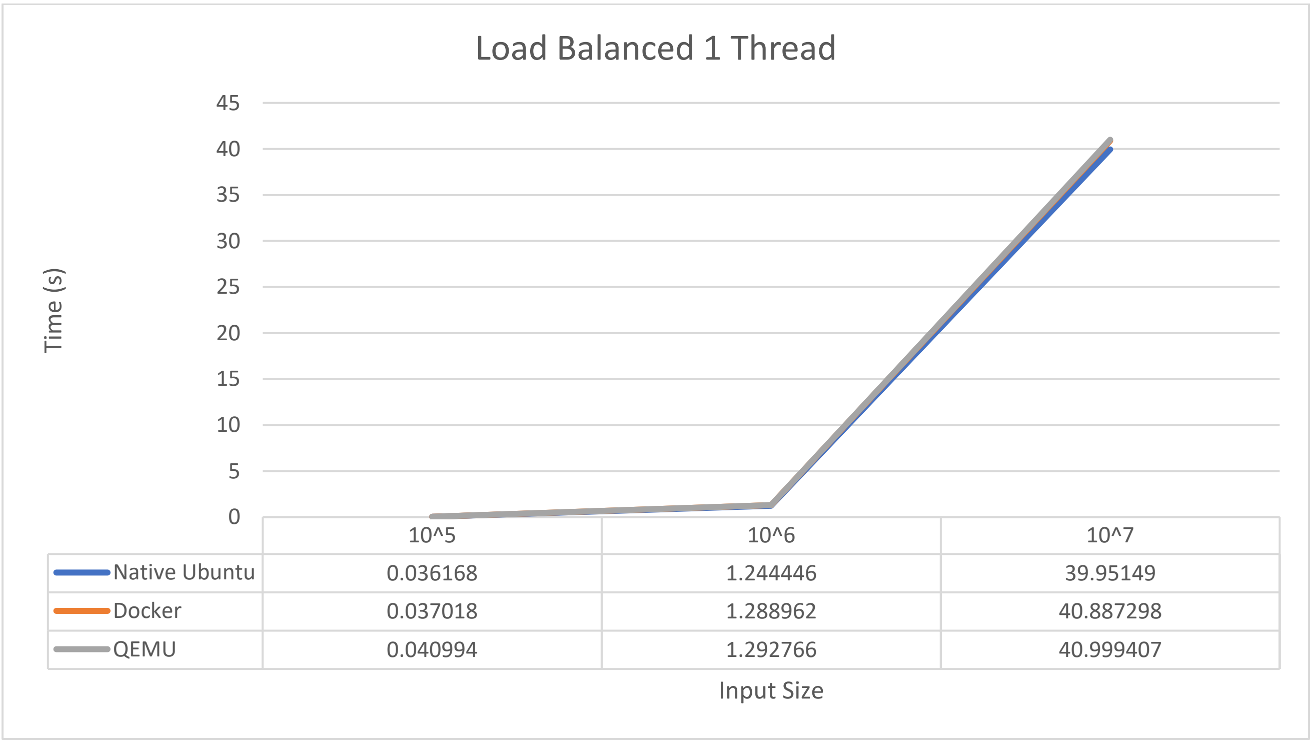


	25M	50M	100M	250M	500M
Native Ubuntu	2.632	4.713	9.244	23.785	50.803
Docker	2.343	4.436	8.998	24.172	50.728
QEMU	4.245	7.31	16.123	38.13	81.934

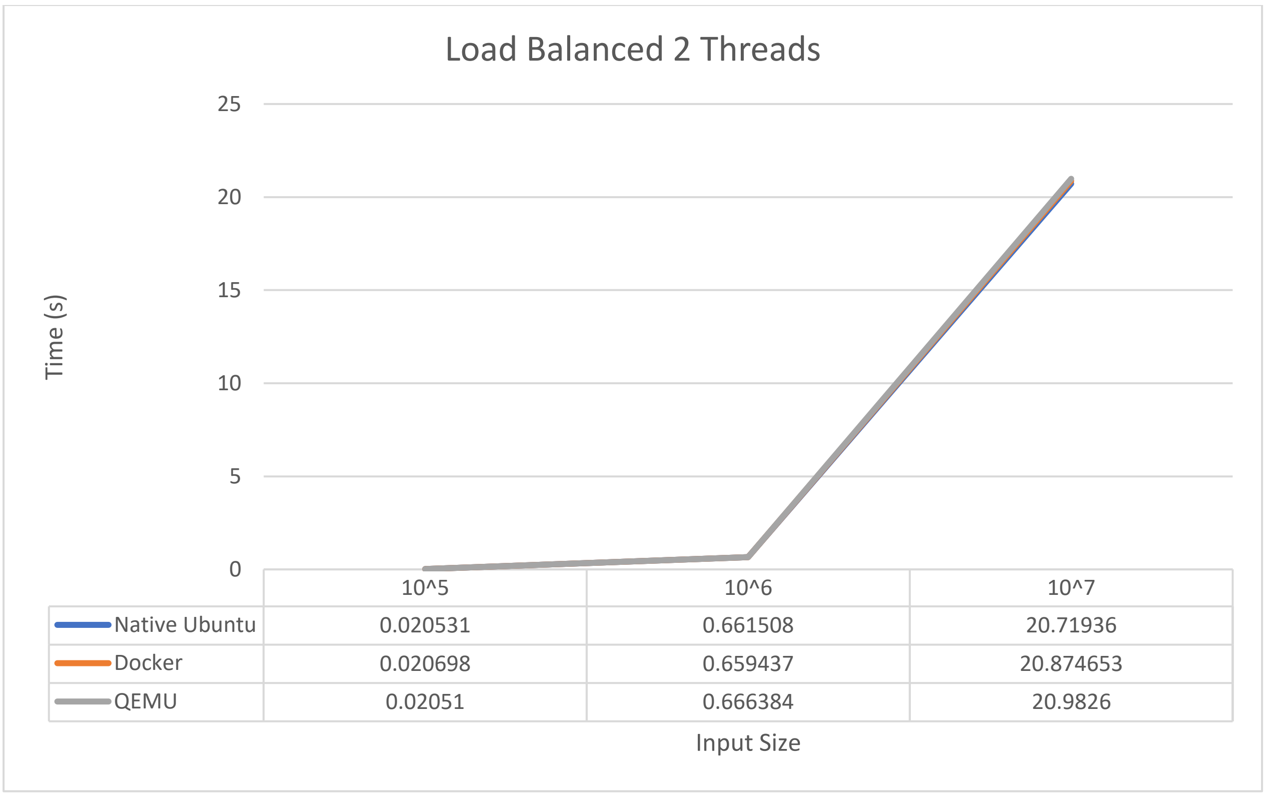
Assignment 2



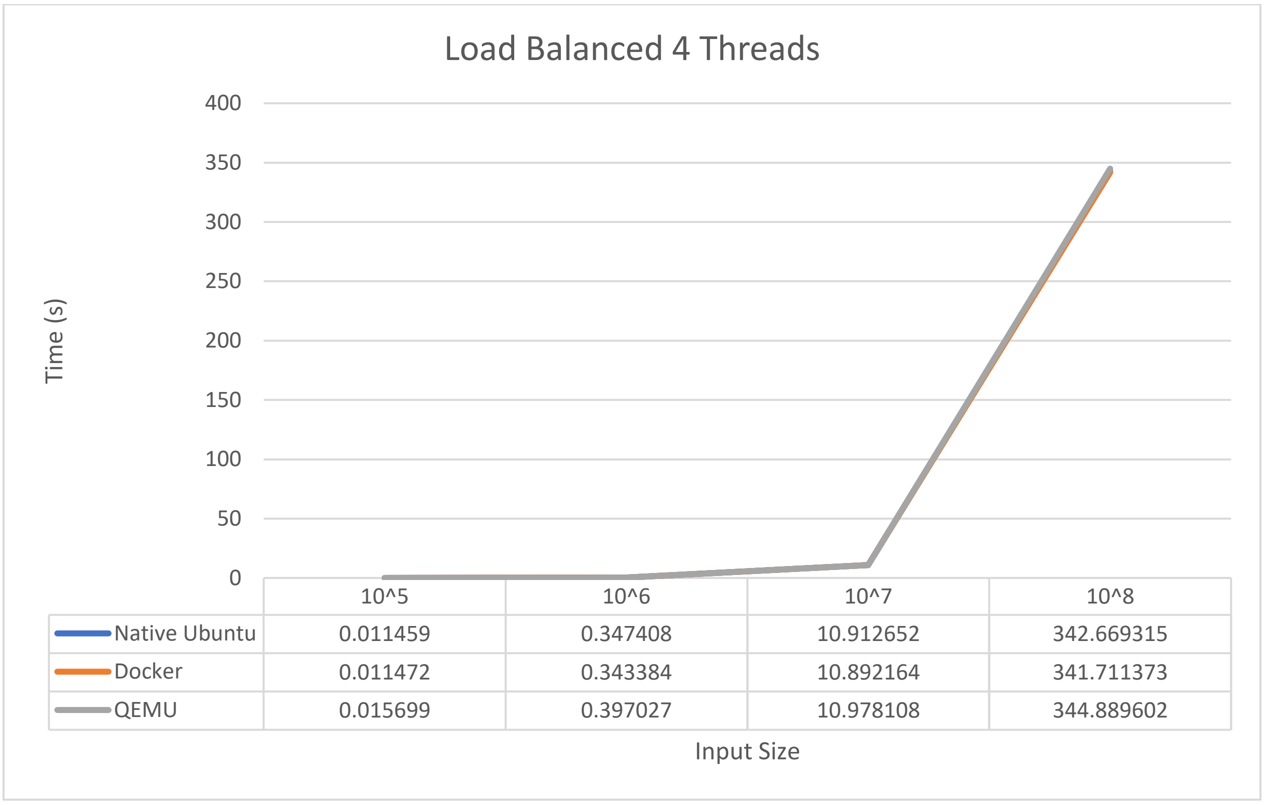
	10^5	10^6	10^7
Native Ubuntu	0.035061	1.225462	39.953862
Docker	0.035549	1.254194	40.698029
QEMU	0.039778	1.282734	40.807272



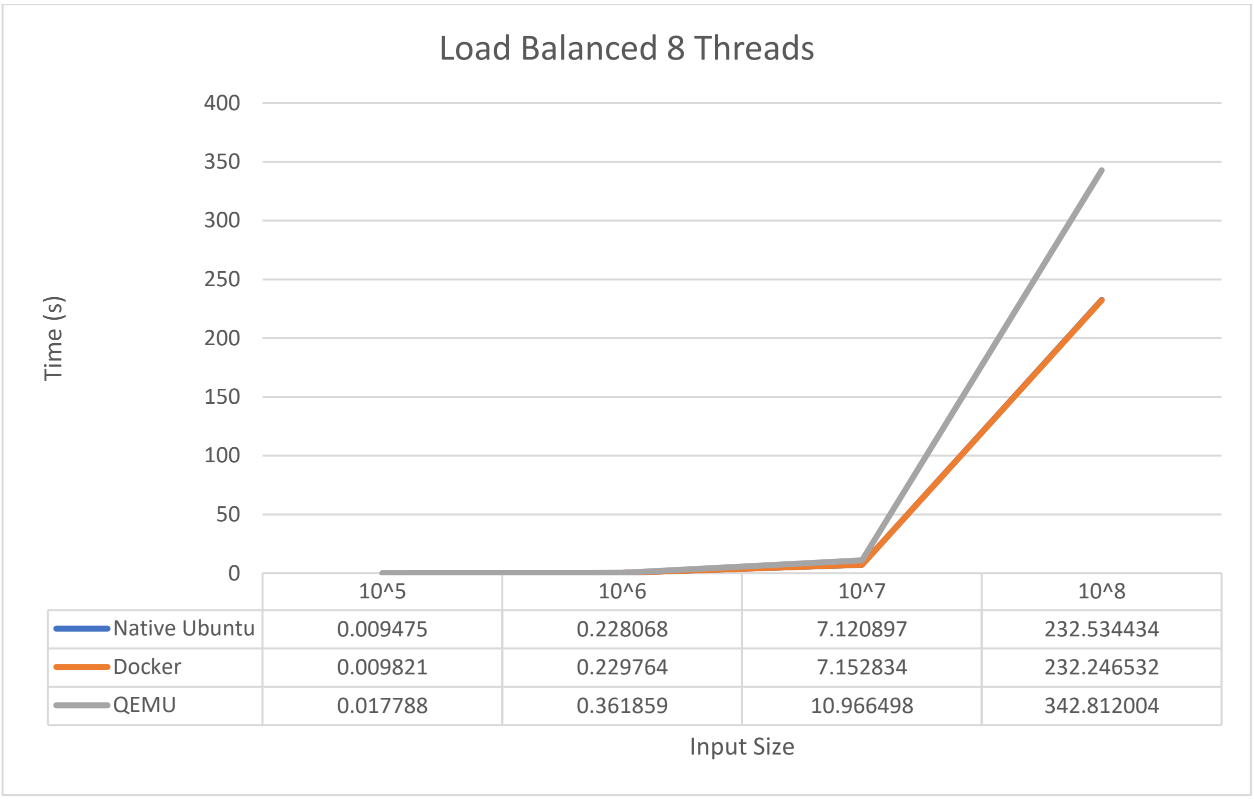
	10^5	10^6	10^7
Native Ubuntu	0.036168	1.244446	39.95149
Docker	0.037018	1.288962	40.887298
QEMU	0.040994	1.292766	40.999407



	10^5	10^6	10^7
Native Ubuntu	0.020531	0.661508	20.71936
Docker	0.020698	0.659437	20.874653
QEMU	0.02051	0.666384	20.9826



	10^5	10^6	10^7	10^8
Native Ubuntu	0.011459	0.347408	10.912652	342.669315
Docker	0.011472	0.343384	10.892164	341.711373
QEMU	0.015699	0.397027	10.978108	344.889602



	10^5	10^6	10^7	10^8
Native Ubuntu	0.009475	0.228068	7.120897	232.534434
Docker	0.009821	0.229764	7.152834	232.246532
QEMU	0.017788	0.361859	10.966498	342.812004

Results

Y-Cruncher

As seen from the above graph there is negligible performance difference in both single thread and multi-threaded operations.

In single thread the general trend was : Native<QEMU<Docker

With docker being the slowest and Native being fastest.

In multi-thread the general trend was : Native<Docker<QEMU

With QEMU being the slowest and Native being fastest.

Assignment 2

As seen above, there is again negligible performance difference except for the last case of 8 threads in which VM was approximately 2 times slower than the other two because of access to only 4 virtual CPUs, so it was effectively running 4 threads in parallel, whereas both Native Ubuntu and Docker had access to 6 cores. QEMU was generally slowest in all cases as expected in case of VMs.

Discussion

In the above experiment docker and native ubuntu had no difference in performance and Virtual Machine QEMU was slightly slower. But the gap in performance would become more noticeable in case of heavier computations using more than 4 threads (and QEMU has only 4 virtual cores).

But even in case of less than equal to 4 threads in Assignment 2 benchmark, QEMU was slower.

Hence as expected, VMs turned out to be slightly slower than docker and native OS.

More files to look at:

Benchmark_data.xlsx : stores the data of benchmark measurements of part B.