Report for Ques 2

Q2: Matrix Multiplication

What I Did

I wrote a program to multiply two 1000×1000 matrices (1 billion operations total). Matrix multiplication takes each row from the first matrix and multiplies it element-by-element with each column from the second matrix. With matrices this big, it takes serious computation, which is why I wanted to parallelize it with OpenMP.

But I didn't just do regular matrix multiplication. I added transpose optimization because normal multiplication has a major problem - you read Matrix A row-by-row (fast) but Matrix B column-by-column (super slow). Column access is slow because computers store data sequentially in rows, so reading columns means jumping all over memory. This causes "cache misses" where the CPU fetches data from slow RAM instead of fast cache.

My solution was to transpose Matrix B before multiplication. Transposing flips the matrix so rows become columns. Now both Matrix A and transposed Matrix B are read row-by-row, which is way more cache-friendly. The CPU can load entire rows into cache at once and zip through calculations much faster.

I implemented two parallel versions:

Version 1 (1D threading): I put #pragma omp parallel for on just the outermost loop (i-loop). Each thread handles complete rows of the result matrix. For example, with 4 threads on 1000×1000: Thread 1 calculates rows 0-249, Thread 2 does 250-499, Thread 3 gets 500-749, and Thread 4 finishes 750-999. This is the simplest parallelization - just split rows among threads.

Version 2 (2D collapse): I used #pragma omp parallel for collapse(2) which parallelizes both i and j loops together. The collapse(2) clause treats the two nested loops as one big combined loop and distributes work more finely. Instead of complete rows, it spreads individual elements across threads. Threads work on scattered elements throughout the matrix rather than contiguous rows. I thought this would give better load balancing.

Results

Version 1 - 1D Threading:

Sequential version took 3.886083 seconds. With 2 threads: 3.838s (1.01× speedup, 50.62% efficiency) - basically no improvement. With 3 threads it got worse: 4.123s (0.94× speedup). Four threads gave 3.740s (1.04× speedup), which was my best result but still disappointing.

As I added more threads, performance stayed bad or got worse. With 5 threads: 4.455s (0.87× speedup), 8 threads: 4.221s (0.92× speedup), 16 threads: 4.109s (0.95× speedup). More threads made it slower instead of faster. Efficiency dropped from 50.62% with 2 threads to just 5.91% with 16 threads - most threads were wasting time.

The perf stat numbers explained everything. CPUs utilized was only 0.872 - the system barely used one full CPU core on average. That's terrible! Context-switches were 63,691, way too high. Each switch means the OS paused one thread and started another, wasting time and clearing cache. With over 63,000 switches, threads were interrupted constantly. Task-clock was 69,181.96ms, elapsed time was 79.311s, but user time was only 68.998s with system time at 0.147s. This gap shows threads were waiting instead of computing.

Version 2 - 2D Threading with Collapse:

Sequential time was 4.061446 seconds, about 4.5% slower than Version 1's baseline. Not sure why - maybe how collapse affects code generation. With 2 threads: 3.730s (1.09× speedup, 54.45% efficiency) - actually better than Version 1! With 3 threads: 3.849s (1.06× speedup). But with 4 threads: 4.230s (0.96× speedup - slower than sequential).

As I added more threads, Version 2 showed weird inconsistent behavior. With 5 threads: 5.071s (0.80× speedup - really bad), 8 threads: 5.698s (0.71× speedup), 11 threads: 5.178s (0.78× speedup), 15 threads: 4.334s (0.94× speedup), 16 threads: 5.602s (0.72× speedup). Performance jumped all over the place.

Performance counters showed task-clock of 63,400.18ms (8% less than Version 1) and CPUs utilized of 0.927 - slightly better than Version 1's 0.872. Context-switches were 58,518 (8% lower than Version 1's 63,691). Page-faults were similar at 7,952. Elapsed time was 68.378s with user time of 63.326s and system time of only 0.058s. So Version 2 showed marginally better CPU utilization and fewer context switches, but overall performance was still terrible.

What I Learned

This was frustrating because I expected parallelization to help with such a big problem. But both versions failed to show consistent speedup and often made things slower.

First, why transpose helped at all: My sequential times of 3.9-4.1 seconds are probably way better than without transpose. Regular column-wise access might have taken 10-15 seconds due to cache misses. Transpose made the sequential algorithm much faster by ensuring row-by-row reads. But it didn't help parallelization.

The Memory Bandwidth Bottleneck: The CPUs utilized numbers (0.872 for Version 1, 0.927 for Version 2) are the smoking gun - threads aren't running in parallel, they're waiting for memory! Even though transpose fixed cache access, all threads still need massive data from RAM. The memory bus can only transfer so much per second, and when multiple threads read simultaneously, they wait in line for memory access.

Think of a highway analogy: transpose made the road straight (good cache), but multiple cars (threads) on the same highway still cause traffic jams because highway capacity is limited. That's memory bandwidth limitation. RAM can only pump out data so fast, and multiple threads fighting for it creates congestion.

Context Switching Chaos: 63,691 switches (V1) and 58,518 switches (V2) are insane. Each time the OS switches threads, it saves current state, loads new state, and usually clears CPU cache. So when a thread finally runs, its data might not be in cache anymore because

another thread kicked it out. Each switch wastes CPU cycles and destroys cache performance. Threads spent more time paused/resumed than computing.

Version 1 vs Version 2: Version 1 gave each thread complete rows with good spatial locality - sequential memory access for row i. It showed consistent performance across thread counts, though never achieved real speedup. Version 2 spreads work more finely, which should give better load balancing (shown by better CPU utilization 0.927 vs 0.872 and fewer context switches). But Version 2's performance was way more inconsistent and unstable. It was better at 2-3 threads (1.09× and 1.06× speedup), but much worse at higher counts (down to 0.71×).

I think the collapse directive adds overhead to merge loops, but when it works well (low thread counts), finer work distribution helps. But as threads increase, managing many threads working on scattered elements kills performance. The inconsistency suggests threads interfere with each other's cache more because they jump around the matrix instead of working on contiguous rows.

Version 1 performed slightly better on average and was way more stable. Version 2 had a couple of better data points but was much worse overall and unpredictable. Version 1 is clearly better for this problem - at least it's consistent, even if that consistency is "consistently no speedup."

Thread Overhead: For 1000×1000 matrices with 16 threads, each Version 1 thread gets only 62-63 rows. Version 2 splits even finer. Creating, distributing, and synchronizing threads takes time. For this problem size, overhead costs more than parallelization benefit. With 10000×10000 matrices, overhead would matter less.

VirtualBox Limitation: Running in a VM likely prevented true parallelism. CPU utilization maxing at 0.927 suggests the VM doesn't give full access to physical cores. Threads might time-share on virtual cores instead of running truly in parallel on separate physical cores.

The Numbers: Task-clock (69s for V1, 63s for V2) roughly equals user time, meaning threads weren't truly parallel. If they were, user time would far exceed elapsed time (multiple cores computing simultaneously). The gap between elapsed and user time (79s vs 69s for V1, 68s vs 63s for V2) shows threads waited, not computed. System time being tiny (0.147s for V1, 0.058s for V2) means OS overhead isn't the issue - it's purely memory bandwidth.

Comparison to Other Labs: DAXPY was memory-bound and too small (no speedup). Pi calculation was compute-bound with minimal memory access (3.88× speedup). Matrix multiplication, despite being large, is fundamentally memory-bound. Each of 1 billion elements needs multiple memory reads, and the memory system can't keep up with multiple threads.

Key Lesson: Algorithmic optimization ≠ parallel optimization. Transpose made sequential code way faster, but didn't solve parallel scaling. Making something fast sequentially doesn't mean it parallelizes well. You need to consider hardware bottlenecks like memory bandwidth, not just the algorithm.

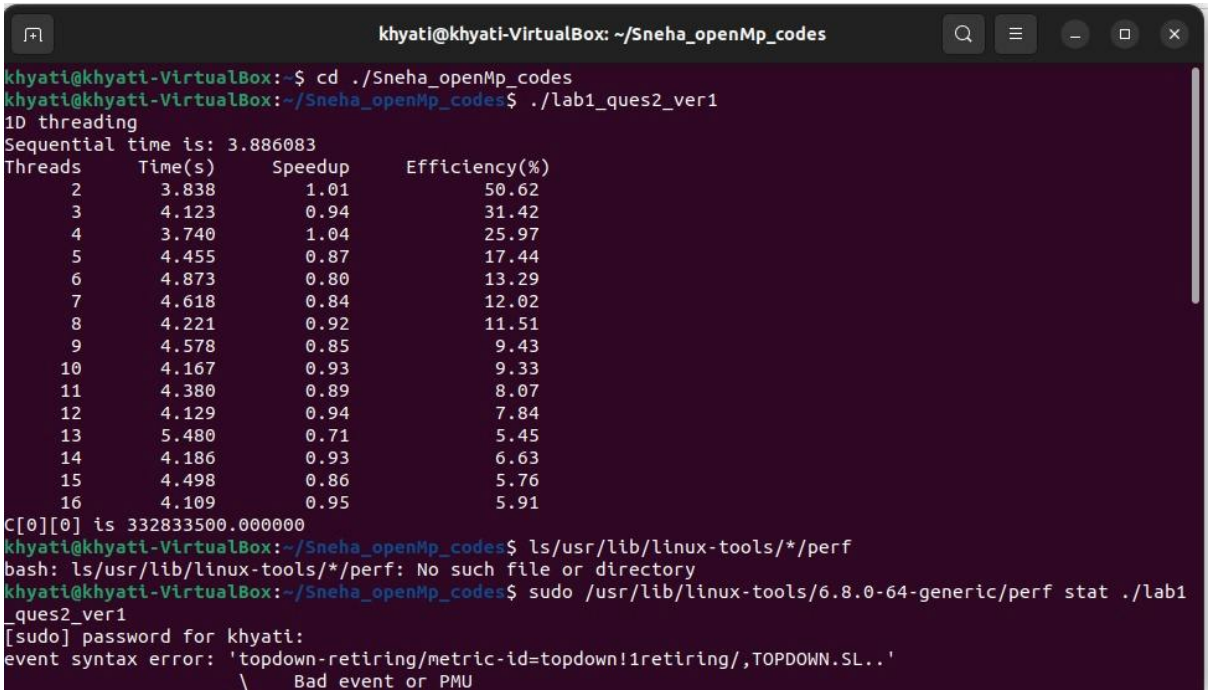
Comparing Version 1 vs Version 2

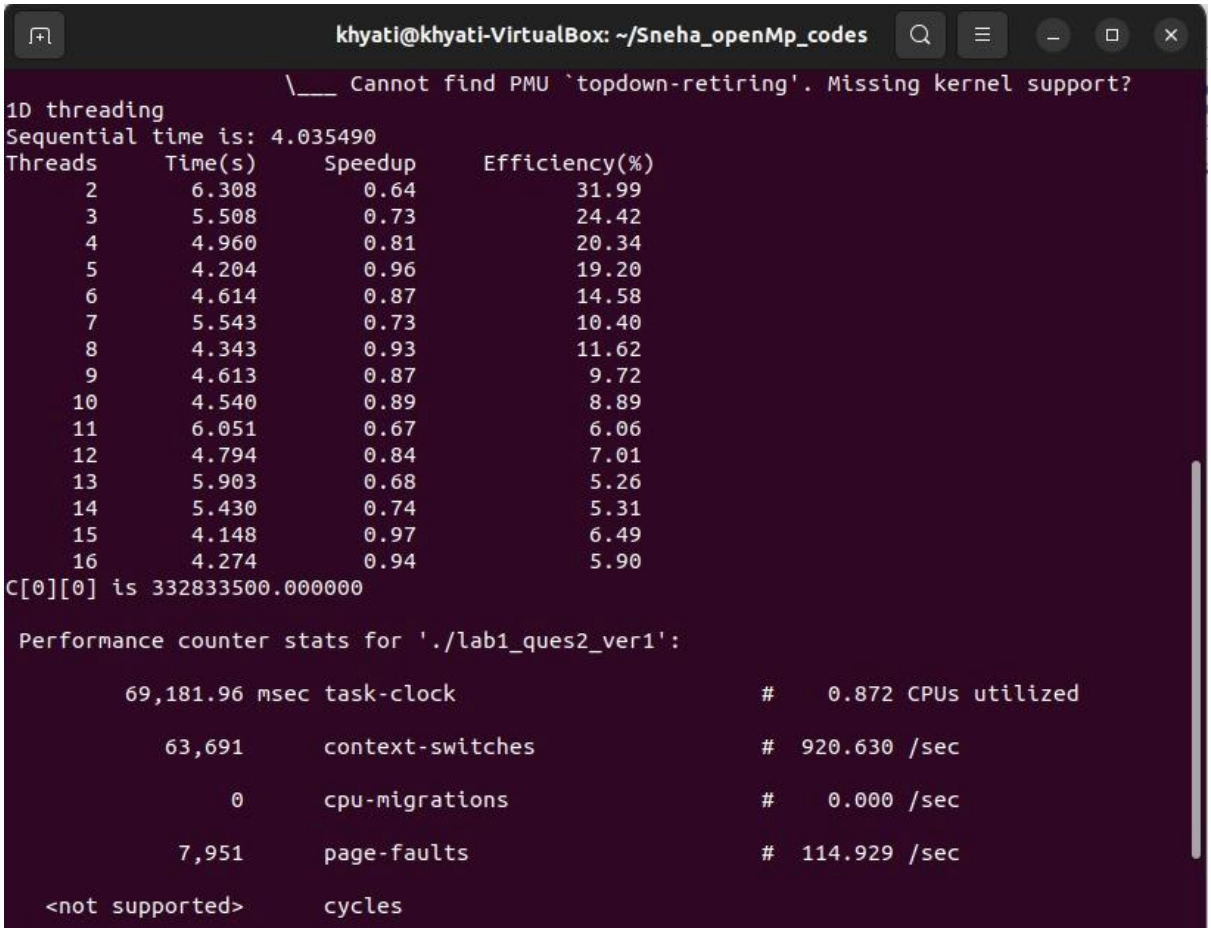
Version 1 was stable and consistent (0.84× to 1.04× speedup range), with slightly worse CPU utilization (0.872) and more context switches (63,691). Version 2 had better CPU utilization (0.927) and fewer context switches (58,518), suggesting better load balancing, but was very inconsistent (0.71× to 1.09× speedup range). Version 2 performed better at low thread counts (2-3 threads got speedup >1.0) but much worse at higher counts. Version 1 never achieved real speedup but was more predictable. Both hit the memory bandwidth wall.

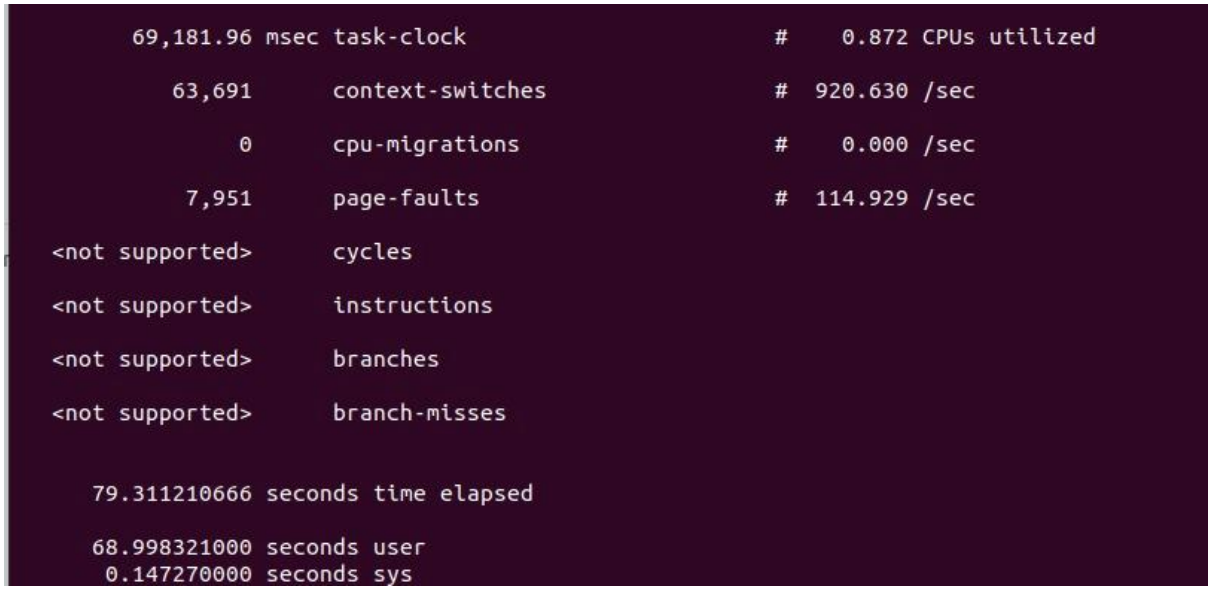
I'd pick Version 1 because stability matters. Even though Version 2 had a couple of better results, inconsistent performance is unreliable. Version 1 behaves predictably, which is important for real applications.

My output:

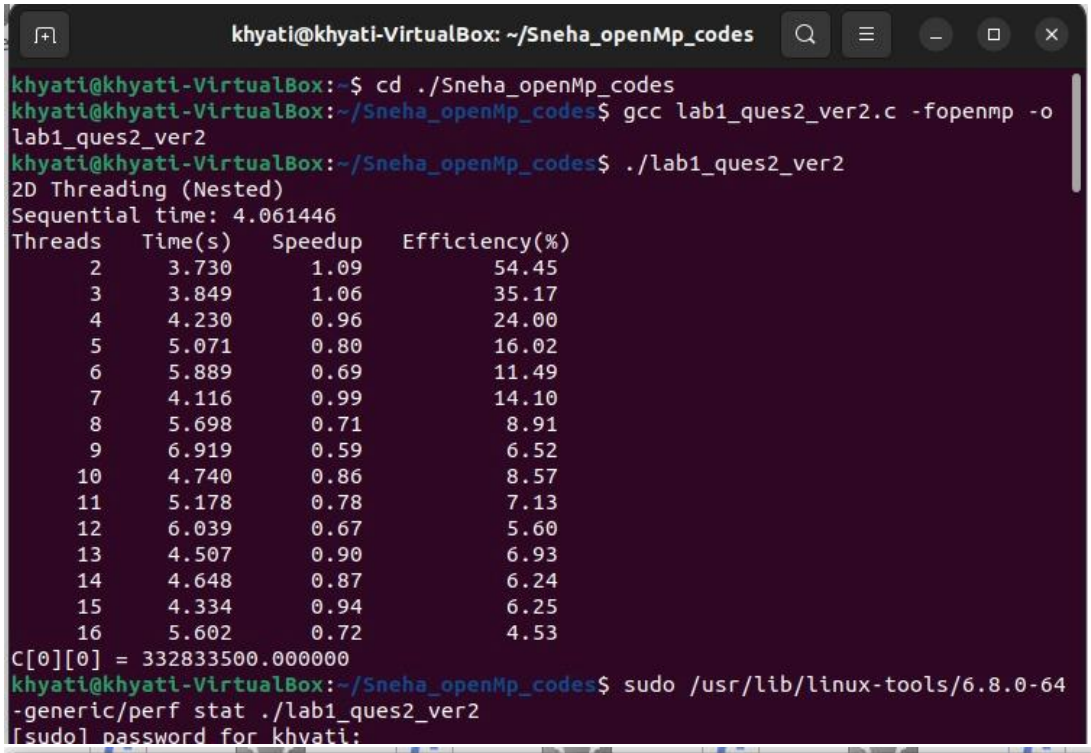
Version 1 terminal output as well as performance stats:

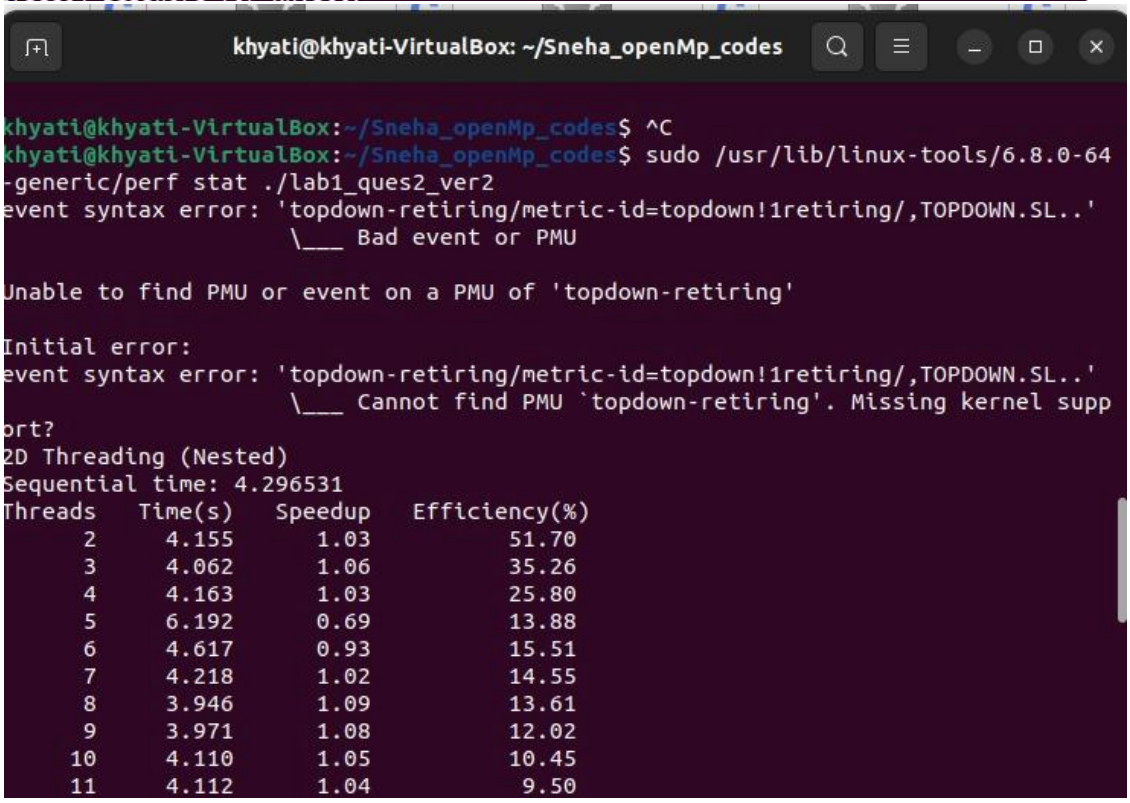


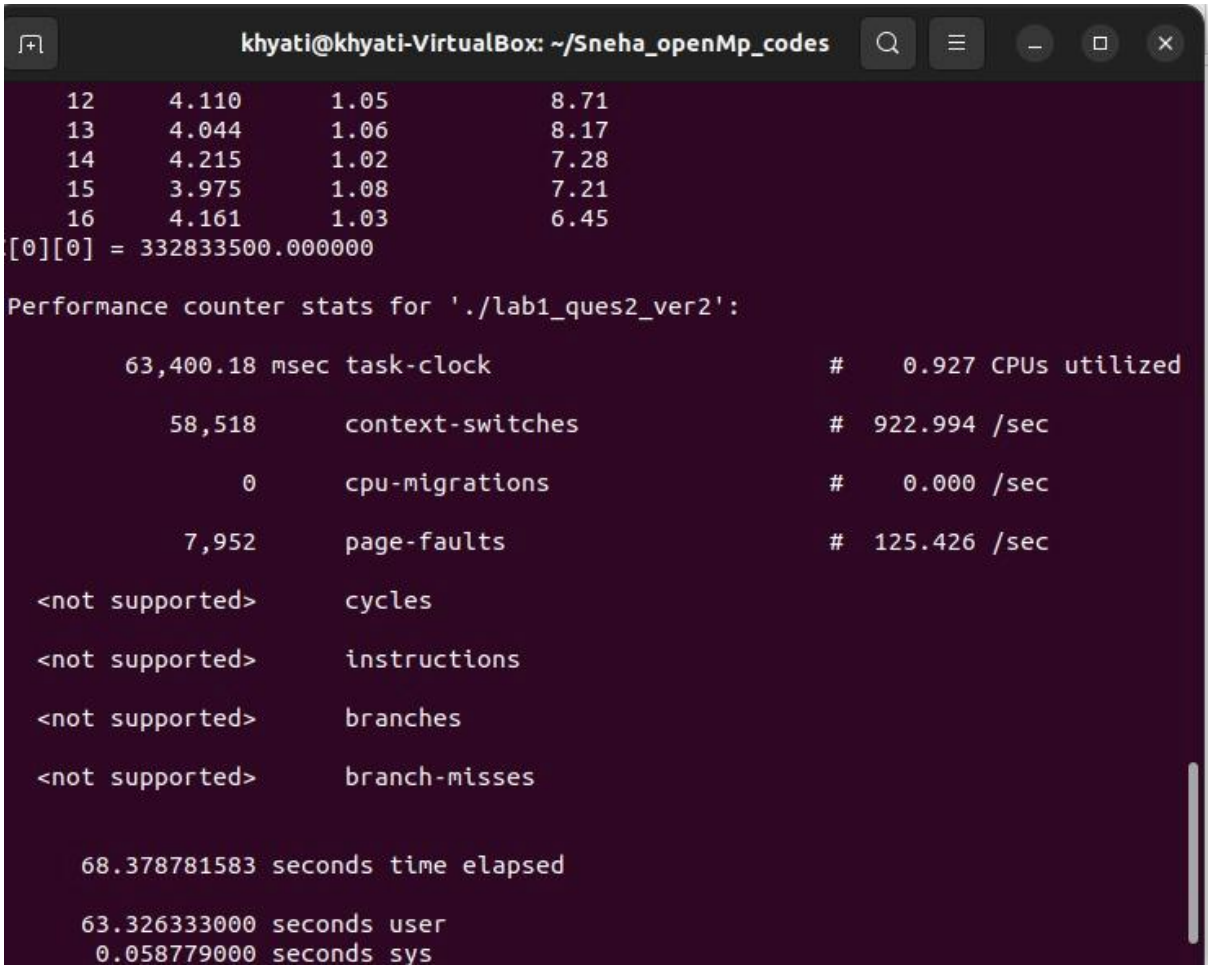

khyati@khyati-VirtualBox: ~/Sneha_openMp_codes
___ Cannot find PMU `topdown-retiring'. Missing kernel support?
1D threading
Sequential time is: 4.035490
Threads Time(s) Speedup Efficiency(%)
2 6.308 0.64 31.99
3 5.508 0.73 24.42
4 4.960 0.81 20.34
5 4.204 0.96 19.20
6 4.614 0.87 14.58
7 5.543 0.73 10.40
8 4.343 0.93 11.62
9 4.613 0.87 9.72
10 4.540 0.89 8.89
11 6.051 0.67 6.06
12 4.794 0.84 7.01
13 5.903 0.68 5.26
14 5.430 0.74 5.31
15 4.148 0.97 6.49
16 4.274 0.94 5.90
C[0][0] is 332833500.000000
Performance counter stats for './lab1_ques2_ver1':
69,181.96 msec task-clock # 0.872 CPUs utilized
63,691 context-switches # 920.630 /sec
0 cpu-migrations # 0.000 /sec
7,951 page-faults # 114.929 /sec
<not supported> cycles


69,181.96 msec task-clock # 0.872 CPUs utilized
63,691 context-switches # 920.630 /sec
0 cpu-migrations # 0.000 /sec
7,951 page-faults # 114.929 /sec
<not supported> cycles
<not supported> instructions
<not supported> branches
<not supported> branch-misses
79.311210666 seconds time elapsed
68.998321000 seconds user
0.147270000 seconds sys

Version 2:


khyati@khyati-VirtualBox: ~/Sneha_openMp_codes
khyati@khyati-VirtualBox:~$ cd ./Sneha_openMp_codes
khyati@khyati-VirtualBox:~/Sneha_openMp_codes$ gcc lab1_ques2_ver2.c -fopenmp -o lab1_ques2_ver2
khyati@khyati-VirtualBox:~/Sneha_openMp_codes$ ./lab1_ques2_ver2
2D Threading (Nested)
Sequential time: 4.061446
Threads Time(s) Speedup Efficiency(%)
2 3.730 1.09 54.45
3 3.849 1.06 35.17
4 4.230 0.96 24.00
5 5.071 0.80 16.02
6 5.889 0.69 11.49
7 4.116 0.99 14.10
8 5.698 0.71 8.91
9 6.919 0.59 6.52
10 4.740 0.86 8.57
11 5.178 0.78 7.13
12 6.039 0.67 5.60
13 4.507 0.90 6.93
14 4.648 0.87 6.24
15 4.334 0.94 6.25
16 5.602 0.72 4.53
C[0][0] = 332833500.000000
khyati@khyati-VirtualBox:~/Sneha_openMp_codes$ sudo /usr/lib/linux-tools/6.8.0-64-generic/perf stat ./lab1_ques2_ver2
[sudo] password for khyati:


khyati@khyati-VirtualBox: ~/Sneha_openMp_codes
khyati@khyati-VirtualBox:~/Sneha_openMp_codes$ ^C
khyati@khyati-VirtualBox:~/Sneha_openMp_codes$ sudo /usr/lib/linux-tools/6.8.0-64-generic/perf stat ./lab1_ques2_ver2
event syntax error: 'topdown-retiring/metric-id=topdown!1retiring/,TOPDOWN.SL..'
___ Bad event or PMU
Unable to find PMU or event on a PMU of 'topdown-retiring'
Initial error:
event syntax error: 'topdown-retiring/metric-id=topdown!1retiring/,TOPDOWN.SL..'
___ Cannot find PMU `topdown-retiring'. Missing kernel support?
2D Threading (Nested)
Sequential time: 4.296531
Threads Time(s) Speedup Efficiency(%)
2 4.155 1.03 51.70
3 4.062 1.06 35.26
4 4.163 1.03 25.80
5 6.192 0.69 13.88
6 4.617 0.93 15.51
7 4.218 1.02 14.55
8 3.946 1.09 13.61
9 3.971 1.08 12.02
10 4.110 1.05 10.45
11 4.112 1.04 9.50


khyati@khyati-VirtualBox: ~/Sneha_openMp_codes
12 4.110 1.05 8.71
13 4.044 1.06 8.17
14 4.215 1.02 7.28
15 3.975 1.08 7.21
16 4.161 1.03 6.45
[0][0] = 332833500.000000
Performance counter stats for './lab1_ques2_ver2':
63,400.18 msec task-clock # 0.927 CPUs utilized
58,518 context-switches # 922.994 /sec
0 cpu-migrations # 0.000 /sec
7,952 page-faults # 125.426 /sec
<not supported> cycles
<not supported> instructions
<not supported> branches
<not supported> branch-misses
68.378781583 seconds time elapsed
63.326333000 seconds user
0.058779000 seconds sys