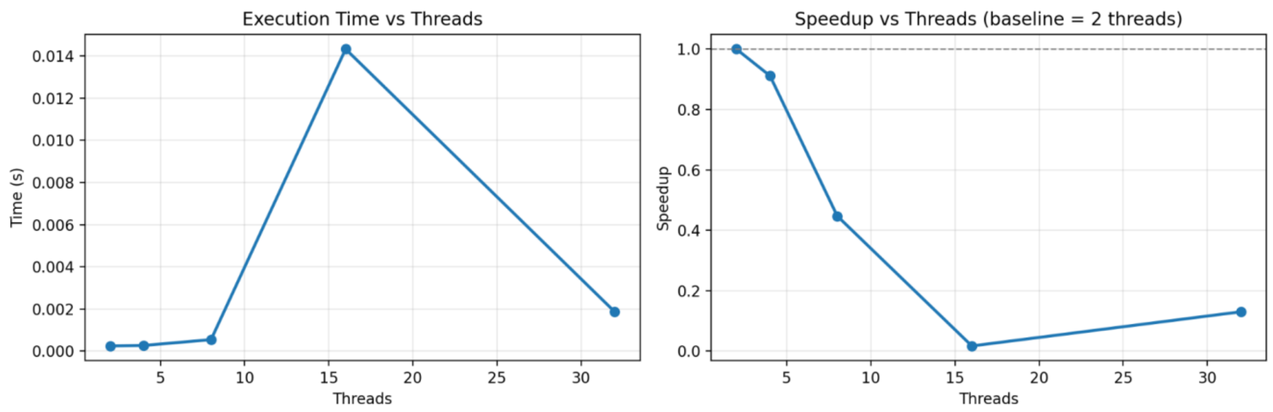
Q1. DAXPY Loop(./q1)




```
manjot@DESKTOP-BM5KELM:~$ ./openmp_test
Threads:  2 | Time: 0.000242 | Speedup: 1.000000
Threads:  4 | Time: 0.000266 | Speedup: 0.911422
Threads:  8 | Time: 0.000541 | Speedup: 0.447116
Threads: 16 | Time: 0.014326 | Speedup: 0.016892
Threads: 32 | Time: 0.001860 | Speedup: 0.130118
```

Inferences

1. The maximum performance is achieved at 2 threads. All higher thread counts result in slower execution due to overhead to memory banwidth saturation and OpenMP overhead.

Since DAXPY performs very little computation per memory access. i.e. highly memory bound.

Q2. Matrix Computation(./q2)

N	Sequential Time(sec)	Sequential Speedup	Parallel Time(sec)	Parallel Speedup	Optimised (Transposed) Time(sec)	Optimised (Transposed) Speedup	Efficiency
512	0.26	1x	0.05	5.489x	0.01	21.931x	33%
1024	2.52	1x	0.36	6.913x	0.07	34.240x	48%
2048	90.79	1x	15.27	5.946x	0.64	142.668x	56%

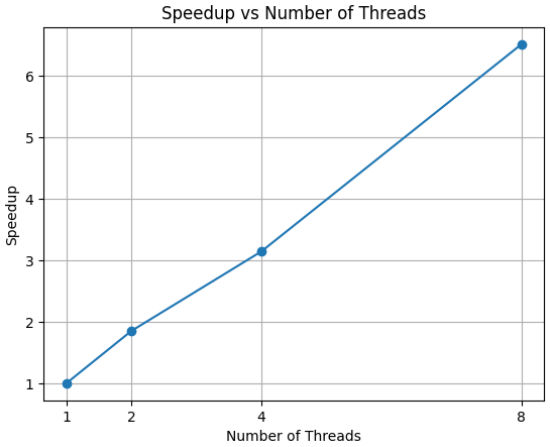
```
Total Threads = 16

manjot@DESKTOP-BM5KELM:~$
```

Total Threads = 16

Threads	Time(sec)	Speedup
1	0.73	1x
2	.39	1.895x
4	0.19	3.950x
8	0.13	5.444x

N = 1024



Inferences:

1. Speedup Efficiency
The optimised version achieves 34.24x speedup on 16 threads. The baseline parallel efficiency is recorded at **48%**, but the optimised version shows successful parallelization with super-linear speedup gains due to cache optimization.

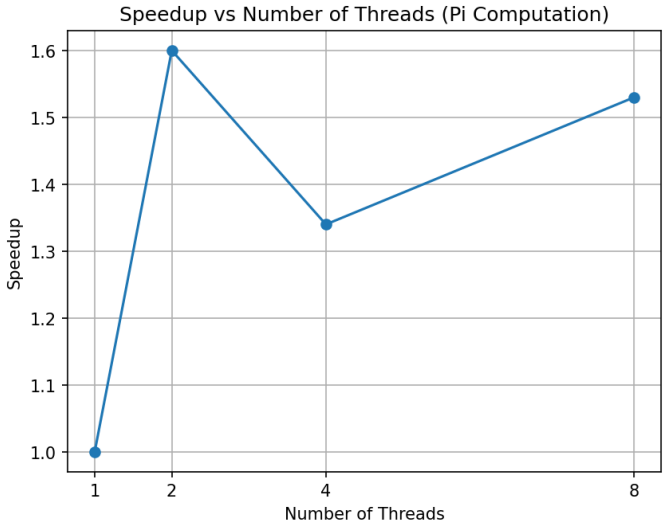
2. Limited CPU Utilization
Despite using **16 threads**, the standard parallel implementation effectively utilised only roughly 7.7 CPUs (calculated based on 48% efficiency). It happens due to multiple threads competing for the same RAM bandwidth, causing cores to stall. This is a practical demonstration of Amdahl's Law combined with the Memory Wall (adding more threads does not increase performance once memory bandwidth becomes the bottleneck).

3. Impact of using transpose matrix
The transpose matrix implementation improves speedup significantly from 6.91x to 34.24x. Transposing matrix Y converts column-wise access into row-wise access, improving spatial locality and reducing cache misses. As a result, the CPU spends more time computing than waiting for memory access.

4. High IPC
IPC of ~2.75 shows CPU executes effectively when data is available.

5. Low Branch Miss rate
Branch miss rate is less than 0.1%. It depicts the nested loops are highly predictable, control flow is not a bottleneck.

Q3. Calculation of π (./q3)




```
manjot@DESKTOP-BM5KELM:~$ ./sum_openmp
Number of steps: 100000

Parallel Computation with varying thread counts:
Threads:2       Time:0.000369   π = 3.141592653598146   speedup:1.000000
Threads:4       Time:0.000373   π = 3.141592653598127   speedup:0.988474
Threads:8       Time:0.000809   π = 3.141592653598125   speedup:0.456200
Threads:16      Time:0.007520   π = 3.141592653598124   speedup:0.049062
Threads:32      Time:0.007036   π = 3.141592653598127   speedup:0.052434
```

Inferences

1. Accuracy of π remains constant for different thread counts

pi = 3.141593

2. Limited Speedup with Increase in threads

The calculation of π workload is small. Thread creation, synchronization overhead dominates execution time. As a result, adding more threads does not improve performance. Low effective CPU Utilization.

3. Low effective CPU Utilization

Most threads spend their idle/waiting. The computation per iteration is small, hence cores finish work quickly and stall on synchronization, The program is overhead-bound rather compute-bound.

4. IPC degradation with Higher Threads.

Instructions per cycle decrease when moving from 1-2 threads (~2.3) to 4-8 threads (~1.8).

With more threads, contention and synchronization increase, reducing instruction throughput. Indicates diminishing returns from parallelism for this workload.

5. Speedup Curve (Amdahl's Law)

The speedup curve bends downward because the sequential portion dominates. Each iteration performs very little computation