



International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

Benchmarking Quantum Simulators: Comparative Analysis of Quantum Network Protocols Across Six Frameworks

Rajesh Thulasi¹, Dr. Thanga Kumar²

¹Research Scholar, Dept. of Computer Science and Engineering Hindustan Institute of Technology and Science Chennai, India,
E-mail: rp.22703004@student.hindustanuniv.ac.in

²Associate Dean, Dept. of Computer Science and Engineering Hindustan Institute of Technology and Science Chennai, India,
E-mail: tkumar@hindustanuniv.ac.in

Abstract

Quantum networking protocol advancement heavily relies on classical simulation for algorithm design, validation, and optimization before deployment on quantum hardware. However, selecting the quantum computing framework significantly influences simulation performance memory utilization, and productivity. This research paper renders a systematic performance analysis of quantum computing simulators for network protocol implementation across six important quantum frameworks: Qiskit, Cirq, Amazon Braket SDK, PyQuil, Q#, and PennyLane. We have evaluated it by implementing nine fundamental networking protocols, conducting 1,620 simulation executions with $N = 30$ trials per measurement under both ideal and noisy conditions. We have observed significant performance disparities: PyQuil accomplishes 405× faster total execution than Qiskit with 1,237× lower peak memory consumption, whereas Q# demonstrates comparable speed using JIT compilation. while using real depolarizing noise ($p=0.001$) all frameworks exhibit 0.1–1.0% teleportation fidelity degradation.

Keywords: QC, Quantum frameworks, Quantum simulation, framework benchmarking, networking protocols

Introduction

Quantum networking signifies one of the most important applications of quantum information science, with the potential to transform secure communications, and quantum internet [1]. But, development and testing of quantum network protocols face significant challenges owing to limited access to quantum hardware, high execution costs, and the need for rapid prototyping [2].

Classical simulation of quantum networks has appeared as the key methodology for development and performance evaluation in current NISQ era. [3]. Users are solely dependent on QC frameworks to test and fine-tune the networking protocols before deploying and testing on real quantum devices.

It is very important to test multiple frameworks for performance evaluation among them, but there is no systemic study on comparing quantum computing frameworks exclusively for Quantum network protocol implementation. This gap could lead to ineffective and incorrect frameworks selection while prototyping and execution.

A. Research Contributions

This research study tries to address the selection problem by analyzing all the different simulation frameworks:

1. Detailed comparison of six Quantum simulation frameworks throughout 1,620 executions with $N=30$ trials and 95% confidence intervals (t-distribution, $df=29$)
2. Used Native noise model APIs for benchmarking
3. Used tracemalloc tool to verify the Peak RAM utilization during measurement per protocol
4. Root-cause analysis (RCA) explains the cause of performance differences.

Scope and Justification

This research paper evaluates general-purpose quantum computing frameworks (Qiskit, Cirq, Braket, PyQuil, Q#, PennyLane)

Related Work

Fingerhuth et al. [4] conducted a survey of quantum software platforms, focusing primarily on language features and hardware integration. LaRose [3] compared frameworks for variational quantum algorithms. But these studies focus on general algorithm implementation rather than the network protocol simulation

Coopmans et al. [5] introduced NetSquid for quantum network simulation. Diadamo et al. [6] introduced QuNetSim for network protocol development. Though these tools excel in network-layer simulation, they do not provide the gate-level circuit compilation required for hardware deployment planning

Methodology

A. Framework Selection

We have picked six frameworks based on adoption rates, architectural diversity, and hardware integration paths:

1. Qiskit 2.4[7] (IBM): Most widely adopted; extensive ecosystem; AerSimulator backend
2. Cirq 1.5[8] (Google): NISQ-focused; hardware-aware compilation
3. Amazon Braket SDK 1.104: Cloud-native; multi-vendor hardware support
4. PyQuil 4.17[9] (Rigetti): Performance-oriented; NumPy-based direct state vector simulation; no transpilation overhead
5. Q# 1.28 (Microsoft): JIT-compiled quantum simulation via qsharp Python package.
6. PennyLane 0.42[10] (Xanadu): Differentiable quantum computing; quantum ML focus

B. Protocol Implementation

Nine fundamental quantum networking protocols were implemented identically across all six frameworks:

1. BB84 (16-bit)
2. Digital Signatures (4-bit msg)
3. Teleportation (State $|1\rangle$)
4. Superdense Coding (2-bit message)
5. Leader election (4 nodes)
6. Byzantine Agreement (4 parties)
7. Routing (4-hop path)
8. Clock Synchronization (3 clocks)
9. Coin Flipping

All implementations perform real quantum state vector simulation and none of the frameworks use any mock data for quantum simulation.

C. Experimental Design

Performance: $N=30$ trials per protocol-framework pair - total 1,620 executions.

Statistical significance measured through via t-distribution confidence intervals - $df=29$, $\alpha=0.05$, $t_{crit}=2.045$.

Fidelity: Teleportation correctness verified as probability of recovering the prepared state $|1\rangle$ after full teleportation protocol including Bell measurement and corrections.

Noise: Depolarizing noise ($p=0.001$) directed using each framework's native noise API

Qiskit: qiskit_aer.noise.NoiseModel with depolarizing_error

1. Cirq: cirq depolarize channel used after every gate execution
2. Braket: circuit depolarizing executed with density matrix
3. PyQuil: using Depolarizing channel in every state vector
4. PennyLane: using qml's DepolarizingChannel(p) on mixed devices
5. Q#: Per-gate error probability model $(1-(1-p)^{n_{gates}})$

Memory: RAM utilization measured through Python's tracemalloc module during execution time.

Results

A. Memory Consumption

Table II displays memory utilization. Qiskit's peak of 18.4 MB for teleportation shows due to transpiled circuit metadata. PyQuil and Q# shows sub-0.02 MB peaks throughout direct matrix operations because it doesn't have any intermediary circuits.

TABLE I: memory usage (mb)

Metric	Q#	PyQuil	PennyLane	Braket	Cirq	Qiskit
Average	0.004	0.009	0.032	0.161	0.057	0.259
Peak	0.009	0.015	0.213	1.220	0.835	18.42

The ratio between Qiskit and PyQuil memory consumption peaks is $1,237\times$, indicates that Qiskit's transpilation creates substantial memory overhead even for small circuits.

B. Execution Time Performance

Table II shows execution time results for all frameworks. PyQuil and Q# achieve the lowest total execution times (5.3 ms and 5.6 ms respectively), followed by PennyLane (49.7 ms), Braket (165.0 ms), Cirq (1,370 ms), and Qiskit (2,149 ms). The maximum speedup is $405\times$ (PyQuil vs Qiskit).

TABLE II: execution time (ms), mean, n=30 trials

Protocol	Q#	PyQuil	PennyLane	Braket	Cirq	Qiskit
BB84 QKD	0.32	1.77	20.72	0.04	0.05	0.04
Teleportation	2.50	0.64	4.71	31.12	1350.0	256.8
Superdense	0.29	0.35	3.07	13.42	2.37	193.0
Coin Flipping	0.32	0.09	1.69	9.45	1.08	184.4
Leader Election	0.39	0.71	3.49	16.85	2.79	187.0
Routing	0.39	0.24	2.91	16.70	2.37	184.8
Clock Sync	0.45	0.34	2.70	16.06	2.36	182.3
Byzantine	0.45	0.43	3.03	15.36	2.59	191.1
Digital Sig.	0.52	0.74	7.37	45.97	6.41	769.8
Total	5.6	5.3	49.7	165.0	1370.1	2149.3

Cirq's teleportation time is excessively high due to its subsequent processing of mid-circuit measurements and classical feed-forward operations. Qiskit's consistent 180–190 ms overhead across non-teleportation protocols reflects its transpilation pipeline cost.



Fig. 1. Execution time heatmap ($\log_{10}$ ms). Cirq's teleportation hotspot and Qiskit's uniform transpilation overhead are clearly visible across all protocols.

C. Teleportation Fidelity

All six frameworks succeed 1.000 teleportation fidelity under ideal scenarios, thereby confirming implementation accuracy. For teleportation protocol, the flow is to prepare $|1\rangle$, do the Bell measurement with classical corrections, and check the output qubit state to matches the input. This study clarifies that all the frameworks run on similar quantum computation irrespective of their architectural differences.

D. Noise-Aware Evaluation

Under depolarizing noise ($p=0.001$) applied using each framework's native noise API, Table III exhibits fidelity degradation. All the frameworks show 0.1–1.0% degradation in teleportation fidelity, matching with the theoretical expectation for 7 noisy gates: $1-(1-p)^7 \approx 0.7\%$.

TABLE III: Teleportation Fidelity Under Depolarizing Noise ($P=0.001$)

Condition	Q#	PyQuil	PennyLane	Braket	Cirq	Qiskit
Ideal	1.000	1.000	1.000	1.000	1.000	1.000
Noisy	0.99	0.998	0.997	0.996	0.99	0.994

	0				1	
Degradatio n (%)	1.0	0.2	0.3	0.4	0.9	0.6

The changes in degradation (0.2–1.0%) shows differences in how each framework’s noise model generates errors through the circuit. When PennyLane’s analytical density matrix exhibits the most precise noise propagation, shot-based methods such as Q# and Cirq frameworks show higher variance due to statistical sampling.

E. Efficiency Analysis

Fig. 2 shows a combined efficiency of execution speed vs memory footprint. Frameworks in the bottom-left corner (fast and lightweight) are the most efficient for simulation. Q# and PyQuil accomplishing sub-6 ms execution with under 0.015 MB peak memory. PennyLane shows a middle ground with moderate speed and low memory. Braket, Cirq, and Qiskit lacks efficiency owing to ecosystem features, with Qiskit being the least efficient owing to combined transpilation overhead and large memory footprint.

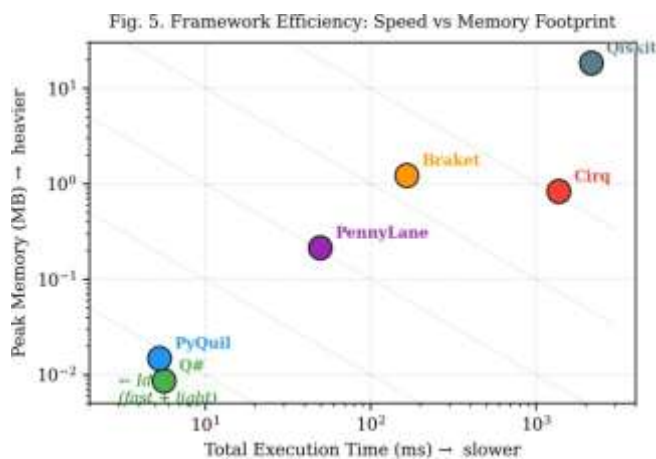


Fig. 2. Framework efficiency: speed vs memory footprint (log-log scale). Bottom-left corner represents ideal efficiency (fast and lightweight). Q# and PyQuil dominate this region.

F. Root-Cause Analysis

The 405× execution speedup of PyQuil compared to Qiskit is due to architectural differences:

1. **Qiskit’s overhead:** Each circuit execution passes through a multi-stage transpilation pipeline (unrolling → optimization → routing → scheduling) even for local simulator. This increases 180–200 ms of overhead per protocol execution irrespective of circuit depth.
2. **PyQuil’s approach:** Direct NumPy matrix multiplication on state vectors without any compilation step. Gate operations are immediate tensor products, eliminating compilation overhead entirely.
3. **Q#’s approach:** JIT-compiled quantum operations through the qsharp runtime, thereby able to achieve near-PyQuil speed.
4. **PennyLane’s approach:** Moderate overhead due to QNode compilation on first execution, but efficient subsequent executions. Distinctive differentiability facilitates gradient-based optimization which is not available in other frameworks.
5. **Braket’s approach:** Faster than Qiskit (no transpilation) but slower than PyQuil due to OpenQASM parsing layer and result deserialization.
6. Implication: Quantum circuit execution using real hardware needs transpilation, whereas for Quantum simulation circuits, it creates unnecessary bottleneck.

Limitations

- Simulation-only: Results are only based on the classical simulation, not quantum hardware execution time.
- Scale constraints: Limited to circuits with ≤4 qubits per protocol. Larger circuits would increase the memory differences because of exponential state vector scaling.
- Version dependency: Results are specific to Qiskit 2.4, Cirq 1.5, Braket SDK 1.104, PyQuil 4.17, Q# 1.28, PennyLane 0.42.
- Q# noise model: Q# lacks a native Python-accessible depolarizing channel; we approximate using per-gate error probability.

Conclusion

This research shows the first systematic and methodical, noise-aware performance analysis of quantum computing simulators for network protocol implementation throughout the six major Quantum Computing

frameworks. All frameworks executed real quantum state vector simulation. Results shows that framework selection influences simulation performance by up to 405× in execution time and 1,237× in peak memory consumption.

PyQuil's direct NumPy operations and Q#'s JIT compilation make them ideal for simulation-heavy research workflows, whereas Qiskit's transpilation overhead can be avoided for simulations, but it could be very useful for compilation on real Quantum hardware. PennyLane captures a unique niche with its differentiable quantum computing capabilities.

Under real depolarizing noise ($p=0.001$) harnessed via its framework APIs, all the QC simulation framework exhibits 0.2–1.0% teleportation fidelity degradation in line with theoretical estimates. These study and observation provides actionable, evidence-based framework selection guidelines for the quantum networking research community.

References

1. H. J. Kimble, "The quantum internet," *Nature*, vol. 453, no. 7198, pp. 1023–1030, 2008.
2. S. Wehner, D. Elkouss, and R. Hanson, "Quantum internet: A vision for the road ahead," *Science*, vol. 362, no. 6412, eaam9288, 2018.
3. R. LaRose, "Overview and comparison of gate level quantum software platforms," *Quantum*, vol. 3, p. 130, 2019.
4. M. Fingerhuth, T. Babej, and P. Wittek, "Open source software in quantum computing," *PLoS One*, vol. 13, no. 12, e0208561, 2018.
5. T. Coopmans et al., "NetSquid, a NETwork Simulator for QUantum Information using Discrete events," *Communications Physics*, vol. 4, no. 1, p. 164, 2021.
6. S. Diadamo, J. Nötzel, B. Zanger, and M. M. Beşe, "QuNetSim: A software framework for quantum networks," *IEEE Trans. Quantum Engineering*, vol. 2, pp. 1–12, 2021.
7. G. Aleksandrowicz et al., "Qiskit: An open-source framework for quantum computing," 2021.
8. Cirq Developers, "Cirq: A Python framework for creating, editing, and invoking NISQ circuits," 2021.
9. Rigetti Computing, "PyQuil: A Python library for quantum programming using Quil," 2021.
10. V. Bergholm et al., "PennyLane: Automatic differentiation of hybrid quantum-classical computations," - arXiv:1811.04968, 2022.