Seminar:
Scalable Computing Systems and Applications in AI, Big Data and HPC

Report on
Scalable Quantum Computer Simulation on HPC Systems

Ughur Mammadzada
ughur.mammadzada@stud.uni-goettingen.de
Universität Göttingen

Supervisor: Tino Meisel

30 September 2024

Contents

1 Introduction **2**

2 Background **2**

2.1 Quantum Computers 2

2.2 Simulation 2

2.3 Schrödinger's method 2

3 Simulators **3**

3.1 Intel Quantum Simulator (IQS) 3

3.2 Quantum Exact Simulation Toolkit - QuEST 3

3.3 IBM Qiskit 3

3.4 Qibo 3

3.5 Qulacs 3

3.6 Cirq 3

3.7 Quantum Toolbox in Python - QuTiP 3

3.8 Overview 4

4 Benchmarking **4**

5 Methods **5**

5.1 Process 5

5.2 Python Script 6

5.3 Environment 7

6 Evaluation **7**

7 Discussion **10**

8 Conclusion **11**

A Appendix **11**

A.1 Slurm Submissions 11

1 Introduction

This report covers practical project work conducted during Summer Semester 2024 at the University of Goettingen in the context of the seminar "Scalable Computing Systems and Applications in AI, Big Data and HPC".

The report will cover the basics of quantum computers / machines (QC or QM) and explain the theories behind their simulation on classical computers. More detailed attention will be given to Schrödinger's method of simulation. After covering those principles the report will cover some of the common Schrödinger style simulators.

The report will also cover standardized ways of benchmarking QC. After that, the report will explain the differences in benchmarking QCs and QSs.

Next chapters will cover the practical project's plan, the benchmarking principle, the executed work and the environment of execution. This will be followed by observed results and conclusions made from those observations.

2 Background

2.1 Quantum Computers

Quantum computing is a type of computing that takes advantage of the principles of quantum mechanics, such as superposition, entanglement and quantum interference. Unlike classical computers, which use bits (0 or 1) as the smallest unit of information, quantum computers use qubits. A qubit can represent a 0, 1, or both 0 and 1 simultaneously, thanks to the principle of superposition. This state is maintained during the calculation to let the calculation consider all possibilities [1].

In addition to superposition, qubits can also be entangled. When qubits are entangled, the state of one qubit is directly related to the state of another, no matter the distance between them.

These features allow QCs to process calculations much faster than classical computers, to the point where until recently, all existing encryption methods were considered useless in a QC world, as QCs can brute-force the decryption key in practical amounts of time. In comparison classical computer would take time that orders and orders of magnitude bigger than the currently agreed age of the universe.

2.2 Simulation

While QCs are so exciting, they are expensive and hard to build, maintain and operate for a variety of reasons, including cooling and noise protection.

Although these problems will most likely be solved in the future, one thing we've learned from history is that waiting for hardware to develop software is not optimal. That is one of the main reasons for QSs to exist.

Besides helping the engineers to develop QCs, such simulators help developers to write software for the hardware that doesn't exists yet or is unavailable.

There are 3 main theories of quantum computer simulations:

- Schrödinger's method
- Feynman's method
- Heisenberg's method

2.3 Schrödinger's method

Schrödinger's method, also known as linear-algebraic method, is one of the most common approaches to simulating quantum systems on classical computers. It is based on solving the wave function of the quantum system, which describes the probabilities of different states the system can be in. In this method, the quantum state of a system of n qubits is represented as a vector in a 2^n-dimensional complex vector space. The evolution of this quantum state is computed by applying unitary transformations, known as quantum gates, to the state vector.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

The Equation (1) shows the representation of a single qubit. Here α and β are complex numbers, and $|\alpha|^2 + |\beta|^2 = 1$, meaning that the total probability is 1. In vector form, a single qubit can be represented as Equation2:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix} \tag{2}$$

and for 2 qubits (3):

$$\begin{aligned}|\psi\rangle &= (\alpha_0|0\rangle + \beta_0|1\rangle) \otimes (\alpha_1|0\rangle + \beta_1|1\rangle) = \\ &= v_{00}|00\rangle + v_{01}|01\rangle + v_{10}|10\rangle + v_{11}|1\rangle = \begin{bmatrix}v_{00}\\ v_{01}\\ v_{10}\\ v_{11}\end{bmatrix}\end{aligned} \tag{3}$$

and hence 2^n. If the qubit is at state $|0\rangle$ the vector will be $\begin{bmatrix}1\\ 0\end{bmatrix}$ meaning probability 1 for 0 and probability 0 for 1.

As more qubits are added, the size of the state vector grows exponentially, leading to a computational cost.

Simulating a system of 10 qubits requires handling a vector with $2^{10} = 1024$ complex entries.

While qubits are represented as vectors, quantum gates are represented as matrices. For example the Hadamard gate, which is used to put the qubit in a superposition looks like this (4) [2]:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4}$$

when applied to a qubit in state $|0\rangle$ will:

$$\begin{aligned} H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \\ = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \end{aligned} \tag{5}$$

With n increasing, the matrices will have dimensions $2^n \times 2^n$. This method is most intuitive and accurate, however due to stated properties it is computationally expensive. That is precisely why it is the most fitting model in the context of High Performance Computing (HPC).

3 Simulators

3.1 Intel Quantum Simulator (IQS)

The Intel Quantum Simulator (IQS)[3] is a high-performance simulation tool designed to emulate quantum circuits based on Schrödinger's method. IQS is highly scalable, enabling both single-instance simulations for large qubit counts and parallel simulations for multiple quantum states or noisy systems. It supports multi-core and multi-node modes out of the box. Furthermore, it supports GPU acceleration. IQS has both C++ and Python interfaces, making it a versatile tool for researchers in quantum computing.

3.2 Quantum Exact Simulation Toolkit - QuEST

The Quantum Exact Simulation Toolkit (QuEST)[4] is an open-source, high-performance quantum circuit simulator (C library) that supports parralelism out of the box. It is designed to run on various platforms, from personal laptops to supercomputers, using a hybrid parallelization strategy with OpenMP and MPI, and it also supports GPU acceleration.

3.3 IBM Qiskit

IBM's Qiskit[5] is an open-source tool for quantum computing, designed to build and run quantum circuits on IBM's quantum computers as well as on simulators. With Qiskit users can build quantum circuits and run them on a backend of their choice, including Qiskit's simulator Qiskit Aer. It has modules (Aer is one of them), which also include support of HPC. However Qiskit itself is not designed for HPC as it is in origin just a frontend to build and send a curcuit to a backend.

3.4 Qibo

Qibo[6] is an open-source framework for quantum simulation, designed to rely on hardware acceleration, particularly through GPUs, to boost the performance of quantum circuit simulations. It uses TensorFlow as its backend, but the backend can be customized, such as CUDA for GPU utilization. Qibo allows for both single- and multi-GPU computations.

3.5 Qulacs

Qulacs[7] is a high-performance quantum circuit simulator designed for fast simulations of quantum algorithms on both classical hardware and specialized systems like GPUs. Written in C++ and with a Python interface, Qulacs is optimized for flexibility and speed, making it suitable for research in quantum computing, especially in simulating large quantum circuits efficiently. It supports both dense and sparse matrices, which enables simulations with varying levels of precision and complexity.

3.6 Cirq

Cirq[8] is an open-source Python framework developed by Google as part of its Quantum AI initiative, aimed at building, simulating, and executing quantum circuits, particularly for Noisy Intermediate-Scale Quantum (NISQ) devices. It allows researchers to design quantum circuits with a variety of gates, simulate these circuits on classical computers, and test quantum algorithms in noisy environments, reflecting real-world quantum hardware conditions. Cirq is well-integrated with Google's quantum processors and can also support other quantum backends, making it flexible for researchers exploring quantum algorithms and hardware.

3.7 Quantum Toolbox in Python - QuTiP

The Quantum Toolbox in Python (QuTiP)[9, 10] is an open-source, high-performance software framework designed for simulating the dynamics of open quantum systems. QuTiP allows researchers to explore quantum phenomena such as dissipation, decoherence, and quantum state evolution. It supports simulations of both closed and open quantum systems, using Schrödinger and Lindblad formalisms[11], respectively.

QuTiP is optimized for performance with support for parallelization and can run on various platforms, including workstations and high-performance computing environments. The software also offers an extensive library of quantum objects, operations, and solvers, which makes it a popular choice for research in quantum optics, quantum information, and quantum control.

3.8 Overview

There are many more simulators such as Pennylane[12], ProjectQ[13] and many more. Most of them use the same mathematical rules based on Schrödinger's method. Many of the listed simulators are in their origin designed to be a frontend for a QC. Actually, one of the advantages of Qulacs according to the developers is that it was not designed to be an interface, but the complete simulator.

There are a lot of simulators and the main property that make one better than another is not the simulation accuracy and modeling, but the functional calls, data distribution, wall clock performance. The computations themselves are deterministic. The goal is to optimize the way workload is distributed on a cluster (if execution on cluster is supported).

In this project 5 simulators with Python interface were used:

- Cirq
- Qibo
- Qiskit
- Qulacs
- QuTiP

4 Benchmarking

The current state of the art method of benchmarking QMs is Qunatum Volume (QV) metric. Developed by IBM, QV measures how well a quantum computer can handle increasingly complex circuits by looking at factors like the number of qubits, gate quality, connectivity between qubits, and error rates. A higher QV means the machine can run larger and deeper circuits before errors take over.

To calculate QV, a quantum computer runs randomized circuits of different sizes, and the results are checked against ideal, error-free outputs. The largest circuit size where the machine produces reliable results gives the QV score. This method is useful because it provides a broad measure of the system's overall performance.

During the benchmark process the generated circuits are always square circuits, meaning number of qubits and layers of gates (depth) is equal.

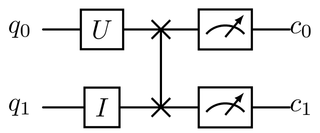

Figure 1: A square quantum circuit with 2 qubits and 2 layers. The first layer consists of U and I gates, the second layer is a SWAP gate, and the third layer includes measurements on both qubits with outputs c_0 and c_1.

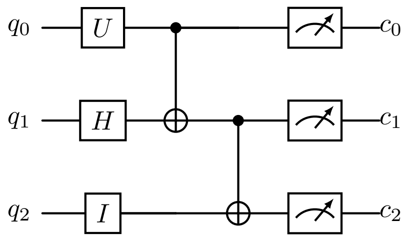

Figure 2: A square quantum circuit with 3 qubits and 3 layers. The first layer consists of U, H, and I gates. The second layer includes CNOT gate between q_0-q_1. The third layer includes CNOT between q_1-q_2. The measurements are stored as c_0, c_1 and c_2

With this benchmark if a QM can run a circuit of size 4×4 that means the QM has QV of $2^4 = 16$.

However, while this benchmark is useful for QMs it is not designed for simulators. The main problem is that in case of a QM at some point it will fail to do the computations right, while the simulator is always deterministic and theoretically the number of qubits and the depth can be increased indefinitely. The only constrain is really just the computation cost.

The reason QV benchmark exists is to understand how reliable are the results of the computation done by a QM. In case of QS they are reliable all the time (unless noise is added on purpose).

That is why we cannot use QV to benchmark the simulator directly. To understand how to benchmark a QS first we need to understand the purpose of the simulator and what are the requirements it has to met. One of such requirements is to be fast. In this project the computation performance will be main focus of benchmarking. QV benchmarking will be a proxy to benchmark the simulators. This means that the benchmark will not be evaluating QV of the QS but instead running QV benchmark and measuring how fast QS can complete the calculations.

5 Methods

5.1 Process

Initially, the goal of the project was to benchmark QSs. QV benchmark which is used for QCs was inspected and was initially chosen as the benchmark. The first codebase was built around running the circuits in different simulators and test their accuracy as the quantum circuits become more complex.

To measure the accuracy of the simulators the entopy of their outputs was calculated. However during the inspection of the result data it was discovered that the simulators in terms of their accuracy are almost identical (Fig. 3):

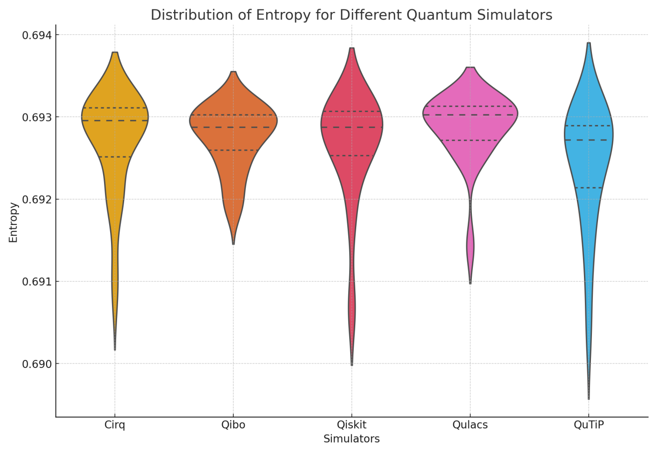

Figure 3: Entropy of different simulators

Additionally, the computations are heavily depended on the backend used and most "simulators" are just interfaces for different backends and the user can switch the backends.

After those discoveries it was decided to fall back on more classical benchmarking methods used for classical computers. It was evaluated, if the execution would fit the project context.

The project is dominately about benchmarking the simulators on HPC, using classical benchmarking approaches was seen as apropiate next step.

The main issue for the next step was to come up with an artificial task for the benchmark. As the main objective of the benchmarking is to identify how good the simulator scales by problem size and scale of parallelization, the task should involve gradually increasing complexity of calculations.

In case of simulating n qubit QC, the complexity of the calculations increase as n increases. Those the straightforward way to give an artificial task is by doing the same kind of computations with increasing number of qubits.

The existing codebase which involves QV benchmarking was revieved and it was decided that the existing benchmark task already meets the requirements. The only alterations to the benchmark had to be done on the measurements.

More classical performance measurements were added to the benchmark. Also after this stage the codebase was refactored and made modular in order to make it easier to add new simulators.

Instead of using random circuits it was decided to use a stable circuit design:

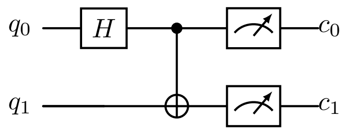

Figure 4: A 2×2 square GHZ circuit.

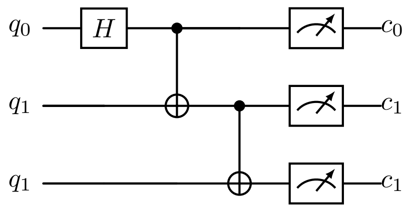

Figure 5: A 3×3 square GHZ circuit.

The circuits in Figures 4 and 5 are Greenberger–Horne–Zeilinger (GHZ) State Circuits[14], which is mainly applied in quantum communication and encryption technologies. In those circuits first the Hadamard gate is used to put the the qubit 0 in superposition. Then the Control gates are applied from each qubit to the next qubit, to entangle them. At the end all qubits end up being entangled.

After those operations all qubits are entangled, meaining the state of each qubit depend on every other qubit, but at the same time all qubits are in the superposition:

$$|GHZ\rangle = \frac{1}{\sqrt{2}}(|000...0\rangle + |111...1\rangle) \tag{6}$$

After the measurement either all states are 0 or 1.

All simulators build the same circuit described above and run it, adding one more qubit and *CNOT* gate each iteration.

5.2 Python Script

The main benchmark toolkit is a list of python scripts that create circuits and run them. Every simulator that has a Python or C interface can be benchmarked.

The main entry point of the script is:

```python
import sys

from helpers import get_args, is_MPI

def main():
    args = get_args()

    if is_MPI():
        from runners import benchmark_runner_MPI
        benchmark_runner_MPI(args)
    else:
        from runners import benchmark_runner
        benchmark_runner(args)

if __name__ == "__main__":
    main()
```

Here the script first determines if the environment supports MPI and if it is needed. The script accepts the following parameters:

- `--simulator`: The simulator to benchmark (qiskit, qibo, qulacs, qutip, cirq)
- `--save_dir`: Directory to save results
- `--total_shots`: Total number of shots
- `--min_qubits`: Minimum number of qubits
- `--max_qubits`: Maximum number of qubits

Depending on the `simulator` parameter the `runner` will import necessary libraries and benchmark the simulator. The circuit builder is designed in a way to work even if not all libraries are installed. For example, if the user wants to benchmark only qiskit they can install all necessary libraries with:

```
$ pip install .[qiskit] # basic install for
   qiskit
$ pip install .[qiskit, mpi] # for mpi support
$ pip install .[<simulator>] # different
   simulator
$ pip install .[all{-mpi}] # full install (
   optionally with mpi)
```

Of course this set up assume openmpi already present on the system.

When the `runner` receives simulator, it will make call to `circuits` which has the build and run codes for all circuits. For qiskit as example the run can be:

```python
def run_qiskit(shots: int, num_qubits: int) ->
    Dict[str, int]:
    """
    Runs a quantum circuit with the specified
        number of qubits and shots using Qiskit
        AerSimulator.

    Args:
        shots (int): Number of measurement shots
            .
        num_qubits (int): Number of qubits in
            the circuit.

    Returns:
        Dict[str, int]: Measurement counts.
    """
    try:
        from qiskit import QuantumCircuit,
            transpile
        from qiskit_aer import AerSimulator
        from qiskit.transpiler import
            CouplingMap

        backend = AerSimulator()

        circuit = QuantumCircuit(num_qubits)

        # Put the first qubit in superposition
        circuit.h(0)

        # Entangle each subsequent qubit
        for qubit in range(num_qubits - 1):
            circuit.cx(qubit, qubit + 1)

        # Take measurements
        circuit.measure_all()

        # Run the Circuit
        coupling_map = CouplingMap.from_full(
            num_qubits)
        transpiled_circuit = transpile(circuit,
            backend, coupling_map=coupling_map)
        result = backend.run(transpiled_circuit,
             shots=shots).result()

        return result.get_counts()
    except Exception as e:
        exception_return(num_qubits, e)
```

The benchmark runner receives this function as a callable reference via:

```python
def get_simulator(args: argparse.Namespace) ->
    Callable[[int, int], Dict[str, int]]:
```

and passes it to the benchmark function. The benchmark function with given number of qubits and total shots calls the `run_<simulator>` which builds a square circuit and runs it for the given amount if shots. The benchmark function measures the total runtime, the build and execution time of the circuit, the entropies from the circuit runs and memory usage. The benchmark runner then saves then saves the results as CSV and PNG files.

The codebase is as modular as possible to make it easy to add new simulators. Furthermore, even if the simulator does not support python, custom interface can be passed to the benchmark runner as long as it is a callable function.

Each simulator function / interface follows a similar pattern, focusing on building a circuit, executing it, and returning the measurement counts. This structure abstracts the differences between simulators, enabling users to benchmark any quantum framework with minimal effort.

Each simulator's specific syntax is handled inside the `run_<simulator>` functions, and the main script doesn't need to worry about the details of circuit construction or execution. This separation of concerns ensures the codebase remains clean and maintainable. Simulators like Qiskit, Qibo, Qulacs, and Cirq all require a different way of defining and running circuits, but the benchmark suite abstracts this so that the benchmark logic remains consistent across different frameworks.

This also reducec risk of mistakes during addition of new simulators as the argument that the runner passes to the circuit builder and the results that the builder return remain the same, so the circuit builder has to be adapted to meet the benchmark requirements, instead of writing a new benchmark for the new simulator.

5.3 Environment

The benchmark was run on NHR-NORD@Göttingen system "Grete"[15]. The cluster itself is designed for GPU workloads and a better alternative would have been the "Emmy" system, however this was the only available option and considering the capabilities of the benchmark were enough.

The system has 2 Zen3 EPYC 7513 processors, each with 32 cores on zen3 architecture[16]. At the time of execution, the system was running on Rocky Linux 8[17].

To install the packages the following commands have to be executed:

```
$ module load openmpi miniforge3
$ conda create -n env python=3.12
$ source activate env
$ pip install .[all-mpi]
```

The benchmark then can be run with:

```
$ srun --mpi=pmix_v3 -n ${workers} python
    benchmark.py \
    --simulator ${simulator} \
    --save_dir ${save_dir} \
    --total_shots ${total_shots} \
    --min_qubits ${min_qubits} \
    --max_qubits ${max_qubits}
```

For simplicity the benchmark on all simulators with different number of workers was run as a job array. See the job submission scripts in A.1.

Initially, it was planned to use the `apptainer` tool on the cluster and run each simulator in a container, however due to circumstances the idea was dropped. Explanations will be in 7 Discussion.

6 Evaluation

First of all we compared the accuracy of the simulators during the runs up to 20 qubits. The reason for limiting the benchmark to 20 qubits was due to Qibo requiring too much memory after around 22-24 qubits. While it does mean that it has poor memory efficiency, more details about the circumstances will be explained later.

As expected, the initial theory that the results of the calculation will be the same for the simulators was confirmed (Fig. 6). This also confirms that when it comes to the accuracy there is no difference in simulators and any of them can be used to achieve the same results.

The second interesting observation for all simulators was that, even though the memory consumption was reduced per worker with increasing number of workers, it had the same pattern for all simulators (Fig. 7). The memory consumption had reduction scale 4. For 1 and 2 workers the memory consumption was the same. It reduced with 4 workers, and then only when it was 16 workers. If the pattern continues this means that the memory consumption reduces when $\log_4(\text{num\_workers})$ is an integer, and it will be reduced with number of workers being 64, then 256 etc.

When it comes to execution time (Fig. 8) and (Fig. 9), some of the simulators have repeating pattern of high execution time at the beginning which then drops and grows as expected. It was tested with different initial number of qubits, and it looks like that is not the factor. Rather it might be more related to the backend initialization and loading into the memory. When running the circuit with 2 qubits and then unloading the program from the memory, to start it again with the another number of qubits, the execution time increases as expected but from the initial high point. For Cirq for example, with 32 workers 2 qubits circuits is executed in ~ 7 seconds. With increasing number of qubits this number grows, if it is allowed to unload the program from the memory.

For this reason the Figure 9 is provided, which shows more practical comparison of the simulators.

While Qibo initially seems to dominate, it doesn't benefit from parallelization when compared to Cirq and Qiskit. At 20 qubits qibo is still faster it also start to consume more memory and as said gets killed due to Out Of Memory error after around 24 qubits (each workers had at least 2 GiB of memory) (Fig. 10).

Another finding with number of qubit more than 20 is that Cirq exectuion time becomes more predictable and follows the same pattern as other simulators except for Qiskit (Fig. 11)

In comparison Cirq and Qiskit have more constant memery consumption and their total execution time, despite being overall higher, grows more linearly instead

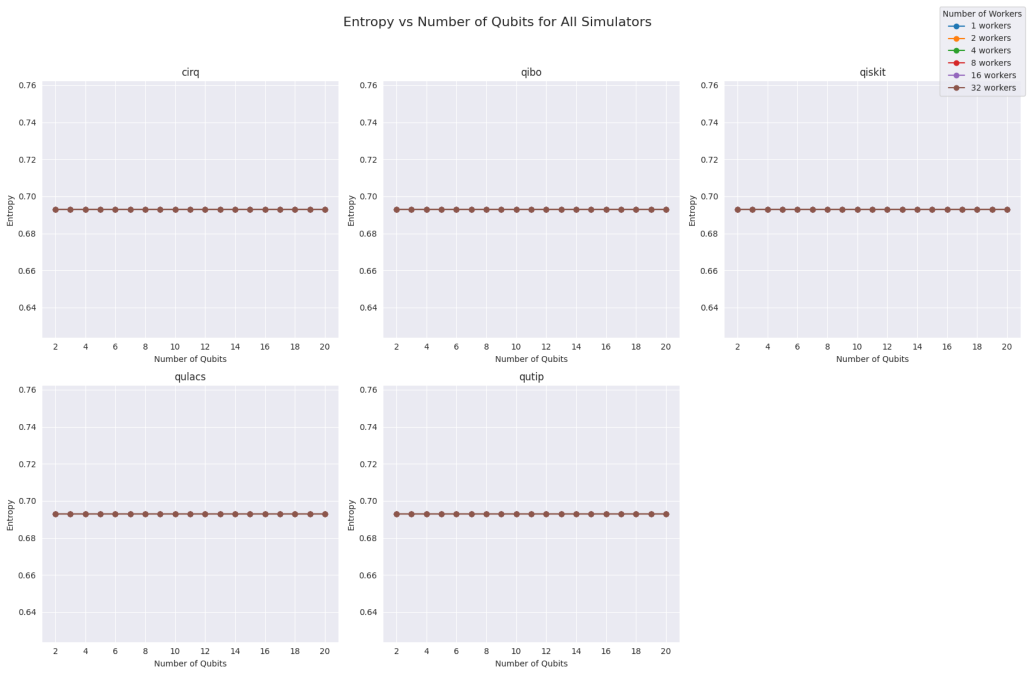

Figure 6: Entropy of different simulators and number of workers. The values almost do not differ, making the comparison meaningless.

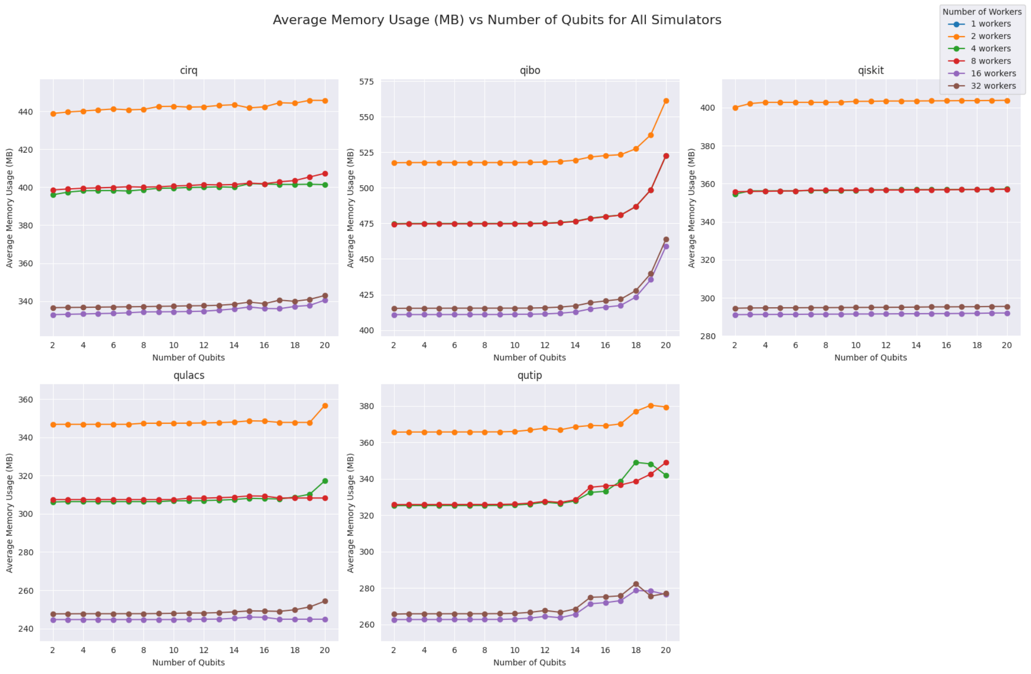

Figure 7: Memory consumption per worker with increasing number of qubits for all simulators.

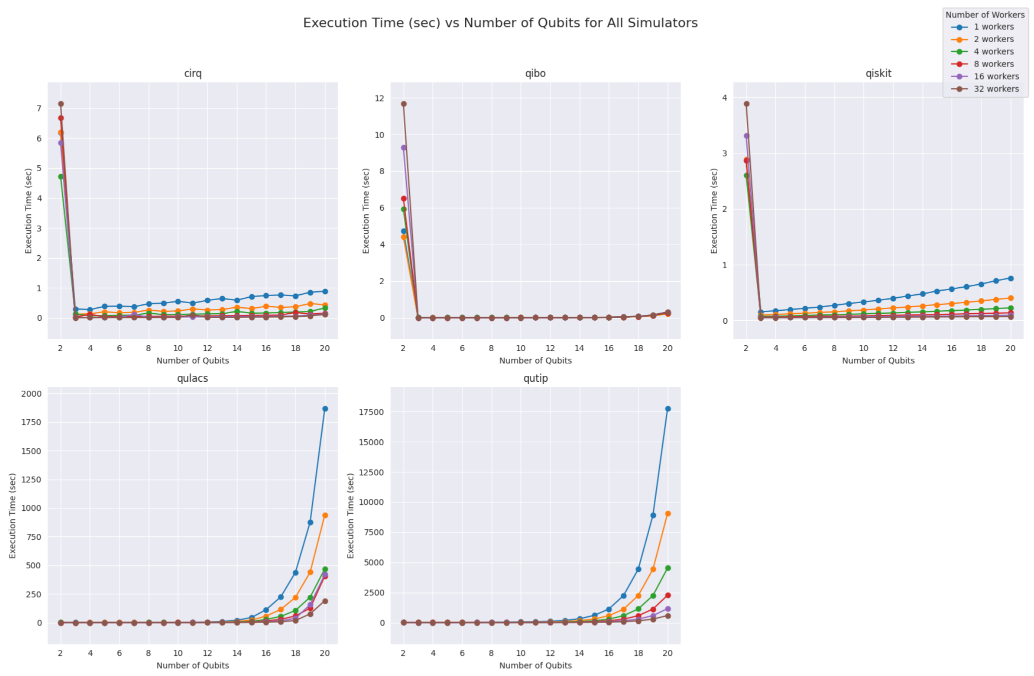

Figure 8: Benchmark execution time with increasing number of qubits for all simulators.

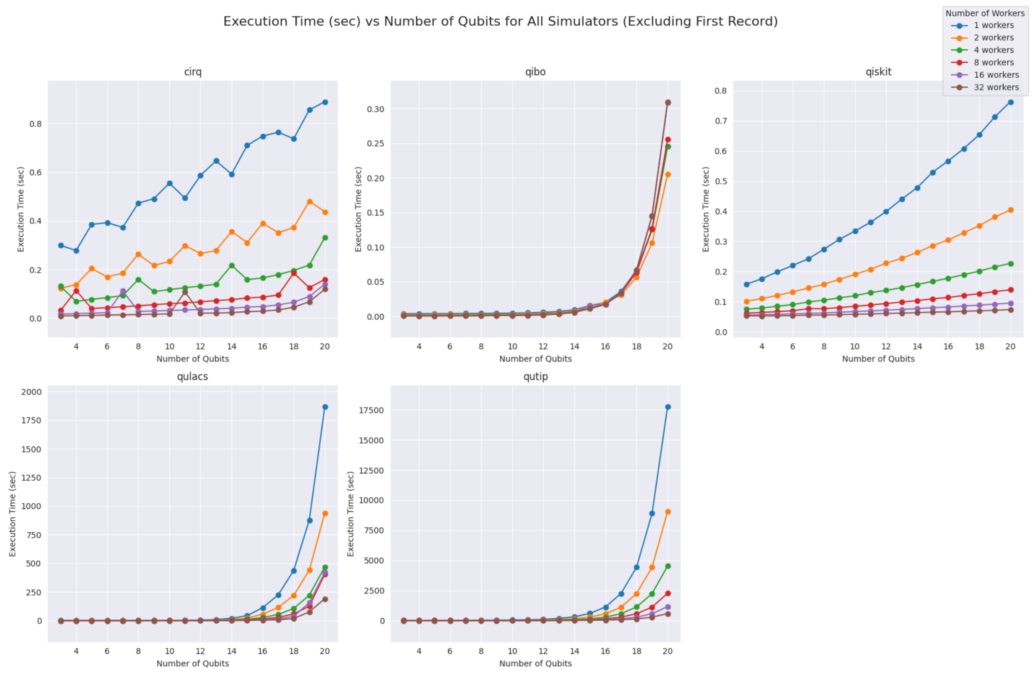

Figure 9: Benchmark execution time with increasing number of qubits for all simulators excluding the first record.

of a defined curve and benefits from the parallelization most. Qulacs and Qutips, as well as Qiskit and Cirq benefit from the parallelization almost linearly, as doubling number of workers decreases the execution time 2 times.

However Qulacs and Qutip have total execution times too high. The pattern, where top 3 simulators in Figure 8 have first initialization time happens in Qulacs and Qutip as well. However compared to the total execution time as the number of qubits grow that initialization time becomes insignificant.

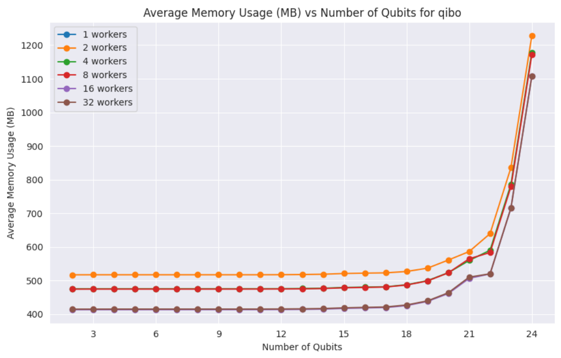

Figure 10: Memory consumption of Qibo as number of qubits increase.

The reason why Qibo was efficient is related to its backend. While most simulators use TensorFlow or NumPy in their backend in one way or another, Qibo here used qibojit which uses `numba` for computations. Numba already has some optimization techniques related to parallelization. More appropriate decision would be to use TensorFlow backend, however according to the developers this backend is not optimized and is designed for Quantum ML mainly. An alternative to numba was their another backend `qibotn`, which uses Tensor Networks to simulate even biggen number of qubits. So, using numba was a middle ground. Additionally, Qibo supports a backend interface for Qulacs.

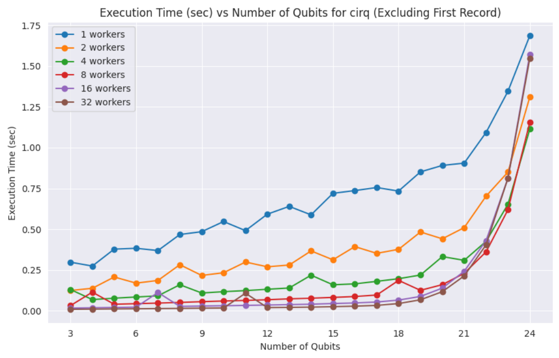

Figure 11: Exectuion time of Cirq starts to curve and have a predictable progression.

While using Numba does explain the efficiency via multi-threading, it still does not explain how parallelization over multiple CPUs did not improve the execution time significantly. Especially considering that the parallelization was not over number of qubits but number of shots:

Listing 1: Parallelization of Shots

```python
shots_per_rank = total_shots // size
remainder = total_shots % size
if rank < remainder:
    shots_per_rank += 1
```

7 Discussion

Initially, the benchmarks were planned to be run in containers. However accessing the host MPI communication from the container did not seem to work as indented, so a conda environment was used instead. A custom container with MPI paths bound to the host did not work either.

Considering the project was also limited to react nodes on the cluster, number of available slots at a time was limited, so debugging on cluster execution in different settings took more time than expected.

Programming the benchmark was not hard, it probably took the least amount of time. Most time was spend in exploring the simulators and trying to set them in similar conditions. While this partially is not fair, since some simulators are designed to perform the best on GPUs, using different hardware was not beneficial for the benchmark.

There was an attempt to do some basic GPU interfacing to run all simulators on GPUs instead, also considering the available cluster partitions, however it was not a successful attempt.

The 5 simulators for the benchmark were selected as they are already present on the cluster as containers. IQS was also planned to be added, however learning the details about that many simulators and running them without accidentally putting them in disadvantage takes time. After the decision to not use containers the benchmark codebase was refactored and made modular with the expectation that new simulators will be added later.

A slightly different approach for the benchmark could have been benchmarking not the simulators but the backends, as long as the backend are indented to be run on the same kind of hardware.

There is also possibility to test how dose the scaling go with the increasing number of shots, however it was not tested to not complicate the visuals and due to the expectation of the linear parallelization because of the parallelization depicted above (Lis. 1).

One of the primary drawbacks of this benchmark is that it was conducted solely on a CPU-based system, even though some simulators are optimized for GPU acceleration. This may have skewed the results. Future work could include a comprehensive benchmark on both CPU and GPU systems to evaluate how the simulators perform across different hardware environments.

8 Conclusion

The project aimed to benchmark different simulators on a HPC system, first using QV as a metric and then as a task and using classical benchmark measurements as metrics instead.

It was quicly discovered that the results of the computations by different simulators do not significantly differ. However their performance differs significantly depending on their backend, when run on the same hardware.

The results showed that while simulators like Qibo performed exceptionally well in smaller qubit benchmarks, their performance diminished as the qubit count grew, particularly due to memory constraints. Simulators like Cirq and Qiskit, on the other hand, demonstrated better scalability and parallelization efficiency, even though they had higher initial execution times.

The main learning outcome from the project work were the internal structures of different simulators and their modification options.

The project was successful in achieving its goals. It gathered valuable measurements about the performance characteristics of the simulators in the context of HPC environments. The modular structure of the benchmark codebase also enables easy extension for future tests with additional simulators or hardware configurations, such as GPUs.

The results of the project can be used to choose simulator for the concrete task, or to contribute to the backend structures of elss efficient simulatros in order to improve them.

A Appendix

A.1 Slurm Submissions

Listing 2: This script runs all sbatch scripts from sbatch-scripts. The reason to run the benchmark job arrays this way is to run them with different number of workers without requesting more workers than needed.

```bash
#!/bin/bash
# Base directory for saving output/error files
    and saving calculation results
base_run_dir=$1
base_save_dir=$2

for script in sbatch-scripts/run_benchmark_n*.sh
    ; do
  workers=$(echo $script | grep -oP '(?<=_n)\d+'
      | sed 's/^0*//')

  # Create a directory for the specific worker
      count
  # e.g run_0001/num_workers_0008 for the base
      dir run_0001 and number of workers 8
  workers_padded=$(printf "%04d" $workers)
  output_dir="${base_run_dir}/num_workers_${
      workers_padded}"
  echo $workers
  echo $output_dir
  echo $base_save_dir

  mkdir -p $output_dir

  sbatch --export=ALL \
    --output=${output_dir}/output-%A_%a.out \
    --error=${output_dir}/error-%A_%a.err \
    $script $base_save_dir
done
```

Listing 3: An example of a job array that request 4 workers for each job. There are several such scripts with only difference in number of workers.

```bash
#!/bin/bash
#SBATCH --job-name=benchmark_runs
#....
#SBATCH --partition=react
#SBATCH --mem-per-cpu=2G
#SBATCH --time=1-12:00:00
#SBATCH --array=0-4
#SBATCH -n 4

# ---------------------------
# Load Required Modules
# ---------------------------
module load openmpi
module load miniforge3

# ---------------------------
# Activate Conda Environment
# ---------------------------
source activate env

# ---------------------------
# Define Simulators and Worker Counts
# ---------------------------

simulators=(qiskit qibo qulacs qutip cirq)
min_qubits=2
max_qubits=20
total_shots=65536

workers=4

num_sims=${#simulators[@]}

# ---------------------------
# Map Job Array Index to Simulator
# ---------------------------

# Calculate simulator index based on
    SLURM_ARRAY_TASK_ID
```

```
sim_index=$(( SLURM_ARRAY_TASK_ID % num_sims ))

# Just in case
if [ $sim_index -ge $num_sims ]; then
    echo "Error: SLURM_ARRAY_TASK_ID=${
        SLURM_ARRAY_TASK_ID} is out of range."
    exit 1
fi

# --------------------------
# Define Save Directory
# --------------------------

base_run_dir=$1

workers_padded=$(printf "%04d" $workers)

save_dir="${base_run_dir}/run_workers_num-${
    workers_padded}/"

mkdir -p "${save_dir}"


# --------------------------
# Execute the Benchmark
# --------------------------

simulator=${simulators[$sim_index]}

echo "==============================="
echo "Job ID: ${SLURM_JOB_ID}"
echo "Array Task ID: ${SLURM_ARRAY_TASK_ID}"
echo "Simulator: ${simulator}"
echo "Workers: ${workers}"
echo "Save Directory: ${save_dir}"
echo "==============================="

srun --mpi=pmix_v3 -n ${workers} python
    benchmark.py \
    --simulator ${simulator} \
    --save_dir ${save_dir} \
    --total_shots ${total_shots} \
    --min_qubits ${min_qubits} \
    --max_qubits ${max_qubits}
```

The benchmark will create in the save directory separate directories for each simulator and run in the format: `result_<simulator>_<i>`.

References

[1] Wikipedia contributors, "Quantum computing — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Quantum_computing&oldid=1247283925`, 2024.

[2] Wikipedia contributors, "Quantum logic gate — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Quantum_logic_gate&oldid=1240670066`, 2024.

[3] G. G. Guerreschi, J. Hogaboam, F. Baruffa, and N. P. D. Sawaya, "Intel quantum simulator: a cloud-ready high-performance simulator of quantum circuits," *Quantum Science and Technology*, vol. 5, p. 034007, May 2020.

[4] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "Quest and high performance simulation of quantum computers," *Scientific Reports*, vol. 9, July 2019.

[5] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen, J. M. Chow, A. D. Córcoles-Gonzales, A. J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S. De La Puente González, E. De La Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I. F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B. G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, Łukasz Herok, H. Horii, Shaohan Hu, T. Imamichi, Toshinari Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Yunho Maeng, M. Marques, F. J. Martín-Fernández, D. T. McClure, D. McKay, Srujan Meesala, A. Mezzacapo, N. Moll, D. M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L. J. O'Riordan, Hanhee Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, Abdón Rodríguez Davila, R. H. P. Rudy, Mingi Ryu, Ninad Sathaye, C. Schnabel, E. Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Y. Siraichi, Seyon Sivarajah, J. A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, Kenso Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J. A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I. Y. Akhalwaya, and C. Zoufal, "Qiskit: An open-source framework for quantum computing," 2019.

[6] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. Garcia-Saez, J. I. Latorre, and S. Carrazza, "Qibo: a framework for quantum simulation with hardware acceleration," *Quantum Science and Technology*, vol. 7, p. 015018, Dec. 2021.

[7] Y. Suzuki, Y. Kawase, Y. Masumura, Y. Hiraga, M. Nakadai, J. Chen, K. M. Nakanishi, K. Mitarai, R. Imai, S. Tamiya, T. Yamamoto, T. Yan, T. Kawakubo, Y. O. Nakagawa, Y. Ibe, Y. Zhang, H. Yamashita, H. Yoshimura, A. Hayashi, and K. Fujii, "Qulacs: a fast and versatile quantum circuit simulator for research purpose," 2020.

[8] Cirq-Developers, "Cirq," May 2024. https://doi.org/10.5281/zenodo.11398048.

[9] J. Johansson, P. Nation, and F. Nori, "Qutip: An open-source python framework for the dynamics of

open quantum systems," *Computer Physics Communications*, vol. 183, p. 1760–1772, Aug. 2012.

[10] J. Johansson, P. Nation, and F. Nori, "Qutip 2: A python framework for the dynamics of open quantum systems," *Computer Physics Communications*, vol. 184, p. 1234–1240, Apr. 2013.

[11] Wikipedia contributors, "Lindbladian — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Lindbladian&oldid=1243066598`, 2024. [Online; accessed 24-September-2024].

[12] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, S. Ahmed, V. Ajith, M. S. Alam, G. Alonso-Linaje, B. AkashNarayanan, A. Asadi, J. M. Arrazola, U. Azad, S. Banning, C. Blank, T. R. Bromley, B. A. Cordier, J. Ceroni, A. Delgado, O. Di Matteo, A. Dusko, T. Garg, D. Guala, A. Hayes, R. Hill, A. Ijaz, T. Isacsson, D. Ittah, S. Jahangiri, P. Jain, E. Jiang, A. Khandelwal, K. Kottmann, R. A. Lang, C. Lee, T. Loke, A. Lowe, K. McKiernan, J. J. Meyer, J. A. Montañez-Barrera, R. Moyard, Z. Niu, L. J. O'Riordan, S. Oud, A. Panigrahi, C.-Y. Park, D. Polatajko, N. Quesada, C. Roberts, N. Sá, I. Schoch, B. Shi, S. Shu, S. Sim, A. Singh, I. Strandberg, J. Soni, A. Száva, S. Thabet, R. A. Vargas-Hernández, T. Vincent, N. Vitucci, M. Weber, D. Wierichs, R. Wiersema, M. Willmann, V. Wong, S. Zhang, and N. Killoran, "Pennylane: Automatic differentiation of hybrid quantum-classical computations," 2018.

[13] D. S. Steiger, T. Häner, and M. Troyer, "Projectq: An open source software framework for quantum computing," 2016.

[14] Wikipedia contributors, "Greenberger–horne–zeilinger state — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Greenberger%E2%80%93Horne%E2%80%93Zeilinger_state&oldid=1237283321`, 2024. [Online; accessed 27-September-2024].

[15] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, "Grete hpc cluster." `https://gwdg.de/en/hpc/systems/grete/`, 2024.

[16] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, "Gpu partitions documentation." `https://docs.hpc.gwdg.de/how_to_use/compute_partitions/gpu_partitions/index.html`, 2024.

[17] Rocky Enterprise Software Foundation, "Rocky linux 8." `https://rockylinux.org/`, 2024.