Towards Scalable Quantum Simulation: Distributed Circuit Cutting for Hybrid Quantum-HPC Systems

Mar Tejedor Ninou

Supervised by: Rosa M. Badia and Alba Cervera-Lierta

Barcelona Supercomputing Center, 08034 Barcelona

July 2025

As quantum computing advances, practical deployment of quantum algorithms remains hindered by hardware limitations such as restricted qubit counts and and limited connectivity. Circuit cutting has emerged as a promising strategy to extend quantum computations beyond these hardware constraints by decomposing large circuits into smaller subcircuits that can be executed individually and recombined through classical post-processing. This master thesis presents `Qdislib`, an open-source software library that integrates quantum circuit cutting with high-performance computing (HPC) infrastructure to enable scalable and hybrid quantum-classical workflows.

`Qdislib` builds on PyCOMPSs, a task-based runtime system, to orchestrate the parallel execution of subcircuits across heterogeneous resources, including CPUs, GPUs, and quantum processors (QPUs). The library supports both wire cutting and gate cutting techniques and introduces an automated cut selection algorithm, FindCut, to optimize circuit partitioning based on user-defined constraints. Benchmarking is performed on Hardware-Efficient Ansatz (HEA) and Random Circuit (RC) workloads, evaluating execution on MareNostrum 5 and IBM Quantum Cloud.

Results demonstrate strong scalability for classical simulations and hybrid execution, achieving near-linear speedups on up to 64 compute nodes and successfully integrating local and remote QPUs. `Qdislib` thus provides a practical and extensible framework for distributed quantum simulation, paving the way for scalable quantum computation in heterogeneous environments.

Mar Tejedor Ninou: mar.tejedor@bsc.es

Acknowledgments

El meu gran agraïment és a la casualitat i la sort del dia que em van presentar el notebook del que seria la introducció d'aquest projecte. Allà vaig descobrir un món completament nou i fascinant que m'ha portat a realitzar aquest màster, pel pur interès d'intentar entendre el món de la quàntica. Gràcies a la Rosa per oferir-me l'oportunitat, guiar-me i a la vegada deixar-me la llibertat de fer-ho tot a la meva manera. Gràcies al Javi per ajudar-me en tot i deixar-me aprendre tantíssim d'ell. Estic profundament contenta de tota la feina feta i feliç de participar en l'inici de molts nous projectes. Finalment, moltes gràcies, Berta i Alba. Gràcies per la vostra feina i per explicar les coses difícils tan senzilles, heu fet que la computació quàntica pogués sonar molt divertida, de vegades :)

Contents

1 Introduction **4**

2 Foundations of Quantum Computation and Circuit Decomposition **5**
- 2.1 Quantum Mechanics as a Computational Framework . . . 5
- 2.2 Entanglement and Quantum Circuit Complexity . . . 5
- 2.3 Theory of Circuit Cutting and Decomposition . . . 6
  - 2.3.1 **Wire Cutting** . . . 6
  - 2.3.2 **Gate Cutting** . . . 8

3 High-Performance and Parallel Computing for Quantum Simulations **11**
- 3.1 Parallelization and Distributed Execution of Subcircuits . . . 11
- 3.2 Quantum Computing Hardware and Execution . . . 12
- 3.3 Integration in Hybrid Quantum-Classical Workflows . . . 12
- 3.4 Orchestration with PyCOMPSs . . . 12

4 Qdislib: A Distributed Circuit Cutting Library **14**
- 4.1 Graph-Based Representation of Quantum Circuits . . . 14
- 4.2 Wire and Gate Cutting Techniques in Qdislib . . . 14
- 4.3 Workflow Execution and Integration . . . 14
- 4.4 The FindCut Algorithm . . . 15

5 Evaluation Methodology and Computational Infrastructure **16**
- 5.1 Computational Backends . . . 16
- 5.2 MareNostrum 5 and Quantum Infrastructure . . . 16
- 5.3 Execution Environment . . . 17
- 5.4 Benchmark Circuits and Physical Relevance . . . 17
  - 5.4.1 **Benchmark 1: Hardware-Efficient Ansatz (HEA)** . . . 18
  - 5.4.2 **Benchmark 2: Random Circuits** . . . 20
- 5.5 Hybrid Executions . . . 22

6 Conclusions **23**

Bibliography **25**

A Structure of the RC Benchmark Circuit **27**

B Illustration of Wire Cutting **28**

C Sample Python Script for Circuit Cutting Experiments **29**

D Using Qdislib: Example Notebook **30**

1 Introduction

Quantum computing is rapidly becoming a central pillar of emerging computational technologies, with the potential to transform fields such as physics, chemistry, optimization, and materials science. Despite significant theoretical progress and improvements in quantum hardware, current quantum devices remain fundamentally constrained. They are not fault-tolerant and typically offer only a limited number of qubits, with short coherence times and high susceptibility to noise. As a result, executing large-scale quantum algorithms on current hardware is challenging, both in terms of computational scale and accuracy.

These limitations give rise to two major obstacles in practical quantum computing. First, it is necessary to manage and mitigate quantum errors arising from hardware imperfections. Second, quantum algorithms must be adapted to run on small devices by decomposing or scaling their structure. An important strategy in addressing both challenges is the integration of quantum devices with classical computing systems—forming hybrid quantum-classical workflows. Many algorithms, particularly those designed for the Noisy Intermediate-Scale Quantum (NISQ) era, rely heavily on iterative classical optimization procedures that interact closely with quantum processors [BCLK+22].

One promising approach to extend the reach of current quantum devices is *quantum circuit cutting*, a technique that decomposes large quantum circuits into smaller, independent subcircuits. This enables simulation or execution of circuits that exceed the size limits of existing quantum or classical hardware by running smaller pieces separately and then recombining their results through classical post-processing.

This master thesis builds upon foundational frameworks in quantum circuit cutting, including wire cutting [PHOW20] and gate cutting [MF21], which respectively partition qubit wires or decompose entangling gates into quasi-probabilistic mixtures of local operations and measurements. These techniques enable the simulation of larger quantum circuits or their execution on smaller quantum devices by decomposing the original circuit into manageable subcircuits and reconstructing the final result from their outcomes—at the cost of executing many more subcircuits and performing post-processing.

Beyond the foundational works, a growing body of research has enhanced circuit cutting methods. For example, algorithmic optimizations and practical implementations for wire cutting have been explored in [TTS+21, CHL+23, LMH+23, BPK23], while gate cutting has seen further theoretical development and experimental investigation in [UPR+23, BBL+23, FMU+22]. In parallel, circuit cutting has been integrated into distributed quantum computing workflows to leverage multi-node classical and quantum hardware, as demonstrated by tools such as FitCut [KDP+24] and IBM's CKT cutting utility [BCE+24].

In the context of quantum circuit simulation, approaches like ATLAS [XCM+24] and CutQC [TTS+21] demonstrate scalable execution of large quantum circuits on GPU-based platforms and smaller quantum devices, respectively. However, these methods often focus on specific hardware or static partitioning and have limitations in flexibility and scalability.

In contrast, this master thesis presents `Qdislib`, a flexible, open-source library designed to support high-performance quantum computing workflows by efficiently managing hybrid quantum-classical execution. `Qdislib` is built on top of PyCOMPSs, a task-based programming model for distributed computing [BCD+15].

A key feature explored in this thesis is the combination of circuit cutting with distributed execution, allowing large quantum circuits to be partitioned into subcircuits that are executed in parallel on different computational resources, including CPUs, GPUs and on-site and cloud QPUs. This enables performance improvements and extends the applicability of hybrid quantum-classical simulations across diverse backend platforms.

2 Foundations of Quantum Computation and Circuit Decomposition

2.1 Quantum Mechanics as a Computational Framework

Quantum computation leverages the principles of quantum mechanics to process information in fundamentally new ways, distinct from classical models. The basic unit of quantum information is the *qubit*, a two-level quantum system whose general state can be written:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \text{with } \alpha, \beta \in \mathbb{C},\ |\alpha|^2 + |\beta|^2 = 1. \tag{1}$$

Unlike classical bits, which are restricted to the values 0 or 1, qubits can exist in a superposition of both states simultaneously, with complex amplitudes α and β encoding probabilistic information.

For a system of n qubits, the state space grows exponentially, forming a vector in a 2^n-dimensional complex Hilbert space. The joint state of the system is described as:

$$|\Psi\rangle = \sum_{i=0}^{2^n-1} c_i\,|i\rangle\,, \quad c_i \in \mathbb{C},\ \sum_i |c_i|^2 = 1, \tag{2}$$

where each basis state $|i\rangle$ corresponds to a binary string $i = i_1 i_2 \cdots i_n$ and is defined as the tensor product:

$$|i\rangle = |i_1\rangle \otimes |i_2\rangle \otimes \cdots \otimes |i_n\rangle\,. \tag{3}$$

Quantum computation proceeds through unitary evolution. Quantum gates, such as the Hadamard, Pauli-X, or CNOT gate, implement unitary operators acting on one or more qubits and collectively define a quantum circuit. Following the unitary evolution, a quantum measurement collapses the system onto a basis state, typically in the computational basis $\{|0\rangle, |1\rangle\}$, with probabilities given by the squared amplitudes $|c_i|^2$.

This framework enables quantum computers to explore a vast computational space via superposition and interference. However, simulating such systems on classical hardware becomes exponentially expensive as n grows, limiting classical simulations to relatively small circuits unless some assumptions are made such as the use of specific quantum gates sets (like Clifford circuits) or the amount of entanglement generated by the circuit is bounded by the size of the system (for which tensor network techniques exist that can represent the quantum evolution efficiently).

2.2 Entanglement and Quantum Circuit Complexity

One of the key non-classical features of quantum mechanics is *entanglement*. An entangled state cannot be written as a product of single-qubit states. For instance, the Bell state

$$\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{4}$$

represents maximal bipartite entanglement, where the measurement outcome of one qubit is perfectly correlated with that of the other.

Entanglement is essential for quantum computational advantage and underpins many protocols such as quantum teleportation, superdense coding, and the exponential speedups seen in algorithms like Shor's and Grover's. However, it also significantly increases simulation complexity, as the full quantum state encode global correlations across the system.

The complexity of a quantum circuit is primarily determined by three key factors:

- **Number of qubits:** This defines the size of the Hilbert space (2^n) and directly affects both memory usage and the computational cost of classical simulation.

- **Circuit depth:** The number of sequential gate layers influences the total number of operations required to evaluate or simulate the circuit.
- **Entanglement structure:** The extent and pattern of entanglement between qubits plays a central role in classical simulability; circuits with highly non-local or global entanglement are significantly more challenging to simulate.

Large, deep, and highly entangled quantum circuits are more expressive but quickly become challenging to simulate or execute, especially when the number of qubits exceeds the capacity of available hardware. In such scenarios, circuit cutting offers a valuable approach by decomposing a large quantum circuit into smaller subcircuits that can fit on limited-size quantum processors.

However, this decomposition is not without cost. The main challenge arises from entanglement: cutting through entangled qubits introduces an exponential overhead, not in memory usage as in full simulation, but in the number of subcircuits that must be generated and processed. This overhead is unavoidable if hardware size is limited, but it also creates an opportunity for parallelizing quantum computation, as the smaller subcircuits can be executed independently.

Thus, circuit cutting provides a practical way to extend the reach of quantum computation beyond current hardware constraints. Understanding and managing entanglement is crucial, both to minimize overhead and to enable efficient distributed quantum processing.

2.3 Theory of Circuit Cutting and Decomposition

Circuit cutting is a technique that simulates large quantum circuits by dividing them into smaller fragments that can be independently simulated or executed. The key motivation is to overcome the exponential scaling of classical resources required to simulate large, entangled circuits by exploiting circuit structure.

The method involves severing a circuit connection—either by *cutting a wire* (interrupting the qubit path between gates) or by replacing a two-qubit entangling gate with a combination of local operations and classical communication. This process introduces intermediate measurements and state preparations, enabling decomposition into smaller subcircuits.

2.3.1 **Wire Cutting**

Wire cutting is a technique that decomposes a large quantum circuit into smaller fragments by *cutting* qubit wires at specific points in the circuit. Conceptually, this replaces a quantum connection (a "wire") with classical communication between two otherwise disconnected subcircuits. This allows each fragment to be simulated or executed independently, with the full circuit behavior recovered via classical post-processing.

The core principle behind wire cutting relies on the completeness and orthogonality of the single-qubit Pauli operator basis $\{I, X, Y, Z\}$. Any single-qubit operator or channel can be expanded as a linear combination of these Pauli operators, which form a basis for the space of 2×2 Hermitian operators. Formally, any density matrix ρ or quantum operation $\mathcal{E}$ acting on a single qubit can be expressed as:

$$\rho = \frac{1}{2} \sum_{i=0}^{3} r_i \sigma_i, \quad \text{where } \sigma_0 = I, \sigma_1 = X, \sigma_2 = Y, \sigma_3 = Z,$$

and the coefficients $r_i = \mathrm{Tr}[\rho \sigma_i]$ are real numbers.

This implies that the identity channel on a qubit, which corresponds to the uncut wire, can be decomposed into a linear combination of operations involving measurement in the Pauli bases followed by preparation of corresponding eigenstates. Concretely, the *quantum identity channel* $\mathcal{I}$ satisfies the expansion:

$$\mathcal{I}(\rho) = \sum_{i=0}^{3} c_i \sum_{s=\pm 1} p_{i,s} \, |\psi_{i,s}\rangle\langle\psi_{i,s}| \, \mathrm{Tr}[|\psi_{i,s}\rangle\langle\psi_{i,s}|\rho],$$

where $\{|\psi_{i,\pm}\rangle\}$ are the eigenstates of σ_i with eigenvalues ± 1, and c_i, $p_{i,s}$ are coefficients chosen to ensure an unbiased reconstruction of expectation values.

Each Pauli operator σ_i has two eigenvalues, ± 1, with eigenstates $|\psi_{i,+}\rangle$ and $|\psi_{i,-}\rangle$. The identity operator can be expanded as:

$$I = |\psi_{i,+}\rangle\langle\psi_{i,+}| + |\psi_{i,-}\rangle\langle\psi_{i,-}|.$$

The wire cutting replaces the identity channel by a weighted sum over measurement-preparation pairs in these eigenbases. The coefficients $c_i = \pm\frac{1}{2}$ arise from expressing the identity superoperator as a sum over Pauli channels with the correct normalization and signs to exactly reconstruct the original channel. Intuitively, the factor $\frac{1}{2}$ comes from the normalization of the Pauli basis $\mathrm{Tr}[\sigma_i\sigma_j] = 2\delta_{ij}$, and the $\pm$ signs correspond to the eigenvalues of the Pauli operators, encoding the measurement outcomes.

Example: Consider a simple circuit with two sequential unitaries U_1 and U_2 on a single qubit, and an observable $\hat{E}$ measured at the end:

(5)

Cutting the wire between U_1 and U_2 partitions the circuit into two subcircuits:

- The first subcircuit: state preparation $|0\rangle$, application of U_1, followed by measurement of the Pauli operator $O_i \in \{I, X, Y, Z\}$.
- The second subcircuit: preparation of the eigenstate $|\psi_i\rangle$ corresponding to the measured Pauli operator O_i, followed by application of U_2 and measurement of $\hat{E}$.

The original expectation value is reconstructed as:

$$\langle\hat{E}\rangle = \sum_{i=1}^{4} c_i \cdot \langle O_i\rangle_{U_1} \cdot \langle\hat{E}\rangle_{U_2,\psi_i},$$

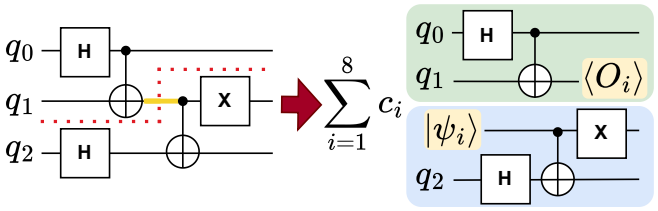


Figure 1: Illustration of wire cutting: a quantum circuit is partitioned by severing a qubit wire, producing subcircuits that are measured and prepared in the Pauli basis. The full circuit's output is reconstructed from these subcircuit evaluations.

where the weights $c_i \in \left\{+\frac{1}{2}, -\frac{1}{2}\right\}$ ensure unbiased reconstruction.

Each wire cut involves measuring in all four Pauli bases and preparing their eigenstates, resulting in $4 \times 2 = 8$ subcircuit evaluations per cut. For k cuts, the total number of subcircuits grows exponentially as 8^k, reflecting the fundamental cost of circuit cutting.

2.3.2 Gate Cutting

Gate cutting is a circuit decomposition technique that targets the modularization of entangling gates—two-qubit operations that cannot be written as a tensor product of single-qubit unitaries, i.e., $U \neq U_1 \otimes U_2$. These gates are fundamental for generating entanglement and are essential to the ability to generate complex quantum states of quantum circuits. However, their nonlocal nature presents a challenge for distributed quantum computing.

The key idea in gate cutting is to express an entangling two-qubit unitary U as a linear combination of tensor products of local (single-qubit) unitaries:

$$U = \sum_{j=1}^{\chi} c_j \, U_j^{(1)} \otimes U_j^{(2)},$$

where $U_j^{(1)}$ and $U_j^{(2)}$ are local unitary operations acting on the individual qubits, $c_j \in \mathbb{C}$ are complex coefficients, and χ quantifies the number of decomposition terms, which is related to the entangling power of the original gate.

If we apply this decomposition within a circuit and attempt to compute the expectation value of an observable $\hat{E}$ on the output state, we are led to expressions involving off-diagonal terms of the form:

$$\langle 0| \left(U_k^{(1)} \otimes U_k^{(2)}\right)^\dagger \hat{E} \left(U_j^{(1)} \otimes U_j^{(2)}\right) |0\rangle,$$

which arise when computing the squared norm of a linear combination of states. These cross terms (when $k \neq j$) require coherent superposition of different unitaries, which is generally infeasible to implement directly in a quantum circuit. As a result, this decomposition is not directly amenable to experimental implementation or efficient classical post-processing.

To circumvent this problem, we adopt a quasi-probabilistic gate decomposition framework introduced by Mitarai and Fujii [MF21]. In this method, the entangling gate is rewritten as a weighted sum over local quantum operations interleaved with mid-circuit projective measurements. This transformation has the form:

$$\mathcal{U}(\rho) = \sum_i c_i \, \mathcal{M}_i \circ \mathcal{U}_i^{(1)} \otimes \mathcal{U}_i^{(2)}(\rho),$$

where $\mathcal{M}_i$ denotes an intermediate measurement operation, and $\mathcal{U}_i^{(1)}, \mathcal{U}_i^{(2)}$ are local unitary transformations. The coefficients c_i are real-valued (or quasi-probabilities) and may be positive or negative. This avoids off-diagonal terms entirely, since the final expectation value is reconstructed as a weighted sum of conditional outcomes from independently executed subcircuits.

The trade-off of this approach is that it requires hardware support for mid-circuit measurement and qubit reset—capabilities that are not yet standard across all quantum computing platforms but that will be. Therefore, for systems where such operations are supported, this method offers a powerful route to decomposing otherwise nonlocal gates into implementable, classically post-processable fragments.

As a concrete example, Figure 3 illustrates the decomposition of a Controlled-Z (CZ) gate into six distinct subcircuits. The CZ gate is a two-qubit entangling gate that applies

a Pauli-Z operation to the target qubit only when the control qubit is in the $|1\rangle$ state:

$$\mathrm{CZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

This specific decomposition uses only four types of single-qubit operations: the Hadamard gate (H), the Pauli-Z gate, and parameterized rotations around the Z and Y axes: $R_z(\theta) = e^{i\frac{\theta}{2}Z}$, $\quad R_y(\theta) = e^{i\frac{\theta}{2}Y}$. It also uses mid-circuit measurements, and follows the probabilistic scheme introduced by Mitarai and Fujii [MF21]. The decomposition expresses the CZ gate as a weighted sum of six simpler quantum circuits:

$$\mathrm{CZ} = \sum_{i=1}^{6} c_i U_i,$$

where each U_i is a local circuit involving rotations, projective measurements, and feed-forward operations, and $c_i \in \mathbb{R}$ are the reconstruction coefficients. These coefficients originate from expressing the two-qubit entangling gate as a linear combination of tensor products of single-qubit operations, sometimes conditioned on measurement outcomes.

The sum of the absolute values of the coefficients defines the sampling overhead factor:

$$\gamma = \sum_{i=1}^{6} |c_i| = 3 \quad \text{and thus} \quad \gamma = 6 \times \frac{1}{2} = 3.$$

This γ-factor determines how much variance is introduced by the decomposition in Monte Carlo estimation of observables and corresponds to the quasi-probabilistic cost described in [EBL18] (see their sampling cost parameter C). In this decomposition, each of the six circuits U_i has a coefficient with absolute value $\frac{1}{2}$.

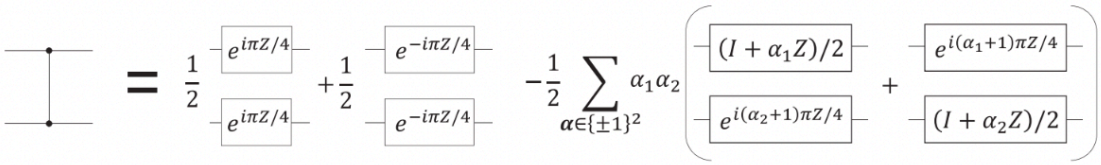


Figure 2: Decomposition of the Controlled-Z (CZ) gate into a linear combination of single-qubit operations and measurements, following the quasi-probabilistic gate decomposition framework proposed in [MF21]. Each term represents a locally implementable subcircuit

To better understand the structure of this decomposition, Figure 2, recall that some terms correspond to deterministic unitary gates (e.g., $e^{\pm i\pi Z/4} \otimes e^{\pm i\pi Z/4}$), while others involve projective measurements in the Z basis and classical post-processing. For example:

- The term $\left(\frac{I+\alpha_1 Z}{2}\right) \otimes e^{i(\alpha_2+1)\pi Z/4}$ corresponds to measuring the control qubit in the Z basis and applying a rotation on the target depending on the outcome.
- The term $e^{i(\alpha_1+1)\pi Z/4} \otimes \left(\frac{I+\alpha_2 Z}{2}\right)$ corresponds to measuring the target qubit in the Z basis and applying a rotation on the control depending on the outcome.

Although the decomposition sums over four combinations of $\alpha_1, \alpha_2 \in \{\pm 1\}$, due to the symmetry of projectors (e.g., $(I \pm Z)/2$ corresponds to the same measurement but different outcomes), only **six distinct subcircuits** are physically implemented.

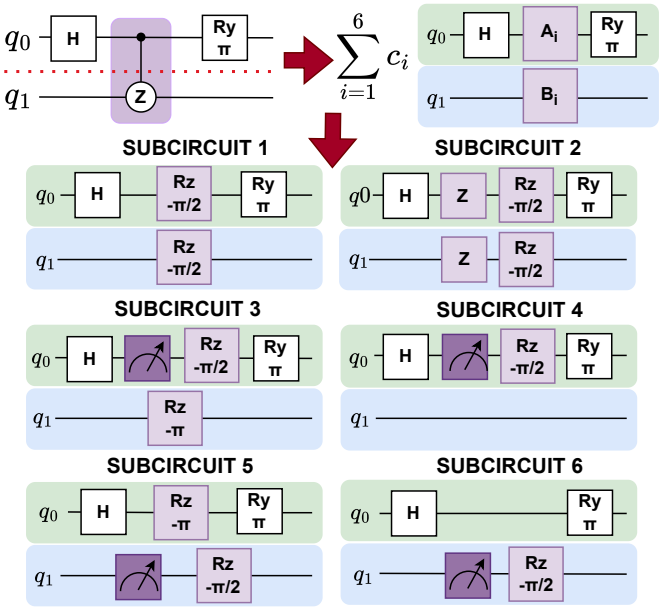


Figure 3: Probabilistic decomposition of the Controlled-Z (CZ) gate into six subcircuits [MF21]. Each circuit corresponds to a term with a reconstruction coefficient $c_i = \pm\frac{1}{2}$. Projective measurements appear in the intermediate steps and influence post-selection weights.

Intermediate projective measurements are denoted with the  symbol in the diagram, and their outcomes are used in classical post-processing to weight measurement results from each subcircuit.

Importantly, the number of required subcircuits to reconstruct a circuit grows exponentially with the number of gate cuts. For instance, cutting k CZ gates leads to 6^k circuit fragments. This exponential scaling is tied directly to the γ-factor of the decomposition, which affects the number of samples needed to estimate an observable to a given precision.

Finally, other two-qubit gates such as the Controlled-NOT (CNOT or CX) gate can be obtained from the Controlled-Z (CZ) gate by conjugation with local single-qubit unitaries. Specifically, the CNOT gate is related to the CZ gate by applying Hadamard gates on the target qubit before and after the entangling operation:

$$\mathrm{CNOT} = (I \otimes H) \cdot \mathrm{CZ} \cdot (I \otimes H)\,,$$

where H is the Hadamard gate defined by

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Because the Hadamard gate exchanges the X and Z bases, satisfying $HZH = X$, conjugating the CZ gate with Hadamards on the target qubit converts the phase-flip action on the $|11\rangle$ state (implemented by CZ) into a bit-flip action on the target conditioned on the control qubit being $|1\rangle$ (implemented by CNOT).

Since the gate cutting decomposition framework expresses an entangling gate as a weighted sum of local operations and intermediate measurements, and local unitaries can be absorbed into the subcircuits before and after the entangling gate, the decomposition of CZ can be extended to CNOT by modifying the local unitaries surrounding the cut gate. Thus, the same coefficients c_i apply, but the single-qubit operators acting on the target qubit are conjugated by Hadamards.

This means the quasi-probabilistic decomposition and gate cutting procedure developed for CZ directly generalize to CNOT, and similarly to other two-qubit entangling gates connected by local unitary transformations. For a general derivation of the quasi-probabilistic decomposition of two-qubit unitaries, we refer the reader to [MF21].

3 High-Performance and Parallel Computing for Quantum Simulations

Classical simulation remains a cornerstone for validating and benchmarking quantum algorithms, since the exponential scaling of the Hilbert space with the number of qubits quickly makes full quantum circuit simulation intractable. This challenge necessitates the use of high-performance computing (HPC) infrastructures that can manage the large memory and compute demands required for simulating non-trivial quantum systems.

In the context of this master thesis, quantum circuits are distributed and simulated through circuit cutting, which decomposes large quantum circuits into multiple independent subcircuits. The focus of this section is to describe how parallel and distributed computing resources are used to execute these subcircuits at scale.

3.1 Parallelization and Distributed Execution of Subcircuits

Each subcircuit produced via circuit cutting is independent and can be simulated in isolation. This naturally lends itself to parallel computing, where the execution of subcircuits can be distributed across multiple processors or nodes without requiring inter-process communication during the quantum evolution stage.

On a single compute node, parallelism is achieved by leveraging the multiple cores of the CPU and, when available, GPU acceleration. Each subcircuit simulation is executed using multi-threaded processes, enabling parallel computation within the node. GPUs are particularly advantageous in this context due to their ability to perform highly parallel linear algebra operations, such as the matrix-vector multiplications involved in simulating quantum state evolution.

When the problem size exceeds the capacity of a single node—either due to memory constraints or computational load—distributed computing is employed. In this case, the workload is spread across multiple nodes in a high-performance computing (HPC) cluster. Each node independently processes several subcircuits, with no need for synchronization during the quantum simulation phase, since the subcircuits are generated to be independent. This combination of parallel execution within each node and distributed execution across nodes enables the system to scale efficiently to large quantum circuits, fully exploiting the computational resources of the supercomputing platform.

Modern supercomputing systems, such as MareNostrum 5, provide the architecture necessary to simulate large batches of subcircuits in parallel. These systems consist of thousands of compute nodes, each containing multiple CPUs and, often, high-performance GPUs. A job scheduler allocates resources for simulation tasks, and parallel runtime environments orchestrate the dispatch and monitoring of subcircuit executions.

Crucially, the lack of inter-node communication during simulation means that network bandwidth and latency have limited impact on performance. This allows the simulation to scale nearly linearly with the number of available nodes, provided that the subcircuits are sufficiently balanced in complexity.

3.2 Quantum Computing Hardware and Execution

While High-Performance Computing (HPC) provides the classical computational backbone for quantum circuit simulation and hybrid workflows, it is equally important to understand the current state and limitations of quantum hardware execution itself. This section briefly reviews the state of the art in quantum processors, their architecture, and communication methods, highlighting challenges and future directions in scalable quantum computing.

Modern quantum processors typically consist of a limited number of qubits arranged on a single chip, where qubits are physically connected through local interactions enabling entangling operations. These quantum chips currently operate with varying response times, coherence properties, and gate fidelities depending on the underlying technology—such as superconducting circuits, trapped ions, or photonic qubits.

One fundamental architectural constraint is that quantum chips today are connected via classical communication channels when integrated into larger systems. This means that quantum information cannot be directly transmitted between separate chips without measurement and classical communication overhead, limiting the scalability of quantum processors. Such classical communication introduces latency and decoherence risks that pose challenges for distributed quantum computation.

Looking forward, quantum interconnects capable of transmitting quantum information coherently between chips are an active area of research. Quantum communication protocols, such as entanglement swapping and quantum teleportation, offer pathways to linking distant qubits with minimal fidelity loss. Realizing quantum networks and quantum repeaters would enable modular and scalable quantum computing architectures, allowing multiple smaller quantum processors to cooperate as a larger quantum system.

In summary, while current quantum hardware imposes size and connectivity limitations, ongoing advances in quantum inter-chip communication promise to transform quantum execution from isolated chips connected classically to integrated quantum networks, paving the way for more powerful and scalable quantum computers.

3.3 Integration in Hybrid Quantum-Classical Workflows

Subcircuit-based simulation is particularly well-suited to hybrid quantum-classical algorithms. In these workflows, each subcircuit can be evaluated at different resources depending on the users preference, producing a high volume of independent simulation tasks.

HPC systems allow these tasks to be distributed and executed in parallel, dramatically reducing the turnaround time for each optimization iteration. Furthermore, subcircuit simulation can be integrated into hybrid execution pipelines that combine quantum hardware and classical resources, with some subcircuits delegated to real quantum devices while others are simulated on HPC platforms.

Overall, the ability to simulate large numbers of subcircuits independently forms the computational backbone of the scalable circuit-cutting framework explored in this master thesis. By leveraging HPC infrastructures effectively, this approach enables the study of quantum algorithms well beyond the capacity of current quantum hardware.

3.4 Orchestration with PyCOMPSs

In order to manage the distributed execution of large numbers of independent quantum subcircuits, a task-based programming model is employed. In this master thesis, we use **PyCOMPSs** (the Python binding of COMPSs), a task-oriented programming framework developed by the Barcelona Supercomputing Center, to orchestrate the parallel execution of simulation tasks across a high-performance computing (HPC) infrastructure, Figure 4.

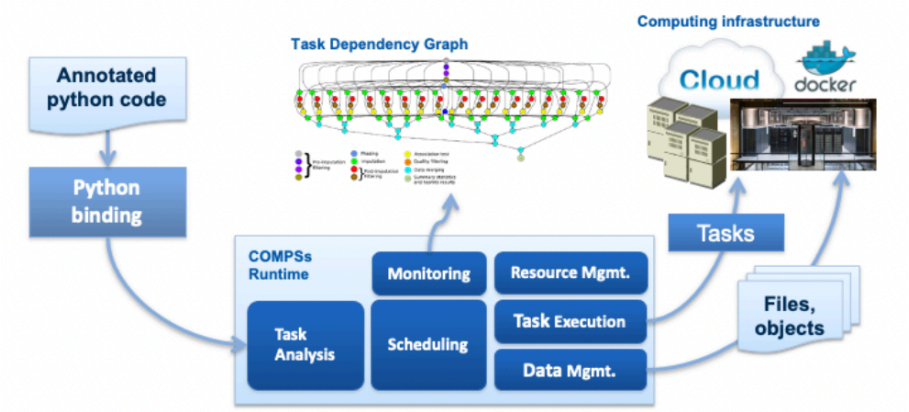


Figure 4: Overview of the PyCOMPSs runtime system. Annotated Python code is analyzed to build a task dependency graph. Tasks are scheduled and executed on different backends such as HPC, cloud, or container-based infrastructures.

PyCOMPSs enables users to express parallelism through simple Python decorators that define computational tasks, Figure 5. These tasks can then be automatically distributed across multiple computing nodes in an HPC cluster. PyCOMPSs handles task scheduling, dependency tracking, data movement, and fault tolerance, allowing scientists to focus on high-level algorithm design rather than low-level parallel programming constructs like message passing or thread management.

```
@task(c=INOUT)
def multiply(a, b, c):
    c += a*b
```

Figure 5: Example of a PyCOMPSs task definition using the `@task` decorator. Tasks can specify input/output behavior for data dependencies.

From the perspective of quantum simulation, PyCOMPSs provides a flexible mechanism to assign subcircuit simulations to different nodes. Each quantum subcircuit generated through circuit cutting can be treated as a distinct task, and PyCOMPSs ensures that these tasks are dispatched to available computational resources. This model is particularly effective when dealing with the embarrassingly parallel structure of the circuit cutting approach, where no inter-task communication is required during quantum evolution.

While PyCOMPSs offers a robust foundation for distributed task execution, it is not natively aware of the specific needs of quantum simulation. For instance, it does not understand quantum circuit structures, quantum-specific execution backends (such as simulators or QPUs), or the dependencies between quantum subcircuits and postprocessing.

To bridge this gap, this master thesis introduces a quantum-aware task orchestration layer built on top of PyCOMPSs. This layer is integrated into `Qdislib`, the software library developed as part of this work. `Qdislib` defines a high-level API for circuit cutting, simulation, and recombination, while extending PyCOMPSs to incorporate quantum-specific metadata, task scheduling policies, and backend abstractions.

Through these contributions, PyCOMPSs is transformed into an effective orchestration engine for hybrid quantum-classical workflows. This extension makes it possible to scale quantum simulations to thousands and millions of subcircuits, while also paving the way for future hybrid execution models that combine quantum hardware and classical HPC resources in a unified workflow.

4 Qdislib: A Distributed Circuit Cutting Library

`Qdislib` is a Python-based software library developed in this master thesis to enable the scalable execution of large quantum circuits through automated circuit cutting, subcircuit generation, and distributed simulation. The library integrates with both quantum programming tools and high-performance computing (HPC) infrastructure, providing a unified workflow for partitioning, simulating, and reconstructing quantum computations.

4.1 Graph-Based Representation of Quantum Circuits

Internally, `Qdislib` models quantum circuits as *directed acyclic graphs* (DAGs), in which each node represents a quantum gate and each directed edge corresponds to a temporal or causal dependency between gates. This abstraction captures the essential structure of the quantum circuit independently of its textual or language-specific representation.

The use of a DAG enables efficient analysis and transformation of circuits. For example, entanglement pathways between qubits can be identified by traversing the DAG. Furthermore, circuit manipulations such as insertions of measurement gates, gate replacements, or rewrites of subcircuits can be performed directly on the graph before re-exporting it to a supported quantum programming language (e.g., Qiskit or Qibo). This graph-centric representation also facilitates interoperability across different quantum software ecosystems making it software agnostic.

4.2 Wire and Gate Cutting Techniques in Qdislib

`Qdislib` supports both wire and gate cutting strategies as means to decompose a quantum circuit into a set of smaller, independent subcircuits. While the theoretical details have been discussed earlier, Qdislib's implementation handles all the practical aspects: inserting measurements and state preparations (for wire cuts), performing quasi-probabilistic decompositions (for gate cuts), and generating the full set of resulting subcircuits.

Due to the exponential growth in the number of subcircuits—8^k for k wire cuts, and 6^k for k gate cuts—`Qdislib` is designed to efficiently generate and organize these combinations. The generated subcircuits are then passed to the task execution engine, where they are independently scheduled for simulation using available HPC or quantum resources.

4.3 Workflow Execution and Integration

The typical workflow, Figure 6 proceeds as follows:

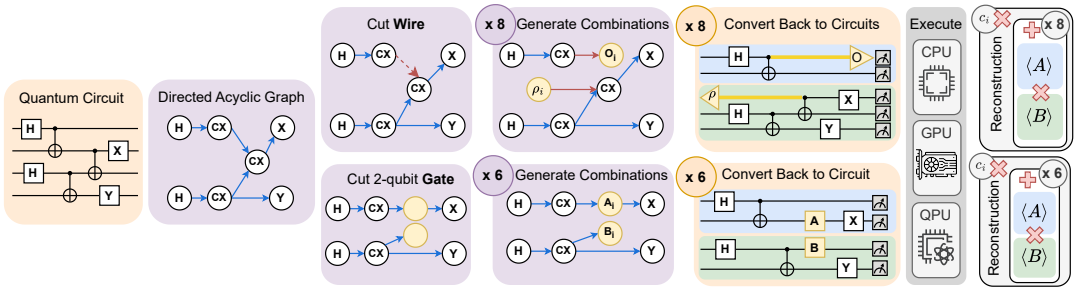


Figure 6: `Qdislib` circuit cutting and distributed execution workflow. The original quantum circuit is transformed into a DAG, cut into subcircuits, expanded into all valid combinations, and distributed for execution across classical and quantum backends. Final observables are reconstructed classically.

1. **Circuit Import and DAG Construction**: The original quantum circuit is loaded from a high-level language such as Qiskit and parsed into a DAG.

2. **Cut Point Selection**: Either manually or via `FindCut`, a set of cuts is chosen.

3. **Subcircuit Generation**: `Qdislib` applies wire and/or gate cutting techniques to generate all necessary subcircuit combinations, encoding measurement, preparation, and decomposition instructions.

4. **Subcircuit Execution**: Each subcircuit is scheduled as a task and executed independently using CPU, GPU or QPU resources.

5. **Result Collection and Postprocessing**: Measurement outcomes and expectation values are collected from all tasks. Classical postprocessing combines the results using either direct summation (wire cuts) or quasi-probability weighted sums (gate cuts).

6. **Final Observable Reconstruction**: Global observables, such as circuit expectation values, are reconstructed from the aggregated data.

This workflow is highly scalable and backend-agnostic. Subcircuits may be executed on a wide range of targets, including local simulators, GPU platforms, or even local or cloud quantum processors (QPU). `Qdislib` ensures reproducibility, modularity, and extensibility, making it suitable for integration into larger hybrid quantum-classical pipelines. The design and implementation of this end-to-end workflow—including DAG-based abstraction, cutting strategies, and distributed orchestration—constitute one of the main contributions of this master thesis.

4.4 The FindCut Algorithm

`Qdislib` includes a dedicated algorithm, `FindCut`, to automate the search for optimal cutting strategies. Given user-defined constraints—such as the maximum number of qubits per subcircuit, the maximum number of cuts, or the preference for wire versus gate cutting—`FindCut` explores the circuit's DAG using several graph partitioning methods. These include classical heuristics such as Kernighan–Lin [KL70], Girvan–Newman [New04], spectral clustering [Chu97], and METIS [KK98]. The workflow of this function can bee analyzed in Figure 7.

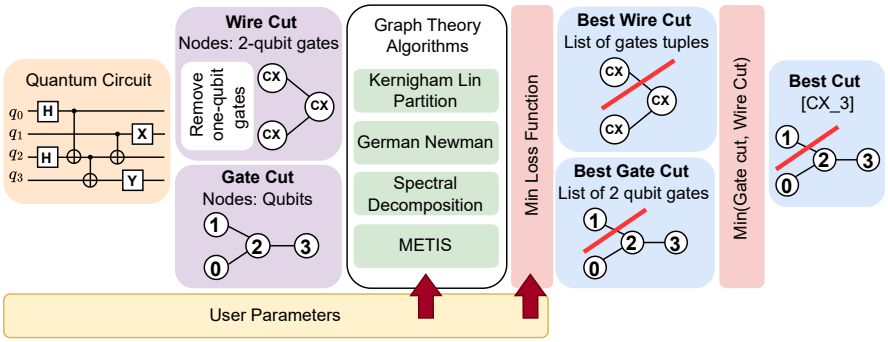


Figure 7: *FindCut* workflow in `Qdislib`.

Each proposed partition is evaluated using a tunable cost function:

$$\text{Loss} = \alpha \cdot \min \text{ cuts} + \beta \cdot \max \text{ components} + \gamma \cdot \min \text{ qubits}$$

where α, β, and γ control the trade-offs between the number of cuts, the degree of parallelism (components), and the resource requirements (qubits per subcircuit). The partition with the lowest loss is selected and used for the circuit decomposition process.

5 Evaluation Methodology and Computational Infrastructure

In this section, we present the benchmarking strategy used to evaluate the capabilities of `Qdislib` in orchestrating large-scale quantum computations across heterogeneous computing platforms. Our focus is on assessing the library's efficiency in partitioning and simulating quantum circuits via circuit cutting.

5.1 Computational Backends

We evaluate `Qdislib` under various systems combining classical and quantum hardware:

- **CPUs:** General-purpose processors for standard classical simulation.
- **GPUs:** Accelerated simulation using NVIDIA H100 GPUs for tensor operations.
- **QPUs (Local):** A on-site superconducting quantum processor.
- **QPUs (Cloud):** Remotely accessed IBM Quantum devices hosted on the IBM Cloud.

Hybrid executions exploit the ability of `Qdislib` to distribute circuit fragments across different backends, executing subcircuits classically or on QPUs as available. All simulations use 1024 shots for statistical consistency. For classical simulation, we use the Qiskit Aer simulator [JATK+24] (CPU-based) and NVIDIA's cuStateVec from cuQuantum [BCC+23] (GPU-based). These simulators compute full wavefunction evolution, making them suitable for validating circuit cutting fidelity and benchmarking performance.

5.2 MareNostrum 5 and Quantum Infrastructure

The primary computational environment is the *MareNostrum 5* (MN5) supercomputer. MN5 is composed of multiple partitions:

- **GPP (General Purpose Partition):** 6,408 nodes, each with dual Intel Xeon Platinum 8480 CPUs, totaling 112 cores per node at 2 GHz.
- **ACC (Accelerated Partition):** 1,120 nodes with Intel Xeon 8460Y+ CPUs (80 cores) and 4 NVIDIA Hopper H100 GPUs (64 GB HBM2) per node.
- **ONA (Quantum Partition):** A superconducting circuit-based quantum processor with a 5-qubit chip in a star topology. A single login node manages submission via a standalone Slurm instance.

The GPP and ACC partitions are connected however with the ONA partition the system is connected via a shared GPFS filesystem, enabling interoperation between classical and quantum partitions. Software for the ONA quantum processor includes: `Qibo` [ERCBP+21] for high-level circuit description and simulation, and `Qililab` [tea25b] for pulse-level control and calibration, interfacing with the quantum control electronics.

Additionally, we execute part of the workloads on IBM Quantum Cloud, targeting the *IBMQ-Marrakesh* system with 156 superconducting qubits arranged in a heavy-hex lattice [tea25a]. Access to this system is provided via Qiskit.

5.3 Execution Environment

Figure 8 illustrates the orchestration of distributed and hybrid quantum-classical computation via `Qdislib`. Execution proceeds as follows:

1. A user submits a quantum circuit for execution through the PyCOMPSs command-line interface on Marenostrum 5.
2. A single Slurm job (ACC or GPP) performs the circuit cutting, partitioning the circuit into subcircuits.
3. Depending on the backend configuration, subcircuits are routed to CPU, GPU, local QPU (ONA), or remote QPU (IBM Cloud) resources.

While GPP and ACC share a scheduler and allow unified job submission, integration with the ONA QPU requires a persistent background agent that monitors a shared GPFS directory for circuit execution requests. Remote access to IBM Cloud QPUs introduces additional complexity due to network firewalls. We address this by employing a two-step port forwarding mechanism: MN5 nodes communicate with a local proxy running on a user workstation, which then forwards requests to IBM Quantum through its API.

This architecture enables seamless exploitation of distributed quantum-classical workflows, unifying cloud and on-premise QPUs with HPC-classical resources.

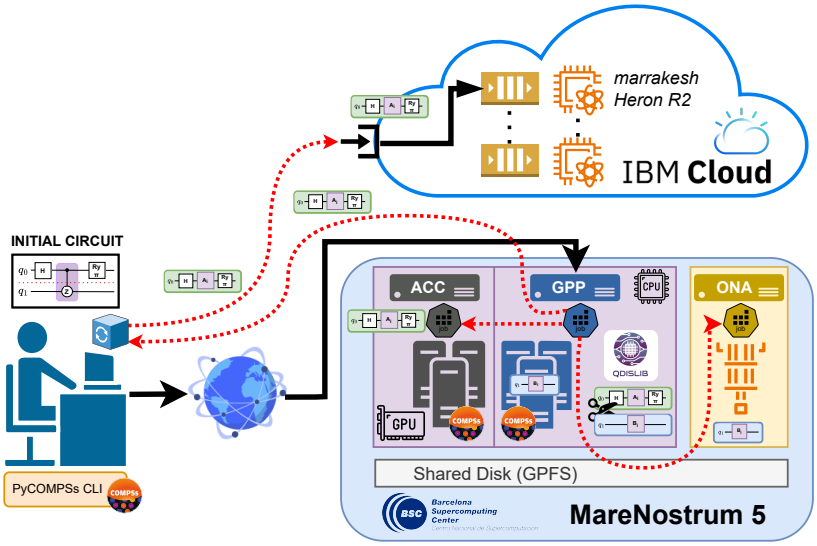


Figure 8: Hybrid execution schema integrating MN5 CPU, GPU, and QPU partitions with IBMQ Cloud.

5.4 Benchmark Circuits and Physical Relevance

To ensure the physical relevance of our tests, we selected two benchmark families representing distinct circuit structures and computational demands.

Hardware-Efficient Ansatz (HEA). This class of circuits, commonly used in variational quantum algorithms for near-term applications [BCLK+22], is designed to be compatible with current hardware connectivity while maintaining sufficient complexity. The HEA structure allows us to explore resource requirements in circuits with constrained depth and limited entanglement range. These properties reflect realistic workloads such as quantum chemistry, optimization, and approximate simulation of many-body systems.

Random Circuits (RC). Inspired by Google's quantum supremacy experiments [AAB+19], RCs serve as a proxy for generic chaotic quantum evolution. They feature deep gate sequences and highly entangled states, making them ideal for stress-testing simulation frameworks. Physically, these circuits resemble dynamics encountered in complex many-body systems, disordered spin chains, and lattice gauge theories.

In all benchmarks, we measure the expectation value of the $Z^{\otimes n}$ operator on the full n-qubit system and we restrict our study to gate cutting only. While `Qdislib` supports both wire and gate cuts, we observe that gate-based decomposition typically requires fewer cuts in structured circuits, especially those with fixed or local connectivity patterns. Wire cutting, in contrast, often leads to an exponential blow-up in the number of subcircuits due to qubit fan-out and long-range correlations.

5.4.1 Benchmark 1: Hardware-Efficient Ansatz (HEA)

The HEA circuit used in this benchmark is generated with the `he_circuit` function from Qibo [ERCBP+21]. It features alternating layers of parameterized single-qubit rotations and entangling CZ gates arranged in a ladder topology, Figure 9. For a system of n qubits and L layers, the total circuit depth scales linearly as $(2 + n)L$. When such a circuit is partitioned into k fragments, each subcircuit also preserves the same layered structure, with depth scaling as $(2 + m)L$, where m is the number of qubits in a subcircuit.

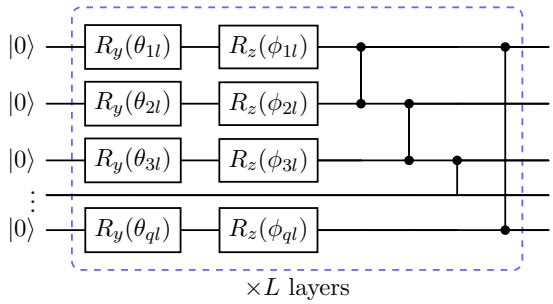


Figure 9: HEA circuit structure used in the benchmarks. Each layer consists of parameterized R_y and R_z single-qubit rotations followed by entangling operations in a ladder topology. The full circuit includes L such layers, and the parameters θ_{ql} and ϕ_{ql} are randomized for each qubit q and layer l.

Figure 10 left shows the simulation times for reconstructing HEA circuits of 32, 64, and 96 qubits, each cut into four equal subcircuits, executed on MareNostrum 5 (MN5-GPP) using only CPUs. Each cut configuration results in fragments of 8, 16, and 24 qubits, respectively. The simulation time increases rapidly with the number of qubits per subcircuit due to the exponential growth in classical memory and computational requirements.

All three circuits exhibit good scalability with increasing numbers of compute nodes; however, the effect is especially pronounced for the larger 96-qubit case. This circuit sees substantial runtime improvements as more nodes are used, highlighting the advantages of distributing large-scale simulation workloads. In comparison, while the 32-qubit and 64-qubit circuits also benefit from parallelization, the relative gains are smaller due to the growing impact of orchestration and communication overhead, which becomes more significant when subcircuit sizes are modest.

To further analyze performance scaling, Figure 10 right focuses on the 96-qubit case. The plot shows both the simulation time and the corresponding speedup as the number of nodes increases from 1 to 64. A near-linear speedup is observed, culminating in a

54.4× acceleration when using 64 nodes, approaching ideal scaling. The initial transitions (1 to 2 nodes, and 2 to 4 nodes) show superlinear speedups. This is attributed to how PyCOMPSs distributes runtime overhead: the master process occupies 12 cores per node for task management, leaving 100 cores for actual computation. Thus, increasing from one to two nodes increases the number of available worker cores from 100 to 212, leading to a significant early performance gain. As more nodes are added, the speedup curve flattens, aligning with expected scaling limits from Amdahl's law.

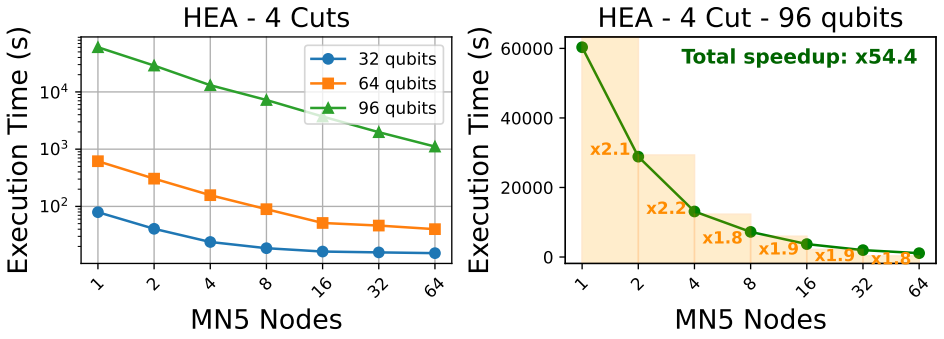


Figure 10: (Left) Execution time for HEA circuits with 4 cuts and different qubit counts on CPU only. (Right) Simulation time and speedup for a 96-qubit HEA circuit with 4 cuts, from 1 to 64 nodes.

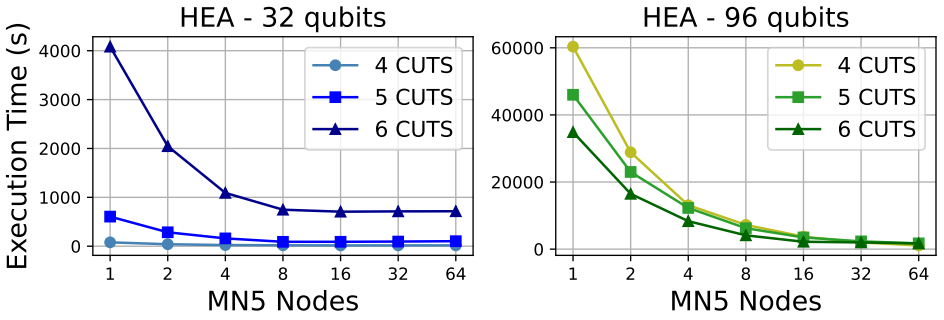


Figure 11: (Left) Execution time for 32- and 96-qubit HEA circuits with 4, 5, and 6 cuts. (Right) Simulation time and speedup for a 96-qubit HEA circuit with 4 cuts, from 1 to 64 nodes.

Figure 11 investigates the effect of the number of cuts on performance. Increasing the number of cuts reduces the size of each fragment but increases the number of subcircuits to simulate and the complexity of postprocessing. For the 32-qubit case (left plot), increasing from 4 to 6 cuts results in a significant performance penalty. This is because the added reconstruction overhead outweighs the benefits of smaller subcircuits, especially when the circuits are already relatively small.

Conversely, the 96-qubit results (right plot) show the opposite trend. Here, more cuts lead to smaller and more manageable subcircuits, dramatically reducing individual simulation times. The case with 6 cuts—producing subcircuits of 25–26 qubits—achieves the best performance. This highlights a key insight: for large circuits, reducing qubit count per fragment significantly accelerates simulation, even if it introduces more tasks to execute. Thus, the optimal number of cuts depends on the trade-off between fragment size and task overhead, and must be tuned based on circuit characteristics.

Finally, Figure 12 compares performance between CPU and GPU-based simulations.

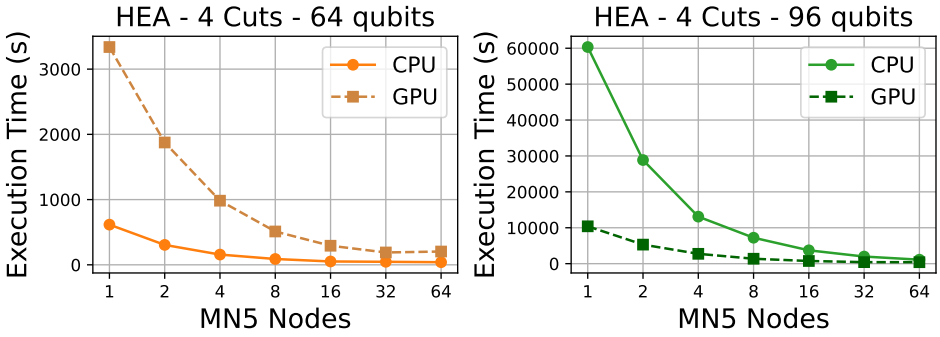


Figure 12: Comparison of execution times on CPU (MN5-GPP) vs GPU (MN5-ACC) for 64- and 96-qubit circuits with 4 cuts.

On the left, the 64-qubit circuit shows better performance on MN5-GPP (CPU) than on MN5-ACC (GPU). This is due to the smaller subcircuits being efficiently parallelized across 112 CPU cores per node, while GPU nodes with only 4 GPUs face memory transfer overheads and less parallelism per node.

In contrast, the 96-qubit circuit (right) is better suited to GPU execution. The larger subcircuits benefit from the massive parallelism available on GPUs, and the reduced number of large simulations aligns well with the GPU's architectural strengths. This confirms that GPUs are advantageous for simulating circuits with a larger number of qubits per fragment, while CPUs may be preferable for workloads with many small subcircuits.

Both experiments show convergence of simulation times as more nodes are used, underscoring the strong scalability of the `Qdislib` approach under both architectures.

5.4.2 Benchmark 2: Random Circuits

As a second benchmark, we evaluate the performance of `Qdislib` using quantum circuits with high qubit connectivity and realistic depth profiles. These circuits are derived from the architecture used in Google's quantum supremacy experiments [AAB+19], which are particularly well-suited for stress-testing circuit partitioning algorithms due to their dense entanglement and layered structure.

In the original Google circuit, two-qubit entangling operations are implemented using the fSim gate, a hardware-native gate that generalizes both the iSWAP and CZ gates. For this benchmark, we simplify the circuits by replacing all fSim gates with CZ gates. This substitution does not significantly alter the circuit's topological complexity but allows us to apply a more tractable gate-cutting protocol. We refer to these modified instances as *Random Circuits* (RC). Although this change facilitates simulation, the same benchmark could be extended to the full fSim gate by either identifying a direct decomposition strategy or expanding it into native gates supported by simulators or quantum hardware. See Appendix A for more detail.

We first use this benchmark to evaluate the performance of the `FindCut` algorithm introduced in Section 4.4. This algorithm identifies optimal circuit partitioning strategies under user-defined constraints, such as a maximum number of qubits per fragment. Figure 13 compares `Qdislib`'s `FindCut` algorithm to IBM's CKT algorithm, available in Qiskit's tutorial repository. We simulate Random Circuits with 20, 30, 36, 42, and 53 qubits, each with a depth of 22 layers. In all cases, a constraint is imposed to ensure that subcircuits contain fewer than 15 qubits.

As shown in Figure 13, `Qdislib` consistently produces fewer cuts than CKT across

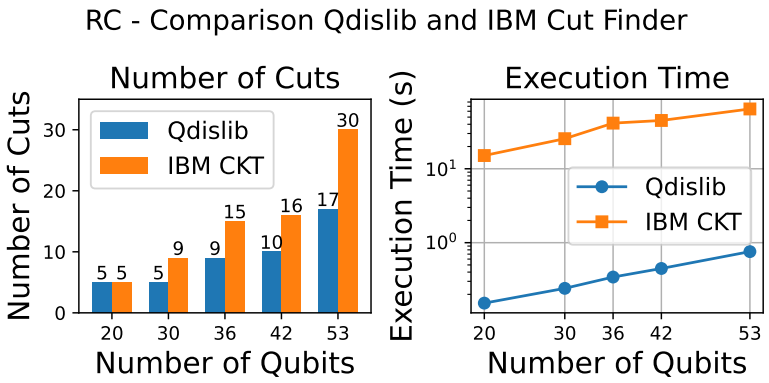


Figure 13: `FindCut` function comparison in `Qdislib` and `IBM CKT` for Random Circuits with a 15-qubit constraint.

all circuit sizes. It also requires less time to compute the optimal cut locations, even as the number of qubits increases. These results align with recent studies [KDP+24], confirming that graph-based partitioning with optimization-guided scoring functions—as implemented in `FindCut`—offers both speed and efficiency. The reduction in the number of cuts is particularly important, as it minimizes the overhead from subcircuit generation and postprocessing, which can grow exponentially with the number of fragments.

To further evaluate runtime performance, we simulate a 36-qubit Random Circuit using 5 wire cuts, resulting in subcircuits that can be efficiently simulated on both CPU and GPU platforms. Figure 14 shows the execution times as a function of the number of compute nodes for both backends.

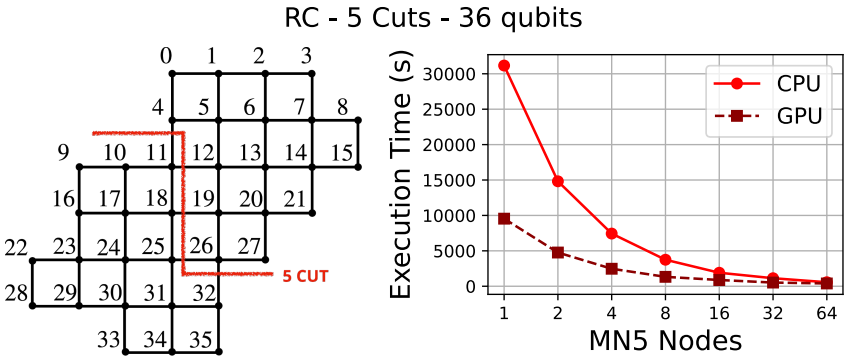


Figure 14: Execution time for a 36-qubit Random Circuit with 5 cuts, comparing CPU and GPU performance as a function of node count.

For small node counts, the GPU clearly outperforms the CPU. This is expected, as the GPU's massively parallel architecture can more effectively accelerate individual subcircuit simulations, especially when the subcircuits are large and involve dense unitary evolution. However, as the number of nodes increases, both architectures begin to converge in performance. At 64 nodes, the total execution time is nearly identical for both CPU and GPU runs. This convergence indicates that the parallel execution model implemented in `Qdislib`, combined with PyCOMPSs task scheduling, is effective in saturating available resources regardless of the underlying architecture.

Moreover, the results show that for large, complex circuits such as the Random Circuit class, GPU acceleration is particularly advantageous at small to medium scales, while CPUs

remain competitive at large node counts due to their higher concurrency and reduced communication overhead. The ability to maintain performance across different architectures highlights the flexibility and scalability of the `Qdislib` framework.

Together, these experiments demonstrate the practical utility of `FindCut` for real-world quantum circuit workloads and confirm that Qdislib's workflow can be applied effectively to irregular, highly entangled circuits—providing efficient circuit partitioning, scalable execution, and hardware-agnostic performance.

5.5 Hybrid Executions

In addition to fully classical simulations, `Qdislib` supports hybrid execution workflows that combine multiple heterogeneous backends—including CPUs, GPUs, and real quantum processing units (QPUs). These hybrid workflows aim to exploit the strengths of each computational resource while overcoming the limitations of current quantum hardware. Large circuits are decomposed into smaller subcircuits using circuit cutting, and each fragment is dispatched to the most suitable backend depending on size, availability, and system constraints.

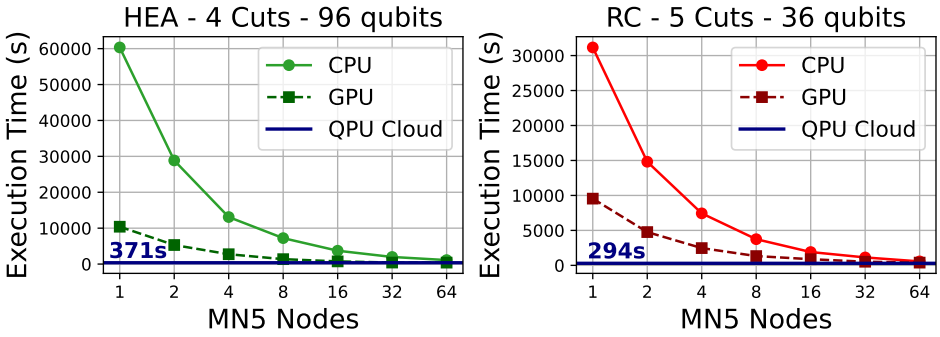


Figure 15: Execution time of circuits using CPUs and GPUs (with circuit cutting) versus QPU execution without cuts (IBM Cloud).

Figure 15 illustrates the hybrid execution model. Each subplot shows the execution time of a complete circuit using CPU and GPU backends (with circuit cutting) alongside a third configuration where the circuit is executed without cuts directly on a QPU (IBM Cloud). These executions without cutting are supported natively in `Qdislib` by providing an empty cut set, which bypasses the decomposition process entirely. This allows a direct comparison between classical and quantum runtimes across different hardware systems.

For the 96-qubit HEA circuit (left subplot), the CPU and GPU runtimes converge as the number of nodes increases, consistent with earlier scaling results. Interestingly, the flat line corresponding to the IBM QPU execution appears in the same range as the CPU/GPU curves at their highest node counts. This convergence highlights an important trade-off: while QPU executions avoid the classical overhead of subcircuit generation and reconstruction, they are still limited by quantum-specific factors such as gate errors, noise, and queuing latency. For the 36-qubit Random Circuit (right subplot), a similar convergence is observed. This suggests that large-scale classical simulation with sufficient parallelism can match—or even outperform—current QPU runtimes, especially for complex, highly entangled circuits.

Table 1 summarizes a series of hybrid executions using different combinations of resources: classical (CPUs and GPUs), local QPU (Ona), and cloud-based QPU (IBM

	Qubits	Cuts	CPUs ncores	GPUs ngpus	QPU nqubits	Cloud QPU nqubits	Time (s)
HEA	10	2	112	–	–	–	7.1
HEA	10	2	–	4	–	–	31.1
HEA	10	2	–	–	5	–	992
HEA	10	2	–	–	–	5	1324
HEA	32	3	112	–	5	5	2061
HEA	32	3	80	4	5	5	1597
HEA	64	2	–	4	–	39	786
HEA	96	2	–	4	–	71	803
HEA	128	2	–	4	–	103	826
RC	36	5	80	4	–	–	19347
RC	36	3	–	–	3	33	937
RC	30	3	112	–	–	18	1318
RC	30	3	–	4	–	18	1251
RC	30	3	80	4	5	5	1636
RC	30	3	–	–	5	25	854

Table 1: Hybrid execution configurations using combinations of CPUs, GPUs, QPUs (Ona), and IBM Cloud QPU.

Quantum). These configurations were selected to match subcircuit sizes to the capabilities of each backend. Small fragments (e.g., 5 qubits) were allocated to the Ona device, big subcircuits (e.g., 18–30 qubits) were executed on GPUs, and the medium ones were assigned to QPU cloud or CPUs, depending on availability an qubit count.

The table also reflects practical considerations. QPU jobs, both on Ona and IBM Cloud, are inherently sequential: only one device is available per execution session, and each job must wait in queue before execution. As such, total simulation time for hybrid workflows involving QPUs is often dominated by the quantum component. Still, in configurations where QPU access is integrated directly into HPC infrastructure (as with Ona), total latency can be reduced significantly. In contrast, cloud-based QPU access suffers from longer queues and network overhead.

Despite these challenges, the hybrid model demonstrates considerable flexibility. By leveraging task-level control and resource-awareness in PyCOMPSs, `Qdislib` can assign each subcircuit to a backend that optimally balances its size, execution cost, and hardware constraints. This enables efficient orchestration of hybrid quantum-classical workflows, with minimal user intervention.

These experiments validate the design goals of `Qdislib`, which include not only support for circuit cutting and distributed simulation, but also seamless integration of diverse backends into a single, unified execution model. As larger QPUs become available and local quantum resources improve, the hybrid paradigm is likely to become the dominant mode for executing complex quantum workloads—making libraries like `Qdislib` essential for bridging the gap between current hardware capabilities and the needs of real-world quantum algorithms.

6 Conclusions

This master thesis has presented `Qdislib`, a distributed and high-performance quantum software library designed for hybrid quantum-classical execution on heterogeneous computing infrastructures. The main objective of this work was to explore and demonstrate the viability and scalability of quantum circuit cutting as a means to extend the applicability of

quantum algorithms on current hardware platforms, including CPUs, GPUs, and quantum processing units (QPUs).

We implemented and evaluated `Qdislib` using two representative quantum benchmarks: Hardware-Efficient Ansatz (HEA) and Random Circuits. These benchmarks were executed in a hybrid computational environment composed of classical processors, hardware accelerators, and real quantum devices accessed both locally and via the cloud. The results demonstrate strong scalability and efficient resource utilization, particularly in classical backends, showing near-ideal speedups on multi-node CPU and GPU configurations. Additionally, we validated the integration of QPUs into the workflow, enabling hybrid quantum-classical executions that leverage the strengths of each platform.

A key design element of `Qdislib` is its graph-based representation of quantum circuits, which, when combined with the task-based PyCOMPSs programming model, enables flexible orchestration of distributed workloads. This hardware-agnostic approach facilitates modular integration of various quantum simulators and backends, making the library adaptable to a broad range of use cases.

Given the current limitations of quantum hardware—namely noise, limited qubit counts, and restricted connectivity—circuit cutting provides a practical pathway for scaling quantum applications both now and in the foreseeable future. In this context, quantum emulation using classical resources remains essential for prototyping and testing large quantum circuits in a noise-free environment. `Qdislib` leverages these emulation techniques efficiently and supports future expansion with other high-performance quantum simulators, such as tensor-network-based methods.

Several avenues for future work have emerged from this master thesis. First, there is significant potential to optimize the number of subcircuits generated by circuit cuts through better cut placement strategies and measurement basis selection. Recent advances in heuristic cut selection [SPS25] and efficient reconstruction protocols [LMH+23] could be integrated into `Qdislib`. Second, incorporating hardware-aware metrics such as qubit fidelity and chip topology could improve the cut decision process in practical deployments. Finally, the library could be extended to support additional use cases beyond circuit cutting, such as variational quantum algorithms and quantum error correction, which also benefit from distributed execution and hybrid scheduling.

In summary, this master thesis has developed and evaluated an open-source [1], scalable, and hardware-agnostic quantum-classical library capable of efficiently executing circuit-cut quantum workloads in high-performance computing environments. The experimental results validate the viability of distributed quantum circuit cutting and demonstrate the potential of hybrid quantum-classical approaches to extend the reach of quantum computing in near term.

[1] `Qdislib` documentation: https://qdislib.readthedocs.io/en/latest/;
source code available at: https://github.com/bsc-wdc/qdislib;
ArXiv preprint available at: https://arxiv.org/abs/2505.01184v2

Bibliography

[AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[BBL+23] Marvin Bechtold, Johanna Barzen, Frank Leymann, Alexander Mandl, Julian Obst, Felix Truger, and Benjamin Weder. Investigating the effect of circuit cutting in qaoa for the maxcut problem on nisq devices. *Quantum Science and Technology*, 8(4):045022, 2023.

[BCC+23] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L. Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, Andreas Hehn, Markus Hohnerbach, Matthew Jones, Tom Lubowe, Dmitry Lyakh, Shinya Morino, Paul Springer, Sam Stanwyck, Igor Terentyev, Satya Varadhan, Jonathan Wong, and Takuma Yamaguchi. cuquantum sdk: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, page 1050–1061, Montréal, September 2023. IEEE.

[BCD+15] Rosa M. Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque, Daniele Lezzi, Francesc-Josep Lordan, Cristian Ramón Cortés, and Raül Sirvent. Comp superscalar, an interoperable programming framework. *SoftwareX*, 3:32–36, 2015.

[BCE+24] Agata M. Brańczyk, Almudena Carrera Vazquez, Daniel J. Egger, Bryce Fuller, Julien Gacon, James R. Garrison, Jennifer R. Glick, Caleb Johnson, Saasha Joshi, Edwin Pednault, C. D. Pemmaraju, Pedro Rivero, Ibrahim Shehzad, and Stefan Woerner. Qiskit addon: circuit cutting. https://github.com/Qiskit/qiskit-addon-cutting, 2024.

[BCLK+22] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S Kottmann, Tim Menke, et al. Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, 94(1):015004, 2022.

[BPK23] Sebastian Brandhofer, Ilia Polian, and Kevin Krsulich. Optimal partitioning of quantum circuits using gate cuts and wire cuts. *IEEE Transactions on Quantum Engineering*, 5:1–10, 2023.

[CHL+23] Daniel T. Chen, Ethan H. Hansen, Xinpeng Li, Aaron Orenstein, Vinooth Kulkarni, Vipin Chaudhary, Qiang Guan, Ji Liu, Yang Zhang, and Shuai Xu. Online Detection of Golden Circuit Cutting Points . In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 26–31, Los Alamitos, CA, USA, September 2023. IEEE Computer Society.

[Chu97] Fan RK Chung. *Spectral graph theory*, volume 92. American Mathematical Society, Providence, 1997.

[EBL18] Suguru Endo, Simon C Benjamin, and Ying Li. Practical quantum advantage in chemical dynamics. *Physical Review X*, 8(3):031027, 2018.

[ERCBP+21] Stavros Efthymiou, Sergi Ramos-Calderer, Carlos Bravo-Prieto, Adrián Pérez-Salinas, Diego García-Martín, Artur Garcia-Saez, José Ignacio Latorre, and Stefano Carrazza. Qibo: a framework for quantum simulation with hardware acceleration. *Quantum Science and Technology*, 7(1):015018, dec 2021.

[FMU+22] Keisuke Fujii, Kaoru Mizuta, Hiroshi Ueda, Kosuke Mitarai, Wataru Mizukami, and Yuya O Nakagawa. Deep variational quantum eigensolver: a divide-and-conquer method for solving a larger problem with smaller size quantum computers. *PRX Quantum*, 3(1):010346, 2022.

[JATK+24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.

[KDP+24] Shuwen Kan, Zefan Du, Miguel Palma, Samuel A Stein, Chenxu Liu, Wenqi Wei, Juntao Chen, Ang Li, and Ying Mao. Scalable circuit cutting and scheduling in a resource-constrained and distributed quantum system. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1077–1088, Montréal, 2024. IEEE.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KL70] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[LMH+23] Angus Lowe, Matija Medvidović, Anthony Hayes, Lee J O'Riordan, Thomas R Bromley, Juan Miguel Arrazola, and Nathan Killoran. Fast quantum circuit cutting with randomized measurements. *Quantum*, 7:934, 2023.

[MF21] Kosuke Mitarai and Keisuke Fujii. Constructing a virtual two-qubit gate by sampling single-qubit operations. *New Journal of Physics*, 23(2):023021, 2021.

[New04] Mark EJ Newman. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[PHOW20] Tianyi Peng, Aram W Harrow, Maris Ozols, and Xiaodi Wu. Simulating large quantum circuits on a small quantum computer. *Physical review letters*, 125(15):150504, 2020.

[SPS25] Lukas Schmitt, Christophe Piveteau, and David Sutter. Cutting circuits with multiple two-qubit unitaries. *Quantum*, 9:1634, February 2025.

[tea25a] IBMQ team. Ibmq processor types. https://docs.quantum.ibm.com/guides/processor-types.html, 2025. Accessed: 2025-04-07.

[tea25b] Qilimanjaro Quantum Tech team. Qililab documentation, 2025. Accessed: 2025-04-04.

[TTS+21] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. Cutqc: using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 473 – 486, Virtual, April 2021. ACM.

[UPR+23] Christian Ufrecht, Maniraman Periyasamy, Sebastian Rietsch, Daniel D Scherer, Axel Plinge, and Christopher Mutschler. Cutting multi-control quantum gates with zx calculus. *Quantum*, 7:1147, 2023.

[XCM+24] Mingkuan Xu, Shiyi Cao, Xupeng Miao, Umut A Acar, and Zhihao Jia. Atlas: Hierarchical partitioning for quantum circuit simulation on gpus. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, Atlanta, 2024. IEEE, IEEE.

A Structure of the RC Benchmark Circuit

This appendix presents the detailed gate-level structure of the Random Circuit (RC) benchmark used in the experimental evaluation. The design reflects the layered architecture of Google-style quantum supremacy circuits, adapted here to use CZ gates for entanglement.

The RC benchmark features a mesh connectivity pattern across qubits, with alternating layers of:

- Random single-qubit gates (`X`, `Y`, `H`, and parameterized `R_z` rotations),
- Entangling layers of `CZ` gates mapped according to a mesh pattern,
- Repeated stacking of these layers to generate depth.

Below, Figure 16 shows one such mesh layer in detail, applied to a 36-qubit circuit. Only the first entangling layer is depicted explicitly; subsequent layers follow a similar pattern.

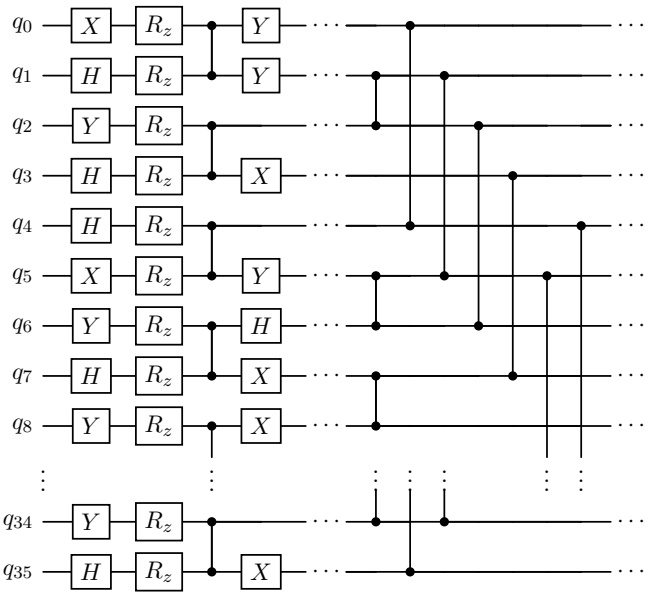


Figure 16: Structure of a Random Circuit (RC) benchmark used in this study, featuring a mesh connectivity pattern. The circuit consists of alternating layers of single-qubit rotations and two-qubit controlled gates CZ (replacing original fsim gates) to emulate the entangling structure of Google's quantum supremacy circuits. Only the first mesh layer is shown in detail; additional layers follow a similar pattern.

B Illustration of Wire Cutting

Figure 17 presents a detailed illustration of the wire cutting technique applied to a simple quantum circuit. This method enables distributed quantum computation by partitioning a quantum circuit into smaller subcircuits, which are independently simulated and recombined classically.

In the figure, a 3-qubit quantum circuit is cut by severing the middle qubit wire. The resulting subcircuits replace the cut wire with all combinations of Pauli basis preparations and measurements.

This appendix figure complements the main discussion in Chapter 2.3.1, providing a visual breakdown of how quantum circuit cutting works in practice and how classical recomposition is achieved.

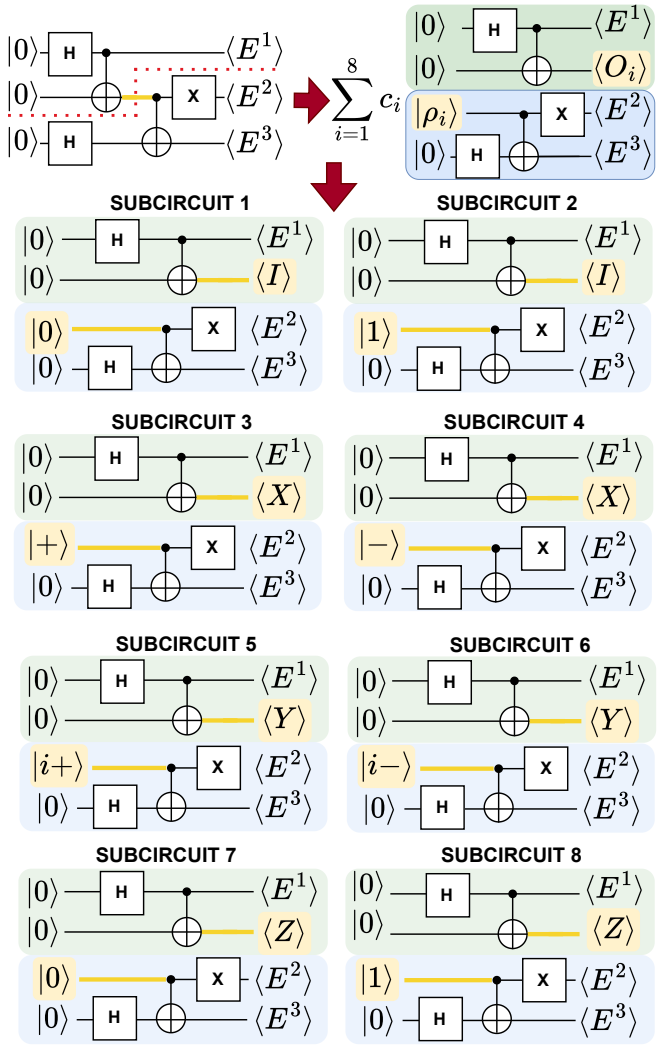


Figure 17: Detailed visualization of quantum circuit wire cutting. The original 3-qubit circuit (top left) is partitioned by severing the middle qubit wire, resulting in two independent subcircuits (top right). Each wire cut introduces a classical mixture over 8 possible Pauli state preparations and measurements, leading to 8 subcircuit configurations shown below. The full expectation value of the original circuit is reconstructed from a weighted sum of the expectation values of these subcircuits.

C Sample Python Script for Circuit Cutting Experiments

The following script illustrates a representative example of the type of experiments conducted in this thesis using `Qdislib` and `PyCOMPSs`. It sets up a 32-qubit Hardware-Efficient Ansatz (HEA) circuit, applies circuit cutting with a maximum subcircuit size, and performs reconstruction using the `automatic` method.

```python
from pycompss.api.api import compss_wait_on
import time
from qibochem.ansatz import he_circuit
import Qdislib.api as qd
import numpy as np

if __name__ == "__main__":
    qubits = 32
    num_cuts = 4
    depth = 1

    a = 'automatic'
    dm = 'density_matrix'
    tn = 'tensor_network'
    mps = 'matrix_product_state'

    method = a

    print("NUM QUBITS ", qubits)
    print("NUM CUTS ", num_cuts)
    print("DEPTH ", depth)
    print("METHOD ", method)

    max_q = qubits // (num_cuts // depth)

    for i in range(2):
        circuit = he_circuit(qubits, depth)

        random_params = [(np.random.uniform(0, 2*np.pi),) for _ in range(len(circuit.get_parameters()))]
        print(random_params)
        circuit.set_parameters(random_params)

        start_time = time.time()
        cut = qd.find_cut(circuit, max_qubits=max_q, implementation='qdislib')
        reconstruction = qd.gate_cutting(circuit, cut, method=method)
        reconstruction = compss_wait_on(reconstruction)
        end_time = time.time() - start_time

        print("CUT ", cut)
        print("RECONSTRUCTION ", reconstruction)
        print("TIME ", end_time)
```

D Using Qdislib: Example Notebook

📓 Qdislib Example Notebook with PyCOMPSs

This document explains each section of the Qdislib Jupyter notebook, showcasing how to apply gate and wire cutting techniques to large quantum circuits using Qdislib, Qibo, Qiskit, and PyCOMPSs.

Import the PyCOMPSs library

In [1]:
```
import pycompss.interactive as ipycompss
```

🔧 1. Environment Setup with PyCOMPSs

Goal: Start the PyCOMPSs runtime for distributed execution.

What Happens:

- Loads required COMPSs XML config files: project.xml and resources.xml.
- Enables optional flags like graph, monitor, debug, and trace.

Why: PyCOMPSs enables parallel task execution — crucial for evaluating subcircuits in distributed environments.

In [2]:
```
ipycompss.start(graph=True, monitor=1000)  # debug=True, trace=True
```

```
************************************************************
*************** PyCOMPSs Interactive *******************
************************************************************
*          .-~~-.--.          ______         ______       *
*         :         )        |____  \       |____  \      *
*   .~ ~ -.\       /.- ~~ .      __) |         __) |      *
*   >       `.   .'       <     |__  |        |__  |      *
*  (         .- -.         )    ____) |   _    ____) |     *
*   `- -.-~  `- -'  ~~-.- -'   |______/  |_|  |______/     *
*     (        :        )           _ _ .-:                *
*      ~--.    :    .--~        .-~  .-~  }                *
*          ~-.-^-.-~ \_      .~  .-~   .~                  *
*                  \ \ '     \ '_ _ -~                     *
*                   \`.\`.    //                           *
*          . - ~ ~-.__\`.\`-.//                            *
*       .-~   . - ~  }~ ~ ~-.~-.                          *
*     .' .-~      .-~       :/~-.~-./:                    *
*    /_~_ _ . - ~                 ~-.~-._                 *
*                                    ~-.<                 *
************************************************************
* - Starting COMPSs runtime...                             *
```

```
* - Log path : /home/mtejedor/.COMPSs/Interactive_248/
* - PyCOMPSs Runtime started... Have fun!              *
******************************************************
```

Import task and compss_wait_on module before annotating functions or methods

In [3]:
```
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on
```

🧠 2. Importing Required Modules

Modules:

- qibo.models, qibo.gates, qibo.hamiltonians: for defining and manipulating quantum circuits.
- qd: likely refers to Qdislib, which contains the circuit cutting functions.

Setup:

- Sets the backend for Qibo ("numpy") to run locally on CPU.

In [4]:
```
import matplotlib.pyplot as plt
import numpy as np
import qibo
from qibo import models, gates, hamiltonians, callbacks
from qibo.models import Circuit
from qibo.symbols import X, Y, Z, I
from qibo.ui import plot_circuit

qibo.__version__
qibo.set_backend("numpy")
```

```
[Qibo 0.2.16|INFO|2025-05-26 15:19:29]: Using numpy backend on /CPU:0
```

Import Qdislib where the circuit cutting is implemented

In [5]:
```
import Qdislib.api as qd
```

⚙️ 3. Define the Main Circuit

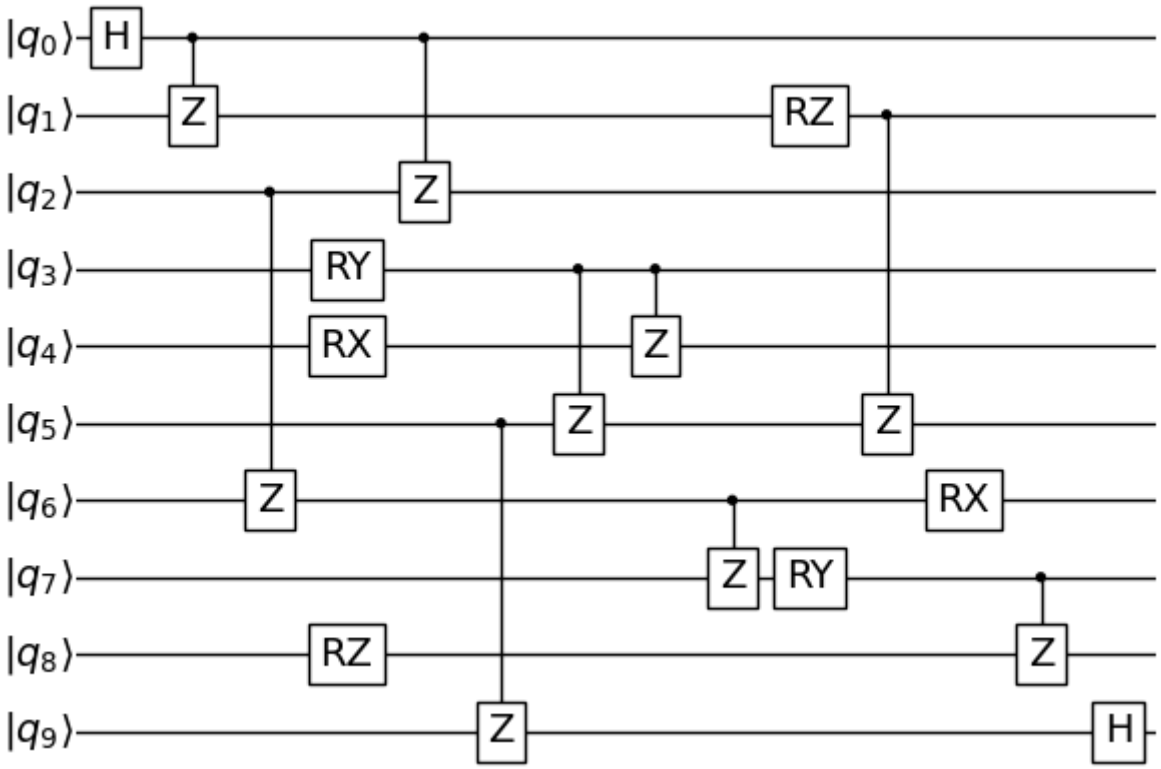
Function entire_circuit(): builds a 10-qubit circuit with:

- Single-qubit gates: H, RX, RY, RZ.
- Two-qubit gates: CZ.

Circuit Objective: Simulates a non-trivial entangled circuit useful for demonstrating cutting algorithms.

In [6]:

```python
def entire_circuit():
    nqubits = 10
    circuit = models.Circuit(nqubits)

    circuit.add(gates.H(0))
    circuit.add(gates.CZ(0, 1))
    circuit.add(gates.CZ(2, 6))
    circuit.add(gates.RZ(8, np.pi / 3))

    circuit.add(gates.RY(3, np.pi / 5))
    circuit.add(gates.RX(4, np.pi / 5))
    circuit.add(gates.CZ(0, 2))
    circuit.add(gates.CZ(5, 9))

    circuit.add(gates.CZ(3, 5))
    circuit.add(gates.CZ(3, 4))
    circuit.add(gates.CZ(6, 7))
    circuit.add(gates.RY(7, np.pi / 5))
    circuit.add(gates.RZ(1, np.pi / 5))

    circuit.add(gates.CZ(1, 5))
    circuit.add(gates.RX(6, np.pi / 5))
    circuit.add(gates.CZ(7, 8))

    circuit.add(gates.H(9))
    return circuit


circuit = entire_circuit()
# print(circuit.draw())
plot_circuit(circuit, scale=0.5)
```



✂️ 4. Gate Cutting Example

qd.find_cut(circuit): Automatically identifies gates suitable for cutting.

- Example result: ['CZ_2']

qd.gate_cutting(circuit, cut):

- Applies the gate cutting algorithm.
- Cuts the circuit at the specified gate.
- Evaluates subcircuits and reconstructs the expectation value.
- Output: A reconstructed value (e.g., 0.0084...).

In [7]:
```
circuit = entire_circuit()

cut = qd.find_cut(circuit)
print(cut)
```

```
['CZ_2']
```

In [8]:
```
reconstruction = qd.gate_cutting(circuit, cut)
print(reconstruction)
```

```
0.01041412353515625
```

🔌 5. Wire Cutting Example

find_cut(..., gate_cut=False):

- Finds cuts (pairs of gates) between which a wire cut is possible.
- Force the algorithm to only finde wire cuts (setting gate_cut=False)
- Example: [('CZ_2', 'CZ_7')]

qd.wire_cutting(...):

- Applies the wire cutting method across the selected gates.
- Calculates the reconstructed expectation value.

In [9]:
```
circuit = entire_circuit()

cut = qd.find_cut(circuit, gate_cut=False)
print(cut)
```

```
[('CZ_2', 'CZ_7')]
```

In [10]:
```
reconstruction = qd.wire_cutting(circuit, cut)
print(reconstruction)
```

```
0.0036580199999999953
```

📊 6. Exact Expected Value

qd.analytical_solution(circuit, "Z"*nqubits):

- Computes the expected value using a symbolic statevector.
- Used as a ground-truth comparison.
- Observable can be modified

Observation: Returns exact expected value — allows evaluation of gate cutting accuracy.

In [11]:
```
circuit = entire_circuit()

analytic = qd.analytical_solution(circuit, "Z" * circuit.nqubits)
print(analytic)
```

```
0.0
```

🧩 7. Gate Cutting with Subcircuits

qd.gate_cutting_subcircuits(...):

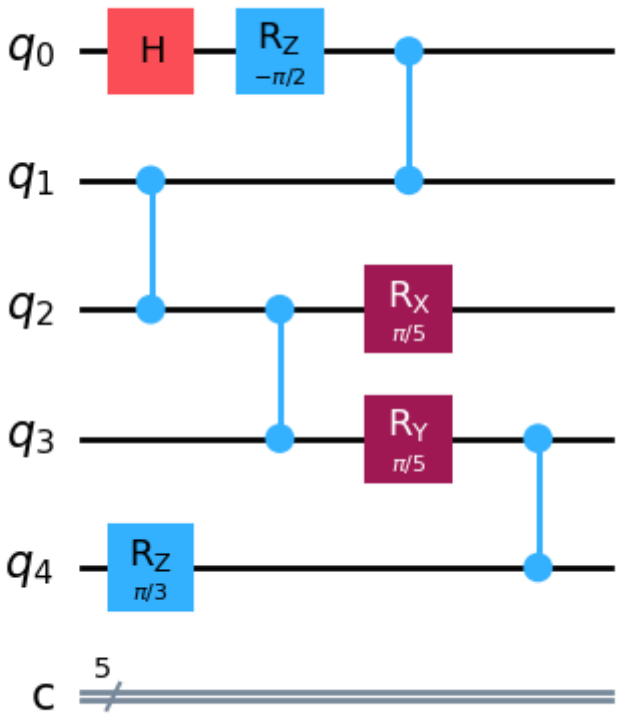
- Performs gate cutting like before but also returns the subcircuits used for reconstruction.

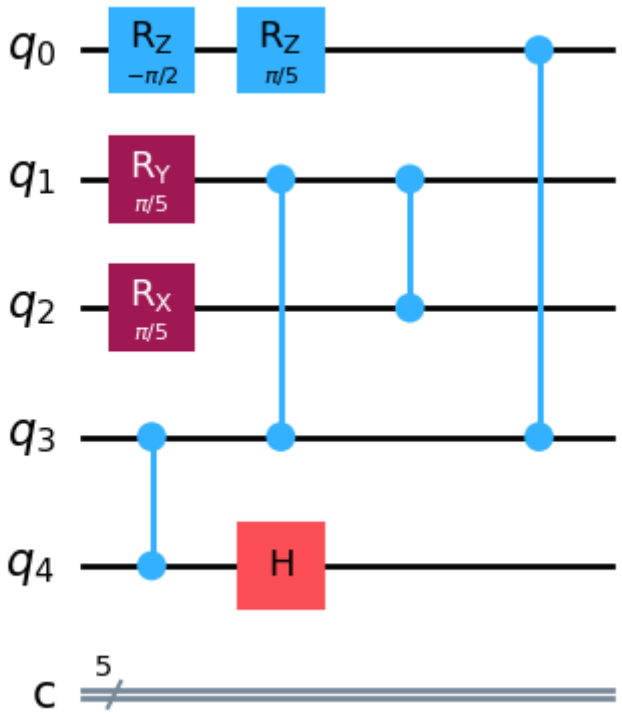
compss_wait_on(subcircuits): Synchronizes and retrieves the results (used with PyCOMPSs).

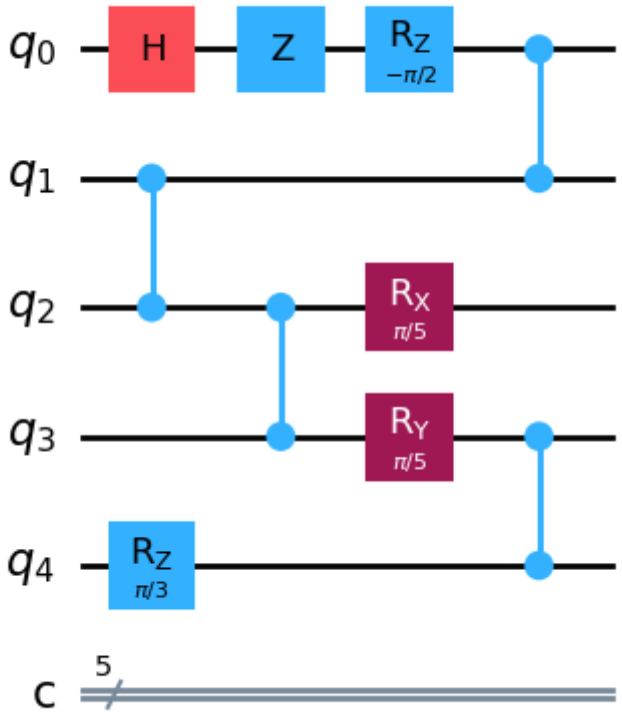
Printed output: Visual representation of generated subcircuits using Qiskit's circuit print.

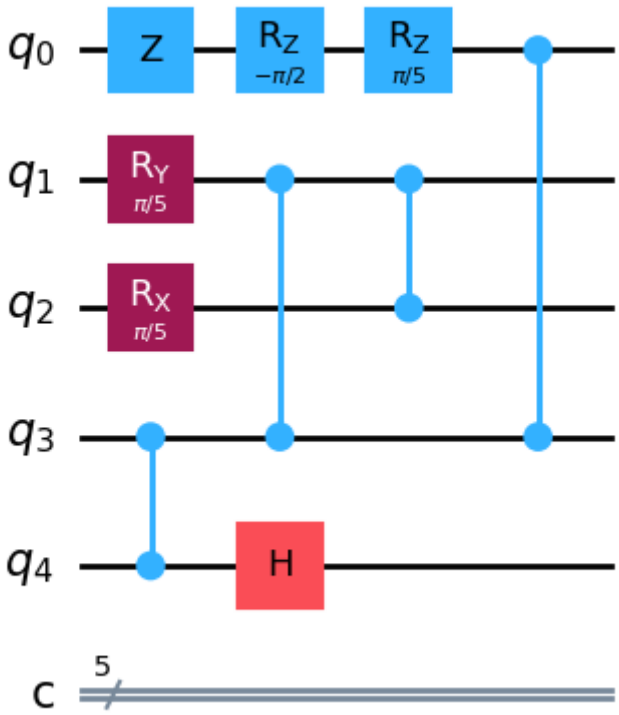
In [12]:
```
circuit = entire_circuit()

cut = qd.find_cut(circuit)
print(cut)
subcircuits = qd.gate_cutting_subcircuits(circuit, cut, "qiskit")

subcircuits = compss_wait_on(subcircuits)

for subcirc in subcircuits:
    display(subcirc.draw(output="mpl"))
```

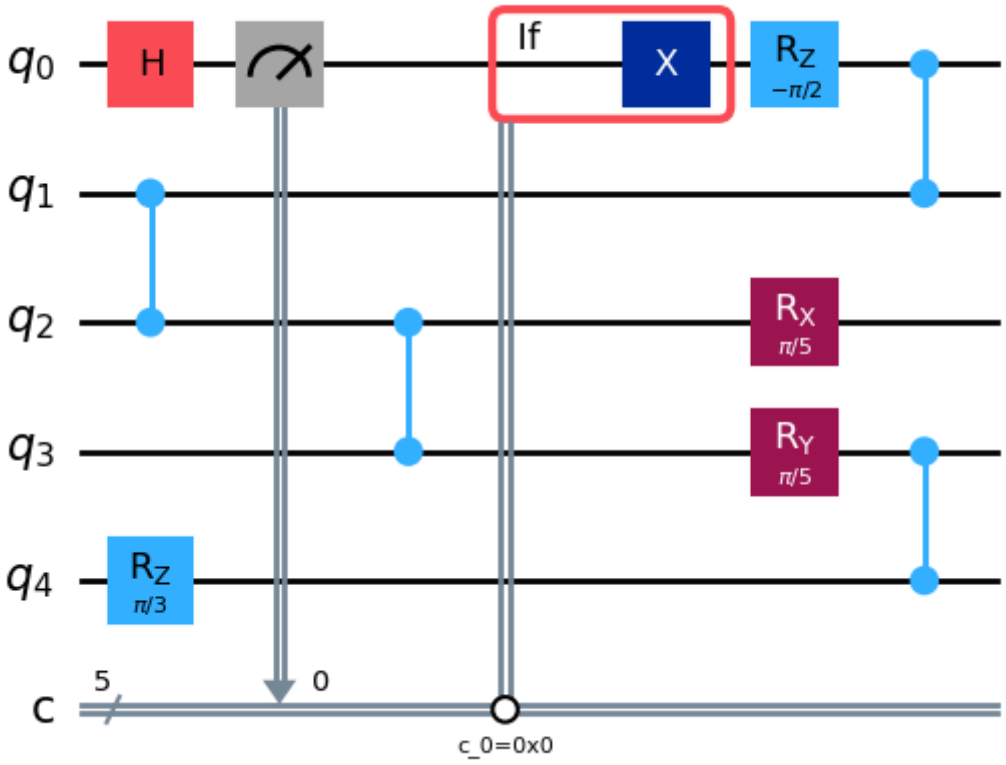
```
['CZ_2']
```

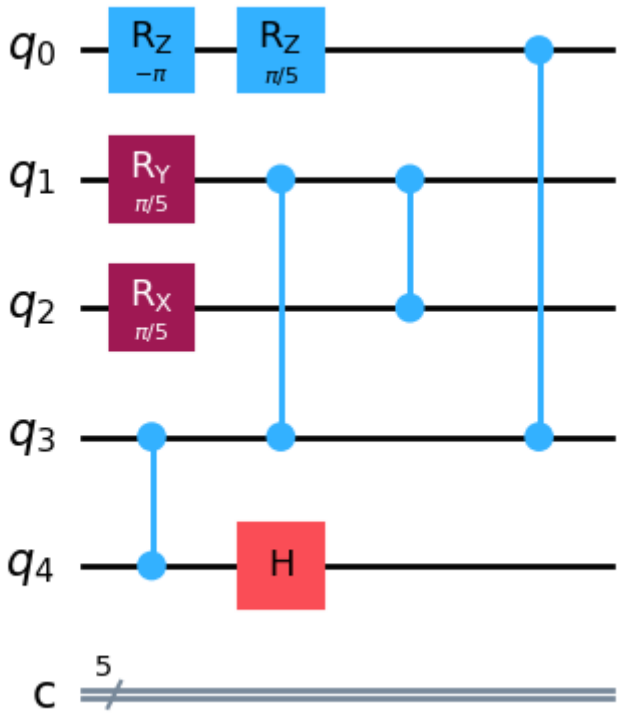


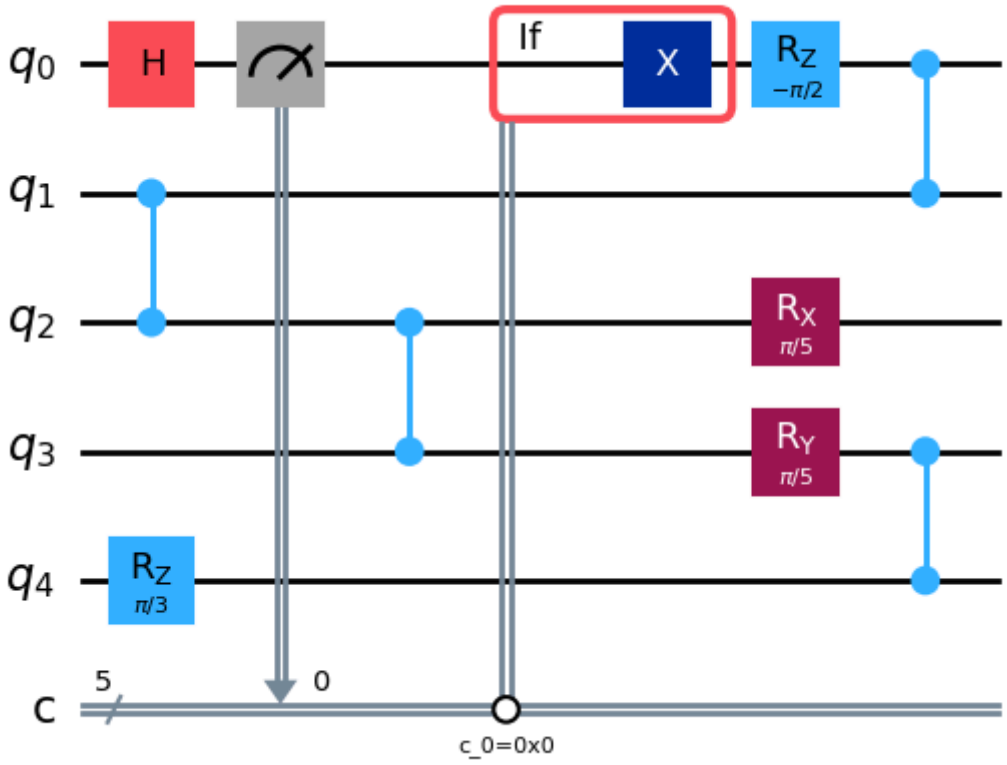

q_0
q_1
q_2
q_3
q_4
c
5
R_Z $-\pi/2$
R_Z $\pi/5$
R_Y $\pi/5$
R_X $\pi/5$
H

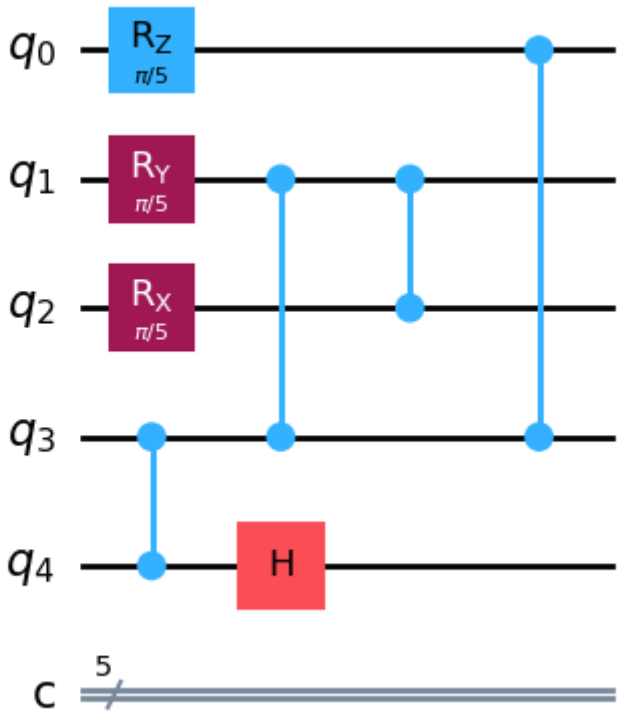

q_0
q_1
q_2
q_3
q_4
c
5
H
Z
R_Z $-\pi/2$
R_X $\pi/5$
R_Y $\pi/5$
R_Z $\pi/3$

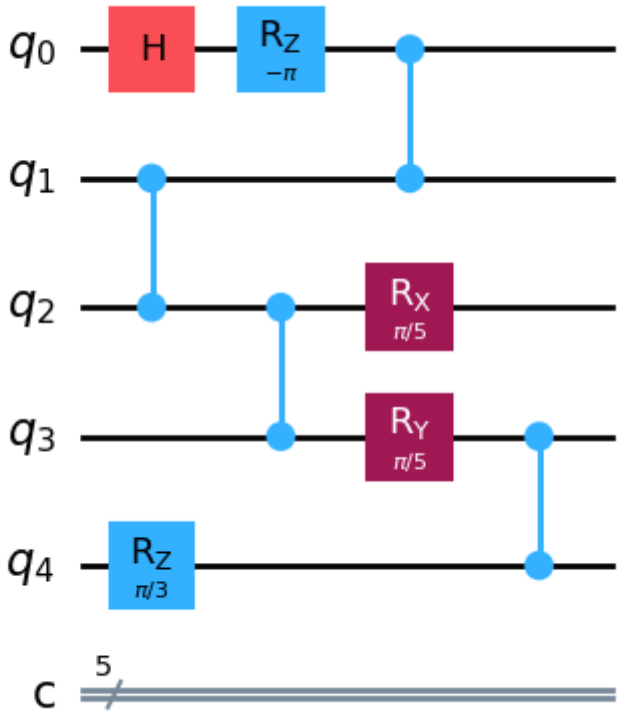

q_0
Z
R_Z $-\pi/2$
R_Z $\pi/5$
q_1
R_Y $\pi/5$
q_2
R_X $\pi/5$
q_3
q_4
H
c
5

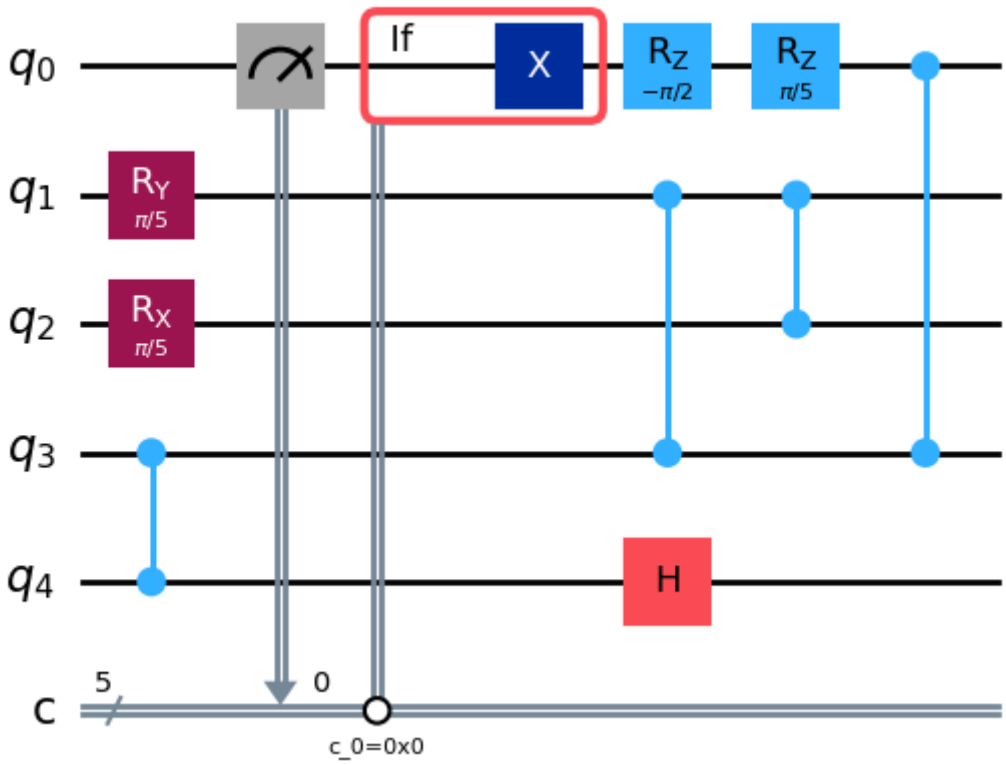

q_0
H
If
X
R_Z $-\pi/2$
q_1
q_2
R_X $\pi/5$
q_3
R_Y $\pi/5$
q_4
R_Z $\pi/3$
c
5
0
c_0=0x0

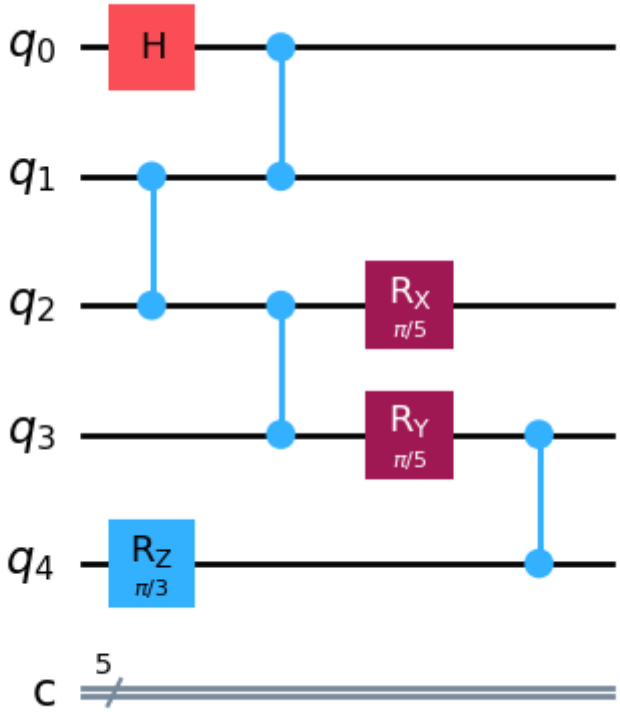

q_0
q_1
q_2
q_3
q_4
c
5
R_Z $-\pi$
R_Z $\pi/5$
R_Y $\pi/5$
R_X $\pi/5$
H

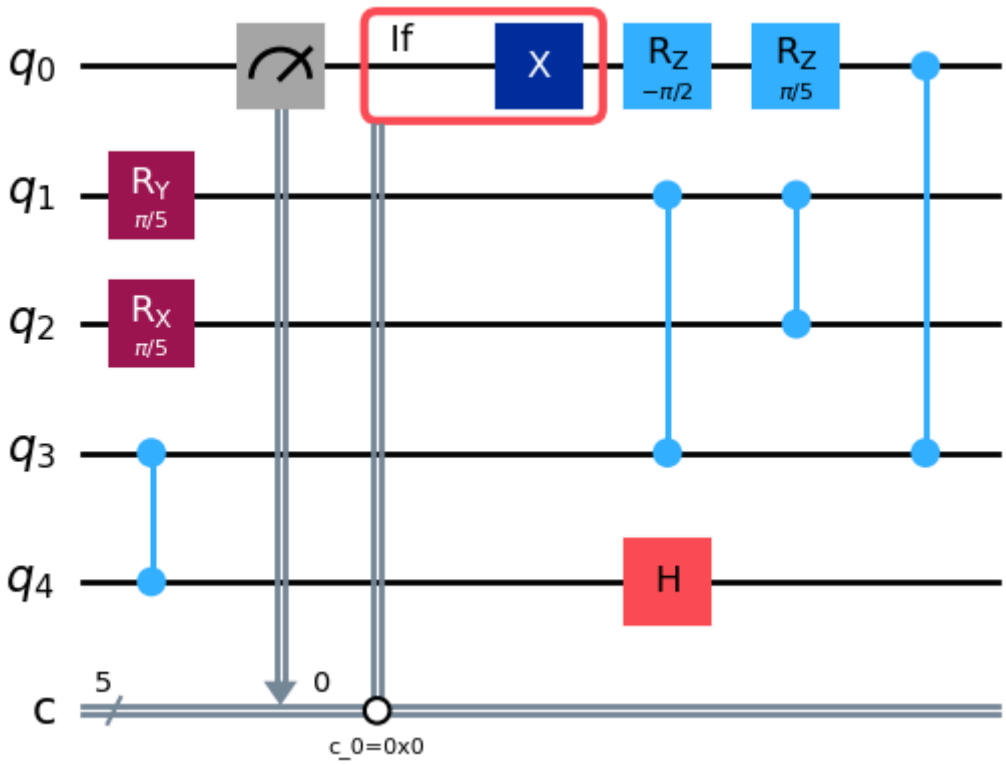

q_0
q_1
q_2
q_3
q_4
c
5
0
c_0=0x0
H
If
X
R_Z $-\pi/2$
R_X $\pi/5$
R_Y $\pi/5$
R_Z $\pi/3$


q_0
q_1
q_2
q_3
q_4
c
5
R_Z $\pi/5$
R_Y $\pi/5$
R_X $\pi/5$
H


q_0
q_1
q_2
q_3
q_4
c
5
H
R_Z $-\pi$
R_X $\pi/5$
R_Y $\pi/5$
R_Z $\pi/3$


q0
q1
q2
q3
q4
c
If
X
RZ
−π/2
RZ
π/5
RY
π/5
RX
π/5
H
5
0
c_0=0x0


q0
q1
q2
q3
q4
c
H
RX
π/5
RY
π/5
RZ
π/3
5



🧩 7.b Reconstruction Gate Cutting with Subcircuits

In this section, we demonstrate the reconstruction of the expectation value of a quantum circuit that has been partitioned into subcircuits using gate cutting. Each subcircuit is simulated independently, and their results are combined to approximate the expectation value of the original (uncut) circuit.

The process follows these main steps:

- Execute each subcircuit individually to obtain its contribution to the total expectation value.
- Use the reconstruction algorithm to combine the individual results and approximate the expectation value of the full circuit.

In [13]:
```
from Qdislib.core.cutting_algorithms.gate_cutting import _expec_value_qiskit

results = []
for subcircuit in subcircuits:
    # Execute individually each subcircuit
    result = _expec_value_qiskit(subcircuit)
    results.append(result)

results = compss_wait_on(results)

# Array with the individual expectation values of each subcircuit
```

```
print(results)

# Reconstruction of the original circuit expected value from the array of re
recons = qd.gate_cutting_subcircuit_reconstruction(results,number_cuts=1)
print(recons)
```

```
[-0.00390625, 0.044921875, 0.01171875, -0.009765625, -0.658203125, 0.0195312
5, -0.6484375, -0.09765625, -0.005859375, 0.013671875, -0.009765625, -0.0195
3125]
-0.03837013244628906
```

🧩 8. Wire Cutting with Subcircuits

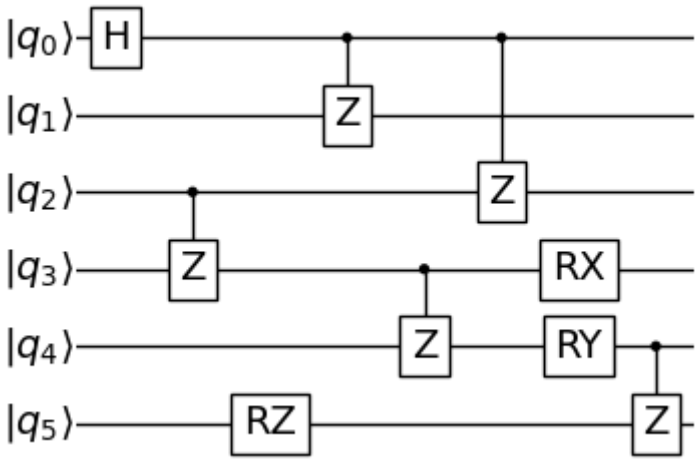
Same as gate cutting, but uses wire cutting logic.

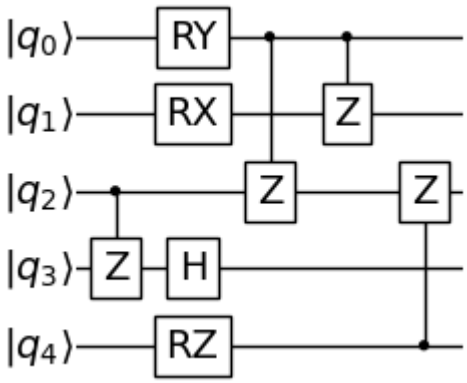
Subcircuits generated are more complex and may include measurements and resets.

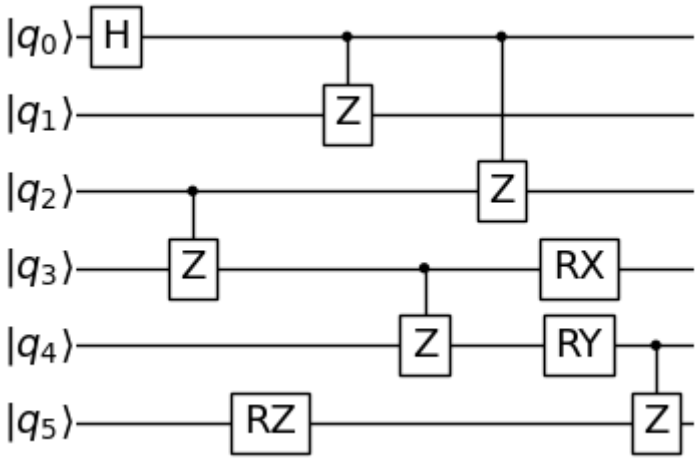
Visuals: Many circuit renderings show various Qiskit circuits built from the wire cut portions.

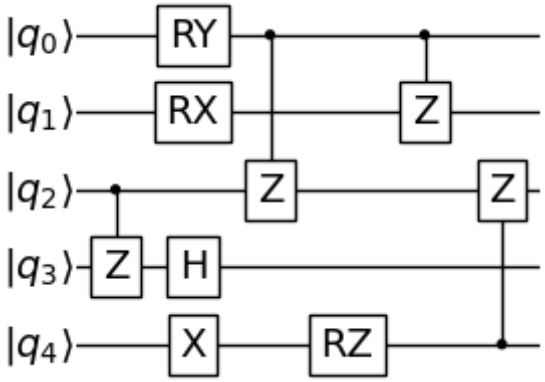
In [14]:
```
circuit = entire_circuit()

cut = qd.find_cut(circuit, gate_cut=False)
print(cut)
subcircuits = qd.wire_cutting_subcircuits(circuit, cut, "qibo")

subcircuits = compss_wait_on(subcircuits)

for subcirc in subcircuits:
    # print(subcirc)
    plot_circuit(subcirc, scale=0.5);
```

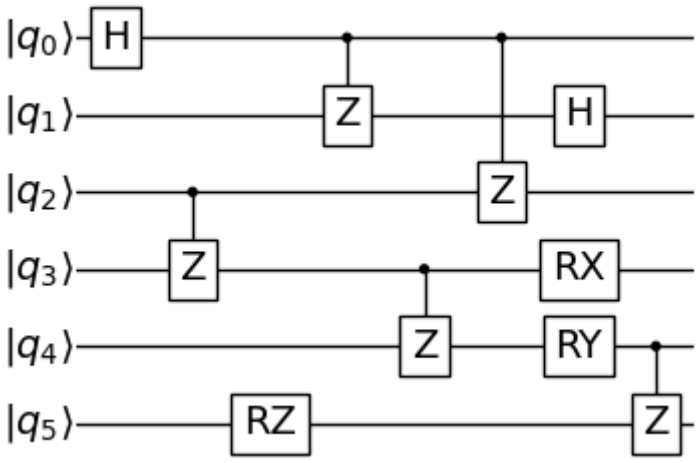
```
[('CZ_2', 'RZ_13')]
```

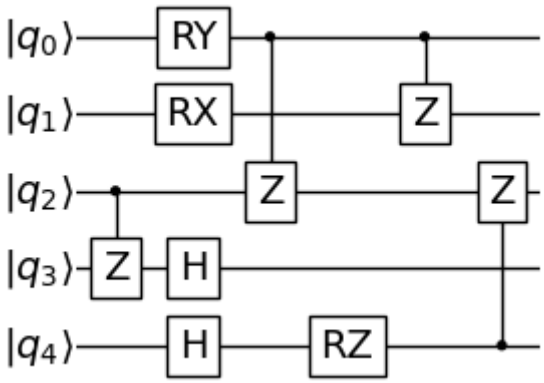


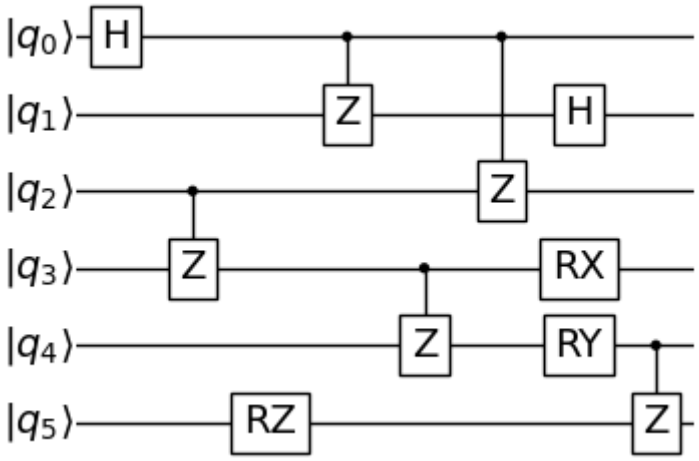

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
RY
RX
Z
Z
Z
Z
Z
H
RZ

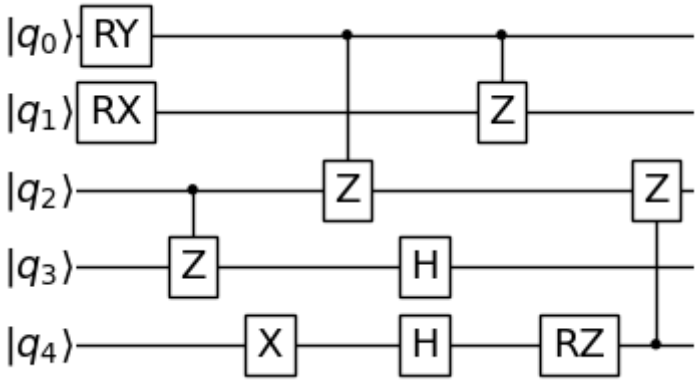

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
$|q_5\rangle$
H
Z
Z
Z
RX
Z
RY
RZ
Z

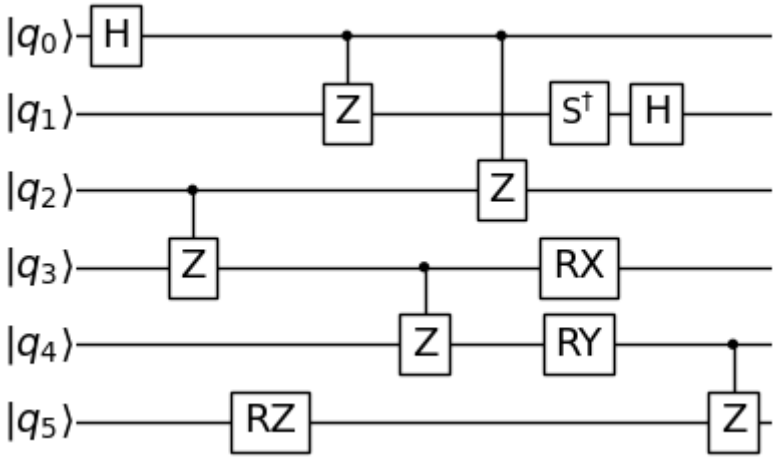

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
RY
RX
Z
Z
Z
Z
H
X
RZ

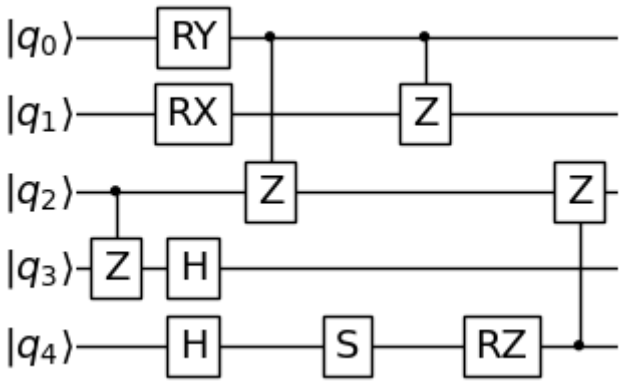

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
$|q_5\rangle$
H
Z
H
Z
Z
RX
Z
RY
RZ
Z

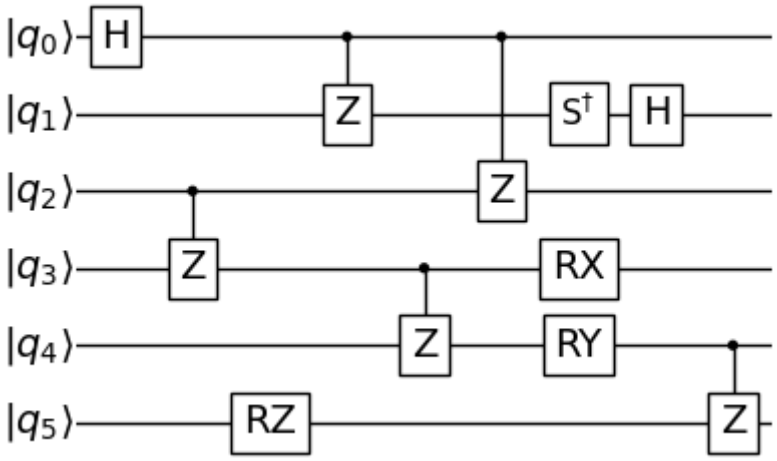

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
RY
RX
Z
Z
Z
Z
H
H
RZ

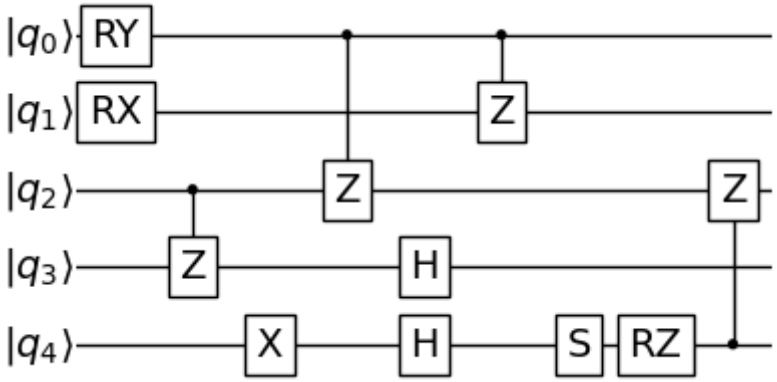

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
$|q_5\rangle$
H
Z
H
Z
Z
RX
Z
RY
RZ
Z

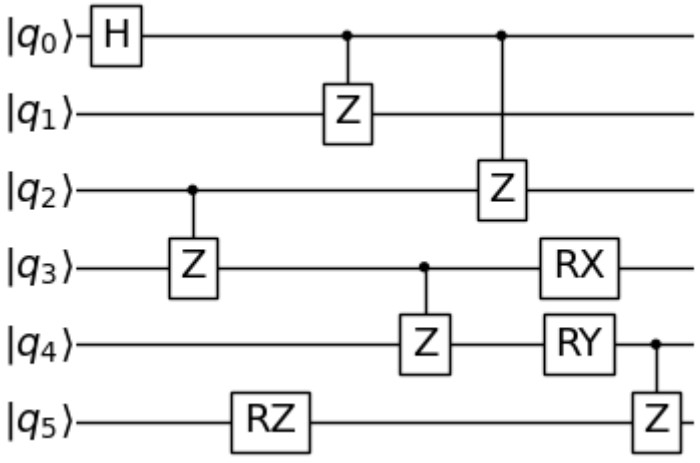

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
RY
RX
Z
Z
Z
Z
H
X
H
RZ

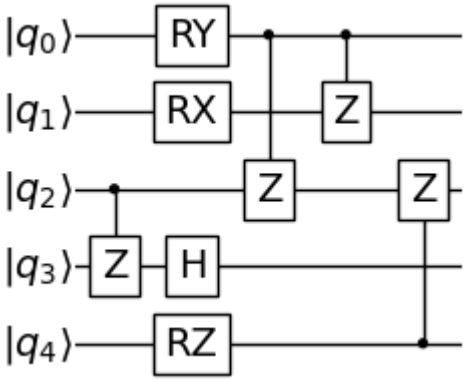

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
$|q_5\rangle$
H
Z
$S^\dagger$
H
Z
Z
RX
Z
RY
RZ
Z

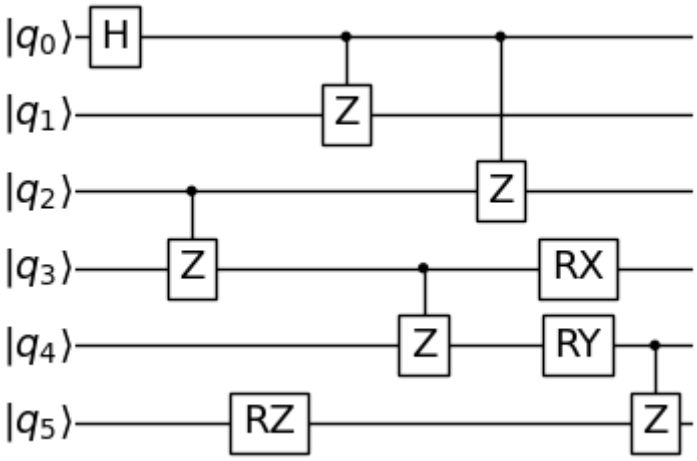

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
RY
RX
Z
Z
Z
Z
Z
H
H
S
RZ

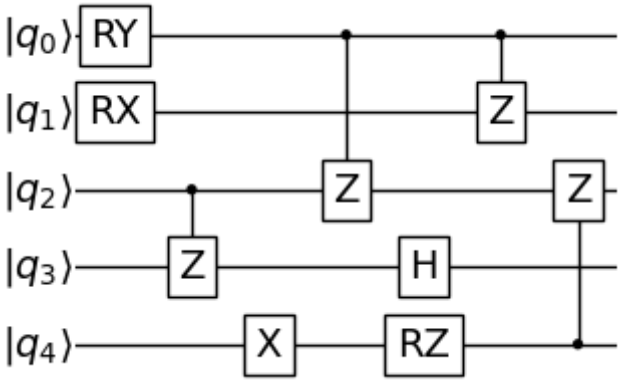

$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
$|q_5\rangle$
H
Z
Z
$S^\dagger$
H
Z
Z
RX
Z
RY
RZ
Z


$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
RY
RX
Z
Z
Z
Z
H
X
H
S
RZ


$|q_0\rangle$
$|q_1\rangle$
$|q_2\rangle$
$|q_3\rangle$
$|q_4\rangle$
$|q_5\rangle$
H
Z
Z
Z
RX
Z
RY
RZ
Z







🧩 8.b Reconstruction Wire Cutting with Subcircuits

In this section, we demonstrate the reconstruction of the expectation value of a quantum circuit that has been partitioned into subcircuits using wire cutting. Each subcircuit is simulated independently, and their results are combined to approximate the expectation value of the original (uncut) circuit.

The process follows these main steps:

- Execute each subcircuit individually to obtain its contribution to the total expectation value.

- Use the reconstruction algorithm to combine the individual results and approximate the expectation value of the full circuit.

In [15]:
```
from Qdislib.core.cutting_algorithms.wire_cutting import _expec_value_qibo

results = []
for subcircuit in subcircuits:
    # Execute individually each subcircuit
    result = _expec_value_qibo(subcircuit)
    results.append(result)

results = compss_wait_on(results)

# Array with the individual expectation values of each subcircuit
print(results)

# Reconstruction of the original circuit expected value from the array of re
recons = qd.wire_cutting_subcircuit_reconstruction(results,number_cuts=1)
print(recons)
```

```
[-0.02734375, -0.0546875, 0.015625, 0.029296875, -0.087890625, -0.02734375,
-0.01171875, -0.072265625, -0.015625, 0.068359375, 0.03515625, -0.04296875,
-0.052734375, 0.0, 0.02734375, 0.03125]
0.00154876708984375
```

🟧 9. Shutting Down PyCOMPSs

ipycompss.stop(sync=True):

- Stops the COMPSs runtime.
- Synchronizes any unresolved futures (e.g., cut, subcircuits).

Why it matters: Proper shutdown is required in PyCOMPSs to finalize all asynchronous tasks cleanly.

In [16]:
```
ipycompss.stop(sync=True)
```

```
**************************************************
**************** STOPPING PyCOMPSs ******************
**************************************************
Checking if any issue happened.
Synchronizing all future objects left on the user scope.
Found a list to synchronize: cut
Found a list to synchronize: subcircuits
Found a list to synchronize: results
Found a future object: result
         - Could not retrieve object: result
**************************************************
```

In [17]:
```
ipycompss.complete_task_graph(fit=True)
```