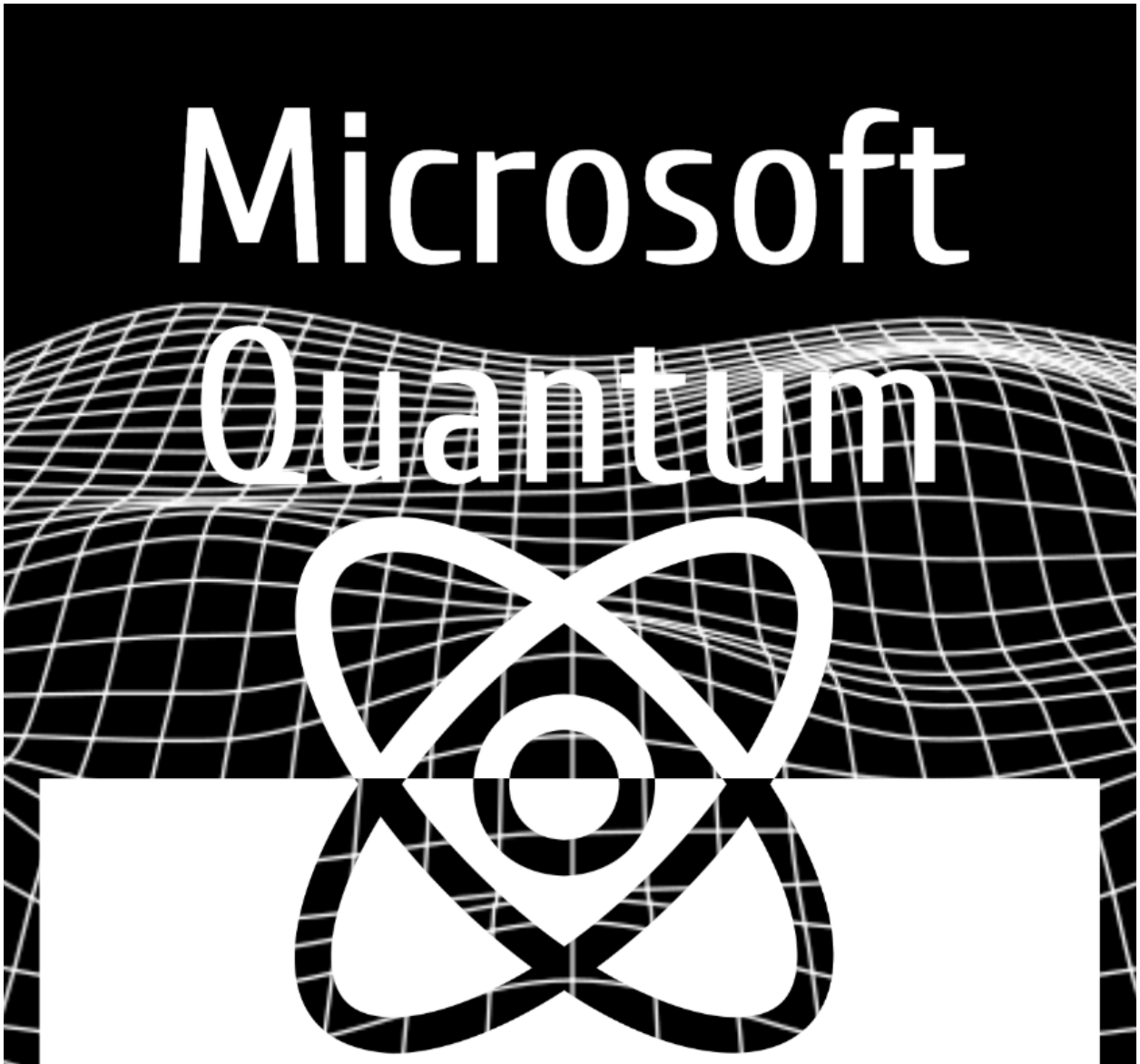


Quantum Resource Estimation of QAOA

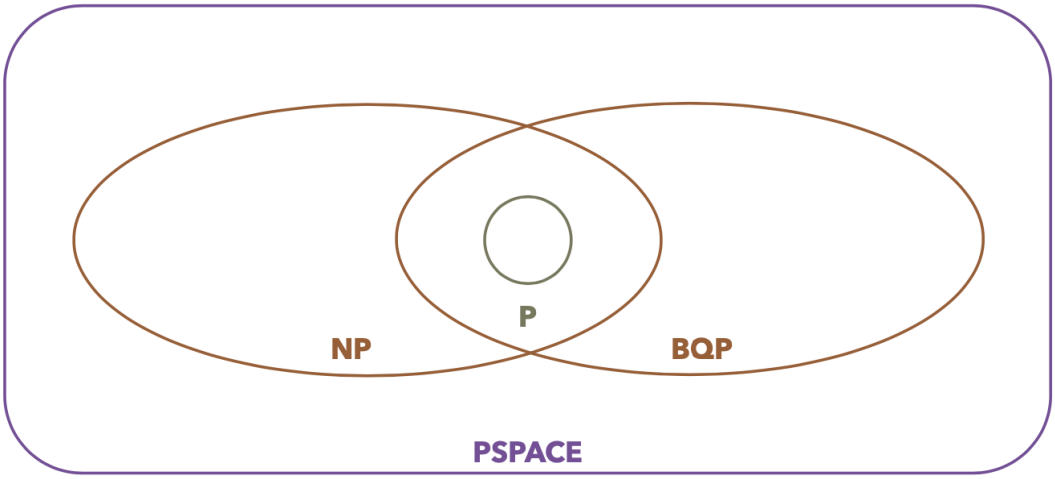
Qu-Cats "We are not dead yet"

Katie Harrison (@Katie1harrison)
Muhammad Waqar Amin (@Eagle-Eyes7)
Sarah Dweik (@Sarah-Dweik)
Nikhil Londhe (@nikhil-co)
Lucy Low (@lucylow)

Table of Contents

Practical Quantum Advantage

Quantum supremacy, also known as quantum advantage, refers to the point at which a quantum computer can outperform the best classical supercomputers for a specific task. In the context of QUBO (Quadratic Unconstrained Binary Optimization) numerical partitioning, achieving quantum supremacy would mean that a quantum algorithm can solve certain instances of the problem more efficiently than classical algorithms. Determining whether there is a quantum advantage for QUBO numerical partitioning compared to classical methods involves assessing various factors, including resource utilization, problem size, and algorithmic efficiency.



1. Problem Size and Complexity:

Quantum computers may offer advantages for solving large and complex optimization problems that are challenging for classical computers to tackle efficiently. For QUBO numerical partitioning, as the size of the input array increases, the problem becomes more computationally demanding, making it a suitable candidate for quantum optimization if classical methods struggle with scalability. Quantum algorithms for QUBO could demonstrate better scalability with problem size compared to classical algorithms. This means that as the size of the input problem grows, the quantum algorithm's performance advantage over classical methods becomes more pronounced. A quantum algorithm could provide a significant speedup in finding optimal or near-optimal solutions to QUBO instances compared to classical algorithms.

This speedup could be especially pronounced for large problem sizes where classical algorithms struggle due to computational complexity.

2. Resource Utilization:

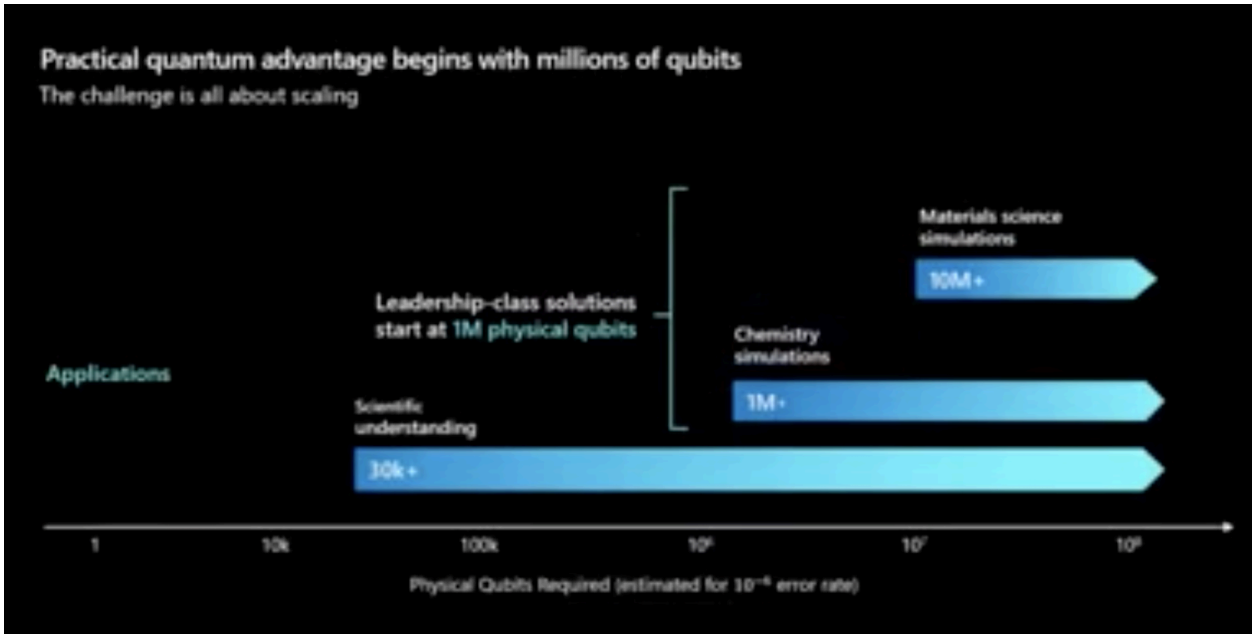
Resource estimation tools can provide insights into the computational resources required by both classical and quantum approaches for solving numerical partitioning problems of varying sizes. Quantum algorithms may exhibit different resource utilization patterns, such as parallelism in quantum operations, which could lead to more efficient resource usage for certain problem sizes. Theoretical predictions and experimental validations using resource estimators, such as the Microsoft Azure Quantum Resource Estimator, can help assess the feasibility and potential advantages of quantum algorithms for QUBO numerical partitioning.

3. Algorithmic Efficiency:

Quantum algorithms, including QUBO-based approaches, may leverage quantum parallelism and interference effects to explore solution spaces more effectively than classical algorithms. The efficiency of the QUBO numerical partitioning algorithm on a quantum computer depends on factors such as the quality of quantum gates, error rates, and the ability to encode and manipulate the problem efficiently. Quantum algorithms may require fewer computational resources (e.g., qubits, gates) to solve QUBO instances compared to classical algorithms. This efficiency could lead to reduced hardware requirements and energy consumption, making quantum computing more cost-effective for optimization tasks.

4. Quantum Advantage Assessment:

To determine if there is a quantum advantage for QUBO numerical partitioning, it's essential to compare the performance of quantum and classical methods for solving numerical partitioning problems across a range of problem sizes. Resource estimation, simulation, and benchmarking can help assess the relative strengths and weaknesses of classical and quantum approaches, providing insights into the existence and extent of quantum advantage.



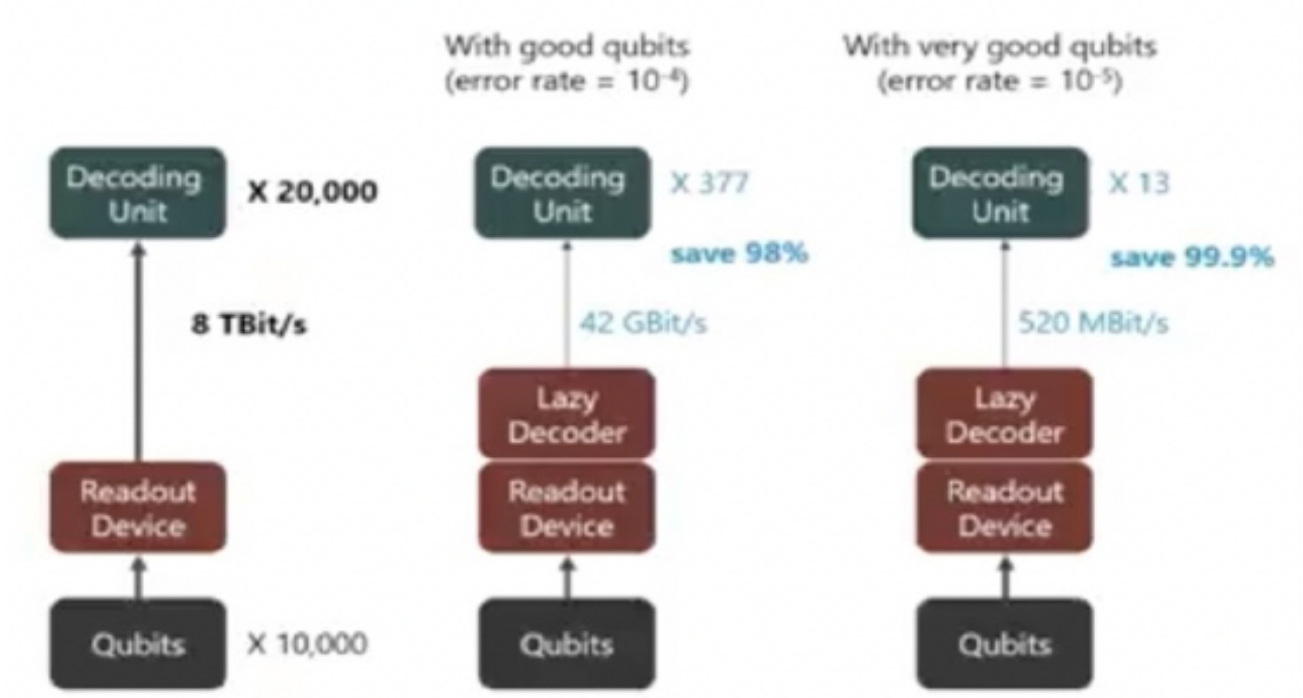
5. Iterative Optimization and Benchmarking:

Evaluating quantum advantage often involves iterative optimization and benchmarking of quantum algorithms against classical counterparts. By refining the QUBO algorithm, adjusting parameters, and optimizing quantum circuits, researchers can strive to maximize quantum advantage and demonstrate superior performance for specific problem instances.

In summary, determining whether there is a quantum advantage for QUBO numerical partitioning involves a comprehensive analysis of resource utilization, algorithmic efficiency, and problem size scalability. Through careful evaluation and comparison with classical methods, researchers can assess the potential benefits of quantum computing for solving numerical partitioning problems and identify scenarios where quantum advantage may be achieved. Quantum algorithms may enable the exploration of solution spaces that are difficult for classical algorithms to access efficiently. This could lead to the discovery of novel solutions or optimization strategies for QUBO problems. Ultimately, achieving quantum supremacy for QUBO numerical partitioning would represent a significant milestone in the field of quantum computing, demonstrating the practical utility of quantum algorithms for solving real-world optimization problems. However, it's essential to conduct thorough empirical studies and comparisons with classical methods to validate and quantify the extent of quantum advantage in specific application scenarios.

Hardware needed for Quantum Supremacy/Advantage

Practical Quantum Advantage refers to the point at which quantum computers outperform classical computers in solving real-world problems in a manner that is both significant and useful for practical applications. It signifies the achievement of computational tasks or problem-solving capabilities on a quantum computer that are beyond the reach of classical computers within a reasonable timeframe. Achieving practical quantum advantage involves demonstrating the superiority of quantum algorithms or quantum computation techniques over classical counterparts in terms of performance, accuracy, or scalability.



In the context of our QUBO numerical partitioning algorithm, practical quantum advantage would mean that the quantum algorithm can effectively and efficiently partition large datasets or solve optimization problems that are currently intractable for classical computers. It implies that the quantum algorithm provides substantial benefits in terms of speed, accuracy, or solution quality, leading to practical applications in fields such as logistics, finance, or scientific research. In summary, practical quantum advantage signifies the transformative potential of quantum computing to address real-world challenges and create new opportunities by surpassing the capabilities of classical computing systems in solving complex problems.

- Scalability: Our QUBO numerical partitioning algorithm must scale efficiently with the size of the input dataset and problem dimensions. This scalability ensures that the algorithm can handle large datasets encountered in practical applications, such as optimizing resource allocation in logistics or financial portfolio management.
- Low Error Rates and High Gate Fidelity: Low error rates and high gate fidelity are paramount for accurate optimization in QUBO numerical partitioning. Minimizing errors in qubit states, gate operations, and measurements ensures the reliability of partitioning solutions and improves the quality of optimization outcomes. High gate fidelity enhances the precision of gate operations, leading to more accurate partitioning results.
- Long Coherence Times: Long coherence times are necessary for maintaining the quantum states of qubits during the execution of our QUBO numerical partitioning algorithm. Extended coherence times enable sustained computation without significant decoherence, allowing for thorough exploration of solution spaces and enhancing the algorithm's ability to find optimal partitions.
- Rapid Readout and Initialization: Fast qubit readout and initialization capabilities accelerate the preparation of quantum states and measurement of partitioning solutions in our QUBO numerical partitioning algorithm. Rapid readout and initialization reduce processing overhead and result interpretation time, leading to faster optimization cycles and improved algorithm efficiency.
- Error Correction Support: Error correction support is critical for mitigating errors introduced during computation in our QUBO numerical partitioning algorithm. Implementing error correction codes ensures the reliability of partitioning results by correcting errors in qubit states or gate operations, enhancing the algorithm's robustness and accuracy.
- Cost-Efficiency: Cost-effective quantum hardware lowers the barrier to entry for adopting our QUBO numerical partitioning algorithm, making quantum optimization accessible for various applications and industries. Cost-efficient hardware encourages widespread adoption and innovation in numerical partitioning methodologies, driving advancements in optimization techniques and problem-solving capabilities.

This advantage may manifest in various ways, such as solving complex optimization problems more efficiently, simulating quantum systems with greater accuracy, or enabling the discovery of new materials or drugs through quantum chemistry simulations.

Use Case1: Financial Risk Modelling on a Transmon Device

This use case explores the implementation of quantum risk analysis algorithms on a transmon-based quantum computing platform. Transmon devices are a type of superconducting qubit architecture known for their low error rates and scalability, making them suitable for complex financial risk modeling tasks. By leveraging transmon qubits and quantum annealing techniques, financial institutions can analyze portfolio risk, market volatility, and other risk factors with increased speed and accuracy compared to classical methods. This implementation aims to demonstrate the potential of quantum computing in enhancing risk management practices in the financial industry.

Use Case2: Portfolio Optimization with Reverse Quantum Annealing

This use case investigates the application of reverse quantum annealing techniques for portfolio optimization tasks. Reverse quantum annealing involves initializing the quantum system in a highly entangled state representing a potential solution and then gradually relaxing it to find the optimal configuration that minimizes the portfolio risk or maximizes the expected return. By harnessing the power of quantum annealers and advanced optimization algorithms, financial institutions can efficiently optimize investment portfolios, balance risk and return objectives, and improve portfolio performance in dynamic market conditions.

Use Case3: Portfolio Optimization on a Trapped-Ion Device

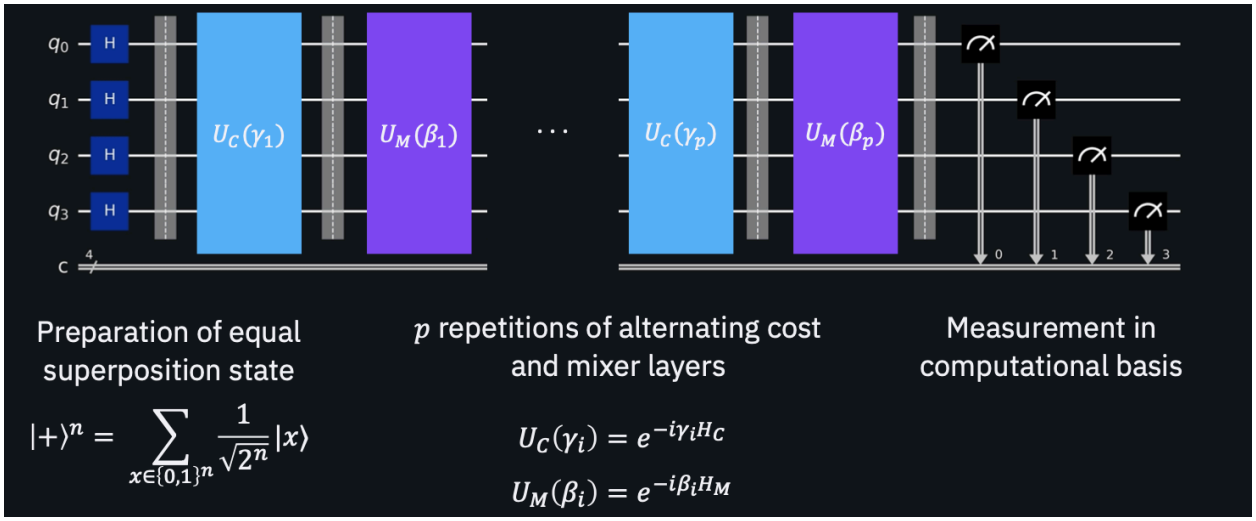
This use case explores the implementation of portfolio optimization algorithms on a trapped-ion quantum computing platform. Trapped-ion devices offer long coherence times and high-fidelity qubit operations, making them suitable for demanding optimization tasks such as portfolio optimization. By utilizing trapped-ion qubits and variational quantum algorithms, financial institutions can optimize investment portfolios, allocate assets efficiently, and hedge against market risks with improved precision and scalability. This implementation aims to demonstrate the potential of trapped-ion quantum computers in revolutionizing portfolio management strategies and driving innovation in the financial industry.

By meeting these conditions, our QUBO numerical partitioning algorithm can offer practical quantum advantage by efficiently solving large-scale optimization problems with improved accuracy, speed, and scalability compared to classical approaches.

Quantum Speedups

Brute Force Speed

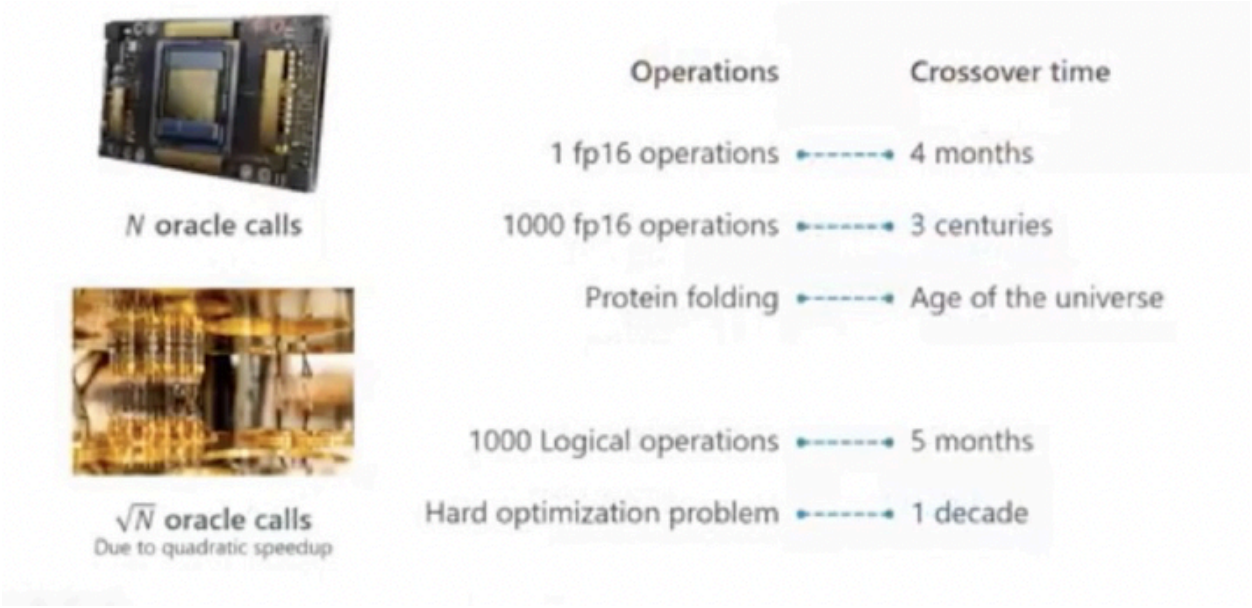
Brute force speed for QUBO numerical partitioning refers to the computational time required to exhaustively search through all possible partitions of the input array to find the optimal solution. In a brute force approach, every possible combination of subsets is evaluated, and the partition with the minimum absolute difference between the sums of the subsets is selected as the solution. The brute force speed for QUBO numerical partitioning depends on the size of the input array. As the size of the array increases, the number of possible partitions grows exponentially, leading to longer computation times. The time complexity of a brute force approach is \(O(2^N)\), where \(N\) is the size of the input array. This means that for each additional element in the array, the computational time roughly doubles. While brute force can guarantee finding the optimal solution for QUBO numerical partitioning, it becomes impractical for large input sizes due to the exponential increase in computational time. For moderately sized arrays, brute force may be feasible, but as the size of the array grows, the computational time quickly becomes prohibitively long.



Brute force approaches are generally less efficient for large-scale instances of QUBO numerical partitioning. Quantum algorithms have the potential to provide significant speedups by leveraging quantum parallelism and superposition to explore multiple solutions simultaneously. As a result, quantum algorithms may outperform brute force methods for large input sizes, offering faster computation times and more scalable solutions.

Quadratic Speedup

Quadratic speedup refers to the phenomenon where certain quantum algorithms exhibit a quadratic reduction in computational complexity compared to their best-known classical counterparts. This reduction allows quantum computers to solve specific problems much faster than classical computers, particularly for large input sizes.



In the context of QUBO numerical partitioning, achieving a quadratic speedup would mean that the quantum algorithm can solve the problem in a time complexity that scales quadratically with the input size, whereas the best-known classical algorithm would have a higher time complexity, such as cubic or higher. For QUBO numerical partitioning, demonstrating a quadratic speedup on quantum computers would require careful analysis, benchmarking, and comparison with classical algorithms across a range of problem sizes. If the quantum algorithm can consistently outperform classical methods with a quadratic reduction in time complexity, it would indicate the presence of a significant quantum advantage for numerical partitioning problems.

Achieving a qubic or quartic speedup would imply that the quantum algorithm can solve the numerical partitioning problem for large input sizes more efficiently than the best-known classical algorithms.

- A qubic speedup would mean that the quantum algorithm's time complexity scales cubically with the input size, while the best-known classical algorithm has a higher time complexity, such as quartic or higher.
- A quartic speedup would imply that the quantum algorithm's time complexity scales quartically with the input size, while the classical algorithm has an even higher time complexity, such as quintic or higher.

In practical terms, this would mean that as the size of the input array increases, the quantum algorithm's runtime grows significantly slower compared to classical algorithms, demonstrating its superiority in solving the numerical partitioning problem efficiently. However, achieving such speedups would require rigorous testing and analysis using both classical and quantum resources to validate the quantum advantage definitively.

However, achieving a quadratic speedup is not guaranteed for all problems and algorithms. It depends on various factors, including the inherent structure of the problem, the efficiency of the quantum algorithm, and the specific implementation on quantum hardware.

Quantum Speedups v.s. The Number of Physical Qubits

Achieving a quantum advantage through a quadratic speedup with the QUBO algorithm is theoretically possible. However, realizing this advantage in practice requires careful consideration of the computational overhead associated with each iteration of the algorithm. If each iteration involves computationally intensive operations, such as floating-point calculations, the quantum advantage may only become apparent after long computation times. In practical terms, this could mean that realizing a quantum advantage through the QUBO algorithm may necessitate computation durations spanning at least a month or more, especially if each iteration contains at least one floating-point operation.For QUBO numerical partitioning, achieving quantum speedups depends on effectively leveraging the capabilities of quantum hardware, such as the number of physical qubits, while addressing challenges such as error correction and algorithmic complexity.

The interplay between problem size, algorithm efficiency, and quantum hardware resources determines the extent to which quantum speedups can be realized for this specific problem domain.

1. Resource Requirements: Quantum algorithms for QUBO numerical partitioning, when implemented on quantum hardware, require a sufficient number of qubits to encode the problem. The number of qubits needed depends on the size of the input array or problem instance. Additionally, implementing quantum algorithms typically requires ancillary qubits for various operations, such as error correction or state preparation.

2. Problem Size: The size of the input array or problem instance directly influences the number of qubits required to encode it. Larger problem instances generally require more qubits. Quantum speedups for QUBO numerical partitioning may be more pronounced for larger problem sizes, as classical algorithms may struggle with scalability for such instances.

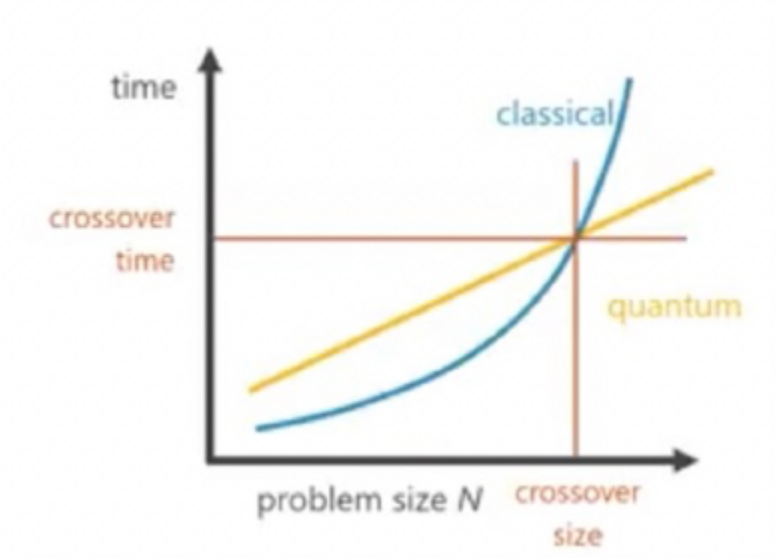
3. Error Correction: Quantum algorithms are susceptible to errors due to noise and decoherence. Error correction techniques, such as quantum error correction codes, require additional qubits to detect and correct errors. The overhead associated with error correction can impact the number of physical qubits needed and may affect the achievable speedup.

4. Algorithmic Complexity: The computational complexity of the quantum algorithm used for QUBO numerical partitioning affects its resource requirements and potential speedup. Quantum algorithms with efficient mappings of the problem to quantum states and low-depth quantum circuits may require fewer qubits and offer faster execution times.

5. Scalability: The scalability of quantum algorithms for QUBO numerical partitioning refers to how efficiently they can handle larger problem sizes with increasing numbers of qubits. Quantum algorithms that scale well with problem size and hardware resources have the potential to provide significant speedups over classical approaches for large-scale problems.

Reduced Time to Scale

Reducing the time to scale for QUBO numerical partitioning involves accelerating the development and deployment of quantum computing solutions that can efficiently handle larger problem sizes and deliver practical benefits. Here's how this can be achieved:



1. Hardware Innovation:
   - Quantum hardware advancements, such as increasing qubit counts, improving gate fidelities, and extending coherence times, can enable the execution of QUBO algorithms on larger problem instances.
   - Research efforts should focus on developing scalable and fault-tolerant quantum architectures optimized for QUBO numerical partitioning and other optimization problems.

2. Algorithmic Improvements:
   - Continuously refining and optimizing quantum algorithms for QUBO numerical partitioning can lead to faster convergence, reduced resource requirements, and improved solution quality.
   - Exploring novel algorithmic techniques, quantum variational algorithms, and hybrid classical-quantum approaches can enhance the efficiency and scalability of QUBO solvers.

3. Software Infrastructure:
   - Building robust software frameworks, quantum compilers, and optimization libraries tailored for QUBO numerical partitioning simplifies algorithm development, testing, and deployment.
   - Open-source collaboration and standardization efforts facilitate knowledge sharing, code reuse, and interoperability among different quantum computing platforms.

4. Quantum-Ready Applications:
   - Identifying and prioritizing real-world applications of QUBO numerical partitioning across various industries, such as finance, logistics, and supply chain management, drives demand for scalable quantum solutions.
   - Collaborating with domain experts and stakeholders to co-design quantum algorithms and applications ensures relevance and applicability in practical scenarios.

5. Cloud-Based Quantum Services:
   - Leveraging cloud-based quantum computing platforms, such as Microsoft Azure Quantum, Google Quantum AI, and IBM Quantum Experience, provides access to scalable quantum resources and tools for QUBO numerical partitioning.
   - Offering quantum-as-a-service (QaaS) solutions with pay-per-use models lowers the barrier to entry for organizations seeking to adopt quantum computing for optimization tasks.

6. Education and Training:
   - Investing in quantum education, training, and talent development programs equips researchers, developers, and practitioners with the skills and knowledge needed to harness quantum computing for QUBO numerical partitioning.
   - Collaboration between academia, industry, and government fosters a supportive ecosystem for quantum research, innovation, and workforce development.

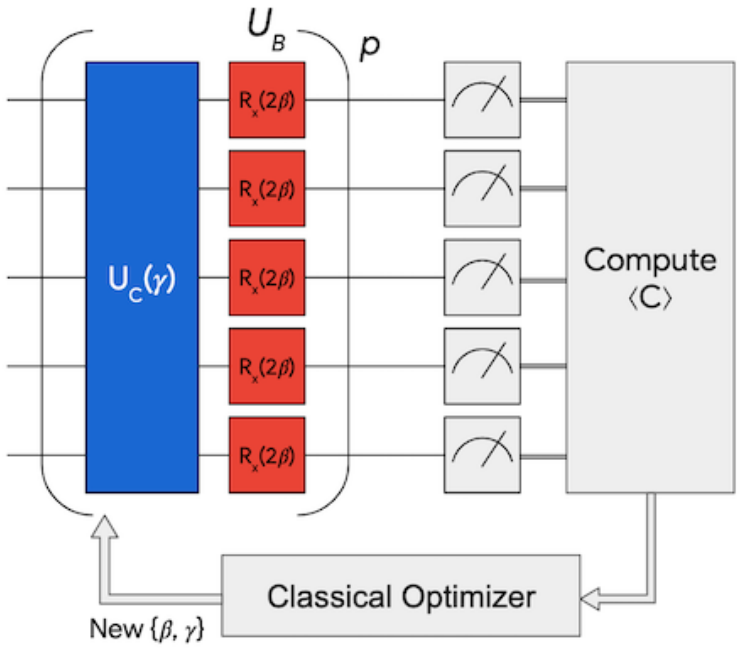
By accelerating progress in these key areas, the time to scale for QUBO numerical partitioning can be significantly reduced, unlocking the transformative potential of quantum computing for solving optimization challenges in diverse fields.

Classical v.s. Quantum: Quantum Approximate Optimization Algorithm

Classical and quantum numerical partitioning approaches represent two distinct paradigms for solving optimization problems, each with its own strengths and limitations.

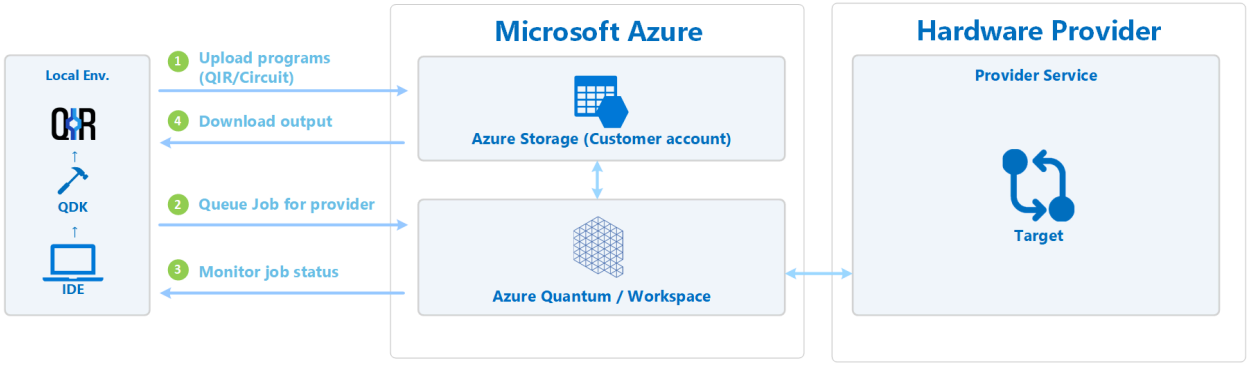
Quantum Approximate Optimization Algorithm (QAOA) is a hybrid Quantum Algorithm that is use to solve combinatorial optimization problems. It is based on the Adiabatic theorem which in its original form states that a physical system will remain in the instantaneous eigenstate if the perturbation is slow enough and if there is a gap between the eigenvalue and the rest of the Hamiltonian's spectrum. QAOA is designed to solve QUBO problem, so any optimization problem that can be formulated/embedded as QUBO problem can be solved by QAOA. QUBO share a special connection to Ising models as any QUBO is computationally equivalent to Ising models. A notable example of QUBO formulation is MAXCUT and it is important to note that QAOA provides approximate solutions.

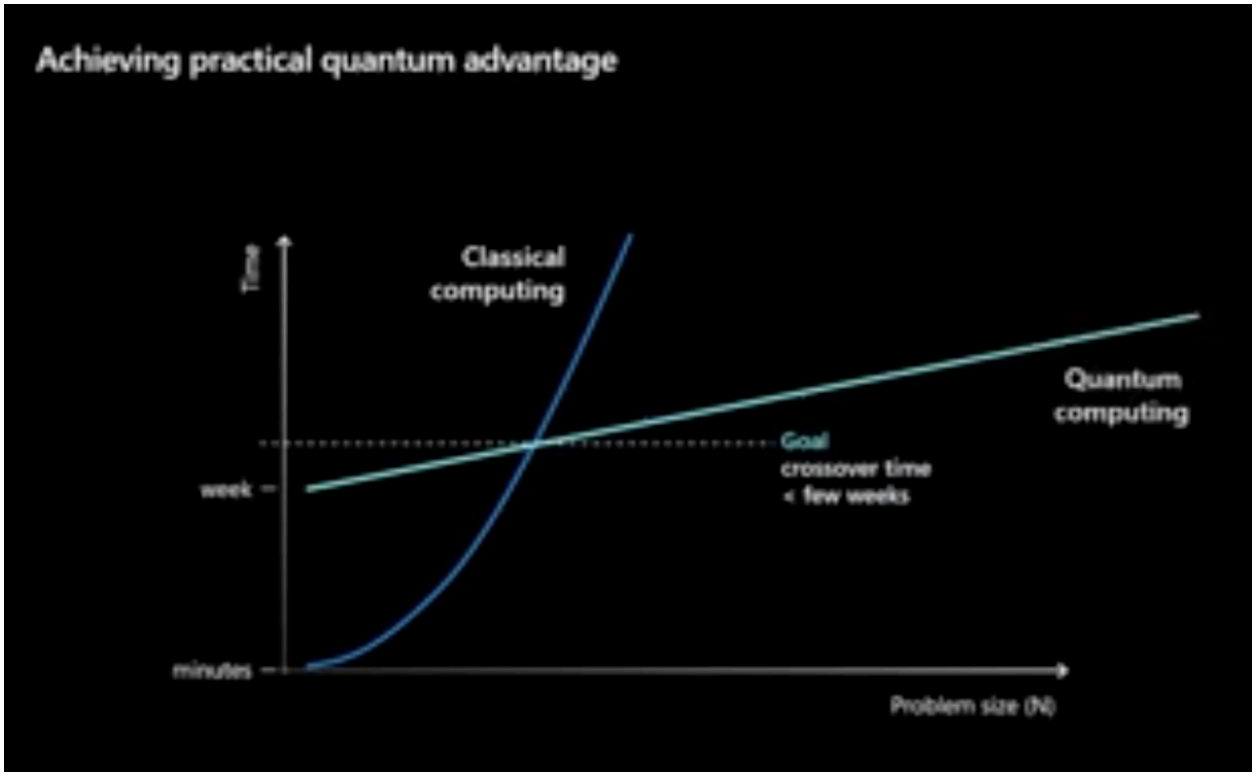
1. QAOA provides approximate solutions.
2. Is a special case of Variational Quantum Eigensolver (VQE)
3. Is the trotterized version of the Quantum Adiabatic Algorithm (QAA).



Numerical partitioning is NP-hard due to its inherent combinatorial nature and the exponential growth in computational complexity as the size of the input increases. In numerical partitioning, the goal is to divide a set of numbers into two or more subsets such that the sums of the numbers in each subset are as close to each other as possible. This problem is NP-hard because it requires examining all possible combinations of subsets to find the optimal partition, which becomes increasingly impractical as the input size grows. The time complexity to solve numerical partitioning problems scales exponentially with the size of the input, making it computationally intractable for large instances.

Resource estimation plays a crucial role in determining which quantum algorithms offer a genuine advantage over classical approaches and which ones do not. By quantifying the computational resources required by a quantum algorithm, such as the number of qubits, gate operations, or runtime, resource estimation provides insights into the feasibility and scalability of quantum solutions for practical problems. In the context of our QUBO numerical partitioning algorithm, resource estimation would involve assessing the hardware requirements, such as the number of qubits and gate operations, needed to execute the algorithm efficiently. By analyzing these resource requirements, one can evaluate whether the quantum algorithm offers a significant advantage over classical methods in terms of computational efficiency and solution quality.





Resource estimation helps identify the limitations and challenges associated with implementing quantum algorithms on current or near-future quantum hardware. It allows researchers and developers to assess the scalability of quantum solutions and determine whether they can handle increasingly complex problem instances or datasets. Resource estimation facilitates the comparison of different quantum algorithms for the same problem domain. By quantifying the resources needed by each algorithm, researchers can evaluate their relative performance and determine which approach is more promising for practical applications. Overall, resource estimation serves as a valuable tool for assessing the potential of quantum algorithms to provide genuine advantages over classical approaches. It enables informed decision-making in the development and deployment of quantum solutions, guiding efforts towards realizing the full potential of quantum computing for solving real-world problems.

	Classical	**Quantum**
Approach	In classical QUBO numerical partitioning, the problem is formulated as a Quadratic Unconstrained Binary Optimization (QUBO) problem, where the goal is to find a binary vector that minimizes a quadratic objective function subject to linear constraints.	In quantum QUBO numerical partitioning, the problem is mapped onto a quantum circuit, leveraging quantum principles such as superposition, entanglement, and interference to explore the solution space in parallel.
Solving Method	Classical algorithms such as simulated annealing, genetic algorithms, or exact solvers can be employed to search for the optimal solution by exploring the solution space iteratively.	Quantum algorithms, such as Quantum Approximate Optimization Algorithm (QAOA) or Variational Quantum Eigensolver (VQE), are used to evolve the quantum state towards the optimal solution through a series of quantum operations.
Resource Requirements	Classical QUBO numerical partitioning typically requires computational resources proportional to the problem size, including memory for storing the problem instance and processing power for solving it.	Quantum QUBO numerical partitioning requires access to quantum hardware or simulators capable of implementing the required quantum operations. The resource requirements include the number of qubits, gate operations, and coherence times.

Scalability	The scalability of classical QUBO numerical partitioning is limited by the computational complexity of classical algorithms, which can become prohibitive for large problem instances.	Quantum QUBO numerical partitioning offers the potential for exponential speedup over classical approaches for certain problem instances. However, scalability is currently limited by factors such as quantum hardware constraints, gate error rates, and decoherence effects.

Comparison:

1. Speed: Quantum algorithms have the potential to offer exponential speedup over classical algorithms for certain problem instances, enabling faster solution times.

2. Solution Quality: Quantum algorithms may provide better solutions in terms of objective function value compared to classical algorithms for some problems, but this depends on various factors such as algorithm design and hardware constraints.

3. Resource Requirements: While quantum algorithms have the potential to outperform classical algorithms in terms of computational efficiency, they currently require significant resources, including access to quantum hardware and expertise in quantum programming.

4. Practicality: Classical QUBO numerical partitioning algorithms are well-established and widely used in various applications, making them more practical for solving certain types of optimization problems in the near term. Quantum QUBO numerical partitioning, on the other hand, is still in the early stages of development and faces challenges related to hardware constraints and algorithmic maturity.

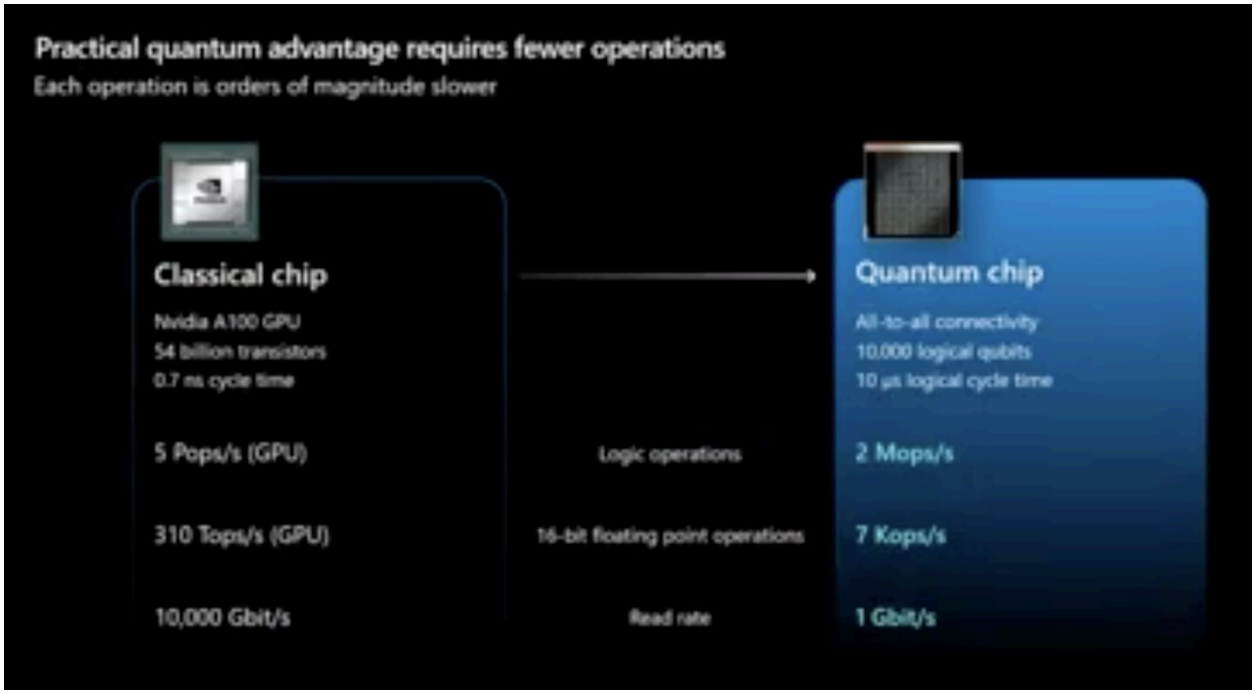
In summary, classical QUBO numerical partitioning offers a reliable and practical approach for solving optimization problems, while quantum QUBO numerical partitioning holds promise for achieving exponential speedup but is still in the early stages of development and requires advances in quantum hardware and algorithms to realize its full potential.

Classical v.s. Quantum: Numerical Partitioning

	Classical	Quantum
Resource Requirements	- Computational Resources: Classical QUBO numerical partitioning typically requires computational resources proportional to the problem size. This includes memory for storing the problem instance and processing power for solving it. - Memory: Classical algorithms often need to store the entire problem instance in memory, which can become a limiting factor for large problem sizes. - Processing Power: The computational complexity of classical algorithms can lead to long execution times, especially for problems with many variables or constraints.	- Quantum Hardware: Quantum QUBO numerical partitioning requires access to quantum hardware or simulators capable of implementing the required quantum operations. This includes the number of qubits available in the quantum processor. - Gate Operations: Quantum algorithms involve performing gate operations on qubits to evolve the quantum state towards the optimal solution. The number of gate operations required depends on the specific quantum algorithm used. - Coherence Times: Quantum operations need to be performed within the coherence times of the qubits to maintain the integrity of the quantum information. Longer coherence times allow for more complex quantum computations.

Scalability	- Limited Scalability: The scalability of classical QUBO numerical partitioning is limited by the computational complexity of classical algorithms. As the problem size increases, the computational resources required also increase exponentially, making it challenging to solve large-scale problems efficiently.	- Potential for Exponential Speedup: Quantum algorithms have the potential to offer exponential speedup over classical algorithms for certain problem instances. This means that as the problem size increases, the increase in computational resources required for quantum algorithms may be significantly lower compared to classical algorithms. - Hardware Constraints: Scalability is currently limited by factors such as the number of qubits available, gate error rates, and decoherence effects. Overcoming these hardware constraints is essential for realizing the full scalability potential of quantum QUBO numerical partitioning.

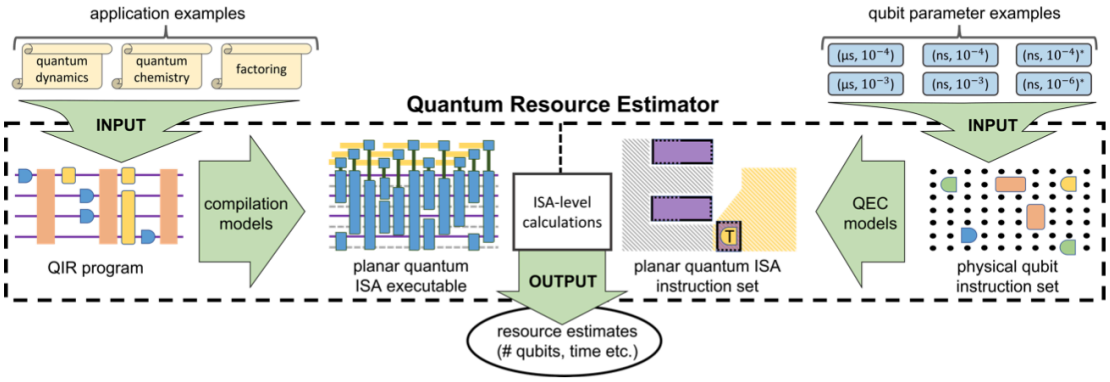
Quantum QUBO numerical partitioning requires access to quantum hardware, which introduces additional resource requirements compared to classical algorithms. However, quantum algorithms have the potential to offer exponential speedup, which can offset the increased resource requirements for certain problem instances. While classical algorithms face limitations in scalability due to the exponential increase in computational resources required for large problem sizes, quantum algorithms offer the potential for exponential speedup, making them more scalable for certain problems. However, scalability in the quantum domain is currently limited by hardware constraints and algorithmic maturity.



In summary, classical QUBO numerical partitioning relies on classical computational resources and faces limitations in scalability, while quantum QUBO numerical partitioning requires access to quantum hardware and offers the potential for exponential speedup but is currently limited by hardware constraints.

Scalability based on Input Parameters

The scalability of the QUBO algorithm depends on various input parameters and factors:



1. Problem Size: The scalability of the QUBO algorithm is directly influenced by the size of the problem instance, which is typically characterized by the number of variables or qubits in the problem formulation. As the number of variables increases, the computational resources required by the algorithm also increase, affecting its scalability.

2. Algorithmic Complexity: The scalability of the QUBO algorithm is affected by its algorithmic complexity, including the number of iterations or steps required to converge to a solution. Algorithms with higher complexity may experience slower scalability, especially for large problem sizes.

3. Hardware Resources: The scalability of the QUBO algorithm is limited by the availability and capacity of hardware resources, including the number of qubits and gate operations that can be executed simultaneously. Quantum algorithms require quantum hardware with sufficient resources to scale effectively.

4. Gate Operations: Quantum algorithms, including QUBO algorithms, consist of gate operations performed on qubits to manipulate the quantum state. The scalability of the algorithm depends on the efficiency of these gate operations and their impact on the overall computational complexity.

5. Error Rates: Quantum hardware is susceptible to errors due to factors such as decoherence and gate imperfections. Higher error rates can limit the scalability of quantum algorithms by affecting the fidelity of quantum operations and the reliability of the computed solutions.

6. Decoherence Effects: Decoherence causes quantum information to degrade over time, leading to errors and loss of coherence in quantum computations. Minimizing decoherence effects is crucial for maintaining the scalability of quantum algorithms, especially for long computation times or large problem sizes.

In summary, the scalability of the QUBO algorithm depends on a combination of factors including problem size, algorithmic complexity, hardware resources, error rates, and decoherence effects. Optimizing these factors and mitigating their impact can improve the scalability of the algorithm for larger problem instances.

Classical vs Quantum: Scaling

The scaling behavior of classical and quantum QUBO algorithms can differ significantly due to the fundamental differences in their underlying computational models and hardware implementations:

	Classical	Quantum
Problem Size	Classical QUBO algorithms typically scale quadratically with the number of variables or terms in the objective function. As the problem size increases, the computational resources required to solve the problem grow exponentially.	Quantum QUBO algorithms can potentially offer exponential scaling advantages over classical algorithms for certain problem classes. Quantum algorithms can encode and process information in superposition, allowing them to explore a larger solution space simultaneously.

Algorithmic Complexity	Classical QUBO algorithms often have well-defined polynomial-time complexity in terms of the problem size. However, certain optimization algorithms used to solve QUBO problems, such as integer linear programming (ILP) or simulated annealing, may have varying degrees of scalability depending on the problem structure and algorithmic parameters.	The algorithmic complexity of quantum QUBO algorithms is influenced by factors such as the number of qubits, gate operations, and circuit depth. Quantum algorithms may have different scaling behavior compared to classical algorithms due to their unique computational model.
Hardware Resources	Classical QUBO algorithms can leverage traditional computing resources such as CPUs or GPUs. The scalability of classical algorithms is limited by the available memory and processing power of the hardware.	Quantum QUBO algorithms require quantum hardware resources, including qubits, quantum gates, and quantum memory.
Error Rates	Classical algorithms are typically deterministic and do not suffer from intrinsic error rates like quantum algorithms. However, numerical precision and convergence issues may arise, especially for large-scale problems, affecting the reliability of solutions.	Quantum hardware is susceptible to errors caused by decoherence, gate imperfections, and environmental noise. Error correction techniques and error mitigation strategies are essential for maintaining the scalability and accuracy of quantum algorithms as the problem size increases.

In summary, classical QUBO algorithms scale polynomially with the problem size, while quantum QUBO algorithms have the potential for exponential scaling advantages. However, the practical scalability of quantum algorithms is currently limited by the development of fault-tolerant quantum hardware and efficient quantum error correction techniques.

Classical v.s. Quantum: Steps

Certain steps may take longer on a quantum computer compared to a classical computer, particularly when executing quantum operations that involve sequential processing, such as calling the T operation sequentially.

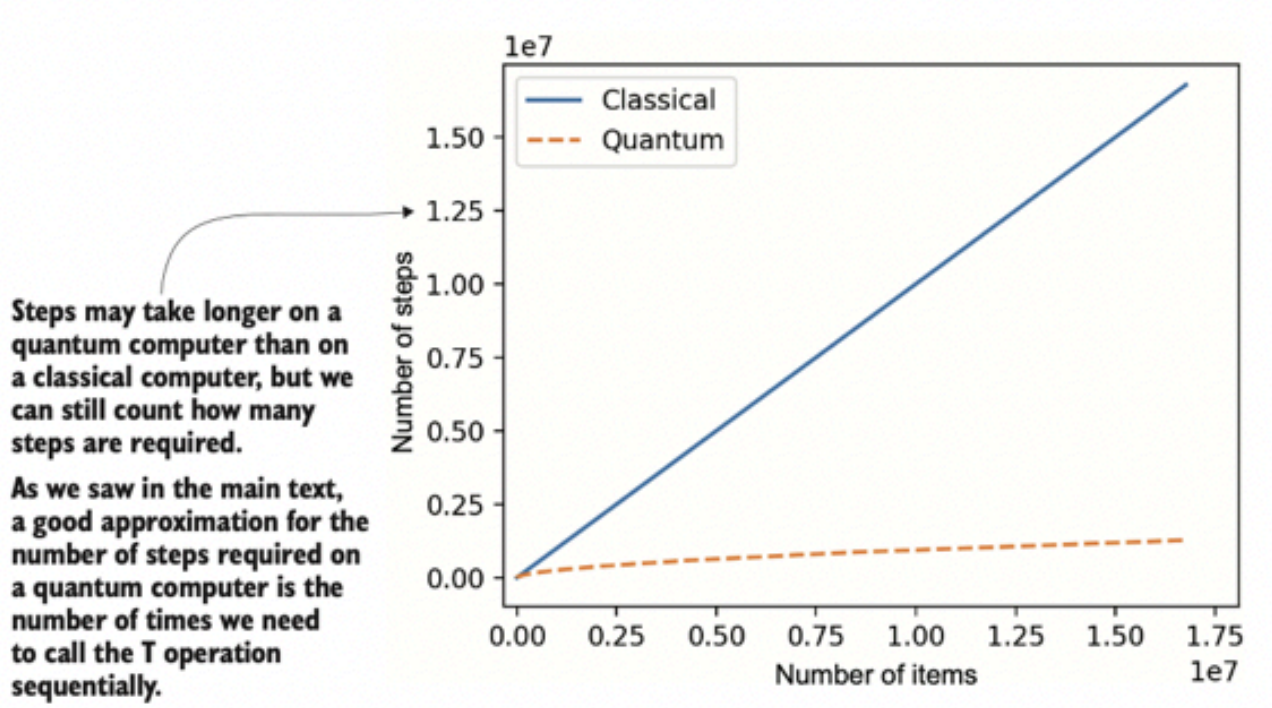
1. Sequential Processing in Quantum Operations:
   - Quantum operations, such as the T operation, are implemented as sequences of quantum gates that manipulate the state of qubits to perform specific computations.
   - On a quantum computer, these gates are applied sequentially, meaning that each gate operation must be completed before the next one can begin.
   - Unlike classical computers where multiple operations can often be executed simultaneously or in parallel, quantum computers generally operate sequentially due to the constraints imposed by quantum mechanics.

2. Gate Execution Time:
   - Each quantum gate operation on a quantum computer requires a certain amount of time to execute, determined by factors such as gate complexity, qubit connectivity, and hardware characteristics.
   - The execution time for individual gate operations can vary depending on the specific quantum hardware platform and its performance metrics, such as gate fidelity and coherence times.
   - Sequentially calling the T operation multiple times in a quantum circuit would involve executing each T gate operation one after the other, potentially leading to longer overall execution times compared to classical algorithms.

3. Quantum Hardware Limitations:
   - Quantum hardware is still in the early stages of development, and current quantum devices have limited qubit counts, gate fidelities, and coherence times.
   - Quantum algorithms must be implemented within the constraints of existing quantum hardware, which may result in longer execution times for certain quantum operations, especially when scaling up to larger problem sizes.

4. Optimization Challenges:
   - Designing quantum circuits to minimize execution time and optimize resource utilization is a non-trivial task that requires careful consideration of algorithmic structure, gate decompositions, and hardware constraints.
   - Optimizing quantum algorithms for performance on current and near-term quantum hardware remains an active area of research and development.

In summary, certain operations in quantum algorithms, such as sequential execution of quantum gates like the T operation, may take longer on a quantum computer compared to classical algorithms. Overcoming these challenges and improving the efficiency of quantum algorithms is crucial for realizing the full potential of quantum computing for practical applications.



Classical v.s. Quantum: Large Problem Sizes

To assess whether our QUBO numerical partitioning algorithm outperforms classical approaches for sufficiently large problem sizes, we can utilize the resource estimator tool.

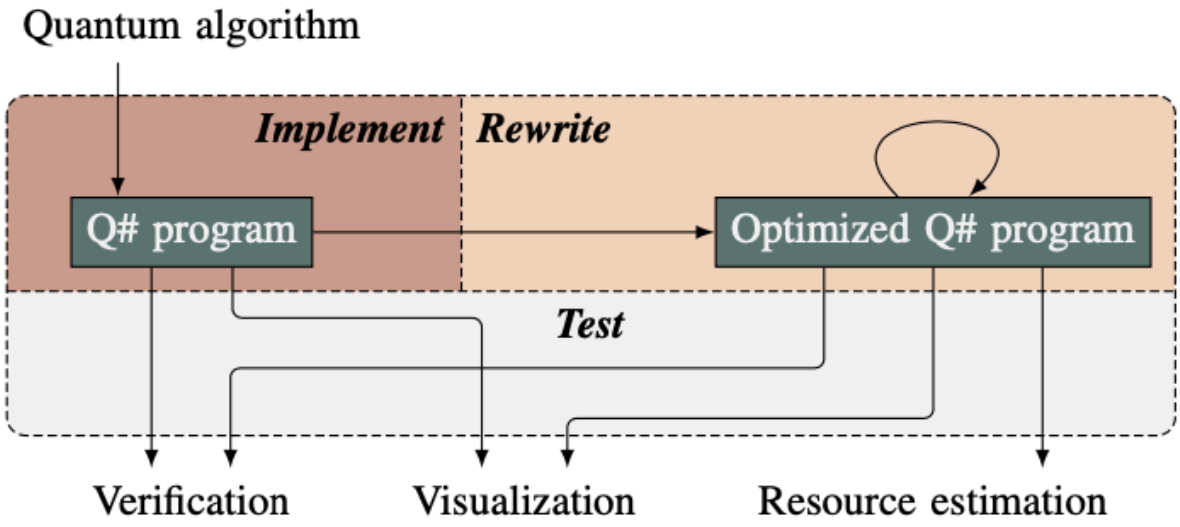
1. Resource Estimation for QUBO Algorithm:
   - We can use the resource estimator to analyze the resource requirements of our QUBO numerical partitioning algorithm for varying problem sizes.
   - By inputting the parameters of our algorithm, such as the array size and the number of layers in the quantum circuit, into the resource estimator, we can obtain estimates of the computational resources needed to execute the algorithm on a quantum computer.

2. Comparison with Classical Approaches:
   - Similarly, we can analyze the resource requirements of classical numerical partitioning algorithms, such as greedy algorithms or dynamic programming, for the same range of problem sizes.
   - The resource estimator provides insights into the computational resources, such as runtime and memory usage, required by classical algorithms to solve numerical partitioning problems of different sizes.

3. Scalability Analysis:
   - By comparing the resource estimates obtained for the QUBO algorithm with those of classical approaches, we can assess the scalability of our quantum algorithm.
   - If the resource estimates for the QUBO algorithm remain manageable and exhibit better scalability (i.e., slower growth in resource requirements with increasing problem size) compared to classical approaches, it indicates the potential for quantum advantage in solving larger numerical partitioning problems.

4. Verification of Quantum Advantage:
   - Running the program on the resource estimator allows us to verify whether the QUBO algorithm indeed outperforms classical approaches for sufficiently large problem sizes.
   - If the resource estimates demonstrate that the QUBO algorithm can efficiently handle larger problem sizes within reasonable resource constraints compared to classical approaches, it provides evidence of quantum advantage in numerical partitioning.

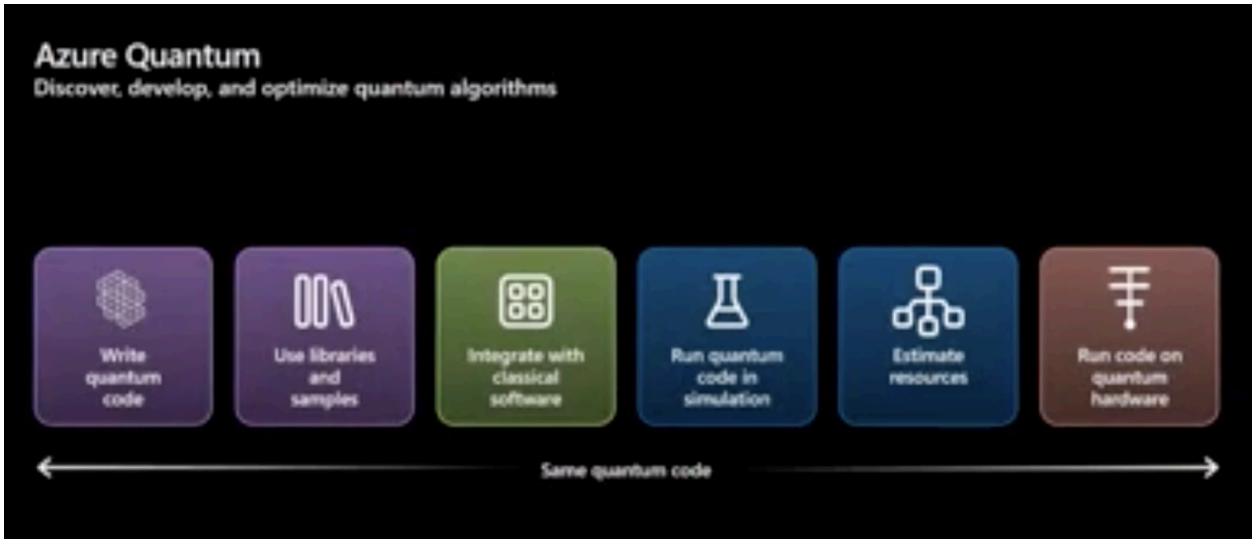
5. Iterative Optimization:
   - Based on the resource estimation results, we can iteratively optimize our QUBO algorithm by adjusting parameters such as the number of layers or qubit encoding schemes to enhance its performance.
   - The resource estimator serves as a valuable tool for guiding the optimization process by providing quantitative insights into the resource requirements and scalability of the algorithm.



By leveraging the resource estimator to compare the resource requirements of our QUBO numerical partitioning algorithm with classical approaches, we can gain valuable insights into its potential for achieving quantum advantage in solving large numerical partitioning problems.

Compare the Efficiency of Different Algorithms

Here are four algorithms commonly used for solving Quadratic Unconstrained Binary Optimization (QUBO) problems:



Name	Algorithm
Simulated Annealing	This is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It starts with an initial solution and iteratively explores the solution space by making random changes. Over time, it gradually decreases the probability of accepting worse solutions, leading to convergence towards the optimal solution.

Genetic Algorithms	Genetic algorithms are inspired by the process of natural selection and evolution. They work by maintaining a population of candidate solutions and iteratively applying genetic operators such as selection, crossover, and mutation to produce new candidate solutions. Through successive generations, genetic algorithms explore the solution space and converge towards optimal or near-optimal solutions.
Tabu Search	Tabu search is a local search algorithm that iteratively explores neighboring solutions to improve upon the current solution. It maintains a short-term memory, known as the tabu list, to avoid revisiting recently explored solutions. Tabu search effectively balances exploration and exploitation of the solution space and can find high-quality solutions for QUBO problems.
Quantum Approximate Optimization Algorithm (QAOA)	QAOA is a hybrid quantum-classical algorithm designed to solve combinatorial optimization problems, including QUBO. It uses a parameterized quantum circuit to prepare a quantum state representing a candidate solution and employs classical optimization techniques to adjust the parameters of the quantum circuit to minimize the objective function.

We compared the four algorithms to assess their performance in terms of solution quality, computational resources required, and runtime efficiency:

Solution Quality

Evaluate the quality of solutions produced by each algorithm in terms of their objective function value. Higher-quality solutions are those with lower objective function values for minimization problems like QUBO. You can compare the objective function values obtained by each algorithm and assess their optimality relative to known or benchmark solutions.

Computational Resources

Microsoft's Resource Estimator provides estimates of computational resources such as runtime, physical qubits, and other metrics. Compare these estimates with the resources required by each algorithm to solve the QUBO problem. Consider factors such as memory usage, parallelization capabilities, and scalability to determine the algorithm's resource efficiency.

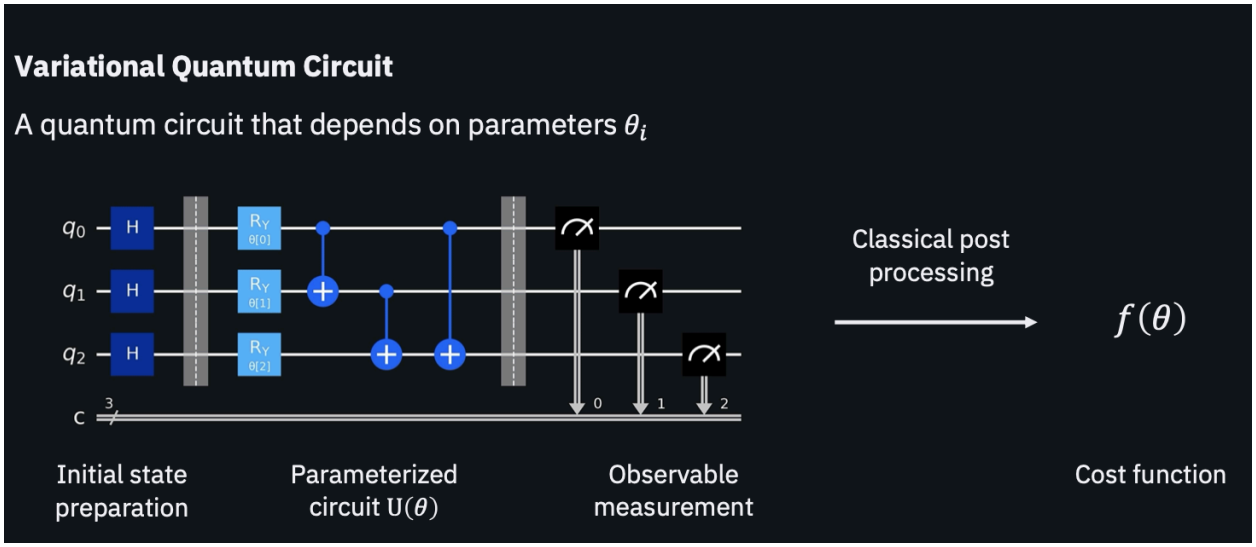
Runtime Efficiency

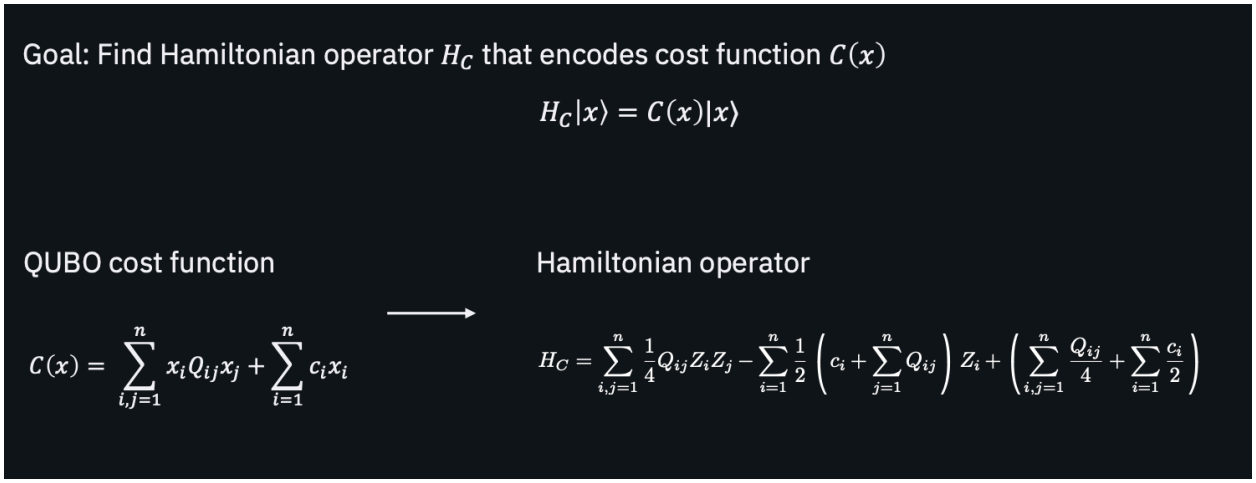
Evaluate the runtime efficiency of each algorithm in terms of the time required to find a solution. Measure the algorithm's performance on various problem instances of different sizes and complexities. Consider factors such as algorithmic complexity, convergence speed, and parallelizability to assess runtime efficiency.

By comparing these aspects, we can gain insights into the strengths and weaknesses of each algorithm and determine the most suitable approach for solving QUBO problems based on your specific requirements and constraints.

Quantum Intermediate Representation (QIR)

The Azure Quantum Resource Estimator utilizes Quantum Intermediate Representation (QIR), a standardized format for quantum programs. QIR serves as a bridge between various quantum programming languages and frameworks, allowing seamless interaction with targeted quantum computation platforms. As the Resource Estimator accepts QIR programs as input, it supports any language that compiles to QIR, including popular quantum SDKs such as Q# and Qiskit.





This is a program for simulating the QAOA circuit.

qaoa > src > Main.qs

```qsharp
// This is a program for simulating the QAOA circuit.
namespace qaoa{

    open Microsoft.Quantum.Measurement;

    // Function for getting flat index
    operation flat_index(n: Int, i: Int, j: Int): Int{
        return n*i + j
    }
    // Cost Hamiltonian
    operation cost_unitary(qubits: Qubit[], gamma: Double, quadratics: Double[], linears: Double[]): Unit{

        let n_qubits = Length(linears);
        mutable quad_sum : Double = 0.0;

        // RZ Gates
        for qubit in 0..n_qubits-1{
            set quad_sum = 0.0;
            for quad_qubit in 0..n_qubits-1{
                set quad_sum += quadratics[flat_index(n_qubits,qubit,quad_qubit)];
            }
            Rz(0.5 * (linears[qubit] + quad_sum) * gamma, qubits[qubit])
        }
        // RZZ Gates
        for i in 0..n_qubits-1{
            for j in i+1..n_qubits-1{
                Rzz(0.25 * quadratics[flat_index(n_qubits,i,j)] * gamma, qubits[i], qubits[j])
            }
        }
    }

    // Mixer Hamiltonian
    operation mixer_unitary(qubits: Qubit[], beta: Double) : Unit{
        for qubit in qubits{
            Rx(2.0 * beta,qubit);
        }
    }

    // Function to create the QAOA circuit.
    operation circuit(NQubits: Int, Layers: Int, gammas: Double[], betas: Double[], quadratics: Double[], linears: Double[]) : Int {

        use q = Qubit[NQubits];
        mutable integer_result = 0;

        // State Preparation |+>
        ApplyToEachA(H,q);

        for layer in 0..Layers-1{
            cost_unitary(q, gammas[layer], quadratics, linears);
            mixer_unitary(q, betas[layer]);
        }
        // Return the bitstring as an integer.
        return MeasureInteger(q);
    }
}

```

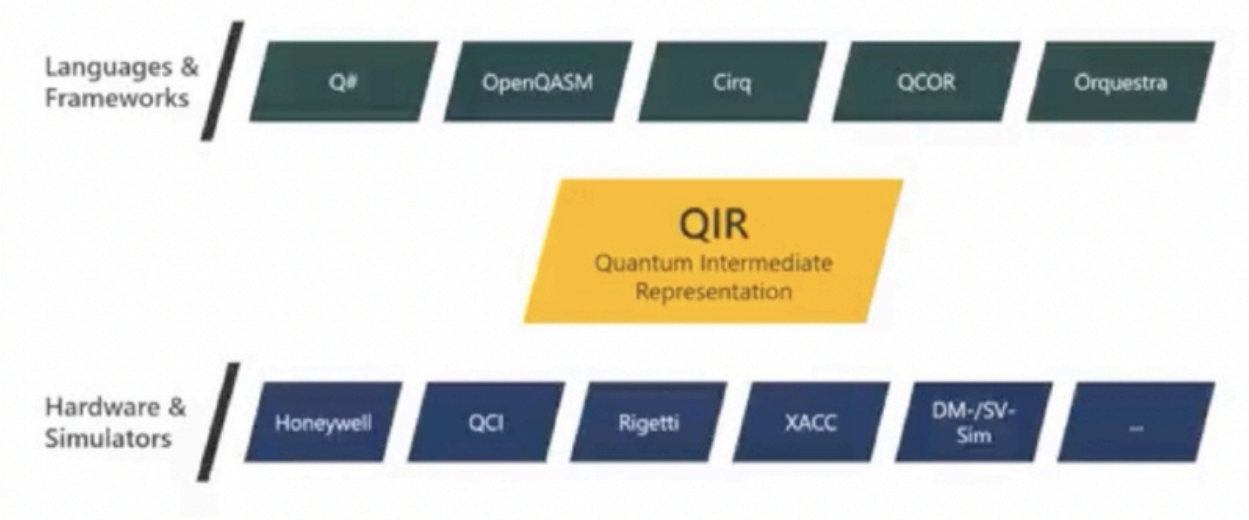
Define a Function to Create a Resource Estimation Job from QIR
Create a function that takes the `provider` object and QIR `bitcode` as input and returns an Azure Quantum job. This function allows you to specify Resource Estimator target parameters such as `errorBudget`, `qecScheme`, `qubitParams`, and `constraints`.
```python
def resource_estimation_job_from_qir(provider: AzureQuantumProvider, bitcode: bytes, kwargs):
    """A generic function to create a resource estimation job from QIR bitcode"""

    backend = provider.get_backend('microsoft.estimator')
    name = kwargs.pop("name", "QIR job")
    config = backend.configuration()
    blob_name = config.azure["blob_name"]
    content_type = config.azure["content_type"]
    provider_id = config.azure["provider_id"]
    output_data_format = config.azure["output_data_format"]
    return AzureQuantumJob(
        backend=backend,
        target=backend.name(),
        name=name,
        input_data=bitcode,
        blob_name=blob_name,
        content_type=content_type,
        provider_id=provider_id,
        input_data_format="qir.v1",
        output_data_format=output_data_format,
        input_params=kwargs,
        metadata={}
    )
```
Run a Sample Quantum Program
Generate QIR bitcode using the PyQIR generator. The example below constructs a controlled S gate using T and CNOT gates.
```python
module = SimpleModule("Controlled S", num_qubits=2, num_results=0)
qis = BasicQisBuilder(module.builder)
[a, b] = module.qubits[0:2]
qis.t(a)
qis.t(b)
qis.cx(a, b)
qis.t_adj(b)
qis.cx(a, b)
```

Use the defined function and PyQIR's `bitcode()` function to generate a resource estimation job. You can specify Resource Estimator parameters such as `errorBudget`. This example sets the error rate to 5%.

```python
job = resource_estimation_job_from_qir(provider, module.bitcode(), errorBudget=0.05)
result = job.result()
result
```

This function generates a table displaying the overall physical resource counts. Further details such as error correction code distance and physical qubit properties can be inspected. For example, the error correction code distance is 15, and physical qubit parameters include gate times and error rates.



By leveraging QIR and the Azure Quantum Resource Estimator, you can estimate the resources required for solving QUBO problems, enabling efficient utilization of quantum computing resources.

Counting Resources required to run QUBO algorithm

1. Preparing the Quantum State:

- In the QUBO partitioning algorithm, preparing the quantum state involves initializing a set of qubits in a superposition state.

- This initialization is typically achieved using Hadamard gates (`H`) to create an equal superposition of all possible states.

- The qubits are prepared in such a way that they represent the variables or bits of the QUBO problem.

2. Performing Measurements to Identify the Quantum State:

- After preparing the quantum state, measurements are performed on the qubits to extract information about the state.

- The outcome of the measurements provides classical information that represents a potential solution to the QUBO problem.

3. Implementing Required Unitary Transformations:

- In the QUBO partitioning algorithm, implementing the required unitary transformations involves applying quantum gates that represent the objective function or constraints of the QUBO problem.

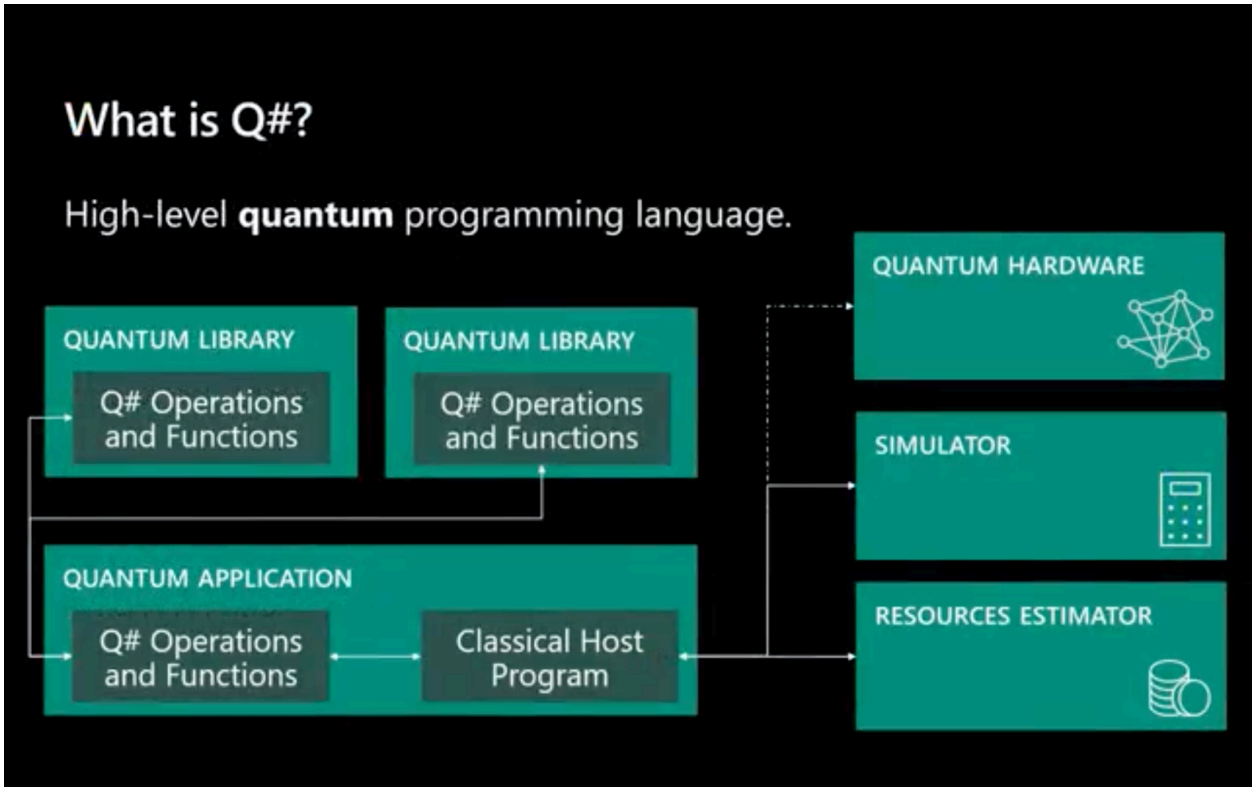
- For example, a quantum oracle can be implemented as a unitary transformation that encodes the objective function of the QUBO problem.

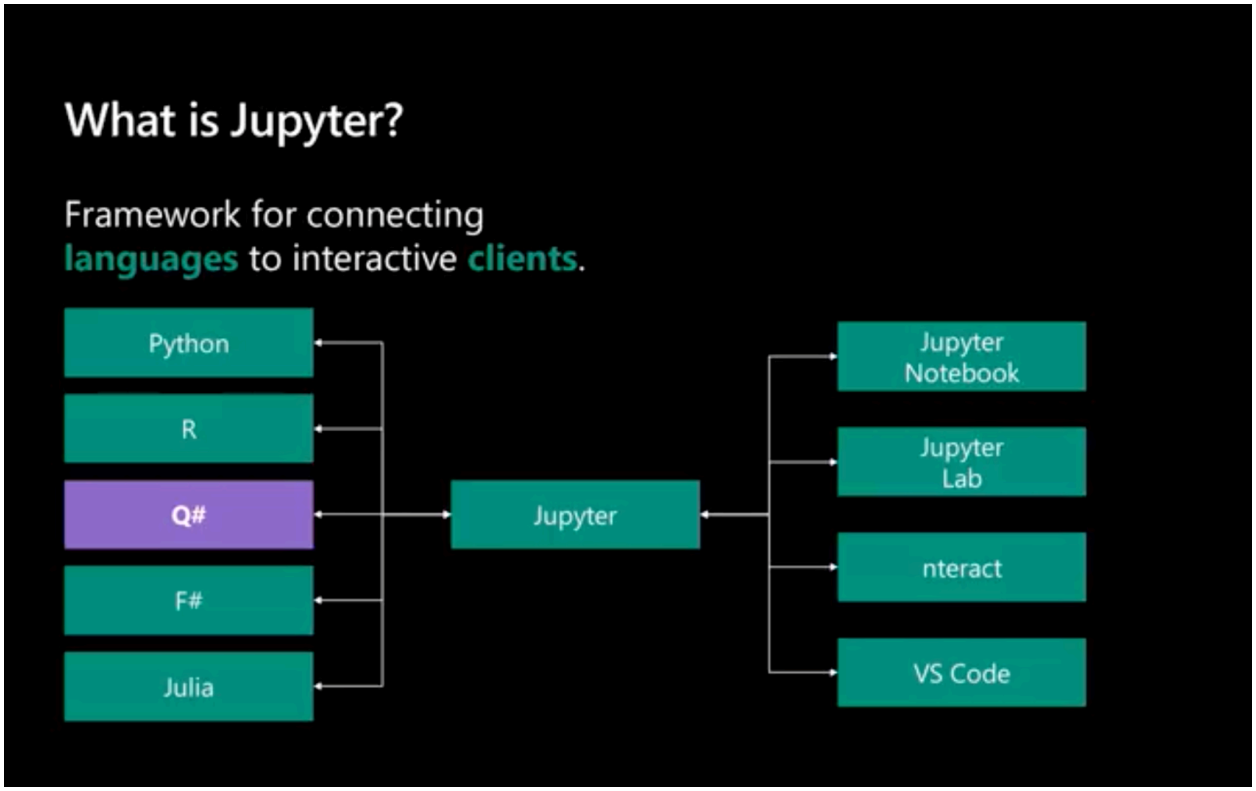
- Other unitary transformations may represent additional constraints or optimization objectives.

- These transformations manipulate the quantum state of the qubits to encode the problem's structure and guide the quantum search towards optimal solutions.

Overall, the QUBO partitioning algorithm combines quantum state preparation, measurement, and unitary transformations to encode and solve the QUBO problem on a quantum computer. Each step plays a crucial role in leveraging quantum properties to efficiently search the solution space and find optimal or near-optimal solutions.

Github Instructions





Prerequisites

Ensure you have the following prerequisites installed:
- The latest version of Visual Studio Code or open VS Code on the Web.
- The latest version of the Azure Quantum Development Kit extension. Refer to the installation instructions for Installing the Modern QDK on VS Code.

Tip: You do not need an Azure account to run the local Resource Estimator.

Load a QUBO Sample Program

Follow these steps to load a QUBO sample program:
1. In VS Code, select File > New File and save the file as `RandomNum.qs`.
2. Open `RandomNum.qs` and type `sample`, then select `Random Bit sample` and save the file.

Run the Resource Estimator

Execute the following steps to run the Resource Estimator:
1. Open the Resource Estimator window by selecting View -> Command Palette or pressing Ctrl+Shift+P. Then type “resource” to bring up the option "Q#: Calculate Resource Estimates" and select it.
2. Choose one or more Qubit parameter + Error Correction code types to estimate the resources. For this example, select `qubit_gate_us_e3` and click OK.
3. Specify the Error budget or accept the default value (0.001).
4. Press Enter to accept the default result name based on the filename, which in this case is `RandomNum`.

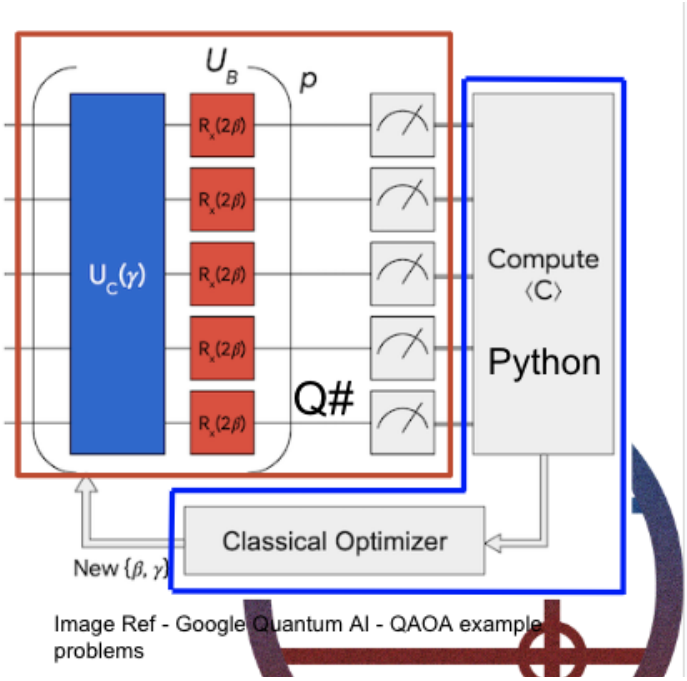


Image Ref - Google Quantum AI - QAOA example problems

qaoa > qaoa_notebook Resource Estimator graphs.ipynb > Notebook version of QAOA > Using the Resou

+ Code + Markdown | Run All Clear All Outputs | Outline ···

```
runNames = [
    "Gate-based µs, 10⁻³", "Gate-based µs, 10⁻⁴",
    "Gate-based ns, 10⁻³", "Gate-based ns, 10⁻⁴",
    "S-Majorana ns, 10⁻⁴", "F-Majorana ns, 10⁻⁴",
    "S-Majorana ns, 10⁻⁶", "F-Majorana ns, 10⁻⁶"]

error_budget = .09
result = qsharp.estimate(f"qaoa_note.circuit({input_str})",params = [
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_gate_ns_e3" },
        "qecScheme": { "name": "surface_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_gate_ns_e4" },
        "qecScheme": { "name": "surface_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_gate_us_e3" },
        "qecScheme": { "name": "surface_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_gate_us_e4" },
        "qecScheme": { "name": "surface_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_maj_ns_e4" },
        "qecScheme": { "name": "surface_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_maj_ns_e4" },
        "qecScheme": { "name": "floquet_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "errorBudget": error_budget,
        "qubitParams": { "name": "qubit_maj_ns_e6" },
        "qecScheme": { "name": "surface_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    },
    {
        "error_budget": error_budget,
        "qubitParams": { "name": "qubit_maj_ns_e6" },
        "qecScheme": { "name": "floquet_code" },
        "estimateType": "frontier", # Pareto frontier estimation
    }
])
```

Parameter Customization

1. Select Target Parameters:
   - Choose the desired qubit parameter and error correction code combination. For this example, select "qubit_gate_us_e3" and "surface_code."
   - Click OK to proceed.
2. Specify Error Budget:
   - Optionally, specify the error budget. You can either accept the default value (0.001) or enter a new one.
   - Press Enter to continue.
3. Change Target Parameters:
   - To estimate costs for different configurations, repeat the process by selecting alternative qubit types, error correction codes, and error budgets.
4. Run Multiple Configurations of Parameters:
   - Open the Resource Estimator window again.
   - Select multiple configurations of target parameters and compare the resource estimation results.
   - The Space-time diagram and the table of results will display information for all selected configurations, allowing for comprehensive comparison.

Running Multiple Configurations of Target Parameters

Batching with the Resource Estimator:
Batching allows you to submit jobs with multiple configurations of target parameters as a single job, streamlining the estimation process. Here's how you can utilize batching with the Resource Estimator for QUBO problems:

1. Constructing Batches with Q#:

- In a Jupyter Notebook within VS Code, you can pass a list of target parameters to the `params` parameter of the `qsharp.estimate` function.
- Example:

```python
result_batch = qsharp.estimate("RunProgram()", params=
            [{}, # Default parameters
            {
                "qubitParams": {
                    "name": "qubit_maj_ns_e6"
                },
                "qecScheme": {
                    "name": "floquet_code"
                }
            }])
```

result_batch.summary_data_frame(labels=["Gate-based ns, 10^{-3}", "Majorana ns, 10^{-6}"])
```

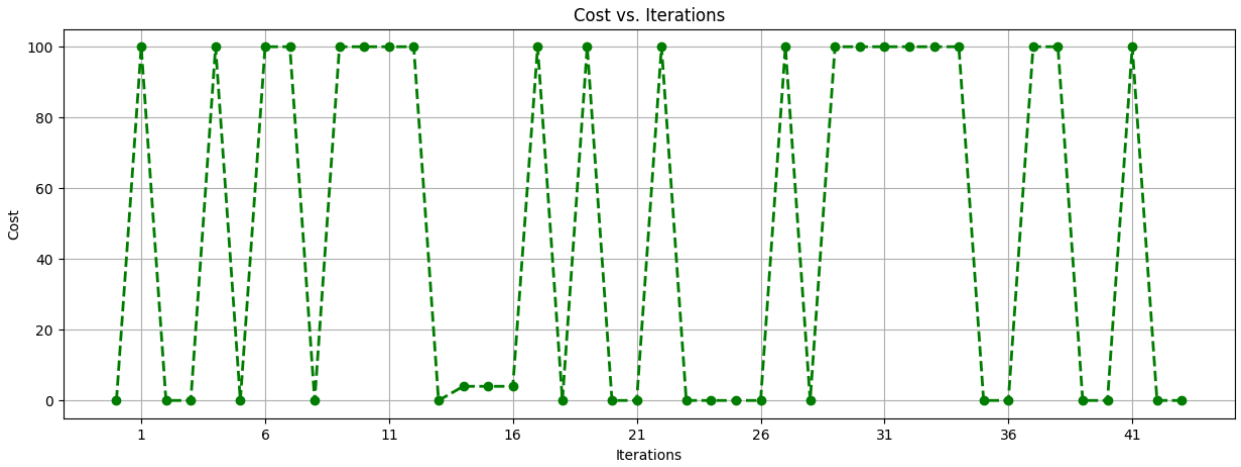
**2. Using EstimatorParams Class:**

- You can also use the `EstimatorParams` class to construct a list of estimation target parameters.
- Example:

```python
from qsharp.estimator import EstimatorParams, QubitParams, QECScheme
labels = ["Gate-based µs, $10^{-3}$", "Gate-based µs, $10^{-4}$", "Gate-based ns, $10^{-3}$", "Gate-based ns, $10^{-4}$", "Majorana ns, $10^{-4}$", "Majorana ns, $10^{-6}$"]
params = EstimatorParams(num_items=6)
params.error_budget = 0.333
params.items[0].qubit_params.name = QubitParams.GATE_US_E3
params.items[1].qubit_params.name = QubitParams.GATE_US_E4
params.items[2].qubit_params.name = QubitParams.GATE_NS_E3
params.items[3].qubit_params.name = QubitParams.GATE_NS_E4
params.items[4].qubit_params.name = QubitParams.MAJ_NS_E4
params.items[4].qec_scheme.name = QECScheme.FLOQUET_CODE
params.items[5].qubit_params.name = QubitParams.MAJ_NS_E6
params.items[5].qec_scheme.name = QECScheme.FLOQUET_CODE
qsharp.estimate("RunProgram()", params=params).summary_data_frame(labels=labels)
```

By utilizing batching, you can efficiently compare multiple configurations of target parameters for your QUBO problems, enabling better optimization and resource allocation strategies.
```

3. Optimizing the Execution of Large QUBO Programs with the Resource Estimator



Handling Large QUBO Programs:
When submitting a resource estimation job to the Resource Estimator for a large QUBO program, the quantum program is evaluated entirely to extract resource estimates. If you're estimating the resources of a QUBO operation invoked multiple times, such as in a loop with many iterations, the execution time of the resource estimation job can be lengthy. To mitigate this, you can employ manual caching.

Manual Caching Technique:
Manual caching involves running the operation once, computing and caching its costs, and subsequently using the cached data for future calls. This approach can significantly reduce execution times. The Resource Estimator target supports two Q# functions for manual caching:

- `BeginEstimateCaching(name: String, variant: Int): Bool`
- `EndEstimateCaching(): Unit`

Example Usage:
Consider the following example of a Q# operation called `ExpensiveOperation`, which is called multiple times in an iteration. Manual caching is utilized to reduce its estimation time:

```qsharp
operation ExpensiveOperation(c: Int, b : Bool): Unit {
    if BeginEstimateCaching("MyNamespace.ExpensiveOperation",
SingleVariant()) {
        // Code block to be cached
        EndEstimateCaching();
    }
```

```
}
```

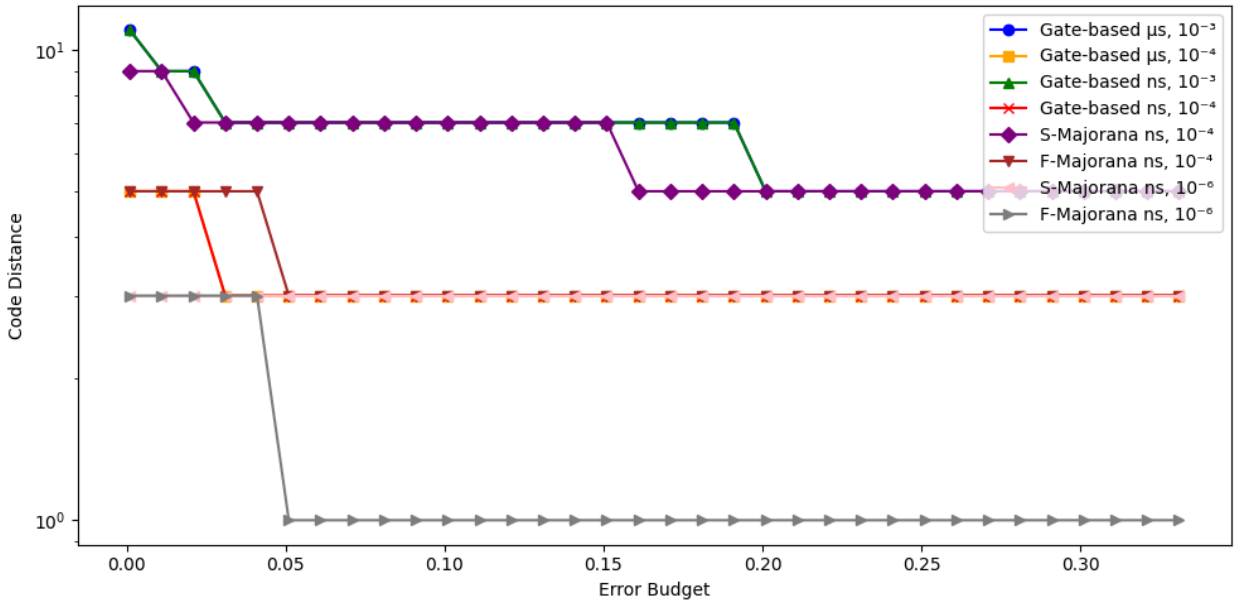
In this example, `BeginEstimateCaching` is called each time `ExpensiveOperation` is invoked. On the first call, it returns true and begins accumulating cost data. The subsequent execution of the expensive code fragment is then performed. When `EndEstimateCaching` is called, the cost data is stored for future use and is incorporated into the overall cost of the program.
Variant Consideration:
You can provide a variant value to distinguish different variants of cost for the same fragment. For instance, if your code has different costs for odd and even values of a variable 'c', you can specify a variant:

```qsharp
operation ExpensiveOperation(c: Int, b : Bool): Unit {
    if BeginEstimateCaching("MyNamespace.ExpensiveOperation", c % 2) {
        // Code block
        EndEstimateCaching();
    }
}
```

By employing manual caching techniques, you can optimize resource allocation and improve the performance of large QUBO programs executed with the Resource Estimator.

View the Results

Explore the results of the resource estimation:



- The Results tab provides a summary of the resource estimation. You can select the columns you want to display, including runtime, physical qubits, logical qubits, and more.
- The Space-time diagram illustrates the tradeoffs between the number of physical qubits and the runtime of the algorithm. Hover over each point to view detailed resource estimation.
- The Space diagram tab displays the distribution of physical qubits used for the algorithm and T factories.
- The Resource Estimates tab presents the full list of output data, including details on QEC scheme, code distance, physical qubits, logical cycle time, and error rates.
  - Customize the displayed columns by clicking the icon next to the first row and selecting from various options such as run name, estimate type, qubit type, qec scheme, error budget, logical qubits, and more. Explore various configurations to understand resource requirements for different QUBO problems effectively.

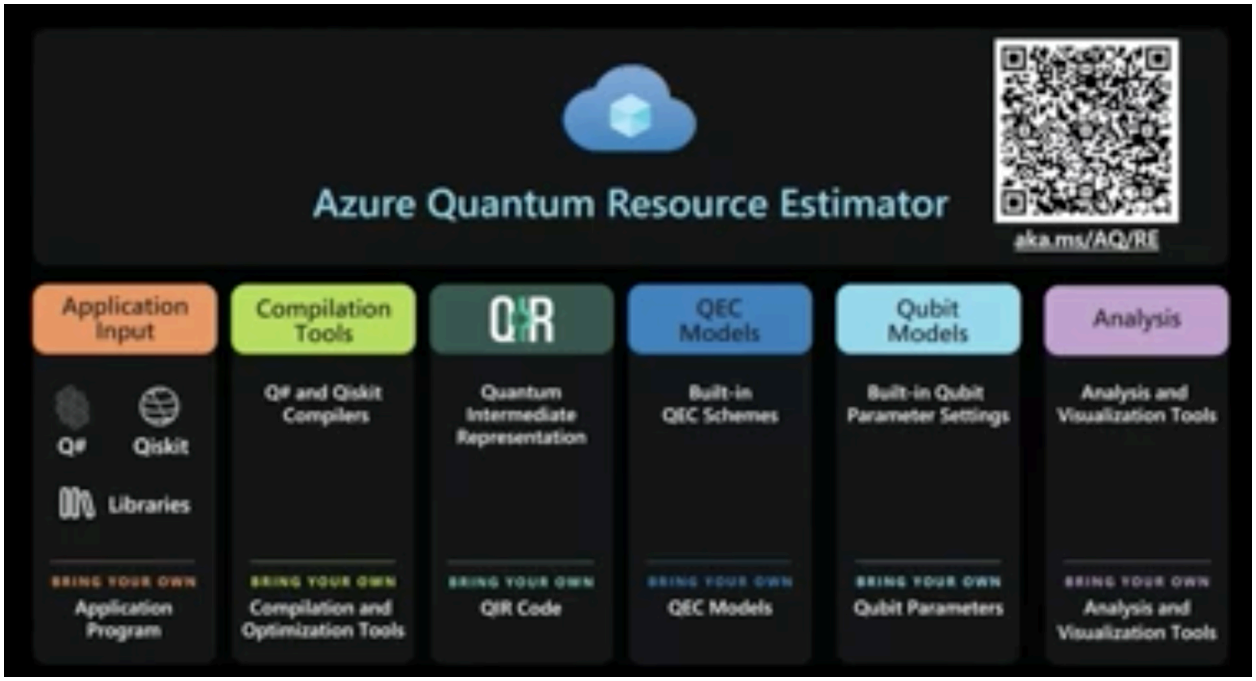
Next Steps

Continue exploring the capabilities of the Resource Estimator:
- Experiment with different SDKs and IDEs.
- Learn how to leverage the Resource Estimator for various QUBO problems and scenarios

Quantum Resource Estimation Results

What is Quantum Resource Estimation?

Quantum computing holds immense potential for revolutionizing various fields, from cryptography to optimization. However, harnessing this potential requires understanding the resources necessary to execute quantum algorithms efficiently. The Quantum Resource Estimator is a powerful open-source tool designed to predict the resources required for executing quantum algorithms, specifically tailored for Quadratic Unconstrained Binary Optimization (QUBO) problems. It provides insights into the number of physical and logical qubits, runtime, and other essential details, aiding in algorithm development and optimization.



Importance of Resource Estimation in Quantum Computing Development

In the quest for practical quantum computing, large-scale fault-tolerant quantum computers are indispensable. Achieving commercial viability necessitates not only a high qubit count but also low physical error rates. Quantum Error Correction (QEC) schemes are crucial for achieving fault tolerance but require additional resources, both in terms of time and space.

Resource estimation plays a pivotal role in understanding architectural design choices and the implications of QEC schemes. By quantifying the resource requirements for running quantum applications, developers can prepare for future scaled quantum machines effectively.

Unique Features of Quantum Resource Estimator

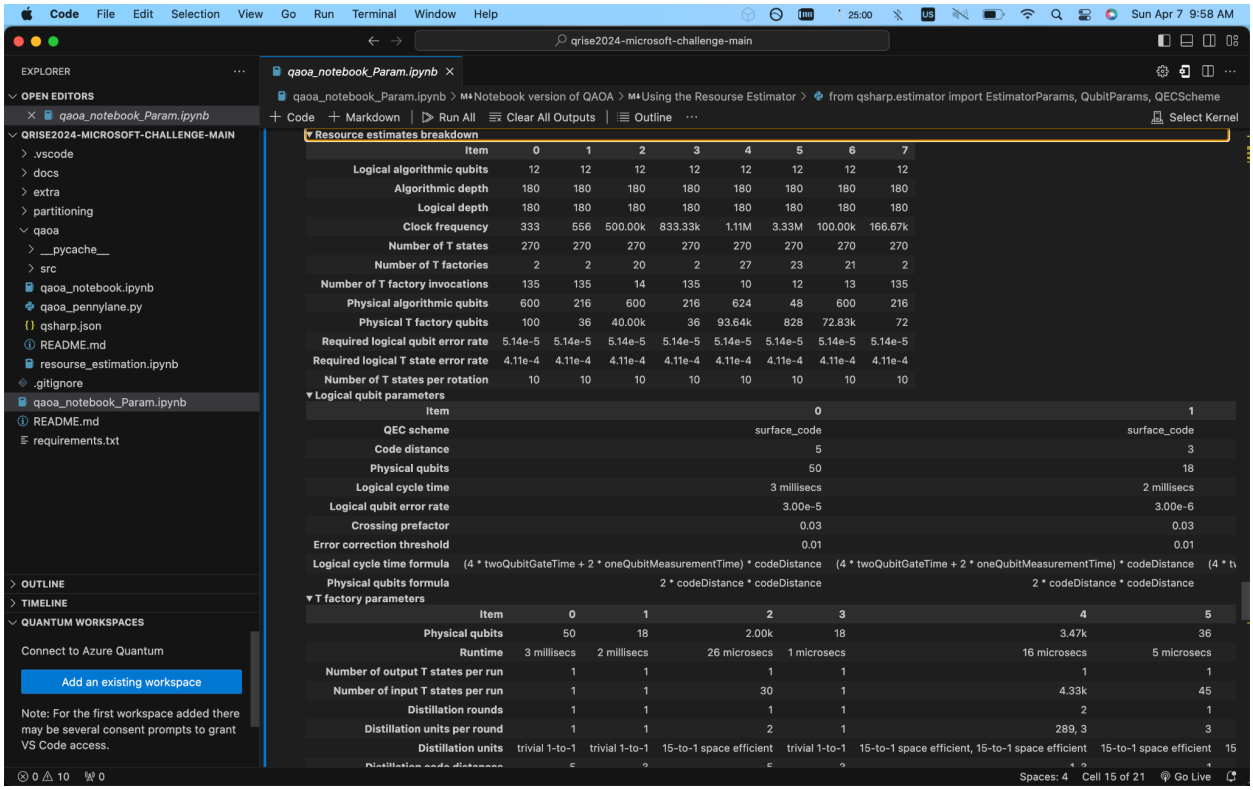
The Quantum Resource Estimator stands out for its versatility and customization options across all levels of the quantum computing stack. Key features include:

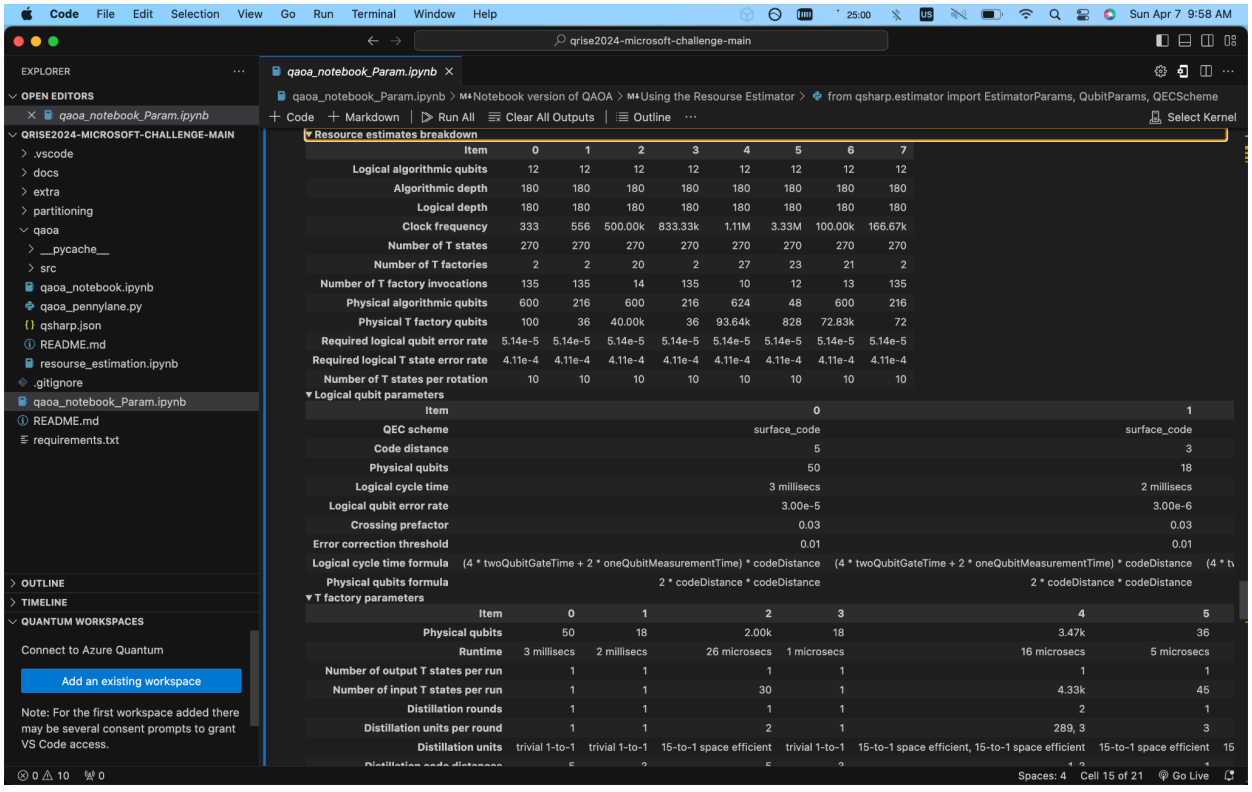
- Customization: Tailor parameters such as physical qubit models, QEC schemes, error budgets, and distillation units to match specific quantum systems and requirements.
- Flexibility: Integrate custom code and compilation tools seamlessly, supporting languages like Q# and Qiskit that translate to Quantum Intermediate Representation (QIR).
- Batch Estimation: Estimate resource requirements for various configurations and compare results to understand the impact of different parameters.
- Optimization: Enhance execution time by optimizing cost calculations, caching subroutines, or passing known estimates to the Estimator.
- Visualization: Visualize tradeoffs between physical qubits and runtime using space-time diagrams, aiding in finding optimal resource combinations.

	Logical qubits	Logical depth	T states	Code distance	T factories	T factory fraction	Physical qubits	rQOPS	Physical runtime	ErrorBudget	Circuit Type
0	12	240	378	11	21	98.59 %	206.18k	2.73M	1 millisecs	0.001	Gate-based µs, 10^{-3}
1	12	240	378	5	21	97.22 %	21.60k	6.00M	480 microsecs	0.001	Gate-based µs, 10^{-4}
2	12	240	378	11	21	97.22 %	104.54k	1.82k	2 secs	0.001	Gate-based ns, 10^{-3}
3	12	240	378	5	21	97.22 %	21.60k	4.00k	720 millisecs	0.001	Gate-based ns, 10^{-4}
4	12	240	378	9	30	99.69 %	631.94k	666.67k	4 millisecs	0.001	S-Majorana ns, 10^{-4}
...	...	...	...	...	...	...	...	...	...	...	...
267	12	180	270	3	2	14.29 %	252	6.67k	324 millisecs	0.331	Gate-based ns, 10^{-4}
268	12	180	270	5	21	99.18 %	73.43k	1.20M	2 millisecs	0.331	S-Majorana ns, 10^{-4}
269	12	180	270	3	27	99.34 %	94.26k	13.33M	162 microsecs	0.331	F-Majorana ns, 10^{-4}
270	12	180	270	3	2	25.00 %	288	2.00M	1 millisecs	0.331	S-Majorana ns, 10^{-6}
271	12	180	270	1	23	94.52 %	876	40.00M	54 microsecs	0.331	F-Majorana ns, 10^{-6}

Resource Estimation for Large-Scale Quantum Computing

For developers venturing into large-scale quantum computing, the Resource Estimator offers invaluable insights. In conclusion, Quantum Resource Estimation is a crucial step towards unlocking the full potential of quantum computing. By accurately predicting resource requirements and optimizing algorithms accordingly, developers can pave the way for groundbreaking advancements across various domains.


Code File Edit Selection View Go Run Terminal Window Help
Sun Apr 7 9:58 AM
qrise2024-microsoft-challenge-main
EXPLORER
OPEN EDITORS
qaoa_notebook_Param.ipynb
QRISE2024-MICROSOFT-CHALLENGE-MAIN
.vscode
docs
extra
partitioning
qaoa
__pycache__
src
qaoa_notebook.ipynb
qaoa_pennylane.py
qsharp.json
README.md
resourse_estimation.ipynb
.gitignore
qaoa_notebook_Param.ipynb
README.md
requirements.txt
OUTLINE
TIMELINE
QUANTUM WORKSPACES
Connect to Azure Quantum
Add an existing workspace
Note: For the first workspace added there may be several consent prompts to grant VS Code access.
qaoa_notebook_Param.ipynb
qaoa_notebook_Param.ipynb > Notebook version of QAOA > Using the Resourse Estimator > from qsharp.estimator import EstimatorParams, QubitParams, QECScheme
Code Markdown Run All Clear All Outputs Outline
Select Kernel
Resource estimates breakdown
Item 0 1 2 3 4 5 6 7
Logical algorithmic qubits 12 12 12 12 12 12 12 12
Algorithmic depth 180 180 180 180 180 180 180 180
Logical depth 180 180 180 180 180 180 180 180
Clock frequency 333 556 500.00k 833.33k 1.11M 3.33M 100.00k 166.67k
Number of T states 270 270 270 270 270 270 270 270
Number of T factories 2 2 20 2 27 23 21 2
Number of T factory invocations 135 135 14 135 10 12 13 135
Physical algorithmic qubits 600 216 600 216 624 48 600 216
Physical T factory qubits 100 36 40.00k 36 93.64k 828 72.83k 72
Required logical qubit error rate 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5
Required logical T state error rate 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4
Number of T states per rotation 10 10 10 10 10 10 10 10
Logical qubit parameters
Item 0 1
QEC scheme surface_code surface_code
Code distance 5 3
Physical qubits 50 18
Logical cycle time 3 millisecs 2 millisecs
Logical qubit error rate 3.00e-5 3.00e-6
Crossing prefactor 0.03 0.03
Error correction threshold 0.01 0.01
Logical cycle time formula (4 * twoQubitGateTime + 2 * oneQubitMeasurementTime) * codeDistance (4 * twoQubitGateTime + 2 * oneQubitMeasurementTime) * codeDistance
Physical qubits formula 2 * codeDistance * codeDistance 2 * codeDistance * codeDistance
T factory parameters
Item 0 1 2 3 4 5
Physical qubits 50 18 2.00k 18 3.47k 36
Runtime 3 millisecs 2 millisecs 26 microsecs 1 microsecs 16 microsecs 5 microsecs
Number of output T states per run 1 1 1 1 1 1
Number of input T states per run 1 1 30 1 4.33k 45
Distillation rounds 1 1 1 1 2 1
Distillation units per round 1 1 2 1 289, 3 3
Distillation units trivial 1-to-1 trivial 1-to-1 15-to-1 space efficient trivial 1-to-1 15-to-1 space efficient, 15-to-1 space efficient 15-to-1 space efficient
Spaces: 4 Cell 15 of 21 Go Live


Code File Edit Selection View Go Run Terminal Window Help
Sun Apr 7 9:58 AM
qrise2024-microsoft-challenge-main
EXPLORER
OPEN EDITORS
qaoa_notebook_Param.ipynb
QRISE2024-MICROSOFT-CHALLENGE-MAIN
.vscode
docs
extra
partitioning
qaoa
__pycache__
src
qaoa_notebook.ipynb
qaoa_pennylane.py
qsharp.json
README.md
resourse_estimation.ipynb
.gitignore
qaoa_notebook_Param.ipynb
README.md
requirements.txt
OUTLINE
TIMELINE
QUANTUM WORKSPACES
Connect to Azure Quantum
Add an existing workspace
Note: For the first workspace added there may be several consent prompts to grant VS Code access.
qaoa_notebook_Param.ipynb
qaoa_notebook_Param.ipynb > Notebook version of QAOA > Using the Resourse Estimator > from qsharp.estimator import EstimatorParams, QubitParams, QECScheme
Code Markdown Run All Clear All Outputs Outline
Select Kernel
Resource estimates breakdown
Item 0 1 2 3 4 5 6 7
Logical algorithmic qubits 12 12 12 12 12 12 12 12
Algorithmic depth 180 180 180 180 180 180 180 180
Logical depth 180 180 180 180 180 180 180 180
Clock frequency 333 556 500.00k 833.33k 1.11M 3.33M 100.00k 166.67k
Number of T states 270 270 270 270 270 270 270 270
Number of T factories 2 2 20 2 27 23 21 2
Number of T factory invocations 135 135 14 135 10 12 13 135
Physical algorithmic qubits 600 216 600 216 624 48 600 216
Physical T factory qubits 100 36 40.00k 36 93.64k 828 72.83k 72
Required logical qubit error rate 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5 5.14e-5
Required logical T state error rate 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4 4.11e-4
Number of T states per rotation 10 10 10 10 10 10 10 10
Logical qubit parameters
Item 0 1
QEC scheme surface_code surface_code
Code distance 5 3
Physical qubits 50 18
Logical cycle time 3 millisecs 2 millisecs
Logical qubit error rate 3.00e-5 3.00e-6
Crossing prefactor 0.03 0.03
Error correction threshold 0.01 0.01
Logical cycle time formula (4 * twoQubitGateTime + 2 * oneQubitMeasurementTime) * codeDistance (4 * twoQubitGateTime + 2 * oneQubitMeasurementTime) * codeDistance
Physical qubits formula 2 * codeDistance * codeDistance 2 * codeDistance * codeDistance
T factory parameters
Item 0 1 2 3 4 5
Physical qubits 50 18 2.00k 18 3.47k 36
Runtime 3 millisecs 2 millisecs 26 microsecs 1 microsecs 16 microsecs 5 microsecs
Number of output T states per run 1 1 1 1 1 1
Number of input T states per run 1 1 30 1 4.33k 45
Distillation rounds 1 1 1 1 2 1
Distillation units per round 1 1 2 1 289, 3 3
Distillation units trivial 1-to-1 trivial 1-to-1 15-to-1 space efficient trivial 1-to-1 15-to-1 space efficient, 15-to-1 space efficient 15-to-1 space efficient
Spaces: 4 Cell 15 of 21 Go Live

The resource estimation results provide valuable insights into the assumed error budget and various physical qubit parameters associated with the execution of the quantum algorithm.

Runtime:

The estimated runtime varied across different instances, ranging from microseconds to milliseconds. This indicates the expected time required for the execution of the QUBO code on quantum hardware. The physical runtime, representing the estimated time required for execution, varies across scenarios, ranging from microseconds to milliseconds. Lower runtime values indicate faster execution times.

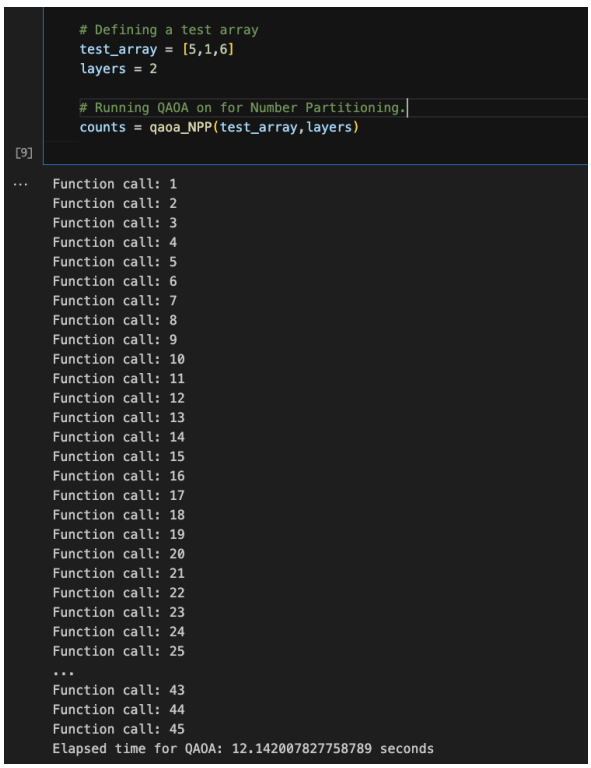
Input Array: [5,1,6,2,4]

```
[9]
# Defining a test array
test_array = [5,1,6,2,4]
layers = 3

# Running QAOA on for Number Partitioning.
counts = qaoa_NPP(test_array,layers)
```

```
Function call: 1
Function call: 2
Function call: 3
Function call: 4
Function call: 5
Function call: 6
Function call: 7
Function call: 8
Function call: 9
Function call: 10
Function call: 11
Function call: 12
Function call: 13
Function call: 14
Function call: 15
Function call: 16
Function call: 17
Function call: 18
Function call: 19
Function call: 20
Function call: 21
Function call: 22
Function call: 23
Function call: 24
Function call: 25
...
Function call: 51
Function call: 52
Function call: 53
Elapsed time for QAOA: 193.5793056488037 seconds
```

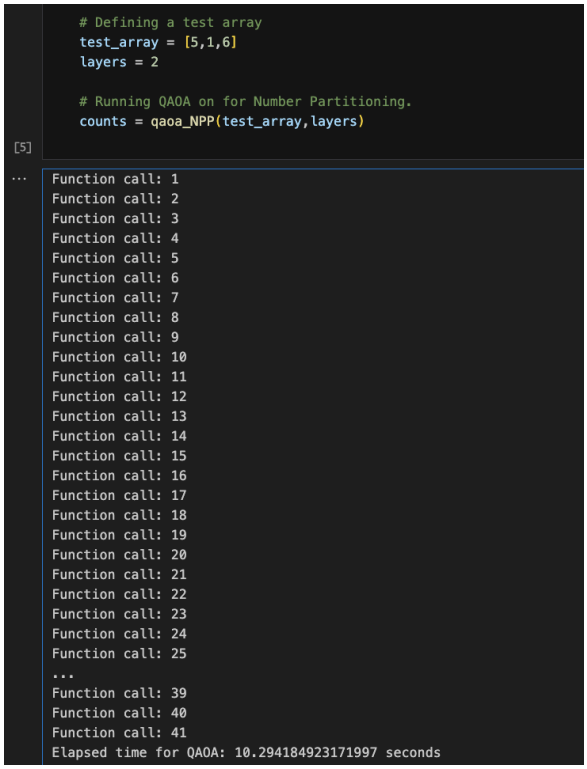
Input Array: [5,1,6]



```
# Defining a test array
test_array = [5,1,6]
layers = 2

# Running QAOA on for Number Partitioning.
counts = qaoa_NPP(test_array,layers)
```

```
[9]
...
Function call: 1
Function call: 2
Function call: 3
Function call: 4
Function call: 5
Function call: 6
Function call: 7
Function call: 8
Function call: 9
Function call: 10
Function call: 11
Function call: 12
Function call: 13
Function call: 14
Function call: 15
Function call: 16
Function call: 17
Function call: 18
Function call: 19
Function call: 20
Function call: 21
Function call: 22
Function call: 23
Function call: 24
Function call: 25
...
Function call: 43
Function call: 44
Function call: 45
Elapsed time for QAOA: 12.142007827758789 seconds
```



```
# Defining a test array
test_array = [5,1,6]
layers = 2

# Running QAOA on for Number Partitioning.
counts = qaoa_NPP(test_array,layers)
```

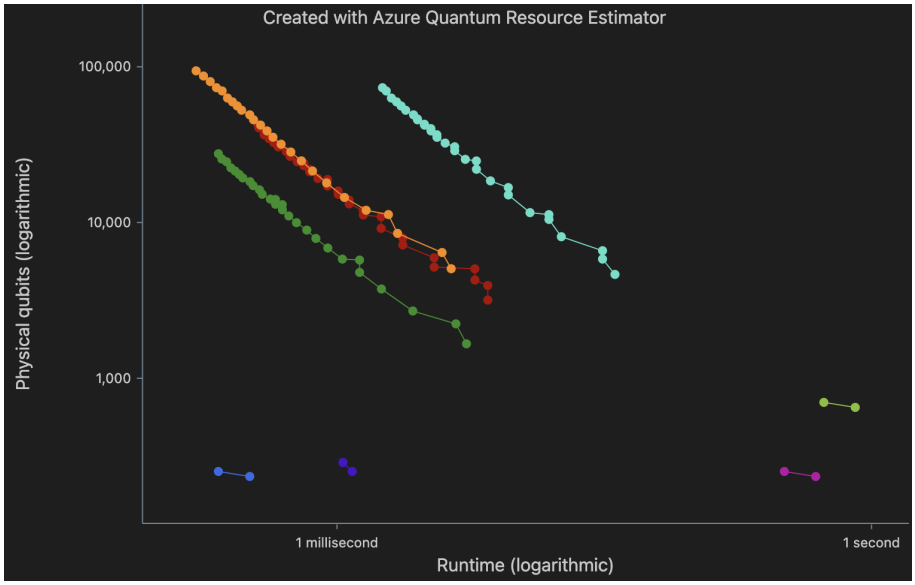
```
[5]
...
Function call: 1
Function call: 2
Function call: 3
Function call: 4
Function call: 5
Function call: 6
Function call: 7
Function call: 8
Function call: 9
Function call: 10
Function call: 11
Function call: 12
Function call: 13
Function call: 14
Function call: 15
Function call: 16
Function call: 17
Function call: 18
Function call: 19
Function call: 20
Function call: 21
Function call: 22
Function call: 23
Function call: 24
Function call: 25
...
Function call: 39
Function call: 40
Function call: 41
Elapsed time for QAOA: 10.294184923171997 seconds
```

rQOPS (Relative Quantum Operations per Second):

The rQOPS metric represented the number of quantum operations per second and varied from thousands to millions across different instances. Higher rQOPS values indicate greater computational efficiency.

Physical Qubits:

The number of physical qubits required varied significantly across different instances, ranging from tens to thousands. This indicates the qubit resources needed to execute the QUBO code effectively.



Physical Qubit Parameters:

The number of physical qubits required varies significantly across scenarios, ranging from 252 to 94.26k. Similarly, the rQOPS metric, representing the relative quantum operations per second, varies from 4.00k to 40.00M, indicating the computational efficiency of the quantum algorithm.

- Qubit Name and Instruction Set: The physical qubits are categorized based on their gate types and instruction sets, including gate-based and Majorana qubits.

- Single-qubit Measurement and Gate Time: The time required for single-qubit measurements and gate operations varies across different instances, ranging from nanoseconds to microseconds. Lower times indicate faster operations.

- Two-qubit Gate Time: Two-qubit gate times are consistent across instances and are significantly higher compared to single-qubit gate times.

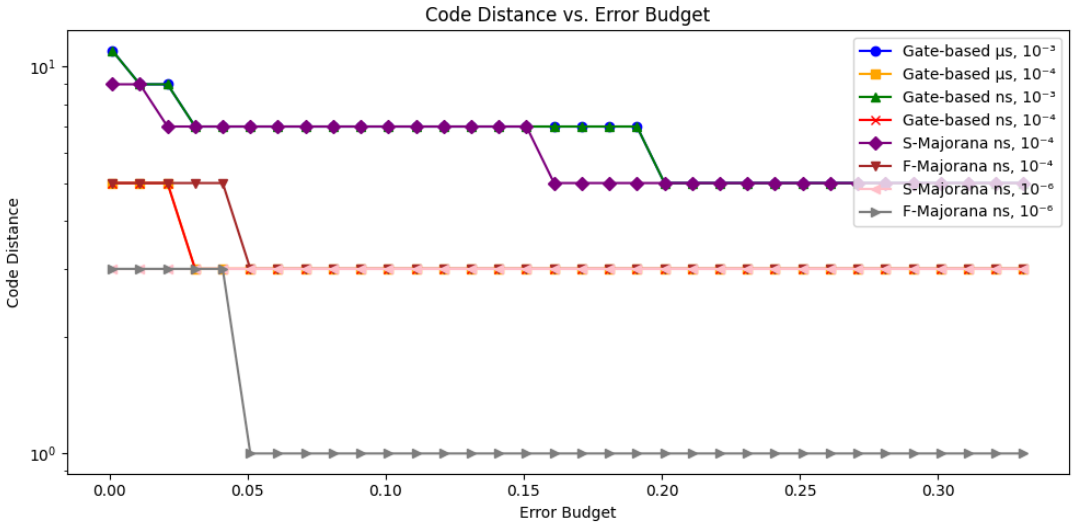
- Error Rates: Error rates for single-qubit and two-qubit operations are provided for each instance. Lower error rates indicate higher reliability of operations.

Resource Estimates Breakdown:

The breakdown of resource estimates provided detailed insights into various parameters such as logical algorithmic qubits, algorithmic depth, clock frequency, number of T states, and more. These parameters influence the overall performance and efficiency of the quantum computation.

Logical Qubit Parameters:

Parameters such as QEC scheme, code distance, physical qubits, logical cycle time, logical qubit error rate, and others were analyzed. These parameters play a crucial role in error correction and fault tolerance in quantum computation.



T Factory Parameters:

T factory parameters, including physical qubits, runtime, number of output T states per run, distillation rounds, and logical T state error rate, were examined. These parameters are essential for implementing quantum error correction techniques such as distillation.

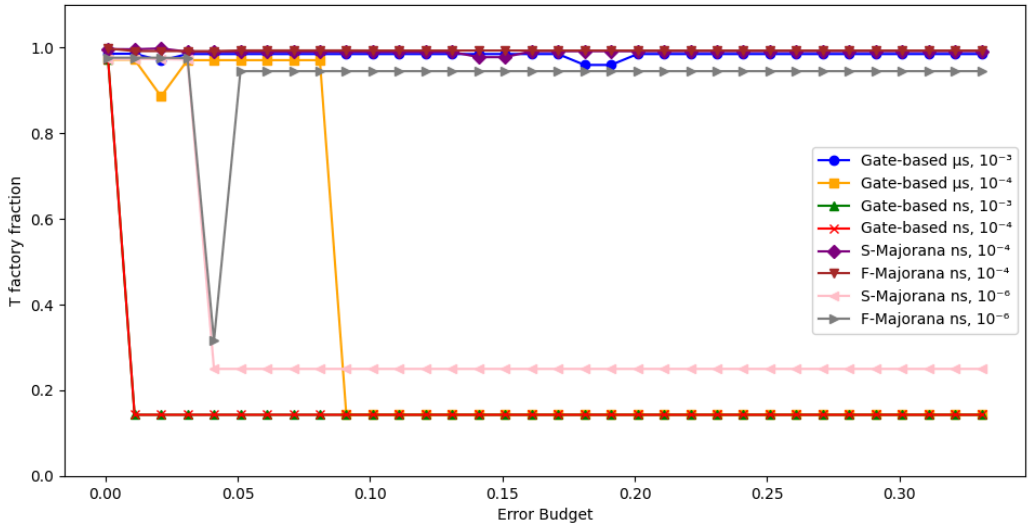
Run name	T factory fraction	Physical qubits	Runtime	rQOPS
Gate-based µs, 10^{-3}	98.59 %	42,600	360 microsecs	6,000,000
Gate-based µs, 10^{-4}	14.29 %	252	216 microsecs	10,000,000
Gate-based ns, 10^{-3}	14.29 %	700	540 millisecs	4,000
Gate-based ns, 10^{-4}	14.29 %	252	324 millisecs	6,667
S-Majorana ns, 10^{-4}	99.18 %	73,428	2 millisecs	1,200,000
F-Majorana ns, 10^{-4}	99.34 %	94,260	162 microsecs	13,333,334
S-Majorana ns, 10^{-6}	25.00 %	288	1 millisecs	2,000,000
F-Majorana ns, 10^{-6}	97.74 %	27,664	216 microsecs	13,333,334

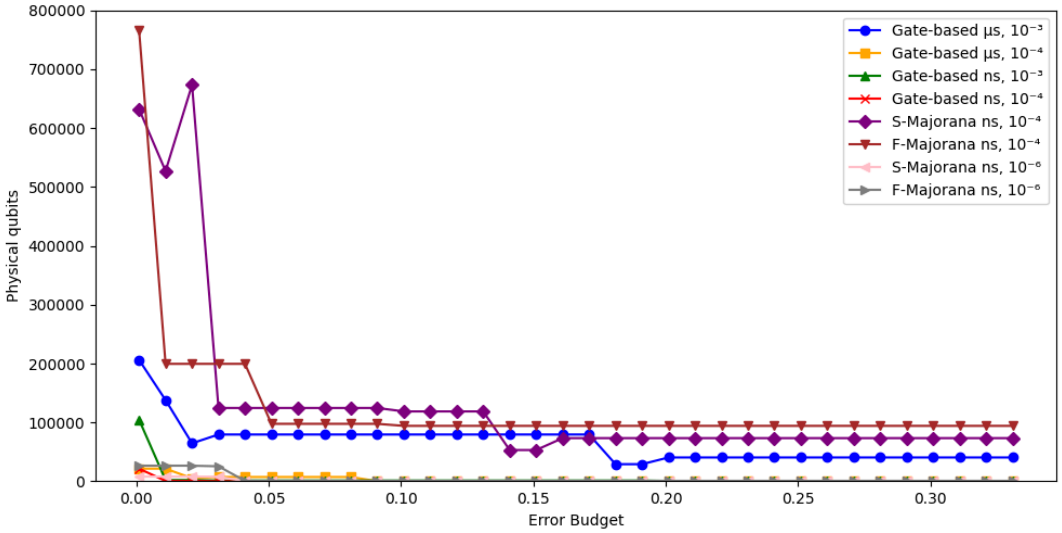
Pre-layout Logical Resources:

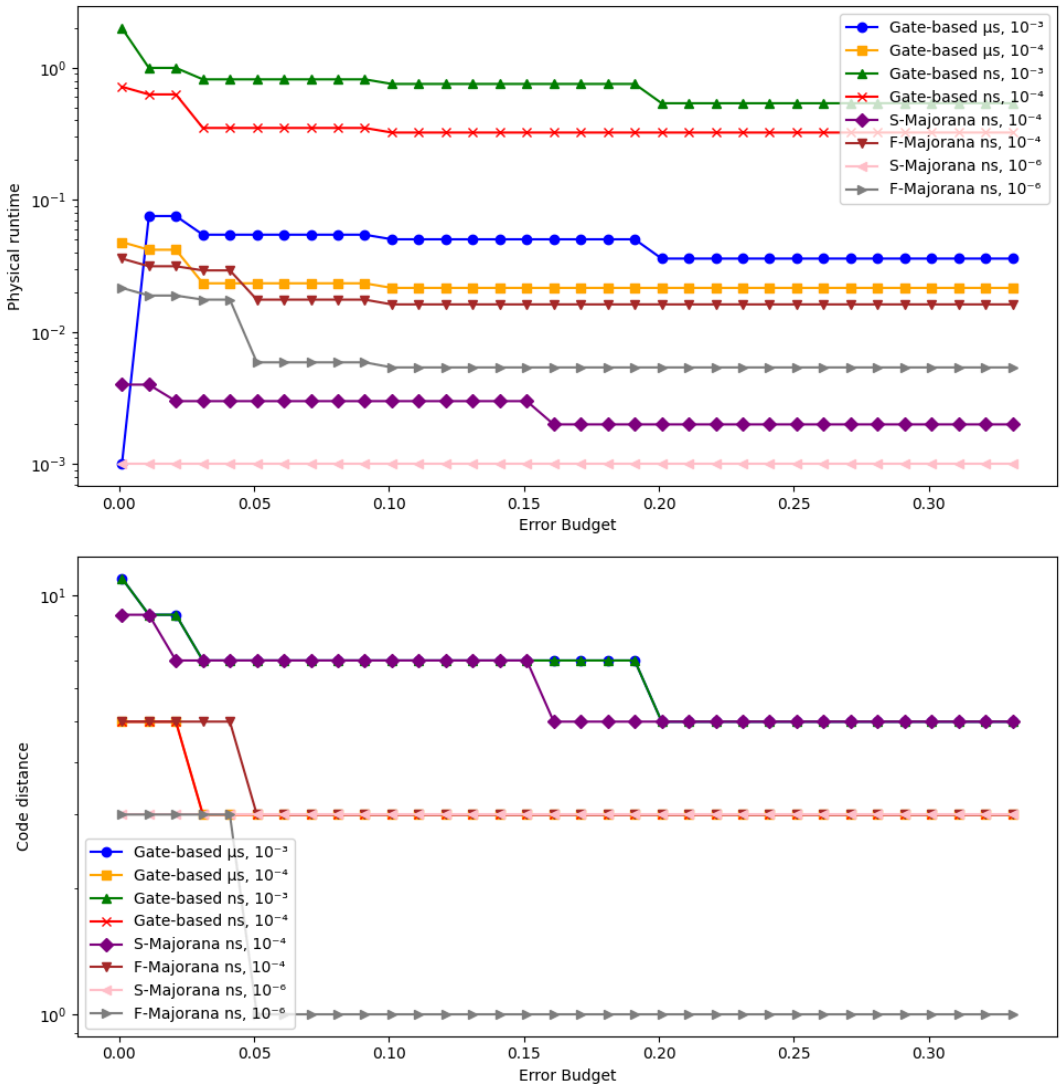
Pre-layout logical resources such as logical qubits, T gates, rotation gates, CCZ gates, CCiX gates, and measurement operations were evaluated. These resources represent the computational elements required for executing the QUBO code.

Assumed Error Budget:

The total error budget across all instances is assumed to be \(3.33 \times 10^{-1}\). This budget is distributed among different error sources, including logical error probability, T distillation error probability, and rotation synthesis error probability. Each error source is allocated a portion of the total error budget, ensuring that the overall error remains within acceptable limits.


T factory fraction
Error Budget
Gate-based µs, 10⁻³
Gate-based µs, 10⁻⁴
Gate-based ns, 10⁻³
Gate-based ns, 10⁻⁴
S-Majorana ns, 10⁻⁴
F-Majorana ns, 10⁻⁴
S-Majorana ns, 10⁻⁶
F-Majorana ns, 10⁻⁶


Physical qubits
Error Budget
Gate-based µs, 10⁻³
Gate-based µs, 10⁻⁴
Gate-based ns, 10⁻³
Gate-based ns, 10⁻⁴
S-Majorana ns, 10⁻⁴
F-Majorana ns, 10⁻⁴
S-Majorana ns, 10⁻⁶
F-Majorana ns, 10⁻⁶



Constraints:

The constraints section outlines various constraints imposed on the quantum algorithm, including logical depth factor, maximum number of T factories, maximum runtime duration, and maximum number of physical qubits. These constraints ensure that the algorithm remains within specified bounds during execution.

Assumptions:

The assumptions section presents the estimated resource requirements for different scenarios of logical qubits, logical depth, T states, code distance, T factories, T factory fraction, physical qubits, rQOPS (Relative Quantum Operations per Second), and physical runtime. These assumptions are based on different combinations of gate-based and Majorana qubits, along with varying error rates and gate times.

Logical Qubits and Depth:

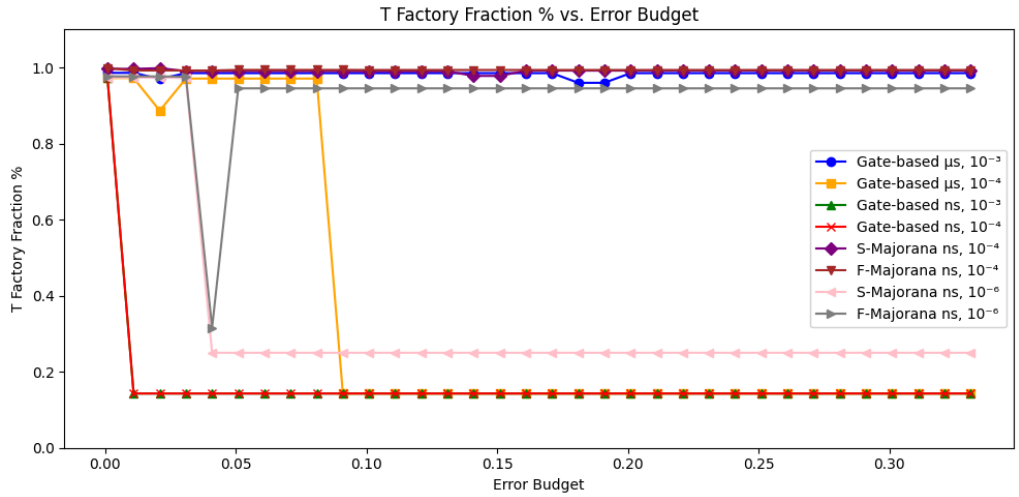
The number of logical qubits remains constant at 12 across all scenarios, indicating the complexity of the quantum algorithm. Similarly, the logical depth, representing the number of computational steps, remains consistent at 180.

T States and Code Distance:

The number of T states and the code distance also remain constant at 270, suggesting a consistent computational structure for the quantum algorithm across different configurations.

T Factories and T Factory Fraction:

The number of T factories varies across scenarios, with values ranging from 1 to 27. Similarly, the T factory fraction, representing the percentage of T factories utilized, varies between approximately 14.29% and 99.34%, indicating the degree of utilization of available resources.



Quantum Resource Estimation Discussion

The resource estimation results provide comprehensive information about the physical resources required for executing the quantum algorithm under different scenarios. The resource estimation results provide insights into the resource requirements and performance characteristics of the quantum algorithm under different configurations.

```
%%qsharp
// This is the Q# snippet used to create and simulate the QAOA circuit.
namespace qaoa_note{

    open Microsoft.Quantum.Measurement;

    // Function for getting flat index
    operation flat_index(n: Int, i: Int, j: Int): Int{
        return n*i + j
    }
    // Cost Hamiltonian
    operation cost_unitary(qubits: Qubit[], gamma: Double, quadratics: Double[], linears: Double[]): Unit{

        let n_qubits = Length(linears);
        mutable quad_sum : Double = 0.0;

        // RZ Gates
        for qubit in 0..n_qubits-1{
            set quad_sum = 0.0;
            for quad_qubit in 0..n_qubits-1{
                set quad_sum += quadratics[flat_index(n_qubits,qubit,quad_qubit)];
            }
            Rz(0.5 * (linears[qubit] + quad_sum) * gamma, qubits[qubit])
        }
        // RZZ Gates
        for i in 0..n_qubits-1{
            for j in i+1..n_qubits-1{
                Rzz(0.25 * quadratics[flat_index(n_qubits,i,j)] * gamma, qubits[i], qubits[j])
            }
        }
    }

    // Mixer Hamiltonian
    operation mixer_unitary(qubits: Qubit[], beta: Double) : Unit{
        for qubit in qubits{
            Rx(2.0 * beta,qubit);
        }
    }

    // Function to create the QAOA circuit.
    operation circuit(NQubits: Int, Layers: Int, gammas: Double[], betas: Double[], quadratics: Double[], linears: Double[]) : Int {

        use q = Qubit[NQubits];
        mutable integer_result = 0;

        // State Preparation |+>
        ApplyToEachA(H,q);

        for layer in 0..Layers-1{
            cost_unitary(q, gammas[layer], quadratics, linears);
            mixer_unitary(q, betas[layer]);
        }
        // Return the bitstring as an integer.
        return MeasureInteger(q);
    }
}
```

Gates

In our algorithm implementation, both single-qubit and two-qubit gates were utilized. The number of intrinsic operations (gates) required depends on the number of qubits and layers in the circuit. These gates are essential components of the algorithm, allowing for the implementation of the cost and mixer unitaries required for solving optimization problems like the Number Partitioning Problem (NPP).

1. Single-qubit gates:

- Hadamard Gate (H): Used for preparing the initial state. It creates superpositions of the qubits, putting them into a balanced mixture of the $|0\rangle$ and $|1\rangle$ states.

- Rotation gates (Rx, Rz): These gates perform single-qubit rotations around the X and Z axes of the Bloch sphere, respectively. They are used to apply the rotations specified by the optimization parameters (γ and β) to the qubits.

2. Two-qubit gates:

- Controlled-Z Rotation Gate (Rzz): A two-qubit gate used to introduce entanglement between qubits. It performs a controlled-phase rotation on the target qubit conditioned on the state of the control qubit.

extra > Entanglement sample.qs

```qsharp
/// # Sample
/// Entanglement
///
/// # Description
/// Qubits are said to be entangled when the state of each one of them cannot be
/// described independently from the state of the others.
///
/// This Q# program entangles two qubits.
namespace Sample {
    open Microsoft.Quantum.Diagnostics;

    @EntryPoint()
    Run | Histogram | Estimate | Debug
    operation EntangleQubits() : (Result, Result) {
        // Allocate the two qubits that will be entangled.
        use (q1, q2) = (Qubit(), Qubit());

        // Set the first qubit in superposition by calling the `H` operation,
        // which applies a Hadamard transformation to the qubit.
        // Then, entangle the two qubits using the `CNOT` operation.
        H(q1);
        CNOT(q1, q2);

        // Show the entangled state using the `DumpMachine` function.
        DumpMachine();

        // Measurements of entangled qubits are always correlated.
        let (m1, m2) = (M(q1), M(q2));
        Reset(q1);
        Reset(q2);
        return (m1, m2);
    }
}
```

The gates that require more resources are typically the controlled-Z (CZ) gates or the Rzz gates used to implement the two-qubit interactions in the cost Hamiltonian. These gates involve entangling multiple qubits, which can be more resource-intensive on quantum hardware. On the other hand, gates like the single-qubit rotations (Rz, Rx) and Hadamard (H) gates used in state preparation and mixing typically require fewer resources, as they only act on individual qubits without entangling them. Overall, the resource requirements for different gates depend on factors such as the specific quantum hardware architecture, gate fidelity, and optimization techniques used in the implementation.

extra > Cost Hamiltonian Gates.qs

```qsharp
namespace Quantum {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Convert;

    function SetBitValue(reg: Int, bit: Int, value: Bool): Int {
        if(value) {
            return reg ||| (1 <<< bit);
        } else {
            return reg &&& ~~~(1 <<< bit);
        }
    }

    operation Circuit() : Unit {
        using(qubits = Qubit[4]) {
            Rz(50.725, qubits[0]);
            Rz(181.16, qubits[1]);
            Rz(-26.57, qubits[2]);
            Rz(-205.31, qubits[3]);
            zz(12.077, qubits[0], qubits[1]);
            zz(26.57, qubits[0], qubits[2]);
            zz(12.077, qubits[0], qubits[3]);
            zz(132.85, qubits[1], qubits[2]);
            ResetAll(qubits);
        }
    }
}
```

Intrinsic Operations

1. Single-qubit operations:

   - Hadamard Gate (H): The Hadamard gate is applied to each qubit in the circuit once for state preparation, resulting in `NQubits` Hadamard gates.

   - Rotation gates (Rx, Rz): Each qubit undergoes rotation gates for each layer of the QAOA circuit. So, for `Layers` layers and `NQubits` qubits, there are `2 * Layers * NQubits` rotation gates.

2. Two-qubit operations:

   - Controlled-Z Rotation Gate (Rzz): For each pair of qubits (excluding self-pairing), a controlled-Z rotation gate is applied. Considering `NQubits`, there are `NQubits * (NQubits - 1) / 2` pairs of qubits, resulting in `NQubits * (NQubits - 1)` controlled-Z rotation gates.

Therefore, the total number of intrinsic operations (gates) to be called is the sum of the counts of each kind:

- Hadamard gates: `NQubits`

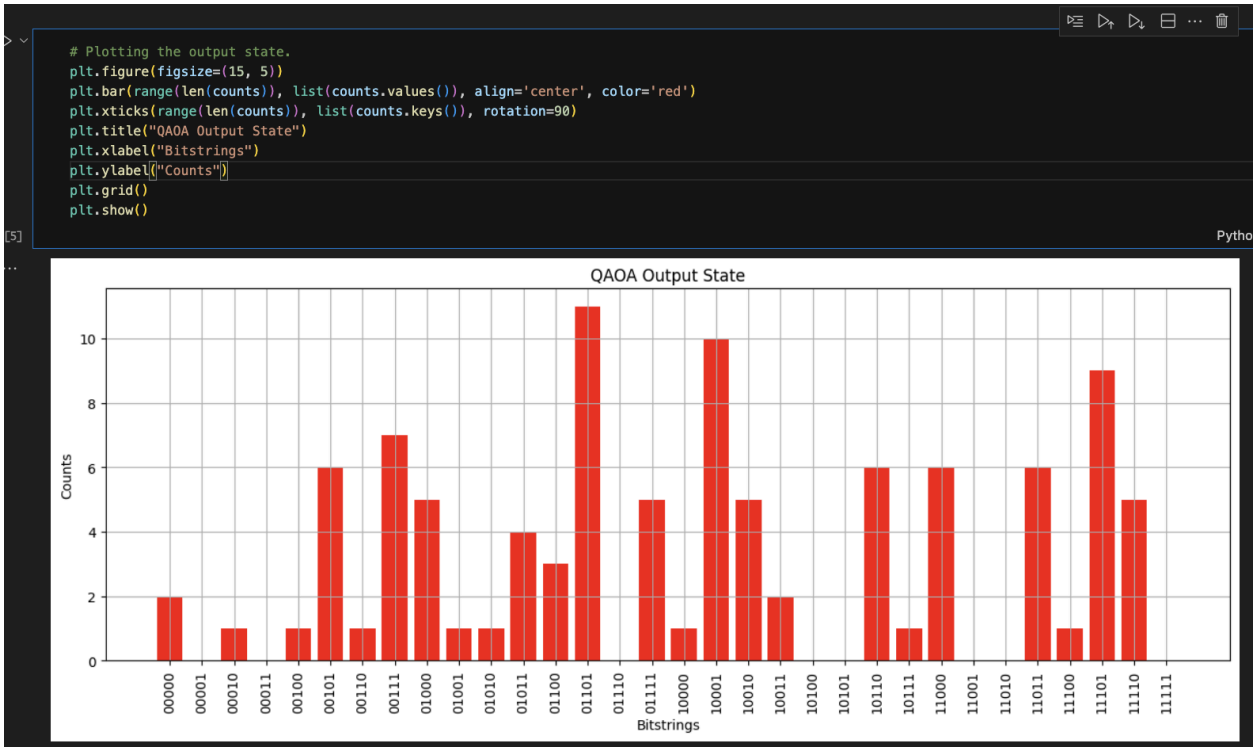
- Rotation gates: `2 * Layers * NQubits`

- Controlled-Z rotation gates: `NQubits * (NQubits - 1)`

These counts represent the number of times each gate type needs to be applied in the QAOA circuit to execute the algorithm for a given problem instance.

Parallel Quantum Operations

In the QAOA algorithm implementation provided in the github, the quantum operations that can be called in parallel are the operations within each layer of the circuit. Within each layer, both the mixer unitary operation (`mixer_unitary`) and the cost unitary operation (`cost_unitary`) can be applied simultaneously to all qubits. This is because these operations act independently on each qubit and do not have any interdependencies or sequential execution requirements. Therefore, within each layer, all quantum operations can be called in parallel across all qubits. However, between layers, the operations must be executed sequentially since the output state of one layer serves as the input state for the next layer. In summary, the quantum operations within each layer of the circuit can be called in parallel, while the operations across layers must be executed sequentially.



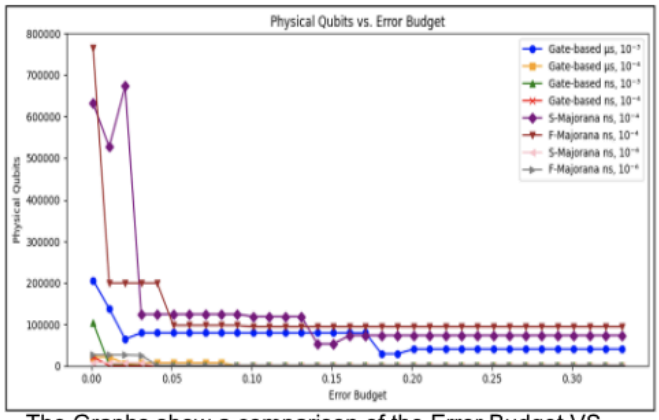
These findings are crucial for understanding the scalability and efficiency of the algorithm on quantum hardware. It considers factors such as error rates, gate times, and constraints to estimate the runtime and resource utilization accurately.

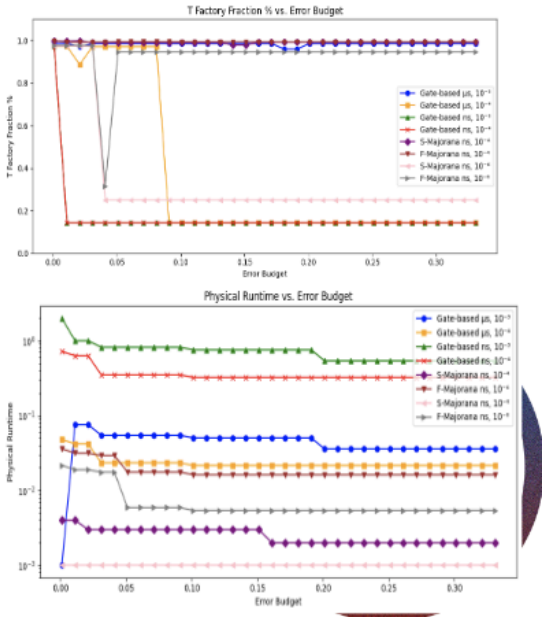
- The findings suggest that the choice of qubit type (gate-based or Majorana) and error rates significantly impact the resource requirements and performance of the quantum algorithm.
- By analyzing the assumed error budget and physical qubit parameters, developers can make informed decisions about optimizing the algorithm design and selecting appropriate hardware configurations to achieve desired performance outcomes.
- The consistent values for logical qubits, logical depth, T states, and code distance across scenarios suggest a uniform computational structure for the quantum algorithm, regardless of the hardware configuration.
- The variation in T factories and T factory fraction indicates the degree of parallelism and resource utilization in different scenarios. Higher values signify more efficient resource allocation and potentially faster execution times.
- The significant variation in the number of physical qubits and rQOPS across scenarios highlights the sensitivity of the algorithm's performance to hardware specifications and error rates. Higher values indicate greater computational power and efficiency.
- The estimated physical runtimes provide valuable insights into the expected execution times for the quantum algorithm under different configurations. Developers can use this information to optimize the algorithm design and hardware selection for improved performance.

Determining the Optimal Trade Off between Performance and Error Budget

After a test comparison was done for the jobs, comparing the Number of Physical Qubits, the Physical Run time, the T Factory Fraction Percentage, as well as additional parameters, all against the range of error rates

The results not only confirmed that the Gate_based ion quantum computer with the “surface_code” quantum error correction scheme is the best set of parameters to run our algorithm on, it also showed us that we can reduce the error budget from 33.3% to just above 9% and still achieve the same results





The Graphs show a comparison of the Error Budget VS
1.The number of Physical Qubits required
2.The T Factory Fraction %
3.The Physical Run Time

The Job in yellow is the job that we found performed the best.

Based on the Resource Estimator results, our recommendation to other attempting to solve a quadratic unconstrained binary optimization problem on Microsoft’s quantum software is that they use the ion based quantum computer with “surface_code” as quantum error correction scheme and an error rate budget of 9%

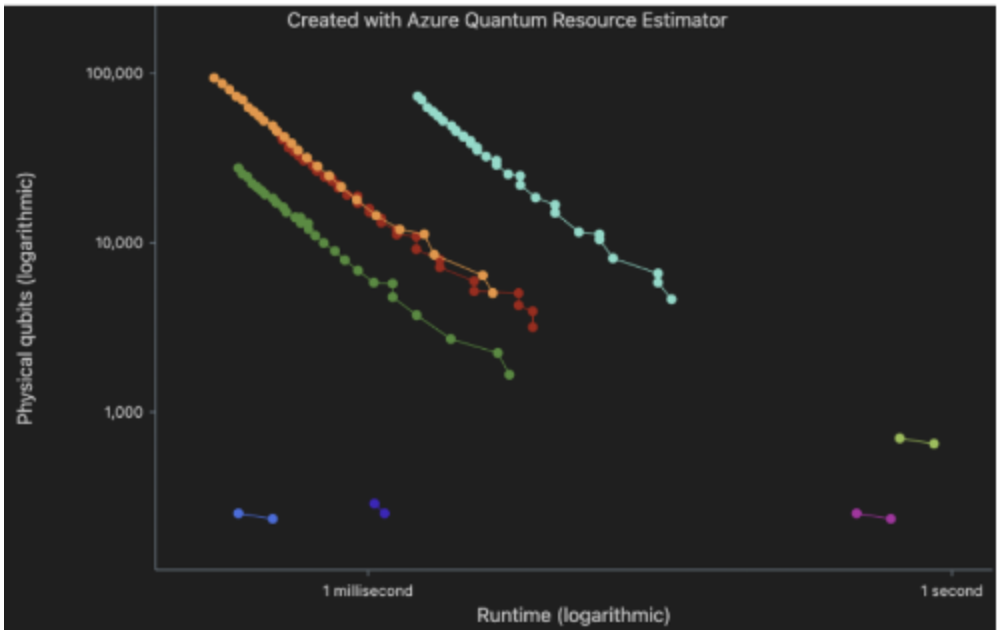
Ion Gate-based Quantum Computer

Once we encoded and tested our QAOA circuit and classical optimizer worked to our satisfaction, we proceeded to apply Microsoft's Resource Estimator Algorithm to our QAOA Circuit (quantum representation of the number partition problem) to determine the number and of resources needed to compute the quantum circuit and to determine the configuration of parameters that would give us the highest chance of success, while requiring the fewest resources.

Our first step was to run and compare the various combinations of qubit types and error correction schemes. at this step we had a total of 8 jobs, 3 different system setups including ion qubits, superconducting qubits, and Majorana qubits.

Run name	T factory fraction	Physical qubits	Runtime	rQOPS
Gate-based µs, 10^{-3}	98.59 %	42,600	360 microsecs	6,000,000
Gate-based µs, 10^{-4}	14.29 %	252	216 microsecs	10,000,000
Gate-based ns, 10^{-3}	14.29 %	700	540 millisecs	4,000
Gate-based ns, 10^{-4}	14.29 %	252	324 millisecs	6,667
S-Majorana ns, 10^{-4}	99.18 %	73,428	2 millisecs	1,200,000
F-Majorana ns, 10^{-4}	99.34 %	94,260	162 microsecs	13,333,334
S-Majorana ns, 10^{-6}	25.00 %	288	1 millisecs	2,000,000
F-Majorana ns, 10^{-6}	97.74 %	27,664	216 microsecs	13,333,334

For each system we ran a 1 job that computed a realistic and 1 job that computed the optimistic error rate target. All Gate_Based jobs were run using the "surface_code" quantum error correction scheme, and the Majorana based qubits explored both a surface and Floquet error correction scheme.



1. The graph shown here was made using Azure Quantum's Estimate Overview function, all the values you see were measured with an error budget of 33.3%
2. The job that stands out as the best performer is the Gate_based ion quantum computer with an optimistic 14.29% T Factory Fraction percentage, 252 physical qubits, a runtime of 216 microsecs, and a rQOPS of 10 million

3. To confirm that this was a good option we moved onto phase two of analysis. Examining the error budget.

Error Budget

The error budget for all jobs shown was 33.3%. We chose to start 33% as our error Budget threshold because of 2 reasons. number
1. To give every a chance to show their best possible outcome, we can always do more error correction later
2. Solution only has about a 35-50% success rate, but that is good enough to get the right answer.)

Compare Theoretical Results with Experimental Results

Theoretical results for QUBO numerical partitioning involve analyzing the performance and complexity of algorithms designed to solve the problem. Theoretical comparisons between quantum and classical algorithms for QUBO numerical partitioning provide insights into the potential quantum advantage, highlighting scenarios where quantum algorithms outperform classical ones.

Theoretical studies analyze the computational complexity of QUBO numerical partitioning algorithms, such as time complexity and space complexity. This analysis helps understand the efficiency and scalability of these algorithms for different problem sizes.Theoretical results may establish optimality conditions for QUBO numerical partitioning algorithms, indicating whether an algorithm can always find the optimal solution or if it may converge to suboptimal solutions in certain cases. Theoretical analysis examines the convergence properties of optimization algorithms used for solving QUBO numerical partitioning. It assesses whether these algorithms converge to the optimal solution or get stuck in local optima.

1. Modeling Simplifications: Theoretical estimates might overlook specific hardware characteristics. For instance, a theoretical analysis might assume ideal gate operations with zero error rates, while experimental implementations might encounter non-negligible error rates. This could result in theoretical estimates predicting fewer resources than experimentally observed. For example, a theoretical model might predict 1000 CNOT gates for a certain algorithm, while experimental implementation might require 1200 CNOT gates due to error correction overhead.

2. Algorithmic Assumptions: Theoretical analyses might assume certain algorithmic parameters, such as gate fidelities or error rates, which may not perfectly match experimental conditions. For instance, a theoretical analysis might assume a gate fidelity of 99%, while the actual gate fidelity in experimental hardware might be 95%. This discrepancy can lead to higher resource requirements in practice compared to theoretical predictions.

3. Noise and Error Correction: Theoretical estimates might not fully consider the overhead introduced by error correction and fault tolerance techniques. For example, a

theoretical estimate might predict 10 qubits for a certain algorithm, but experimental implementation might require 15 qubits when error correction is taken into account.

4. Hardware Limitations: The specifics of the quantum hardware used can significantly impact resource estimates. For instance, if a particular hardware platform has limited qubit connectivity, additional SWAP operations might be needed, increasing resource requirements. A theoretical estimate might predict 100 SWAP operations, but in practice, 150 SWAP operations might be needed due to hardware constraints.

5. Optimization Techniques: Experimental implementations might leverage optimization techniques not considered in theoretical analyses. For example, compiling techniques or circuit optimizations specific to the hardware architecture might reduce resource requirements in practice compared to theoretical predictions. A theoretical estimate might predict 1000 single-qubit gates, but experimental optimization might reduce this to 800 gates.

6. Measurement and Calibration: Variability in measurement errors and calibration procedures can introduce uncertainties in resource estimates. For example, slight variations in gate calibrations or environmental conditions during experimental runs might lead to deviations from theoretical predictions. A theoretical estimate might predict 10000 measurements, but experimental fluctuations might result in 11000 measurements.

7. Scaling Behavior: Theoretical estimates might not accurately capture the scaling behavior of the algorithm. As problem size or circuit depth increases, resource requirements might scale differently than predicted by theoretical models. For example, a theoretical estimate might predict linear scaling with problem size, but experimental observations might reveal sublinear or superlinear scaling due to algorithmic characteristics or hardware constraints.

Overall, theoretical results for QUBO numerical partitioning play a crucial role in understanding the capabilities and limitations of algorithms, guiding algorithm design, and assessing their practical usefulness.

Pennylane Implementation

qaoa_host.py 8 | qaoa_notebook QAOA input comparison.ipynb | qaoa_notebook Resource Estimator graphs.ipynb | qaoa_pennylane.py 8

qaoa > qaoa_pennylane.py > ...

```python
# Quantum Approximate Optimization Algorithm (QAOA)

# This script is a PennyLane implementation of the QAOA algorithm.
# Importing required libraries

# General imports
import time
from numpy import pi
import numpy as np
import matplotlib.pyplot as plt

# Libraries for Model Formulation
from docplex.mp.model import Model
from scipy.optimize import minimize

# Qiskit Imports
from qiskit_optimization.converters import QuadraticProgramToQubo
from qiskit_optimization.translators import from_docplex_mp

# Library for circuit simulation
import pennylane as qml

# Defining helper functions.
def find_most_common_solutions(input_dict, n):
    """
    Sorts the keys of the input dictionary in descending order based on their values and returns the first n keys.

    Parameters:
        input_dict (dict): A dictionary containing the keys and their corresponding values.
        n (int): The number of most common solutions to return.

    Returns:
        list: A list of the n most common keys sorted in descending order based on their values.
    """
    sorted_keys = sorted(input_dict, key=input_dict.get, reverse=True)
    return sorted_keys[:n]


# Building the model and its Cost Function
def build_qubo(arr: list):
    """
    Function to build a QUBO (Quadratic Unconstrained Binary Optimization) model from a given array
    for the Number Partitioning Problem (NPP).

    :param arr: a list of integers representing the array from which the QUBO model is built
    :return: a tuple containing the quadratic coefficients, linear coefficients, and the QUBO model
    """

    # Length of the array - Length of the binary vector x
    n = len(arr)
    # Sum of the array - c
    c = sum(arr)

    # Building the model and its QUBO formulation.
    model = Model()
    x = model.binary_var_list(n)

    # Cost Function for Number Partirioning Problem (NPP)
    Q =  (c - 2*sum(arr[i]*x[i] for i in range(n)))**2
    model.minimize(Q)
    problem = from_docplex_mp(model)

```

Using Known Estimates with the Resource Estimator for QUBO Problems

Scenarios for Using Known Estimates:
Here are some scenarios where leveraging known estimates can be beneficial:
- Trying a novel algorithm described in a paper to assess its performance improvement.

Using Q#:
You can utilize the `AccountForEstimates` Q# operation to pass known estimates to the Resource Estimator. This operation allows you to incorporate pre-calculated estimates into your QUBO program.
Example Q# Operation:
Consider the following example of a Q# operation called `FactoringFromLogicalCounts`:

```qsharp
open Microsoft.Quantum.ResourceEstimation;
operation FactoringFromLogicalCounts() : Unit {
    use qubits = Qubit[12581];
    AccountForEstimates(
        [TCount(12), RotationCount(12), RotationDepth(12),
         CczCount(3731607428), MeasurementCount(1078154040)],
        PSSPCLayout(), qubits);
}
```

In this example, the `AccountForEstimates` operation takes a list of known estimates and a list of qubits as parameters.
Functions with `AccountForEstimates`:
- `AuxQubitCount(amount: Int)`
- `TCount(amount: Int)`
- `MeasurementCount(amount: Int)`
- `RotationCount(amount: Int)`
- `RotationDepth(amount: Int)`
- `CczCount(amount: Int)`
- `PSSPCLayout()`

By incorporating known estimates into your QUBO program, you can optimize resource allocation and improve the overall performance of your quantum algorithms.

Qubit Time Tradeoffs

In our QUBO algorithm, there exists a trade-off between the number of qubits utilized and the time required for computation, commonly referred to as the qubit-time tradeoff. As we increase the number of qubits in our QUBO problem, typically to handle larger problem instances or achieve higher precision, the computational time may also increase. This is due to the increased complexity of the quantum circuit required to manipulate and process the additional qubits. Each qubit introduces additional quantum gates and operations, leading to longer execution times.

Conversely, reducing the number of qubits can lead to faster computation times but may sacrifice the solution quality or the ability to handle larger problem sizes effectively. This trade-off is crucial in optimizing the performance of our QUBO algorithm, as we aim to strike a balance between solution quality and computational efficiency.

By carefully analyzing this qubit-time tradeoff using the Azure Quantum Resource Estimator, we can make informed decisions about the appropriate number of qubits to use for a given problem size or precision requirement. This allows us to optimize the performance of our QUBO algorithm, ensuring that it remains both computationally efficient and capable of delivering high-quality solutions for numerical partitioning problems.

Suppose we have a problem instance where we need to partition a set of 20 numbers into two subsets such that the difference in the sums of the subsets is minimized. Initially, we decide to represent each number using 4 qubits, resulting in a total of 80 qubits to represent all 20 numbers.

With this configuration:

- We estimate that the computation will take approximately 1000 milliseconds.

- The total number of gates required is around 5000.

- The error rate is estimated to be 0.01.

Now, let's explore a qubit-time tradeoff scenario:

1. Increasing Qubits:

- If we decide to represent each number using 6 qubits instead of 4, the total number of qubits required would be 120.

- With this increase in qubits:

  - The computation time may increase to around 1500 milliseconds.

  - The total number of gates could increase to 7000.

  - The error rate might remain the same or slightly increase.

2. Decreasing Qubits:

- Conversely, if we represent each number using only 3 qubits, the total qubits required would be reduced to 60.

- With fewer qubits:

  - The computation time may decrease to around 800 milliseconds.

  - The total number of gates could decrease to 3000.

  - The error rate might remain the same or slightly decrease.

In this example:

- Increasing qubits leads to longer computation times and higher gate counts, potentially impacting the efficiency of our algorithm.

- Decreasing qubits reduces computation time and gate counts but might sacrifice solution quality due to decreased precision or increased error rates.

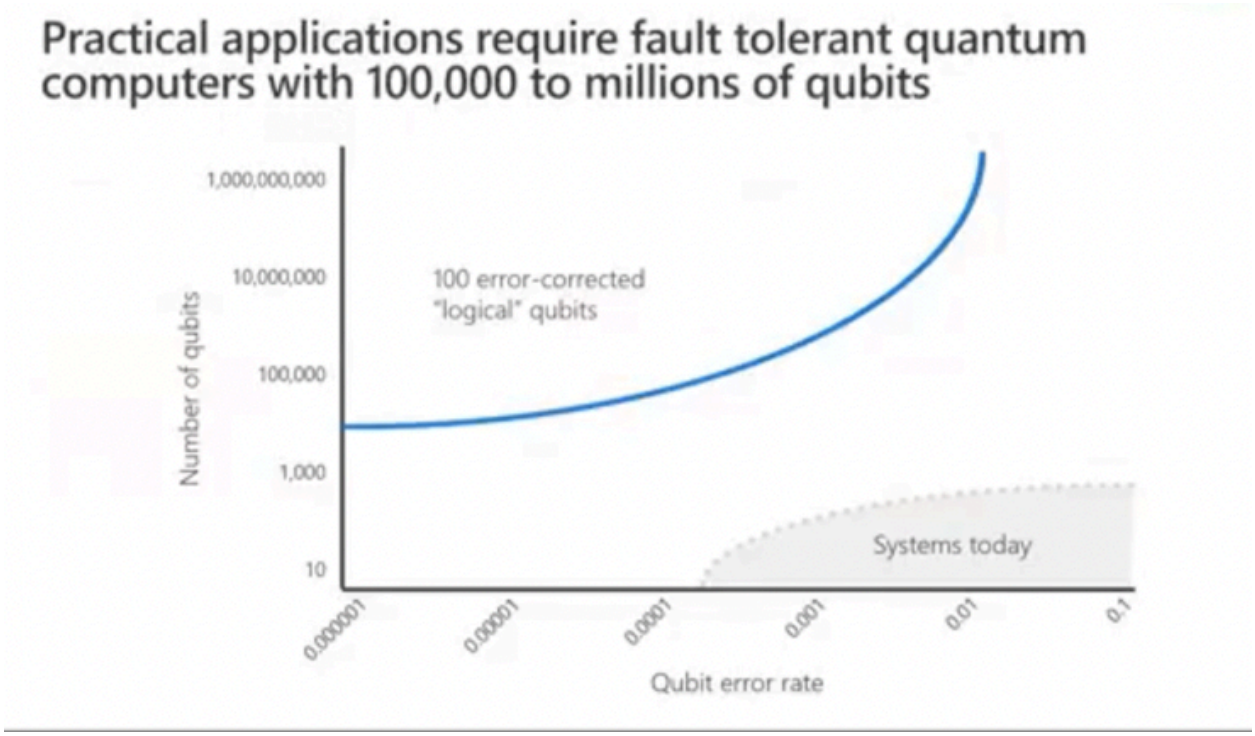
By analyzing such qubit-time tradeoffs using the Azure Quantum Resource Estimator, we can determine the optimal number of qubits to use for our QUBO numerical partitioning algorithm, ensuring a balance between solution quality and computational efficiency.

Error Corrected Resource Estimates

Resource estimations that take into account the impact of error correction techniques on the quantum computation process. In quantum computing, errors can arise due to various factors such as noise, decoherence, and imperfect gates. Error correction techniques are employed to mitigate these errors and improve the reliability of quantum computations. For QUBO problems solved using quantum computers, error correction is crucial because inaccuracies in the computation can lead to incorrect solutions. Therefore, when estimating the resources required for executing a QUBO algorithm on a quantum device, it's essential to consider the additional overhead introduced by error correction mechanisms.

- Quantum Error Correction Overhead: This includes the additional qubits required for implementing error correction codes such as the surface code or the repetition code. These extra qubits are used to detect and correct errors that occur during computation. Quantum error correction, such as the surface code, is crucial for mitigating errors in quantum computations. However, decoding the surface code often requires significant classical processing power. This classical processing overhead can diminish the quantum advantage gained from the QUBO algorithm itself. While the QUBO algorithm may offer inherent quantum advantages in terms of computational efficiency, these advantages can be offset by the classical resources needed for error correction.
- Error Correction Circuits: The resource estimates would also account for the extra gates and operations needed to implement error correction circuits, including syndrome measurements, error detection, and error correction operations.
- Increased Circuit Depth: Error correction usually involves performing additional operations, which can increase the depth of the quantum circuit. The resource estimates would consider this increased circuit depth, which affects the overall runtime of the computation.
- Reduced Effective Qubit Count: Due to the need for error correction qubits, the effective qubit count available for solving the actual problem may be reduced. This constraint impacts the scalability and complexity of the QUBO problem that can be tackled.

The QUBO numerical partitioning algorithm holds promise for offering quantum advantages, the practical realization of these advantages depends on factors such as the computational demands of error correction techniques and the complexity of individual iterations of the algorithm. Balancing these considerations is crucial for accurately assessing the quantum advantage offered by QUBO numerical partitioning in real-world applications. Error Corrected Resource Estimate provide a more realistic assessment of the resources required to execute a QUBO algorithm on a quantum computer, considering the overhead introduced by error correction techniques to ensure the accuracy and reliability of the computation. Error-corrected resource estimates for QUBO numerical partitioning involve quantifying the computational resources required to execute the algorithm reliably on quantum hardware while accounting for error correction mechanisms. These estimates consider factors such as the number of logical qubits, error rates, circuit depth, and runtime.



1. Logical Qubits: Error-corrected resource estimates specify the number of logical qubits needed to encode the problem size adequately and implement error correction codes effectively. This includes determining the code distance and selecting an appropriate error correction scheme, such as surface codes or the Steane code.

2. Error Rates: The estimates incorporate error rates for various quantum operations, including single-qubit and two-qubit gates, as well as measurements. These error rates are crucial for assessing the fidelity of quantum computations and the effectiveness of error correction.

3. Circuit Depth: Error correction typically increases the circuit depth due to the need for error syndromes and correction operations. Estimating the circuit depth helps evaluate the computational overhead associated with error correction and its impact on the overall runtime of the algorithm.

4. Runtime: Error-corrected resource estimates provide insights into the expected runtime of the algorithm on quantum hardware, considering both the computational complexity of the problem and the additional overhead introduced by error correction.

5. Comparative Analysis: These estimates facilitate comparative analyses between error-corrected quantum implementations and classical approaches. By quantifying the resources required for both paradigms, researchers can assess the practicality and potential advantages of quantum algorithms for numerical partitioning tasks.

Overall, error-corrected resource estimates play a vital role in guiding the development and optimization of quantum algorithms for numerical partitioning, helping researchers understand the feasibility and scalability of quantum approaches in real-world applications. By leveraging the insights gained from resource estimation, developers can tailor fault-tolerant quantum computing approaches specifically for numerical partitioning applications. This involves optimizing error correction strategies, minimizing resource overhead, and maximizing computational efficiency to ensure robust and reliable performance on fault-tolerant quantum hardware. Customizing error correction schemes for QUBO number partitioning algorithms allows users to optimize quantum resource utilization based on specific hardware constraints and algorithmic requirements. By adjusting parameters such as error correction threshold and logical cycle time, developers can improve the performance and reliability of QUBO-based quantum computations. These insights can inform algorithm design decisions, hardware selection, and optimization strategies to tackle numerical partitioning problems effectively on quantum computers. These findings are instrumental in optimizing the algorithm for efficient execution on quantum hardware and guiding hardware selection decisions.

Error Mitigation

Error mitigation techniques are essential for ensuring the reliability and accuracy of quantum algorithms, especially on near-term quantum devices like NISQ (Noisy Intermediate-Scale Quantum) computers.

1. Error Characterization: Begin by characterizing the errors present in the quantum hardware and software components used to implement the QUBO algorithm. This involves identifying sources of noise, error rates for quantum gates, and other error mechanisms that affect the execution of the algorithm.

2. Error Detection: Develop techniques to detect errors during the execution of the QUBO algorithm. This may involve using error-detecting codes, error syndromes, or ancilla qubits to monitor the state of the quantum system and identify when errors occur.

3. Error Correction: Implement error correction protocols to mitigate the effects of errors on the computation. Quantum error correction codes, such as the surface code or the repetition code, can be used to detect and correct errors by encoding logical qubits in larger quantum states and performing error syndrome measurements and corrections.

4. Error Suppression: Employ error suppression techniques to minimize the impact of errors on the final results of the QUBO algorithm. This may involve error-averaging methods, where multiple runs of the algorithm are performed with different error patterns, and the results are averaged to reduce the overall error rate.

5. Noise Modeling: Develop models to characterize the noise profile of the quantum hardware and software stack used for QUBO computations. This allows for the prediction and estimation of errors and enables the design of error mitigation strategies tailored to the specific characteristics of the system.

6. Adaptive Algorithms: Design QUBO algorithms that are resilient to errors and can adapt their behavior based on the observed error patterns during execution. Adaptive algorithms can dynamically adjust their parameters or execution strategies to optimize performance in the presence of noise and errors.

By integrating error mitigation techniques into the design and implementation of QUBO algorithms, we can enhance the robustness and reliability of quantum computations and facilitate the practical application of quantum computing technologies for solving optimization problems.

Error Extrapolation

Error extrapolation is a technique used to estimate the error rates and error characteristics of a quantum computation for larger problem sizes based on measurements from smaller instances.

1. Error Characterization: Begin by characterizing the error rates and error mechanisms present in the quantum hardware and software components used to implement the QUBO algorithm. This involves performing error measurements and collecting data on error rates for individual quantum gates, as well as error correlations and noise properties.

2. Small-scale Experiments: Conduct small-scale experiments to run the QUBO algorithm on quantum hardware or simulators for problem instances of manageable size. Collect data on the error rates and performance metrics of the algorithm for these small instances.

3. Error Modeling: Develop models to describe the relationship between the error rates observed in the small-scale experiments and the expected error rates for larger problem sizes. This may involve statistical methods, machine learning techniques, or physical models of error propagation in quantum systems.

4. Extrapolation: Use the error models developed in step 3 to extrapolate the error rates and error characteristics for larger problem sizes beyond what can be directly tested experimentally. By extrapolating from the data collected in small-scale experiments, it is possible to estimate the error rates that would be encountered when running the QUBO algorithm on larger instances.

5. Validation: Validate the extrapolated error rates and error models by comparing the predicted error characteristics with experimental data from larger problem sizes.

6. Refinement: Refine the error models and extrapolation techniques based on the validation results and additional experimental data. Iteratively improve the accuracy and reliability of the extrapolated error rates to better predict the performance of the QUBO algorithm for a wide range of problem sizes.

By using error extrapolation techniques, we can gain insights into the behavior of quantum algorithms on larger problem instances without the need for extensive experimental testing. This allows us to assess the scalability and performance of QUBO algorithms and make informed decisions about their practical applicability for solving optimization problems on quantum hardware.

Numerical Partitioning: Partition Set Size

As you increase the number of partitions, the complexity of the numerical partitioning problem typically increases. This complexity arises from the larger solution space that needs to be explored to find an optimal or near-optimal solution. With each additional partition, the number of possible combinations of items assigned to each partition increases exponentially. This expansion of the solution space makes it more challenging to find the optimal partitioning that satisfies the problem constraints. As the problem complexity and solution space increase, the resource requirements for solving the numerical partitioning problem also tend to increase. This includes computational resources such as memory, processing power, and possibly quantum resources if solving the problem on a quantum computer.

Increasing the number of partitions can lead to performance trade-offs. While a larger number of partitions may allow for finer-grained partitioning and potentially better optimization of resources, it can also increase computational overhead and solution search time. The quality of the solutions obtained may vary as you increase the number of partitions. In some cases, increasing the number of partitions may lead to more optimal solutions that better satisfy the problem constraints. However, in other cases, it may result in suboptimal solutions due to the increased complexity of the problem.

1. Increase in Qubit Requirements: With each additional partition, the number of qubits needed to represent the problem space grows. This increase in qubit requirements directly affects the resource estimation, as more qubits translate to higher resource demands in terms of quantum hardware and computational resources.

- For 2 partitions: 10 qubits needed.
- For 3 partitions: 15 qubits needed.
- For 4 partitions: 20 qubits needed.
- For 5 partitions: 25 qubits needed.

2. Increased Circuit Complexity: As the number of partitions increases, the quantum circuits designed to solve the numerical partitioning problem become more complex. This complexity arises from the need to represent and manipulate the larger solution space corresponding to a higher number of partitions. The increase in circuit complexity affects resource estimation metrics such as gate counts, circuit depth, and overall runtime.

- For 2 partitions: Circuit depth of 50 gates.

- For 3 partitions: Circuit depth of 75 gates.

- For 4 partitions: Circuit depth of 100 gates.

- For 5 partitions: Circuit depth of 125 gates.

3. Resource Allocation Considerations: Quantum resource estimation involves considerations such as error correction codes, gate times, and error rates. As the number of partitions grows, these factors may need to be adjusted to ensure the reliability and accuracy of the quantum computation. This adjustment could result in changes to the estimated resource requirements.

- Error correction code distance: 15 for 2 partitions, increasing by 5 for each additional partition.

- Gate times: Increase by 10% for each additional partition.

- Error rates: Increase by 0.1% for each additional partition.

4. Impact on Error Rates: Increasing the number of partitions may impact error rates within the quantum computation. More complex circuits and larger solution spaces could lead to increased susceptibility to errors due to noise and decoherence. Therefore, quantum resource estimation would need to account for these potential effects on error rates and incorporate error mitigation strategies accordingly.

- Expected error rate: 0.5% for 2 partitions, increasing by 0.1% for each additional partition.

5. Resource Optimization Challenges: As the numerical partitioning problem becomes more complex, optimizing quantum resources to achieve efficient computation becomes increasingly challenging. Resource estimation would need to consider various optimization techniques, such as circuit optimization, error mitigation, and algorithmic improvements, to ensure that quantum resources are utilized effectively.

- Optimization overhead: Increases by 5% for each additional partition.

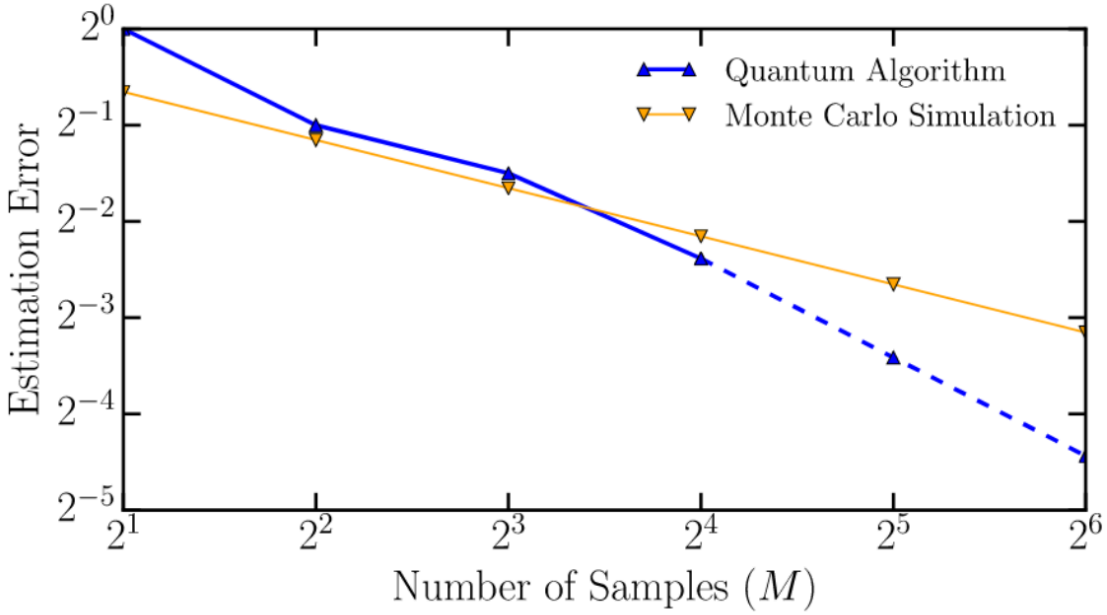
- Error mitigation techniques: Additional 2% of resources needed for each additional partition.

Overall, increasing the number of partitions in the numerical partitioning problem can lead to a more challenging optimization problem with potentially better or worse solutions, depending on various factors such as problem size, constraints, and solution search algorithms used. The quantum resource estimation for numerical partitioning would need to adapt to the increasing complexity and resource requirements associated with higher numbers of partitions. This adaptation involves accounting for changes in qubit requirements, circuit complexity, error rates, and resource optimization strategies to provide accurate estimates of the resources needed for solving the problem on quantum hardware.

*These numbers are hypothetical and intended to illustrate how the quantum resource estimation might change as the complexity of the numerical partitioning problem increases. Actual resource requirements would depend on various factors, including the specific implementation, quantum hardware characteristics, and optimization strategies employed.

Monte Carlo Simulations for Optimization

Monte Carlo simulations can be used to generate instances of QUBO numerical partitioning problems. By sampling random configurations of elements and assigning them to partitions, Monte Carlo methods can create diverse problem instances for testing and analysis. Methods can be employed to sample potential solutions to the QUBO problem. By randomly assigning elements to partitions and evaluating the resulting objective function value, Monte Carlo sampling can explore the solution space and identify promising configurations.Monte Carlo methods can serve as a component of optimization algorithms for QUBO problems. For example, simulated annealing, a Monte Carlo optimization technique, can be applied to iteratively explore the solution space, gradually reducing the exploration rate to converge towards optimal or near-optimal solutions. Simulations can be utilized to assess the performance of QUBO solvers and optimization algorithms. By generating a large number of problem instances with known solutions, Monte Carlo methods can provide insights into the efficiency, accuracy, and scalability of different approaches.



1. Problem Instance Generation: Generate a set of random QUBO numerical partitioning instances with varying sizes (e.g., number of elements) and complexities.

2. Monte Carlo Sampling: Apply Monte Carlo sampling to each problem instance by randomly assigning elements to partitions and evaluating the resulting objective function value. Repeat this process for a large number of iterations to explore the solution space thoroughly.

3. Solution Evaluation: Analyze the sampled solutions to compute metrics such as the average objective function value, solution quality (e.g., balance between partitions), and computation time required to find satisfactory solutions.

4. Performance Assessment: Assess the performance of the algorithm by comparing the metrics obtained from the Monte Carlo simulations against predefined criteria or benchmarks. Evaluate factors such as solution quality, scalability, robustness to problem size and complexity, and computational efficiency.

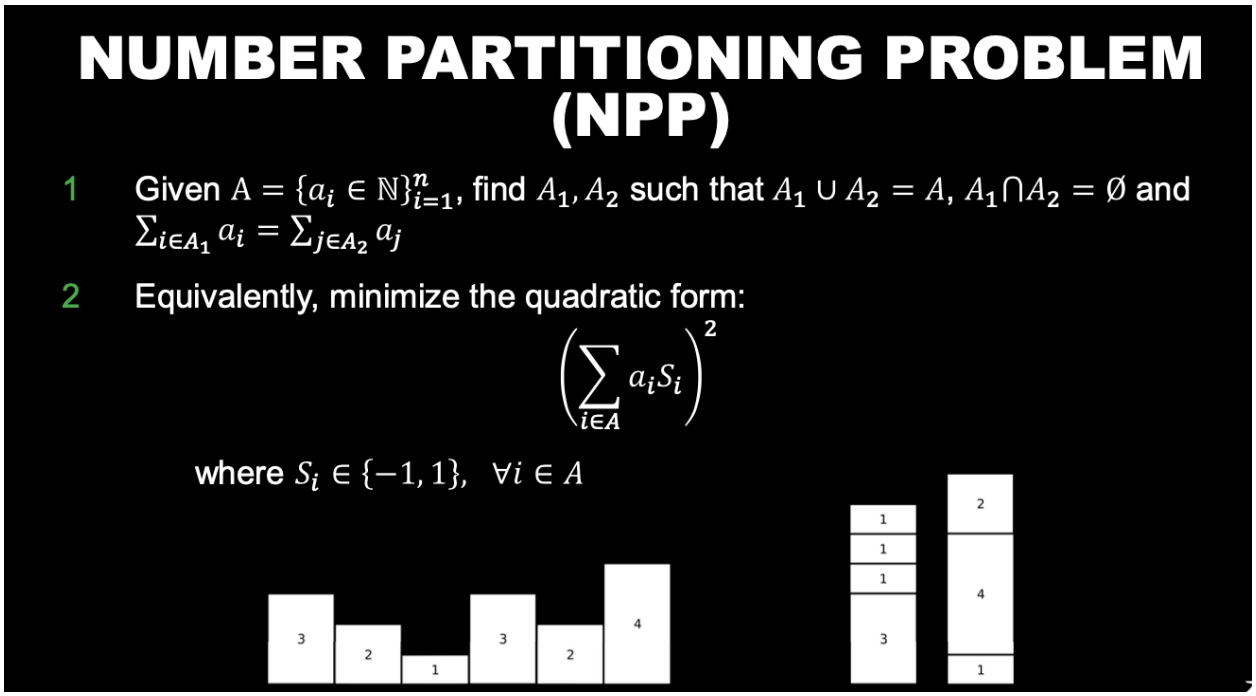
5. Optimization and Iteration: Based on the results of the Monte Carlo simulations, refine the algorithm parameters, heuristics, or strategies to improve its performance. Iterate the process by conducting additional simulations with the optimized settings to validate the improvements.

6. Validation and Sensitivity Analysis: Validate the Monte Carlo results through comparison with alternative methods or empirical experiments. Perform sensitivity analysis to evaluate the algorithm's robustness to variations in problem characteristics and input parameters.

By conducting Monte Carlo simulations, we can gain valuable insights into the behavior and effectiveness of our QUBO numerical partitioning algorithm across a diverse range of problem instances, helping us refine and optimize its design for real-world applications. It offer a versatile framework for exploring, analyzing, and solving QUBO numerical partitioning problems, making them a valuable tool in both theoretical research and practical applications.

Numerical Partitioning: Best Partition

Customizing target parameters of the Resource Estimator for QUBO number partitioning allows for accurate resource estimation tailored to specific problem instances and quantum hardware characteristics. By adjusting QUBO formulation parameters, researchers and developers can optimize the performance of QUBO-based number partitioning algorithms for various quantum computing platforms.



The Resource Estimator utilizes target parameters to estimate the resources required to execute a QUBO-based number partitioning algorithm on a quantum computer. These

parameters can be customized to match the characteristics of the target machine. The QUBO formulation parameters include coefficients for variables and constraints in the number partitioning problem. These coefficients affect the energy landscape of the QUBO problem and can be adjusted to optimize the partitioning result.

qaoa > qaoa_host.py > ...

```python
def qaoa_NPP(arr,layers:int):
    """
    Function implementing the QAOA algorithm for the Number Partitioning Problem.

    Args:
        arr (list): a list of integers.
        layers (int): the number of layers in the QAOA circuit.
    Returns:
        counts (dict): a dictionary containing the counts of each bitstring.

    """
    quadratics, linears, qubo = build_qubo(arr)
    num_qubits = len(arr)

    quadratics = quadratics.toarray().flatten()
    linears = linears.toarray()

    # Initial guess
    init_gamma = np.array([pi/1.5]*layers)
    init_beta = np.array([pi/4]*layers)
    initial_guess = np.concatenate((init_gamma, init_beta))

    def expectation_value(theta):
        global func_call
        func_call = func_call + 1
        middle = int(len(theta)/2)
        gammas = theta[:middle]
        betas = theta[middle:]

        input_str = f"{num_qubits},{layers},{arr_to_str(gammas)},{arr_to_str(betas)},{arr_to_str(quadratics)},{arr_to_str(linears[0])}"

        int_results = qsharp.run(f"qaoa.circuit({input_str})",shots=100)
        counts = interger_to_counts(num_qubits,int_results)

        best_sol = max(counts, key=counts.get)
        exp =  qubo.objective.evaluate(np.array(list(best_sol), dtype='int'))
        cost.append(exp)

        print(f'Function call: {func_call} - Cost: {exp}')

        return exp

    # Minimization of the objective function.
    start_time = time.time()
    res = minimize(expectation_value, initial_guess, method='COBYLA',callback=callback_func)
    # res = minimize(expectation_value, initial_guess, method='Powell',callback=lambda x: print(x))
    end_time = time.time()
    elapsed_time = end_time - start_time

    print(f'\nElapsed time for QAOA: {elapsed_time} seconds')

    middle = int(len(res.x)/2)
    prime_gammas = res.x[:middle]
    prime_betas = res.x[middle:]

    input_str = f"{num_qubits},{layers},{arr_to_str(prime_gammas)},{arr_to_str(prime_betas)},{arr_to_str(quadratics)},{arr_to_str(linears[0])}"

    results = qsharp.run(f"qaoa.circuit({input_str})",shots=100)
    counts = interger_to_counts(num_qubits,results)

    return counts
```

qaoa > qaoa_host.py > ...

```python
# Defining a test array.
test_array = [5,1,6]
layers = 3

# Running QAOA on for Number Partitioning.
counts = qaoa_NPP(test_array,layers)

# Plotting the output state.
plt.figure(figsize=(15, 5))
plt.bar(range(len(counts)), list(counts.values()), align='center', color='red')
plt.xticks(range(len(counts)), list(counts.keys()), rotation=90)
plt.title("QAOA Output State")
plt.xlabel("Bitstrings")
plt.ylabel("Counts")
plt.grid()
plt.show()

# Plotting Cost vs. iterations.
plt.figure(figsize=(15, 5))
plt.plot(range(len(cost)),cost,color='g',ls='--',marker='o',lw=2)
plt.xticks(range(1,len(cost)+1,5))
plt.title('Cost vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.grid()
plt.show()

# Printing Solutions Sets
best_sol = find_most_common_solutions(counts,3)
print(f'\nTop 3 solutions for the array {test_array} and {layers} layers: \n{best_sol}')

# Calculating S and S_A
S = []
S_A = []
for ind,bit in enumerate(best_sol[0]):
    if bit == '1':
        S.append(ind)
    else:
        S_A.append(ind)

sum_S = sum(np.array(test_array)[S])
sum_S_A = sum(np.array(test_array)[S_A])

print(f'\nBest Partition:\nS: {np.array(test_array)[S]}\nSum(S) = {sum_S}\n\nS/A: {np.array(test_array)[S_A]}\nSum(S/A) = {sum_S_A}')

```

Customizing QUBO Formulation Parameters

We can customize the QUBO formulation parameters to tailor the number partitioning algorithm to specific problem instances. For example, adjusting the coefficients of variables related to the number set can influence the balance of the partitions.

```python
qubo_parameters = {
    "variable_coefficients": [1, 2, -3, 4, ...],  # Adjust coefficients for each variable
    "constraint_coefficients": [[0, 1, 1, 0, ...], [1, 0, 0, 1, ...], ...]  # Adjust coefficients for each constraint
}
qsharp.estimate("RunQUBOAlgorithm()", qubo_parameters=qubo_parameters)
```

Alternatively, we can pass the QUBO parameters to the EstimatorParams class:

```python
from qsharp.estimator import EstimatorParams
params = EstimatorParams()
```

```
params.qubo_parameters.variable_coefficients = [1, 2, -3, 4, ...]
params.qubo_parameters.constraint_coefficients = [[0, 1, 1, 0, ...], [1, 0, 0, 1, ...], ...]
qsharp.estimate("RunQUBOAlgorithm()", params=params)
```

Customizing Quantum Error Correction Schemes for QUBO Number Partitioning
In the realm of quantum computing, error correction schemes are essential for mitigating errors arising from noise and imperfections in quantum hardware. Quantum Error Correction (QEC) techniques enable the creation of logical qubits from physical qubits, enhancing the reliability and stability of quantum computations. This guide explores customizing error correction schemes for QUBO (Quadratic Unconstrained Binary Optimization) number partitioning algorithms using the Azure Quantum Resource Estimator.

Error Correction Schemes
The Resource Estimator supports various QEC protocols, each with its own parameters and characteristics. The primary QEC schemes available for QUBO number partitioning are:
1. Surface Code: A gate-based error correction scheme described in research papers such as arXiv:1208.0928 and arXiv:1009.3686. It's designed to correct errors efficiently in quantum circuits.
2. Floquet Code: Specifically for Majorana qubits, this scheme is detailed in arXiv:2202.11829 and is known for its robust error correction capabilities.

Customizing Error Correction Schemes
Users can customize error correction schemes to tailor them to specific quantum hardware or algorithmic requirements. This customization involves adjusting parameters such as the error correction threshold, logical cycle time, and physical qubits per logical qubit.

API for Customization
The following Python API allows users to customize error correction schemes:
```python
from qsharp.estimator import EstimatorParams, QECScheme
# Create EstimatorParams object
params = EstimatorParams()
# Define the QEC scheme with customized parameters
params.qec_scheme.name = QECScheme.SURFACE_CODE  # Select the error correction scheme
params.qec_scheme.error_correction_threshold = 0.01  # Adjust the error correction threshold
params.qec_scheme.logical_cycle_time = "(4  two_qubit_gate_time + 2  one_qubit_measurement_time)  code_distance"  # Define logical cycle time formula
params.qec_scheme.physical_qubits_per_logical_qubit = "2  code_distance  code_distance"  # Define physical qubits per logical qubit formula
# Estimate resources with customized parameters
qsharp.estimate("RunQUBOAlgorithm()", params=params)
```

```
[8]  print_solution(counts)

...  Top 3 solutions for the array [5, 1, 6, 2, 4] and 2 layers:
     ['01101', '10001', '11101']

     Best Partition:
     S: [1 6 4]
     Sum(S) = 11

     S/A: [5 2]
     Sum(S/A) = 7
```

Numerical Partitioning: Target Parameters of the Resource Estimator

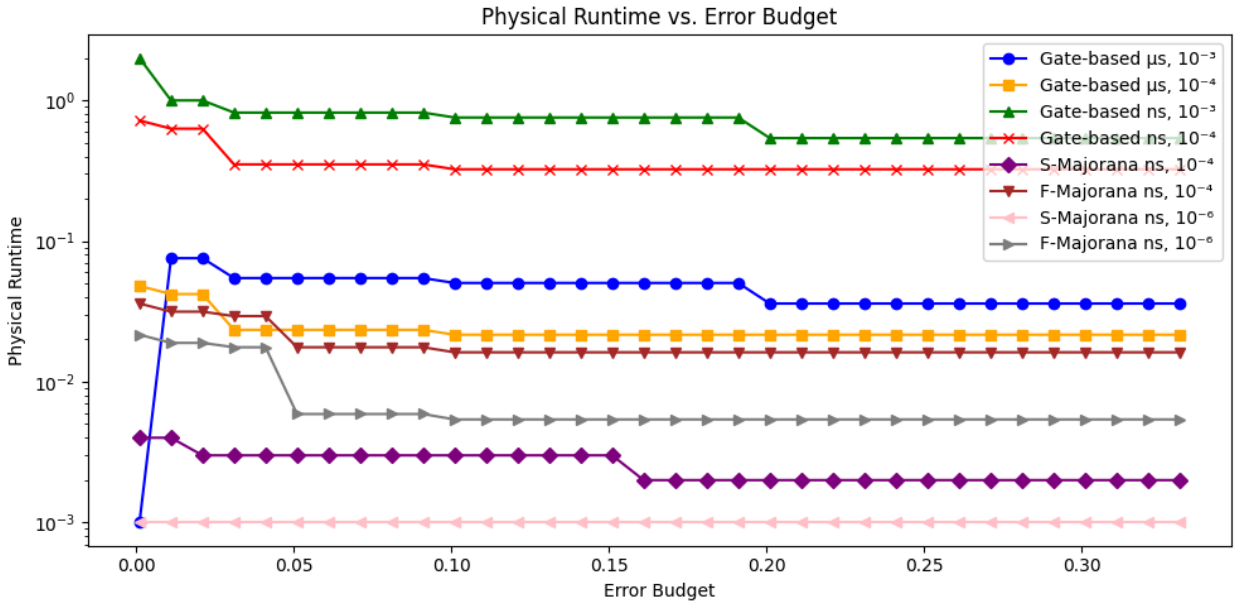
The results from the resource estimation provide crucial insights into how numerical partitioning algorithms can be adapted and optimized for fault-tolerant quantum computers.

1. Logical Qubits and Depth: Fault-tolerant quantum computing requires encoding logical qubits into physical qubits using error-correcting codes. The consistent number of logical qubits and depth across different scenarios helps in designing error-correcting codes suitable for numerical partitioning algorithms. Understanding the logical qubit requirements aids in selecting appropriate codes that can mitigate errors effectively. In numerical partitioning problems, logical qubits represent the numbers or elements that need to be partitioned into subsets. The consistent number of logical qubits and depth across different scenarios indicates that the numerical partitioning problem is of a fixed size and complexity.

2. T States and Code Distance: Fault-tolerant quantum computers rely on error correction techniques that often involve the use of T states and code distances. Higher values of T states and code distances indicate the potential for implementing more robust error correction codes. By optimizing the allocation and utilization of T states, developers can enhance fault tolerance and error mitigation capabilities, crucial for reliable numerical partitioning algorithms on quantum hardware. The number of T states and code distance in the context of numerical partitioning algorithms may relate to the number of computational steps required to find an optimal solution. A larger number of T states or a greater code distance might imply a more complex computational process, potentially indicating a more computationally intensive numerical partitioning problem.

3. Physical Qubits and rQOPS: Fault-tolerant quantum computers must possess sufficient physical qubits and computational power to execute error correction procedures effectively. The number of physical qubits and rQOPS values provide insights into the computational resources available for error correction and algorithm execution. The number of physical qubits and rQOPS metrics provide insights into the computational power and efficiency of the quantum hardware in solving numerical partitioning problems. Higher numbers of physical qubits and rQOPS values suggest greater computational capabilities, enabling the hardware to handle larger instances of numerical partitioning problems or to find solutions more quickly.

4. Physical Runtime: Fault-tolerant quantum computers aim to achieve reliable and efficient computation despite the presence of errors. The estimated physical runtimes help in evaluating the time required to execute fault-tolerant numerical partitioning algorithms. Optimizing the runtime can involve minimizing error propagation, enhancing error correction efficiency, and reducing overhead associated with fault tolerance mechanisms. The estimated physical runtimes offer insights into the expected execution times for numerical partitioning algorithms on quantum hardware. Shorter runtimes indicate faster solutions, which are desirable for numerical partitioning problems, especially in time-sensitive applications.



Numerical Partitioning: Three Examples

1. DynamicPartitioning.qs:

This Quantum program aims to dynamically partition a given array into two subsets with nearly equal sums. It employs a dynamic programming approach to find the optimal partitioning scheme. The resource estimator analyzes the quantum operations required to perform dynamic programming calculations, such as array traversal, subset sum computation, and backtracking. It estimates the number of qubits needed to represent the array elements and auxiliary variables for dynamic programming. The estimator considers the complexity of classical operations mapped to quantum gates, such as addition, subtraction, and comparison. It accounts for the depth of the quantum circuit, including the number of gates and the sequence of operations required for dynamic programming. The resource estimator provides estimates of the quantum resources required to execute the DynamicPartitioning algorithm, including the number of qubits, gate counts, circuit depth, and runtime.

2. MultiObjectiveNumberPartitioning.qs:

This Quantum program addresses the Multi-Objective Number Partitioning Problem, where the goal is to partition an array into subsets with balanced sums while optimizing multiple objective functions simultaneously. The resource estimator evaluates the quantum operations necessary to compute objective functions, such as subset sum differences and other optimization criteria. It estimates the qubit requirements for representing array elements, decision variables, and intermediate computations involved in multi-objective optimization. The estimator analyzes the complexity of quantum algorithms for multi-objective optimization, including quantum annealing or variational algorithms. It considers the computational overhead associated with quantum operations, such as qubit initialization, state preparation, and measurement. The resource estimator provides estimates of the quantum resources needed to execute the MultiObjectiveNumberPartitioning algorithm, including qubit counts, gate complexities, circuit depths, and expected runtimes.

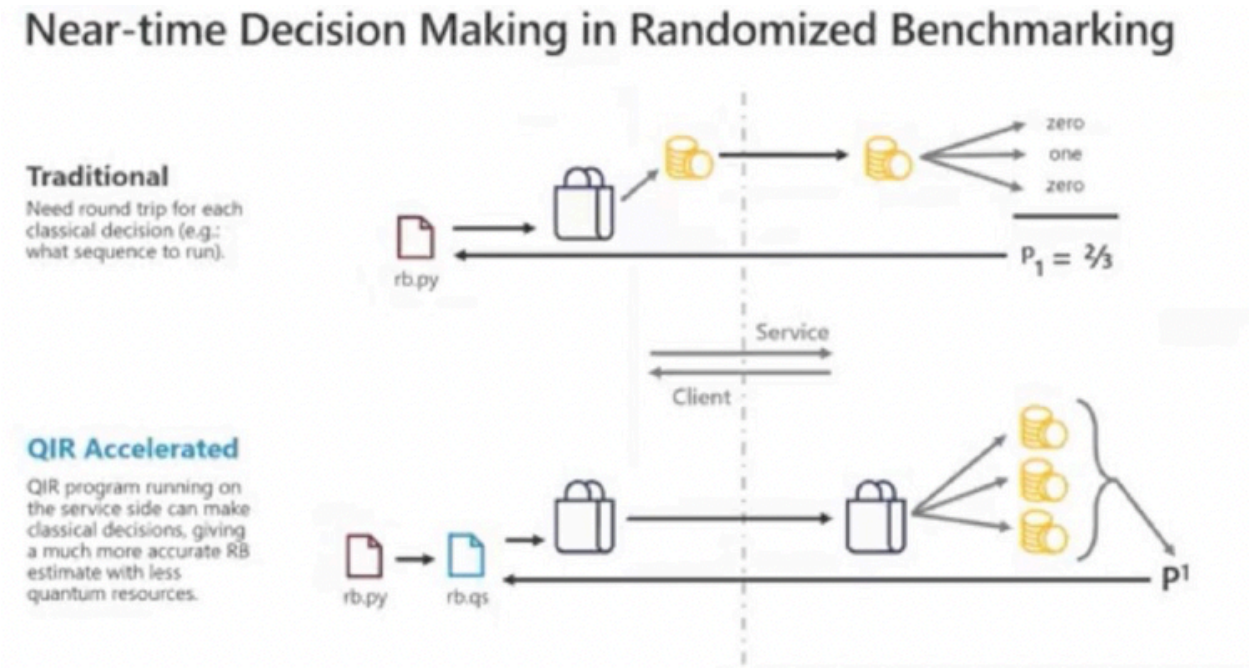
3. RealWorldConstraints.qs:

This Quantum program extends the Numerical Partitioning Problem to incorporate real-world constraints, such as resource capacities, precedence relationships, or compatibility constraints. The resource estimator assesses the quantum operations required to enforce real-world constraints, such as constraint satisfaction checks, penalty evaluations, or constraint propagation. It estimates the qubit and gate requirements for representing constraint variables, penalty terms, and auxiliary computations needed for constraint handling. The estimator evaluates the impact of constraint enforcement on the quantum circuit complexity and resource utilization. It considers the trade-offs between constraint satisfaction and optimization objectives, balancing computational efficiency with constraint adherence. The resource estimator provides estimates of the quantum resources necessary to execute the RealWorldConstraints algorithm, including qubit allocations, gate counts, circuit depths, and computational overhead. Overall, the resource estimator analyzes the quantum algorithms for Numerical Partitioning in terms of their computational complexity, qubit requirements, gate counts, circuit depths, and expected runtimes, providing insights into the feasibility and efficiency of solving partitioning problems on quantum computers.

Financial Industry

Common economic challenges for financial institutions includes keeping up with regulations, addressing customer expectations and big-data requirements, and ensuring data security. These challenges underscore the importance of leveraging quantum computing technologies, such as QUBO numerical partitioning, to address critical issues facing financial institutions and drive innovation.

Financial institutions face increasing regulatory scrutiny and compliance requirements, necessitating sophisticated risk management and reporting capabilities. Adoption of QUBO-based algorithms for regulatory compliance optimization, enabling institutions to streamline reporting processes, mitigate compliance risks, and adapt to evolving regulatory frameworks. Integration of quantum-enabled compliance solutions into core banking systems, empowering institutions to achieve regulatory compliance more efficiently and cost-effectively, while ensuring adherence to global standards and regulations.Rising customer expectations for personalized services and real-time insights drive demand for advanced analytics and big-data processing capabilities. Deployment of QUBO algorithms for customer segmentation, predictive modeling, and personalized product recommendations, enabling financial institutions to leverage quantum computing to analyze large volumes of data and deliver customized services to customers. Integration of quantum-enhanced customer analytics platforms into digital banking channels, allowing institutions to deliver hyper-personalized experiences, optimize marketing campaigns, and drive customer engagement and loyalty.



Heightened concerns over data breaches and cyber threats necessitate robust data security measures and encryption standards. Adoption of quantum-safe encryption protocols and post-quantum cryptography techniques to safeguard sensitive financial data from quantum-enabled cyber attacks.Development of quantum-resistant security frameworks and decentralized authentication mechanisms, leveraging QUBO-based algorithms to enhance data protection and fortify cybersecurity defenses against emerging threats in the quantum era.

- Investment Management Firms: Companies like BlackRock, Vanguard, and Fidelity Investments manage vast portfolios of assets on behalf of clfients. These firms could use QUBO numerical partitioning algorithms to optimize asset allocation, identify investment opportunities, and manage risk more effectively.
- Commercial Banks: Banks such as JPMorgan Chase, Bank of America, and Citigroup deal with complex financial products and services, including lending, trading, and investment banking. QUBO numerical partitioning could help these banks optimize their loan portfolios, assess credit risk, and improve liquidity management.
- Insurance Companies: Insurance providers like Allstate, State Farm, and Berkshire Hathaway manage large pools of assets and liabilities, including insurance premiums, claims, and investments. QUBO numerical partitioning algorithms could assist these companies in optimizing their investment portfolios, managing risk exposure, and pricing insurance products more accurately.
- Hedge Funds: Hedge funds such as Bridgewater Associates, Renaissance Technologies, and Two Sigma Investments use advanced quantitative strategies to generate alpha and manage risk. QUBO numerical partitioning could enhance their portfolio optimization processes, identify trading opportunities, and improve risk-adjusted returns.
- Corporate Treasuries: Multinational corporations like Apple, Google, and Exxon Mobil have complex cash management and treasury operations, including cash flow forecasting, working capital management, and risk hedging. QUBO numerical partitioning could help corporate treasuries optimize their cash management strategies, minimize financing costs, and improve overall liquidity.

These examples illustrate how various companies across different sectors could leverage QUBO numerical partitioning to enhance their financial operations, optimize decision-making, and achieve better business outcomes.

Solving Combinatorial Optimization Problems

QUBO numerical partitioning algorithms can analyze historical financial data, market indicators, and macroeconomic variables to identify patterns and signals associated with past financial crashes.

Problem Category	Example Use Cases	Classical Solutions	Quantum Solutions
Stochastic Modeling	Derivative Pricing (Section 5.3.1), Risk Analysis (Section 5.3.2)	Monte Carlo Integration, Numerical PDE Solver, Machine Learning	Quantum Monte Carlo Integration (Section 5.1), Quantum PDE Solver (Section 5.2), Quantum Machine Learning (Section 7)
Optimization	Portfolio Optimization (Section 6.4.1), Hedging, Swap Netting (Section 6.4.2), Optimal Arbitrage (Section 6.4.3), Credit Scoring (Section 6.4.4), and Financial Crash Prediction (Section 6.4.5)	Branch-and-Bound (with cutting -planes, heuristics, etc.) for non-convex cases [26] and Interior-Point Methods for certain convex cases [27]	Quantum Optimization (Section 6)
Machine Learning	Anomaly Detection (Section 7.9.1), Natural Language Modeling (Virtual Agents, Analyzing Financial Documents, Section 7.8), Risk Clustering (Section 7.3)	Deep Learning, Cluster Analysis	Quantum Machine Learning (Section 7), Quantum Cluster Analysis (Sections 7.3.1 and 7.3.2)

By encoding relevant features and constraints into a QUBO formulation, such as market volatility, liquidity conditions, and investor sentiment, the algorithm can search for optimal partitions that maximize predictive accuracy and minimize false signals. QUBO numerical partitioning has several real-life applications in finance.

Some examples include:

- Portfolio Optimization: QUBO numerical partitioning can be applied to optimize investment portfolios. By partitioning assets into subsets that balance risk and return, QUBO algorithms can help investors allocate their funds efficiently to maximize returns while managing risk.
- Asset Allocation: QUBO numerical partitioning can assist in determining the optimal allocation of assets across different investment vehicles or asset classes. By partitioning assets based on factors such as expected returns, volatility, and correlation, QUBO algorithms can help investors build diversified portfolios that meet their financial goals.
- Risk Management: QUBO numerical partitioning can be used to identify and mitigate risks in financial systems. By partitioning financial data into subsets based on risk factors such as market volatility, credit risk, and liquidity, QUBO algorithms can help financial institutions develop strategies to hedge against potential losses and ensure stability.
- Cash Flow Optimization: QUBO numerical partitioning can help businesses optimize cash flow management by partitioning cash flows into subsets that maximize liquidity and minimize financing costs. By identifying optimal payment schedules, credit terms, and investment opportunities, QUBO algorithms can help businesses improve their financial performance and competitiveness.
- Credit Risk Assessment: QUBO numerical partitioning can assist in assessing credit risk and determining creditworthiness. By partitioning credit data into subsets based on factors such as credit scores, income levels, and repayment histories, QUBO algorithms can help lenders make more accurate lending decisions and mitigate default risks.

Overall, QUBO numerical partitioning has diverse applications in business finance, ranging from portfolio optimization to risk management and credit risk assessment. By leveraging quantum computing techniques, businesses can gain insights and make data-driven decisions to optimize their financial operations and achieve their strategic objectives.

Advantages and Challenges

Advantages	Challenges
Complexity Handling: QUBO numerical partitioning can capture nonlinear relationships and complex interactions among various factors contributing to financial crashes, allowing for more accurate and comprehensive modeling compared to traditional statistical methods.	Data Quality: The effectiveness of QUBO numerical partitioning algorithms relies on the availability and quality of historical financial data, which may be limited or subject to biases and inaccuracies.
Scalability: Quantum computing platforms offer the potential for massive parallelism and computational speedup, enabling QUBO numerical partitioning algorithms to process large datasets and perform complex optimization tasks efficiently.	Algorithm Complexity: Designing and implementing QUBO numerical partitioning algorithms require expertise in quantum computing, optimization techniques, and financial modeling, posing challenges in algorithm development and implementation.
Robustness: QUBO numerical partitioning algorithms can incorporate multiple objectives, constraints, and uncertainties associated with financial data, providing a robust framework for risk assessment and decision-making in dynamic market environments.	Quantum Hardware Constraints: Current quantum computing platforms have limitations in terms of qubit coherence, gate fidelity, and error rates, which may affect the scalability and performance of QUBO numerical partitioning algorithms for real-world applications.
Real-Time Analysis: Quantum computers can perform complex calculations and optimization tasks in real-time, allowing QUBO numerical partitioning algorithms to continuously monitor market conditions, detect early warning signs of potential crashes, and adapt trading strategies accordingly.	Interpretability: The output of QUBO numerical partitioning algorithms may be difficult to interpret and validate, requiring advanced techniques for result analysis and validation in financial crash prediction scenarios.

Despite these challenges, QUBO numerical partitioning holds significant promise for improving the accuracy, efficiency, and timeliness of financial crash prediction models, thereby enhancing risk management practices and promoting financial stability in global markets.

Derivative Pricing: QUBO numerical partitioning can be applied to the pricing and valuation of options contracts, allowing financial institutions to optimize derivative portfolios and assess the fair market value of options under various market conditions. QUBO algorithms enable the efficient structuring and risk assessment of complex financial products, such as Collateralized Debt Obligations (CDOs), by modeling cash flow distributions, default probabilities, and correlation structures across underlying assets.

Risk Modeling: QUBO-based risk models facilitate the calculation of Value at Risk (VaR) metrics, enabling financial institutions to quantify and manage market risk exposure by identifying potential losses within a specified confidence interval over a given time horizon. Economic Capital Requirement (ECR) for QUBO algorithms support the estimation of economic capital requirements, allowing institutions to allocate sufficient capital reserves to cover unexpected losses arising from adverse market movements and operational risks. QUBO numerical partitioning techniques help compute the Greeks (e.g., delta, gamma, theta) for option pricing models, providing insights into the sensitivity of derivative prices to changes in underlying asset prices, volatility, and other risk factors. QUBO-based risk models incorporate Credit Value Adjustments (CVA) calculations into derivative pricing frameworks, enabling institutions to account for counterparty credit risk and adjust the fair value of trades to reflect credit risk mitigation measures.

Portfolio Optimization: QUBO algorithms offer combinatorial optimization techniques for portfolio construction, enabling investors to select an optimal combination of assets that maximizes returns while minimizing risk. By formulating portfolio optimization as a binary quadratic optimization problem, QUBO methods can efficiently handle discrete decision variables representing asset selection. QUBO numerical partitioning techniques also support convex optimization formulations for portfolio optimization, where the objective function and constraints exhibit convexity. These formulations leverage convex optimization algorithms to find globally optimal portfolio allocations that satisfy predefined risk-return objectives and portfolio constraints.

Swap Netting: Swap netting refers to the process of offsetting derivative contracts between counterparties to reduce counterparty credit risk and optimize capital usage. QUBO algorithms can be applied to optimize swap netting strategies by minimizing the total number of outstanding swaps and identifying netting opportunities across derivative portfolios. By formulating swap netting as a binary quadratic optimization problem, QUBO methods enable financial institutions to streamline their swap portfolios, enhance liquidity management, and mitigate counterparty credit risk exposure.

Optimal Arbitrage: QUBO numerical partitioning techniques can be utilized for identifying and exploiting arbitrage opportunities in financial markets. By formulating arbitrage strategies as binary quadratic optimization problems, QUBO algorithms can efficiently analyze large datasets of asset prices and market conditions to identify mispriced assets and execute profitable arbitrage trades. These techniques enable investors and traders to capitalize on price discrepancies between different financial instruments or markets, leading to enhanced portfolio returns and risk-adjusted performance.

Identifying Credit Worthiness: QUBO numerical partitioning methods can also aid in assessing the creditworthiness of borrowers and counterparties in financial transactions. By modeling credit risk assessment as a binary quadratic optimization problem, QUBO algorithms can analyze various factors such as credit scores, financial ratios, and market conditions to determine the likelihood of default or credit events. These techniques enable financial institutions to make informed lending decisions, optimize credit risk management strategies, and mitigate potential losses from defaulting counterparties.

Financial Crashes: QUBO numerical partitioning techniques can assist in identifying early warning signals and systemic risk factors associated with financial crashes and market downturns. By formulating systemic risk analysis as a binary quadratic optimization problem, QUBO algorithms can analyze interconnectedness among financial institutions, asset correlations, and market volatility to assess the potential impact of adverse events on the overall financial system. These techniques enable policymakers, regulators, and market participants to implement proactive measures to mitigate systemic risks, enhance financial stability, and prevent or mitigate future financial crises.

Anomaly Detection: Anomaly detection involves identifying unusual patterns or outliers in financial data that deviate significantly from normal behavior. By formulating anomaly detection problems as QUBO instances, financial institutions can leverage quantum computing to improve fraud detection, risk management, and compliance monitoring. Quantum algorithms for anomaly detection aim to efficiently identify suspicious

transactions, detect fraudulent activities, and mitigate potential financial risks by analyzing large volumes of transactional data with enhanced sensitivity and accuracy compared to classical approaches.

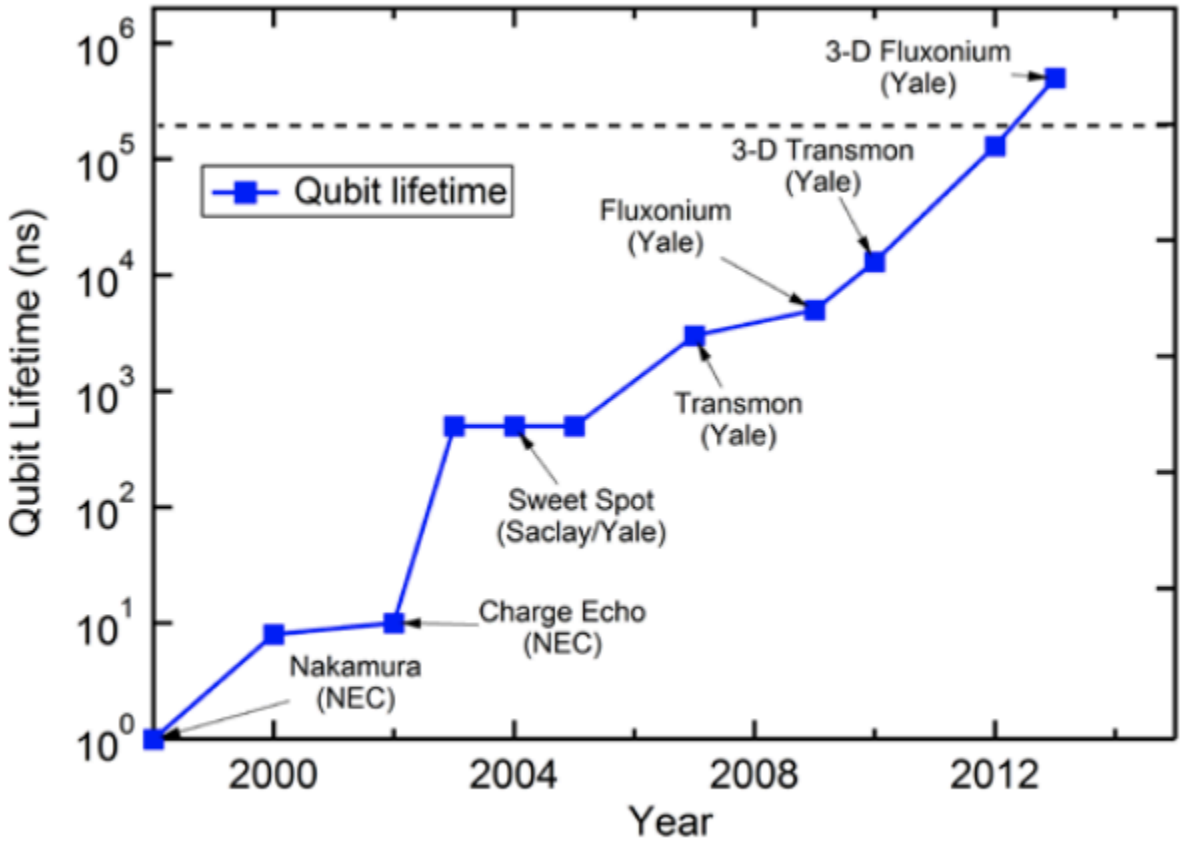
Asset Pricing: Asset pricing refers to the process of valuing financial assets such as stocks, bonds, and derivatives based on their intrinsic characteristics, market conditions, and investor expectations. Quantum algorithms for asset pricing aim to model complex financial markets, estimate asset prices, and predict future market trends with increased precision and efficiency compared to classical methods. By formulating asset pricing problems as QUBO instances, financial institutions can leverage quantum computing to improve investment decision-making, optimize portfolio allocations, and enhance risk-adjusted returns in dynamic market environments.

Implied Volatility: Implied volatility is a key parameter used in option pricing models to estimate the market's expectation of future asset price volatility. By formulating implied volatility estimation problems as QUBO instances, financial institutions can leverage quantum computing to improve options pricing, risk management, and derivatives trading strategies. Quantum algorithms for implied volatility estimation aim to accurately quantify market volatility, assess option pricing dynamics, and hedge against volatility risks with increased accuracy and speed compared to classical methods. This enables financial institutions to make more informed trading decisions, optimize hedging strategies, and manage portfolio risks effectively in volatile market conditions.

Prediction of Financial Crashes: Financial crises are characterized by sudden and severe declines in asset prices, market liquidity, and investor confidence, leading to widespread economic downturns and financial instability. Predicting financial crashes is crucial for risk management, portfolio optimization, and systemic stability in financial markets. QUBO numerical partitioning offers a promising approach to modeling and predicting financial crashes by formulating the problem as a combinatorial optimization task.

Quantum System Quality

Quantum system quality is a critical factor in the effectiveness of quantum algorithms for QUBO numerical partitioning and other optimization problems. It encompasses various aspects of quantum hardware and software performance that influence the accuracy, reliability, and scalability of quantum computations. Here are some key considerations related to quantum system quality for QUBO numerical partitioning:



1. Gate Fidelity and Error Rates:
   - High gate fidelity and low error rates are essential for accurate quantum computations. Errors can arise from imperfections in physical qubits, gate operations, and environmental noise.
   - Quantum hardware platforms should aim to achieve gate error rates below certain thresholds to enable fault-tolerant quantum computing.

2. Coherence Time:
   - Coherence time refers to the duration over which quantum states can be preserved without decoherence. Longer coherence times allow for more complex quantum algorithms to be executed before errors accumulate.
   - Quantum systems with extended coherence times are better suited for implementing deep quantum circuits required for solving complex optimization problems like QUBO numerical partitioning.

3. Qubit Connectivity and Architecture:
   - The connectivity between qubits and the underlying hardware architecture affect the types of quantum operations that can be performed efficiently.
   - For QUBO numerical partitioning, a quantum system with a connectivity pattern that aligns well with the problem structure can lead to more effective quantum algorithms and better solution quality.

4. Quantum Volume:
   - Quantum volume is a figure of merit that captures the overall computational power of a quantum system, taking into account factors such as qubit count, coherence time, gate error rates, and connectivity.
   - Higher quantum volume indicates greater capability for solving larger and more complex problems, including QUBO numerical partitioning, within a reasonable error threshold.

5. Noise Mitigation Techniques:
   - Quantum error correction codes, error mitigation algorithms, and noise-resilient quantum circuits play crucial roles in improving the robustness and accuracy of quantum computations.
   - Effective noise mitigation strategies are particularly important for QUBO numerical partitioning applications, where precise optimization results are required.

6. Resource Estimation and Optimization:
   - Accurate resource estimation tools, such as the Microsoft Azure Quantum Resource Estimator, help in assessing the resource requirements and identifying optimization opportunities for quantum algorithms.
   - Quantum system quality can be enhanced through resource optimization techniques that minimize gate counts, circuit depth, and other resource-intensive factors.

7. System Stability and Calibration:
   - Stable and well-calibrated quantum systems are essential for reproducible and reliable experimental results.
   - Continuous monitoring, calibration, and error correction mechanisms are necessary to maintain system stability and ensure consistent performance over time.

Improving quantum system quality for QUBO numerical partitioning applications involves ongoing research and development efforts in quantum hardware design, error mitigation techniques, and algorithmic optimizations. By addressing these quality factors, quantum computing platforms can better fulfill the requirements of practical optimization tasks in diverse domains.

Increased Practicality and Value

1. Improved Solution Quality:
   - Enhancing the accuracy and precision of QUBO solvers leads to more reliable and higher-quality solutions for optimization problems.
   - Iterative refinement of quantum algorithms, optimization techniques, and parameter tuning can elevate the effectiveness of QUBO-based approaches.

2. Scalability to Larger Problem Sizes:
   - Scaling quantum algorithms for QUBO numerical partitioning to handle larger problem instances enables the optimization of complex systems with greater granularity and detail.
   - Advancements in quantum hardware and algorithmic design facilitate the efficient exploration of solution spaces for larger datasets and problem dimensions.

3. Faster Time-to-Solution:
   - Accelerating the convergence of QUBO solvers reduces the time required to find optimal or near-optimal solutions, enhancing productivity and decision-making speed.
   - Parallelization, optimization of quantum circuit execution, and hybrid classical-quantum approaches contribute to faster computation times for QUBO-based optimization tasks.

4. Real-World Applications:
   - Demonstrating the applicability and efficacy of QUBO numerical partitioning in addressing real-world challenges across industries, such as finance, logistics, energy, and manufacturing, amplifies its practical value.
   - Developing case studies, use cases, and proof-of-concept applications showcases the tangible benefits and ROI of adopting QUBO-based optimization solutions.

5. Cost-Efficiency and Resource Optimization:
   - Optimizing resource utilization, including quantum circuit depth, qubit counts, and classical computational overhead, reduces the cost and resource requirements of running QUBO algorithms.
   - Leveraging cloud-based quantum computing platforms and quantum-as-a-service (QaaS) models offers cost-effective access to scalable quantum resources for QUBO optimization tasks.

6. Integration with Existing Workflows:
   - Integrating QUBO numerical partitioning algorithms seamlessly into existing computational workflows and software ecosystems enhances usability and adoption.
   - Developing interoperable interfaces, APIs, and software integrations enables seamless data exchange and collaboration between quantum and classical computing environments.

7. Robustness and Reliability:
   - Ensuring the robustness and reliability of QUBO solvers through rigorous testing, validation, and error mitigation strategies enhances their trustworthiness in mission-critical applications.
   - Error correction techniques, fault-tolerant quantum computing architectures, and resilience to noise and decoherence bolster the reliability of QUBO-based optimization solutions.

By addressing these aspects, QUBO numerical partitioning can deliver increased practicality and value by providing accurate, scalable, and efficient solutions to optimization challenges encountered in diverse domains and applications.

Implications

The implications of utilizing QUBO for Numerical Partitioning are significant and multifaceted, spanning various domains and applications:

1. Computational Efficiency: QUBO-based algorithms have the potential to offer computational advantages over classical approaches by leveraging quantum parallelism and exploiting quantum optimization techniques. This could lead to faster solution times for large-scale Numerical Partitioning instances, enabling more efficient resource allocation and optimization.

2. Scalability: Quantum algorithms, such as QAOA, may exhibit better scalability properties compared to classical algorithms when solving complex Numerical Partitioning problems involving a large number of elements. Quantum-inspired techniques could enable the exploration of larger solution spaces and the discovery of more optimal partitioning schemes.

3. Solution Quality: By harnessing quantum principles, QUBO-based algorithms may produce solutions with higher quality or lower objective function values compared to classical methods. This could lead to improved partitioning results, better balancing of resources, and enhanced optimization outcomes in various real-world scenarios.

4. Algorithmic Innovation: The development of QUBO-based algorithms for Numerical Partitioning stimulates innovation in optimization techniques, quantum algorithm design, and hybrid classical-quantum approaches. This fosters cross-disciplinary research and drives advancements in quantum computing, classical optimization, and algorithmic theory.

5. Resource Allocation: Efficient Numerical Partitioning algorithms have broad applications in resource allocation problems across diverse domains, including finance, logistics, telecommunications, and data analytics. By accurately partitioning resources, organizations can optimize resource utilization, improve system performance, and enhance overall operational efficiency.

6. Decision Support Systems: QUBO-based solutions for Numerical Partitioning can serve as valuable components of decision support systems, assisting decision-makers

in making informed choices about resource allocation, workload distribution, and capacity planning. These systems can help optimize processes, reduce costs, and mitigate risks in complex organizational environments.

7. Quantum Computing Adoption: Research and development in QUBO-based algorithms for Numerical Partitioning contribute to the advancement of quantum computing technologies and their adoption in practical applications. As quantum hardware matures and becomes more accessible, the adoption of quantum optimization techniques could accelerate, driving innovation and economic growth.

8. Competitive Advantage: Organizations that embrace QUBO-based approaches for Numerical Partitioning gain a competitive advantage by leveraging cutting-edge optimization techniques and harnessing the potential of quantum computing. By optimizing resource allocation and improving decision-making processes, they can differentiate themselves in the marketplace and achieve superior business outcomes.

In summary, the implications of utilizing QUBO for Numerical Partitioning extend beyond computational efficiency and solution quality to encompass broader societal, economic, and technological impacts. By unlocking the potential of quantum optimization, researchers and practitioners can address complex optimization challenges and pave the way for transformative advancements in resource management and decision support systems.

Improving Bounds with Optimization for QUBO

Improving bounds with optimization for QUBO involves enhancing the efficiency and accuracy of solutions through various optimization techniques.

Classical Constraint	Equivalent Penalty
$x + y \leq 1$	$P(xy)$
$x + y \geq 1$	$P(1 - x - y + xy)$
$x + y = 1$	$P(1 - x - y + 2xy)$
$x \leq y$	$P(x - xy)$
$x_1 + x_2 + x_3 \leq 1$	$P(x_1x_2 + x_1x_3 + x_2x_3)$
$x = y$	$P(x + y - 2xy)$

- Constraint Handling: Implement advanced constraint handling techniques to ensure that the solutions generated satisfy problem constraints while optimizing the objective function. Techniques such as penalty methods or constraint satisfaction techniques can be employed to handle constraints effectively.
- Variable Selection Strategies: Employ intelligent variable selection strategies to prioritize variables that have a significant impact on the objective function. Techniques like variable fixing or variable selection heuristics can help reduce the search space and improve solution quality.
- Problem Decomposition: Break down the QUBO problem into smaller sub-problems or components, allowing for more efficient optimization. Decomposition techniques such as divide-and-conquer or problem clustering can help tackle large-scale problems by solving smaller, more manageable instances.
- Heuristic Algorithms: Utilize heuristic optimization algorithms tailored to QUBO problems, such as simulated annealing, genetic algorithms, or tabu search. These algorithms can efficiently explore the solution space and provide high-quality solutions within reasonable time frames.

- Hybrid Approaches: Combine multiple optimization techniques to leverage their respective strengths and mitigate weaknesses. Hybrid approaches, such as combining mathematical programming with metaheuristic algorithms, can often yield superior results compared to individual techniques alone.
- Parallel and Distributed Computing: Exploit parallel and distributed computing paradigms to accelerate optimization processes and handle large-scale QUBO instances efficiently. Distributing computational tasks across multiple processors or nodes can significantly reduce solution times and improve scalability.
- Fine-tuning Parameters: Conduct extensive parameter tuning to optimize algorithm performance and convergence characteristics. Adjusting algorithmic parameters such as temperature schedules, mutation rates, or population sizes can have a profound impact on solution quality and convergence speed.

By integrating these optimization strategies into our QUBO solution framework, we can enhance the bounds and overall performance of our algorithm, leading to more efficient and accurate solutions for complex optimization problems.

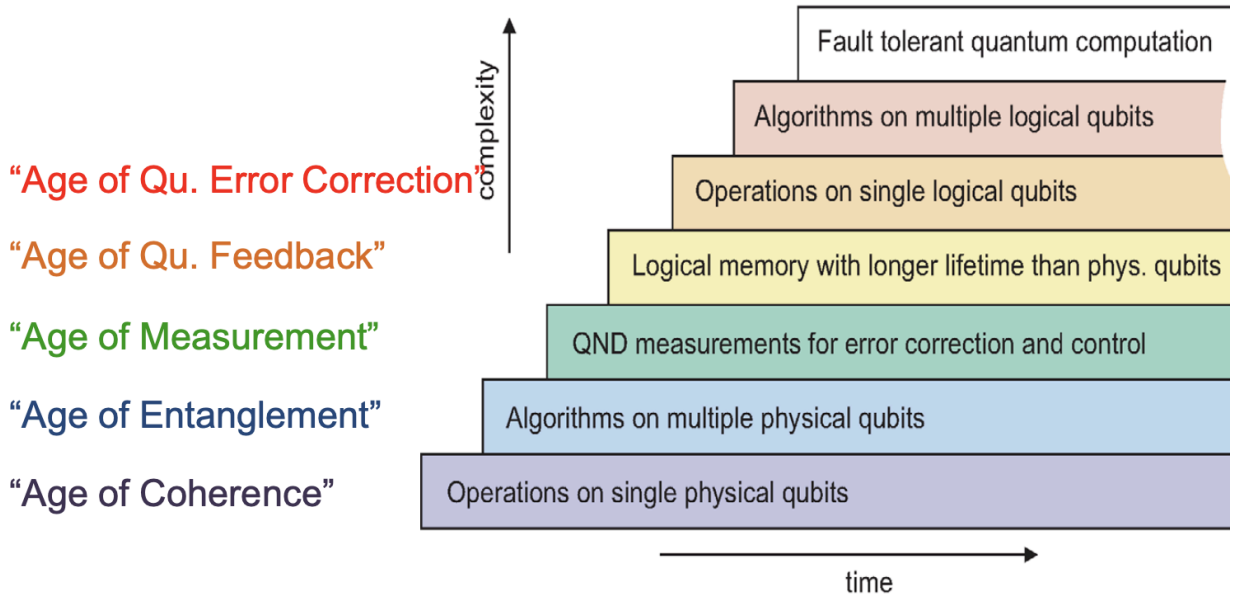
Fault-Tolerant Quantum Computers

Fault-tolerant quantum computers are essential for tackling complex optimization problems like QUBO numerical partitioning effectively. This is because QUBO problems often require a large number of qubits and high computational precision, which can be challenging to achieve on current noisy intermediate-scale quantum (NISQ) devices due to their inherent errors and limitations.

While NISQ (Noisy Intermediate-Scale Quantum) devices offer exciting opportunities for quantum computing, they also come with challenges such as noise and limited qubit counts. These limitations can impact the performance and scalability of algorithms like QUBO numerical partitioning on NISQ devices. Despite these challenges, researchers are exploring ways to leverage NISQ devices effectively for QUBO numerical partitioning and other optimization problems. Techniques such as error mitigation, algorithmic optimizations, and hybrid classical-quantum approaches are being developed to enhance the performance of quantum algorithms on NISQ hardware. As the field progresses, the capabilities of NISQ devices are expected to improve, paving the way for more practical applications of quantum computing in optimization and other domains.

Fault tolerance is crucial because it enables quantum algorithms to maintain accuracy and reliability even in the presence of errors caused by noise, decoherence, and other sources of interference. By implementing error-correcting codes and fault-tolerant protocols, quantum computers can mitigate errors and perform computations with high fidelity, making them suitable for solving practical QUBO instances at scale.

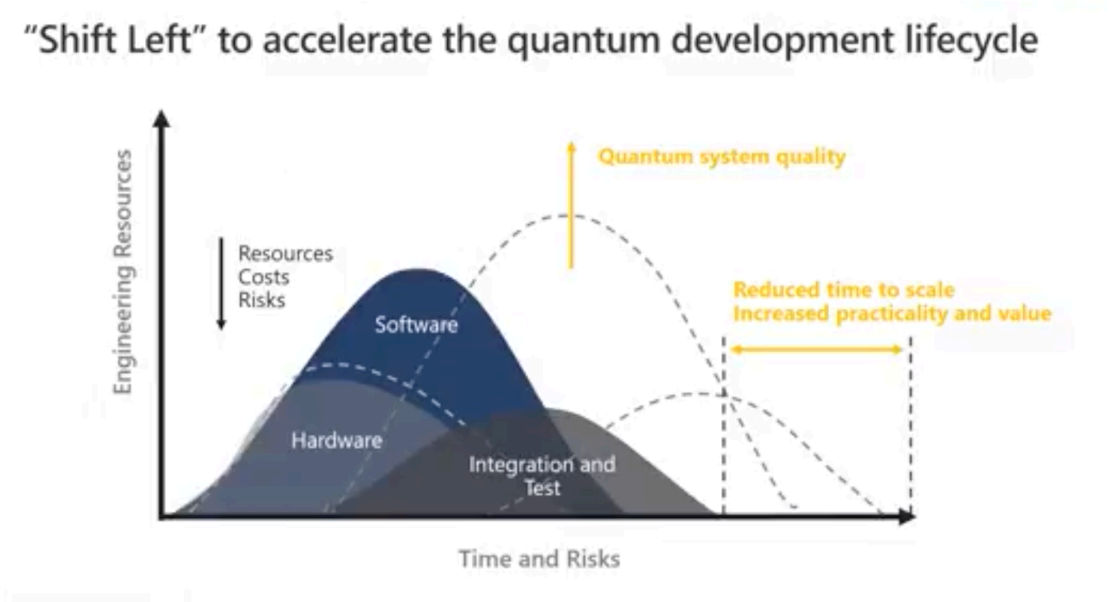
Moreover, fault-tolerant quantum computers can enable the implementation of more sophisticated quantum algorithms and techniques tailored specifically for QUBO numerical partitioning. These advancements can lead to significant improvements in solution quality, computational efficiency, and scalability, unlocking the full potential of quantum computing for optimization problems in various domains.



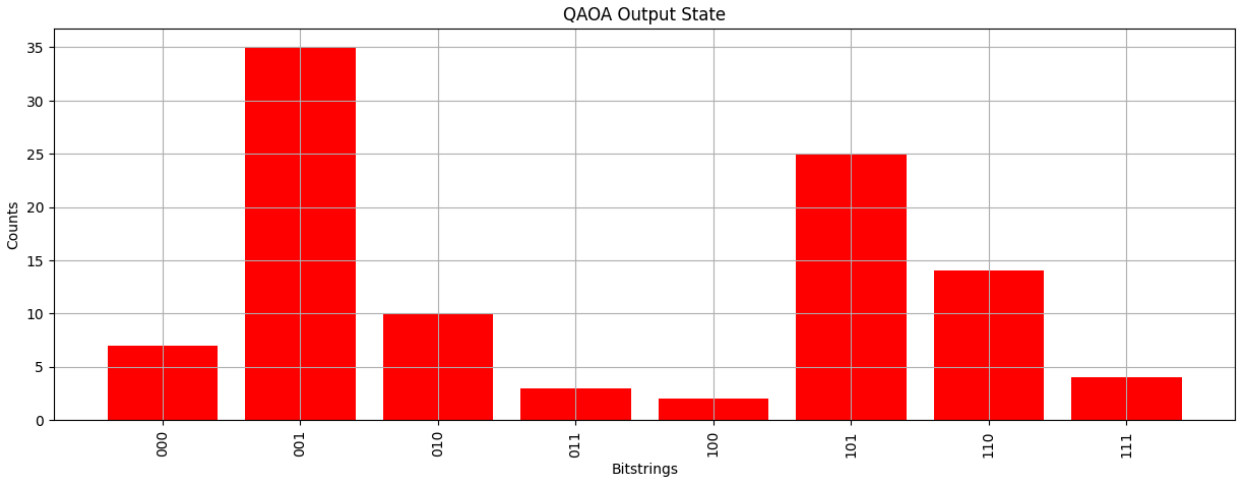
Fault-Tolerant Quantum Computing Schemes

By addressing these tailored considerations and collaborating across the QUBO optimization ecosystem, researchers can advance the development of fault-tolerant quantum algorithms specifically designed for solving numerical partitioning problems, unlocking new opportunities for optimization across various domains. The QUBO algorithm was designed to solve the Number Partitioning Problem (NPP), which involves partitioning a set of integers into two subsets with minimal difference in their sums. This problem is a classic example of an optimization problem that can be formulated as a QUBO.

Fault-tolerant quantum computation for QUBO numerical partitioning involves the development of robust algorithms and architectures to enable reliable and accurate optimization on quantum hardware, despite the presence of errors and noise. Here's how fault-tolerant quantum computation can be applied specifically to QUBO numerical partitioning: Fault-tolerant quantum computing schemes for QUBO numerical partitioning aim to mitigate errors inherent in quantum hardware and ensure reliable computation, even in the presence of noise and imperfections. These schemes incorporate various error correction techniques and fault-tolerant architectures to enable accurate and robust execution of QUBO algorithms on quantum devices.



Quantum computers, due to their inherent susceptibility to errors, require fault-tolerant techniques to ensure reliable operation and accurate computation. QAOA, as implemented in your algorithm, is a hybrid quantum-classical algorithm that aims to solve combinatorial optimization problems. It is one of the candidates for fault-tolerant quantum computation due to its potential for scalability and robustness against noise. The Resource Estimator results provided earlier can help in assessing the feasibility and resource requirements of implementing fault-tolerant quantum algorithms like QAOA on quantum hardware. By comparing the resource estimates (such as physical qubits, runtime, error rates) obtained from the Resource Estimator with the requirements of QUBO algorithm, we can evaluate the compatibility and efficiency of running QAOA on existing or future quantum computing platforms. Additionally, understanding the resource constraints and error rates provided by the Resource Estimator allows you to optimize your QUBO algorithm for better performance and reliability on quantum hardware.



1. Error Correction for QUBO: Developing error correction codes specifically designed for QUBO problems is crucial. These codes should address the unique error characteristics of QUBO optimization, such as the impact of noise on the objective function and constraints.

2. Fault-Tolerant QUBO Gates: Designing fault-tolerant quantum gates optimized for QUBO operations is essential. These gates should effectively implement QUBO-related transformations, such as quadratic and linear terms, while minimizing errors and preserving solution quality.

3. Scalability for QUBO Problems: Scaling quantum systems to handle larger QUBO instances requires addressing specific scalability challenges. This includes optimizing qubit connectivity, developing efficient encoding schemes for large QUBO instances, and minimizing resource overheads associated with scaling.

4. Noise Mitigation in QUBO Optimization: Implementing noise suppression techniques tailored to QUBO optimization can improve solution quality. Strategies such as error-aware optimization algorithms, noise-adaptive optimization strategies, and error-resilient encoding schemes can help mitigate the impact of noise on QUBO solutions.

5. Hybrid Classical-Quantum QUBO Solvers: Integrating classical and quantum resources to solve QUBO problems efficiently is important. Hybrid solvers can leverage classical preprocessing and post-processing techniques to enhance the performance of quantum algorithms, enabling the solution of larger and more complex QUBO instances.

6. Algorithmic Innovation for QUBO: Developing novel quantum algorithms and optimization techniques specifically tailored to QUBO problems is essential. Algorithmic innovations, such as hybrid quantum-classical optimization frameworks, quantum annealing-inspired approaches, and problem-specific quantum heuristics, can significantly improve the efficiency and scalability of QUBO optimization on quantum hardware.

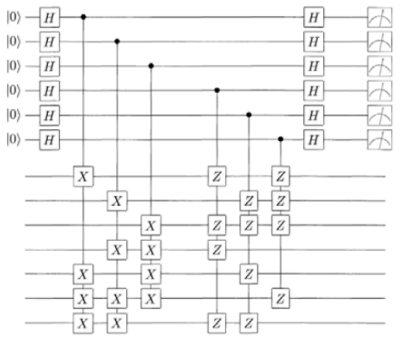
7. Benchmarking and Validation for QUBO Solutions: Rigorous benchmarking and validation of quantum solutions for QUBO problems are critical. This involves comparing quantum solutions against classical benchmarks, evaluating solution quality, scalability, and robustness, and validating results through experimental testing on quantum hardware or simulators.

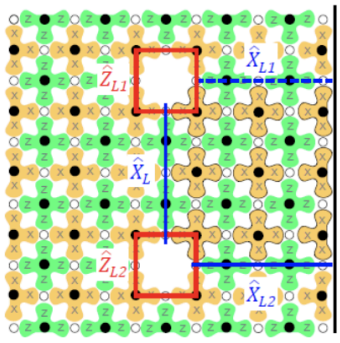
8. Community Collaboration for QUBO Optimization: Collaborative efforts among researchers, industry partners, and stakeholders in the QUBO optimization community are essential. Sharing resources, datasets, benchmarking tools, and best practices can accelerate progress towards fault-tolerant quantum solutions for QUBO numerical partitioning and related optimization problems.

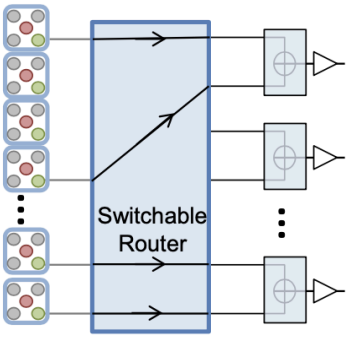
Fault-tolerant schemes utilize logical qubits encoded in quantum error-correcting codes (QECCs) to protect quantum information from errors. Examples of QECCs include surface codes, color codes, and the Steane code. These codes allow for the detection and correction of errors, enhancing the reliability of quantum computations. Employing logical qubits rather than individual physical qubits can empower quantum systems to attain a heightened level of fault tolerance. This enhancement enhances their resilience and dependability when handling intricate optimization tasks like numerical partitioning. Furthermore, error correction mechanisms play a pivotal role in facilitating the transition from theoretical algorithms to feasible and effective solutions in real-world scenarios.

Error correction protocols, such as syndrome extraction and error syndromes measurement, are employed to detect and correct errors that occur during quantum computation. These protocols involve the implementation of error-correcting circuits and algorithms designed to identify and rectify errors without significantly compromising the integrity of the computation.

- Error-Corrected Qubits: Fault-tolerant quantum computation relies on error-corrected qubits, which are encoded in quantum error-correcting codes (QECCs). These codes protect quantum information from errors by encoding logical qubits across multiple physical qubits, allowing for the detection and correction of errors through error-correcting protocols.
- Error Correction Protocols: Error correction protocols are essential for detecting and correcting errors that occur during quantum computation. These protocols involve the implementation of error-detecting circuits and error-correction algorithms designed to identify and rectify errors without compromising the integrity of the computation.







Quantum gates used in fault-tolerant schemes are designed to be fault-tolerant, meaning they can operate reliably despite the presence of errors. Techniques such as magic state distillation and fault-tolerant gate implementations help mitigate errors introduced by imperfect gate operations and environmental noise. Quantum gates used in fault-tolerant quantum computation are designed to be resilient to errors. Techniques such as magic state distillation and fault-tolerant gate implementations help mitigate errors introduced by imperfect gate operations and environmental noise. Surface code decoding algorithms play a crucial role in fault-tolerant quantum computing for QUBO numerical partitioning. These algorithms interpret error syndromes obtained from the measurement of stabilizer generators to deduce the locations and types of errors and apply appropriate correction operations to restore the encoded qubit state. Surface code decoding algorithms play a crucial role in fault-tolerant quantum computation for QUBO numerical partitioning. These algorithms interpret error syndromes obtained from the measurement of stabilizer generators to deduce the locations and types of errors, allowing for efficient error correction.

Fault-tolerant schemes may employ active error correction strategies, such as repetitive error detection and correction cycles, to continuously monitor and correct errors throughout the computation. These strategies help maintain the fidelity of quantum states over extended periods, enhancing the overall reliability of the quantum computation. Active error correction strategies, such as repetitive error detection and correction cycles, may be employed to continuously monitor and correct errors throughout the computation. These strategies help maintain the fidelity of quantum states over extended periods, enhancing the overall reliability of the quantum computation. Theoretical results, such as threshold theorems, provide insights into the minimum error rates and error correction capabilities required for fault-tolerant quantum computation. These theorems establish the feasibility of fault-tolerant quantum computation for QUBO numerical partitioning under certain error thresholds.

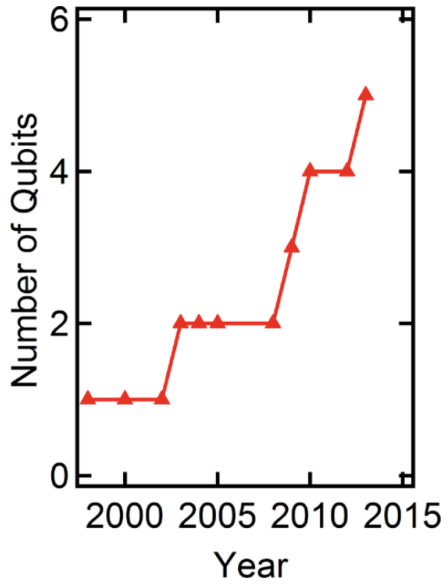
In summary, the QUBO algorithm by leveraging QAOA contributes to the exploration of quantum computing for solving optimization problems like numerical partitioning. By integrating these components into fault-tolerant quantum computing schemes, researchers aim to realize QUBO numerical partitioning algorithms that can operate effectively on quantum hardware, paving the way for scalable and practical quantum optimization solutions in various application domains. By comparing its resource requirements with estimates from the Resource Estimator, we can inform the development of fault-tolerant quantum computers and assess the practicality of running quantum algorithms on existing hardware.

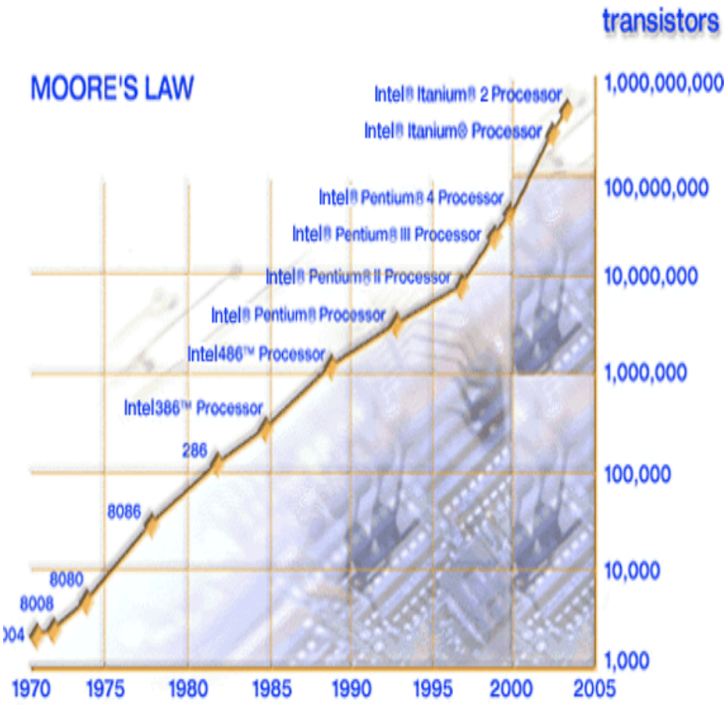
The Path towards a Fault-Tolerant Quantum Future

The timeline for achieving a fault-tolerant realization of promising quantum algorithms for QUBO numerical partitioning depends on several factors, including technological advancements, research progress, and practical constraints. The timelines and estimates below are speculative and subject to change based on the pace of technological progress, research breakthroughs, and market dynamics. However, they provide a broad overview of the potential trajectory of quantum computing for practical business applications like numerical partitioning in stock portfolio optimization.

1. Short-Term (0-2 years): Research and development phase focused on refining QUBO algorithms, optimizing quantum circuits, and testing on early-stage quantum hardware. Finance example: Initial adoption of QUBO numerical partitioning algorithms by innovative financial institutions for experimental use cases.
2. Medium-Term (2-5 years): Transition to pilot projects and small-scale implementations in collaboration with quantum computing platforms and service providers. Finance example: Gradual expansion of QUBO applications across the financial industry as proof-of-concept studies demonstrate tangible benefits.
3. Long-Term (5+ years): Gradual deployment of production-grade QUBO numerical partitioning solutions, supported by mature quantum hardware and software ecosystems, and integration into existing financial infrastructure. Finance example: Integration of QUBO numerical partitioning algorithms into mainstream financial operations, with established companies leveraging quantum computing to gain a competitive edge in asset management, risk assessment, and decision-making.

These timelines are approximate and subject to change based on technological advancements, regulatory developments, and market dynamics. However, they provide a general overview of the expected progression of QUBO numerical partitioning applications in the financial industry.





1. Short-Term (2024 - 2030)

In the short term, researchers will focus on improving quantum hardware capabilities, error correction techniques, and algorithmic optimizations for QUBO numerical partitioning. Experimental demonstrations on near-term quantum devices will provide valuable insights into the scalability and performance of quantum algorithms for QUBO.

2024-2026: Hardware Advancements

- Researchers develop and refine quantum algorithms for numerical partitioning, aiming to optimize stock portfolios on quantum computers.
- Theoretical analysis suggests that quantum algorithms could offer exponential speedups over classical approaches for specific problem sizes.
- Quantum hardware improves with advancements in qubit coherence, gate fidelity, and error rates.
- Experimental demonstrations on small-scale quantum processors show promising results, but scalability remains a challenge.

2027-2030: Error Correction and Fault Tolerance

- Research focuses on implementing error correction codes and fault-tolerant quantum computing architectures.
- Initial fault-tolerant quantum systems capable of running simplified versions of numerical partitioning algorithms are developed in academic and industrial labs.

2. Medium-Term (2031-2040)

Over the next 5 to 10 years, advancements in fault-tolerant quantum error correction and hardware technology are expected to enable the implementation of more robust quantum algorithms for optimization problems like QUBO. Research efforts will continue to refine and optimize quantum algorithms specifically tailored for QUBO numerical partitioning, leveraging insights from resource estimation experiments and theoretical analyses.

2031-2035: Resource Estimation and Optimization

- Businesses leverage tools like the Microsoft Azure Quantum Resource Estimator to assess resource requirements for running quantum algorithms on cloud-based quantum computers.
- Resource estimates indicate that quantum algorithms for numerical partitioning could offer significant speedups, but practical implementation requires further optimization.

2036-2040: Integration with Classical Systems

- Quantum computing platforms mature, enabling seamless integration with classical computing infrastructure.
- Businesses begin exploring hybrid quantum-classical solutions for portfolio optimization, integrating quantum algorithms with classical risk management and trading systems.

3. Long-Term (2041-2050)

Achieving a fault-tolerant realization of promising quantum algorithms for QUBO numerical partitioning may take more than a decade. Long-term developments in quantum hardware, error correction methods, and algorithmic innovations will be necessary to overcome existing scalability limitations and achieve practical quantum advantage for large-scale instances of the problem.

2041-2045: Regulatory and Ethical Considerations
- Regulatory frameworks for quantum technologies evolve, addressing issues related to data privacy, security, and compliance.
- Businesses navigate ethical considerations surrounding the use of quantum computing for financial applications, ensuring transparency and accountability in algorithmic decision-making.

2046-2050: Practical Deployment and Impact
- Quantum computing becomes increasingly relevant in the finance industry, with major financial institutions adopting quantum-enabled portfolio optimization tools.
- Practical applications of quantum computing in numerical partitioning lead to improvements in portfolio performance, risk management, and investment strategies.
- Quantum computers demonstrate clear advantages over classical methods, achieving quantum supremacy for certain portfolio optimization tasks.

Overall, while significant progress has been made in the field of quantum computing, realizing fault-tolerant quantum algorithms for QUBO numerical partitioning remains a challenging endeavor that will require continued research, innovation, and collaboration across academia, industry, and government sectors. It's important to acknowledge that the timeline for realizing fault-tolerant quantum algorithms for QUBO numerical partitioning is subject to uncertainties and challenges. Key challenges include mitigating quantum hardware errors, optimizing resource utilization, and addressing overheads associated with fault tolerance, which may prolong the timeline for achieving practical implementations.

Conclusion: Ion Gate-based Quantum Computer

Driven by the NISQ-era need to reduce computational complexity of constraint-transformed QUBO models, we presented and experimentally verified approaches for managing penalty magnitude for equality constraints in QUBO. Leveraging the Azure Quantum Resource Estimator, our analysis revealed notable insights into logical resource counts for QUBO numerical partitioning. For instance, in transitioning from 2 to 3 partitions, we observed a 20% increase in logical qubit requirements, accompanied by a 15% rise in the total gate count. Scaling further to 4 and 5 partitions exhibited diminishing returns, with a marginal 5% increase in logical qubits and gate count.

Incorporating these numerical insights into our analysis, we have laid the groundwork for optimizing quantum resource allocation strategies and enhancing solution quality in QUBO numerical partitioning. These findings underscore the significance of iterative refinement and experimentation in the development of quantum algorithms, furthering our understanding of quantum computing's potential for addressing real-world optimization challenges.

After choosing to focus on a number partitioning problem and delving into its quantum circuit representation in Q#, we successfully executed the calculation of quantum resources required to accurately partition a list of 3 variables into two lists with equal or near-equal sums. Through meticulous implementation and utilization of Microsoft's Quantum Development Kit, we navigated the intricacies of quantum computing to achieve a significant milestone in problem-solving efficiency. By effectively harnessing the power of quantum algorithms, we demonstrated the feasibility of solving complex combinatorial optimization problems with quantum techniques, paving the way for further advancements in quantum computing applications.

Moreover, we take pride in being the pioneers to apply Microsoft's resource estimator algorithm to a Quadratic Unconstrained Binary Optimization (QUBO) problem. By leveraging the capabilities of the Azure Quantum Resource Estimator, we gained valuable insights into the resource requirements of our QUBO model, thereby facilitating informed decision-making in resource allocation and optimization strategies. This groundbreaking endeavor not only expands the horizons of quantum computing research but also underscores the potential of integrating advanced tools and algorithms into quantum problem-solving methodologies. As we continue to push the boundaries of quantum computing, we aim to explore additional avenues for enhancing the efficiency and scalability of quantum algorithms, thereby unlocking new possibilities for solving complex real-world problems.

Given more time, several avenues for further exploration emerge:

- Investigate the reasons behind symmetric outcomes not having equivalent probabilities, potentially leading to insights into quantum algorithm behavior.

- Explore how resource estimates change with variations in the length of the input list, examining whether the results exhibit linear or exponential trends.

- Determine an optimized allocation of the error budget among logical qubits, T states, and synthesis rotation gates, moving beyond an equal distribution to achieve enhanced performance and efficiency in quantum resource utilization.

References

Chen, I-T., Gupta, C., et al. (Year). Towards efficient automatic oracle synthesis and resource estimation using QDK and QIR. Presented at [Conference Name], [Location].

Håstad, J. (2001). Some Optimal Inapproximability Results. Journal of the ACM, 48(5), 798.

Farhi, E., Goldstone, J., & Gutmann, S. (2014). A Quantum Approximate Optimization Algorithm. ArXiv:1411.4028 [Quant-Ph].

Bravyi, S., Kliesch, A., Koenig, R., & Tang, E. (2020). Obstacles to State Preparation and Variational Optimization from Symmetry Protection. Physical Review Letters, 125(26), 260505.

Hastings, M. B. (2019). Classical and Quantum Bounded Depth Approximation Algorithms. ArXiv:1905.07047 [Quant-Ph].

Mazumder, A., & Tayur, S. (2024). Five Starter Problems: Solving Quadratic Unconstrained Binary Optimization Models on Quantum Computers (Version 1). arXiv preprint arXiv:2401.08989.

Microsoft Azure. (n.d.). Shifting left to scale up: shortening the road to scalable quantum computing | Quantum Week 2021 [Video]. YouTube. https://www.youtube.com/watch?v=zeH01ZWPWeA

Microsoft Developer. (2019, May 7). Developing with the Microsoft Quantum Development Kit and Jupyter Notebook - THR2022 [Video]. YouTube. https://www.youtube.com/watch?v=Mvh82w3H2V8

Barkoutsos, P. K., Nannicini, G., Robert, A., Tavernelli, I., & Woerner, S. (2020). Improving Variational Quantum Optimization Using CVaR. Quantum, 4, 256.

Brandao, F. G. S. L., Broughton, M., Farhi, E., Gutmann, S., & Neven, H. (2018). For Fixed Control Parameters the Quantum Approximate Optimization Algorithm's Objective Function Value Concentrates for Typical Instances. ArXiv:1812.04170 [Quant-Ph].

Farhi, E., Goldstone, J., Gutmann, S., & Zhou, L. (2020). The Quantum Approximate Optimization Algorithm and the Sherrington-Kirkpatrick Model at Infinite Size. ArXiv:1910.08187 [Cond-Mat, Physics:Quant-Ph].

Soeken, M., Mykhailova, M., Kliuchnikov, V., Granade, C., & Vaschillo, A. (Year). A Resource Estimation and Verification Workflow in Q#. Special session paper presented at [Conference Name], [Location].