SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Real-time and parallel task scheduling for Quantum Computing

Marcin Praski

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Real-time and parallel task scheduling for Quantum Computing

Echtzeit- und parallele Aufgabenplanung für Quantencomputer

Author:	Marcin Praski
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Emmanouil Giortamis
Submission Date:	15.12.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.12.2024 Marcin Praski

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Pramod Bhatotia, for providing me an opportunity of exploring the fascinating area of distributed systems and more specifically, the realm of quantum scheduling.

I am also deeply grateful to my advisor, Emmanouil Giortamis, for guiding my progress, answering my questions about quantum computing, and exchanging invaluable ideas and research directions during my academic journey.

I would like to extend my gratitude to the team of Systems Research Group.

Abstract

In this thesis, we investigate the problem of scheduling jobs on quatum computing units at a scale. We identify the bottlenecks and unique challenges posed by the landscape of modern quantum compting, dubbed Noisy Intermediate-Scale Quantum (NISQ). Those observations lead us to formulate the design goals for our multi-objective quantum scheduler. The scheduler employs swappable optimizers chosen depending on the problem size. The quantities we aim to optimize include the makespan of the schedule and its average quantum fidelity.

Our work begins by defining the precise mathematical formulation of the scheduling problem, factoring in the constraints like estimated circuit runtime, fidelity and temporal dependencies. We then proceed to translate this model to distinct concrete optimization algorithms, namely the Mixed Integer Programming (MIP), Genetic Algorithm (GA) and Non-dominated Sorting GA. Those approaches are implemented, documented and evaluated for their scalability and solution quality.

The evaluation of the schedulers is performed on synthetically-generated datasets of circuits, expected runtimes, fidelities based on previous studies. Benchmarking reveals interesting insights into the performance and characteristics of the optimizers. We investigate how the tradeoff between conflicting objectives is handled by the algorithms, and how it affects the final results.

This thesis addresses the necessity for a scalable, transparent scheduling system for quantum cloud, designed to satisfy the growing demand for affordable quantum computing demands from both academia and industry. Our findings show promise, and we idendify multiple venues for further research and improvement of our solutions.

Zusammenfassung

In dieser Arbeit untersuchen wir das Problem der Planung von Aufträgen auf Quatum-Recheneinheiten auf einer Skala. Wir identifizieren die Engpässe und einzigartigen Herausforderungen, die sich durch die Landschaft des modernen Quantencomputer, genannt Noisy Intermediate-Scale Quantum (NISQ). Diese Beobachtungen führen uns zur Formulierung der Designziele für unseren multikriteriellen Quanten-Scheduler. Der Scheduler verwendet austauschbare Optimierer, die in Abhängigkeit von der Problemgröße ausgewählt werden. Die Größen die wir optimieren wollen, sind die Zeitspanne des Zeitplans und seine durchschnittliche Quantentreue.

Unsere Arbeit beginnt mit der Definition der genauen mathematischen Formulierung des Scheduling-Problems, Dabei werden Einschränkungen wie die geschätzte Laufzeit des Schaltkreises, die Zuverlässigkeit und zeitliche Abhängigkeiten berücksichtigt. Anschließend übersetzen wir dieses Modell in verschiedene konkrete Optimierungsalgorithmen, nämlich die Gemischt Ganzzahlige Programmierung (MIP), den Genetischen Algorithmus (GA) und das nicht-dominierte Sortierverfahren GA. Diese Ansätze werden implementiert, dokumentiert und hinsichtlich ihrer Skalierbarkeit und Lösungsqualität bewertet. Qualität.

Die Evaluierung der Scheduler erfolgt auf synthetisch erzeugten Datensätzen von Schaltungen, erwartete Laufzeiten und Treue auf der Grundlage früherer Studien. Das Benchmarking offenbart interessante Einblicke über die Leistung und die Eigenschaften der Optimierer. Wir untersuchen, wie der Kompromiss zwischen Zielkonflikten von den Algorithmen gehandhabt wird und wie sie die Endergebnisse beeinflussen.

Diese Arbeit befasst sich mit der Notwendigkeit eines skalierbaren, transparenten Scheduling-Systems für die Quanten-Cloud, um die wachsende Nachfrage nach erschwinglichen Quantencomputern sowohl in der Wissenschaft und Industrie. Unsere Ergebnisse sind vielversprechend, und wir zeigen mehrere Möglichkeiten für weitere Forschung und Verbesserung unserer Lösungen.

Contents

1 Introduction

Quantum computing has been steadily gaining traction as an enticing alternative to classical computing, opening new possibilities for solving problems that are intractable for classical computers. The ability to leverage quantum entanglement, superposition and interference offers tangible advantages in areas such as cryptography, particle simulation, optimisation and machine learning.

The current state of the art Quantum Processing Unit (QPU)s are functionally similar to Central Processing Unit (CPU)s, but operate on quantum bits (qubits) instead of classical ones, in the ranges of hundreds or thousands. The major challenge posed by quantum hardware design is the scalability of number of qubits - current QPUs are not yet large enough to solve landmark problems like Shor's Factoring or Grover Search, which require the order of millions of qubits to be solved efficiently. QPU scaling is hampered by quantum noise - decoherence of the qubits due to interactions with the environment, imperfections in quantum gates and lossy connections between individual qubits. These factors necessitate elaborate error correcting schemes, and give name to the current era of quantum computing - Noisy Intermediate-Scale Quantum (NISQ).

In recent years, academic and commercial interest in quantum computing in the cloud has been rapidly increasing. Industry leaders like IBM, Microsoft, Google or Honeywell have begun offering access to their growing quantum computing fleets via the familiar cloud provider interfaces and dedicated technologies like IBM Quantum [5] framework or Microsoft quantum programming language #*Q* [6]. These toolkits provide a high-level interface to quantum computers, allow users to write quantum programs with IDE support, while lower level operations like circuit synthesis, transpiling and scheduling on the actual QPU are implemented by the provider. As the quantum processors become more powerful and readily available for general use, the growth of quantum computing as a cloud service is expected to accelerate in the coming years.

This increased interest and availability of quantum computing resources necessitates a smart approach to scheduling the programs, or jobs, submitted by the customers of the cloud providers. The relative scarcity of quantum machines leads to 10s-1000s of user jobs awaiting in queues before they can be executed on QPUs. These bottlenecks lead to many hours of waiting time for the users from academia and industry, hampering their ability to iterate on their research and development. Worse still, the submitted programs need to be transpiled for the exact QPU they will run on for optimal performance, and it is a well-known fact that QPUs need to be recalibrated on a regular basis to account for quantum decoherence [7]. Thus, jobs sitting in the pipeline for too long may need to be re-transpiled for the new state of the QPU, adding a new, temporal aspect to the already complex scheduling problem.

Continuously growing demand for predictable quantum computing resources necessitates the development of a real-time, scalable scheduler aware of the unique challenges posed by quantum hardware. The scheduling system will act as an intermediary between the users and the cloud providers, accepting jobs and allocating them to QPUs in a way that maximises objectives we deem important, like the decreased waiting time and improved fidelity of the executed program. Our goal is to propose a design of such a scheduling system, evaluate it and consider future directions of development. The core contributions of this thesis are:

Design of a swappable-algorithm scheduler: In Chapter 3 we propose an open-ended design for a quantum scheduling system independent of the underlying optimization objectives and algorithms, thus perfect for quick iterations and scaling to accomodate for the problem size at hand.

Formulation of the Scheduling Problem: In Chapter 4 we formulate the requirements and constraints of our scheduling problem, expressing it as a multi-objective optimization task suitable for translation to many optimization methods.

Optimization Algorithm Exploration: We delve into two optimization techniques, the Mixed Integer (Linear) Programming (MIP) and Genetic Algorithm (GA), as the optimizers For our scheduling system. We discuss the benefits and drawbacks of each solution.

Optimization Algorithm Implementation: In Chapter 5 we present the reference implementations of both optimizers, discuss possible parallelization of the computations and discuss the tradeoff between the two methods.

Scheduler Evaluation: In Chapter 6 we evaluate the performance of both optimizers at the level of micro-benchmarks and end-to-end tests. We also discuss the quality of solutions obtained via each method, and the possible tradeoffs between them and the time/resources required to obtain them.

2 Background

This outlines foundational concepts crucial to understanding the context of this thesis. It covers key concepts in quantum computing, classical job scheduling, multi-objective optimization, linear programming and genetic algorithms.

2.1 Quantum Computing

2.1.1 Foundations

To understand the fundamentals of quantum computing, we briefly describe core principles that distinguish it from classical computing. Two key concepts, quantum interference and entanglement [3], play a vital role in leveraging the computational capabilities of quantum systems:

- **Quantum interference** is a fundamental principle stemming from the wave-like behavior of quantum particles. It involves constructive or destructive combination of probability amplitudes associated with different quantum states. This enables quantum computers to process multiple possibilities simultaneously, exponentially enhancing their computational power compared to classical systems.
- **Entanglement** describes a phenomenon where quantum particles become intrinsically correlated, sharing a joint quantum state. When one entangled particle's state changes, the other's state is instantaneously affected, regardless of the distance between them. This unique property allows quantum computers to establish highly interconnected states, enabling phenomena like quantum teleportation and error correction schemes.

2.1.2 Quantum Circuits

Quantum circuits are the equivalent of classical computer programs, enabling the execution of quantum algorithms on QPUs. Understanding the complexity of quantum computation requires delving into the fundamental components of quantum circuits, which consist of qubits, gates, and measurements.

At the core of quantum circuits are quantum bits, or qubits, which serve as the quantum counterpart to classical bits. Unlike classical bits, which can only exist in binary states, qubits can exist in a superposition of basis states simultaneously. This unique characteristic is fundamental to quantum computing, enabling parallel information processing and significantly expanding the range of computational possibilities.

Quantum gates, analogous to classical logic gates, operate on qubits to perform specific tasks. These gates include single-qubit gates, which manipulate state of individual qubits. Multiple qubit gates allow for operations on multiple qubits simultaneously, which forms a basis for more complex algorithms. Among the variety of quantum gates, several are commonly distinguished:

- **Pauli-X Gate (X)**: Similar to a classical NOT gate, it flips a qubit's state between the basis states.
- **Hadamard Gate (H)**: Places a qubit in a superposition of the two basis states.
- **CNOT Gate (Controlled-NOT)**: Flips the target qubit's state only if the control qubit is in basis state 1.
- **SWAP Gate**: Interchanges the states of two qubits.
- **CCNOT Gate (Controlled-Controlled-NOT)**: Like the CNOT gate, except it requires two control qubits rather than one.

Quantum circuits composed out of the above gates and more eventually have their qubits measured, which triggers a collapse of the quantum superposition to classical information. Measuring a qubit results in either a 0 or 1, based on the probabilities defined by its quantum state. These measurements are the only way to extract meaningful information from quantum computations, bridging the gap between quantum and classical data.

2.1.3 Quantum Processing Units

The current generation of quantum processing units (QPUs) is referred to as NISQ. This designation indicates that there are inherent imperfections in the gate construction and inter-qubit links resulting in the "noise" of measurements, thus requiring QPU designers to implement error correction schemes, repeat experiments to measure statistical moments of the noise, and periodically calibrate the hardware to minimize the rate of such errors.

Fault-tolerant quantum computation relies on implementing aforemetnioned error-correction codes, which require a number of physical qubits to construct a single logical qubit free from errors. However, current quantum computers lack the required number of qubits to completely eliminate errors and achieve full fault tolerance.

Recent advancements in quantum computing have led to systems in range of 1,000s of qubits. The landscape of quantum devices is highly diverse, with each system exhibiting unique characteristics. These include the number of qubits, qubit layout, gate error rates, measurement error rates, gate lengths, and other parameters that collectively define the performance of a QPU.

The number of qubits and their connectivity play a vital role in determining a quantum computer's computational capabilities. The layout of a QPU, referred to as its topology, dictates which qubits can directly interact, a crucial factor in designing and optimizing quantum algorithms for specific hardware.

Additionally, error rates associated with quantum gates and measurements vary significantly across quantum processors, directly impacting the reliability and precision of quantum computations. Gate length, which measures the duration of quantum operations, further contributes to the overall performance of a QPU.

It is important to note that the properties of quantum backends are dynamic, as quantum computers require frequent recalibration due to variations in qubit properties, environmental factors, and other conditions. As a result, the characteristics of a QPU, including error rates and qubit behavior, can fluctuate significantly over time.

2.1.4 Execution Pipeline

Analogous to programming for classical computing, a quantum program needs to be compiled and optimized for execution on a specific QPU. This process transforms of a quantum program from a high-level representation, like the *Q#* language, into a series of quantum circuits that can be scheduled for execution on the target QPU.

The need for transpilation arises from the architectural diversity of quantum processors. Different QPUs feature distinct qubit layouts, connectivity constraints, and gate sets, meaning a circuit designed for one device may not be directly executable on another.

The transpilation process typically includes the following steps:

- **Qubit Remapping:** Adjusting of the mapping of logical qubits from the circuit to the physical qubits available on the quantum device. This step is essential as the connectivity between qubits often varies across quantum processors.
- **Gate Decomposition:** Breaking down complex quantum gates into a sequence of simpler gates that are natively supported by the target QPU. Since many quantum devices cannot directly implement all types of gates, this decomposition ensures compatibility.

- **Optimization:** Streamlining the circuit to enhance its efficiency and reduce the total gate count. This step minimizes the effects of noise and reduces errors during execution.

Transpilation is a critical aspect of the quantum programming workflow, enabling quantum algorithms written in a hardware-agnostic manner to be executed on specific quantum devices. This flexibility allows researchers and developers to focus on algorithmic logic without being constrained by hardware-specific requirements.

Once transpilation is complete, the quantum circuit is ready for execution on the QPU. However, due to the inherent noise in quantum hardware, a single execution, also called a shot, is insufficient to produce meaningful results. Quantum circuits are typically executed thousands of times to mitigate the probabilistic nature of quantum measurements, so that we can extract trends and patterns from the aggregated outcomes.

A key metric for assessing the performance of quantum circuits is execution fidelity, which measures how closely the actual results of a circuit align with its expected outcomes. High fidelity is indicative of well-designed circuits that effectively account for the impact of noise and hardware limitations.

2.1.5 Quantum Cloud

Cloud platforms such as IBM, Microsoft Azure, Google Cloud and AWS increasibgly introduce quantum compute-as-a-service to the general audience. This trend democratizes quantum computing by making it accessible to a broader audience and enabling users to leverage quantum processing for various applications, from protein folding to operations research.

Users of quantum platforms like IBM Quantum submit their programs via the online dashboard and have the option to choose a specific QPU for the execution. The quantum program then undergoes a transpilation phase for the architecture of the chosen quantum device. The transpiled circuits are then executed on the QPU, and users are charged on a pay-as-you-go basis, based on the actual quantum resource usage time.

This model eliminates the need for on-site quantum computing infrastructure, enabling seamless access to quantum computing resources. In the fast evolving field of quantum computing, this approach ensures that users can access the latest quantum devices, thereby supporting innovation and facilitating advancements in research.

2.2 Scheduling

2.2.1 Problem Domain

We consider two primary categories of scheduling, as defined in [12]:

- **Resource allocation** involves assignment of physical resources such as compute and memory to cloud tasks in a way that utilization of these resources is maximized. This is also known as vertical autoscaling in e.g. Kubernetes.
- **Task scheduling** involves placing cloud tasks on available resources without provisioning or deprovisioning them. The placement needs to balance individual needs of the tasks, e.g. GPU availability, with global objectives like maximizing resource usage.

The focus area of this paper is the second category. More specifically, we will explore the methods to produce an optimal schedule based on a number of criteria. These criteria, which in our case will include goals like minimized makespan of the schedule, maximized average fidelity of executed circuits, are often completing and a tradeoff needs to be identified. This is known as multi-objective optimization.

2.2.2 Objective Scalarization

In order to quantify the quality of a solution to a multi-objective optimization problem, we need to be able to tell which solution is "better" than the other. Some methods we'll explore in following chapters rely on scalarization of the multiple objectives into a single, usually numeric value. This may be done by summing the objectives in a weighted manner: $M = \sum_{i=1}^{I} w_i m_i$. This method may work well for some problems, however it requires a-prori knowledge of the preferences between objectives, and scaling to accomodate for the different units across objectives.

2.2.3 Pareto Dominance Order

An alternative method of comparing multi-objective solutions is the Pareto dominance order. We define it as a partial order on the set of possible solutions $\mathcal{Q}$ and the following relation: Let x_1 and x_2 be two solutions, and let $f(x) = [f_1(x), f_2(x), \ldots, f_k(x)]$ denote the vector of objective function values. $x_1 \preceq x_2$, which reads "x_1 dominates x_2", is defined as:

$$x_1 \preceq x_2 \iff \forall i \in \{1, 2, \ldots, k\}\ f_i(x_1) \leq f_i(x_2)\ \land\ \exists j \in \{1, 2, \ldots, k\}\ f_j(x_1) < f_j(x_2)$$

meaning x_1 dominates x_2 if it is at least as good in all objectives and strictly better in at least one. This partial order allows us to partition the set of all solutions into subsets called Pareto fronts, where a member of the front is not dominated by any other element in this set. Given a subset $\mathcal{P} \subseteq \mathcal{Q}$, we define the Pareto front as [23]:

$$\mathcal{PF} = \{x \in \mathcal{P} \mid \nexists\, y \in \mathcal{P} \text{ s.t. } f(y) \preceq f(x)\}$$

We can generate successive fronts by progressively removing the dominant solutions from $\mathcal{Q}$ and thus having a smaller $\mathcal{P}$ with each front. This technique will underpin some of the algorithms we'll explore, as it offers a precise definition of "as good as" solutions, and the ability to choose from them. This becomes handy when we need to balance the objectives in a way that is not possible with scalarization. Essentially this is a mathematical formulation of a compromise between objectives.

2.2.4 Adjacent Problems

The scheduling problem is a known area of research in computer science, and many distinct sub-problems have been solved over time. Authors of [9] propose a following classification:

- **Optimal job scheduling** focuses on determining the optimal sequence for processing a set of tasks on a single machine. The main objectives are typically to minimize the makespan and reduce the total flow time.
- **Parallel job scheduling** involves multiple machines available to execute tasks simultaneously, with each machine operating independently. This setup enables parallelization, and the optimization goals often include minimizing the makespan, reducing idle time and ensuring a balanced workload distribution across machines.
- **Job shop scheduling** is when tasks can be processed on multiple machines in any sequence, depending on specific requirements and constraints. The primary challenge is to determine the best order for executing tasks while considering these constraints. Objectives typically include minimizing the makespan, reducing idle time and achieving an even distribution of workloads.
- **Flow shop scheduling** involves a sequential processing system where tasks pass through a series of machines, each dedicated to a specific operation. The key challenge is to optimize the order in which tasks are processed across the machines. Common objectives include minimizing the makespan and ensuring a smooth flow of tasks through the system.

Quantum job scheduling shares most similarities with the parallel job scheduling problem, yet poses unique challenges such as inter-job order or strict job-to-QPU affinity requirements.

2.3 Linear Programming

Linear Programming (LP) is an optimization technique where the problem is expressed as a set of objectives and conditions, which both are only linear functions defined on numerical variables. The word "programming" does not refer to any particular implementation, rather is used in the sense of "planning" or "defining" the mathematical model. The conditions are usually expressed in form of inequalities that accurately express the constraints of a problem e.g. "the length of a schedule cannot express a maximum M". The canonical form of any LP problem in matrix form is:

$$\begin{gathered}\max \vec{c}^{\top} \vec{x} \\ A\vec{x} = \vec{b} \\ \vec{x} \geq 0\end{gathered}$$

One can imagine an example problem with two variables x and y and a set of 6 constraints (inequalities), which are functions of x and y. If we were to plot these inequalities on a 2D plane, the polygon formed my the intersection of the curves would represent the set of possible solutions, and the at the intersection of this polygon with the curves representing the objective functions would lie optimal solutions. This notion can be generalized to any number of dimensions, and real-world problems usually define hundreds or thousands variables. Algorithms that have been developed to find solutions to LP problems often inspect the *polytope* (i.e. the bounded region) and "walk" its edges to identify the global maximum using heuristics like the Simplex algorithm by Dantzig [8]. LP is a viable technique for many operations research problems, such as network flow, job shop scheduling, or resource allocation.

2.3.1 Mixed Integer Linear Programming

In this paper we explore the application of a subtype of LP, the Mixed Integer Linear Programming, to the problem of quantum scheduling. Decision variables in MIP formulations are integers, though it permits some non-discrete ones, hence the prefix "mixed". The motivation behind discretization of variables is that in real world scenarios, such as production planning, teritorial division or transportation schedule one often assigns resources (e.g. slots, CPUs, time slices) to individual objects (e.g. lanes, vehicles, programs), and the temporal aspect is also discrete (e.g. buses arrive at stops at fixed interval down to a minute resolution). Our problem, as we will see in Chapter 4, can be discretized and lends itself well to being expressed as an MIP model.

Since all LP are expressed in a uniform manner, the key design question aside from efficient linear representation is the choice of the solver. Solvers for MIP problems are known as Branch And Bound, or more recently Branch And Cut. The underlying method is as follows [11]:

Relaxation: The integrality restrictions of the equalities are removed, and the resulting LP problem is solved as outlined above. Once the solver identifies an upper or lower bound for a certain variable (in the context of the objective function), it proceeds to the next step.

Branching: The established real value for some variable e.g $x = 4.12$ is used to create two new MIP subproblems, one where $x \leq 4$ and the other where $x \geq 5$. The better solution of the two is propagated upwards in this search tree.

2.4 Genetic Algorithms

GAs, otherwise known as Evolutionary Algorithm (EA)s, are a class of optimisation techniques seeking to find a near-optimal solution to a problem by navigating through the space of all possible solutions (the search space). They do so by maintaining a population of chromosomes (problem solutions encoded in a certain way, for instance as a sequence of bits or graph nodes) and iterative application of genetic operators such as crossover, mutation or survival selection in order to mimic the process of natural selection, whereby the *fitness value* of each chromosome decided its chances to “pass on” its genes and contributing to finding the final (optimal or suboptimal) set of solutions. GA have been successfully applied to a range of scheduling problems, including job shop scheduling in [20] and [17].

Routines used within the main loop are known as genetic operators - functions simulating some part of the evolutionary process. Commonly encountered are:

- **Intialization:** The population of chromosomes is initialized according to some heuristic e.g. numeric chromosomes are sampled from a distribution. This population will then be evolved in following steps of the algorith.

- **Fitness Evaluation:** Each chromosome is assessed for its likelihood to “pass on” its genes onto the next generation. This function usually assigns a real value based on how well the given solution achieves specific objective(s).

- **Elitism:** A small number of best solutions is preserved from the current generation, effectively bypassing the selection and crossover steps. This may ensure smoother convergence and prevent loss of best solutions.
- **Selection:** Individual from the current generation are selected for the crossover to produce the next generation. The probability of selection is usually proportional to the fitness value, with underlying strategy as simple as taking *N* best individuals or more complex like tournament selection.
- **Crossover:** The selected parents are mated to produce one or more children chromosomes. The genetic information of parents is combined in a randomized manner to mimic genetic recombination, and introduce variability to the next generation.
- **Mutation:** The newly created offspring chromosomes undergo randomized changes to their genome with the intent of exploring the search space more effectively and preventing premature convergence.
- **Termination:** The generation loop of the algorithm will terminate if user-defined criteria are met .e.g. a fixed number of generations has passed, or the population has attained a satisfactory fitness threshold that has not changed over last *G* generations.

Pseudocode presented in Algorithm 1 shows a blueprint of a genetic algorithm. After initializing a population of given size we perform the main loop of the algorithm. In each generation, we evaluate the fitness of each individual, sort the population by fitness, select the best individuals (elites) to be preserved, select parents for crossover, perform crossover and child mutation, and finally add the elites back to the population. The algorithm terminates when user-defined criterion is met, for instance a fixed number of generations has passed or population has reached a certain fitness threshold.

2.4.1 Parallelization

With the emergence of commercial multi-core processors and cost-effective cloud computing, achieving a speedup in the execution time of a GA through parallelization has become a practical reality, leading to the development of Parallel Genetic Algorithms (PGAs) [4]. A closer examination of the sequential GA procedure reveals its inherently concurrent structure: the fitness evaluation function is idempotent and can be independently applied to each individual in the population. This makes it an embarrassingly parallel problem, as the function can be parallelized with minimal effort, allowing all evaluations within a single generation to be executed concurrently. This approach is

Algorithm 1 Pseudocode of a genetic algorithm

```
Require: $pop\_size \geq 0$
Ensure: $population \leftarrow initialize(pop\_size)$
 1: $generation \leftarrow 1$
 2: while true do
 3:     for $chromosome \in population$ do
 4:         $chromosome.fitness \leftarrow$ calcFitness($chromosome$)
 5:     end for
 6:     sortByFitness($population$)
 7:     $elites \leftarrow$ takeElites($population$)
 8:     $population \leftarrow population \setminus elites$
 9:     $children = \emptyset$
10:     for $(parent1, parent2) \in$ selectParents($population$) do
11:         $(child1, child2) \leftarrow$ crossover($parent1, parent2$)
12:         $children \leftarrow children \cup \{child1\} \cup \{child2\}$
13:     end for
14:     for $child \in children$ do
15:         mutate($child$)
16:     end for
17:     $population \leftarrow children \cup elites$
18:     $generation \leftarrow generation + 1$
19:     if canTerminate($generation, population$) then
20:         break
21:     end if
22: end while
```

particularly effective for panmictic GAs [4], which treat the population as a single collection on which all genetic operators are applied (e.g., the sequential model discussed earlier). The parallel implementation retains the functionality of the sequential baseline, with the only modification being that the loops responsible for evaluating fitness or similar functions are executed in parallel. This can involve distributing individuals across worker threads or partitioning the loop among threads to optimize computation.

2.4.2 NSGA

Non-dominated Sorting Genetic Algorithm (NSGA) is an extension of the classical GA specifically designed for multi-objective optimization. Regular GAs require us to scalarize our potentially conflicting multiple objectives into a single real or numeric value, which by default is not guaranteed to produce Pareto-efficient solutions. The NSGA addresses this limitation by introducing a novel chromosome evaluation mechanism based on a non-dominated sorting algorithm. In each generation, chromosomes are categorized into Pareto fronts which intuitively represent different levels of their quality.

After grouping solutions into the fronts, the parent selection proceeds by selecting the whole fronts until we reach the mandated selection size. Should the last front not fit the selection size fully, its solutions are prioritized according to the crowding distance, which is higher for solutions that differ more from one another. This mechanism promotes diversity among the selected solutions. This structure is visualized below:

NSGA shines in problems with a large number of objectives which may be conflicting, so we will explore it as one optimization technique for our scheduling problem.

NSGA-II is a successor of NSGA, whose notable improvements include a performance non-dominated sorting algorithm, which significantly reduces the computational complexity and enhancing the its performance. To promote biodiversity NSGA-II relies on tournament selection rather than simple natural selection. Chromosomes are categorized into successive rounds, where the winners have succeeded in "lower" rounds. They are then sorted by rank (i.e. the index of their Pareto front) and then, if we have to select fewer candidates than the last front size, crowding distance.

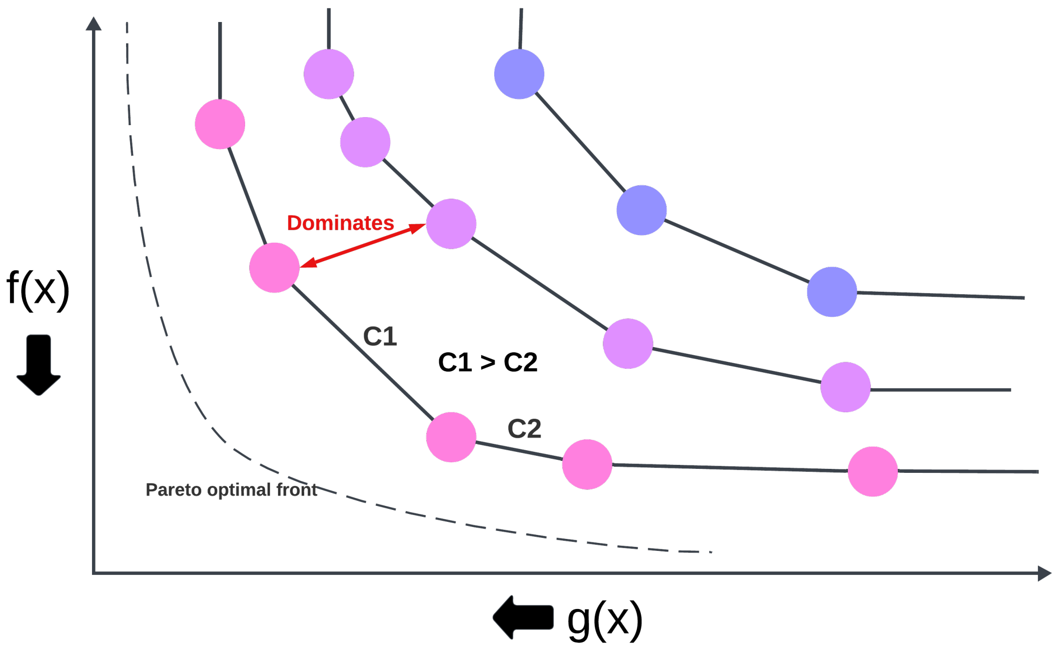

Figure 2.1: Pareto fronts and crowding distance

3 Overview

This chapter outlines the high level design of the proposed quantum scheduling system. It highlights the inherent challenges of quantum scheduling and contrasts them to those of classical systems. It defines the key design goals, functional and non-functional requirements of the system and proposes an abstract workflow that governs the operation of the system.

3.1 System Overview

We propose a quantum computing scheduling system illustrated by Figure 3.1:

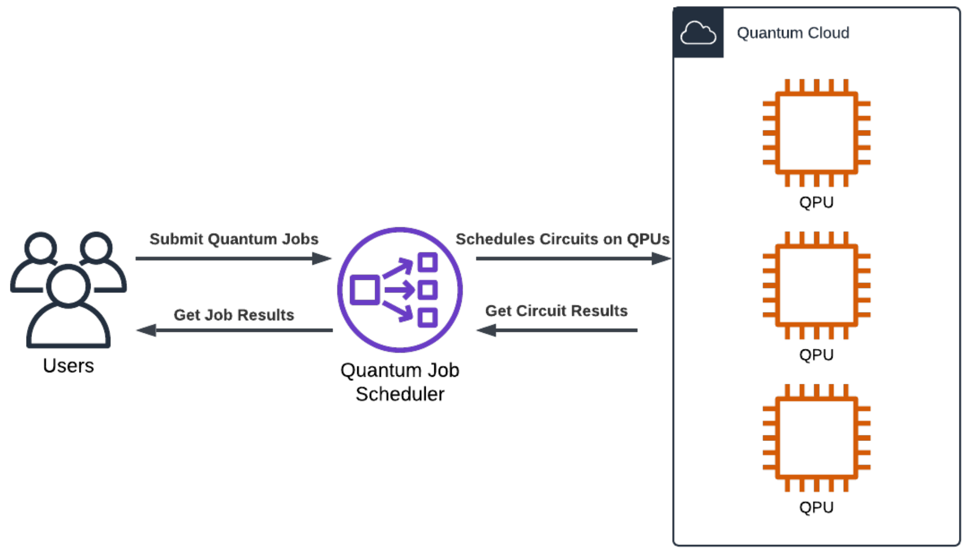

Figure 3.1: Quantum Scheduler overview

The primary goal of the system is to abstract the complexity of quantum circuit scheduling from the end users, providing them with a simple interface to submit their quantum programs and offer predictable execution times and computation fidelity. It would act as a gateway to the QPU fleet, accepting quantum jobs from users, optimizing them internally to find most suitable backends, and scheduling them for execution on the providers quantum fleet.

3.2 Challenges

Quantum processing units differ significantly from their classical counterparts, and scheduling jobs for quantum fleets at scale poses a number of unique challenges. Following issues, pointed by [15], are the most prominent:

- **Spatial Variability of QPUs**: Individual quantum computer exhibit different characteristics, such as error rates and waiting times across job executions, even if the machines have identical numbers of qubits. This heterogeneity separates quantum scheduling from classical datacenter environment, where processing units of given class are assumed to have almost identical properties.
- **Temporal Variability of QPUs**: The variable nature of qubit fidelity necessitates recalibrations of the QPU in regular intervals, and the scheduler needs to be aware of the latest estimations of waiting times and fidelities obtained during such updates. Should a transpiled circuit stay in the queue for too long, it might no longer be optimized for the current state of the QPU, implying the scheduler needs to adapt to the changing quality of the quantum cluster.
- **Intra-program dependencies**: Quantum programs are transpiled to a set of quantum jobs, where each job represents a single circuit. Some circuits may depend on each other in happen-after relation, and some may need to start and complete at exactly the same time on different QPUs to minimize the risk of decoherence. The scheduler needs to be aware of these constraints and embed them into the final schedule.
- **Scalability**: Whereas characteristics of classical systems are very predictable and, aside from hardware fault, do not change in time, quantum systems change over time as outline above. The error of a complex program is a product of errors in qubit connectivity and dynamic fidelities. The scheduler needs to incorporate both program constraints, QPU characteristics that vary in time and across machines, and still remain scalable.

3.3 Design Goals

Taking into consideration the characteristics of quantum systems and requirements for our scheduler, we formulated following key design goals:

- **Batch vs Online scheduling**: The scheduler should be able to able to optimize the fidelity of individual circuits, as well as the utilization of the whole QPU cluster to prevent long waiting times. It ought to operate on batches of circuits at a time to balance the load of the fleet, as well as any other "global" objectives like average error rate.
- **Multi-objective awareness**: The scheduler should be capable of optimizing for multiple conflicting objectives, namely the maximized fidelity of executed jobs and the maximized utilization of the QPU fleet. Achieving a tradeoff that is deemed optimal is crucial for the long-term success of the system, and intelligent strategies must be employed to achieve this.
- **Adaptability to problem size**: The scheduler should prioritize scalability to a certain degree. As discussed, the increase in number of quantum jobs, QPU fleet size and the incorporation of constraints like fidelity and circuit dependencies may lead to combinatorial explosion of the solution space. This complexity must be tackled in order for the scheduler to be viable in the ever evolving landscape of quantum scheduling.
- **Swappable scheduling algorithms**: Recognizing the requirements posed by the aformentioned challenges, we propose a system that can dynamically choose the most suitable scheduling strategy for the given problem size. The scheduler should include a decision tree that choose and configure algorithm on per-batch basis, factoring in the problem size, QPU fleet state and any other metadata.

3.4 System Workflow

Our scheduling system will be comprised of multiple stages in a pipeline. Upon accepting jobs from the user, the system will first transpile them into separate circuits, and carry out estimation of metadata required in the next stages: the execution times and fidelities of circuits on a per-QPU basis, as well as inter-circuit dependencies. This information is crucial for the decider to choose the most suitable scheduling algorithm for the given problem dimensions. The scheduling algorithm will be chosen depending on the number of circuits, the size of QPU fleet, and the number of dependencies between the circuits. It will then produce an optimal schedule according to our scheduling policy, that will then be submitted to the QPU fleet for actual execution. Figure 3.2 illustrates this process:

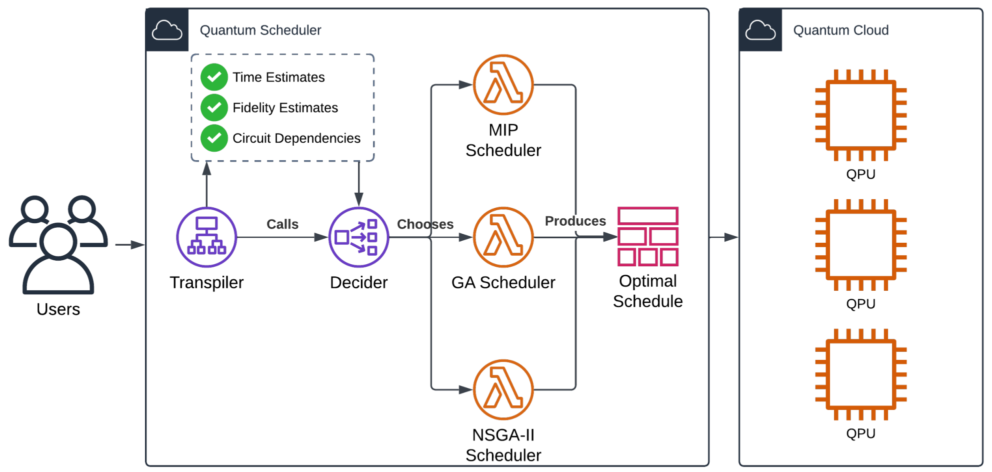

Figure 3.2: Scheduling pipeline overview

In following chapters we will focus primarily on the decider and individual scheduling algorithms, and treat the transpiler as a black box whose purpose is to provide us with the input circuits and their metadata.

4 Design

4.1 Multi-Objective Optimization

The centerpiece of the proposed scheduling system is a Multi-Objective Optimization component, whose goal is to find a schedule that balances the tradeoff between fidelity of individual jobs and the intelligent allocation of resources of the quantum cluster. This component relies on transpiled quantum programs and the estimations of fidelity and execution times defined in Chapter 3. It also takes the current state of the cluster, the initial QPU waiting times, into consideration. These inputs accurately reflect the state of the world from the the perspective of the scheduler, and will serve as inputs to every scheduling algorithm discussed below.

4.1.1 Problem Formulation

We intend to find an optimal assignment of quantum circuits to QPUs, where optimal is defined by having minimal makespan and minimal average error. The leads us to the following formulation:

$$
\begin{aligned}
&\min \begin{pmatrix} M_{\max} \\ E_{\text{avg}} \end{pmatrix} \quad \text{where} \\
&M_{\max} = \max_{j=1,\ldots,Q} \left(\sum_{i=1}^{N} S_{ij} + t_{ij} \right) \\
&E_{\text{avg}} = \frac{1}{N} \sum_{i=1}^{N} \left(1 - x_{ij} f_{ij}\right) \\
&\text{s.t.} \\
&\quad \sum_{j=1}^{Q} x_{ij} = 1 \quad \forall i = 1, \ldots, N \quad \text{(each task assigned to exactly one CPU)} \\
&\quad x_{ij} \in \{0, 1\} \quad \forall i = 1, \ldots, N; \quad \forall j = 1, \ldots, Q \\
&\quad S_{ij} \geq S_{kj} + t_{kj} \quad \text{if job } k \text{ precedes job } i \text{ on QPU } j \\
&\quad S_{ij} \geq 0 \quad \forall i = 1, \ldots, N; \quad \forall j = 1, \ldots, Q \quad \text{(each job starts at some time)}
\end{aligned}
$$

where x_{ij} is a binary variable indicating whether circuit i is assigned to QPU j; S_{ij} represents the start time of circuit i on QPU j; t_{ij} is the estimated execution time of circuit i on QPU j; f_{ij} is the estimated fidelity of the circuit i on QPU j; N is the number of circuits; Q is the number of available QPUs. Additionally, we define a follwing binary relation to express ordering constraints between circuits:

$$
\begin{aligned}
&\mathcal{R} = \{(a, b) \in C \times C \mid b \text{ starts after } a \text{ finishes}\}, \quad C \in \mathbb{Z} \\
&\text{s.t.} \\
&\quad a \,\hat{\mathcal{R}}\, a \quad \forall a \in C \quad \text{(reflexive)} \\
&\quad a \,\hat{\mathcal{R}}\, b \implies b \,\not\hat{\mathcal{R}}\, a \quad \forall a, b \in C \quad \text{(antisymmetric)} \\
&\quad a \,\hat{\mathcal{R}}\, b \land b \,\hat{\mathcal{R}}\, c \implies a \,\hat{\mathcal{R}}\, c \quad \forall a, b, c \in C \quad \text{(transitive)}
\end{aligned}
$$

The relation $\mathcal{R}$ specifies dependencies between circuits. Some circuits need to complete before others can begin. This relation is reflexive, antisymmetric and transitive. It is defined on the subset C of all circuits C_{all}. This subset together with the relation $\mathcal{R}$ forms a partial order $\mathcal{P} = (C_{all}, \mathcal{R})$, since some pairs are comparable e.g. dependent. This partial order needs to respected and embedded into any other sets or relations we use to express the model.

Our model addresses the binary optimization problem of two objectives, maximum makespan $M_{\max}$ and mean error E_{avg}. A makespan is defined as the time when the last circuit scheduled on a given backend completes. We calculate the maximum makespan of a schedule by finding the maximum such runtime on all backends which are part of the schedule. Minimizing the maximum makespan leads to minimized average makespan of the whole schedule and ensures the variance of makespan is also minimized. We consider error to be the inverse of circuit fidelity, leading to a natural minimization objective of average error across all backends.

4.1.2 Optimization Algorithms

Methods for solving the task scheduling problem on parallel systems can be broadly classified into following categories: state-space search-based methods, evolutionary algorithms, and tailored heuristics. State-space search methods rely on traversing a set of all states a problem can be in, in hope of finding the state optimal according to some criteria. Such methods typically traverse a graph of possible states, either with no prior information about the solution like depth- and breadth-first search, or with prior information like branch-and-bound or A*. These methods can find reasonably good solutions when coupled with a heuristic function describing the quality of the solution, however may suffer from exponential search space explosion in practice. These methods form a bedrock for mathematical programming methods, which require us to formulate the problem as a system of (usually linear) equations that is then solved by a solver like in the MIP or Satisfiability Modulo Theory approaches. Evolutionary methods, also known as genetic algorithms, can be thought of as gradient-free optimization schemes that rely on patterns of natural selection and gene propagation to iteratively produce dominant optimal solutions.

In our work, we explore approaches from either category:

- **Mixed Integer Linear Programming**: we extend the problem statement to a bi-objective MIP model and solve it using the Python PuLP library.
- **Genetic Algorithm**: we define chromosome and genetic operators most suitable for finding optimal schedules according to our multi-objective model.
- **Genetic Algorithm with NSGA-II**: we modify our genetic algorithm to select solutions from successive Pareto fronts in place of classical selectionstrategies such as natural or roulette selection.

4.2 Mixed Integer Linear Programming

This section outlines the precise mathematical formulation of the scheduling problem as a bi-objective MIP model, including the decision variables, constraints and any extensions required to produce optimal schedule.

4.2.1 Mathematical Formulation

In this section, we propose a bi-objective MIP model to solve the scheduling problem introduced in Section 4.1.1. The model attempts to find solutions with minimized makespan M as well as average mean error rate E.

A valid schedule is a set of assignments of circuits to QPUs which respects the dependency order. A solution is therefore defined by allocation of QPU, start time and duration to each circuit $i \in C$:

$$\begin{aligned} r_i &= \text{backend which circuit } i \text{ is assigned to (integer)} \\ t_i &= \text{scheduled start time of circuit } i \text{ (integer)} \\ d_i &= \text{execution time of circuit } i \text{ (integer)} \end{aligned}$$

We use following notation in equations below to express the domain of our decision variables and a-prori information about the problem:

$$\begin{aligned} C &= \text{Set of circuits, } C = \{1, ..., |C|\} \\ B &= \text{Set of QPUs (backends), } B = \{1, ..., |B|\} \\ \gamma &= \text{Set of dependencies, } (i,j) \in \gamma \text{ if } i,j \in C \text{ and} \\ &\quad\ \text{circuit } i \text{ must complete before } j \text{ can start.} \end{aligned}$$

We define following binary decision variables to express QPU assignment and temporal ordering constraints, which allow MIP solvers to efficienly traverse solution space:

$$h_{ib} = \begin{cases} 1 & \text{if circuit } i \text{ is assigned to QPU } b, \\ 0 & \text{otherwise.} \end{cases}$$

$$\sigma_{ij} = \begin{cases} 1 & \text{if circuit } i \text{ finishes before circuit } j \text{ starts,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\epsilon_{ij} = \begin{cases} 1 & \text{if QPU of circuit } i \ (r_i) \text{ is less than that of circuit } j, \\ 0 & \text{otherwise.} \end{cases}$$

The variables σ_{ij} define precedence constraints between circuits as required by the single-QPU and inter-circuit dependency order. Together with variables ϵ_{ij} the solver can express sequential, non-overlapping schedule of circuits on a single QPU.

Variables h_{ib} (and their auxiliaries $h1_{ib}$ and $h2_{ib}$) are used to express circuit-to-QPU assignments, which allow us to succinctly constrain the execution time d_i to discrete estimated circuit-on-QPU runtimes $runtime_{ib}$, and efficiently compute the mean error rate from estimated fidelities $mean\_error_{ib}$. For improved solvability of our MIP formulation, and to avoid rounding issues with floating point numbers, we discretized those key variables.

We further define auxiliary constants $M_{\max}$ to be the upper bound on the makespan and Δ to be a sufficiently small float point value necessary in formulation of the $h1_{ib}$ and $h2_{ib}$ constraints. $M_{\max}$ is used a the big-M value in $A8$, and is computed as the maximum potential runtime of all circuits on a single QPU.

$$\begin{aligned} M_{\max} &= \text{MAX}(initial\_waiting\_time) + \text{MAX}(runtime) * |C| \\ \Delta &= 0.01 \end{aligned}$$

where $initial\_waiting\_time_b$ is the earliest time when the first circuit can be scheduled on QPU b.

The following bi-objective constraint system is an extension of the problem statement defined in Section 4.1.1. Constraints $A1$, $A2a$ and $A3a$ express our intent to minize both the makespan M and mean error rate E with "static" lookup maps *runtime* and *mean_error*; $A4$ - $A8$ define the decision variable h_{ib} in terms of binary r_i; $A9$ - $A11$ ensure that circuits are scheduled sequentially on the same QPU, and only overlap in time if not; $A12$ - $A13$ constrain variables r_i and ϵ_{ij}; $A14$ - $A15$ ensure causal job order is reflected in their starting times; and finally $A16$ - $A21$ define domains for the decision variables.

$$\min \begin{pmatrix} M \\ E \end{pmatrix} \tag{A1}$$

s.t.

$$t_i + d_i \leq M \qquad \forall i \in C \tag{A2a}$$
$$E = \sum_{i \in C} \sum_{b \in B} (h_{ib} - 1) mean\_error_{ib} \tag{A2b}$$
$$d_i \geq \sum_{b \in B} (h_{ib} - 1) runtime_{ib} \qquad \forall i \in C \tag{A3}$$
$$|B| h1_{ib} \geq r_i - b + \Delta \qquad \forall i \in C, \forall b \in B \tag{A4}$$
$$|B|(1 - h1_{ib}) \geq b - r_i - \Delta \qquad \forall i \in C, \forall b \in B \tag{A5}$$
$$|B| h2_{ib} \geq b - r_i + \Delta \qquad \forall i \in C, \forall b \in B \tag{A6}$$
$$|B|(1 - h2_{ib}) \geq r_i - b - \Delta \qquad \forall i \in C, \forall b \in B \tag{A7}$$
$$h_{ib} \geq h1_{ib} + h2_{ib} \qquad \forall i \in C, \forall b \in B \tag{A8}$$
$$\sigma_{ij} + \sigma_{ji} \leq 1 \qquad \forall i \neq j \in C \tag{A9}$$
$$\epsilon_{ij} + \epsilon_{ji} \leq 1 \qquad \forall i \neq j \in C \tag{A10}$$
$$\sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 \qquad \forall i \neq j \in C \tag{A11}$$
$$r_j + r_i - 1 - (\epsilon_{ij} - 1)|B| \geq 0 \qquad \forall i \neq j \in C \tag{A12}$$
$$r_j + r_i - \epsilon_{ij}|B| \leq 0 \qquad \forall i \neq j \in C \tag{A13}$$
$$t_j + d_i - (\sigma_{ij} - 1) M_{\max} \leq t_j \qquad \forall i \neq j \in C \tag{A14}$$
$$t_i + d_i \leq t_j \qquad \forall (i, j) \in \gamma \tag{A15}$$
$$\sigma_{ij} \in \{0, 1\} \qquad \forall i \neq j \in C \tag{A16}$$
$$\epsilon_{ij} \in \{0, 1\} \qquad \forall i \neq j \in C \tag{A17}$$
$$h_{ij}, h1_{ij}, h2_{ij} \in \{0, 1\} \qquad \forall i \neq j \in C \tag{A18}$$
$$r_{ij} \in \{1, ..., |B|\} \qquad \forall i \in C \tag{A19}$$
$$t_i \geq 0 \qquad \forall i \in C \tag{A20}$$
$$d_i \geq 1 \qquad \forall i \in C \tag{A21}$$

4.2.2 Implementation Caveats

The above formulation, at a first glance, does not account for initial waiting times of circuits on QPUs. We have decided to represent the initial waiting times as the last W jobs by fixing the values of d_i, r_i and t_i variables to the waiting time, QPU index and 0, respectively. This ensures those variables are treated as constants by the solver, and allow us to elegantly express the initial waiting times in the model.

4.3 Genetic Algorithm

This section outlines the layout of the chromosome and the design of genetic operators that comprise the genetic solver. The choice of data structures and algorithms is crucial to efficient convergence of the algorithm to the optimal solution.

4.3.1 Chromosome Structure

Since we are attempting to solve a job scheduling problem, our chromosome should represent a single schedule, or an assignment of jobs to backends. We treat our chromosome as a contiguous array of tuples, where each tuple is comprised of a a pair of integers: the first one points to the job, and the second one to the backend to which that job is assigned. This representation is illustrated in Figure 4.1.

i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	
(J_1, B_1)	(J_2, B_2)	(J_3, B_3)	(J_4, B_4)	(J_5, B_5)	(J_6, B_6)	(J_7, B_7)	(J_8, B_8)	(J_9, B_9)	...

Figure 4.1: Chromosome representation

We chose to store explicit job index rather than rely on array indexing $i_0, ..., i_J$ to ensure that all genetic operators access jobs in a consistent and well-defined order during array iteration. This is necessitated by the inter-circuit dependency constraints defined in Section 4.1.1. We need to embed the partial order of the dependency happens-after relationship. One approch to doing so would be to maintain a mapping from circuit indices to their positions in the sorted order that embeds the partial order, another would be to pre-sort every chromosome according to this order and keep the circuit index inside the cells of the array themselves. We choose the second approach.

4.3.2 Embedding Dependency Partial Order

We rely on topological sorting to embed the partial dependency order into the schedule vector. This allows us to safely and efficiently perform genetic operations such as fitness evaluation, which relies on the fact that at index i_N we have already processed all jobs that job N depends on, or crossover, which is a simple array slicing operation that nevertheless produces offspring sorted in the same order as the parents. We can do this without additional indirection or sorting before every operation, thus improving the throughput of the algorithm at the cost of increase memory usage. Figure 4.2 presents an example circuit dependency relation, and to the right two possible linearizations of this order. Similar approach has been evaluated [17], where authors efficiently compute the makespan using the longest path algorithm on the linearized schedule, and perform the crossover much like we do.

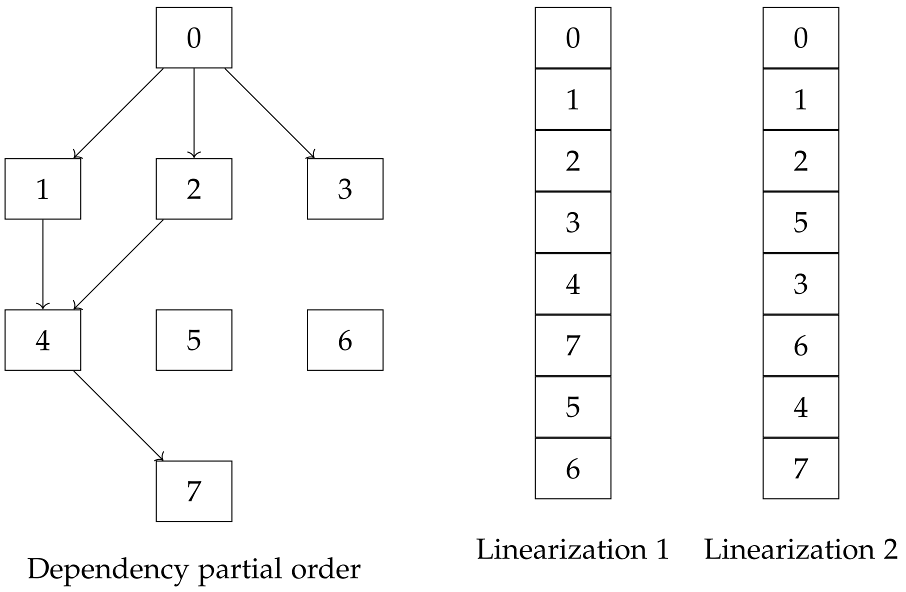

Figure 4.2: Dependency order linearizations

Since there exist a number of possible linearizations of the partial order, and our genetic operators calculate the fitness of the schedule based on that exact order, reliance on only one linearization could potentially limit the search space of the algorithm. Some linearizations might translate to more compact or less error-prone schedules than others. Our algorithm ensures that linearization of dependecy order gives chromosome evolutionary advantage, and "better" linearizations are more likely to be propagated to the next generations. The implementation of this technique is presented in Section 4.3.3.

4.3.3 Genetic Operators

We tailor our genetic operators to the specifics of the problem at hand. The initiation of a single chromosome is performed by generating a number of dependency order linearizations, the topologically sorted job indices. We do this to introduce some variability into the order of jobs in the chromosome, since otherwise all candidate solutions in a given run would be evaluated in the same job order. This would limit the search space for optimal schedule significantly.

Chromosome Initiation

We opted for a compromise of pre-generating a fixed number of topological orders and randomly selecting one of them for each chromosome during initialization, and refering to this linearization as ORDER of the chromosome. We then generate an assignment of jobs to backends by sampling from a uniform distribution of backend indices. The pseudocode for this operation is presented in Algorithm 2.

Algorithm 2 Chromosome initiation pseudocode

```
1: procedure InitChromosome($j, b, topological\_orders$)
2:     $order \overset{R}{\leftarrow} topological\_orders$
3:     $B_1, B_2, ..., B_j \overset{iid}{\sim} \mathcal{U}(0, b)^j$
4:     $chromosome \leftarrow \emptyset$
5:     for $i \in 1, \ldots, j$ do
6:         $chromosome \leftarrow chromosome \cup (order[i], B_i)$
7:     end for
8:     return $chromosome$
9: end procedure
```

Fitness Function

The fitness function iterates over job-to-backend assignments in the topological order and computes the maximum makespan of any backend, as well as the total fidelity of the schedule. We calculate the earliest possible starting time of a job by consulting the initial waiting times, the next free time of the assigned backend, as well as the end times of jobs that the current job depends on. A single iteration over the schedule is sufficient to compute both values and return the weighted sum of the two.

We can adjust the levels of relative important of each of the two objectives by tweaking the λ parameter, which adjusts the weight of the two objectives. This allows us to prioritize either of the objectives, however does not guarantee the solutions to be Pareto-efficient. This will be addressed in Section 4.4. The pseudocode for this operation is presented in Algorithm 3.

Algorithm 3 Fitness function pseudocode

```
1: procedure CalcFitness($chromosome$)
2:     $total\_fidelity \leftarrow 0$
3:     $max\_makespan \leftarrow 0$
4:     $job\_end\_times \leftarrow \emptyset$
5:     $backend\_end\_times \leftarrow \emptyset$
6:     for $(job, backend) \in chromosome$ do
7:         $start\_time \leftarrow 0$
8:         $start\_time \leftarrow \textsc{max}(start\_time, waiting\_times[backend])$
9:         $start\_time \leftarrow \textsc{max}(start\_time, backend\_end\_times[backend])$
10:        for $(dep\_job, dependent) \in dependencies$ do
11:            if $job = dependent$ then
12:                $start\_time \leftarrow \textsc{max}(start\_time, job\_end\_times[dep\_job])$
13:            end if
14:        end for
15:        $makespan \leftarrow start\_time + execution\_times[(job, backend)]$
16:        $max\_makespan \leftarrow \textsc{max}(max\_makespan, makespan)$
17:        $job\_end\_times[job] \leftarrow makespan$
18:        $backend\_end\_times[backend] \leftarrow makespan$
19:        $total\_fidelity \leftarrow total\_fidelity + fidelities[(job, backend)]$
20:    end for
21:    return $\lambda \frac{1}{max\_makespan} + (1 - \lambda) \frac{total\_fidelity}{\textsc{len}(chromosome)}$
22: end procedure
```

Mutation

Mutation is carried out in two steps: first, we swap a certain number of backend assignments at random indices in the chromosome. This swap is performed with a certain probability specified in the algorithm hyperparameters. The second step is the so-called condensation, which identifies the gaps in the schedule and attempts to fill them by moving jobs to idle backends. This is in principle similar to memory allocators attempting to coalesce free memory into contiguous blocks for better locality and utilization. The condensation has been observed to improve the fitness of the schedule by reducing the makespan. The pseudocode for this operation is presented in Algorithm 4.

Other Operators

The selection, elititism and crossover operators are fairly standard and do not necessitate the pseudocode visualization:

- **Selection**: we implement natural selection i.e. taking N fittest individuals from each generation.
- **Elitism**: much like selection, we pick N fittest individuals from each generation before the parent selection.
- **Crossover**: we apply two-point list crossover, selecting at random two indices i_1, i_2 within the parent chromosomes, then swapping the corresponding slices $0..i_1$, $i_1..i_2$ and $i_2..N$, where N is the length of the chromosome. We only apply crossover between parents scheduled according to the same topological order, ensuring resulting children are also valid schedules.
- **Termination Criteria**: we stop the algorithm after a fixed number of generations or when the population reaches a certain fitness threshold.

4.4 Genetic Algorithm with NSGA

We extend the genetic algorithm defined in Section 4.3 with a custom selection operator based on the NSGA algorithm. As outlined in Chapter 2, the NSGA approach relies on categorizing the solutions of a multi-objective optimization problem into so-called Pareto fronts, where solutions in every front represent different levels of dominance [24].

Algorithm 4 Mutation pseudocode

```
procedure MUTATE($chromosome$)
    if DICETHROW() then                                         ▷ Schedule mutation
        for $i \in 1, \ldots, m$ do
            $I_1, I_2 \overset{iid}{\sim} \mathcal{U}(0, \textsc{len}(chromosome))^2$
            SWAPBACKENDS($chromosome, I_1, I_2$)
        end for
    end if
    $gaps \leftarrow \emptyset$
    $job\_end\_times \leftarrow \emptyset$
    $backend\_end\_times \leftarrow \emptyset$
    for $(job, backend) \in chromosome$ do                      ▷ Preparation for condensation
        $start\_time \leftarrow \textsc{calcStartTime}(job, backend)$
        $gap \leftarrow start\_time - backend\_end\_times[backend]$
        if $gap > 0$ then
            $gaps[backend] \leftarrow gaps[backend] \cup (backend\_end\_times[backend], start\_time)$
        end if
        $makespan \leftarrow start\_time + execution\_times[(job, backend)]$
        $job\_end\_times[job] \leftarrow makespan$
        $backend\_end\_times[backend] \leftarrow makespan$
    end for
    for $top\_backend \in \textsc{busiestBackends}()$ do        ▷ Schedule condensation
        $job \leftarrow \textsc{firstFreeJob}(top\_backend)$
        $idle\_backend \leftarrow \textsc{findIdleBackend}(top\_backend)$
        for $gap \in gaps[idle\_backend]$ do
            if $\textsc{gapFits}(gap, job)$ then
                $\textsc{moveJobToGap}(job, gap)$
                break
            end if
        end for
    end for
end procedure
```

4.5 Multi-Objective Definition

First, we define an abstract type for the objective of our model. The concrete instances of this type will implement two methods:

```
pub trait Objective {
    type Sol;
    type Distance: Sized;

    fn total_order(&self, a: &Self::Sol, b: &Self::Sol) -> Ordering;
    fn distance(&self, a: &Self::Sol, b: &Self::Sol) -> Self::Distance;
}
```

Listing 4.1: NSGA objective contract

where TOTAL_ORDER() compares any two solutions in the scope of the objective (e.g. real number comparison for makespan / mean fidelity), and DISTANCE(), which returns the absolute difference between the two objectives (e.g. absolute real difference for either of our objectives).

We then define the concept of a dominance order in the concext of multi-objective optimization. A solution will dominate another, if it is better it at least one objective, and at least as good in the others. Conversely, if it worse in all objectives, it will be dominated, otherwise they will be considered equal. We then define a type *MultiObjective* to be a collection of *Objective* objects, and implement our dominance order trait method on it:

```
pub struct MultiObjective<'a, S: 'a, D: 'a> {
    pub objectives: &'a [&'a dyn Objective<Solution=S, Distance=D>],
}

impl DominanceOrd for MultiObjective<Sol>{
    fn dominance_ord(&self, a: &Self::Sol, b: &Self::Sol) -> Ordering {
        let mut less_cnt = 0;
        let mut greater_cnt = 0;

        for objective in self.objectives.iter() {
            match objective.total_order(a, b) {
                Ordering::Less => { less_cnt += 1; }
                Ordering::Greater => { greater_cnt += 1; }
                Ordering::Equal => {}
            }
        }

        if less_cnt > 0 && greater_cnt == 0 {
            Ordering::Less
        } else if greater_cnt > 0 && less_cnt == 0 {
            Ordering::Greater
        } else {
            Ordering::Equal
        }
    }
}
```

Listing 4.2: Dominance order implementation for multi-objective

We assert that a multi-objective dominates (i.e. is greater than) another if it is greater in all sub-objectives; likewise it is dominated if lower in all sub-objectives, and equal otherwise. This total order allows us to group all solutions into a list of lists, where each list l_i contains solutions that are dominated by s_i. This is exactly the operation performed by NonDominatedSort().

4.6 Rank and Select

We then proceed to assign the crowding distance to every member of the first Pareto front. In case a front has more solutions than we are allowed to select per generation, we proritize those solutions with higher crowding distance to maintain diversity of the population, which is also where the distance() of each objective is used to calculate the higher-dimensional distance between solutions.

As illustrated in Algorithm 5, the NSGA selection is performed by first sorting the population into non-dominated fronts according to the multi-objective, then assigning crowding distances to each individual in the front, and finally selecting as many individuals as we are allowed to for the next generation, potentially traversing through multiple fronts.

Algorithm 5 NSGA-II Rank and Select

```
procedure RankAndSelect($chromosomes$)
    $result \leftarrow \emptyset$
    $front \leftarrow$ NonDominatedSort($chromosomes$)
    $missing\_solutions \leftarrow$ Len($chromosomes$)
    while true do
        $pop\_with\_distance \leftarrow$ AssignCrowdingDistance($front$)
        if len($pop\_with\_distance$) $> missing\_solutions$ then
            SortBySortingDistanceRev($pop\_with\_distance$)
        end if
        $take \leftarrow$ Min($pop\_with\_distance, missing\_solutions$)
        $result \leftarrow result \cup$ Take($pop\_with\_distance, take$)
        $missing\_solutions \leftarrow missing\_solutions - take$
        if $missing\_solutions = 0$ then
            break
        end if
        $front \leftarrow$ NextFront($front$)
    end while
end procedure
```

5 Implementation

This chapter examines the technological stack and provides an in-depth analysis of the implementation details of the proposed system. We delve into particular technologies and programming techniques used to implement the algorithms, as well as rationale for our choices.

5.1 Mixed Integer Linear Programming

This section provides an overview of our MIP model implementation.

5.1.1 Technological Stack

The model and auxiliary tools are implemented in Python [16], a language widely known for its versatility, spanning ecosystem of libraries and frameworks, and ease of use. For these reasons it is a perfect choice for rapid prototyping and evaluation of our model.

5.1.2 Model Definition

We employ the PuLP (Python Linear Programming) [21] library to formulate, solve and evaluate our MIP model. PuLP offers a rich Domain Specific Language (DSL) for defining variables, optimization goals, as well as binary, integer, and continuous variables and their constraints. This enabled us to succinctly express our optimization problem without losing performance. PuLP transpiles the model to an intermediate format compatible with various solvers. During our tests we used the open-source Coin-or branch and cut solver (CBC), though we may easily switch to other priopprietary solvers such as Gurobi, CPLEX, or GLPK, which often offer superior performance for larger problem sizes.

Pulp's DSL allows to express our MIP formulation in an almost one-to-one fashion. Code snippet below showcases the initial initial equations of the model, as well as optimization goals:

```python
# Create the problem variable
m = op.LpProblem("m", op.LpMinimize)
```

```python
# Assigned QPU for circuit i
r = {i: op.LpVariable(f"r{i}", 0, len(J), op.LpInteger) for i in I}

constraints = [
    # A1
    {i: (t[i] + d[i] <= W, f"eq_makespan_{i}") for i in I},
    # A2
    {i: (d[i] >= op.lpSum((h1[(i, j)] + h2[(i, j)] - 1) * runtime[(i, j)]
    for j in J), f"eq_1_{i}") for i in IA},
    # A3
    {(i, j): (sigma[(i, j)] + sigma[(j, i)] <= 1, f"eq_2_{i}-{j}") for i, j
    in it.product(I, I) if i != j},
    {(i, j): (theta[(i, j)] + theta[(j, i)] <= 1, f"eq_3_{i}-{j}") for i, j
    in it.product(I, I) if i != j},
    {(i, j): (sigma[(i, j)] + sigma[(j, i)] + theta[(i, j)] + theta[(j, i)]
    >= 1, f"eq_4_{i}-{j}") for i, j in it.product(I, I) if i != j},
    # ...
]

# Minimize total makespan (W) and the average mean error (E)
m += W
m += op.lpSum((h1[(i, j)] + h2[(i, j)] - 1) * mean_error[(i, j)] for i, j in
     it.product(IA, J))
# Add all the constraints
for c in constraints:
    for i in c:
        m += c[i]
# Solve
m.solve()
```

Listing 5.1: PuLP Snippet

For sample data generation and overall plumbing, we employ a set of well known libraries like `numpy` (vectorized sampling execution, waiting times and fidelities from normal distributions), `networkx` (building the Directed Acyclic Graph (DAG)s of circuit dependencies and linearizing them via topological sort) or `itertools` (easy formulation of cartesian products of sets of PuLP variables in the model constraints).

5.2 Genetic Optimization

This section provides an overview of our GA and NSGA scheduler implementation.

5.2.1 Technological Stack

The non-functional requirements for the genetic optimizer are more stringent than for the MIP: the optimizer should be able to scale much better, and be parallelizable. For this reason we wrote the genetic and NSGA optimizers in Rust, a modern systems programming language offering type and memory safety, as well as predictable and performant runtime. Rust allows for relatively fast prototyping and impressive multi-core performance when compiled. Its vast standard library and collection of open-source libraries called crates ensures smooth development experience.

5.2.2 Program Design

Rust's type system allows us to abstract the contract of the optimizer and the genetic algorith, and pick implementation dynamically during tests. We chose to decouple the optimizer from the genetic algorithm logic as both GA and NSGA optimizers share the same underlying genetic operators, but differ mostly in the selection strategies. This contract, depicted in Listing 5.3, allows us to elegantly encapsulate both implementations and to easily swap them during test runs.

```rust
// This trait represents a single problem solution
pub trait Chromosome: Send + Sync + Debug + Clone {
    fn fitness(&self) -> f64;
    fn order(&self) -> usize;
}

// This trait represents the stopping condition of the algorithm
pub trait Evaluator<C: Chromosome>: Send + Sync + Debug {
    fn can_terminate(&mut self, pop: &[C], generation: i32) -> bool;
    fn generation(&self) -> i32;
}

// This trait encapsulates the optimizer logic (GA, NSGA-II, etc.)
pub trait Optimizer<C: Chromosome>: Send + Sync + Debug {
    fn optimize(&mut self, eval: &mut Box<dyn Evaluator<C>>) -> Vec<C>;
}

// This trait encapsulates the underlying genetic algorithm
pub trait Algorithm<C: Chromosome>: Send + Sync + Debug {
    fn generate(&self) -> Vec<C>;
    fn evaluate(&self, pop: Vec<C>) -> Vec<C>;
    fn mutate(&self, pop: Vec<C>) -> Vec<C>;
    fn crossover(&self, parents: &Vec<C>) -> Vec<C>;
    fn select(&self, pop: &[C]) -> Vec<C>;
    fn elitism(&self, pop: &[C]) -> Vec<C>;
}
```

Listing 5.2: Rust traits describing our contract

We constrain the `Chromosome` trait to be safe for multithreaded operations by mixing in the `Send` and `Sync` traits, as the chromosomes are constantly copied between threads during the map-reduce-like execution of the parallel GA. Our quantum schedule is essentially an ordered mapping of jobs to backends, and so is easily represented in memory as a vector of tuples:

```
#[derive(Clone, Debug, Default)]
pub struct QuantumSchedule {
    pub genes: Vec<(u32, u32)>,
    pub order: usize,
    pub fitness: f64,
    pub makespan: u32,
    pub mean_fidelity: f64,
}
```

Listing 5.3: Rust traits describing our contract

5.2.3 Data Parallelism

In order to take advantage of the embarrassingly parallel structure of the fitness function, crossover and mutation operators, we use the `rayon` [10] library for simple and opaque data-level parallelism. Much like in Open Multi-Processing (OpenMP), all we need to do is define parallel sections of the algorithm, and the runtime takes care of scheduling work units onto available workers [2]. In code this is expressed using map-reduce paradigm with Rust's immutable iterators guaranteeing freedom from race conditions. `rayon` spawns a number of software threads and maps the work units onto them, employing queues and work stealing to adapt to CPU's load. The data flow of our GA implementation is shown in the diagram below (Figure 5.1). We can safely paralleize the CalcFitness, Crossover and Mutate steps, as they require no dependencies between individual chromosomes.

5.2.4 Benchmarking

Evaluating the performance of the optimized and tweaking the hyperparameters of the algorithm was necessary for quick iteration during development. We used the Criterion library to perform microbenchmarks of the genetic algorithm, measuring the system time and CPU operations per unit time of genetic operators. We also developed a test

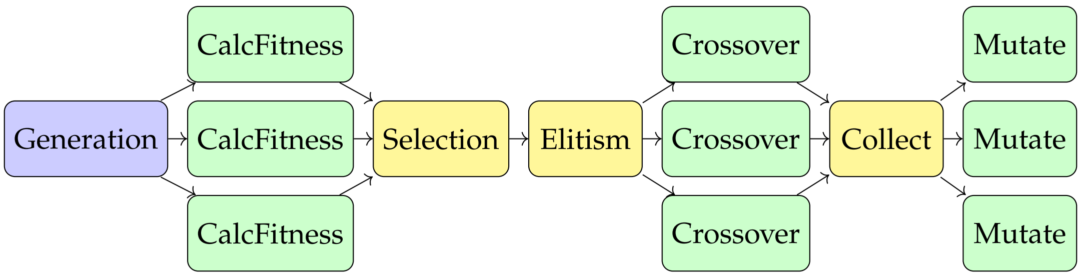

Figure 5.1: Dataflow in parallelized GA

framework for evaluating the runtime and quality of runs of the genetic and NSGA optimizers. We would define our test cases as a Rust struct, whose parameters would be combined in a cartesian product, and each each combination would be evaluated by the optimizer multiple times to measure the average runtime and schedule quality:

```
TestSchema {
    nsga2: vec![false, true],
    jobs: vec![512, 1024, 2048, 4096],
    backends: vec![32, 64],
    selection_size_percentage: vec![0.25],
    elitism_size_percentage: vec![0.05],
    mutation_pairs_percentage: vec![0.15],
    mutation_rate_percentage: vec![0.15],
    dependency_percentage: vec![0.1, 0.2, 0.3],
    multi_objective_ratio: vec![0.0, 0.25, 0.5, 0.75, 1.0],
}
```

Listing 5.4: Example test GA and NSGA-II test case

For plotting the visualizations of output schedules of our algorithms we employed the `plotters` create, a well-known Rust library for creating high resolution plots, figures and charts.

6 Evaluation

6.1 Experimental Setup

We execute our benchmarks on a MacBook Pro equipped with an Apple M3 Pro chip, a 11 (5 performance and 6 efficiency) core system, and 18 gigabytes of RAM. See table for details on the benchmarking environment. We execute the benchmarks by running our Rust test suite with the `-release` flag enabled, ensuring it is compiled with optimizations and without debug information.

Table 6.1: Specification of the benchmarking environment

Spec	**Apple M3 Pro**
Cores	11
Memory	18gb
Architecture	Arm v8
Minimum Frequency	2.75 GHz
Maximum Frequency	4.06 GHz
L1 I-Cache + D-Cache	192+128 KiB
L2 Cache	16 MiB

6.2 Synthetic Data Generation

To accurately evaluate the performance of our optimization algorithms, we generate synthetic data that closely resembles the real-world quantum environment. We've generated following metadata for each test run of the scheduler:

- **Circuit Execution Times:** we sampled execution times for each (circuit,QPU) pair from a uniform distribution between 5 and 25 seconds.
- **Initial Waiting Times:** we sampled initial waiting times of each QPU from a uniform distribution between 5 and 20 seconds.

- **Circuit Fidelities:** we simulated variability in QPU fidelities by first generating a normal distribution with a mean sampled from a uniform distribution between 0 and 1, and a standard deviation of 1. We then sampled fidelities for each (circuit,QPU) pair from the QPU-specific distribution.
- **Circuit Dependencies:** we generated a directed acyclic graph whose nodes represent circuits and edges the dependencies between circuits. We then sampled a number of disconnected components from this graph to represent the dependencies within quantum programs.

6.3 Evaluation of MIP Scheduler

We have found the MIP scheduler to be a viable solution for small problem sizes of up to 10 jobs and backends, with the runtime increasing prohibitively (> 120 seconds) for problems larger than this. A choice of different MIP solvers or machine configurations might have an impact on the runtime, however as this was not the scope of this thesis we chose to focus on benchmarking the much more scalable GA and NSGA optimizers.

6.4 Evaluation of GA Scheduler

We evaluated following benchmark scenarios for the GA scheduler:

Table 6.2: Benchmark Scenarios for TestSchema

Schema	Jobs	Backends	Dependencies (%)	Multi-Objective Ratio (%)
1	{128, 256, 512}	{16, 32}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}
2	{256, 512}	{32, 64}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}
3	{512, 1024}	{64, 128}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}
4	{1024, 2048}	{128, 256}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}
5	{2048, 4096}	{256, 512}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}
6	{4096, 8192}	{512, 1024}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}
7	{8192, 16384}	{1024, 2048}	{0.1, 0.2, 0.3}	{0.0, 0.25, 0.5, 0.75, 1.0}

We evaluated the cartesian product of the array-like variables in every benchmark schema, with each individual combination bein evaluated 5 times. We have chosen to fix certain hyperparameters of the GA after deeming them to be optimal for most of the scenarios, as seen in Table 6.3. We execute every benchmark scenario with both GA and NSGA optimizers.

Table 6.3: Specification of the benchmarking environment

Hyperparameter	**Value**
Population size	$\textsc{max}(10000, jobs * 5)$
Selection size	$\textsc{max}(2000, jobs * 0.25)$
Elitism size	Selection size $*0.25$
Mutation rate	0.15

A single run of our GA scheduler produces a schedule that can be visualized on a stacked bar chart, where each bar represents the circuit, arranged in lanes representing QPUs. The auxiliary chart on the right highlights circuit dependencies to help us visually assess their correctness:

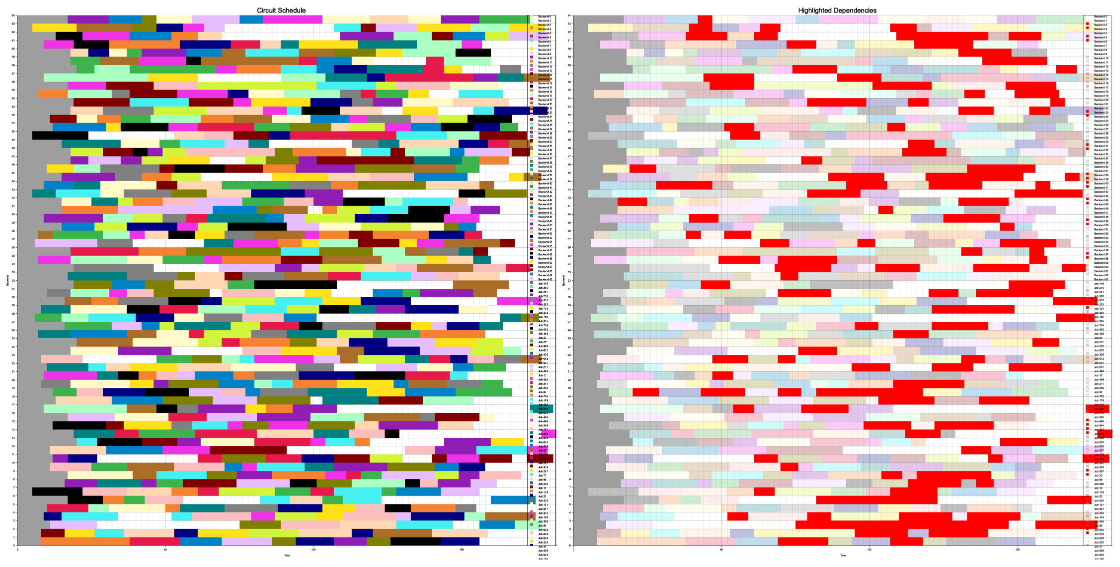

Figure 6.1: Sample run of GA scheduler

We first inspect the relationship between the number of jobs, backends and the mean runtime, fitness, makespan and overall fidelity of the schedules by visualizing them in figures below. We plot the qualities of interest, along with sub-divisions by percentage of dependencies (10-30%) and the multi-objective ratio λ (0-100% in 20% increments). The X axis is meant to be read as a list of tuples $(backends, dep\_percentage)$, likewise Y axis is a list of tuples $(jobs, \lambda)$.

We observe relatively small runtime (median 1.6 seconds) in virtually all executions. The runtime only varies in the problem size ranges 2048/128. The runtime appears to not depend on the variation of number of dependencies or the multi-objective ratio.

Likewise, the distribution of mean fitness appears to be uniform across test scenarios. We observe that mean fitness increases significantly with the priotitization of the fidelity objective ($\lambda \to 0.0$). We note a decrease in overall fitness with the scale of the problem, indicating finding the optimal schedule becomes more difficult with the increase in the number of jobs and backends. The difference most likely stems from different scales of both objectives - the makespan objective is in fact an inverse of the schedule makespan, which is bound to be smaller in absolute terms than the mean fidelity. Figure 6.3, on which we observe that the fitness, which is a weighted sum of inverse of the max makespan and mean fidelity, clearly shows that biasing the optimizer towards fidelity versus makespan has a significant impact on the fitness. This could be explained by the fact that both objectives are not directly comparable i.e. the mean fidelity will constitute a larger component of the fitness value when we deprioritize the makespan objective, which is the inverse of max. makespan and thus will be quickly dominated by the "linear" fidelity score. We will investigate how NSGA handles this tradeoff in the following subsections.

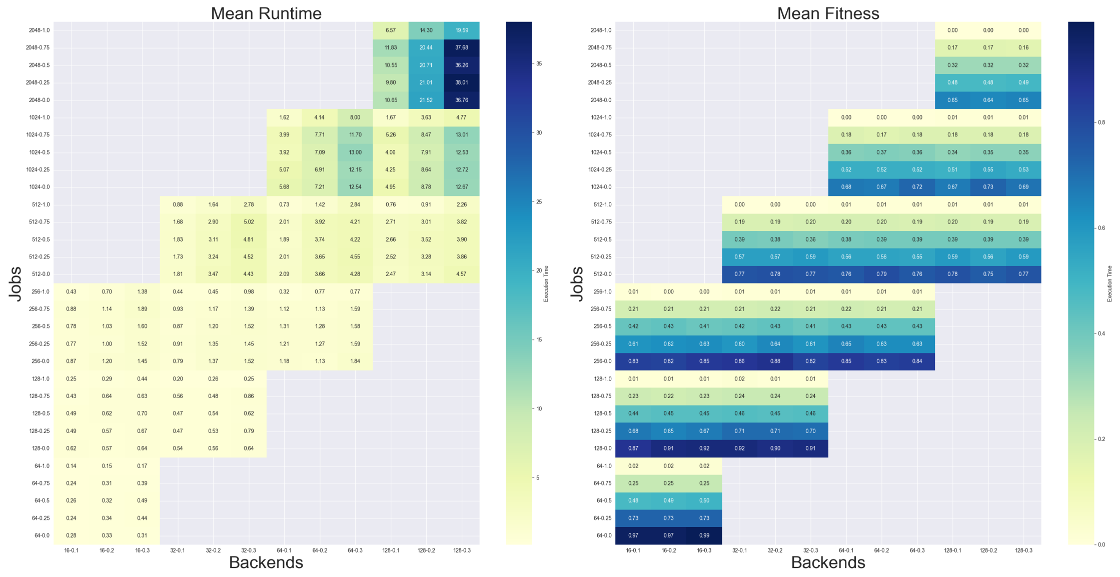

Figure 6.2: Genetic Algorithm performance overview

Interestingly, the mean fidelity of optimal schedules appears to decrease uniformly with the scaling of number of jobs/backends, with the exception of scenarios where fidelity is maximally prioritized ($\lambda = 0.0$). This could indicate that the algorithm is ihnerently biased towards the makespan objective optimization, or that the fidelity optimization is not as effective as the makespan optimization.

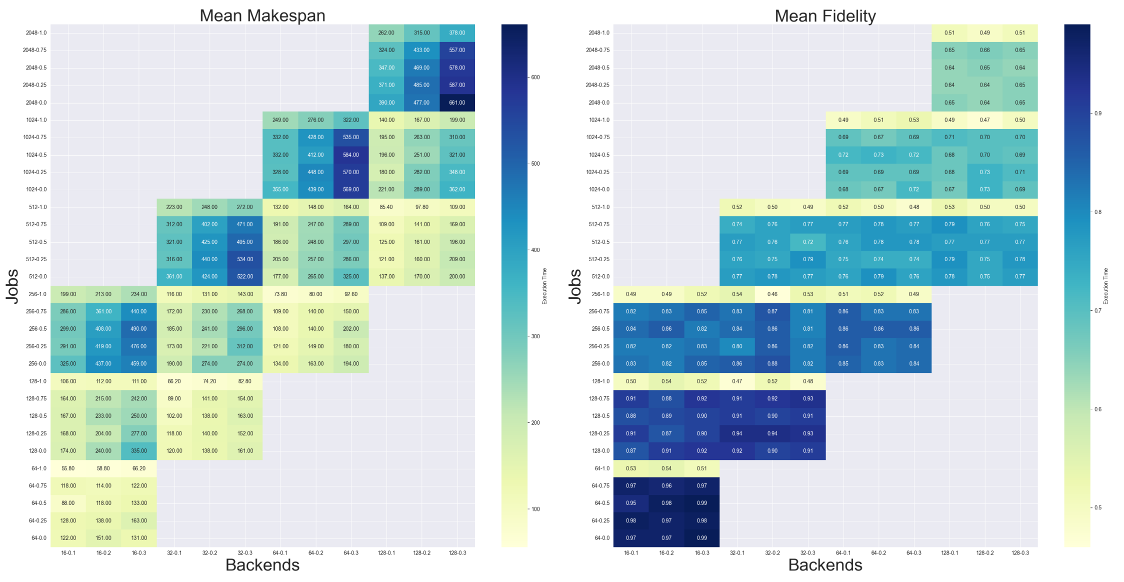

Figure 6.3: Genetic Algorithm results overview

6.4.1 Multi-Objective Weighing Sensitivity Analysis

Having established that the GA scheduler scales sufficiently well to fulfil our design goals, we proceed to evaluate the sensitivity of the schedule quality to the multi-objective weighing. We have observed a variation in the tradeoff between the makespan and fidelity objectives, and would like to statistically quantify the impact the weights have on that tradeoff.

Table 6.4 indicates that the change from 100% priotitization of makespan objective to just 75% leads to 59% decrease in mean makespan and 59% increase in mean fidelity, whereas the changes from 75% to other levels have less pronounced impact. More interestingly, going from 25% to 0% induces only 4% increase in average makespan and 0.2% increase in average fidelity. This indicates that the the objective weighing does not have significant impact on the schedule quality when the makespan is not 100% prioritized. This could be explained by the fact that both objectives have different units and scales, and the fidelity objective tends to dominate the fitness value, or that the optimizer is biased towards the makespan objective thanks to the "condensation" step of the mutation operator.

For further analysis, we might gather more benchmark runs and perform a sensitivity analysis like ANOVA to determine the percentage of objective variance explained by the changes in the multi-objective ratio.

Table 6.4: Impact of Multi-Objective Ratio on Mean Makespan and Mean Fidelity

Multi-Objective Ratio	MM	MMS	MF	MFS
0.00	287.983	141.965	0.804	0.097
0.25	276.667	139.298	0.802	0.098
0.50	269.528	137.784	0.804	0.094
0.75	256.150	129.147	0.803	0.095
1.00	161.156	86.105	0.506	0.021

MM: Mean Makespan, MMS: Mean Makespan Variance, MF: Mean Fidelity, MFS: Mean Fidelity Variance

6.4.2 Evaluation of NSGA Scheduler

We have evaluated every benchmark scenario with both GA and NSGA optimizers. Below we present the percentage difference between the objective values relative to the GA-optimized schedules. We immediatelly observe significant (37% median) improvements in the maximum makespan in all settings where $\lambda \neq 1.0$. These improvements decrease with the problem size. The makespan objective appears to be on average lower in GA-optimized schedules when we maximally prioritize this singular objective at the cost of fidelity.

Interestingly, the inverse trend appears to be pronounced for the fidelity goal in NSGA-optimized schedules. In all cases where $\lambda \neq 0.0$ we observe a stark (23% median) decrease in average fidelity. Only when $\lambda = 1.0$ we note a slight increase (15% median) in fidelity across schedules. This penalty appears to decrease with problem size.

These patterns indicate that the NSGA optimizer is biased towards generating schedules with lower makespan at the cost of average fidelity. This merits a further investigation into how the dominance order defined for the two objectives impacts the schedule quality. It is also conceivable that the mutation operator which compacts the schedule compounds the bias in successive generations.

We further note that NSGA optimizer has a diffrent performance profile than the GA, with major runtime increase for the smaller problem ranges (0-256/0-64), and moderate increase to a small decrease in the upper registers of jobs/backends ranges. The number of generations is usually 50% lower in NSGA runs. The higher runtime is explained by the resource-intensive operations of non-dominated sorting and Pareto front classification. Those cannot be easily paralellized, and thus for smaller problem

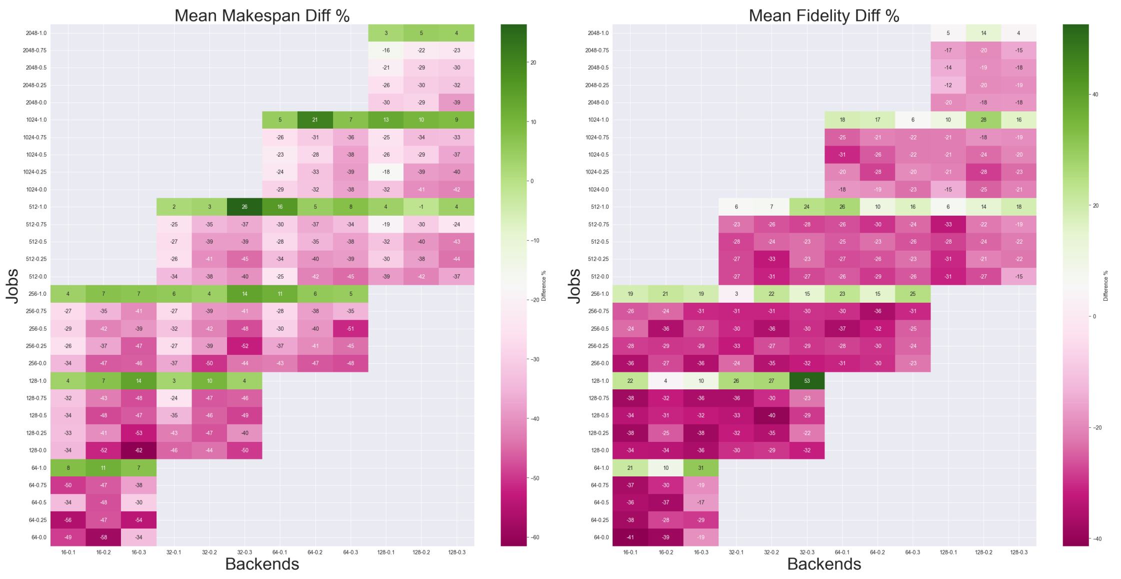

Figure 6.4: NSGA-to-GA objective relative difference

sizes seem to dominate the runtime. The lower number of generations could be an evidence for faster convergence of the Pareto-efficient solutions.

Table 6.5: Comparison of Percent Differences for Various Ranges

Job Range	**Backend Range**	**MRD (%)**	**MGD (%)**	**MMD (%)**	**MFD (%)**
0-256	0-32	290.47	-46.31	-32.14	-20.90
0-256	32-64	112.75	-50.16	-30.80	-19.56
256-512	0-32	17.88	-56.36	-26.21	-18.97
256-512	32-64	30.09	-48.90	-26.47	-17.20
256-512	64-128	40.11	-45.62	-27.51	-17.16
512-2048	32-64	-25.89	-58.49	-22.75	-15.47
512-2048	64-128	-29.80	-55.38	-22.73	-12.95

MRD: Mean Runtime Difference, MGD: Mean Generations Difference, MMD: Mean Makespan Difference, MFD: Mean Fidelity Difference

7 Related Work

This chapter reviews existing forays into job scheduling, quantum resource allocation and quantum cloud computing. We explore notable solutions proposed by the autors of these works.

7.1 Parallel Job Scheduling

Our work is predominantly based on top of the vast body of research into classical job scheduling. We present the papers that inspired our scheduling algorithms and methodology of testing.

7.1.1 Mixed Integer Programming

This thesis fundamentally explores different approaches to generating optimal parallel job schedule with given constraints and optimization goals. [19] investigate the problem of optimizing the classical multi-processor program schedule for the minimized makespan and energy consumption. The authors also enforce the dependency constaints by decomposing programs into DAGs of tasks. They note the NP-hardness of the general case of optimal job scheduling, and propose a bi-objective MIP model that aims to find Pareto-efficient schedules.

The key difference between their work and ours is a stronger focus on limitations of the MIP approach. The authors further refine their model by factoring in the idle time of CPUs where little to no energy is expedited, thus leading to a more accurate representation. They also explore methods of MIP multi-objective scalarization via weighted sum or ϵ-constraint.

Idle Time Consideration

Authors of [22] focus on building a parallel job scheduler that minimize scheduling delays in an Unrelated Parallel Machine Scheduling Problem (UPMSP). Their motivation is to optimize processess such as manufacturing pipelines or JIT compilers, where much like our case, code block dependencies can induce significant gaps in schedule on individual work streams. They propose a model which minimizes the weighted sum of earliness (i.e. the time a job is completed before its deadline) and tardiness (i.e. the time a job is completed after its deadline) of the jobs.

The key novelty is explicit insertion of idle time blocks before jobs, which leads to an improved on-time job completion rate. Our approach only schedules the circuits at earliest possible times and has no concept of expected schedule time, yet if we wanted to add a "second pass" in our scheduling system to further refine the expected start times, this could be a worthwile starting point.

7.1.2 Genetic Algorithms

Flow Shop Scheduling

Authors of [18] have worked on implement a genetic algorithm scheme for flow shop scheduling of an industrial food production process. They represent each chromosome as a list of permutations - the assignments of jobs to particular work-streams. The optimization objective is again the makespan of the schedule.

Their primary focus is generating initial populations in a way that leads to best output of the actual GA. They initialize populations using well-known schemes likes first-come-first-serve, shortest processing time, shortest processing times and others. This "swappable" approach to population generation would complement our work, as besides pre-generating the chromosomes sorted in topological order to satisfy the dependencies, we have not explored how population initialization impacts the quality of results.

Job Shop Scheduling

Authors of [20] have proposed a novel GA model for optimizing the makespan of a DevOps pipeline, itself a job shop scheduling problem. They phrase it as a multi-objective optimization problem, where the objectives are the makespan and machine count allocated for that Continuous Integration (CI) pipeline.

The key design choice of their work is the two-part chromosome representation, where the first part if a topologically-sorted priority list of jobs to schedule, and the second component is the machine availability. Our system relies on a fixed QPU configuraiton, however if were to ever operate on fractions of the total QPU fleet like, algorithm like this could prove useful for picking the most suitable subset of avilable QPUs for that schedule.

7.2 Quantum Resource Allocation

The authors of [7] and [13] explore various quantum resource allocation strategies tailored for the dynamic quantum cloud. Much like our approach, these papers aim to optimize multiple objectives, addressing competing metrics such as the cost of provisioning additional resources to improve quantum circuit execution and decrease the waiting times. The key difference lies in their focus on resource allocation versus resource utilization, as our work focuses primarily on job scheduling.

7.3 Quantum Cloud Analysis

A significant body of work exists in the domain of quantum cloud analysis. For instance, [14] provides a statistical analysis of a dataset comprising over 6,000 quantum job executions. This dataset forms the basis for their execution time estimation model. However, unlike our analysis, their work is confined to studying the dataset, while we focus more on theoretical models and mock data. This narrower focus reduces the generalizability of their findings. Additionally, their model relies on a limited set of features, analyzing only the first circuit in each job. This approach is inadequate for jobs consisting of multiple dependent circuits, as the first circuit may not accurately represent the job as a whole. Nevertheless, their results have proven to be a good basis for us to develop our synthetic datasets of execution and waiting times, fidelities for benchmarking our schedulers.

8 Summary and Conclusion

In this thesis we have defined the scheduling problem for quantum circuits, explored the state-of-the-art scheduling algorithms and applied select few to our unique setting. We have designed, implemented and evaluated a proof-of-concept multi-objective quantum scheduler with pluggable optimizers.

The MIP scheduler serves as a useful mathematical formulation of our constraints - the a-priori knowledge of circuit dependencies and runtime & fidelities can be easily encoded in the model. While we could venture further and formulate other goals like energy consumption, or explore pareto-efficiency, this approach does not scale well enough to justify this effort in context of real-life schedulers.

The GA scheduler, on the other hand, has proven to be a more scalable solution. With careful formulation of the chromosome and genetic operators we've been able to handle up to 10s of QPUs and 100s of circuits. Whereas with GA we need to scalarize our multi-objective fitness function and deduce the weights, the NSGA scheduler offers a native concept of Pareto-dominance, which shows promise for generating more diverse and higher-quality schedules, yet still needs to be fine-tuned and understood in more detail. The novelty of applying "condensation" to the chromosomes in our GA schedules ensures the schedules are more compact, which might serve a foundation for idle time minimization should the scheduler also need to account for that.

We foresee rapid and significant growth in the field of quantum computing in the coming years, akin to the expansion of commercial classical cloud computing. As quantum computing continues to advance, we expect automated scheduling solutions to become indispensable for both cloud providers and end users, with an increasing number of quantum providers integrating such solutions into their public offering. We believe our system serves as a foundational step toward a future of efficient and optimized quantum scheduling, setting the stage for further innovations and progress in this exciting and rapidly growing field.

The full source code and artifacts of our Rust-based GA scheduling system are accessible at `https://github.com/mpraski/genetic-quantum-scheduler`, while the MIP model is available at `https://github.com/manosgior/ScalableQuantumCloudScheduling`.

9 Future Work

In the previous chapters we have demonstrated a working prototype of the multi-objective quantum scheduler. Building upon this foundation, we envision several areas of focus where the functionality and performance of the scheduler can be further improved, leading to a more robust and viable solution for real-world quantum clusters.

9.1 Performance Improvements

In the journey of developing a scalable quantum scheduler we have explored both mathematical MIP models and state-space search based methods like the GA and NSGA. While the MIP model may provide a Pareto-efficient solution provided we choose the right solver and objective scalarization method, it is not scalable to large quantum circuits. The GA and NSGA methods scale much better with the size of the problem. Both approaches could be further optimized for performance and solution fidelity:

9.1.1 MIP Scheduler Implementation

During our development and evaluation we relied on the CBC solver. This project is maintained as an open-source package and well-renowned in the community, however there may exist more performant alternatives like the prioprietary Gurobi solver. We might want to investigate the performance of different solvers combined with scalarization methods like dychotomic or box [19] could be viable directions for making the MIP model viable for larger schedules.

9.1.2 GA Parallelization Strategies

So far we have explored the so-called global parallelization model of our GA, where only the genetic operators are parallelized, yet there is a single global population. There are other parallel GA models offering parallisation at the level of the population itself, where multiple sub-populations are arranged in a topology such as 2D Grid. These sub populations function as their own instances of the genetic algorithm, yet periodically exchange "migrants" between themselves along the routes of the topology. Each such sub-population may converge to a different local minimum of the solution space, while migrants might allow for "polineation" of the local-best solutions from group to group. This might increase diversity of the population and prevent premature convergence as with a single global population only, leading to higher quality results, and perhaps improved scalability for really large problems [1].

9.1.3 Custom Heuristic Algorithm

While we explored well-known directions of job scheduling optimizer design, there are other approaches we might yet explore. With our system being designed with swappable scheduler implementations in mind, we could easily experiment with other heuristics. We could take inspiration from systems which already solved similar problems such as Kubernetes, whose scheduling system has been proven to scale to thousands of nodes and tens of thousands of containers. Kubernetes relies on a multi-phased scheduler, which first filters the jobs eligible for scheduling, then scores them according to multiple criteria (disk/memory requirements, node affinity, port allocation, etc.) and then packs them into bins of "similar" jobs to be then scheduled on the nodes. We might investigate if this heuristic is compatible with our problem and adapt it if possible to also handle circuit dependencies.

Abbreviations

NISQ Noisy Intermediate-Scale Quantum

QPU Quantum Processing Unit

CPU Central Processing Unit

LP Linear Programming

EA Evolutionary Algorithm

GA Genetic Algorithm

NSGA Non-dominated Sorting Genetic Algorithm

DAG Directed Acyclic Graph

MIP Mixed Integer (Linear) Programming

DSL Domain Specific Language

CBC Coin-or branch and cut solver

OpenMP Open Multi-Processing

UPMSP Unrelated Parallel Machine Scheduling Problem

CI Continuous Integration

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

Bibliography

[1] E. Alba and M. Tomassini. "Parallelism and evolutionary algorithms." In: *IEEE Transactions on Evolutionary Computation* 6.5 (Dec. 2002), pp. 443–462. DOI: 10.1109/TEVC.2002.800880.

[2] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007. Chap. 2. ISBN: 9780262255905.

[3] G. Diaz et Al D. Barrala F. F. Cardamab. "Review of Distributed Quantum Computing. From single QPU to High Performance Quantum Computing." In: (2024), pp. 3–10.

[4] W. Pedrycz D. Sudholt J. Kacprzyk. "Parallel evolutionary algorithms." In: *Springer Handbook of Computational Intelligence* (2015), pp. 929–959.

[5] *IBM Quantum*. 2023. URL: https://quantum.ibm.com.

[6] *Introduction to the quantum programming language Q#*. Oct. 2024. URL: https://learn.microsoft.com/en-us/azure/quantum/qsharp-overview.

[7] R. Kaewpuang, M. X., D. Niyato, H. Y., Z. Xiong, and J. Kang. *Stochastic Qubit Resource Allocation for Quantum Cloud Computing*. 2023. arXiv: 2210.12343 [quant-ph]. URL: https://arxiv.org/abs/2210.12343.

[8] G. Kalai. "Linear programming, the simplex algorithm and simple polytopes." In: *Mathematical Programming* 79.1 (1997), pp. 217–233. DOI: 10.1007/BF02614318. URL: https://doi.org/10.1007/BF02614318.

[9] D. Lei. "Multi-objective production scheduling: a survey." In: *The International Journal of Advanced Manufacturing Technology* 4 (Aug. 2008), pp. 926–938.

[10] N. Matsakis. *Rayon: data parallelism in Rust*. Dec. 2015. URL: https://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust.

[11] A. Mishra. "Understanding Branch and Bound in Optimization Problems." In: Walmart Global Tech Blog (July 2021). URL: https://medium.com/walmartglobaltech/understanding-branch-and-bound-in-optimization-problems-d8117da0e2c5.

[12] S. A. Murad, A. J. M. Muzahid, Z. R. M. Azmi, M. I. Hoque, and M. Kowsher. "A review on job scheduling technique in cloud computing and priority rule based intelligent framework." In: *Journal of King Saud University - Computer and Information Sciences 34.6* (Mar. 2023), pp. 2306–2330. ISSN: 1319-1578. DOI: `https://doi.org/10.1016/j.jksuci.2022.03.027`.

[13] N. Ngoenriang, M. Xu, S. Supittayapornpong, D. Niyato, H. Yu, Xuemin, and Shen. *Optimal Stochastic Resource Allocation for Distributed Quantum Computing*. 2022. arXiv: `2210.02886 [cs.DC]`. URL: `https://arxiv.org/abs/2210.02886`.

[14] G. S. Ravi, K. N. Smith, P. Gokhale, and F. T. Chong. "Quantum Computing in the Cloud: Analyzing job and machine characteristics." In: *2021 IEEE International Symposium on Workload Characterization (IISWC)* (2021), pp. 36–42.

[15] G. S. Ravi, K. N. Smith, P. Murali, and F. T. Chong. "Adaptive job and resource management for the growing quantum cloud." In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)* (2021), pp. 36–42.

[16] G. Van Rossum and F.L. Drake. *Python 3 Reference Manual: (Python Documentation Manual Part 2)*. Documentation for Python. CreateSpace Independent Publishing Platform, 2009. ISBN: 9781441412690. URL: `https://books.google.pl/books?id=KIybQQAACAAJ`.

[17] A. Somani and D. P. Singh. "Parallel Genetic Algorithm for solving Job-Shop Scheduling Problem Using Topological Sort." In: *IEEE International Conference on Advances in Engineering and Technology Research (ICAETR - 2014)* (2014), pp. 3–7.

[18] A. Sosimi, F. Oladimeji Ogunwolu, and T. Adegbola. "A Makespan Optimization Scheme for NP-Hard Gari Processing Job Scheduling Using Improved Genetic Algorithm." In: *Journal of Industrial Engineering* (2013), pp. 1–6.

[19] R. Stewart, A. Raitha, and O. Sinnen. "Optimising makespan and energy consumption in task scheduling for parallel systems." In: *Computers and Operations Research* 154 (2023), pp. 1–6.

[20] B. Tagtekin, M. U. Öztürk, and M. K. Sezer. "A Case Study: Using Genetic Algorithm for Job Scheduling Problem." In: *Huawei Turkey R and D Center, Istanbul, Turkey* (2021), pp. 2–5.

[21] PuLP documentation team. "Optimization with PuLP." In: (2024). URL: `https://coin-or.github.io/pulp/`.

[22] C. Y. Tsai and Y. C. Wang. "A Mixed Integer Linear Programming Solution for Insertion of Idle Time in the Jobs Scheduling." In: *Proceedings of the 2nd International Conference on Soft Computing in Information Communication Technology*. Atlantis Press, May 2014, pp. 54–56. ISBN: 978-94-6252-014-1. DOI: `10.2991/scict-14.2014.13`. URL: `https://doi.org/10.2991/scict-14.2014.13`.

[23] M. Voorneveld. "Characterization of Pareto dominance." In: *Operations Research Letters* 31.1 (2003), pp. 7–11. ISSN: 0167-6377. DOI: `https://doi.org/10.1016/S0167-6377(02)00189-X`. URL: `https://www.sciencedirect.com/science/article/pii/S016763770200189X`.

[24] X. Xu and Z. Shi. "Multi-objective based spectral unmixing for hyperspectral images." In: *ISPRS Journal of Photogrammetry and Remote Sensing* 124 (2017), pp. 56–60. ISSN: 0924-2716. DOI: `https://doi.org/10.1016/j.isprsjprs.2016.12.010`.