Towards Distributed Quantum Algorithms

Pablo Andres-Martinez

Master of Science by Research

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

Acknowledgements

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pablo Andres-Martinez*)

Table of Contents

Chapter 1

Introduction

Chapter 2

Quantum Computing: A brief overview

Quantum computing aims to take advantage of quantum mechanics to speed-up computations. There are many examples of problems that, although solvable by a standard (classical) computer, the time it would take to compute them is unreasonable in practice, regardless how large or fast your classical computer is. Some of these *intractable problems* can be solved efficiently on a quantum computer. This exhibits the vast gap in computational power between classical and quantum computers. Well known examples of such problems are:

- *Factorisation of large numbers*: In classical computers, all known factorisation algorithms take exponential time with respect to the input size. It is strongly believed that there is no way a classical computer can solve the problem efficiently – in fact, we are so confident about it that most widely used encryption systems, like RSA, rely on this. However, quantum computers are capable of solving the problem in polynomial time (i.e. making it tractable), using Shor's algorithm (Shor, 1999).

- *Unstructured search*: The aim is to perform a brute-force search (i.e. requiring no prior knowledge about the search space) over N data-points. Classical computers have no other option than testing each data-point, so the time they take to perform the search is proportional to N. With a quantum computer, using Grover's algorithm (Grover, 1996), the search is done in time proportional to $\sqrt{N}$.

Besides, in May of the present year, Raz and Tal (2018) gave formal proof of the existence of a large family of problems that a classical computer may never solve in polynomial time, but are solvable in polynomial time on a quantum one. Nevertheless,

all of these results are theoretical in nature, and giving experimental evidence of this gap in computational power is a highly active area of research, known as *quantum supremacy*.

Quantum computing would be very valuable in many areas of research that deal with problems that are intractable on classical computers. Some of the main applications that have been discussed in the literature are:

- *Chemistry, medicine and material sciences*: Calculating molecular properties on complex systems is extremely demanding for classical computers. However, polynomial algorithms for this problem are known for quantum computers (Lanyon et al., 2010).

- *Machine learning*: Finding patterns in a large pool of data is the essence of machine learning. Multiple quantum algorithms have been shown to be able to detect patterns that are believed not to be efficiently attainable classically (Biamonte et al., 2017).

- *Engineering*: Optimization and search problems are common in almost every area of engineering. Quantum computers are particularly well suited for these tasks, with Grover's algorithm being an obvious example.

For any of these applications we will require large scale quantum computers. Due to the obstacles in the way of building a large mainframe quantum computer (see §2.2.1), some authors have advocated the alternative of building a *quantum multicomputer*: a grid of small quantum computer units that cooperate in performing an overall computation (Van Meter et al., 2010). In the present work, particularly in Chapter 4, we contribute to this perspective, providing a method for efficiently distributing any quantum program originally designed for a monolithic quantum computer.

2.1 The principles of quantum computing

The advantages of using quantum mechanics to perform computations come down to the following three principles:

- *Superposition*: In classical computing, the unit of information is the *bit*, which may take one of two values: 0 or 1. In quantum computing, the *bit*'s counterpart

is the *qubit*, whose value may be *any linear combination* of the 0 state and the 1 state, known as a *superposition*, and usually written as:

$$|qubit\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers that must satisfy: $\alpha^2 + \beta^2 = 1$.

A popular analogy of a qubit's superposition is a coin spinning[1]: the classical states (0 and 1) are *heads* and *tails*, but when the coin is spinning, its state is neither of them. If we knew exactly how the coin was spinning, we would be able to describe the probability distribution of seeing heads or tails when it stops; these would be our α^2 and β^2 values. Besides, we may *measure* a qubit, and doing so corresponds in our analogy to abruptly stopping the coin, then checking if it is *heads* or *tails*. The essential aspect of this analogy is that, before measurement, the *qubit*'s state is neither 0 nor 1. Through certain operations – that would correspond to altering the axis of spin of the coin –, we may change the coefficients α and β of the superposition.

Interestingly, in quantum computing, we encode input and read output (after measurement) as standard classical binary strings, and thus, for input/output we use as many qubits as bits would be required. What superposition provides is the ability to – during mid-computation – maintain a superposition of all potential solutions to the problem, and update all of them simultaneously with a single operation to the qubits. In some sense, superposition allows us to explore multiple choices/paths of the computation, using only the resources required to explore a single one of those paths. And the number of paths we can explore simulatenously can be up to exponential, as a collection of N qubits may be in a state of superposition of all the possible 2^N classical states.

- *Interference*: As we just discussed, superposition gives us the ability to simultaneously explore different paths to solve a problem. However, in the end we will need to measure the qubits – stop the coins – and the result will be intrinsically random. For quantum computing to be any better than a probabilistic classical computer, we require the ability to prune the paths that have led to a result we do not want. This is precisely what *interference* provides: some operations on the qubits may make different classical states in the superposition cancel each other

[1] Note this is just an analogy, and while a coin spinning can be perfectly modelled using classical physics, a qubit can not.

out. Interference is at the core of any speed-up achieved by a quantum algorithm, and taking advantage of it is the main challenge when designing quantum algorithms.

- *Entanglement*: Quantum mechanics allows the possibility of having a pair of qubits a and b in a superposition such as:

$$|a,b\rangle = \frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle$$

This implies that, when we measure the qubits, we may either read $a = 0, b = 0$ or $a = 1, b = 1$ as outcome, never $a \neq b$ (the coefficients for $|0,1\rangle$ and $|1,0\rangle$ are both 0). Then, what happens if we only measure a? In this particular case, we would also know b's outcome, without measuring it. In short, acting on one qubit has an instantaneous effect on the other. Whenever a group of qubits exhibits this property, we say they are *entangled*. Entanglement holds regardless how far apart a is from b; for instance, they could be on two different quantum processing units of a distributed grid. Indeed, entanglement will be key in our discussion of distributed quantum algorithms, and we explain how to use it to perform non-local operations in §3.2.

2.2 Building quantum computers

Physicists have come up with different ways of realising qubits in labs. The key idea is to find a physical system that displays non-classical behaviour, and put it under the appropriate circumstances so we can manage its quantum properties, but noise in the environment may not interfere with these. Van Meter and Devitt (2016) give an excellent survey of the state of the art of quantum architectures. Among them, the three most developed are:

- *Quantum optics*: The state of a qubit is represented in the properties of photons, for instance, their polarization (Kok et al., 2007). A great advantage of this technology is that photons can be easily sent over long distances, while preserving the quantum state. Thus, protocols in quantum information that heavily rely on communication, such as Quantum Key Distribution (Shor and Preskill, 2000), are usually discussed and experimented using quantum optics. The downside of this technology is that it is very difficult to make photons interact, which is required for multi-qubit operations.

- *Ion-traps*: Each qubit is embodied as an ion, confined inside a chamber by means of an electric or magnetic fields. The qubit is acted upon by hitting the ion with electromagnetic pulses (e.g. laser light or microwave radiation). Groups of experimentalists have proposed how to scale up the technology (Weidt et al., 2016), and are currently building prototypes.

- *Superconductors*: Small circuits, similar to classical electrical circuits, are cooled down to near absolute zero so the quantum interactions of electrons are not obscured by other perturbations. Different parts of the circuit encode different qubits, which can be acted upon by applying electric potentials. One of the main advantages of this technology is that the technology required for building the chips is fairly similar to the one for classical computer chips. This technology seems to be the most experimentally developed. In fact, using it, both IBM and Intel have already built small generic-purpose quantum computers of 17-20 qubits.

However, for quantum computers to be useful in real world applications, their qubit count should raise up, at the very least, one order of magnitude. And, unfortunately, increasing the amount of qubits in a quantum computer is particularly difficult, due to some challenes we will now discuss.

2.2.1 Scalability challenges

There are two main challenges to overcome in order to build large scale quantum computers:

- *Decoherence*: In §2.1 we discussed the importance of having superposition in quantum computing, and we compared a qubit in superposition with a coin spinning. For similar reasons why a coin spinning will eventually stop, a qubit in superposition will eventually degenerate into a classical state (i.e. either $|0\rangle$ or $|1\rangle$): Physical systems have a tendency towards the state at which they are most stable, for the coin it is laying flat, for the qubit it is losing its superposition. This phenomenon is known as *decoherence*. Experimentalists attempt to increase the time it takes for the state of the qubit to degenerate[2]. Decoherence constitutes

[2]A fundamentally different approach, *anyonic* (a.k.a. topological) quantum computing, has been proposed to avoid the problem of decoherence altogether, as it would use physical systems that, theoretically, can be completely protected against decoherence (Nayak et al., 2008). Although promising, currently this proposal has little experimental underpinning, and it is not regarded as attainable in the near future.

the main constraint to scalability of quantum computers, as it dictates the lifespan of qubits, limiting the number of operations that can be applied in a single program.

Certainly, the state of bits also degenerates in classical computers. However, in their case this is easier to account for: we can keep monitoring the bits, and make sure to correct any unwanted change. This is not so simple in quantum computers, as monitoring a qubit would require *measuring* it, and that destroys any quantum superposition. Nevertheless, it is still to some extent possible to protect our quantum state from errors – either due to decoherence or imperfect hardware – through quantum error correction routines (Knill et al., 2000). This is a very active area of research, and it will be key for the implementation of reliable large scale quantum computers.

- *Connectivity*: In order to run complex computations on qubits, we will need to be able to apply multi-qubit operations on any subset of the computer's qubits. However, it is not realistic to expect that quantum computers will have fast connectivity between all qubits, for instance due to spatial separation of these in the hardware. In classical systems, this problem is solved by a memory hierarchy, with a ceaseless flow of data going up and down of it, from main memory to registers (where computation is performed) and back. The memory hierarchy model works because data can stay idly in main memory while computation on the registers is carried out. However, in quantum computers we must avoid qubits being idle, as decoherence prevents the existence long-lasting memory. An alternative found in classical computers is to distribute the computation across different processing units each having its own local memory, which they use intensively, and communicating – through message passing – as little as possible. In §3.3, we discuss an abstract distributed quantum architecture in detail.

2.2.2 Models of computation

In this section, we give a brief introduction to some models of quantum computation relevant to the this thesis.

- *Circuit model*: Any operation on n qubits – as long as measurement (i.e. destruction of information) is not involved – can be represented as square matrix on complex numbers, of dimension 2^n. These matrices are always unitary, which

means that a matrix U satisfies $UU^\dagger = I = U^\dagger U$, where I is the identity matrix and $A^\dagger$ is the conjugate transpose of A. Essentially, unitarity ensures that any operation on qubits can be reversed (i.e. undone), reason why this model is sometimes called the reversible model. Multiplying matrices AB corresponds to applying the operation described by B first, then A, on the same qubits. Application of two operations on disjoint set of qubits corresponds to the Kronecker product of the matrices $A \otimes B$. The fundamental concept is that any matrix can be represented as a product of other matrices, so we may decompose any operation into smaller building blocks: quantum gates.

Qubits are pictured as wires to which quantum gates are applied, similarly to a classical digital circuit. The set of quantum gates used is dependent on the architecture. There exist an (uncountable) infinite amount of different quantum operations, but a small finite set of them is enough to approximate any of them up to a desired error factor. The most common choice of such a universal gate-set is *Clifford+T*, which contains six one-qubit gates, and a single two-qubit gate. The depiction of the gates in that set, and some of their most important properties are shown in Figures **TODO**. Circuits are read from left to right.

TODO: A figure depicting, and giving the matrix of, each of CNOT, X, Y, Z, H, S and T. Figures showing HXH = Z; XX = I, ZZ = I, YY = I, HH = I; SS = Z; TT = S; H2 CNOT H2 = NOTC, and also their algebraic notation.

The CNOT gate (Figure **??**) is particularly interesting. The qubit where the filled dot is acts as the 'control', and the qubit with $\oplus$ acts as 'target'. Whenever the control is $|0\rangle$, no change is made in either of the qubits; but if it is $|1\rangle$, an X gate (Figure **??**) is applied to the target, flipping the state of the qubit. This works in any superposition, so if in

$$|c,t\rangle = \alpha|0,0\rangle + \beta|0,1\rangle + \gamma|1,0\rangle + \delta|1,1\rangle$$

$|c\rangle$ were acting as control and $|t\rangle$ as target, the outcome would be:

$$CNOT \cdot |c,t\rangle = \alpha|0,0\rangle + \beta|0,1\rangle + \gamma|1,1\rangle + \delta|1,0\rangle$$

TODO: A figure showing an abstract quantum circuit.

- *MBQC model*: The initials stand for Measurement Based Quantum Computing. Unlike the circuit model, where measurements are done at the very end of the

circuit, MBQC carries out computations by means of repeatedly measuring an initially entangled resource. The process, sketched in Figure **??** can be thought of as sculpting a rock. The initial entangled resource, which is a collection of entangled qubits forming a lattice structure, would be the rock. By measuring some qubits in the lattice – hitting the rock with a chisel – we remove some of the excess qubits, changing the overall state in the process. The outcome of measurements is probabilistic so, in order to provide deterministic computation, we must apply corrections on the neighbouring qubits whenever the measurement outcome deviated from the desired result. After multiple iterations of measurements and corrections, we end up with a set of qubits encoding the result. In this model, the input is incorporated into the lattice at the beginning of the process.

TODO: Figure from slides

In this way, any computation may be performed by applying 1-qubit measurements and 1-qubit correcting gates (controlled by classical signals). The initial resource state contains all the entanglement that is required, which may be prepared experimentally through multi-qubit interactions, such as Ising interactions (Raussendorf and Briegel, 2001), which are within our experimental capabilities. This model deals with the connectivity problem by applying a single operation involving *all* the qubits at the beginning of the process, then only requiring cheap single qubit operations for the rest of the computation. The main drawback of MBQC is the large amount of qubits that are required for even the simplest of operations. The MBQC model was proposed for the first time by Raussendorf and Briegel (Raussendorf and Briegel, 2001) under the name of *one-way quantum computer*, which highlights its main difference with the circuit (reversible) approach.

- *Distributed model*: We may find a balance between the circuit model and MBQC. In it, multiple small quantum processing units (QPUs) would run fragments of the overall circuit. Communication is achieved through a shared entangled resource, reminiscent of the MBQC approach. This model has been discussed in detail in the literature (Van Meter et al., 2010) and it is at the core of the main project from the Networked Quantum Information Technologies Hub (NQIT)[3]. In §3.3, we discuss an abstract distributed quantum architecture in detail.

[3] A project supported by the UK National Quantum Technology program, aiming to provide scalable quantum computing.

2.3 Programming on quantum computers

As of today, most quantum programming languages are merely high level circuit descriptors: They provide the means to define circuits gate by gate, or build them up from combinations of smaller circuits. In this category fall all the well-known languages, such as *QCL* (Ömer, 2003) (imperative paradigm, and one of the first quantum programming languages ever implemented), *Q#* (Svore et al., 2018) (imperative, designed by Microsoft), and *Quipper* (Green et al., 2013) (functional, built on top of Haskell).

Besides, there are attempts at designing quantum programming languages that are completely hardware agnostic, meaning they aim to describe the computation, rather than a particular circuit that implements it. Examples of these are the different attempts at defining a quantum lambda calculus, for instance the ones by Van Tonder (2004) or Díaz-Caro (2017). However, these are not particularly programmer friendly, as they are generally quite verbose.

Most of the literature on quantum algorithms describes these by explicitly giving circuits that implement them. Fortunately, there is a constructive procedure, given by the Solovay-Kitaev theorem –of which Dawson and Nielsen (2005) give a good introductory review–, that takes any circuit and a choice of universal gate-set and outputs an efficient equivalent circuit using only those gates. Hence, programmers do not need to worry about the gates they are using when describing their circuits.

Unfortunately, the fact that algorithms are almost exclusively defined in the circuit model implies that other models of quantum computing (introduced in §2.2.2) are disregarded by a large portion of the community. In order to make other models of computation accessible, we need to provide automated procedures for transforming algorithms from the circuit models to these (and vice versa). Work has been done on the transformation from circuit to MBQC and backwards, the latter being the most challenging (Duncan and Perdrix, 2010). However, there is little amount of literature describing how to go from the circuit model to the distributed model. In §3.2 we give an overview of the existent work on that aspect, and identify the gap on the literature we aim to answer in this thesis.

2.4 Summary

As a wrap up, here are the key concepts to keep in mind while reading the rest of this thesis:

- Quantum computers provide a computing power well beyond the capabilities of classical computers, which would be exploitable in many areas of science.
- Small quantum computers are already available.
- Scaling up is a challenging problem due to: *decoherence*, which may be overcome by the joint effort of error-correction, physics and engineering communities; and *connectivity*, which may be solved using distributed architectures.
- There is practically no programming support for distributed architectures.

Chapter 3

Distributed Quantum Computing

As we discussed in § 2.2, there are different approaches on how to build quantum computers. Now that many of these have been experimentally demonstrated, having built small quantum computers, the question of how to scale up is increasingly relevant. This has led to the proposal of distributed architectures (Van Meter and Devitt, 2016).

In classical computing, an standard example of a distributed computer is the Non-Uniform Memory Access (NUMA) architecture: A system of independent computing nodes, each having its own local memory. In order to collaborate to perform an overall computation, the different nodes will need to communicate. In NUMA, they do so by accessing each other's memory. While nodes can manage their own local memory efficiently, accessing another node's memory is slow. Hence, we always attempt to minimise the amount of communication between nodes. A distributed quantum computer would follow the same principles, where each quantum processing unit (QPU) would own a collection of qubits (its local memory) and may access another QPU's qubits at the cost of some overhead, using entanglement.

3.1 Communication through entanglement

For a QPU to be able to access another's qubit, we must provide them with some sort of communication channel. Simply using a classical channel (sending bits) is not acceptable, as the whole point of representing the state of the computation in qubits is that they may be in a superposition of classical states, which would take up to an exponential amount of space and processing on classical bits. We could consider physically transporting the system that encodes the qubit from one QPU to another, and while that is certainly possible with photons, in general it is not feasible to have a channel that is

both fast and protects well against information loss due to decoherence.

In § 2.1, we explained it was possible to affect a distant qubit by acting on another qubit with which it was entangled. We wish to exploit this property in order to allow a QPU to query another's QPU qubit. There are different levels of how strong a pair of qubits is entangled – intuitively, how much they affect each other. This is often formalised as a metric on the correlation between the qubits possible measurement outcomes; for instance, the pair of qubits $\frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle$ is said to be *maximally entangled*, as the possible measurement outcomes are exclusively either $|0,0\rangle$ or $|1,1\rangle$, always matching in both qubits[1]. Naturally we will want to take advantage of entanglement in its strongest form, and so we will make use pairs of qubits entangled in this particular maximally entangled state. This qubit pair configuration is generally known as a Bell state, and Figure **??** shows how to prepare it.

TODO: fig:bell, show the circuit to make a bell state, how it is represented usually as a wire C and its Diract state.

An interesting property of the Bell state is shown in Figure **??**: if a quantum gate, whose matrix representation is symmetric, is applied to one of the qubits, it is the same as if the gate was applied to the other qubit. In some sense, the gate can 'slide' through the entanglement, like beads on a string; as if the entangled state were a curved wire, connecting the pair of qubits. Hopefully, this serves as a first intuition of how Bell states can be used to communicate quantum information.

TODO: fig:sliding, show how a H gate can slide through an ebit.

So far, we have explained how to generate a Bell state in a machine (as in Figure **??**). However, what we aim for is that two different QPUs each own one of the qubits from the Bell state. The challenge is then to send the qubits to the two QPUs, while preserving their entangled state. Fortunately, the problem of sharing a Bell state between two parties has already been solved by the *entanglement distillation* protocol (Bennett et al., 1996), which ensures the Bell pair is transmitted to their destination with an arbitrarily small error factor. We will now give a brief description of this protocol.

3.1.1 Entanglement distillation

TODO: This section

[1] In total, there are four *maximally entangled* states of a pair of qubits: $\frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle$ and $\frac{1}{\sqrt{2}}|0,0\rangle - \frac{1}{\sqrt{2}}|1,1\rangle$ give perfect correlation, while $\frac{1}{\sqrt{2}}|0,1\rangle + \frac{1}{\sqrt{2}}|1,0\rangle$ and $\frac{1}{\sqrt{2}}|0,1\rangle - \frac{1}{\sqrt{2}}|1,0\rangle$ give perfect anti-correlation

Sending an arbitrary quantum state across a channel is difficult, we could do it either fast or reliably, but not both. But if you do know what you are sending, you just send a lot of them.

Non-fidelity. If the ebit is not exactly in a state of the form $\alpha|0,0\rangle + \beta|1,1\rangle$ (i.e. it has a non-zero coefficient for $|0,1\rangle$ or $|1,0\rangle$)

Often in distributed quantum computing literature, a Bell state shared by two QPUs is known as an *ebit* (entangled-bit). We will also use this convention, and refer to each of the two qubits in the Bell state as the two *halves* of the ebit.

3.2 Distributing circuits

Up to this point, we have not explained how ebits are used to allow a QPU to peek into another's memory. We will explain it in this section, following the proposal of Yimsiriwattana and Lomonaco Jr (2004). Later on, in § 4.1, we will extend this work with our own contributions.

We aim to split a given circuit and distribute the fragments across multiple QPUs. The gates that should operate over qubits on different QPUs are known as *non-local* gates, and that just means that one of the two QPUs will access another's memory in order to execute the gate. Given that any circuit can be converted to Clifford+T circuit – using, for instance, Solovay-Kitaev's algorithm – The only gate that operates on more than one qubit will be the CNOT. Hence, we only need to understand how CNOTs can executed non-locally, in order to have universal distributed computation.

TODO: A figure with a simple circuit and dashed lines identifying how we want to split it, and highlighting a non-local CNOT.

The construction we will use is a slight variation of what was proposed by Yimsiriwattana and Lomonaco Jr (2004), and it is shown in Figures **TODO**. In principle, we will use an *ebit* per non-local CNOT. Each half of the ebit is sent to a different block. We will call the block that holds the target qubit (the one with a $\oplus$) the 'target block' and similarly for the control qubit.

TODO: showing the overall scheme, with cat-entangler and cat-disentangler as black boxes.

We must first apply what the authors refer to as the *cat-entangler* (Figure **??**), which 'copies'[2] the state of the control qubit into the ebit half in the target block.

[2]Note that there is no such thing as 'copying' a quantum state (due to the non-cloning theorem). What we mean here by 'copying' is generating, from $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the state $|\psi\rangle = \alpha|0,0\rangle + \beta|1,1\rangle$

To do so, the ebit half in the control block is measured (and thus destroyed), and the outcome is used to correct the other half, in the same spirit as in the MBQC model (see § 2.2.2). Notice that the only information physically crossing the boundary between blocks is the *classical* outcome of the measurement (a bit, either 0 or 1).

TODO: Show the circuit for the cat-entangler and the resulting state

Then, the CNOT gate may be applied between the ebit half in the target block and the target qubit itself. After it, the *cat-disentangler* must be applied (Figure **??**), which simply destroys – with a measurement – the remaining ebit half and then corrects the control qubit, so the randomness of the measurement is counteracted. Once again, only classical information crosses the boundary.

TODO: Show the circuit for the cat-disentangler and the resulting state

In this way, we have implemented a non-local CNOT gate using one ebit and two classical single bit messages between blocks. However, the true advantage of this approach is attained when multiple non-local CNOTs are implementing using a single ebit. The original paper (Yimsiriwattana and Lomonaco Jr, 2004) proposed one way of doing this. We will extend it on § 4.1. What they propose simply consists in realising that, after the cat-entangler is applied, any number of CNOTs that are controlled by the same qubit, and that target different places in a single target block, may all be implemented by using the ebit as control, as shown in Figure **??**.

TODO: Show the circuit for many CNOTs, now with cat-entangler and everything

Now, depending of how we choose to partition the circuit, there will be different groups of CNOTs that we may be able to implement using a single ebit. We will then wish to find the partition that requires the least amount of ebits to implement all of its CNOT gates. This optimization problem is not discussed in the original paper, nor in any other work, as far as we know. It will be our main contribution in this thesis, along with an extension of the results just explained, both found in Chapter 4.

3.3 Distributed quantum architectures

A distributed quantum computer will have multiple quantum processing units (QPUs), each managing a small collection of qubits as their local memory. It should also have a subsystem specialised in generating and sharing ebits. As we discussed in § 3.1.1, there is a compromise between the quality of the ebit – the quality of the communication channel – and the time it takes to prepare it. Fortunately, Cirac et al. (1999) showed

which is fundamentally different from $|\psi,\psi\rangle = \alpha^2|0,0\rangle + \alpha\beta|0,1\rangle + \alpha\beta|1,0\rangle + \beta^2|1,1\rangle$.

that efficient distributed quantum computation using noisy ebits is feasible. Still, the generation of *ebits* remains the main bottleneck of the architecture, and thus we will want to minimise the required ebit count as much as possible.

TODO Authors such as Van Meter et al. (2010) have discussed the experimental construction of a similar architecture. It is a multicomputer that uses photons for communication across QPUs, entangling stuff.

Chapter 4

Automated Distribution of Quantum Algorithms

4.1 Implementing non-local CNOT gates

In § 3.2, we explained the proposal by Yimsiriwattana and Lomonaco Jr (2004) of how to implent a non-local CNOT. We will now extend their results.

The first thing to notice is that the CNOT gates need not immediately follow one another. Some of the 1-qubit gates from the `Clifford+T` set commute with the CNOT. We may transform a circuit so the controls of different CNOT gates are brought together. Some of the gates do not commute, but they can be interchanged with the CNOT if an additional 1-qubit gate is added. All of these transformations are shown in Figures **TODO**. These can be checked by calculating the corresponding matrices and verifying they match.

TODO: Figures of how to push gates through control.

The second improvement comes by realising that the trick used to implement multiple CNOT gates controlled by the same wire can also be applied if multiple CNOT gates target the same qubit. The derivation is shown in Figure **??**, which uses some of the properties listed in § 2.2.2.

TODO: CNOTtargetProof

4.2 Finding an efficient distribution

In this section we explain how we search for a suitable partition of the circuit. As discussed in § 3.3, our notion of an optimal distribution of a given circuit is one such

Table 4.1: Correspondence between the graph partitioning problem and the efficient distribution of quantum circuits.

Graph partitioning	*Efficient distribution*
Vertices	Wires
Edges	CNOT gates
Partitioned graph	Distributed circuit
Subgraph	QPU
Min. cut edges	Min. non-local gates
Uniform subgraph size	Load balance

that:

- *Minimal amount of quantum communication* between the QPUs as possible, i.e it requires as little number of ebits as possible. In comparison, message passing of classical bits is considered negligible.
- The *QPUs should be load-balanced* up to a tolerance margin. Our notion of load-balance is that the different QPUs have a similar number of qubits from the original circuit assigned to them, i.e. we care about having a uniform breadth of local circuits. A uniform depth would be desirable, however, the distributed circuit depth is inherited from the original circuit, as none of our distribution techniques change the depth in a significant way. Hence, we will assume that circuit depth reduction methods, such as the one described by da Silva et al. (2013), have already been applied on the input circuit (and they could be also applied to each QPU's local circuit after distribution). As circuit depth is not something we aim to optimise, we consider the cost of local gates negligible.

The problem at hand is similar to the $(k, \varepsilon$ *graph partitioning* problem, where a graph partition in k subgraphs has to be found, minimising the number of *cut edges* – edges that have their incident vertices in different blocks – and ensuring the number of vertices in each block is within the ε tolerance factor: $(1 \pm \varepsilon)\frac{N}{k}$, where N is the total number of vertices in the graph. In Table **??** we list the matches between these two problems.

But there is a caveat. If we use graph partitioning naively, we will not be exploiting the fact that multiple CNOT gates may be implemented using a single ebit. In what follows, we will explain how to make use of *hypergraph* partitioning, instead of simple

graph partitioning, to account for this aspect. A more detailed review of hypergraph partition is given in Appendix A, here we summarise the key concepts:

- Hypergraphs extend graphs to accommodate edges that may have more than two incident vertices. More formally, a hypergraph is a pair of sets (V,H), where V is the set of vertices and $H \subseteq 2^V$ is the *collection*[1] of hyperedges. Each hyperedge is represented as the subset of vertices from V it connects. We will not consider any notion of directionality, i.e. all vertices of a hyperedge play the same role.

- Hypergraph partitioning follows the same premise as graph partitioning. The user provides a hypergraph and the two parameters (k,ε), which have the exact same meaning as before. What the problem now attempts to minimise is a metric known as $\lambda-1$, which is defined as follows: given a partition of the hypergraph, the function $\lambda: H \to \mathbb{N}$ pairs each hyperedge with the number of different *blocks*[2] its vertices are in. Then, $\lambda-1 = \sum_{h\in H} \lambda(h) - 1$ provides a measure of not only how many hyperedges are 'cut' but also across how many blocks they are connecting[3].

In the following subsections, we explain how hypergraph partitioning can be used to find the best distribution of a circuit. First, we only use the implementation of non-local gates described by Yimsiriwattana and Lomonaco Jr (2004) and reviewed in § 3.2. Later on, we extend the algorithm to include the improvements we have proposed in § 4.1.

4.2.1 Vanilla algorithm

The key concept is how to use hyperedges to represent a collection of CNOT gates that, in case of being non-local, they could all be implemented using a single ebit. As we already explained in § 3.2, a single ebit may be used if all the CNOT gates are controlled by the same wire, and there are no other gates in between their connections to that wire, as in Figure **??**). Therefore, for such a collection of CNOT gates, we will create a single hyperedge whose vertices correspond to the controlling wire and each of

[1] We will allow multiple hyperedges connecting the same vertices, in the same way as multigraphs allow multiple edges across any pair of edges.

[2] The term *block* is often used to refer to each of the sub-hypergraphs that comprise the hypergraph partition. It is the term we will use throughout this thesis.

[3] Simply minimising the number of hyperedges 'cut' is also an often used approach, but it is not as useful for our problem.

the different wires the CNOT gates target. The algorithm is described in Algorithm 4.1. It should be noticed that all CNOT gates from the input circuit are represented once and only once in the hypergraph.

Algorithm 4.1: Builds the hypergraph of a given circuit. H may contain multiple hyperedges connecting the same vertices.

```
1  input: circuit
2  output: (V,H)
3  begin
4    V ← ∅
5    H ← ∅
6    hedge ← ∅
7    foreach wire in circuit do
8      V ← V + {wire}
9      H ← H + {hedge}
10     hedge ← {wire}
11     foreach gate in wire do
12       if gate == CNOT and controlOf(gate) == wire then
13         hedge ← hedge + {targetOf(gate)}
14       else
15         H ← H + {hedge}
16         hedge ← {wire}
17 end
```

We then solve the hypergraph partitioning problem (see Appendix A) on the resulting hypergraph. Once an efficient partition of the hypergraph is obtained, we map the partition back to the circuit, distributing it: The way the vertices are assigned in the blocks dictates how the corresponding wires are allocated to the different QPUs. Whenever a hyperedge has all of its vertices in the same QPU (i.e. it is not cut), all the CNOT gates it represents are implemented locally in that QPU, so no ebit is required (Figure **??**a). If a hyperedge is cut once, some of the wires targeted by the CNOTs will appear in the same QPU as the control wire, so those CNOTs will be implemented locally; the rest of the target wires will be in a different QPU, so the corresponding CNOTs will have to be implemented non-locally, all using the same ebit (Figure **??**b). In the case a hyperedge is cut more than once – i.e. its vertices are split among more than two blocks – we will proceed in the same manner, but now we will require multiple ebits (Figure **??**c). As we previously defined, the function $\lambda: H \to \mathbb{N}$ tells us how many blocks a cut hyperedge connects. In order to connect $\lambda(h)$ blocks, we need at least $\lambda(h) - 1$ block-to-block connections: $\lambda(h) - 1$ ebits, the first halves of all the

ebits is kept by the QPU with the controlling wire, while the rest of the QPUs receive a single half each. Therefore, the metric we will need our hypergraph partitioner to minimise is $\lambda - 1$.

TODO Figure showing the different cases from the paragraph above, a, b and c. Give both the original circuit, the hypergraph and the black boxed distributed circuit (fig:vanillaCuts)

Following this intepretation of the hypergraph partition, we transform the original circuit to obtain its distributed version. To do so, we insert $\lambda(h) - 1$ cat-entanglers (Figure **??**) just before the group of CNOT gates that each of the hyperedges h represent; and $\lambda(h) - 1$ cat-disentanglers (Figure **??**) right after them. Then, each of the CNOTs to be implemented non-locally are modified so their control wire is the corresponding QPU's local ebit half.

TODO: A figure showing a simple circuit, its hypergraph and its distributed circuit (with cat-(dis)entangler as a box). It'd be great if the example on these three subsections was the same (fig:distribProcess)

A simple example of the distribution process is shown in Figure **??**. The resulting circuit is a distributed version of the original one, that is efficient in the sense described in § 4.2. Each of the QPUs can be set to implement its own local circuit, using its local ebit halves and classical communication whenever indicated.

4.2.2 Bringing CNOT gates together

In § 4.1 we have shown that any 1-qubit gate in the `Clifford+T` set acting on the control wire of a CNOT gate, with the exception of the Hadamard gate, can commute with the CNOT up to some byproduct. Here we use this fact, applying some preprocessing on the input circuit that brings together nearby CNOT gates, allowing us to implement more non-local CNOT gates using a single ebit. Figure **??** gives an example of how these transformations – listed in Figures **TODO** – can lead to a more efficient distribution of the circuit.

TODO Figure with (hopefully) the same circuit as the previous figure, its version after CNOTpulling and its hypergraph partition, which should have less lambda-1 (fig:pulledCNOTexample)

The preprocessing procedure is fairly straight-forward: From the beginning of the circuit, find each CNOT gate and use the transformations listed in Figures **TODO** to move every CNOT gate as early in the circuit as possible. The procedure introduces

some additional X gates, due to the transformations in Figures **TODO**. Fortunately, X is its own inverse (i.e. $XX = I$) and every 1-qubit gate in `Clifford+T` can be interchanged with X in a simple way (as shown in Figures **TODO**). Hence, we should not expect a significant increase in the depth of the circuit: most byproduct gates will cancel each other out.

So far we have been talking about standard 1-qubit gates, but in practical circuits we are likely to find 1-qubit gates that are *classically-controlled*, meaning that a classical signal (a bit, either 0 or 1) decides whether the gate is applied or not. This is no concern for our distribution of the circuit, as this classical control may only require classical communication between QPUs, whose cost we discussed to be negligible in comparison to quantum communication through ebits (see § 4.2). Concerning the preprocessing we just described, classically-controlled 1-qubit gates can commute with CNOT under the exact same circumstances as their uncontrolled version. The only difference is that, whenever a byproduct gate is created, we must make sure it is controlled by the same classical signal that spawned it, as shown in Figure **??**.

TODO Figure with an X gate classically controlled before the control of a CNOT. Then the resulting circuit after exchanging them (fig:classicalControl)

This preprocessing can commute 1-qubit gates both across the control wire and the target wire of the CNOT gate (although in the latter case, apart from H, S and T can not commute either). This has no effect at all on the vanilla version of the algorithm, but it will be beneficial after we apply our next extension, which requires CNOT gates to be directly contiguous on the target wire.

4.2.3 CNOT gates with common target wire

In § 4.1 (see Figure **??**) we showed that the trick for implementing multiple CNOT gates using a single ebit also works if they are contiguous on the target wire (instead of the control wire). This makes our optimisation problem more intricate: before, for each CNOT gate we only had two options, whether to make it local or non-local – i.e. whether to cut the hyperedges or not – but now, when the CNOT are to be implemented non-locally, there are two possible choices of how to do it, which we will represent as two different kinds of hyperedges, whether what their CNOTs have in common is the control or the target. Figure **??** shows a simple circuit where neither of the options is a priori better.

TODO: Figure where with 3 CNOTs we show that depending on how they are

grouped together, we may gain something or not, which ultimately depends on how the hypergraph is split. Include both hypergraphs (fig:BothEndsSimple) Caption: The hypergraph of the circuit built as in Algorithm 4.1 uses the common control of gates α and β, while the other one is built similarly, but on common target. Different line format for the hyperedges whether control or target.

Now, imagine such a circuit to be a fragment of a larger one where, for instance, allocating wires A and B in one QPU and wire C in another is the most efficient configuration. As shown in Figure **??**, in this particular case using the trick on the common target saves us one ebit. Conversely, if A were the wire on a separate QPU and B,C were together, using the common control would be best. The conclusion is that the choice of using control or target hyperedges on a particular fragment of the circuit is dependent on the overall partitioning. However, if we intend to use hypergraph partitioning, these CNOT gates must somehow be represented in the hypergraph. The best solution would be to build the hypergraph in such a way that the decision of using control or target hyperedges is not done a priori, but by the hypergraph partitioner itself. A naive approach, also shown in Figure **??**, would be to include all of the hyperedges, both of control and target type, in a single hypergraph. However, then either partitioning of the resulting hypergraph is cutting three hyperedges, so the hypergraph partitioner sees no difference between the two options, preventing it from taking into account this subtlety when optimising.

TODO: Figure of hypergraphs from prev Fig; with control hyps, target hyps, naively. The dashed lines represent two hypothetical ways of partitioning the hypergraph. Different line format for the hyperedges whether control or target (fig:BothEndsChallenge)

We propose to build the hypergraph as in Algorithm 4.2. This hypergraph requires an additional vertex per CNOT in the circuit, and the block where the hypergraph partitioner decides to include that vertex in dictates whether the CNOT is implemented as a control or target hyperedge. In Figure **??** shows how the hypergraph is built step by step for our running example.

Algorithm 4.2: Builds the hypergraph of a given circuit, without choosing whether CNOT gates are implemented through common control or common target. H may contain multiple hyperedges connecting the same vertices.

```
1  input: circuit
2  output: (V,H)
3  begin
4    V ← ∅
```

```
5   H ← ∅
6   hedge ← ∅
7   foreach wire in circuit do
8     V ← V + {wire}
9     H ← H + {hedge}
10    hedge ← {wire}
11    hType ← unknown
12    foreach gate in wire do
13      if gate == CNOT then
14        if controlOf(gate) == wire then
15          if hType == target then
16            H ← H + {hedge}
17            hedge ← {wire}
18          hType ← control
19        if targetOf(gate) == wire then
20          if hType == control then
21            H ← H + {hedge}
22            hedge ← {wire}
23          hType ← target
24        hedge ← hedge + {labelOf(gate)}
25      else
26        H ← H + {hedge}
27        hType ← unknown
28        hedge ← {wire}
29  end
```

TODO: Figure where the hypergraph for the three CNOTs example is built step by step (as in the Algorithm above) (fig:BothEndsProcess)

The hypergraph built by Algorithm 4.2 has one caveat: When discussing load-balancing in § 4.2, we explained that we were interested in allocating a uniform number of wires on each QPU. Previously, the hypergraph partitioner took care of this, as it tries to assign a uniform number of vertices to each block. But now, the hypergraph partitioner has no way of distinguishing between 'wire' vertices and 'CNOT' vertices, the latter being an artificial gadget that should not count towards load-balancing. The solution is simple, instead of the standard hypergraph partition problem, we apply a version of it where vertices can have a weight assigned (see Appendix A). Then, each 'wire' vertex is given weight 1, and each 'CNOT' vertex is given weight 0, effectively ignoring them for the load-balancing aspect.

We now explain how the hypergraph partition dictates how to distribute the circuit. An example of the process is shown in Figure **??**. First, let's look back at what we

discussed to happen when a hyperedge was cut in $\lambda(h)$ blocks in the vanilla version of our algorithm (§ 4.2.1): $\lambda(h) - 1$ ebits were generated, the first halves of all of them kept by the QPU with the control wire, the rest of the QPUs receiving one ebit half each. Now, the exact same thing will happen for cuts on 'control' hyperedges, while in the case on 'target' hyperedges the only difference being that control/target wire roles are switched, so in this case it is the QPU holding the target wire the one that keeps the first halves of all the ebits. Then, it is time to modify the CNOT gates so they are implemented non-locally when needed. It is in this step where the 'CNOT' vertices are given meaning: the block where they are assigned dictates in which QPU the CNOT operation itself is carried out. The CNOT gate will act on the wires it originally did in the input circuit, either directly because the wire is in the same QPU, or indirectly, by acting on the QPU's local ebit half that connects to the desired wire allocated in a distant QPU. Notice that the existence of such an ebit is ensured: the wire's vertex and the CNOT's vertex are, by construction (Algorithm 4.2), connected by a hyperedge, so whenever they are not assigned to the same QPUs, the hyperedge will be cut, and the ebit will be generated.

TODO: Simple example of building step by step the previous hypergraph (fig:BothEndsDistrib)

Remark: The distributed version of the circuit we would obtain if our hypergraph partitioner was restricted to never cut 'target' hyperedges, is equivalent to:

1. Always executing CNOTs in the same QPU where their target wire is allocated.
2. Never using the common-target trick.
3. Using the vanilla algorithm, from § 4.2.1.

Under this interpretation it is apparent that the problem of distributing a circuit using the common-control and common-target tricks, and the problem of partitioning the hypergraph built by Algorithm 4.2 are indeed the same. All the relevant information from the circuit – i.e. all the possible choices of how to implement each an every CNOT in the circuit – is encoded in the hypergraph, while each element of the hypergraph and cut on it can be directly mapped back to the circuit. When a partition of the hypergraph is decided, each of the possible ways the vertex set may be partitioned indicates a different way of distributing the circuit – the allocation of wires to QPUs correspond to which block the 'wire' vertices have been assigned – while each CNOT (both local and non-local) ultimately appear in the QPU where its 'CNOT' vertex has been assigned. Therefore, finding the optimal hypergraph partition will automatically

provide the most efficient way of using the common-control and common-target tricks to distribute the circuit.

It is interesting to discuss the particular case in Figure **??**. Here, a CNOT gate that must be applied non-locally between QPU 1 and QPU 2, ends up having its local fragment – the 'CNOT' vertex, and thus the CNOT gate itself – in neither of them, but in a distant QPU 3. This happens whenever QPU 1 and QPU 2 do not already share an ebit, but they are each sharing an ebit with QPU 3. Intuitively, it can be seen as it often being best to pass a message through an already existing middle-man, rather than put up a full blown connection just for sporadic messages. The hypergraph partitioner acting on the hypergraph built by Algorithm 4.2 will use this strategy whenever it reduces the number of ebits required.

TODO: The case where a non-local CNOT is actually applied in neither of the QPUs at its ends (fig:farCNOT)

Some criticism must be considered here: if we put no constraint on the exploitation of 'middle-men' QPUs, it may happen that communication across many of the QPUs all use the same middle-man QPU to deliver their messages, potentially creating a bottle neck. We see two ways around this; the first one is to accept that some circuits will naturally be prone to such a centralised communication network, where a few QPUs must be prepared to carry most of the communication work. This does happen in classical networking too, and sometimes the best option is a centralised system. As far as the hardware is designed with these needs in mind, centralisation may not be a bottle neck, but even an advantage. The second approach of tackling the problem is quite the opposite; assume you really wish a uniform network, then find a way to ensure the hypergraph partition you get has some kind of load-balancing on communication. Fortunately, there is an simple way of load-balancing the usage of ebits: instead of giving CNOT vertices weight 0 as we previously discussed, we may give them some weight $\mu > 0$ that indicates how important communication load-balancing is, compared to the uniformity of wire allocation across QPUs. Even better, we could use a version of the hypergraph partitioning problem where we provide three parameters instead of two, k, ε, η, where now ε acts as the tolerance for imbalance of 'wire' vertices, while η is a separate tolerance for imbalance of 'CNOT' vertices, both tolerances being enforced in any partition.

Across the figures of this subsection we have drawn differently hyperedges which group CNOTs with common-control or common-target. However, it should be noticed that the hypergraph partitioner does not need to use this information at any point. This

distinction is made for explanation purposes only. Once a hypergraph partition is decided, cat-entanglers and cat-disentanglers are added before and after each group of CNOTs that corresponds to a cut hyperedge. Then, the control and/or target of each CNOT gate is adjusted accordingly, so they connect to their local ebit half.

As a wrap up of this section, in Figure **??** we show the same circuit being distributed in four different ways: with or without the extension from § 4.2.2 and with or without the extension from § 4.2.3. This provides a simple example where both extensions are shown to reduce the number of ebits required to distribute a circuit.

TODO Figure example 'interesting' with different modes (fig:modes)

4.3 Interchanging CNOT gates

CNOT gates can be interchanged trivially when none of their wires are the same. When one of the wires is in common and has the same role (control or target), we can take advantage of it and implement them using a single ebit. If both wires are in common and with the same role, the CNOTs cancel each other. But what happens if two CNOTs act on the same wire with different roles? In that case, we can still interchange the gates as in Figures **TODO**, but that creates additional CNOTs.

TODO: Interchange challenge (picture from phone) (fig:interchangeChallenge)

It may seem like preprocessing the circuit so it has the minimum possible number of CNOT gates would always be the best option for partitioning. However, this is not always true, as shown in Figure **??**. In some cases interchanging CNOTs, although adding more CNOTs, may unlock a more efficient partitioning of the circuit. In other cases, it will have no benefit, while adding an additional CNOT to take into account when partitioning. This compromise prevents us from knowing a priori the best choice. Providing the hypergraph partitioner with the flexibility of deciding either to interchange a particular pair of CNOTs or not – in the same spirit we did in § 4.2.3 – would be the best solution. However, encoding such a choice in a hypergraph partitioning problem is very difficult, due to the following reasons:

1. The way CNOT gates are ordered in the circuit is important. This is something we omit in the hypergraph representation – looking at the final hypergraph in Figure **??**, we can not tell whether α goes before or after β. This information is key when interchanging, as it will dictate which are the new neighbours of the interchanged CNOTs. This could potentially be accounted for by imposing

that any hyperedge is an ordered list of vertices[4]. However, the standard hypergraph partitioning problem does not take into account the ordering of vertices, so we may need to define a custom hypergraph partitioning problem where this information is somehow taken into account.

2. Interchanging CNOT gates adds new CNOTs. This problem is substantially different from the problem in § 4.2.3, where the count of CNOTs remained the same, the only degree of freedom being whether each CNOT was implemented as control or as target hyperedge. The essence of our solution in § 4.2.3 is to represent all of the options in the hypergraph. However, if were to interchange a pair of CNOTs, new choices become available: is it worth to interchange the CNOTs again with their new neighbours? Should the byproduct CNOT, generated by the previous interchange, be itself interchanged further? Although the number of options to take into account is finite, it is considerably larger. And what is worse, each choice would not be independent from the rest – as some interchanges are only available if others have been done before – so the structure of the hypergraph would likely need to be quite complex to accommodate this aspect.

We propose to apply some postprocessing once the circuit has been partitioned, exhaustively applying the transformations in Figures **TODO** to each pair of potentially interchangeable CNOTs, keeping those transformations that reduced the ebit count. **TODO** Is this a greedy algorithm? should I give more details on the strategy?

4.4 An upper bound

TODO: This section

Finally, for the sake of comparison, we will use some theoretical results on quantum circuit decomposition, in order to estimate an upper bound of the number of ebits needed to distribute any quantum process on N qubits.

Discuss structured vs unstructured, this being the reason why we expect our algorithm to perform way better.

[4]For instance, the first vertex corresponding to a circuit wire, and the other corresponding to the different CNOT gates, ordered as in the circuit

Chapter 5

Implementation details and Results

Appendix A

Hypergraph Partitioning

Bibliography

Bennett, C. H., Brassard, G., Popescu, S., Schumacher, B., Smolin, J. A., and Wootters, W. K. (1996). Purification of noisy entanglement and faithful teleportation via noisy channels. *Physical review letters*, 76(5):722.

Biamonte, J., Wittek, P., Pancotti, N., Rebentrost, P., Wiebe, N., and Lloyd, S. (2017). Quantum machine learning. *Nature*, 549(7671):195.

Cirac, J., Ekert, A., Huelga, S., and Macchiavello, C. (1999). Distributed quantum computation over noisy channels. *Physical Review A*, 59(6):4249.

da Silva, R. D., Pius, E., and Kashefi, E. (2013). Global quantum circuit optimization. *arXiv preprint arXiv:1301.0351*.

Dawson, C. M. and Nielsen, M. A. (2005). The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*.

Díaz-Caro, A. (2017). A lambda calculus for density matrices withclassical and probabilistic controls. In Chang, B.-Y. E., editor, *Programming Languages and Systems*, pages 448–467, Cham. Springer International Publishing.

Duncan, R. and Perdrix, S. (2010). Rewriting measurement-based quantum computations with generalised flow. In *International Colloquium on Automata, Languages, and Programming*, pages 285–296. Springer.

Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., and Valiron, B. (2013). Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, volume 48, pages 333–342. ACM.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING*, pages 212–219. ACM.

Knill, E., Laflamme, R., and Viola, L. (2000). Theory of quantum error correction for general noise. *Physical Review Letters*, 84(11):2525.

Kok, P., Munro, W. J., Nemoto, K., Ralph, T. C., Dowling, J. P., and Milburn, G. J. (2007). Linear optical quantum computing with photonic qubits. *Reviews of Modern Physics*, 79(1):135.

Lanyon, B. P., Whitfield, J. D., Gillett, G. G., Goggin, M. E., Almeida, M. P., Kassal, I., Biamonte, J. D., Mohseni, M., Powell, B. J., Barbieri, M., et al. (2010). Towards quantum chemistry on a quantum computer. *Nature chemistry*, 2(2):106.

Nayak, C., Simon, S. H., Stern, A., Freedman, M., and Sarma, S. D. (2008). Non-abelian anyons and topological quantum computation. *Reviews of Modern Physics*, 80(3):1083.

Ömer, B. (2003). *Structured quantum programming*. na.

Raussendorf, R. and Briegel, H. J. (2001). A one-way quantum computer. *Physical Review Letters*, 86(22):5188.

Raz, R. and Tal, A. (2018). Oracle separation of bqp and ph. In *Electronic Colloquium on Computational Complexity*.

Shor, P. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332.

Shor, P. W. and Preskill, J. (2000). Simple proof of security of the bb84 quantum key distribution protocol. *Physical review letters*, 85(2):441.

Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., and Roetteler, M. (2018). Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM.

Van Meter, R. and Devitt, S. J. (2016). Local and distributed quantum computation. *arXiv preprint arXiv:1605.06951*.

Van Meter, R., Ladd, T. D., Fowler, A. G., and Yamamoto, Y. (2010). Distributed quantum computation architecture using semiconductor nanophotonics. *International Journal of Quantum Information*, 8(01n02):295–323.

Van Tonder, A. (2004). A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135.

Weidt, S., Randall, J., Webster, S., Lake, K., Webb, A., Cohen, I., Navickas, T., Lekitsch, B., Retzker, A., and Hensinger, W. (2016). Trapped-ion quantum logic with global radiation fields. *Physical review letters*, 117(22):220501.

Yimsiriwattana, A. and Lomonaco Jr, S. J. (2004). Generalized ghz states and distributed quantum computing. *arXiv preprint quant-ph/0402148*.