# 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 $\alpha$ and $\beta$ 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 $\alpha^2$ and $\beta^2$ 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 $\alpha$ and $\beta$ 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 $|1\rangle$). 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 the main constraint to scalability of quantum computers, as it dictates

[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 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

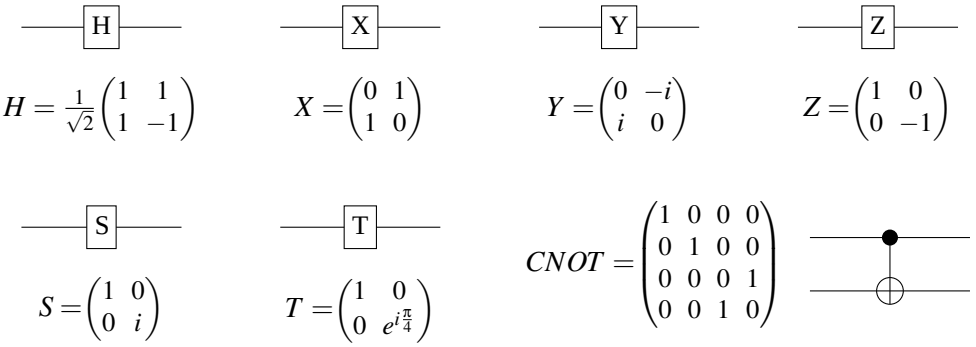



**Figure 2.1:** The Clifford+T gate set. Both their matrix and circuit representation are given.

$A^\dagger$ 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 2.1 and 2.2. All the properties from Figure 2.2 can be easily checked by calculating the matrix representing each of the circuits[3]. Circuits are read from left to right.

The CNOT gate 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 is applied to the target, flipping the state of the qubit. This works in any superposition, so

[3]Overall factors of norm 1 are irrelevant to the computation, so they are ignored in the circuit representation. E.g., algebraically: $iXZ = Y = -iZX$, but the $i$ and $-i$ factors are ignored in Figure 2.2.

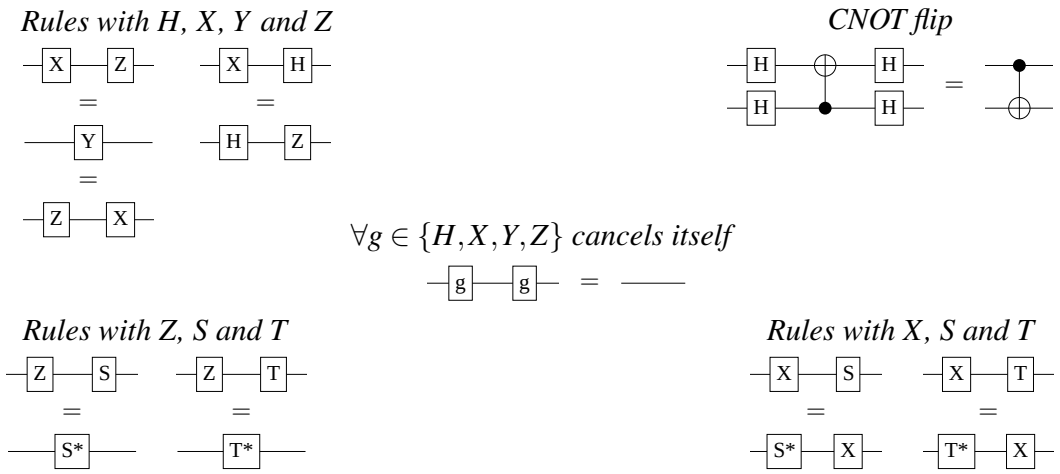



**Figure 2.2:** Some basic properties of the gates in the Clifford+T set. Here, $S^*$ and $T^*$ represent the inverse gates of $S$ and $T$, i.e. their conjugate transpose.

given

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

if the CNOT were to act in $|c\rangle$ as control and in $|t\rangle$ as target, the outcome would be:

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

- *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 can be thought of as sculpting a rock. The rock would be the initial entangled resource, which is a collection of entangled qubits forming a lattice structure. 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.

  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)[4]. In §3.3, we discuss an abstract distributed quantum architecture in detail.

## 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.

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

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 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 model 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.

**Remark 2.1.** 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 helpful: 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

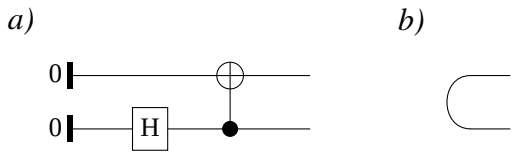



**Figure 3.1:** Generation of the Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle$ shown in *a)*. Its shorthand circuit notation is given in *b)*.

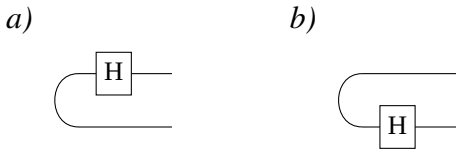



**Figure 3.2:** Applying a gate on a Bell state. The circuits shown in *a)* and *b)* are equivalent.

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 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, the most efficient communication channel will 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 3.1 shows how to prepare it.

An interesting property of the Bell state is shown in Figure 3.2: 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 are a natural choice for implementing quantum comunication.

[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

### 3.1.1 Entanglement distillation

So far, we have explained how to generate a Bell state $|\Phi\rangle = \frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle$ inside a QPU (as in Figure 3.1). However, what we aim for is that two different QPUs each own one of the qubits from $|\Phi\rangle$. The challenge is then to send one of the qubits to another QPU, while preserving the state. The problem of sharing a Bell state between two parties is solved by the *entanglement distillation* protocol (Bennett et al., 1996), which ensures the shared pair of qubits are in the Bell state, up to some small error factor. In this section, we will give a brief explanation of how this protocol works.

In practice, there are no perfect communication channels. This means that it is impossible to send a quantum state without potentially altering it, introducing errors. And, as we mentioned earlier in §2.2.1, detecting and correcting errors in quantum information is particularly challenging. Fortunately, in our case there is an easy way around it. We can create multiple $|\Phi\rangle$ states in the first QPU, and send one of the qubits of each of these pairs to the second QPU, through a noisy channel. Then, we would have shared multiple imperfect $|\Phi\rangle$ pairs, whose fidelity we can improve by making them interact with each other, destroying some of the pairs in the process. Before going into details on how we perform these interactions, we need some definitions:

- *Werner state*: It is the result of sending one of the qubits from $|\Phi\rangle$ through a noisy channel characterised by a constant $F \in [0,1]$. $F$ determines the *fidelity* of the channel, i.e. the probability that the qubit is sent without altering its state. Rather than an actual quantum state, the Werner state is a probability distribution of quantum states, known in the literature as a *mixed state*. In particular, it is the probability distribution of actually having $|\Phi\rangle$, with probability $F$, having $|\Phi\rangle$ with an $X$ gate mistakenly applied to the sent qubit, or having $|\Phi\rangle$ with a $Z$ gate on it, or having $|\Phi\rangle$ with both $X$ and $Z$ errors, each of these three cases[2] occurring with equal probability $\frac{1-F}{3}$.

- *Bilateral XOR (BXOR)*: Simply, two CNOT gates applied locally inside two different QPUs. Its intended use is to be applied to two Werner states; one QPU should have one of the qubits of each Werner state, and apply a CNOT on them,

[2] Although a channel will rarely introduce each of these errors with equal $\frac{1-F}{3}$ probability, it is always possible to apply some local operations on each of the qubits so the probability distribution converges to that of the Werner state. Moreover, there may be channels that introduce errors other than $X$ and $Z$; however the resulting state after sharing $|\Phi\rangle$ through any of these will always be some Werner state with some particular $F$.

while the other should do the same on the other two qubits. Thus, one of the Werner states acts as control of the CNOTs, and the other as their target.

The entanglement distillation protocol takes a collection of Werner states and repeats the following three steps multiple times, until the fidelity of the Werner states is the one desired:

1. Make pairs of two Werner states, and apply a BXOR to each pair. The new probability distribution (mixed state) on each of these four qubits is given in Table 3.1.

2. For each pair of Werner states, measure the qubits corresponding to the Werner state that acted as the target of the BXOR – this will require one measurement in each QPU. The outcome of the two measurements will match precisely in the cases where the fifth column of Table 3.1 shows no $X$ error. Examining the table, we can calculate the probability of that event to be:
$$T(F) = F^2 + \frac{2}{3}F(1-F) + \frac{5}{9}(1-F)^2$$
which, for $F > \frac{1}{2}$, is always over a half.

3. If the outcome of both measurements matched, keep the other two qubits (i.e. the control Werner state). Otherwise, discard them, not to be used again. Among the eight cases when we keep the Werner state, in two of them (grayed rows in Table 3.1) the state we keep has no errors, happening with probability:
$$P(F) = \frac{F^2 + 1/9(1-F)^2}{T(F)}$$
Thus, the kept qubit pairs are all Werner states with fidelity $P(F)$ which, as shown in Figure 3.3, is greater than $F$ whenever $F > \frac{1}{2}$.

Figure 3.3 implies that we may use a communication channel as bad as to introduce errors slightly less than half of the times, and still generate a Werner state whose fidelity is arbitrarily close to one. Naturally, the more fidelity we wish to attain, and the worse the fidelity we start with is, more iterations of the protocol will be required. This increases the amount of time and resources – number of initial Werner states – we must pay in order to obtain a single entangled qubit pair.

Often in distributed quantum computing literature, a Bell state shared by two QPUs is known as an *ebit* (entangled-bit) which, put another way, is just a Werner state with a fidelity close to one. Thus, this protocol produces ebits, which is the fundamental resource for quantum communication in distributed circuits. During the rest of this thesis, we use the term *ebit half* to refer to any of the two qubits that comprise an ebit.

**Table 3.1:** Probability distribution defining a pair of Werner states, and the errors present in each case, before and after the BXOR is applied.

| Probability | Errors before | | Errors after | | Kept? |
|---|---|---|---|---|---|
| | Source | Target | Source | Target | |
| $F^2$ | – | – | – | – | ✓ |
| $\frac{1}{3}F(1-F)$ | – | $Z$ | $Z$ | $Z$ | ✓ |
| $\frac{1}{3}F(1-F)$ | – | $X$ | – | $X$ | |
| $\frac{1}{3}F(1-F)$ | – | $X,Z$ | $Z$ | $X,Z$ | |
| $\frac{1}{3}F(1-F)$ | $Z$ | – | $Z$ | – | ✓ |
| $\frac{1}{9}(1-F)^2$ | $Z$ | $Z$ | – | $Z$ | ✓ |
| $\frac{1}{9}(1-F)^2$ | $Z$ | $X$ | $Z$ | $X$ | |
| $\frac{1}{9}(1-F)^2$ | $Z$ | $X,Z$ | – | $X,Z$ | |

| Probability | Errors before | | Errors after | | Kept? |
|---|---|---|---|---|---|
| | Source | Target | Source | Target | |
| $\frac{1}{3}F(1-F)$ | $X$ | – | $X$ | $X$ | |
| $\frac{1}{9}(1-F)^2$ | $X$ | $Z$ | $X,Z$ | $X,Z$ | |
| $\frac{1}{9}(1-F)^2$ | $X$ | $X$ | $X$ | – | ✓ |
| $\frac{1}{9}(1-F)^2$ | $X$ | $X,Z$ | $X,Z$ | $Z$ | ✓ |
| $\frac{1}{3}F(1-F)$ | $X,Z$ | – | $X,Z$ | $X$ | |
| $\frac{1}{9}(1-F)^2$ | $X,Z$ | $Z$ | $X$ | $X,Z$ | |
| $\frac{1}{9}(1-F)^2$ | $X,Z$ | $X$ | $X,Z$ | – | ✓ |
| $\frac{1}{9}(1-F)^2$ | $X,Z$ | $X,Z$ | $X$ | $Z$ | ✓ |

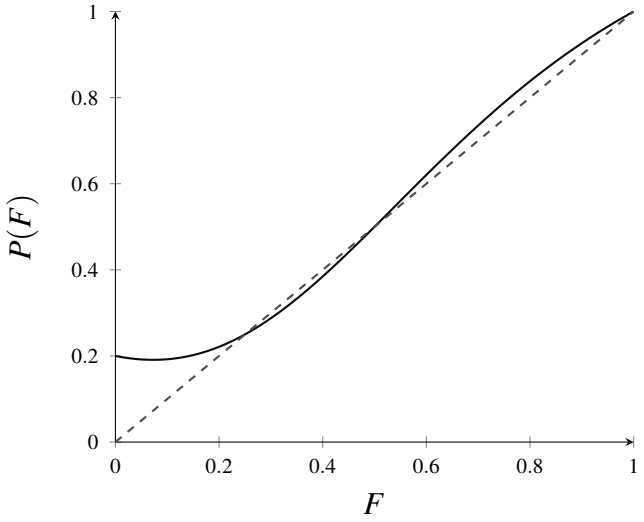



**Figure 3.3:** Fidelity after a single iteration of the distillation protocol, versus the original fidelity. The figure shows that $F > \frac{1}{2} \Rightarrow P(F) > F$.

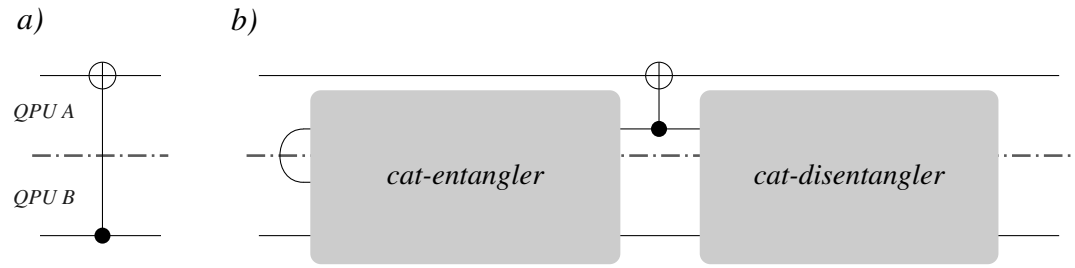



**Figure 3.4:** A non-local CNOT, shown in *a)*. The dashed line indicates how the circuit is separated into two QPUs. The implementation scheme is given in *b)*.

## 3.2 Distributing circuits

We now explain how ebits are used to allow a QPU to peek into another's qubits. Here, we will introduce 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. As we previously mentioned, any circuit can be converted to Clifford+T circuit (thanks to the Solovay-Kitaev theorem). In Clifford+T, the only gate that operates on more than one qubit is the CNOT. Hence, we only need to understand how CNOTs can be implemented non-locally.

The construction we will use is a slight variation of what was proposed by Yimsiriwattana and Lomonaco Jr (2004), and its scheme is shown in Figure 3.4. In principle, we will use an *ebit* per non-local CNOT. We will call the QPU that holds the target qubit (the one with a $\oplus$) the 'target QPU' and similarly for the control qubit.

The implementation of a local CNOT gate has three steps. First, we must apply what the authors refer to as the *cat-entangler* (Figure 3.5), which creates a local 'copy'[3] of the control qubit inside the target QPU. In the process, the ebit half in the control QPU is measured (and thus destroyed), and the outcome is used to correct the other half, in the same spirit as in the MBQC model. These corrections are done via *classically controlled gates*; devices that either apply a transformation to the qubit or none, depending on the value of a classical bit. Notice that the only information physically crossing the boundary between blocks is the *classical* outcome of the measurement (a bit, either 0 or 1).

[3] 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 $|\psi'\rangle = \alpha|0,0\rangle + \beta|1,1\rangle$ which is fundamentally different from an actual copy: $|\psi\rangle \otimes |\psi\rangle = \alpha^2|0,0\rangle + \alpha\beta|0,1\rangle + \alpha\beta|1,0\rangle + \beta^2|1,1\rangle$.

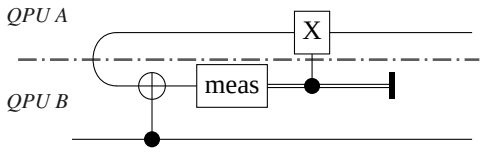



**Figure 3.5:** Implementation of the cat-entangler. An ebit is used to make the information in QPU B's wire available to QPU A. A doubled line indicates the wire holds classical information (a bit). For any input state $\alpha|0\rangle + \beta|1\rangle$, the output is $\alpha|0,0\rangle + \beta|1,1\rangle$.

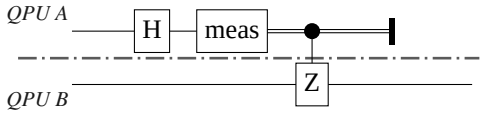



**Figure 3.6:** Implementation of the cat-disentangler. Essentially, it destroys the ebit half (top wire) that the cat-entangler coupled with QPU B's wire. A doubled line indicates the wire holds classical information (a bit). For any input state $\alpha|0,0\rangle + \beta|1,1\rangle$, the output is $\alpha|0\rangle + \beta|1\rangle$.

Then, the CNOT gate is *applied locally* inside the target QPU, between its ebit half and the target qubit. At the end, the *cat-disentangler* must be applied (Figure 3.6), 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.

In this way, we have implemented a non-local CNOT gate using one ebit and two classical bit messages between QPUs. However, the true advantage of this approach is attained when multiple non-local CNOTs are implementing using a single ebit: Once the cat-entangler is applied, any number of CNOTs whose target is in the same QPU, and that are controlled by the same qubit, may all be implemented by using the same ebit half as control, as shown in Figure 3.7.

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 fewest 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.

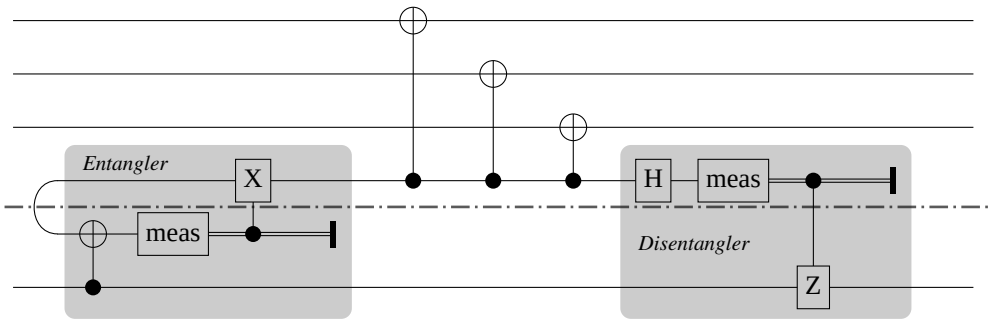



**Figure 3.7:** Implementing multiple non-local CNOTs using a single ebit. The bottom-most wire acts as control for the three of them. Only classical information crosses the boundary between QPUs.

## 3.3 Distributed quantum architectures

In this section we propose an abstract distributed quantum architecture. We claim that any distributed quantum computing technology will be characterised by the following features:

- *Multiple quantum processing units (QPUs)*: Each of the QPUs should be able to perform universal quantum computations on its collection of 'workspace' qubits. It should be possible to prepare these qubits to hold the input data of a program, and read output from (measure) them. It should also be possible to apply classically controlled gates on qubits.

- *Specialised space for ebits*: Each QPU should have some extra qubits meant to be used as ebit halves. These should be specialised so the operations necessary for ebit generation can be applied on them fast and reliably. The QPU should support the application of CNOTs (or some other suitable 2-qubit gate) between these specialised qubits and the ones in its workspace.

- *Ebit generation hardware*: This includes the (noisy) quantum communication channel itself and the ability to perform the distilling process. The generation of ebits may be done either in a centralised manner, with an specialised device that creates Bell states and sends them to the different QPUs, or decentralised, each QPU having their own hardware for creating and sharing the ebits. Depending on the technology used, entanglement distillation may be more or less essential; for instance, if the step of sharing Bell states is done using quantum optics (photons),

the quantum channel is likely to be fairly reliably, so it will only require a few iterations of the distilling process.

- *Classical communication network*: Which will be required in the process of entanglement distillation, as well as to perform cat-entanglers and cat-disentanglers. The QPUs will send signals through the network when they measure their qubits, and read from it to apply corrections.

As we discussed in §3.1.1, there is a compromise between the quality of the ebits and the effort put into preparing them. Fortunately, Cirac et al. (1999) showed that efficient distributed quantum computation using noisy ebits is feasible. Nevertheless, ebit generation will always be the main bottleneck of any distributed quantum architecture, as it is by far more expensive than classical communication and any local operation (in fact, multiple local operations are required in order to generate and use an ebit). Therefore, we will want to minimise the number of ebits required to implement any given algorithm.

Van Meter et al. (2010) have proposed an experimental distributed quantum architecture. They explain how each of the features listed above may be implemented, particularly focusing on how entanglement across QPUs is achieved (ebit generation). Their proposal is to use cavities (traps) where the particles encoding the qubits are kept, and use laser pulses (photons) to entangle cavities of different QPUs together.

# 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 implement a non-local gate. We will now extend their results.

In order to implement multiple non-local gates using a single ebit, there must be no operation on the control qubit between the non-local gates. Our first extension comes from the fact that some of the 1-qubit gates from the Clifford+T set commute with the CNOT gate. This means that, if there are operations in between CNOTs, we may transform the circuit to an equivalent version where the CNOT gates are brought together. All of the relevant circuit transformations are shown in Figure 4.1. Some gates do not commute with the CNOT, but we can still interchange them if we add an extra 1-qubit gate.

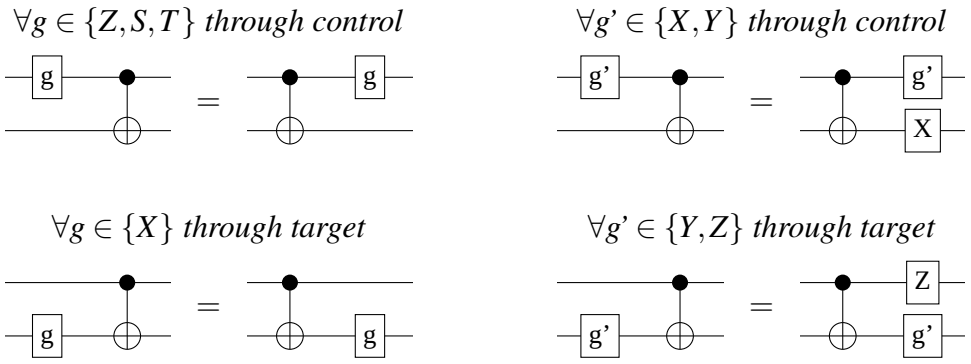



**Figure 4.1:** Different cases when a 1-qubit gate can be pushed through a CNOT gate.

The second improvement comes by realising that the method used to implement multiple CNOT gates controlled by the same wire (explained in §3.2) can also be applied if multiple CNOTs have a common target qubit instead. We will refer to the former method as the *remote-control* method, and we will refer to the later as the *remote-target* method. The derivation of the remote-target method is shown in Figure 4.2, which uses some of the properties listed in §2.2.2. The main difference is that the CNOT itself is now applied in the control QPU instead of the target, so the cat-entangler and cat-disentangler must change accordingly.

## 4.2 Finding an efficient distribution

In this section we explain how we search for a suitable distribution of the circuit. But first, we establish what we mean by a distributed circuit to be efficient. This is based on our discussiong in §3.3 about the characteristics of any distributed quantum architecture.We say that a distributed circuit is efficient when there is:

- *Minimal amount of quantum communication* between the QPUs, meaning it requires as little number of ebits as possible. In comparison, message passing of classical bits is considered negligible and it is not taken into account.

- *Load-balance across the QPUs*, up to a tolerance margin. Our notion of load-balance is that the different QPUs have a similar number of qubits assigned to them. A uniform depth of the local circuits (i.e. length of the circuits) 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 not take circuit depth into account, and instead assume that already known methods for depth reduction, such as the one described by da Silva et al. (2013), have already been applied on the input circuit, and may be applied again to each QPU's local circuit. 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. In it, a graph partition in $k$ subgraphs has to be found, minimising the number of *cut edges*: edges that have their incident vertices in different subgraphs. Additionally, the resulting partition must satisfy that the number of vertices in each subgraph is less than $(1\pm\varepsilon)\frac{N}{k}$, where $N$ is the total number of vertices in the graph. In Table 4.1 we list the correspondences between these two problems.

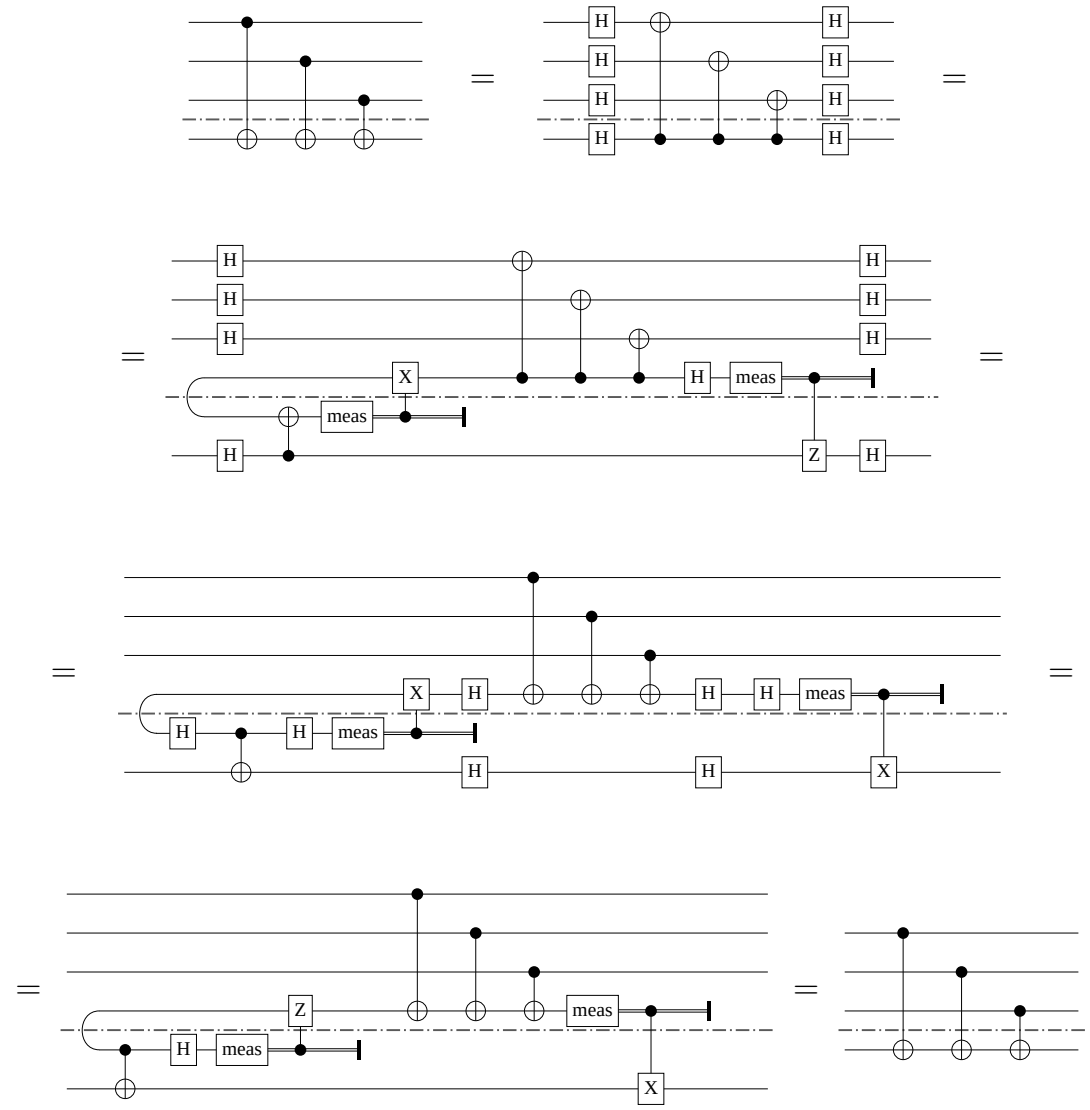



**Figure 4.2:** Proof of the implementation of multiple non-local CNOT gates that have common target. The proof uses properties given in Figures 2.2, 3.2 and 3.7. The result is similar to the case for common control (Figure 3.7). Both cat-entangler and cat-disentangler slightly differ.

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

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

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 the hypergraph partition problem 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 $(V,H)$, 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.

- Hypergraph partitioning follows the same premise as graph partitioning. The user provides a hypergraph and two parameters $(k,\varepsilon)$, 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 $\lambda\colon H \to \mathbb{N}$ 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 how many blocks are they 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) 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 challenge 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. In this first version of the algorithm, we will group CNOTs together only if they have a

[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 cut hyperedges is also an often used approach, but it is not as useful for our problem.

**Algorithm 4.1:** Builds the hypergraph of a given circuit. $H$ may contain multiple hyperedges connecting the same vertices. This algorithm runs in time $O(g)$, where $g$ is the number of gates in the input circuit.

```
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
```

common control wire and there are no other gates in between their connections to that wire. We will create *a single hyperedge* for every such a collection of CNOT gates. The hyperedge's vertices will correspond to the controlling wire and each of the different wires the CNOT gates target. Algorithm 4.1 receives a circuit as input and builds its hypergraph in that way.

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 determines how the corresponding wires are allocated to the different QPUs. The $\lambda - 1$ metric of the partition indicates the number of ebits that will be necessary to implement all the non-local CNOT gates. Hence, the problem of finding the optimal partition for a hypergraph built by Algorithm 4.1, and the problem of efficiently[4] distributing a circuit are equivalent – given any solution to one of them, we can compute a solution for the other. We formalise this fact in the following theorem:

[4]Where efficiency is assessed as discussed in the beginning of this section.

**Theorem 4.1.** *Given any circuit $\mathcal{C}$, and its hypergraph $\mathcal{H}$ generated by Algorithm 4.1, there is a bijection between partitions of $\mathcal{H}$ with $\lambda_c$ cuts and vanilla [5] distributions of $\mathcal{C}$ that use $\lambda_e$ ebits.*

*Proof.* We define this bijection inductively. First, we provide the bijection between the *trivial configurations*:

- The partition of $\mathcal{H}$ where all vertices are in the same block corresponds one-to-one to
- circuit $\mathcal{C}$ being executed in a single QPU.

Then, we define a *primitive transformation* for both problems, which allows us to move vertices/wires around. To do so, we use the one-to-one correspondence between vertices and wires that Algorithm 4.1 imposes. We additionally require to have an indexed set of hypergraph blocks and QPUs.

- Given a partition of $\mathcal{H}$, moving vertex $x$ from block $i$ to block $j$ corresponds one-to-one to
- picking wire $x$ – which is guaranteed to be in QPU $i$ – and allocating it in QPU $j$.

Using this primitive, any partition/distribution can be reached, so we indeed have a bijection. Notice that wire $x$ was guaranteed to be in QPU $i$ thanks to our inductive definition of the bijection itself. It remains to proof that the given bijection preserves the match between the number of cuts $\lambda_c$ and the number of ebits required $\lambda_c$. We give a proof by induction on the sequence of primitives that generated any given configuration, starting from the trivial configuration.

- The *trivial configuration* of both problems has $\lambda_c = \lambda_e = 0$.
- Given a sequence of $n+1$ primitives, we assume that $\lambda_c = \lambda_e$ holds by the time the $n$-th primitive was applied (our induction hypothesis). Then, reallocating vertex $x$ from block $i$ to block $j$, as determined by the $(n+1)$-th primitive, may:

  a). *Increase* $\lambda_c$. $\lambda_c$ increases by one iff a new cut is added to an arbitrary hyperedge $h$. This happens iff the block/QPU $j$ did not already contain a

[5] Meaning that non-local CNOT gates may only be implemented using the remote-control method from Figure 3.7.

vertex/wire from $h$. In that case, $x$ becomes an isolated non-local CNOT, and to implement it, it will be necessary and sufficient to create an extra ebit, increasing $\lambda_e$ by one. Therefore, $\lambda_e$ will increase iff $\lambda_c$ does so, both by the same amount.

b). *Decrease* $\lambda_c$. In the same spirit, $\lambda_c$ decreases by one iff a previously isolated vertex/wire is allocated where some fellow vertices/wires are. One less ebit will be required when and only when this happens. Therefore, $\lambda_e$ will decrease iff $\lambda_c$ does so, both by the same amount.

Therefore, starting from $\lambda_c = \lambda_e = 0$ and applying the whole sequence of primitives one by one will maintain $\lambda_c = \lambda_e$.

□

**Remark 4.2.** Figures 4.3, 4.4 and 4.5 provide a simple example of the one-to-one correspondence discussed in Theorem 4.1. Interestingly, in Figure 4.5, the second cat-entangler 'copies' the information held by the local ebit half previously entangled with $C$, instead of being directly coupled with $C$. The latter option would also be correct[6]. Either way is represented by the same hypergraph partition, so in order to maintain our one-to-one correspondence, we should rather say that there is a bijection between hypergraph partitions and equivalence classes[7] of circuit distributions. Although for our algorithm all of the solutions in one such equivalence class are indistinguishable, some of them will offer a more decentralised network of ebits than others. Optimisations taking into account this fact could be done as post-processing of our proposed algorithm.

A closer look at the bijection proposed in Theorem 4.1 reveals that we can transform back and forth between hypergraph partition and circuit distribution in polynomial time: We just need to read where each of the vertices/wires are allocated, and apply the primitive once for each of them – which affects all the hyperedges/CNOTs on that vertex/wire. Hence, the time complexity of the transformation in either direction is $O(n \cdot m)$, where $n$ is the number of vertices/wires and $m$ the number of hyperedges/CNOTs. This leads us to two results, one per direction of the bijection:

**Corollary 4.3.** *The best vanilla distribution of a circuit can be efficiently derived from an optimal partition of the hypergraph built from it by Algorithm 4.1.*

[6]However, in that case, wires $B$ and $C$ should interchange places in the circuit representation, so it remains planar.

[7]Families of circuit distributions that are equivalent in the sense that their wires are allocated in the same way, and that the number of ebits required also matches.

*Proof.* We know it will be the best distribution thanks to Theorem 4.1: The optimal partition will have the lowest possible $\lambda_c$, and given that at any point $\lambda_c = \lambda_e$, the corresponding circuit distribution will also have the lowest $\lambda_e$ possible.

We say the circuit distribution is efficiently derived because the required transformations – from input circuit to hypergraph, and from hypergraph partition to circuit distribution – both run in polynomial time.

□

**Corollary 4.4.** *The quantum circuit distribution problem is an NP-complete problem.*

*Proof.* To proof NP-completeness we simply need to show that, if the quantum circuit distribution problem could be solved in polynomial time, we would be able to solve some NP-complete problem in polynomial time. The hypergraph partitioning problem happens to be NP-complete (Lyaudet, 2010), and Theorem 4.1 allows us to take any solution for the circuit distribution problem and provide the optimal partition of its hypergraph. The only caveat is that we need to be able to do this for any hypergraph so, given any hypergraph $\mathcal{H}$, we must be able to build a non-distributed circuit $\mathcal{C}$ that is represented by $\mathcal{H}$ – the opposite direction of what Algorithm 4.1 does. There will be multiple such circuits; building one of these in polynomial time is possible, and the details are straight-forward.

□

On the pessimistic side, this means that unless $P = NP$, finding the best distribution of an arbitrary quantum circuit will take exponential time. On the optimistic side, many problems that compilers from classical computer science deal with are also NP-complete. In our order to have fast compilers that prepare quantum algorithms to be run in distributed architectures, we will not need to look for better algorithms to solve our particular problem: we may use the already rich research on fast algorithms for hypergraph partitioning (Akhremtsev et al., 2017), as the polynomial overhead of transforming back and forth between problems will be negligible.

A simple circuit, its optimally partitioned hypergraph and the resulting distributed circuit are shown in Figure 4.6. The obtained distribution is the most efficient one, in the sense described in the beginning of this section. Each of the QPUs can be set to implement its own local circuit, distilling ebits and using them along classical communication whenever indicated.

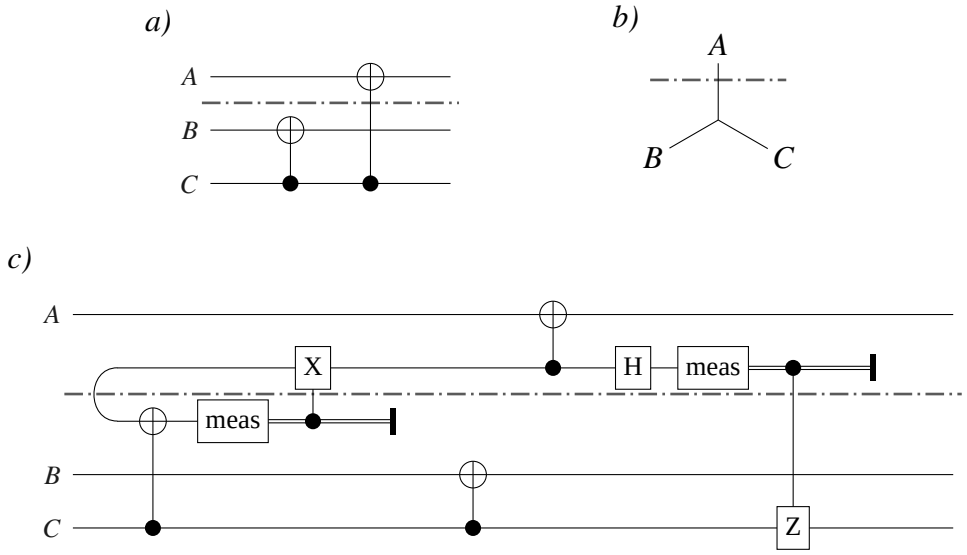



**Figure 4.3:** The CNOTs in the circuit *a)* are adjacent at their control wire. Therefore, a single hyperedge is used to represent both in *b)*. The proposed cut makes only one of the CNOTs non-local, which is implemented in *c)* using one ebit.

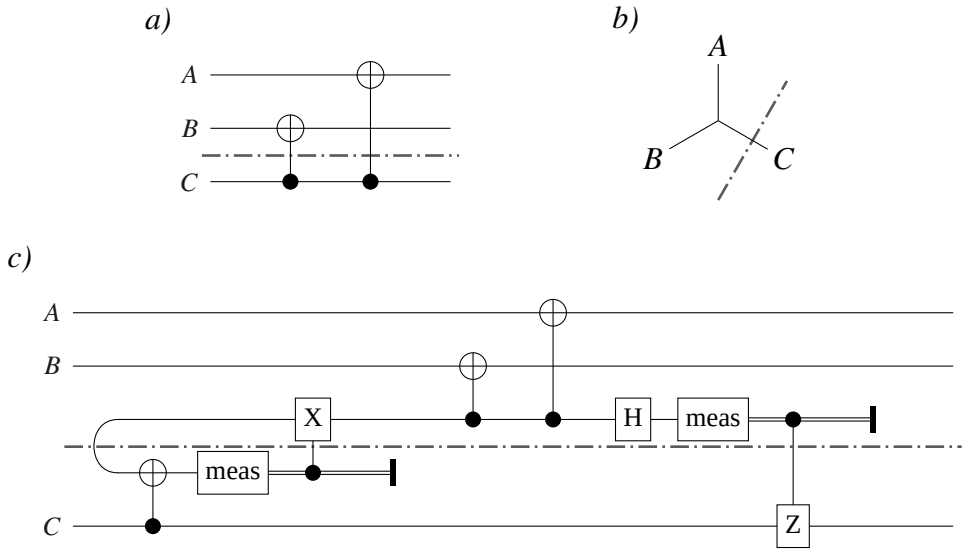



**Figure 4.4:** Same as in Figure 4.3, but now the cut makes both CNOTs non-local. Still, only one ebit is required, as implied by the hypergraph *b)*. When CNOTs share their control wire, an ebit is required iff any of the target wires is in a different QPU than the control wire's QPU.

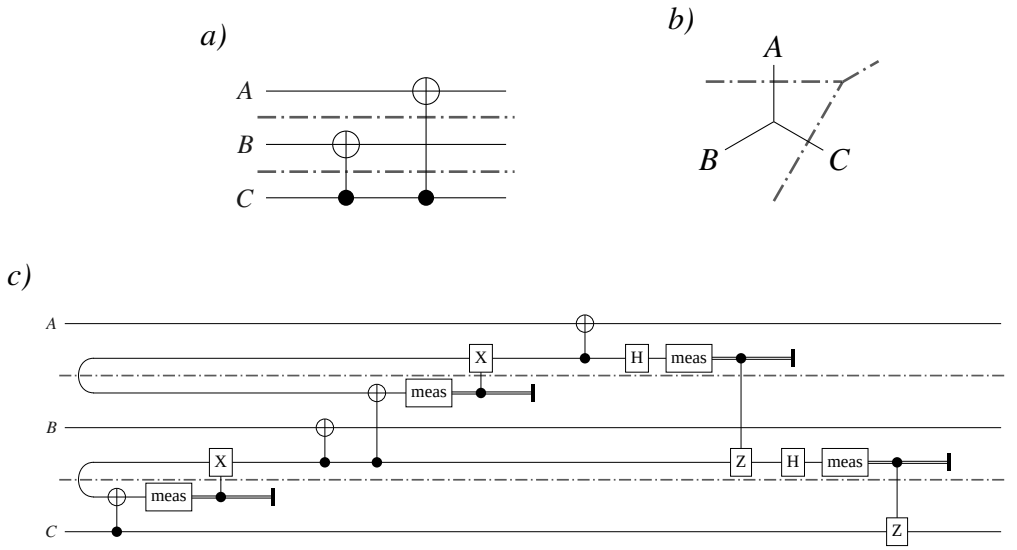



**Figure 4.5:** Same as in Figure 4.3, but now there are two cuts, distributing the circuit across three QPUs.

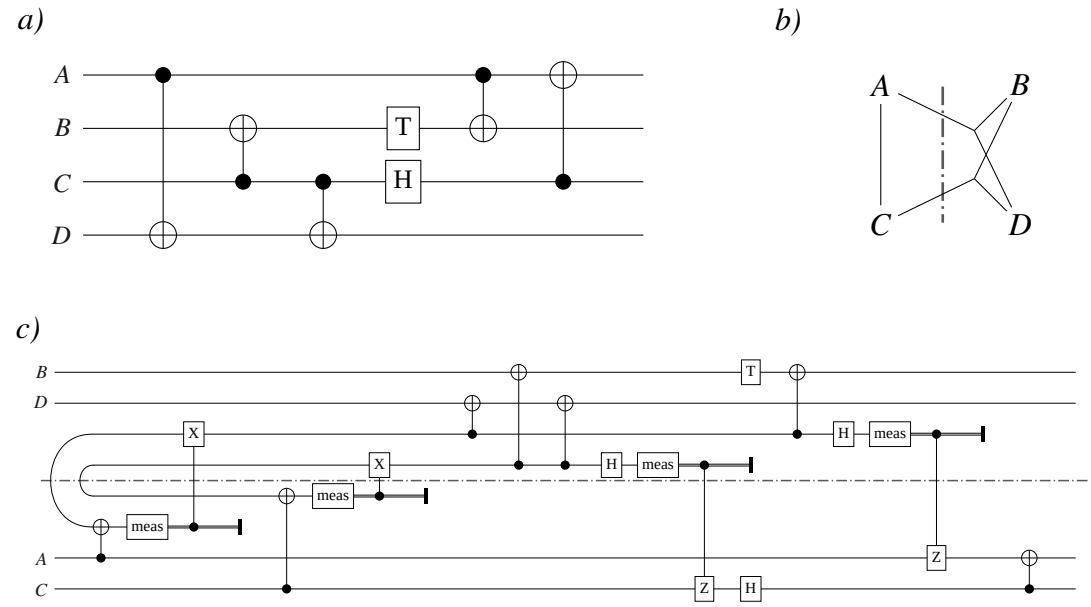



**Figure 4.6:** The circuit is distributed using only 2 ebits. The other two possible distributions: $\{\{A,B\},\{C,D\}\}$ and $\{\{A,D\},\{B,C\}\}$ both require 3 ebits to be implemented. The wires in the distributed version of the circuit have been rearranged, so it is possible to visualise in a planar diagram that no quantum information crosses the boundary.

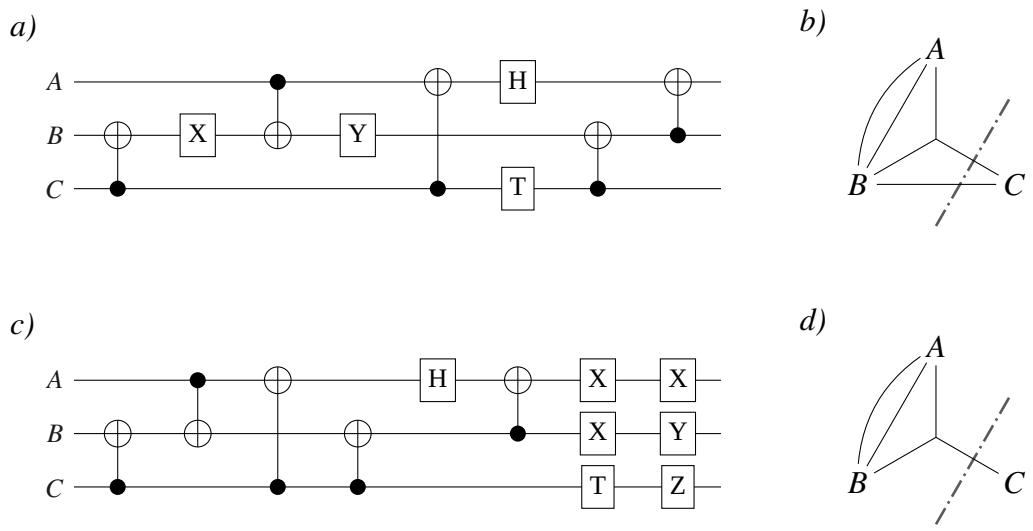



**Figure 4.7:** Example where an ebit can be saved by moving CNOTs closer together. This figure shows the original circuit *a)*, its optimally partitioned hypergraph *b)*, the preprocessed circuit *c)*, and its optimally partitioned hypergraph *d)*, which has one less cut hyperedge.

### 4.2.2 `SlideCNOTs`: 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 H gate, can commute with the CNOT up to some byproduct. Here we use this fact, applying some pre-processing on the input circuit that brings together nearby CNOT gates, allowing us to implement more non-local CNOT gates using a single ebit. Figure 4.7 gives an example of how these transformations – listed in Figure 4.1 – can lead to a more efficient distribution of the circuit.

The pre-processing procedure is fairly straight-forward: Exploring the circuit from left to right, whever a CNOT gate is found, use the transformations listed in Figure 4.1 to move it as early in the circuit as possible. The procedure introduces some additional $X$ gates. 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 Figure 2.2). Hence, we should not expect a significant increase in the depth of the circuit, as 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. These are no issue for the distribution of the circuit, as this classical control may only require classi-

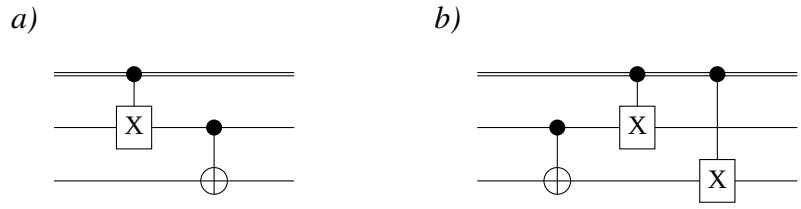



**Figure 4.8:** Pushing a classically controlled gate through a CNOT. The same rule as in Figure 4.1 is applied, while making sure any new gate is also controlled. Here, only the case for $X$ gate is shown, but this works for any of the transformations in Figure 4.1.

cal communication between QPUs. Concerning the pre-processing 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 controlled the original gate, as shown in Figure 4.8.

The same procedure can be used to commute 1-qubit gates across the target wire of the CNOT gates, although in this case, apart from $H$, $S$ and $T$ gates can not commute either. This additional pre-processing would have 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 adjacent on the target wire.

**Remark 4.5.** *Enabling* `SlideCNOTs` *will never increase the ebit count required to implement a partition.*

This extension simply moves the CNOTs along their wires. The same CNOTs that were non-local without `SlideCNOTs` enabled will still be non-local. On the worst case, these non-local CNOTs could be implemented in the exact same way as they were before, so the ebit count would remain the same.

### 4.2.3 `EitherRemote`: Adding the remote-target method

In §4.1 we showed that the trick for implementing multiple CNOT gates using a single ebit also works if they share a common target wire (instead of the control wire). This makes our optimisation problem more intricate: Now, when the CNOTs are to be implemented non-locally, we can choose to implement them using the remote-control or remote-target method. For explanation purposes, we will use two different kinds of hyperedges in this subsection's figures, depending on whether their CNOTs are controlled by the same wire, or act on the same target (thicker line).

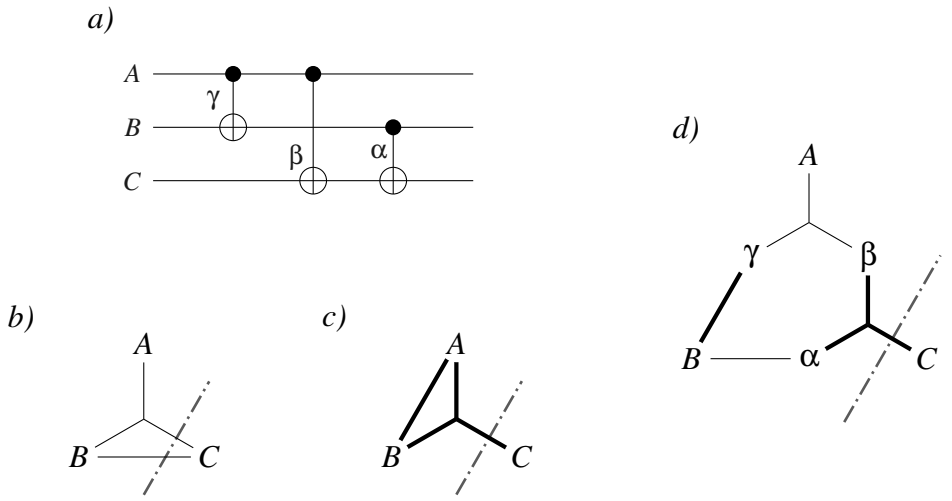



**Figure 4.9:** A circuit *a)* and its hypergraph *b)* as built by Algorithm 4.1. Hypergraph *c)* is the result of running the same algorithm but grouping CNOTs in the same hyperedge if and only if they share the same target (instead of control). Hypergraph *d)* is the one built by Algorithm 4.2.

Figure 4.9*a)* shows a simple circuit where a CNOT $\beta$ shares its target wire $C$ with another CNOT $\alpha$. The remote-target method would allow to implement these two non-locally, using a single ebit. We represent this fact in hypergraph *c)*, as the three-vertex hyperedge. Gate $\gamma$ shares its target with no other gate, so it is represented with an edge $(A,B)$. Hypergraph *b)* is the one built by Algorithm 4.1, and considers that all non-local CNOTs are implemented as remote-control, so it is now gate $\alpha$ the isolated edge $(B,C)$. If we were to partition the circuit as shown in the figure, hypergraph *b)* would suggest that two qubits are required, when actually only one is enough, as shown by hypergraph *c)*. But, if the partition is $\{\{A\},\{B,C\}\}$ instead, hypergraph *c)* would be the one overestimating the number of ebits needed. Hence, both hypergraphs *b)* and *c)* are biased.

If we want to guarantee that the most efficient circuit distribution can always be found by a graph partitioner, we must find an alternative hypergraph representation of the circuit. Notice that it is not an option to simply find the optimal partition for both hypergraphs *b)* and *c)* and choose the one with least cuts: The best circuit distribution is unlikely to have all of its CNOTs implemented in the same way. Therefore, our hypergraph representation must have the following characteristics:

1. Both options of how to implement each non-local CNOT are represented.

2. A partition of the hypergraph must determine, for each non-local CNOT, which method should be used to implement it.

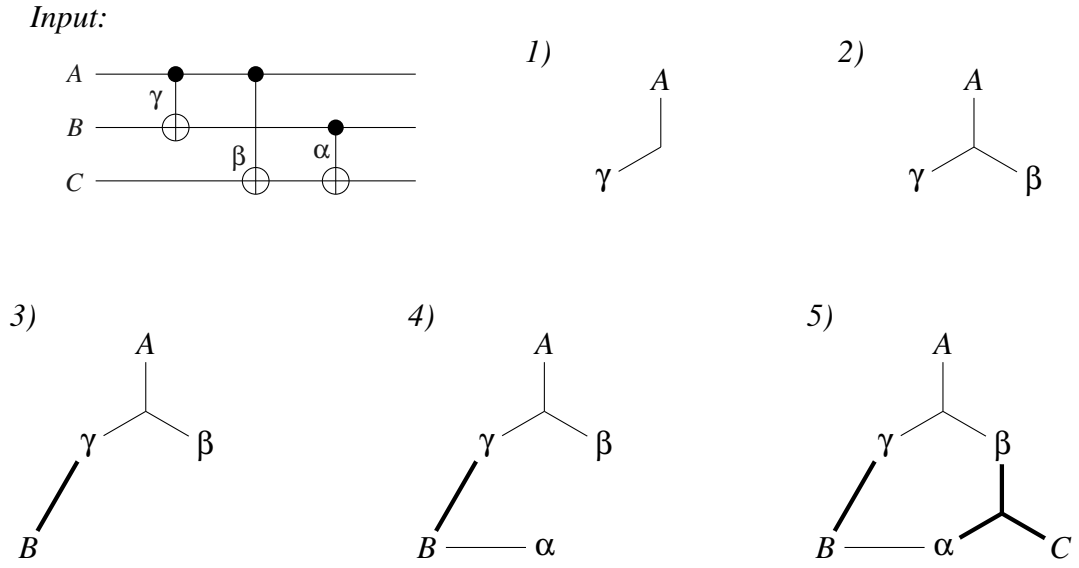



**Figure 4.10:** Step by step execution of Algorithm 4.2.

3. The $\lambda - 1$ metric of a partition should accurately determine how many ebits are required to implement the circuit distribution it describes.

We propose our new hypergraph representation to be as *d)* from Figure 4.9. Algorithm 4.2 builds such a hypergraph for any given circuit, and Figure 4.10 shows how the hypergraph is built step by step for our running example. These hypergraphs satisfy the three requirements defined above: First, we include all of the control-hyperedges and target-hyperedges. For the second requirement, we give additional structure to the hypergraph, adding vertices that represent CNOT gates; how a CNOT is implemented will be determined by the block its CNOT-vertex is assigned to. The third requirement is the most subtle one. Intuitively, we satisfy it by, imposing that each CNOT-vertex participates in only two hyperedges; one connecting it to its control wire-vertex, the other to its target. A more detailed account of how this third requirement holds is given in the proof of Theorem 4.6. Corollary 4.7 confirms that the optimal partition of our hypergraph determines the most efficient circuit distribution.

**Theorem 4.6.** *Given any circuit $\mathcal{C}$, and its hypergraph $\mathcal{H}$ generated by Algorithm 4.2, there is a bijection between partitions of $\mathcal{H}$ with $\lambda_c$ cuts and distributions of $\mathcal{C}$ that use $\lambda_e$ ebits.*

*Proof.* We define the same trivial configuration from Theorem 4.1. In the case of §4.2.1, the CNOT operation itself was always applied in the same QPU as its target,

**Algorithm 4.2:** Builds the hypergraph of a given circuit, without choosing whether CNOT gates are implemented through common control or common target. This algorithm runs in time $O(g)$, where $g$ is the number of gates in the input circuit.

```
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         V ← V ∪ {labelOf(gate)}
15         if controlOf(gate) == wire then
16           if hType == target then
17             H ← H + {hedge}
18             hedge ← {wire}
19           hType ← control
20         if targetOf(gate) == wire then
21           if hType == control then
22             H ← H + {hedge}
23             hedge ← {wire}
24           hType ← target
25         hedge ← hedge ∪ {labelOf(gate)}
26       else
27         H ← H + {hedge}
28         hedge ← {wire}
29         hType ← unknown
30 end
```

regardless of the CNOT being local or not (see Figure 3.7). Thus, the primitive from Theorem 4.1 for reallocating a wire-vertex $x$ is defined here with the extra constraint that all CNOT-vertices connected by a target-hyperedge to $x$ must move with it to the same block/QPU $x$ does. In this way, the primitive has on our new hypergraph the exact same meaning it originally had.

We need to add another primitive that allows us to move a CNOT-vertex $x$ independently from its wire-vertices; otherwise not every hypergraph partition would be reachable. Conversely, in order to reach any arbitrary circuit distribution, we need to be able to change the QPU where a CNOT gate $x$ is implemented. This pair of primitives is defined to correspond one-to-one to each other. Let's consider the four distinct cases of how $x$ may be allocated regarding its two neighbouring wire-vertices, which act as $x$'s control $c_x$ and target $t_x$:

- *Local*: The three vertices $x$, $c_x$ and $t_x$ are in the same block $i$. In this case, $x$'s allocation causes *no cut* to either its control-hyperedge or its target-hyperedge. The CNOT gate is implemented locally in QPU $i$, so *no ebit* is required to implement it.

- *Remote control*: Both $x$ and $t_x$ are in a block $i$, while $c_x$ is in a different block $j$. In this case, $x$'s allocation causes its control-hyperedge to be cut. The cut will already be present if there are other vertices from the hyperedge allocated in $i$; otherwise *one cut* will be added. On the other hand, an ebit is required to implement CNOT $x$. If there are other CNOTs in $i$ controlled by the same $c_x$ wire (i.e. they are in the same control-hyperedge), the ebit they require can be used to implement $x$; otherwise *one ebit* will be added.

- *Remote target*: Both $x$ and $c_x$ are in a block $i$, while $t_x$ is in a different block $j$. In this case, $x$'s allocation causes its target-hyperedge to be cut. Again, the cut will be present if there are other vertices from the hyperedge allocated in $i$; otherwise *one cut* will be added. On the other hand, an ebit is required to implement CNOT $x$. If there are aother CNOTs in $i$ that target the same $t_x$ wire (i.e. they are in the same target-hyperedge), the ebit they require can be used to implement $x$; otherwise *one ebit* will be added.

- *External*: The three vertices are in separated blocks, with $x$ in some block $i$. An example of this situation is shown in Figure 4.11. In this case, $x$'s allocation contributes to *both a cut* on its control-hyperedge and on its target-hyperedge.

In this case, *two ebits* are required; the one used to access $c_x$ may be shared with other CNOT gates in $i$ that have that wire as common control, while the ebit used to access $t_x$ may be shared with CNOTs in $i$ that have that target in common.

The proof by induction discussed in Theorem 4.1 holds here if we manage to show that any application of the new CNOT-vertex reallocation primitive still preserves $\lambda_c = \lambda_e$. Changing the allocation of a CNOT-vertex $x$ may change its situation among the four cases above. As we detailed, in each case $x$'s allocation contributes to the same amount of cuts as ebits the CNOT $x$ requires. Hence, changing the allocation of $x$ always preserves $\lambda_c = \lambda_e$.

□

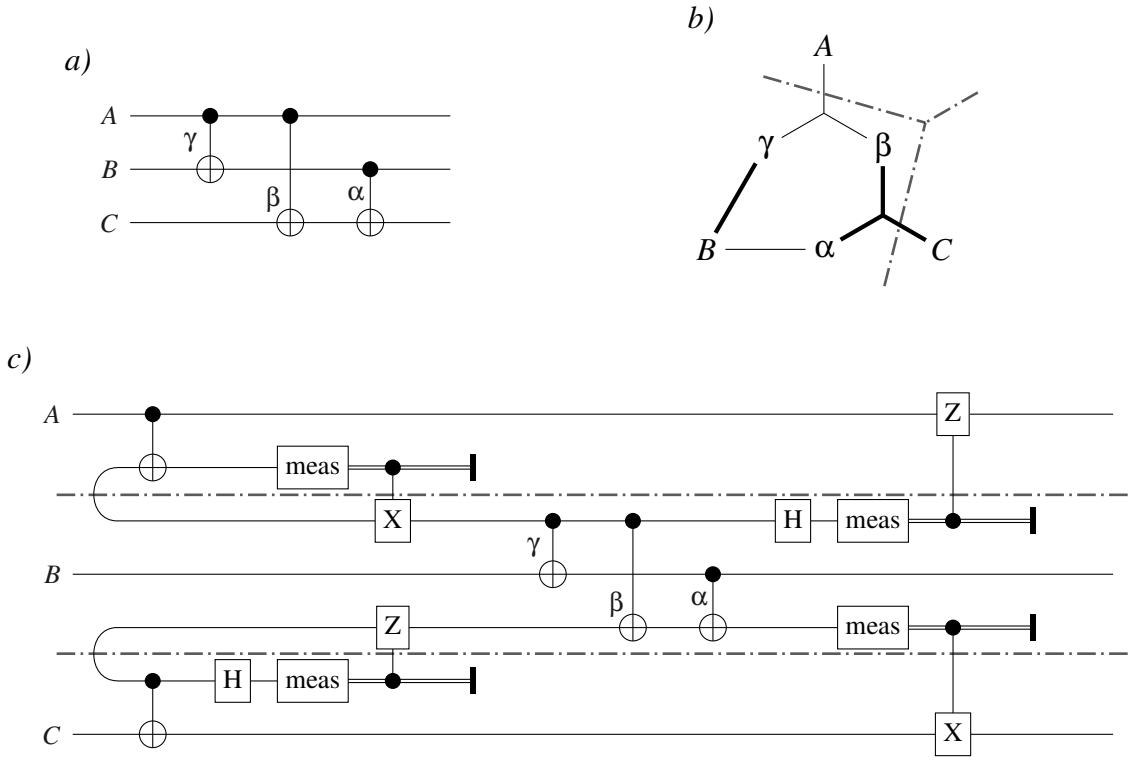



**Figure 4.11:** Distribution of circuit *a)* when each wire is in a different QPU. Gate β is implemented as an 'external' CNOT, i.e. both its control and target are in a remote QPU.

The procedure for distributing the circuit following a given hypergraph partition is very similar to that in the vanilla version of the algorithm, now taking into account the new primitive defined in Theorem 4.6. The time complexity of the procedure still is $O(n \cdot m)$, where $n$ is the total number of vertices and $m$ is the number of hyperedges. Also, notice that the cat-entangler and the cat-disentangler are slightly different

whether the CNOTs are implemented through remote-control or remote-target (see Figure 4.2).

**Corollary 4.7.** *The best distribution of a circuit can be efficiently derived from an optimal partition of the hypergraph built from it by Algorithm 4.2.*

*Proof.* Follows directly from Theorem 4.6, by the same argument given for Corollary 4.3.

□

**Corollary 4.8.** *The distributed circuit we obtain using the vanilla algorithm (from §4.2.1) is the same as the one obtained if we restrict the approach in this subsection so either:*

*a). No CNOT is implemented using the remote-target method.*

*b). Our hypergraph partitioner never cuts target-hyperedges.*

*c). CNOT gates are always executed in their target QPU.*

*All of these restrictions are equivalent.*

*Proof.* Follows directly from the proof of Theorem 4.6.

□

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 to each QPU. Previously, the hypergraph partitioner took care of this, as it tried 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.

The case illustrated in Figure 4.11 can be seen intuitively as passing a message through a middle-man. Sometimes, using communication through an already available middle-man is better than putting up a full blown connection just for a single message. The hypergraph partitioner acting on the hypergraph built by Algorithm 4.2 will automatically use this strategy whenever it reduces the number of ebits required. Some

criticism is relevant 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 solutions:

1. Specialise your hardware as centralised communication network, where the 'middle-man' QPU has a similar role to a classical server. The server should be capable of managing a large number of ebits fast and reliably. This is the natural option if the algorithms we wish to run have an intrinsically centralised behaviour – for instance, if all the gates are controlled by the same wire.

2. In case we wish to have a decentralised network, we will need to find a way to ensure the hypergraph partition we get has some kind of load-balancing the usage of ebits. Fortunately, there is a simple way to do this: instead of giving CNOT vertices weight 0 as we previously discussed, we may give them some weight $\mu > 0$ that indicates how relevant communication load-balancing is in comparison to the uniformity of wire allocation across QPUs. Even better, we could use a custom version of the hypergraph partitioning problem where we provide three parameters instead of two, $k, \varepsilon, \eta$, where now $\varepsilon$ acts as the tolerance for imbalance of wire-vertices, while $\eta$ is a separate tolerance for imbalance of CNOT-vertices, both tolerances being enforced in any partition.

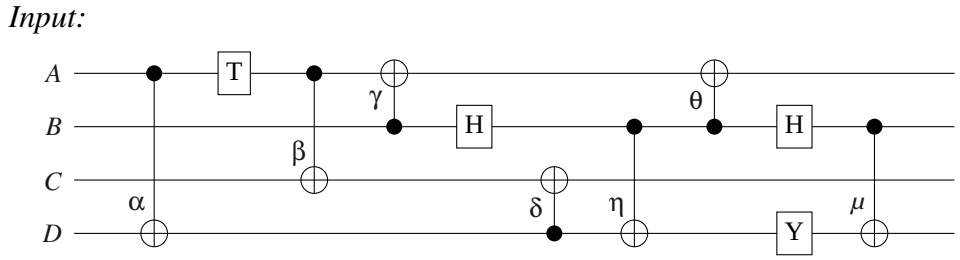
**Remark 4.9.** *Enabling* `EitherRemote` *will never increase the ebit count required to implement a partition.*

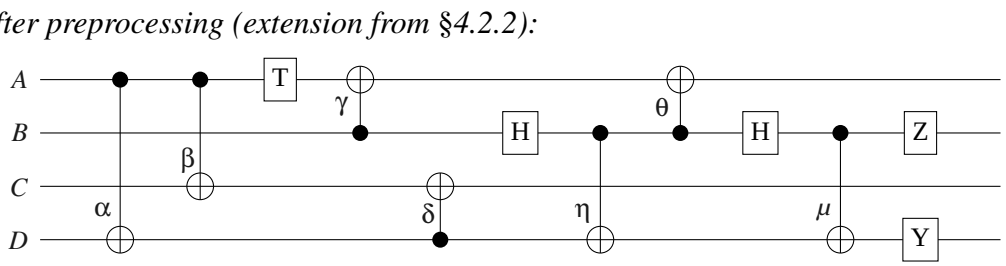
This extension simply changes how each non-local CNOT is implemented. On the worst case, these non-local CNOTs could still be implemented in the exact same way as they were before, so the ebit count would remain the same.

As a wrap up of this section, Figure 4.12 showd 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.

## 4.3 Interchanging CNOT gates

When applied to disjoint sets of wires, CNOT gates commute trivially. If one of the wires is in common, in case it has the same role for both CNOTs (control or target),



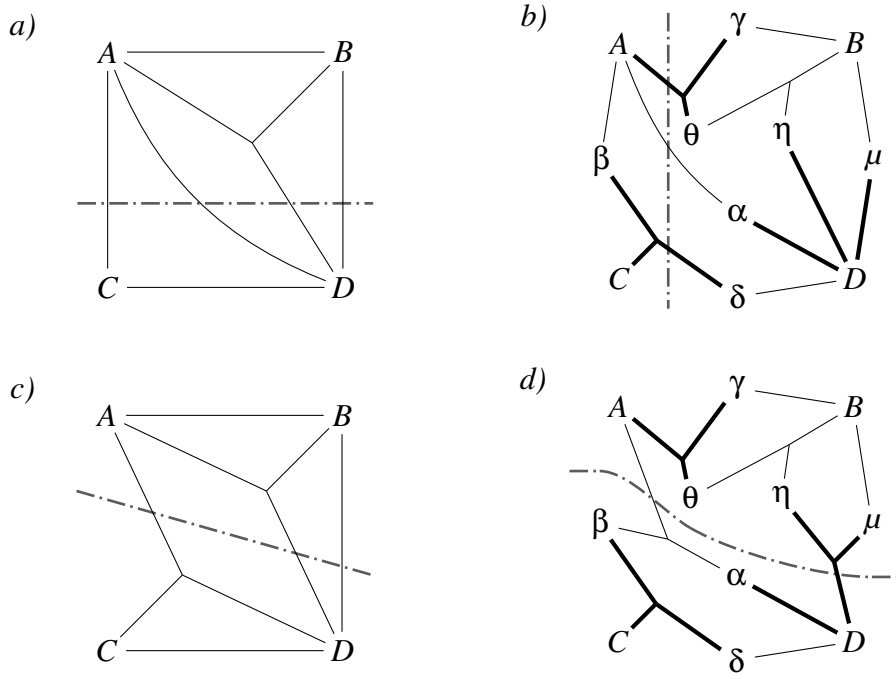




**Figure 4.12:** Different hypergraphs representing the input circuit. *a)* corresponds to no extensions active. *c)* and *d)* use the extension from §4.2.2. *b)* and *d)* use the extension from §4.2.3. For each hypergraph, the optimal $k = 2$ partition is shown. The number of ebits required in each case is: *a)* 4, *b)* 3, *c)* 3, *d)* 2.

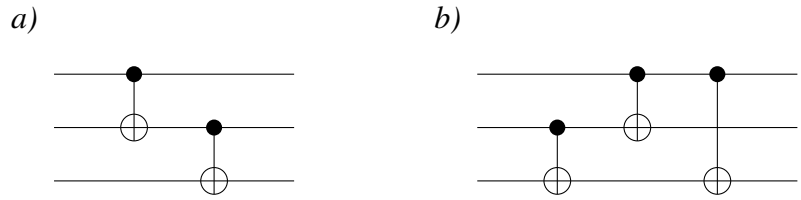



**Figure 4.13:** Equivalent circuits with the two gates in *a)* being interchanged in *b)*. A byproduct CNOT gate is created in the process.

we can take advantage of it and implement both gates using a single ebit. If both wires are in common and have 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 Figure 4.13, but that creates an additional CNOT per interchange.

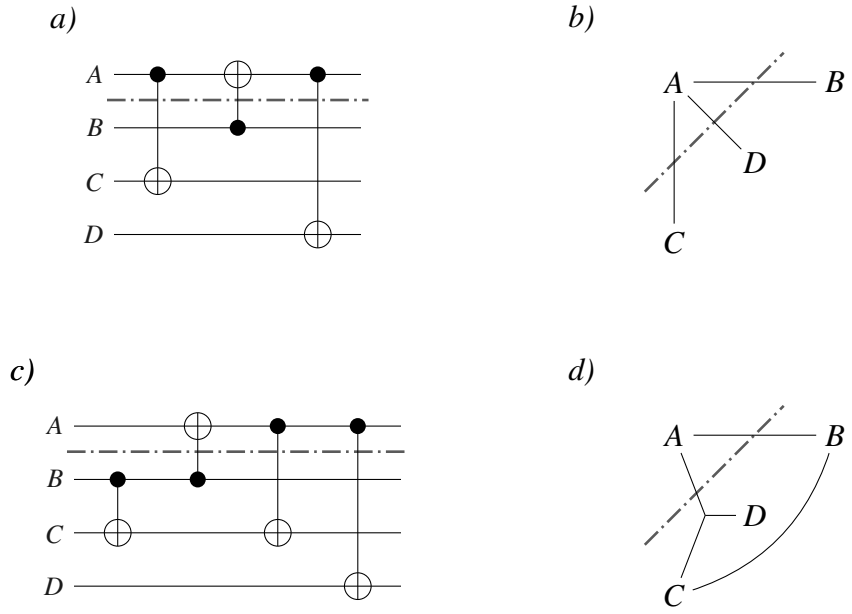



**Figure 4.14:** Example of a circuit *a)* where interchanging the first two gates saves an ebit for the given partition. The byproduct gate in circuit *c)* will be implemented locally, so its addition has no impact. Two of the CNOTs in *c)* can be implemented using a single ebit. Hypergraph *b)* corresponds to the input circuit. Hypergraph *d)* corresponds to *c)*, and has one less cut.

It may seem like pre-processing 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 4.14. In some cases interchanging CNOTs may unlock

a more efficient distribution of the circuit. Therefore, the effect of CNOT interchange should also be encoded in our hypergraph representation, if we were to exploit its full potential. However, encoding that information in a hypergraph is not natural, due to the following reasons:

- *The way CNOT gates are ordered in the circuit is important*: This is something we omit in the hypergraphs built by Algorithm 4.2; looking at the final hypergraph in Figure 4.10, we can not tell whether $\alpha$ goes before or after $\beta$. This information is key when interchanging, as it will determine which are the new neighbours of the interchanged CNOTs. A possible approach would be to impose that every hyperedge is an ordered list of vertices[8]. However, the standard hypergraph partitioning problem does not take into account such ordering. We would need to define a custom hypergraph partitioning problem in order to manage this new aspect.

- *Interchanging CNOT gates adds new CNOTs*: The CNOT interchange problem is substantially different from the one we discussed and solved in §4.2.3. The essence of our solution was to represent all of the potential choices in a single hypergraph. However, if were to interchange a pair of CNOTs, new choices would become available: Is it worth to interchange the CNOTs again with their new neighbours? Should the byproduct CNOT be itself interchanged further? Although the number of options to take into account is finite, it increases considerably fast. Furthermore, 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 be quite complex in order to accommodate this information.

Instead of encoding this choice within the hypergraph partioning problem, we may abandon the idea of guaranteeing the optimal solution, and approach this problem through pre-processing and post-processing. We suggest that a reasonable approach in that case would be to:

1. Apply a pre-processing phase that finds an equivalent circuit whose hypergraph has the minimum possible number of hyperedges.

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

2. Apply the procedure described in the previous section, which decides how to partition the circuit.

3. Apply a post-processing phase that exhaustively explores each possible interchange of CNOTs. Those that reduce the number of ebits required to implement the partition are used.

**TODO**: Is this a greedy algorithm? (most probably, a greedy algorithm won't be optimal... maybe dynamic programming?) If I have time I should I give more details on the strategy? Maybe implement it.

## 4.4 An upper bound

**TODO**: 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.

**TODO**: Discuss structured vs unstructured as the reason why we expect our algorithm to perform better.

**NOTE**: I probably won't do this section. The upper bound I've got is very loose, and the discussion does not seem to fit that well into this thesis.

# Chapter 5

# Evaluation

In the previous chapter we have proposed an algorithm that finds an efficient distribution of any given quantum circuit. Additionally, we have given two extensions (§**??** and §4.2.3) that allow the algorithm to apply some extra optimisations. In this chapter, we will evaluate the distributed circuits our algorithm generates for a collection of quantum programs. The quantum programs we consider have been discussed in the literature as examples of what we would like to run on quantum computers.

The code of our algorithm's implementation, all the generated executables and all the data discussed in this chapter can be found at: **TODO**: GitHub.

## 5.1 Implementation details

The algorithm described in §4.2.1 and its two extensions (§**??** and §4.2.3) have been implemented in Haskell. We chose Haskell because Quipper, a quantum circuit description language, is embedded on it. The quantum programs we use in our evaluation (see §5.2) are all available as part of the Quipper system, so using Haskell we could easily manage them. Our implementation provides a Quipper circuit as output, and thus it can be integrated in any Quipper program. Besides, the hypergraph partitioning is performed by an specialised third-party software, `KaHyPart` (Akhremtsev et al., 2017), that is called by our program when needed. A brief overview of `KaHyPart` is given in Appendix **??**.

The main contributions of this thesis were described in Chapter 4. Our implementation is meant to be a demonstration of our algorithm, and it is intended for evaluation purposes only. Thus, efficiency was not a main concern. Nevertheless, it is efficient enough to manage circuits with up to 300 wires within a reasonable amount of re-

sources (see §5.3).

Our implementation receives a circuit described in Quipper's internal data structure and outputs its distributed version, also in Quipper's format. However, along the process the circuit is managed as a list of gates, rather than using Quipper's internal data structure. This is the main cause of inefficiencies in our code. Quipper is presented to its users as a language to define circuits, rather than a language to define circuit transformations. Therefore, there are not enough functionalities available at user-level to fully implement our algorithm within Quipper. If we intended to achieve better efficiency, we would require to learn the internal workings – the back-end – of Quipper, which is beyond the scope of this thesis.

Once our input circuit is converted[1] from Quipper's data structure to a standard list of gates, implementing our algorithm is straight-forward. Depending on two input flags, the extensions from §**??** and §4.2.3 are applied or not; and the parameters $k, \varepsilon$ – number of QPUs and load-imbalance tolerance – are also given as input. The hypergraph generated by Algorithm 4.1 (or Algorithm 4.2) is written in a file using `KaHyPart`'s format, and the resulting partition is read from `KaHyPart`'s output file and used to create the distributed circuit as described in §4.2.1.

## 5.2 Test suite

The quantum programs we will use to evaluate our algorithm are available as part of the Quipper system. On the date this thesis is written, the Quipper system provides seven quantum programs, each with their own default configuration parameters. We use four of these seven examples, with their default configuration unless stated otherwise. The three programs we omit in our evaluation either lack an explicit – gate by gate – implementation of some fragment of their circuits, or their size is beyond the capabilities of our implementation (see §5.1). Detailed information about each of these quantum programs can be found in Quipper's online documentation[2]. The four programs we consider are the following:

- *Boolean Formula (BF)*: Ambainis et al. (2007) showed that the problem of evaluating a boolean formula over $N$ variables could be solved in time $\sqrt{N}$ on a quantum computer. We consider the circuit implementing the main part of the algorithm – the quantum walk. Implemented by A. Green.

[1] This conversion and its backwards counterpart are provided by Quipper.

[2] Link to Quipper's documentation: `https://www.mathstat.dal.ca/~selinger/quipper/doc/`.

- *Binary Welded Tree (BWT)*: Childs et al. (2002) proposed the problem of finding a path between two nodes in a particular kind of graph (a binary welded tree), and gave a quantum program that solves it exponentially faster than any known classical algorithm. We consider the circuit implementing the overall algorithm, making the tree height twice as large as the default Quipper's input. Implemented by P. Selinger and B. Valiron.

- *Ground State Estimation (GSE)*: Whitfield et al. (2011) proposed how to efficiently calculate the energy of a molecular system's ground state, which is relevant in chemistry. We consider the circuit implementing the overall algorithm. We doubled the number of basis functions and the number of occupied orbitals. Implemented by A. Green et al.

- *Unique Shortest Vector (USV)*: Regev (2004) proposed a problem where some characteristic vector of an input lattice must be found. This problem requires a large amount of resources, so we only consider a part of it, labelled 'R' in Quipper's library. We reduced the default dimension of the lattice from 5 to 2. Implemented by N. Ross.

Apart from these four programs, we will include in our test suite the circuit for the *Quantum Fourier Transform (QFT)*. The QFT is a key component of many quantum algorithms, Shor's factorisation algorithm being one of them. Its implementation is also provided in Quipper's libraries. In §5.3 we consider QFT-$N$, for different values of $N$, to refer to the QFT circuit of dimension $N$, which uses $N+1$ qubits.

## 5.3 Results

Our code was compiled using GHC version 8.0.2, with the optimisation flag `-O2` active. The tests we discuss in this section were run each on a single core of a i7-4710HQ processor, with 8GB of available RAM, on a Ubuntu 16.04 machine. The time it takes to distribute a circuit mainly depends on the number of CNOT gates it has. Table 5.1 shows the CNOT count and how long it takes to run the algorithm for each of the tests we discuss in this section. Beyond certain circuit size – around $50,000$ CNOTs –, the time it takes is considerably long. This is because our code was not implemented with scalability as a concern, rather than a characteristic of the algorithm itself.

Figure 5.1 shows, for different circuits, the proportion of CNOTs that become non-local once the circuit is partitioned. We also display the proportion of ebits the circuit

**Table 5.1:** CNOT count and the CPU time it takes to distribute each of the circuits discussed in this section. The data shown corresponds to executions with both extensions `pullCNOTs` and `EitherRemote` enabled and using distribution parameters $k=5,\ \varepsilon=5$.

| *Circuit* | *CNOTs* | *Time (s)* |
|---|---|---|
| BF | 25,590 | 64 |
| BWT | 13,824 | 15 |
| GSE | 70,158 | 675 |
| USV | 101,806 | 1273 |

| *Circuit* | *CNOTs* | *Time* |
|---|---|---|
| QFT-10 | 450 | 0.6 |
| QFT-25 | 3,000 | 3 |
| QFT-50 | 12,250 | 17 |
| QFT-100 | 49,500 | 255 |
| QFT-200 | 199,000 | **TODO** |

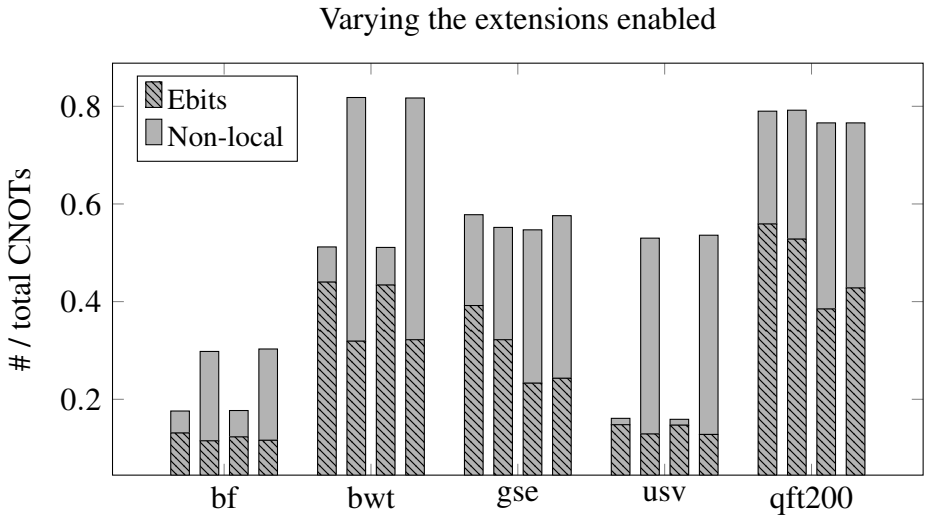



**Figure 5.1:** For each circuit, the bars correspond from left to right to: the vanilla algorithm, only `SlideCNOTs` enabled, only `EitherRemote` enabled and both extensions enabled. Each bar shows the number of non-local CNOTs and ebits required, normalised over the total number of CNOTs the original circuit had. The partition parameters are: $k=5, \varepsilon=0.03$.

requires. The key idea behind our algorithm was to take advantage of the remote-control and/or remote-target methods to implement multiple CNOTs using a single ebit. The larger the difference between the two proportions shown in Figure 5.1 is, the better this idea has paid off. However, the definitive measure of success – and what our algorithm aims to optimise – is the proportion of ebits itself.

Each of the four bars shown for every circuit corresponds to a different configuration of extensions enabled (from left to right): the vanilla algorithm, only `SlideCNOTs` enabled, only `EitherRemote` enabled and both extensions enabled. Among the four configurations, the best one is that whose proportion of ebits is the lowest[3]. Interest-

[3]Notice that for a given circuit, the number of CNOTs in it is independent of the extensions enabled.

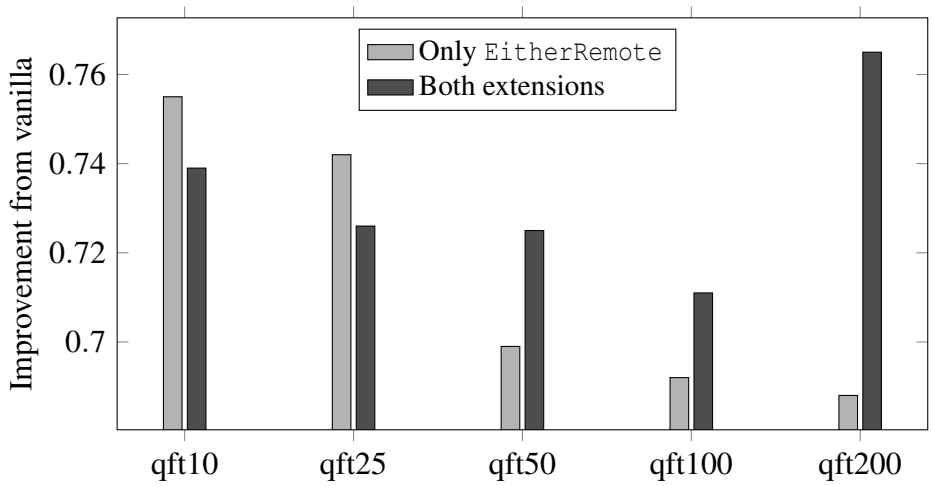



**Figure 5.2:** Improvement of the ebit count from vanilla, when using only `EitherRemote` or both extensions. The circuits considered are all QFT with different dimensions. Each circuit is distributed across five QPUs with a tight load-balance ($k = 5,\ \varepsilon = 0.03$).

ingly, in cases such as BWT when `SlideCNOTs` is enabled, the algorithm may find a more efficient solution by partitioning the circuit in a way that increases the amount of non-local CNOTs, but ultimately reduces the number of ebits.

As discussed in Remarks 4.5 and 4.9, in theory, enabling any of the two extensions should never increase the ebit count. However, Figure 5.1 shows that this does not hold in practice. This is because we were assuming that the optimal solution to the hypergraph partitioning problem was always found, but this is not the case in practice. Considering the hypergraph partitioning problem is NP-complete, a partitioner that ensures that the optimal solution is found can not possibly be efficient (unless $P = NP$). Whenever an extension is enabled, partitioning the hypergraph becomes a more complex task: `SlideCNOTs` increases the number of vertices reached by each hyperedge, while `EitherRemote` creates a hypergraph at least as twice as large. The problem in cases such as QFT-200 is that one of the extensions (`EitherRemote` in this case) does exceptionally well, while the other has little impact. Then, when both are enabled, the latter extension essentially only makes the partitioner's job more difficult, and when the hypergraph is large enough, this results in the partitioner yielding a worse solution. Figure 5.2 makes this evident, as for QFT with dimension 25 and lower, enabling both extensions is best: the hypergraph is small enough for the partitioner to find the optimal solution.

Still, in all of our tests, enabling any combination of extensions was always better

Thus, the number of ebits is normalised by the same value for each of the four configurations.

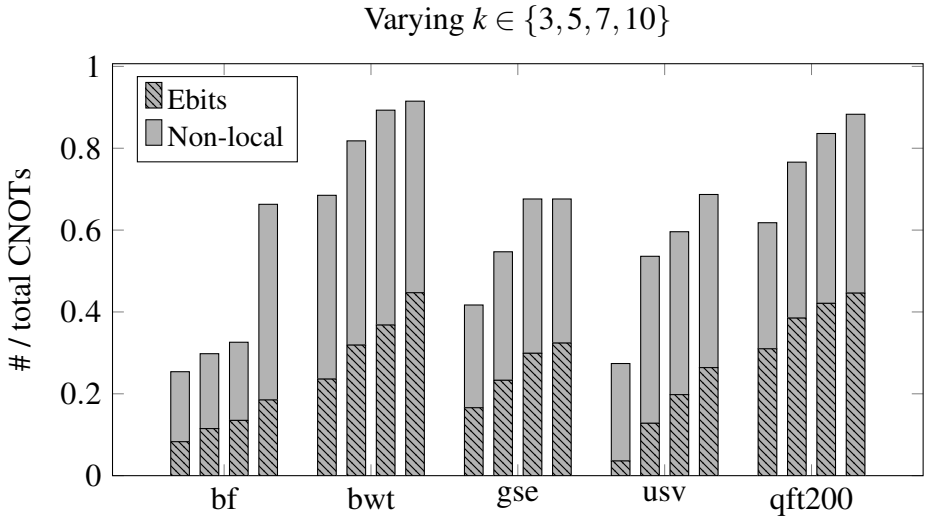



**Figure 5.3:** For each circuit, the bars correspond from left to right to a value of $k$: 3, 5, 7 and 10. Each bar shows the number of non-local CNOTs and ebits required, normalised over the total number of CNOTs the original circuit had. The results shown are for the best combination of enabled extensions in each case.

than running the algorithm with none. In practice, we would distribute the circuit with different extensions enabled, and choose the best one. Considering that, in all of our tests, enabling both extensions was always in the top two best configurations, we would propose it as the default configuration to use. This is similar to how classical compilers provide a default set of optimisation flags (e.g. $-O2$), but you may obtain a more efficient object code if you disable some of these flags manually.

Figure 5.3 shows the same information as in Figure 5.1, but now the different bars correspond to different values of the parameter $k$ – the number of QPUs the circuits are distributed across. For each circuit, its best configuration of extensions is used. As we would expect, distributing the circuit across more QPUs makes more CNOTs non-local and, correspondingly, the ebit count also increases. However, we can see how this increase is not necessarily uniform: When distributing the GSE circuit, the number of non-local gates and ebits required to distribute it across 10 QPUs is almost the same as if we distribute across 7 QPUs. The opposite situation happens for the BF circuit, as it seems to have a natural way of being partitioned across up to 7 QPUs – possibly because the qubits form groups that, for the most part, work isolated from other groups –, but beyond 10 QPUs wires that communicate intensively have to be separated – as a single QPU can not hold all of the qubits in a group. Still, in this particular case our algorithm manages to maintain a low ebit count.

Finally, Figure 5.4 shows the ratio between QPU space dedicated to communication

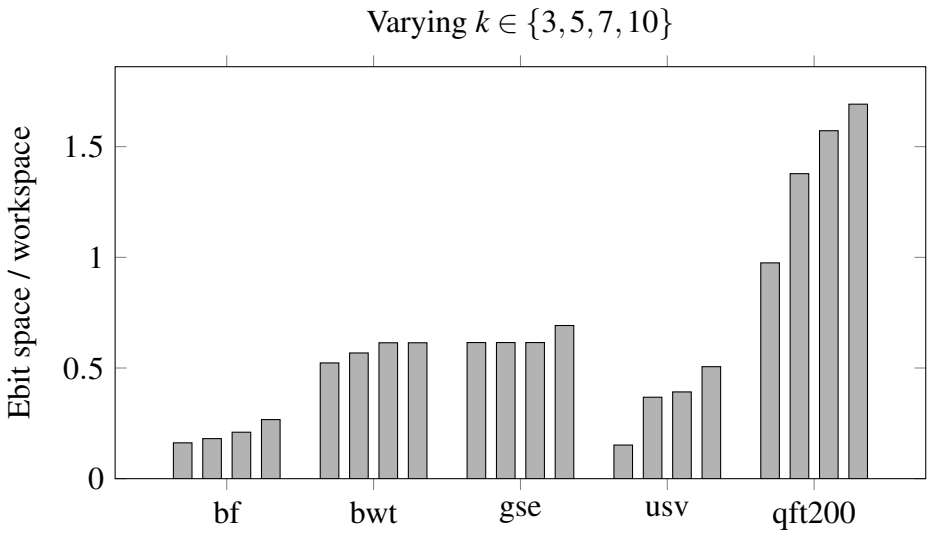



**Figure 5.4:** For each circuit, the bars correspond from left to right to a value of $k$: 3, 5, 7 and 10. Each bar shows the ratio between the space that must be dedicated to communication (i.e. managing ebit halves) and the space dedicated to the computation itself. The results shown are for the best combination of enabled extensions in each case.

(ebit space) versus space dedicated to computation (workspace qubits). A lower ratio is more desirable, as we want as much computation space as possible. Naturally, the ratio is worse as we increase the number of QPUs we distribute across, because the workspace required stays the same, while more ebits are needed (as we just saw in Figure 5.3). Therefore, there is a tension here, as we would like to use as many QPUs as possible, but we want them to manage a small space each. Figure 5.4 shows that this tension is different for each circuit: In cases such as BWT and GSE, the ratio barely changes, so if we decide to distribute them, we should use a large amount of QPUs. On the other hand, circuits such as QFT-200 are not good candidates for distribution, as using only 3 QPUs already doubles the amount of space needed, and the ratio keeps increasing for larger $k$.

# Chapter 6

# Conclusions

## 6.1 Further work

Better implementation (integrated in Quipper from a data-structure point of view; this should make the program faster)

Introduce more functionality: - CNOT interchange optimisation. - Black boxes: Sometimes we know some part of the circuit requires a lot of interaction between some qubits. If we tell the algorithm in advance to consider that circuit fragment as a black box which can not be distributed, the hypergraph should be way simpler, and therefore the partitioner should be able to find a better overall distribution. - Non-uniform partition: some QPUs get more vertices (this is trivial)

# Appendix A

# Hypergraph Partitioning

# Bibliography


Akhremtsev, Y., Heuer, T., Sanders, P., and Schlag, S. (2017). *Engineering a direct ¡italic¿k¡/italic¿-way Hypergraph Partitioning Algorithm*, pages 28–42.

Ambainis, A., Childs, A. M., Reichardt, B. W., Spalek, R., and Zhang, S. (2007). Any and-or formula of size n can be evaluated in time $n^{1/2+o(1)}$ on a quantum computer. In *Foundations of Computer Science, 2007. FOCS '07. 48th Annual IEEE Symposium on*, pages 363–372.

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.

Childs, A. M., Cleve, R., Deotto, E., Farhi, E., Gutmann, S., and Spielman, D. A. (2002). Exponential algorithmic speedup by quantum walk. *eprint arXiv:quant-ph/0209131*.

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.

Lyaudet, L. (2010). Np-hard and linear variants of hypergraph partitioning. *Theoretical Computer Science*, 411(1):10 – 21.

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*.

Regev, O. (2004). Quantum computation and lattice problems. *SIAM J. Comput.*, 33(3):738–760.

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.

Whitfield, J. D., Biamonte, J., and Aspuru-Guzik, A. (2011). Simulation of electronic structure hamiltonians using quantum computers. *Molecular Physics*, 109(5):735–750.

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