Parallelizing quantum circuit synthesis

by

Olivia Di Matteo

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Science
in
Physics - Quantum Information

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We present an algorithmic framework for parallel quantum circuit synthesis using meet-in-the-middle synthesis techniques. We also present two implementations thereof, using both threaded and hybrid parallelization techniques.

We give examples where applying parallelism offers a speedup on the time of circuit synthesis for 2- and 3-qubit circuits. We use a threaded algorithm to synthesize 3-qubit circuits with optimal T-count 9, and 11, breaking the previous record of T-count 7. As the estimated runtime of the framework is inversely proportional to the number of processors, we propose an implementation using hybrid parallel programming which can take full advantage of a computing cluster's thousands of cores. This implementation has the potential to synthesize circuits which were previously deemed impossible due to the exponential runtime of existing algorithms.

Acknowledgements

I am very grateful to my supervisor Michele Mosca for introducing me to an interesting area of work, and always being willing to try out new ideas. I also thank my advisory committee members Richard Cleve, Daniel Gottesman, and Roger Melko for their useful input on my progress, and Vlad Gheorghiu for reading through this entire thesis and providing very helpful comments.

I am lucky to have been able to consult with many senior (now former) members of our research group. David Gosset and Vadym Kliuchnikov are indispensable, and I can't thank them enough for helping me understand their T-count algorithm, offering suggestions for my implementations, and simply being around and willing to let me bounce ideas off them. Stacey Jeffery helped get me on the right track with complexity analysis of my algorithm, in addition to providing company as an office mate.

Conversations with Erik Schnetter about MPI were very helpful while getting started, and ultimately led to the ideas behind the hybrid parallel scheme of Chapter 6. I also thank Matt Amy, who's original meet-in-the-middle algorithm and code served as a starting point for my own implementation. Theo Belaire and Gary Graham helped sanity check my code when it did strange things.

I'm grateful to my parents for encouraging me to pursue a graduate degree, and Gary for supporting me throughout and making sure I took time to relax. Friends in the circuit synthesis group, IQC members, and my office mates Mária and Chunhao were very supportive and provided much-needed distractions. They also put up with my ramblings, such as "look at this pretty graph!", "my program found a claw!" and "all my computing jobs failed". I'm lucky to have had my coworkers at Udacity cheering me on from across the continent; my work there has taught me many useful things which helped with the writing of this thesis. Finally, I thank my undergraduate supervisor Hubert de Guise for his excellent mentorship, and for reminding me that a 50-page thesis will only get me 50%.

I acknowledge SHARCNET for use of their computing resources, and conservations with Jemmy Hu which taught me how to make best use of their machines. Funding for this research was provided by NSERC, the Department of Physics and Astronomy, and the Institute for Quantum Computing.

Contents

1 Introduction **1**
- 1.1 Quantum compilation . . . 1
- 1.2 Quantum circuits . . . 1
  - 1.2.1 Graphically representing circuits . . . 1
  - 1.2.2 Paulis, Cliffords, and universal gate sets . . . 2
- 1.3 Parallel computing . . . 3
- 1.4 Thesis objective . . . 5

2 Quantum circuit synthesis **6**
- 2.1 An overview of existing algorithms . . . 6
- 2.2 Meet-in-the-middle circuit synthesis . . . 8
- 2.3 An algorithm for the T-count . . . 10
  - 2.3.1 Smallest denominator exponents . . . 10
  - 2.3.2 The channel representation . . . 11
  - 2.3.3 Coset labels . . . 11
  - 2.3.4 Finding T-optimal circuit decompositions . . . 12

3 Parallel collision finding **15**
- 3.1 Overview . . . 15
- 3.2 Algorithm details . . . 15
  - 3.2.1 Collisions and claws . . . 15
  - 3.2.2 Collision finding . . . 16
  - 3.2.3 Claw finding . . . 18

4 Application of parallel framework to circuit synthesis **20**
- 4.1 Framework . . . 20
- 4.2 Runtime estimation and algorithm complexity . . . 21
- 4.3 Applications and use cases . . . 21

5 Implementation details and results of threaded algorithm **23**
- 5.1 MITM for optimal T-count circuit synthesis . . . 23
- 5.2 Implementation . . . 24
  - 5.2.1 Language and computer specifications . . . 24
  - 5.2.2 Special techniques to make the program faster . . . 24
  - 5.2.3 Program flow . . . 26
- 5.3 Results . . . 30
  - 5.3.1 Timing random parallel code . . . 30
  - 5.3.2 2-qubit synthesis . . . 30

5.3.2.a Controlled-Hadamard . . . 30
5.3.2.b Controlled-phase . . . 32
5.3.2.c Varying the T-count . . . 32
5.3.3 3-qubit synthesis . . . 33
5.3.3.a T-count 7 . . . 33
5.3.3.b A new regime: T-count 9 and T-count 11 . . . 38
5.3.3.c Varying the number of threads . . . 39
5.3.3.d Varying the fraction of distinguished points . . . 40
5.4 Algorithm limitations . . . 41

6 Advanced implementation: hybrid parallel programming 42
6.1 Program flow . . . 42
6.2 Preliminary results . . . 43
6.3 Advantages and limitations . . . 45

7 What else can we do with this framework? 47

References 50

A Binary symplectic representation 52

List of Figures

1.1 Quantum circuit depth 2
1.2 Commonly used gates in quantum circuits 3

2.1 Timeline of circuit synthesis algorithms 7

3.1 Schematic of trail used in collision finding 16
3.2 Trails and distinguished points 16
3.3 Colliding trails 17
3.4 Claw-finding trails 19

5.1 Algorithm flowchart for OpenMP implementation 28
5.2 Subroutine flowcharts 29
5.3 Controlled-Hadamard 31
5.4 Histogram of controlled-Hadamard runtimes 31
5.5 Controlled-phase 32
5.6 Histogram of controlled-phase runtimes 33
5.7 Circuit series with varying T-count 33
5.8 Synthesis times of circuits with varying T-count 34
5.9 3-qubit circuits with T-count 7 35
5.10 Histogram of Toffoli synthesis times 35
5.11 Histogram of Peres synthesis times 36
5.12 Histogram of QOR synthesis times 36
5.13 Histogram of negated Toffoli synthesis times 37
5.14 Histogram of negated Fredkin synthesis times 37
5.15 Histogram of negated Fredkin synthesis times with outliers removed 38
5.16 3-qubit circuits with T-count 9 and T-count 11 38
5.17 Synthesis times for T-count 9 circuit 39
5.18 Synthesis times for T-count 11 circuit 39
5.19 Synthesis time vs. thread count 40
5.20 Synthesis time vs. distinguished point fraction 40

6.1 Algorithm flowchart for hybrid OpenMP/MPI implementation. 44

List of Tables

5.1 Average runtimes for 3-qubit circuits with T-count 7 35

6.1 Average runtimes for Toffoli synthesis with varying MPI process distribution 45

Chapter 1

Introduction

1.1 Quantum compilation

Compilers are an indispensable tool for programmers. Computers do not understand a human-readable programming language, such as C or Java, directly - they understand only a fixed set of instructions known as assembly, or machine code. Any operation you can imagine can ultimately be broken down into a sequence of the building blocks which comprise machine code. Writing machine code, however, is very cumbersome and prone to errors. The task of a compiler is to "translate" from human-readable code to machine code so that the computer can understand and execute the desired operations.

A quantum computer, once one is built, will require something akin to a compiler. The quantum compilation process will look very different from its classical counterpart, but the underlying idea remains the same: a quantum computer will need a translator. It may only know how to implement a fixed set of operations both efficiently and fault-tolerantly (i.e. with proper error-correction protocols). It will thus need a program with the ability to take any arbitrary operation it is given, and turn it into something it can understand and implement well. This part of the compilation process is called *quantum circuit synthesis*, and its implementation is the focus of this thesis.

1.2 Quantum circuits

1.2.1 Graphically representing circuits

Operations that run on a standard quantum computer are represented as unitary matrices. We commonly organize sequences of these operations graphically in the form of circuits. These circuits are somewhat analogous to the Boolean logic circuits of classical computers, but with operations acting on qubits rather than bits.

Suppose we have n qubits, and we would like to execute a sequence of unitary operations on them. Let us start with our qubits in some initial state $|\psi_0\rangle$. Say we want to apply the operations $\{U_1, \ldots, U_d\}$ to the qubits, in that order, with $U_i \in \mathcal{U}(2^n)$ (the set of $2^n \times 2^n$ unitary matrices).

In matrix notation, the state vector $|\psi_0\rangle$ evolves as follows:

$$
\begin{aligned}
|\psi_1\rangle &= U_1|\psi_0\rangle && (1.1)\\
|\psi_2\rangle &= U_2|\psi_1\rangle = U_2U_1|\psi_0\rangle && (1.2)\\
&\vdots \\
|\psi_d\rangle &= U_d|\psi_{d-1}\rangle = U_dU_{d-1}\cdots U_2U_1|\psi_0\rangle. && (1.3)
\end{aligned}
$$

Instead of writing the evolution of the state in equation form, we can write it as a circuit diagram (shown in Figure 1.1).

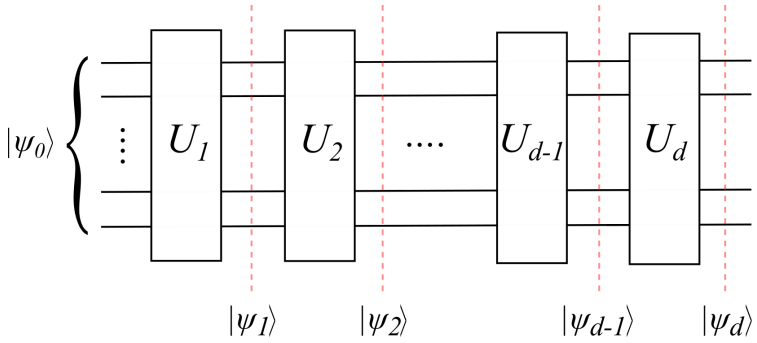


Figure 1.1: A simple circuit diagram acting on a register of qubits. Unitary operations $U_1, \ldots U_d$ are applied, and the resultant state is labeled below (corresponding to Eqs. (1.1-1.3)). As the circuit has d operations, or layers, we say this circuit has *depth* d.

Each horizontal line in Figure 1.1 represents a qubit. A group of qubits is called a *register*. Unitary operations are represented as boxes; they act only on the qubits whose lines they sit atop. Each unitary operation can be imagined as one *layer* of a circuit. The number of layers in the circuit is called the *depth*. Note that the ordering of the operations pictured appears reversed. Operations in the circuit are applied in the order in which they are read, from left to right, whereas the matrix operations in Eqs. (1.1 - 1.3) operate from right to left.

1.2.2 Paulis, Cliffords, and universal gate sets

Just like machine code provides a classical computer with a fixed set of instructions, a quantum computer has an analogous fixed set of unitary operations which can be used to implement an arbitrary operation. Such a set is termed a *universal gate set*. There are many of examples of universal sets; we will focus primarily on just one of them.

We introduce here a few very important gates which are frequently used in quantum operations: the single-qubit Hadamard gate H; the phase gate S; the $\frac{\pi}{8}$ gate T; and the 2-qubit controlled-NOT gate CNOT. Their matrix representations and circuit diagram elements are shown in Figure 1.2.

Another group of important gates is the familiar set of single-qubit Pauli operators:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}. \tag{1.4}$$

These matrices and their n-qubit tensor products form a group which we will call $\mathcal{P}_n$, the n-qubit Pauli group. We define the n-qubit Clifford group $\mathcal{C}_n$ as the group of operations which are the *normalizer* of the Pauli group. In other words, for every $P \in \mathcal{P}_n$, there exists a $P' \in \mathcal{P}_n$ such that

$$CPC^\dagger = (-1)^b P', \qquad C \in \mathcal{C}_n, \quad b \in \{0, 1\}. \tag{1.5}$$

Essentially Cliffords map Paulis to Paulis (up to a possible phase of -1) under conjugation.

The Clifford group can be specified using some of the gates in Figure 1.2. Its generators are

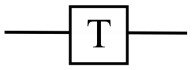
$$\mathcal{C}_1 = \langle H, P \rangle, \tag{1.6}$$

$$\mathcal{C}_n = \langle H_{(i)}, S_{(i)}, \mathrm{CNOT}_{(i,j)} \rangle, \quad i, j = 1, \ldots, n, \qquad i \neq j, \quad n \geq 2, \tag{1.7}$$

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

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix}$$



$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

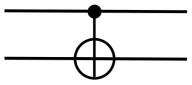


Figure 1.2: Commonly used quantum operations and their graphical representation in circuit diagrams.

where we borrow the notation of [1] and use subscripts to denote which qubits the gates act on.

It is well known that the Hadamard and T gate can be combined to produce arbitrarily precise approximations of any single-qubit unitary [2]. With the addition of the 2-qubit CNOT, this becomes universal, meaning any n-qubit gate can be written in terms of only 1- and 2-qubit gates [3, 4]. In this thesis, we focus on this so-called "Clifford+T" universal gate set, which consists of $\mathcal{C}_n$ and the single-qubit T gate [2, 5].

It is important to note that the Clifford+T set is not unique - there are other possible choices, for example the Clifford group and the Toffoli gate [2]. A quantum computer, depending on its physical implementation, may be very good at performing just one of these gates sets efficiently and fault-tolerantly. At this time, many error correction and fault-tolerant protocols work efficiently with Clifford gates [6], so the focus of many existing circuit synthesis algorithms [7, 8, 9, 10, 11, 12] has been on this set.

1.3 Parallel computing

Advances in classical computing technology have seen processing speeds increase by orders of magnitude in the last decade or so. This is largely accomplished by making smaller chips, with smaller components packed closer and closer together - the closer the transistors are, the less distance the information must travel, and thus the faster the processor runs. Furthermore, placing transistors closer together allows us to fit more of them on a chip of the same size. This trend can not continue forever, though. There is physical limit, at the atomic level, to how close you can place two transistors together. Furthermore, a computed bound on how fast a quantum state can change showed that there is a limit to the operation rate of *any* system [13]. At some point, we simply won't be able to make processors any faster. Our computational problems, however, are only getting bigger. So, if we can't make our processors faster, the next best thing is to use *more*. This is the essence of parallel computing: rather than a single processor doing all the work, work is distributed among multiple processors.

Take the simple example of image processing, for instance. Suppose we are working with an image containing millions of pixels, and we want to apply some operation to each pixel to change the appearance of the image (for example, changing it to gray scale, or making it brighter). One way to approach this problem would be to use a single processor and iterate sequentially over each pixel and apply the trans-

formation. The parallel approach would be to send groups of pixels to different processors, and process all the groups simultaneously. Even if the time for each parallel processor to do a single transformation is greater than that of the single one, parallelism will outperform the single processor because everything is processed at the same time. For example, in a video game, where images are processed and rendered continuously at high frame rates, parallelization is essential.

Aside from graphics, parallel programming has found many uses in the scientific community. For example, in astrophysics, simulating supernovae in large regions of space is accomplished using a grid of processors, each assigned to a physical coordinate - differential equations are then discretely solved on the appropriate processor in every region of space. Many linear algebra operations can also be parallelized, such as matrix multiplication - give each processor a different row and column index to multiply together and all new matrix entries can be computed simultaneously.

A problem like image processing is called *embarrassingly parallel* - essentially, the entirety of the algorithm can be run in parallel. True embarrassingly parallel problems are extremely rare, as generally at least some of the runtime is spent initializing data serially (or waiting). Other problems contain only certain sections which can be parallelized. The speedup one obtains depends on the degree of parallelizability of the problem, and is determined by a relationship called Amdahl's law [14]. Suppose that a fraction P of a computing problem is parallelizable (the remaining $1-P$ must be done sequentially). If we use N processors for a problem, Amdahl's law tells us that the maximum possible speedup we could expect is [15]

$$S(N) = \frac{1}{(1-P) + P/N}. \tag{1.8}$$

If $P = 1$ (embarrassingly parallel), we can easily see that the speedup would be linear in the number of processors. If $P \neq 1$, and we simply increase the number of processors, the limit of the speedup tends to $1/(1-P)$ - this means that for some problems we will never see much of a speedup, regardless of how many processors we use. Thus, choosing whether or not to use parallel computing is highly dependent on the problem at hand.

There are many different types of hardware and techniques which can execute code in parallel. One such example is the GPU (Graphical Processing Unit). These are specialized cards which contain thousands of tiny processors, and are used primarily for image processing on millions of pixels. The processors themselves are not particularly powerful, but their sheer number provides a huge advantage [16]. In the image processing example provided above, using a GPU would be the most effective means of solving that problem. GPUs, however, require special programming techniques and also suffer from bottlenecks due to the time spent transferring memory between the GPU and the host computer. This is a point we will return to in Chapter 7.

In this thesis, we focus on parallel computing using only CPUs (Central Processing Units). We will often speak of *nodes*: a collection of processors, or cores. There are many different computational models using CPUs. They differ in how the nodes are used, organized, and how the memory is shared.

A simple approach is called threading. Threading is often accomplished by placing instructions called *directives* around the code we wish to run in parallel, using for example a library such as OpenMP [17]. Suppose we have a single node with 16 cores. Each core may be assigned a single thread. Generally, all the threads are allowed to access a shared memory bank. Threads cannot pass messages between themselves, but can exchange information using the shared memory. The number of threads, however, is limited by the number of physical cores on the node (one can *hyperthread* and assign multiple threads to each core, but this often results in poor performance).

A way around this limitation is to use a library such as MPI (Message Passing Interface) [18] which provides a means of communication between different nodes, each of which has multiple cores. Each node has its own memory - another node cannot access that memory, and must be sent any data it requires

via messages. Implementations of such programs are more complex, as one has to deal not only with the problem at hand, but to coordinate a number of nodes and make sure all the data gets passed between them correctly. This approach is better than simple threading, however it may not take full advantage of the fact that each node has multiple cores.

The final approach, which we explore in Chapter 6, is *hybrid* programming. MPI is used to send messages between nodes, while threading is used within each node to further subdivide the problem and take advantage of its multiple cores. There are a multitude of ways to divide up the computation and the message passing done by the threads [19, 20]. These techniques make use of the entire node, and are thus advantageous for many problems.

1.4 Thesis objective

The goal of this work is to develop and implement an algorithm for quantum compilation (quantum circuit synthesis) which leverages parallel computing techniques. Full-scale implementations of quantum computers will emerge in the (perhaps not-so-distant) future. When this occurs, having a working algorithm which can efficiently compile large operations on many qubits will be extremely advantageous and will allow for faster testing, development, and progress.

This thesis is organized as follows. Chapter 2 introduces the main ideas behind quantum circuit synthesis, and details some of the key algorithms of the field. Chapter 3 explores the framework of parallel collision finding techniques and the advantages thereof. Chapter 4 combines the ideas of Chapters 2 and 3 and explains in detail how to apply parallel collision finding to circuit synthesis. We present a general framework for these techniques, and provide complexity and runtime estimates of the algorithm. Chapter 5 presents a fully-functioning, threaded parallel implementation of this framework applied to a specific synthesis algorithm. Chapter 6 details the hybrid parallel implementation, which is in progress. Finally, in Chapter 7 we discuss the future uses and improvements of our scheme.

Chapter 2

Quantum circuit synthesis

Quantum circuit synthesis is an important part of the quantum compilation process. Let us return to the analogy of classical compilers: a compiler translates human-readable code into computer-readable code. Quantum circuit synthesis is quite similar in spirit. A quantum computer will not be able to implement just any operation requested of it. Instead, it will understand and be able to efficiently, and fault-tolerantly, implement only a small subset of gates: those of the so-called universal gate set.

The goal of quantum circuit synthesis is, given an arbitrary operation, to break it down, or *synthesize*, an equivalent circuit using only gates from the universal set. In other words, given an arbitrary operation U, quantum circuit synthesis is the process which constructs the sequence of operations $U_1, \ldots, U_k$ from a universal gate set $\mathcal{G}$ such that

$$U_k \cdots U_1 = U. \tag{2.1}$$

There are two types of synthesis: *exact*, and *approximate*. Exact synthesis constructs a sequence $U_1, \ldots, U_k$ such that Eq. (2.1) holds, and their product is exactly equal to U. On the other hand, approximate synthesis yields a set of operators whose product is within some distance ε from the target unitary U, where the distance between two operators is measured using some distance function. The work presented in this thesis focuses on exact synthesis.

2.1 An overview of existing algorithms

A number of algorithms exist for circuit synthesis; they are varied in their nature and techniques, and in what they can accomplish. Figure 2.1 is a timeline which shows the development history of such techniques. At the beginning is one of the pillars of circuit synthesis, the Solovay-Kitaev (SK) algorithm [21]. The SK algorithm allows one to efficiently perform approximate synthesis of a single-qubit[1] operation over a given set of gates. Synthesis is performed by recursively generating a sequence of gates which at every step moves closer and closer to the target gate, up to a distance of ε. The complexity analysis of this algorithm shows that it scales very well - it runs in time $\mathcal{O}(\log^{2.71}(1/\varepsilon))$ and produces a sequence of gates with length $\mathcal{O}(\log^{3.97}(1/\varepsilon))$. This algorithm is also advantageous because it works over any arbitrary gate set. Provided that [21]:

1. All gates in the set are unitary and have determinant 1,

2. For each gate in the set, its adjoint is also in the set,

[1]The SK algorithm can be generalized to both single qudits, as well as multiple qubits. In the qudit case, however, the runtime scales much worse, increasing exponentially with the dimension of the system [21]. For consistency and for comparison with the other algorithms discussed here, we restrict ourselves to the single-qubit version.

3. The chosen set is *dense* over the set of all single qubit unitaries $SU(2)$, i.e. for any ε, an arbitrary operation chosen from $SU(2)$ will always be within ε of some operation in the gate set,

we can always find an approximation over it. The disadvantages, however, are just that - we can only ever find an approximation.

Other approximation algorithms were developed which improved on the runtime of the SK algorithm. For example, with the addition of two ancillary (i.e. extra) qubits, the algorithm in [7] can synthesize a single-qubit operation over the Clifford+T set with only $\mathcal{O}(\log(1/\varepsilon))$ gates in time $\mathcal{O}(\log^2(1/\varepsilon)\log\log(1/\varepsilon))$. An algorithm without ancillae was developed shortly afterward in [8]. It is a randomized algorithm, based on some principles of number theory, which approximates single-qubit circuits to Clifford+T gates with (heuristic) runtime polynomial in $\log(1/\varepsilon)$. The same paper also proved a bound on the T-count of the resultant circuits. The algorithm of [9] found a means of optimally implementing Z-rotations over Clifford+T, also with no ancillae and optimal T-count, up to an accuracy of 10^{-15}. The resultant algorithm however, took time and memory exponential in the T-count. A subsequent algorithm [10] uncovered a means of finding these optimal T-count Z rotations efficiently. Optimizing the number of T gates is a pertinent problem, as T gates are 'expensive' to implement fault-tolerantly (see Section 2.3).

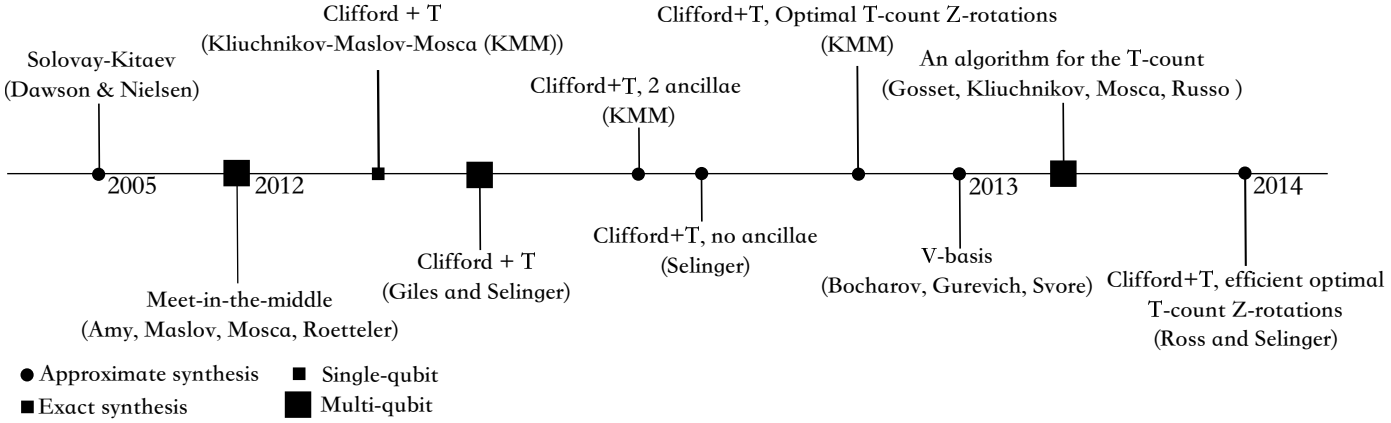


Figure 2.1: A timeline showing recent advances in quantum circuit synthesis. Algorithms are ordered chronologically based on their first appearance on the arXiv.

Approximations inevitably introduce some error. In a large quantum computing protocol with many operations, continuous approximations lead to the accumulation of these errors. We thus want to synthesize the gates as precisely as possible; it is even better if we can synthesize the gates *exactly*. An exact synthesis algorithm for single-qubit gates was introduced in [11]. Here the gate set is limited to the Clifford+T set, as we can take advantage of the fact that all the matrix elements of the Clifford+T gates are in the following ring:

$$\mathbb{Z}\left[i, \frac{1}{\sqrt{2}}\right] = \left\{ \frac{a + b\sqrt{2} + ci + di\sqrt{2}}{\sqrt{2}^k} : \quad a, b, c, d \in \mathbb{Z}, \quad k \in \mathbb{N} \right\}. \tag{2.2}$$

As a result, any combination of these gates also has elements over this ring. Thus, given an arbitrary operation with elements over this ring, there should be a sequence of Clifford+T gates which produce it.

The algorithm uses special properties of the ring structure to reverse-engineer, or unravel, a sequence of H and T which produced it. The synthesis algorithm makes use of patterns in the power k of $\sqrt{2}$ in the denominators of matrix elements which occur after successive multiplications by H and T: when considering the absolute value squared of the matrix elements, k increases by 1 each time the combination (HT) is applied. A small set of combinations of H and T are precomputed and stored in a database.

Next, synthesis is performed by iteratively producing a sequence of H and T combinations, starting from the value of k observed in the candidate matrix, and unraveling it all the way down to the level of the database, in which a search for the remaining part of the sequence is done. The resultant sequence is *optimal* [11].

This algorithm can indirectly perform approximate synthesis: for an arbitrary gate, use the SK algorithm, or any of the above approximation algorithms to produce an approximation of the gate with all elements over $\mathbb{Z}\left[i, \frac{1}{\sqrt{2}}\right]$. Then, use the exact synthesis method to find a decomposition of the approximated gate. The downside to this exact synthesis technique is of course that it is a single-qubit algorithm. A conjecture was made for a similar technique for multiple qubits, but would require ancillae [11].

The need for ancillae was considered in [12]. An exact synthesis algorithm for multiple qubits was provided, and it was proven that though ancillae are necessary, only a single one is ever required. Their algorithm is far from optimal however - the number of gates in the synthesized circuits is of the order $\mathcal{O}(3^{2^n} nk)$ where n is the number of qubits and k is the highest denominator exponent of ring elements in the target operation. Reference [12] also details further conditions on which operations can be exactly synthesized without any ancillae. Due to the fact that the determinant of a product of matrices is the product of their determinants, these exact synthesis algorithms are limited to synthesizing gates which not only have elements over $\mathbb{Z}\left[i, \frac{1}{\sqrt{2}}\right]$ but also have determinants which belong to the set of determinants possible from combinations of gates of the universal set. For example, for 2 qubits the set is $\{1, -1, i, -i\}$. Any 2-qubit gate which does not have a determinant here, for example the controlled-T gate, requires a single ancilla for exact synthesis.

Finally, we note that algorithms also exist for synthesis over other universal sets. The so-called V-basis is a universal gate set being explored as an alternative for the Clifford+T set. There are six gates in the V-basis and they represent rotations by $\cos^{-1}(-3/5)$ around the x, y and z axes. An approximation algorithm for synthesizing single-qubit circuits over the V-basis (and allowing Pauli operations) was presented in [22]. It was previously proven that decomposition over this basis could be done in time linear in $\mathcal{O}(\log(1/\varepsilon))$ [23]. Two variations were presented in [22], one a randomized algorithm which achieves this expected runtime using $\leq 12 \log_5(2/\varepsilon)$ gates, and the other a direct search which produces circuits 1/4 to 1/3 of this length, and runs in time $\mathcal{O}(\log^3(1/\varepsilon) \log\log(1/\varepsilon))$.

2.2 Meet-in-the-middle circuit synthesis

Perhaps the most important recent advance in quantum circuit synthesis is the meet-in-the-middle (MITM) approach considered in [24]. This algorithm forms the basis of the parallel framework we later present, so it is prudent to devote some time to understanding both how it works, and its advantages and disadvantages.

The MITM approach relies on generating a large database of possible circuits, which is subsequently searched through to find a solution. It begins with a gate set $\mathcal{G}$ - we will assume that $\mathcal{G}$ is a universal set suitable for quantum computing applications. Let γ be the number of single-qubit gates in $\mathcal{G}$. Recall that in the quantum circuit formalism, we can arrange the circuit in *layers*; the total number of layers in a circuit is its *depth*. Hence, we define the set $\mathcal{V}_{(n,\mathcal{G})}$, where n is the number of qubits, to be the set of all operations which constitute a single layer of depth in the circuit. Given that there are γ single-qubit gates, using simple combinatorics we can compute an upper bound of the size of this set: $|\mathcal{V}_{(n,\mathcal{G})}| \leq \gamma^n$.

Now, suppose we are presented with an arbitrary unitary operation U. The goal of circuit synthesis is to find a set of $\{U_i\}, i = 1, \ldots, k$ such that

$$U_k U_{k-1} \cdots U_1 = U, \tag{2.3}$$

where all $U_i \in \mathcal{V}_{(n,\mathcal{G})}$, and the depth k is optimal.

The brute force approach would be to take our set $\mathcal{V}_{(n,\mathcal{G})}$, generate all possible combinations of the elements, from depth 1 up to depth k, and check if any of them are equal to U. The runtime of such an algorithm would be abysmal: $\mathcal{O}(|\mathcal{V}_{(n,\mathcal{G})}|^k)$ or $\mathcal{O}(\gamma^{nk})$, which is exponential in both the number of qubits and the depth. Such an algorithm would hardly scale well past small numbers of qubits and low depth.

The MITM approach offers roughly a square-root speed-up over the brute force case. Rather than generating combinations for all k possible layers, we split Eq. (2.3) as follows:

$$U_{\lceil \frac{k}{2} \rceil} \cdots U_1 = U^\dagger_{\lceil \frac{k}{2} \rceil + 1} \cdots U_k^\dagger U, \tag{2.4}$$

$$V = W^\dagger U, \tag{2.5}$$

where we define

$$V := U_{\lceil \frac{k}{2} \rceil} \cdots U_1, \tag{2.6}$$

$$W := U_k \cdots U_{\lceil \frac{k}{2} \rceil + 1}. \tag{2.7}$$

We can see here an immediate advantage: rather than generating all possible combinations of circuits up to depth k, we only need to generate the sets up to depth $\lceil \frac{k}{2} \rceil$.

Let S_i represent the set of combinations of $\mathcal{V}_{(n,\mathcal{G})}$ having depth i, and let $S_i^\dagger$ be the set containing the corresponding adjoints having depth i. Then, a solution to the synthesis problem exists if we can find some $V \in S_{\lceil \frac{k}{2} \rceil}$ and $W \in S^\dagger_{\lfloor \frac{k}{2} \rfloor}$ such that Eq. (2.5) holds. In other words, a solution exists if we find some k such that $S_{\lceil \frac{k}{2} \rceil} \cap S^\dagger_{\lfloor \frac{k}{2} \rfloor} U \neq \emptyset$.

The algorithm runs as follows: we begin with $i = 1$, and sequentially generate the next S_i by building off the previous set one layer at a time. We take $S_0 = \mathbb{1}$ and $S_1 = \mathcal{V}_{(n,\mathcal{G})}$. For each new set S_i, a search is executed to compare it with its possible 'other halves' - sets $S_{i-1}^\dagger U$ and also $S_i^\dagger U$, taking into account the fact that the depth k may be odd or even, respectively. The algorithm terminates if a match is found in one of these sets; otherwise, it continues to the next level of depth, up to a specified limit k.

What is the runtime of this algorithm? For this, we look to the best classical search algorithms. The data structure used to store the sets S_i plays an important role in how long the searching process takes. A well-chosen structure such as a binary tree or red-black tree can be completely searched for matches from a newly generated S_i in time $\mathcal{O}(|S_i| \log(|S_i|))$. Searching itself takes $\mathcal{O}(\log(|S_i|))$ [25], and this must be done for all $|S_i|$ elements.

Thus, at each step in the algorithm, comparing the new set with the other halves would take time $|S_i| \log(|S_{i-1}^\dagger|) + |S_i| \log(|S_i^\dagger|) \leq 2|S_i| \log(|S_i|)$, since the i^{th} set is always larger than the $(i-1)^{\text{th}}$, and $|S_i| = |S_i^\dagger|$ [24]. Furthermore, we know that $|S_i| \leq |\mathcal{V}_{(n,\mathcal{G})}|^i$. Thus, we obtain an upper bound of $\mathcal{O}(|\mathcal{V}^i_{(n,\mathcal{G})}| \log(|\mathcal{V}^i_{(n,\mathcal{G})}|))$ for the i^{th} iteration.

To obtain an upper bound for the total runtime needed to find a circuit of optimal depth k, we sum over all depths i [24]:

$$T_{MITM} = \sum_{i=1}^{\lceil \frac{k}{2} \rceil} |\mathcal{V}_{(n,\mathcal{G})}|^i \log(|\mathcal{V}_{(n,\mathcal{G})}|^i) \tag{2.8}$$

$$\leq \sum_{i=1}^{\lceil \frac{k}{2} \rceil} |\mathcal{V}_{(n,\mathcal{G})}|^i \log(|\mathcal{V}_{(n,\mathcal{G})}|^{\lceil \frac{k}{2} \rceil}) \tag{2.9}$$

$$\leq |\mathcal{V}_{(n,\mathcal{G})}|^{\lceil \frac{k}{2} \rceil} \log(|\mathcal{V}_{(n,\mathcal{G})}|^{\lceil \frac{k}{2} \rceil}), \tag{2.10}$$

where in each successive step we have used the highest exponent to generate an upper bound. Since $|\mathcal{V}_{(n,\mathcal{G})}| \leq \gamma^n$, we can rewrite the above as $\mathcal{O}(\gamma^{n\lceil \frac{k}{2} \rceil} \log(\gamma^{n\lceil \frac{k}{2} \rceil}))$. This is roughly a square root improvement over the brute force result of $\mathcal{O}(\gamma^{nk})$.

In [24], this framework was used to generate circuits with a number of properties, such as optimal depth, optimal T-depth (i.e. optimizing the number of layers which contain T gates), and minimization of specified cost functions. Some of the largest depth-optimal circuits synthesized were depth 12 for 2 qubits, 10 for 3 qubits, and 6 for 4 qubits.

The disadvantages of the MITM algorithm lie in the fact that databases must be generated and stored. This is of course a procedure which must only be done once - subsequent executions of the program can reuse the databases as synthesis is always taking place over the same gate set. Databases were stored as red-black trees in binary files generated by the program. Rather than storing entire unitaries, storage was simplified by storing only the action of each candidate unitary on an arbitrary vector 'key' - in a sense, it stored a type of hash which was highly likely to be unique to each unitary [24]. Furthermore, the databases were pruned by choosing canonical representatives of unitaries which differed only by a global phase. For example, to synthesize the Toffoli gate, the largest database file which needed to be generated was about 36 MB in size and took about 24 minutes to generate. Subsequent searching for matches through the largest database took about 7.5 minutes [2]. The time and space, however, are highly dependent on the synthesis problem at hand. For example, in the case of optimal T-depth circuits, generation of the 3-qubit Clifford group required almost four days [24]. Regardless, the principles of MITM synthesis offer significant benefits which we take advantage of in Chapter 4.

2.3 An algorithm for the T-count

Most of the algorithms presented above synthesize circuits over the Clifford+T gate set. However, T gates are an 'expensive' resource. To implement a T gate fault-tolerantly requires the addition of an ancilla qubit, a fault-tolerant measurement procedure, and additional fault-tolerant operations of other gates [2, 26]. The T-count of a circuit is the total number of T and $T^\dagger$ gates it contains. Due to their 'cost', it is important to have an algorithm which produces circuits with optimal T-count. Such an algorithm was developed in [1] - it determines whether the T-count of a unitary is less than a candidate value, and finds the corresponding circuit decomposition. We proceed by first stating a few key concepts that we use here and in future chapters. We then go through a brief overview of how the algorithm determines optimal T-count circuits.

2.3.1 Smallest denominator exponents

Consider an element over the ring $\mathbb{Z}\left[i, \frac{1}{\sqrt{2}}\right]$. Similar to how we can reduce simple fractions when there are common factors, we can also reduce ring elements so that the coefficients are as small as possible.

Definition 1 (sde)**.** *The* smallest denominator exponent *of x,* $\mathrm{sde}(x)$*, is defined as the smallest value of k such that*

$$x = \frac{a + b\sqrt{2} + ci + di\sqrt{2}}{\sqrt{2}^k}, \tag{2.11}$$

holds, where a, b, c, d are integers and $k \geq 0$ [1, 11]. We also define $\mathrm{sde}(0) = 0$.

The sde of a matrix with its elements over the ring is defined as the smallest sde of all its constituent elements [1].

[2] Reported times are from a test run on a machine with a 2.80 GHz Intel i7, and 32 GB of RAM.

2.3.2 The channel representation

The channel representation is a very important tool used in the optimal T-count algorithm [1]. It also provides an extremely convenient representation of T gates, and as a consequence was used in the implementation discussed in Chapter 5.

Consider the conjugation of a Pauli operator by some matrix U, $UP_sU^\dagger$. We can expand the result by using the fact that the Pauli operators form an operator basis:

$$UP_sU^\dagger = \sum_{P_r \in \mathcal{P}_n} \widehat{U}_{rs} P_r, \tag{2.12}$$

where the $\widehat{U}_{rs}$ are the expansion coefficients. The channel representation $\widehat{U}$ of a unitary operation U is a superoperator with these coefficients as its matrix entries:

$$\widehat{U}_{rs} = \frac{1}{2^n}\mathrm{Tr}(P_r U P_s U^\dagger). \tag{2.13}$$

Defining $N = 2^n$, the channel representation is essentially an $N^2 \times N^2$ matrix with its rows and columns indexed by Paulis.

Channel representations have an additional useful property for our chosen gate set. When the entries of U are from $\mathbb{Z}\left[i, \frac{1}{\sqrt{2}}\right]$, so are the entries of $\widehat{U}$. However, since one may note that $\widehat{U}$ is Hermitian, the entries must be over the sub-ring

$$\mathbb{Z}\left[\frac{1}{\sqrt{2}}\right] = \frac{a + b\sqrt{2}}{\sqrt{2}^k}, \quad a, b \in \mathbb{Z}, \quad k \in \mathbb{N}. \tag{2.14}$$

Finally, channel representations respect matrix multiplication: $\widehat{UV} = \widehat{U}\widehat{V}$.

2.3.3 Coset labels

Cosets are partitions of a group, built with respect to a subgroup, which divide the original group into equal sized partitions. Any element of a coset can serve a representative for that coset. We define a coset label that is unique for each coset and can be computed easily from any of its constituent elements. The coset label of a circuit matrix is a means of checking its equivalency with other circuits. Computation of the coset label of a channel representation $\widehat{V}$ is done according to Algorithm 1. Reference [1] proves that two channel representations $\widehat{V}, \widehat{W}$ have the same coset label if and only if they are related by a Clifford: $\widehat{W} = \widehat{V}\widehat{C}$ (in other words, they are in the same coset).

Algorithm 1 Computation of the coset label

1: Given a channel representation $\widehat{V}$, rewrite $\widehat{V}$ so that each non-zero entry of the matrix has a common denominator of $\sqrt{2}^{\mathrm{sde}(\widehat{V})}$.
2: For each column of $\widehat{V}$, look at the first non-zero entry. If $a < 0$, or if $a = 0$ and $b < 0$, multiply all elements in the column by -1; otherwise, do nothing.
3: Permute the columns so that they are in lexicographical order[3].

[3]Sorting by lexicographical order can be accomplished using any deterministic sorting algorithm. For example, in the C++ implementation of Chapter 5, we use a combination of `std::lexicographical_compare` in conjunction with `std::sort` from the `<algorithm>` library.

2.3.4 Finding T-optimal circuit decompositions

We now review the basic framework of the algorithm and how it can be used to determine an optimal T-count decomposition. Recall the MITM decomposition of a circuit in Eq. (2.3). Since we expect the unitaries U_i to be part of the Clifford+T gate set, we can consider a similar decomposition which explicitly displays the presence of T gates within the circuit U:

$$U = e^{i\phi} C_{(k)} T_{(q_k)} C_{(k-1)} T_{(q_{k-1})} \cdots T_{(q_1)} C_{(0)} \tag{2.15}$$

where the subscripts on the $T_{(q_i)}$ indicate that they act on qubit q_i, and ϕ is a global phase, which we will soon show is irrelevant. This circuit has k T gates acting on a single qubit, thus it has T-count k.

We can rearrange this expression into a more convenient form by inserting extra Cliffords so that every T gate is conjugated by a Clifford, like so:

$$U = e^{i\phi} \prod_{i=k}^{1} \left(D_i T_{(q_i)} D_i^\dagger \right) D_0, \tag{2.16}$$

where the D_i are products of the original Cliffords C_i according to $D_i = \prod_{j=k}^{i} C_j$.

Now, let's expand a single-qubit T gate in the Pauli basis. Since there are no off-diagonal elements, we know that it must be combination of only I and Z:

$$T = aI + bZ. \tag{2.17}$$

Solving for coefficients a and b yields

$$a = \frac{1}{2}\left(1 + e^{\frac{i\pi}{4}}\right) \tag{2.18}$$

$$b = \frac{1}{2}\left(1 - e^{\frac{i\pi}{4}}\right). \tag{2.19}$$

We can extend this to multiple qubits: $T_{(q_i)} = aI_{2^n} + bZ_{(q_i)}$, where the identity matrix has dimensions $2^n \times 2^n$ and n is the number of qubits. With this, we can rewrite $D_i T_{(q_i)} D_i^\dagger$ as

$$D_i T_{(q_i)} D_i^\dagger = \frac{1}{2}(1 + e^{\frac{i\pi}{4}}) D_i I_{2^n} D_i^\dagger + \frac{1}{2}(1 - e^{\frac{i\pi}{4}}) D_i Z_{(q_i)} D_i^\dagger \tag{2.20}$$

$$= \frac{1}{2}(1 + e^{\frac{i\pi}{4}}) I_{2^n} + \frac{1}{2}(1 - e^{\frac{i\pi}{4}}) P, \tag{2.21}$$

where P is a Pauli since the D_i are Cliffords and $Z_{(q_i)}$ is a Pauli.

We use the shorthand notation [1]

$$R(P) = \frac{1}{2}(1 + e^{\frac{i\pi}{4}}) I_{2^n} + \frac{1}{2}(1 - e^{\frac{i\pi}{4}}) P \tag{2.22}$$

so that we can rewrite Eq. (2.16) as

$$U = e^{i\phi} \left(\prod_{i=k}^{1} R\left(P_i\right) \right) D_0. \tag{2.23}$$

The benefit of writing the equation in this form is that now, to synthesize an optimal T-count circuit, it suffices to find a set of Paulis $\{P_i\}$ and a Clifford D_0 which satisfy Eq. (2.23) up to a phase. Even

more convenient is that, if we consider the channel representation of all the above matrices, the global phase vanishes (as promised) [1]:

$$\widehat{U} = \left(\prod_{i=k}^{1} \widehat{R\left(P_i\right)} \right) \widehat{D_0}. \tag{2.24}$$

Eq. (2.24) is the structure we will use for the rest of this thesis, and also in the implementation presented in Chapters 5 and 6. With this expression, we will generate an optimal T-count circuit. The process is vaguely similar to the database generation and search of the MITM algorithm presented previously. Rather than circuit databases consisting of products in the universal gate set, we compute databases of coset labels. A *sorted coset database* $\mathcal{D}_k^n$ is defined in [1] as a list of n-qubit unitaries in channel representation form such that:

1. All elements in $\mathcal{D}_k^n$ have T-count k.
2. For any $\widehat{V}$ with T-count k, there is a unique coset label in $\mathcal{D}_k^n$.
3. The database is lexicographically sorted (to allow for easy binary search).

Coset databases are built as follows. $\mathcal{D}_0^n = \{\mathbb{1}\}$, the $N^2 \times N^2$ identity matrix. Subsequent levels of the database are built by left multiplying by a single $\widehat{R(P)}$ to each element in the previous level, and selectively adding only the unique new products. This has the effect, at each step, of adding a single T gate.

Suppose we want to check if the T-count of an n-qubit circuit $\widehat{U}$ is $\leq m$. The procedure for accomplishing this is shown in Algorithm 2 [1]. Such an algorithm not only determines the T-count, but also the sequence of Paulis used and the terminal Clifford D_0 (since we can invert Eq. (2.24) to recover its channel representation once we know the Paulis). Efficient algorithms exist for subsequently converting the $R(P_i)$ back to Clifford and T gates [1, 27].

Algorithm 2 Generation of optimal T-count circuits

1: Generate all sorted coset databases from $\mathcal{D}_0^n, \ldots \mathcal{D}^n_{\lceil \frac{m}{2} \rceil}$.

2: Sequentially execute a binary search to determine if there exists an element in one of the databases which is equal to the coset label of $\widehat{U}$. If such an element exists in database $\mathcal{D}_i^n$, then we have found the T-count is i and we stop.

3: If no such element is found in the previous step, then the T-count must be greater than $\lceil \frac{m}{2} \rceil$. Rather than generating further coset databases, we can perform a MITM-style search within them to find combinations with higher T-counts. Begin at $r = \lceil \frac{m}{2} \rceil + 1$. Starting with coset database $\mathcal{D}^n_{r-\lceil \frac{m}{2} \rceil}$, for each $\widehat{V}$, use binary search to check if there exists a $\widehat{W}$ such that the coset labels of $\widehat{W^\dagger U}$ and $\widehat{V}$ are equal. If a pair is found, then we know that the T-count is r. If no pair is found, increase r by 1 until we complete the case $r = m$, in which case the algorithm terminates as the T-count is larger than m.

This algorithm, like some others examined so far [12, 24], runs in exponential time. It also has the disadvantage of generating databases of exponential size, in both the number of qubits and the desired T-count. Let us briefly compute its storage space and runtime.

For the storage space, the largest database has size on the order of $N^{2\lceil \frac{m}{2} \rceil}$, where we recall $N = 2^n$. Each item in the database is an $N^2 \times N^2$ matrix - therefore an upper bound on the estimate for space

required by the database is $\mathcal{O}(N^m \text{poly}(m, N))$, where poly indicates a polylogarithmic term. In the implementation of [1], database generation was done in C++ and databases were stored as trees in binary files. The largest such file generated was on the order of 4 GB.

For the runtime, the first step is to generate all the databases - this takes time $\mathcal{O}(N^m \text{poly}(m, N))$. The next step is to execute the binary search on all the databases with the coset label of the unitary - this takes time $\mathcal{O}(\text{poly}(m, N))$ since binary search can be done in logarithmic time. Finally, to compare the remaining steps takes time $\mathcal{O}(N^m \text{poly}(m, N))$, since we need to search through the entire database at each iteration to find the MITM-style pair. Thus, the total runtime is equivalent to the storage time: $\mathcal{O}(N^m \text{poly}(m, N))$, a quantity exponential in both m and the number of qubits. This algorithm is thus impractical for circuits larger than about 3 qubits and T-count 7 (the largest case solved in [1]).

Chapter 3

Parallel collision finding

3.1 Overview

The parallel collision finding algorithm we adapt to circuit synthesis was developed by van Oorschot and Wiener in 1996 [28]. Their initial application was cryptographic in nature: how does one find collisions within a hash function, or an encryption function? Parallel collision finding offers an elegant, distributed, and scalable means of finding collisions in large spaces, many of which are exponential in size. For example, the space size of the double DES encryption scheme is 2^{112}, a number far too great to simply brute force with one processor, but manageable using parallel techniques [29].

The initial framework has a multitude of uses, from cryptography, to physics, and to mathematics. Any large search problem in which one is searching for a collision can benefit from these techniques, if the problem at hand can be framed in the formulation that follows. In this chapter, we provide an overview of the algorithm, the time-memory tradeoffs on which it depends, and the runtime estimate.

3.2 Algorithm details

3.2.1 Collisions and claws

In what follows, for the purpose of example let us consider f and g to be hash functions. Their spaces have size N_f and N_g, and they are many-to-one (otherwise there would be no collisions to find). Let their domains be $\mathbb{D}_f$ and $\mathbb{D}_g$, and ranges $\mathbb{R}_f$ and $\mathbb{R}_g$ respectively.

We begin by defining the objects for which we are searching.

Definition 2. *Let f be a hash function, and x, y be elements in $\mathbb{D}_f$. The elements x and y are said to be a* collision *if*

$$f(x) = f(y), \quad x \neq y. \tag{3.1}$$

Definition 3. *Let f and g be (different) hash functions, and $x \in \mathbb{D}_f$, $y \in \mathbb{D}_g$. Let $\mathbb{R}_f = \mathbb{R}_g$. The elements x and y are said to be a* claw *if*

$$f(x) = g(y). \tag{3.2}$$

In essence, a collision is a pair of elements in the domain of a function which map to the same element. A claw, on the other hand, is an instance of two elements which cause the two separate functions map to the same element.

3.2.2 Collision finding

Parallel collision finding is essentially a series of random walks through the hash space, which (hopefully) collide at some point. Suppose we have some hash function f; if f is a 'good' hash function, given an arbitrary input, the output should be roughly random. Repeatedly applying f to a given input, say x_0, produces a "trail" of random points in the space:

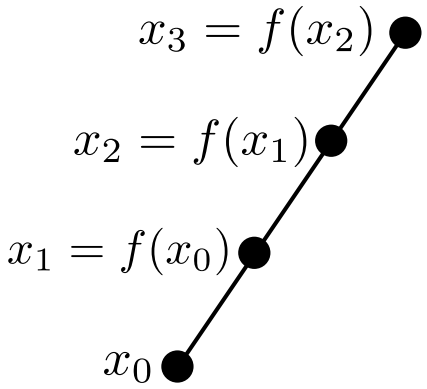


Figure 3.1: Basic structure of a trail; an initial value is hashed repeatedly, which is equivalent to a random walk through the hash space.

Now, we must define a stopping condition for these random walks. A certain fraction of the points in the space are denoted as *distinguished* points; if at any point in the walk, x_i is found to be distinguished, then the trail terminates. These terminal points are generally chosen as having some easily distinguishable property (for example, if we convert the x_i to binary strings, we might denote all strings starting with a certain number of 0 bits as distinguished). A complete trail will now look similar to Figure 3.2.

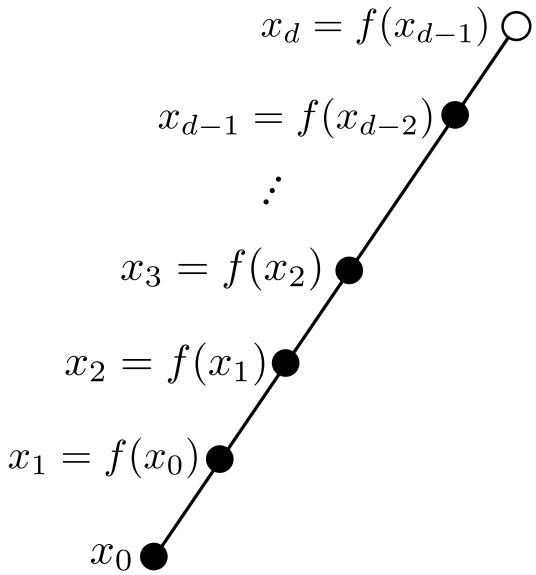


Figure 3.2: Trails stop when they hit a distinguished point (or reach a specified maximum iteration limit).

At the beginning of the algorithm, a number of processors set out, with randomly chosen starting points, to generate trails. A location common to all processors is used to store the collection of trails in which unique distinguished points are found. As a trail is completely deterministic we need only store

the starting point, ending point, and number of steps in between.

The fraction of points which are marked as distinguished is a *time-memory tradeoff*. Suppose we denote a fraction θ of the points as distinguished. Therefore on average, a trail will have to take $1/\theta$ steps before it finds a distinguished point, if the function is roughly random. The value of θ is chosen according to the computational resources available. More distinguished points means more storage space but less time required for each trail to terminate. Fewer distinguished points take up less space but each trail may have to run for a longer time. When searching through small spaces, there may be enough memory available to let all the points be distinguished; on the other extreme, some spaces may be so large that even storing a small fraction of the distinguished points is enough to consume all the allotted RAM. A means of choosing this parameter wisely is discussed in [28]. Unfortunately, it is possible for cycles to occur, as well as for trails to go a very long time without finding anything. Thus, an iteration limit is generally set (as some function of θ) in order to guarantee no processes are wasting unnecessary time on trails that may not end.

Now, suppose a processor finds a distinguished point, tries to store it, and sees that there is already a trail with the same distinguished end point in storage. There are two possibities:

1. The two trails started at the same place, and thus ended at the same place.

2. The two trails merged, i.e. there is a collision.

This second possibility is exactly what we're looking for, and is shown graphically in Figure 3.3. We can see that x_1 and y_3 are a collison of f, since

$$f(x_1) = f(y_3). \tag{3.3}$$

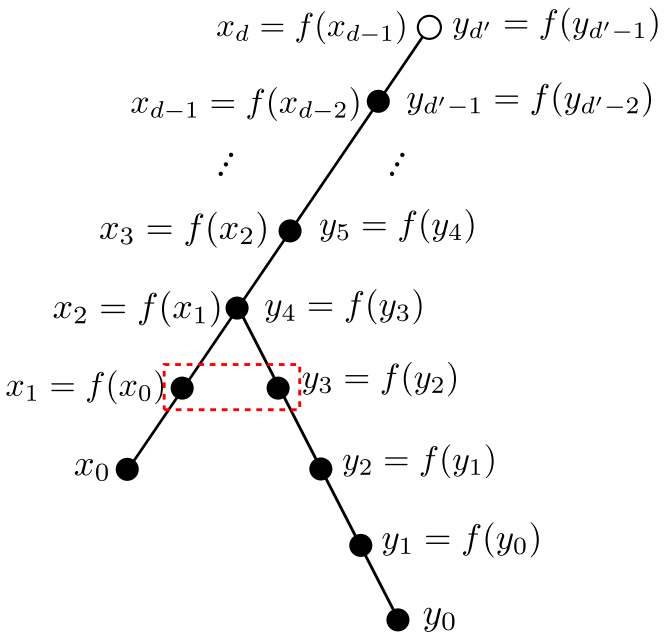


Figure 3.3: A depiction of the collision between two trails, $f(x_1) = f(y_3)$. The trails move forward together after the collision at point $x_2 = y_4$ and thus terminate at the same distinguished point $x_d = y_{d'}$.

What the processor does next is trace back through the sequence of both trails, and find the point where they collided. We start with the longer of the two trails, and apply f to the initial value a number of times equal to the difference in trail lengths (in the case of the example, we would apply f twice to

arrive at y_2). Then, the two trails apply f simultaneously (in the example, the short trail will compute x_1 and the longer y_3). The results are tested for equality at the end of each step - if we find two equal values, we've found a collision and the problem is solved (here, we'd eventually find $x_2 = y_4$).

The authors of [28] produced an estimate of a runtime for this algorithm. They suppose that, though there were many collisions, only one such collision is the solution to the problem at hand (i.e. the *golden* collision). There are also many different versions of f, all of which contain a golden collision (for example, if f was an encryption function and our goal was to find a collision in encrypted messages, perhaps we might denote different keys as leading to different versions of f). Let m be the number of processors, and let w be the number of distinguished points that the memory can hold. Simulations run in [28] with varying values of w and N_f showed that the best achievable runtime was when they choose $\theta = 2.25\sqrt{w/N_f}$ and generated $10w$ distinguished points per version of f before choosing a new one. The runtime was

$$T_{col} = \left(2.5\sqrt{\frac{N_f^3}{w}}\frac{1}{m}\right)\tau, \tag{3.4}$$

where τ is the amount of time required for a single iteration of f in a trail. They estimate that on average, $0.45N_f/w$ functions must be tried before finding a solution.

We note that in theory this algorithm depends inversely on the number of processors used; it is also inversely proportional to the square root of memory available to store distinguished points. This implies that more processors and more memory should make the algorithm run faster. However, in practice the time spent accessing the shared memory causes a bottleneck [28].

3.2.3 Claw finding

Finding claws rather than collisions is largely the same idea, however we must now consider two functions, f and g. Rather than randomly walking just over an element of the domain, we represent each step in the trail as a pair (x_i, b_i), where b_i is a flag which tells us whether to use function f or g in the next step. Trails begin by choosing x_0 and b_0 at random. Every subsequent step must first compute the result of the function f or g. This result is then used to deterministically derive a new b_i so that it 'knows' which function to perform in the next step. The result is also used to determine the next x_i such that is it in the proper domain corresponding to b_i. This deterministic step is termed a *reduction function.*

Execution of the algorithm is very similar to the collision case. An example of two trails merging to produce a claw is shown in Figure 3.4. Here, we find that $f(x_1) = g(y_3)$, and then the trails continued on to end at the same distinguished point.

An estimate for the runtime of claw finding is also provided in [28]. Let our functions f and g be hash functions on spaces of size N_f and N_g respectively. Let g be a function over a larger space $N_g = cN_f$ for some constant $c \geq 1, c \in \mathbb{Z}$. If N_g is larger than N_f, we divide N_g into c chunks of size N_f and execute searches over each set sequentially.

Note that the search is not executed just over a space of size N_f. Now the space has size $2N_f$, because at each step, we have an element of f or g and a flag which can take one of two values. We can apply the result of Eq. (3.4) directly to these parameters:

$$T_{claw} = \frac{N_g}{N_f}\left(2.5\sqrt{\frac{(2N_f)^3}{w}}\frac{1}{m}\right)\tau \tag{3.5}$$

$$= 2.5 * \sqrt{8}N_g\sqrt{\frac{N_f}{w}}\frac{1}{m}\tau \tag{3.6}$$

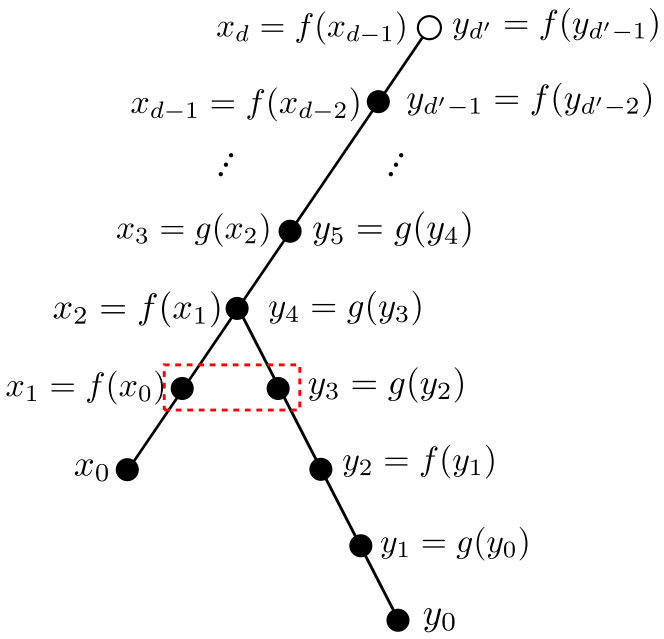


Figure 3.4: Finding a claw between two trails, $f(x_1) = g(y_3)$. Similar to the collision case, the trails move forward together after the claw occurs at $x_2 = y_4$. This is because the next step is completely and deterministically determined from the previous step using a reduction function.

$$\approx \quad 7N_g\sqrt{\frac{N_f}{w}}\frac{1}{m}\tau \tag{3.7}$$

where τ is now the runtime of a whole step in the trail, and where we added the prefactor N_g/N_f because the process must be repeated for each of the c subsets of g. Once again we see that the runtime depends inversely on the number of processors and inversely on the square root of available memory. This equation will be important for us later in Chapters 4 and 5 when we analyze the runtime of the parallelized circuit synthesis algorithm.

Chapter 4

Application of parallel framework to circuit synthesis

In this chapter we combine the ideas of the previous two chapters. We show how the parallel claw finding algorithm of Chapter 3 can be applied to the MITM circuit synthesis algorithm of Chapter 2 to produce a new framework for parallel quantum circuit synthesis.

4.1 Framework

We begin with an arbitrary quantum circuit U on n qubits, which can be implemented over a universal gate set $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$, where the G_i are unitary matrices. We can write U as

$$U_k \cdots U_1 = U, \tag{4.1}$$

where the U_i are depth 1 unitaries from $\mathcal{G}$ for $i = 1, \ldots k$, as in the MITM approach, and k is the depth of the circuit. Our goal is, given U, to recover the set of $\{U_i\}$ such that Eq. (4.1) holds.

We use the MITM approach to rewrite Eq. (4.1) by splitting it in half:

$$U_{\lceil \frac{k}{2} \rceil} \cdots U_1 = U^\dagger_{\lceil \frac{k}{2} \rceil + 1} \cdots U_k^\dagger U. \tag{4.2}$$

We propose the following new method for circuit synthesis, using the time-memory tradeoffs offered by parallel search techniques to synthesize our circuits. Let

$$V := U_{\lceil \frac{k}{2} \rceil} \cdots U_1, \tag{4.3}$$

$$W := U^\dagger_{\lceil \frac{k}{2} \rceil + 1} \cdots U_k^\dagger U \tag{4.4}$$

be the LHS and RHS of Eq. (4.2) respectively. We use the parallel claw finding method presented in Chapter 3 to execute a search over unitaries (circuit matrices) of type V and W. Our goal is to find a pair (V, W) that satisfy Eq. 4.2, i.e. they are a claw.

Let the function g be a function which produces a circuit matrix of type V, and function f one which produces a circuit matrix of type W. Parallel collision search over the possible circuits is executed as follows. First, we define a means of hashing a circuit matrix of type V or W to a simple data type, such as an integer or a string. Then, we require a means of converting the latter back to a matrix.

To perform claw finding, we begin with a number of processors. Each processor chooses a random starting point and flag and uses its data to generate a new matrix of type V or W. The resultant matrix

is then hashed down to a new value and flag. This process continues until a distinguished point is found, and the remainder of the algorithm proceeds exactly as the claw finding introduced in Chapter 3. The idea is that if two matrices V and W are the solution to the circuit synthesis problem, they will have identical matrix products and must hash to the same value - thus, they constitute a claw and can be found by the parallel search technique.

4.2 Runtime estimation and algorithm complexity

We can use Eq. (3.7) to compute an estimate for the runtime of this algorithm. If the size of the set of depth 1 gates is ζ, then the size of the spaces of V and W are

$$N_g = \zeta^{\lceil \frac{k}{2} \rceil} \tag{4.5}$$

$$N_f = \zeta^{\lfloor \frac{k}{2} \rfloor} \tag{4.6}$$

Then, the runtime can be approximated as

$$T_{frame} \approx \zeta^{\lceil \frac{k}{2} \rceil} \sqrt{\frac{\zeta^{\lfloor \frac{k}{2} \rfloor}}{w}} \frac{1}{m} \tau \tag{4.7}$$

$$= \zeta^{\lceil \frac{k}{2} \rceil + \frac{1}{2} \lfloor \frac{k}{2} \rfloor} \sqrt{\frac{1}{w}} \frac{1}{m} \tau \tag{4.8}$$

We can recover the dependence on the number of qubits by noting that $\zeta \leq \gamma^n$ where γ is the number of single-qubit gates in $\mathcal{G}$. Since the most time is spent performing matrix multiplication and we are working with unitaries of size $2^n \times 2^n$, $\tau \approx \lceil \frac{k}{2} \rceil 2^{3n}$. Thus, we obtain the final expression

$$T_{frame} \approx 2^{3n} \gamma^{n(\lceil \frac{k}{2} \rceil + \frac{1}{2} \lfloor \frac{k}{2} \rfloor)} \frac{1}{\sqrt{w}} \frac{1}{m} \left\lceil \frac{k}{2} \right\rceil . \tag{4.9}$$

Recall that m is the number of processors, and that w is the number of distinguished points that the memory can hold. Of course, this algorithm is still exponentially dependent on n and the circuit depth. However, it has the advantage of depending inversely on the number of processors and memory, just like the parallel collision finding framework. Thus, if we have enough processors and memory available for distinguished point storage, leveraging parallelization may help us synthesize larger circuits more efficiently. Large scale implementations with thousands of processors may even counteract a significant amount of the exponential dependence, provided we can efficiently deal with storage and communication between processors.

4.3 Applications and use cases

A very important point to consider is that the aforementioned algorithm can be applied to any MITM circuit synthesis problem. It suffices only to find a means of uniquely mapping combinations of gates to simple elements in a space which is easy to perform a random walk over (and designing suitable hash/reduction functions).

We chose, for simplicity of initial implementation, to apply it to synthesizing circuits with optimal T-count. This is the work of Chapters 5 and 6. Other possibilities include but are not limited to:

- the Clifford+T framework of the original MITM framework,

- synthesis algorithms using different universal gate sets,
- approximate synthesis algorithms.

Similar to the parallel collision finding techniques it is built upon, this framework is extremely versatile and has the potential to improve many of the existing synthesis algorithms.

Chapter 5

Implementation details and results of threaded algorithm

5.1 MITM for optimal T-count circuit synthesis

We implemented the optimal T-count synthesis algorithm within the parallel framework. Our rationale is explained in Section 5.2.2. For now, we provide the structure of the problem.

We begin with a circuit U on n qubits. As per Eq. (2.24), we can rewrite the decomposition of U using the channel representation as

$$\widehat{R(Q_t)}\widehat{R(Q_{t-1})}\cdots\widehat{R(Q_1)}\widehat{C} = \widehat{U}, \tag{5.1}$$

where $\widehat{C}$ is some Clifford, and t is the T-count of U. Using the MITM approach of [24], we split Eq. (5.1) in half:

$$\widehat{R(Q_{\lceil \frac{t}{2} \rceil})}\cdots\widehat{R(Q_1)}\widehat{C} = \widehat{R(Q_{\lceil \frac{t}{2} \rceil+1})}^{\dagger}\cdots\widehat{R(Q_t)}^{\dagger}\widehat{U} \tag{5.2}$$

$$\widehat{V}\widehat{C} = \widehat{W}, \tag{5.3}$$

where we have relabeled the products of $\widehat{R(Q_i)}$ as $\widehat{V}$ and $\widehat{W}$ for convenience. It is known that if the coset labels of $\widehat{V}$ and $\widehat{W}$ are the same, then there exists a Clifford C such that the above holds [1].

We can apply parallel collision finding to this situation by generating sequences of Paulis to produce candidates $\widehat{V}$ and $\widehat{W}$, and use their coset labels to compute the next point and flag in the trail. This way, if a collision event occurs, it will be because we found two coset labels which were equal. If such a situation was the result of a claw, then we have found both $\widehat{V}$ and $\widehat{W}$, and as a consequence can recover $\widehat{C} = \widehat{V}^{\dagger}\widehat{W}$.

We can estimate the runtime of this algorithm as follows. Let g be the function representing the generation of the LHS $\widehat{V}$ of Eq. (5.2). Let f represent the generation of the RHS $\widehat{W}$. The sizes of the spaces are

$$N_g = 4^{n\lceil \frac{t}{2} \rceil}, \tag{5.4}$$

$$N_f = 4^{n\lfloor \frac{t}{2} \rfloor}. \tag{5.5}$$

We can apply Eq. (3.7) directly. Ignoring the prefactor of 7, we find that the runtime is of the order

$$T_{tcount} \approx 4^{n\lceil \frac{t}{2} \rceil}\sqrt{\frac{4^{n\lfloor \frac{t}{2} \rfloor}}{w}}\frac{1}{m}\tau \tag{5.6}$$

$$= \quad 2^{n\left(2\left\lceil \frac{t}{2}\right\rceil +\left\lfloor \frac{t}{2}\right\rfloor \right)}\frac{1}{\sqrt{w}}\frac{1}{m}\tau. \tag{5.7}$$

Here τ is the time it takes to go through a single iteration of a trail. The most expensive operations in a trail are the multiplication of $\left\lceil \frac{t}{2}\right\rceil$ channel representations. As these matrices are of size $4^n \times 4^n$, we can overestimate this runtime as

$$\tau \approx \left\lceil \frac{t}{2}\right\rceil \left(4^n\right)^3. \tag{5.8}$$

In reality, some shortcuts are made when left-multiplying by a Pauli channel representation (see Section 5.2.2 for details). However, the multiplication by the circuit matrix $\widehat{U}$ still constitutes a full matrix multiplication so we stick with the above τ as our function estimate.

Putting this all together, we obtain

$$T_{tcount} \approx 2^{n\left(6+2\left\lceil \frac{t}{2}\right\rceil +\left\lfloor \frac{t}{2}\right\rfloor \right)}\frac{1}{\sqrt{w}}\frac{1}{m}\left\lceil \frac{t}{2}\right\rceil. \tag{5.9}$$

In cases where the T-count is even, this collapses to

$$T_{tcount} \approx 2^{n\left(6+3\frac{t}{2}\right)}\frac{1}{\sqrt{w}}\frac{1}{m}\frac{t}{2}. \tag{5.10}$$

5.2 Implementation

5.2.1 Language and computer specifications

We implemented this algorithm on Linux using C++11. The code and documentation is located at `https://github.com/glassnotes/Circuit-Synthesis`. Development was done on SHARCNETs' Orca cluster [1]. This first implementation uses OpenMP [17] parallelization meaning that the number of threads in tandem was limited by the number of cores in each node of the cluster

The code was compiled with `gcc` version 4.8.1 and -O3 compiler optimizations. We used version 3.1 of OpenMP. It also makes use of Boost 1.57.0 header libraries [2] [30]. The code was tested on AMD Opteron processors which have 4 cores and processing speeds of 2.2GHz. On Orca, Opteron processors are grouped into sockets containing three processors, totalling 12 cores each. Each full node contains two sockets, leaving us with a total of 24 cores available per node. All processors in a node have access to a shared memory of 32 GB (in which we can store a common set of distinguished points). Orca also contains a subcluster of slightly faster Intel Xeon processors, but these only have 16 cores per node. There are also two subsets of Xeon nodes with different processing speeds, and it was not possible to specify which one to use so the processing speed would not be a constant across all trials. We thus chose the Opteron nodes for consistency, and ease of access due to the fact that there are many more Opteron nodes than there are Xeon nodes.

5.2.2 Special techniques to make the program faster

As previously mentioned, there were a number of reasons why we chose to implement optimal T-count synthesis. Designing a function which can randomly select elements of the Clifford group is not a trivial task [31]. Consequently, performing a random walk over the Clifford+T set is also not trivial. Performing a random walk over the integers or binary strings, however, is quite simple, and can be done using any

[1] Documentation for Orca can be found on SHARCNETs webpage at `https://www.sharcnet.ca/my/systems/show/73`.

[2] We used Boost as there was no implementation of dynamic bitsets in standard C++11. Boost.MPI was also used in the implementation of Chapter 6

generic hash function. Thus, the key reason we chose initially to implement the optimal T-count algorthm was because of the interconnection between products of Pauli matrices and binary strings. The output of the T-count algorithm is a sequence of Paulis $\{Q_i\}$ and C such that Eq. (5.1) is satisfied. Each Pauli, or combination thereof, has a representation as a binary string using binary symplectic representation (see Appendix A for details). This means we have a 1-to-1 map from a sequence of $\{Q_i\}$ to the integers. One then simply needs to execute a parallel search over the integers. This also greatly simplifies the storage of distinguished points, since we only need to store a single integer, rather than an array of Paulis, or worse, a matrix.

Another reason was due to the structure of the channel representations: such matrices have a very regular, sparse structure. Any given row in a channel representation $\widehat{R(Q_i)}$ contains either a single 1 on the diagonal, or two entries which are either $\pm 1/\sqrt{2}$. To save both time and space, I implemented an algorithm akin to sparse matrix multiplication. Instead of storing the entire matrix, I stored a single vector with all the non-zero elements of the matrix in sequence, as well as a vector of vectors containing the positions of the elements in each row (which contain only 1 or 2 elements each). This way, we need only multiply and sum specific elements rather than iterating over entire rows and columns which had mostly empty entries. We note that in theory, this matrix multiplication can also be parallelized. However, for this implementation we are limited by hardware - all the cores in the node are being used to search for distinguished points. Parallelizing the matrix multiplication would add another layer of parallelization and require the cores to hyperthread, which would likely result in poor performance.

A final advantage is that the size of the space can be reduced by noting that not every sequence of t Paulis has exactly T-count t. If any of the following occur, the T-count may be reduced:

1. Some Q_i is the identity Pauli I_{2^n}.
2. Two adjacent Paulis Q_i and Q_{i+1} are identical.
3. Two identical Paulis are separated by Paulis with which they commute.

The reasons are as follows:

1. It is easy to see that $R(I_{2^n}) = I_{2^n}$ would 'disappear', meaning that there will be one less term in the sequence. As each $R(Q_i)$ effectively represents a single T gate, the T-count would be reduced to $t - 1$.
2. One can check that, if two Paulis are identical, then $R(Q)R(Q)$ produces a Clifford gate [1], which does not contribute to the T-count.
3. Suppose the algorithm produces a sequence like

$$R(Q_t) \cdots R(P)R(Q_5)R(Q_4)R(P)R(Q_2) \cdots \tag{5.11}$$

and that P commutes with both Q_4 and Q_5. Now, if P and Q_5 commute, then so do $R(P)$ and $R(Q_5)$. We can then interchange them and obtain the same effective matrix product. We can do the same for Q_4 as well - however, this would mean that this product is equivalent to a product where two adjacent Paulis are now identical - thus, the original product did not have T-count t.

In the implementation, Pauli sequences are constructed one n-qubit Pauli at a time from the output of a 160-bit SHA-1 hash value. For each additional Pauli we add, the algorithm checks that none of the above three criteria are satisfied. If the candidate Pauli satisfies any of them, then it is discarded and the next Pauli is chosen instead. If we run out of bits to use from the original SHA-1 hash, the hash is run

through SHA-1 again to generate more. This checking process does add some extra work, but it has the benefit of reducing the size of the search space and eliminating much of the redundancy that would occur if we allowed identical Paulis or identities. We note that this set of conditions may not be exhaustive, but that these restrictions alone result in a significant decrease in run-time [3].

5.2.3 Program flow

Let us look back to Chapter 1.3, where different parallel techniques were discussed. The simplest technique to implement is parallelization via threading. Hence, as a first implementation and for proof of principle, we implemented the program using the OpenMP threading library on a single node. The general structure of this version of the algorithm is shown in Figure 5.1. Necessary subroutines are presented in Figure 5.2. The algorithm itself is presented in Algorithm 3.

The program takes as input a circuit (which it converts to a matrix), the desired T-count, a number of threads, and a string to indicate what fraction of the points in the space are designated as distinguished (this is usually a string of 0s, such as "000" to indicate that integers are distinguished if the first part of their binary representation is equal to this string).

All spawned threads can access shared memory. A C++11 STL `unordered_map` is used to store the set of found distinguished points. Distinguished points are stored as objects of a `struct` which contains the first integer and flag (an integer or character which indicates which side of Eq. 5.2 to perform), the last (distinguished) integer and flag, and the number of steps in between. A boolean variable common to all threads is used to keep track of whether or not a claw has been found.

Circuits are read in in text format and then converted to a matrix over elements of the ring $\mathbb{Z}\left[i, \frac{1}{\sqrt{2}}\right]$ (both the ring elements and the matrix are classes). The channel representation of the initial matrix is then computed (channel representations are also a class, separate from the original matrix class).

Once all initialization has taken place, OpenMP threads are spawned and the execution proceeds in parallel. Each thread chooses a random initial integer and flag and proceeds to make a trail - these trails end either at a distinguished point, or stop after some predetermined number of iterations (to avoid trails self-looping endlessly).

Trails are produced as follows. For an n-qubit circuit of T-count k, a random walk is executed over the space of the integers from 0 to $4^{n\left\lfloor \frac{k}{2} \right\rfloor} - 1$. The binary representation of each integer forms a unique sequence of Paulis. If the T-count is odd, we divide the larger space into subsets of this size and execute the search sequentially over them.

The initial flag tells us whether to generate the LHS or RHS of Eq. (5.2). The channel representations of these matrices are computed, as is their product. We then compute the coset label of this product, convert it to a unique bit string, and hash it using SHA-1. This is because it is highly unlikely that two different coset labels will have the same SHA-1 hash value - thus if two separate trails produce the same hash at some point, it is highly likely that they had the same coset label, and may have collided (or produced a claw).

We then use a reduction function to convert the hash into a new integer and flag, for the next iteration of the trail. A reduction function is analogous to the opposite of a hash function - it deterministically takes a hash value to a (different) element in the domain of the hash function. The reduction function used by the program is very simple. It takes a SHA-1 value as a hexadecimal string, and sums a sequence of bit shifts performed on the decimal representation of the characters in the string. This produces an integer, the modulus of which is taken such that the integer is within the size of the space we are

[3] In some of the very first versions of the code, synthesis of the Toffoli gate took regularly around 45 minutes. After eliminating the redundancy, and adding in sparse matrix multiplication, it now takes less than 5 minutes on average (see Section 5.3).

Algorithm 3 OpenMP algorithm

Input: `circuit`, T-`count`, `num_threads`, `distinguished_string`
`found_distinguished_points` $\leftarrow \{\}$
`claw_found` $\leftarrow$ **false**
`num_steps` $\leftarrow 0$
`function_id` $\leftarrow 0$
`subset_id` $\leftarrow 0$
#pragma omp parallel (`num_threads`)
while `claw_found` is **false do**
    `start_int, start_flag`$\leftarrow$ a random initial integer and flag
    `trail_end` $\leftarrow$ result of trail starting from `start_int, start_flag`
    if `trail_end` contains a distinguished point **then**
        `matching_point` $\leftarrow$ `NULL`
        #pragma omp critical {
        search for it in `found_distinguished_points`
        if no match exists in `found_distinguished_points` **then**
            add the point to `found_distinguished_points`
        else
            `matching_point` $\leftarrow$ match from `found_distinguished_points`
        end if
        }
        if `matching_point` is not `NULL` **then**
            check for collision/claw
            if collision event was a claw **then**
                `claw_found` $\leftarrow$ **true**
            else
                discard `trail_end` and restart loop
            end if
        else
            discard `trail_end` and restart loop
        end if
    end if
    (atomic) `num_steps`++
    if `num_steps` reached max iteration limit **then**
        if T-`count` $\%\ 2 == 0$ **then**
            (atomic) `function_id` ++
            (atomic) `num_steps` $\leftarrow 0$
        else
            `subset_id` ++
            if `subset_id` $>$ maximum subset identifier **then**
                (atomic) `subset_id` $\leftarrow 0$
                (atomic) `function_id`++
            end if
        end if
    end if
end while
Output: collection of Q_i and $\widehat{C}$ which satisfy Eq. (5.2), if found.

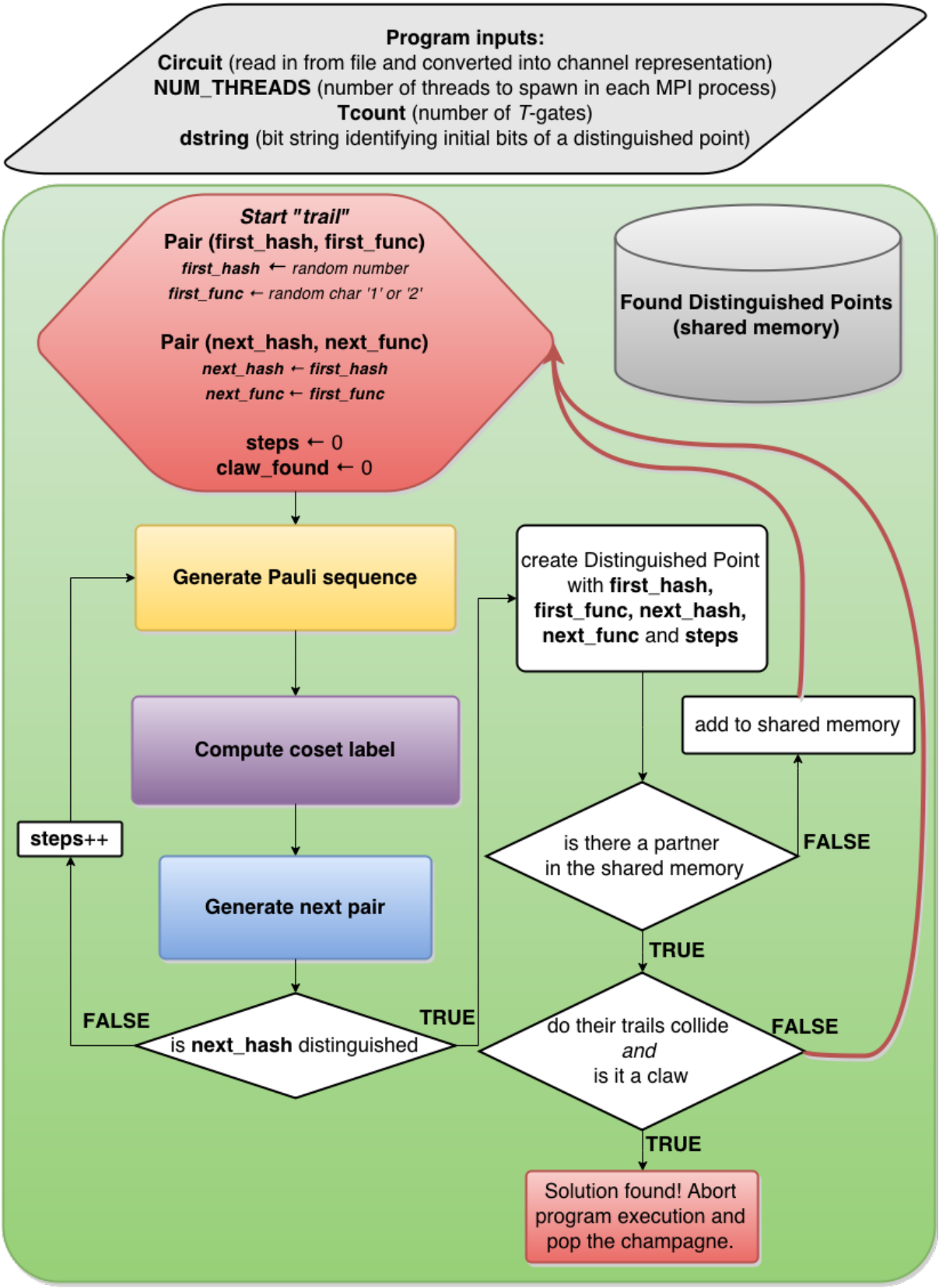
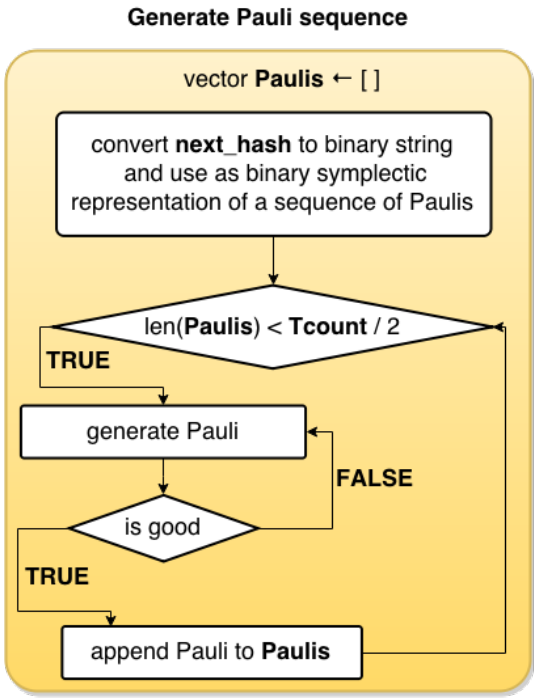
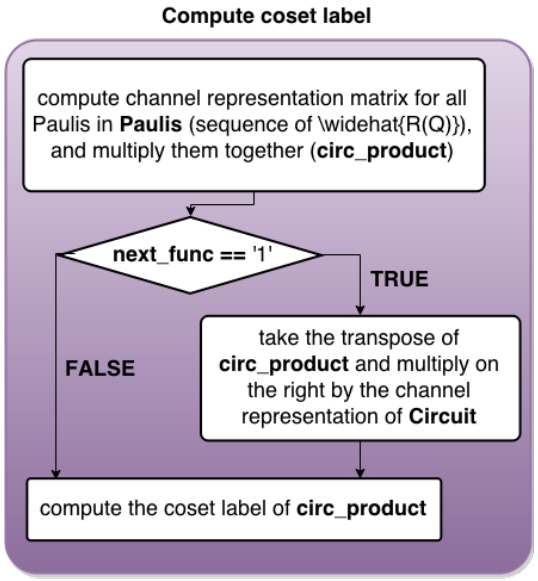


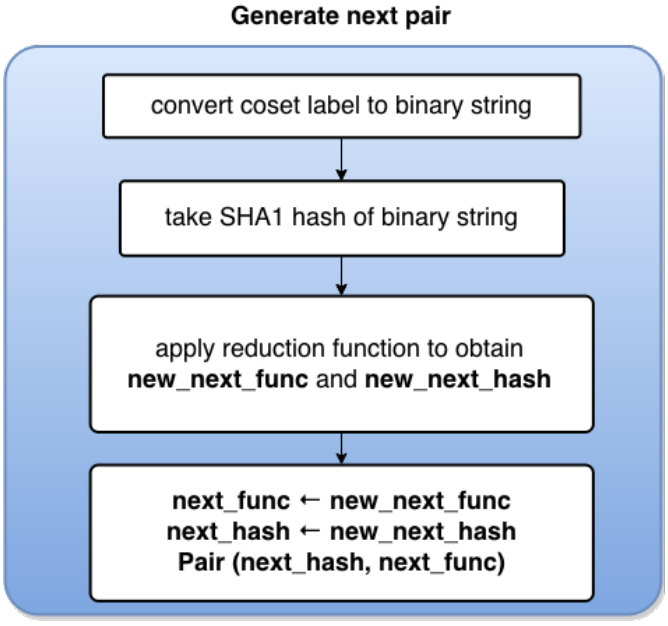
Figure 5.1: Flowchart of the program execution for the OpenMP optimal T-count implementation. See Figure 5.2 for expansions of the yellow, purple, and blue subroutines.



(a) Subroutine used to generate a sequence of Pauli matrices from a binary string.



(b) Subroutine used to compute the coset label of a given Pauli sequence.



(c) Subroutine used to hash a coset label and generate the next pair in a trail.

Figure 5.2: Subroutines used in Figure 5.1 and Figure 6.1.

searching. We check if the new integer is a distinguished point by comparing the first few digits of its binary representation to the input string indicating the distinguishing condition. If they match we terminate the trail and return the distinguished point. If not, we continue on with the next iteration of the trail.

Valid distinguished points must be added to the shared memory, which is a costly operation. For one, this occurs in a 'critical' code block, meaning that only a single thread may read and modify the shared memory at a time. Each thread must search through the current set to see if a distinguished point with the same value already exists in it. If not, it must be added. If it is already present, then the distinguished point information from the set is copied and the thread exits the critical section.

If a thread finds a match, it proceeds to check if the two trails merge due to a claw. Checking this involves tracing through the execution of both trails from their beginnings to a point where the coset label is equal, a technique which was introduced in Chapter 3. If a claw is not found, then the thread discards its current trail, and begins a new one. If one is found, then the communal boolean variable is set to `true` so that each thread knows to terminate before starting its next iteration.

We keep track of the number of trails of for two reasons. First, when the T-count is odd, the larger space is partitioned into chunks the size of the smaller space. After a specified number of trails are generated, we switch to a new subset by dumping the stored distinguished points and resetting the trail counter.

Second, we try a different reduction function after a specified number of trails are generated (usually 10 times the number of possible distinguished points in the space [28]). To vary the reduction function, we include an ID value which gets incorporated into the bit shift sequence before the modulus is taken. Different versions are required to "scramble" the mappings around. Sometimes the mappings are bad, either many self-loops were created, or some elements mapped to themselves and trails would go nowhere.

5.3 Results

5.3.1 Timing random parallel code

I would like to take a moment to make an important point, before presenting the results of the synthesis runs. This algorithm is random. This algorithm also runs in parallel. Even for the same circuit, every time the algorithm runs, the output sequence of operations may be different if there are Paulis within it that commute (and thus multiple solutions to the problem). Even two runs with the same initial seed may take different amounts of time if there are slight differences in how the threads run and interact with the central distinguished point storage.

Benchmarking the algorithm is difficult. In the results which follow below, we present the timed results of 100 separate trials for each set of parameters. Timing begins immediately before the OpenMP threads are spawned, and ends once the claw has been found. We present the average values for each circuit, as well as the standard deviation, and use histograms to plot the distributions. We also offer comparison with runtimes from some previous algorithms. These runtimes are not directly comparable as this algorithm is ultimately different from its predecessors. The times are meant to offer a rough comparison, and be a goal to strive for and improve on.

5.3.2 2-qubit synthesis

5.3.2.a Controlled-Hadamard

The first 2-qubit circuit we synthesized was the controlled-Hadamard gate, shown in Figure 5.3. This circuit is small, having only T-count 2. The previously reported synthesis time in [24] was 0.5s. The

machine they used was very different, and used a single quad-core Intel i5 processor with speed 2.80GHz, and 16GB RAM (we note the difference in clock speed with our AMD Opteron nodes at 2.2GHz).

To test the claw finder, we ran 100 trials on one of the Opteron nodes using the same parameters (8 threads, 1/4 of points distinguished). The average time taken was 0.3783s, with a standard deviation of 0.7643s. Memory usage of the program was negligible and not reported by the SHARCNET job scheduler for most jobs - it was always less than 1 GB when reported. The Paulis Q_1 and Q_2 we obtained were

$$Q_1 = I \otimes Y, \quad Q_2 = Z \otimes Y. \tag{5.12}$$

We note that these Paulis commute, so many of the trials found the Paulis in the other order.

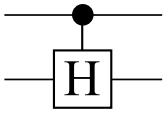


Figure 5.3: Circuit diagram of the controlled-Hadamard gate. This operation has T-count 2.

The distribution of runtimes is shown in Figure 5.4. We note that most of the runtimes fall below the 0.25s mark. Recall that the algorithm requires multiple versions of the reduction function - outliers 1 second and above occur when the algorithm does not find a claw within the designated number of iterations on the first reduction function. A second (or third) reduction function must sometimes be tried, a process which takes some time as the central storage location must be reset before threads can add new distinguished points to it.

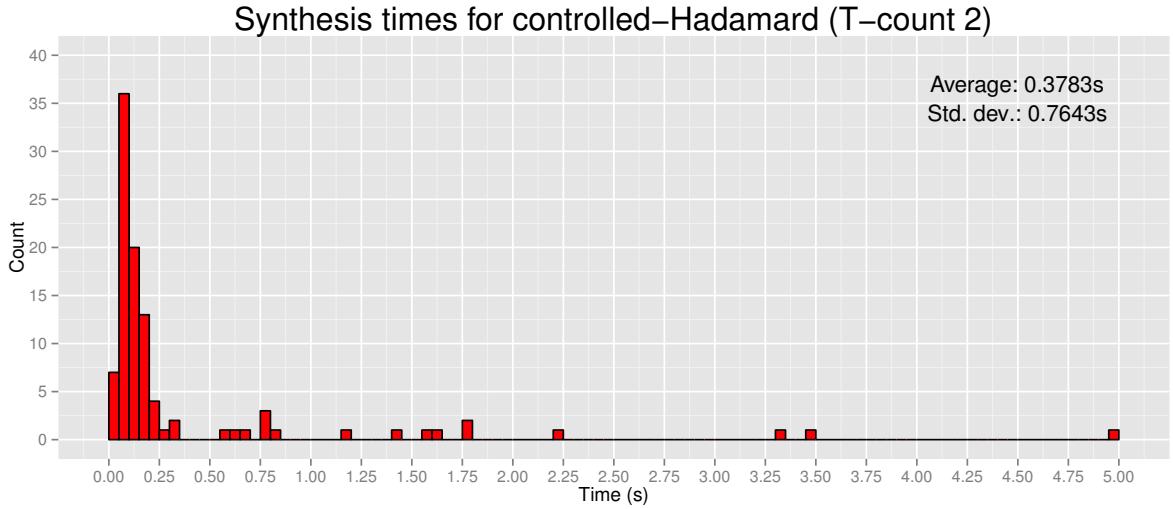


Figure 5.4: Distribution of runtimes for synthesizing the controlled-Hadamard gate with 8 threads and 1/4 of the search space marked as distinguished points. The outliers at the high end of the runtime occur when a claw is not found in the first reduction function tried.

In the case of the longest-running trial, 4.977s, the claw was found 4 times, and 2 different reduction functions were tried. Finding multiple instances of the claw takes more time because each processor that finds a claw must follow through to the end of its execution by tracing through the two trails, and output its results.

5.3.2.b Controlled-phase

A similar experiment was done with the controlled-phase gate (C-S), shown graphically in Figure 5.5. The C-S gate has T-count 3.

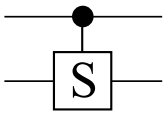


Figure 5.5: Circuit diagram of the controlled-phase gate. This operation has T-count 3.

We use Eq. (5.2) and divide it as follows:

$$\widehat{R(Q_2)}\widehat{R(Q_1)}\widehat{C} = \widehat{R(Q_3)}^{\dagger}\widehat{U} \tag{5.13}$$

The size of the space being searched on the RHS of this equation is $4^2 = 16$, as generation of the RHS involves the creation of only a single Pauli (and then multiplication by the target operation U). The LHS on the other hand requires generation of two Paulis - the space is therefore of size $4^{2\cdot 2} = 256$, 16 times larger than the RHS. We note that in reality, the techniques from Section 5.2.2 will decrease these sizes, but they are still necessary here to divide the larger space into subsets. To perform the claw finding algorithm, we partition the larger space into 16 chunks; each of these subsets is searched sequentially. As a result, program execution for C-S and any odd-depth circuit takes slightly longer as some of the subsets may have no solutions and the algorithm must generate a designated number of trails before continuing.

The Paulis which were found to constitute the C-S gate are

$$Q_1 = Z \otimes Z, \quad Q_2 = Z \otimes I, \quad Q_3 = I \otimes Z. \tag{5.14}$$

The distribution of runtimes is shown in Figure 5.6. The average runtime was 0.9386s, with a standard-deviation of 1.2902s.

5.3.2.c Varying the T-count

To test the limits of the 2-qubit version of the algorithm, we designed a series of circuits from T-count 2 up to T-count 15. They consisted solely of Hadamard and T gates - a series of examples is given in Figure 5.7. For each T-value, we ran 100 trials, taking 1/8 of the points designated as distinguished, using 16 threads of the Opteron nodes. The results are depicted in Figure 5.8. In this plot, we see precisely the exponential increase in runtime predicted by the algorithm. For 2 qubits runtime begins to skyrocket at around T-count 10.

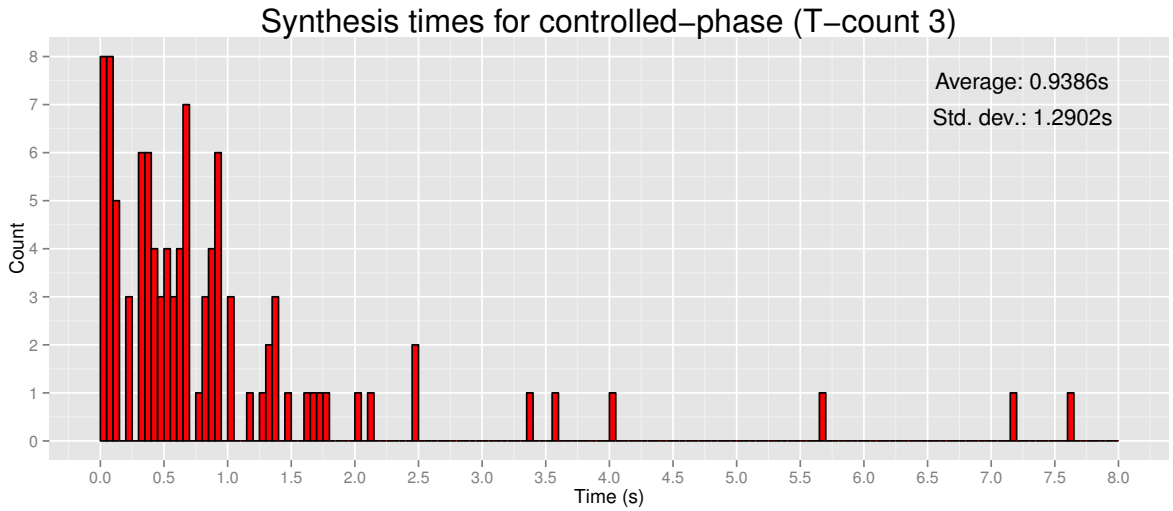


Figure 5.6: Distribution of runtimes for synthesizing the controlled-phase gate with 8 threads and 1/4 of the search space marked as distinguished points. Synthesis takes longer for odd T-counts because each subset of the divided space must be searched sequentially for a fixed period of time.

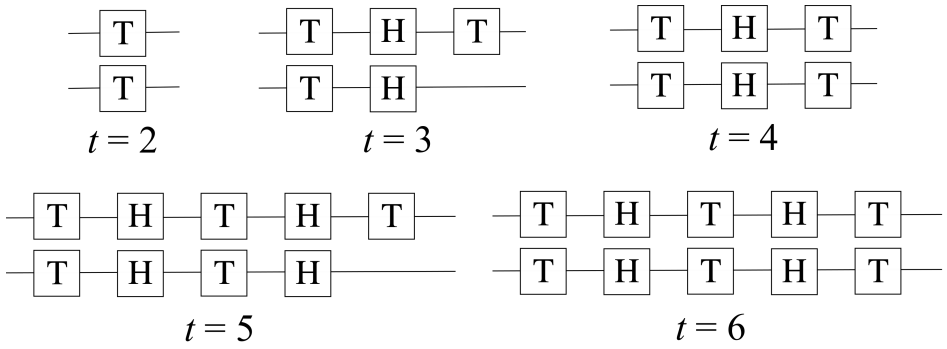


Figure 5.7: Examples of 2-qubit circuits synthesized with varying T-count (denoted by t). These circuits consist only of H and T gates. The T gates are added one at a time, with layers of T gates separated by Hadamards on each qubit.

5.3.3 3-qubit synthesis

5.3.3.a T-count 7

Some of the largest circuits that were previously synthesizable with the T-optimal algorithm are a small number of 3-qubit circuits with T-count 7. We tested five such circuits: the Toffoli gate, the negated Toffoli gate, the Fredkin gate (controlled swap), the Peres gate, and the Quantum OR gate. Gates are depicted in Figure 5.9.

Synthesis using the original MITM algorithm took about 415s for the Toffoli gate [24]. As the algorithms perform different tasks (the original algorithm outputs a sequence of Cliffords rather than Paulis, and does not consider optimal T-count), we use this timing only as a rough means of gauging the success of the algorithm.

All trials were run with the same set of parameters on the same type of nodes on SHARCNETs' Orca. We used 16 threads rather than the full 24 due to the increased wait time of getting a full nodes worth of

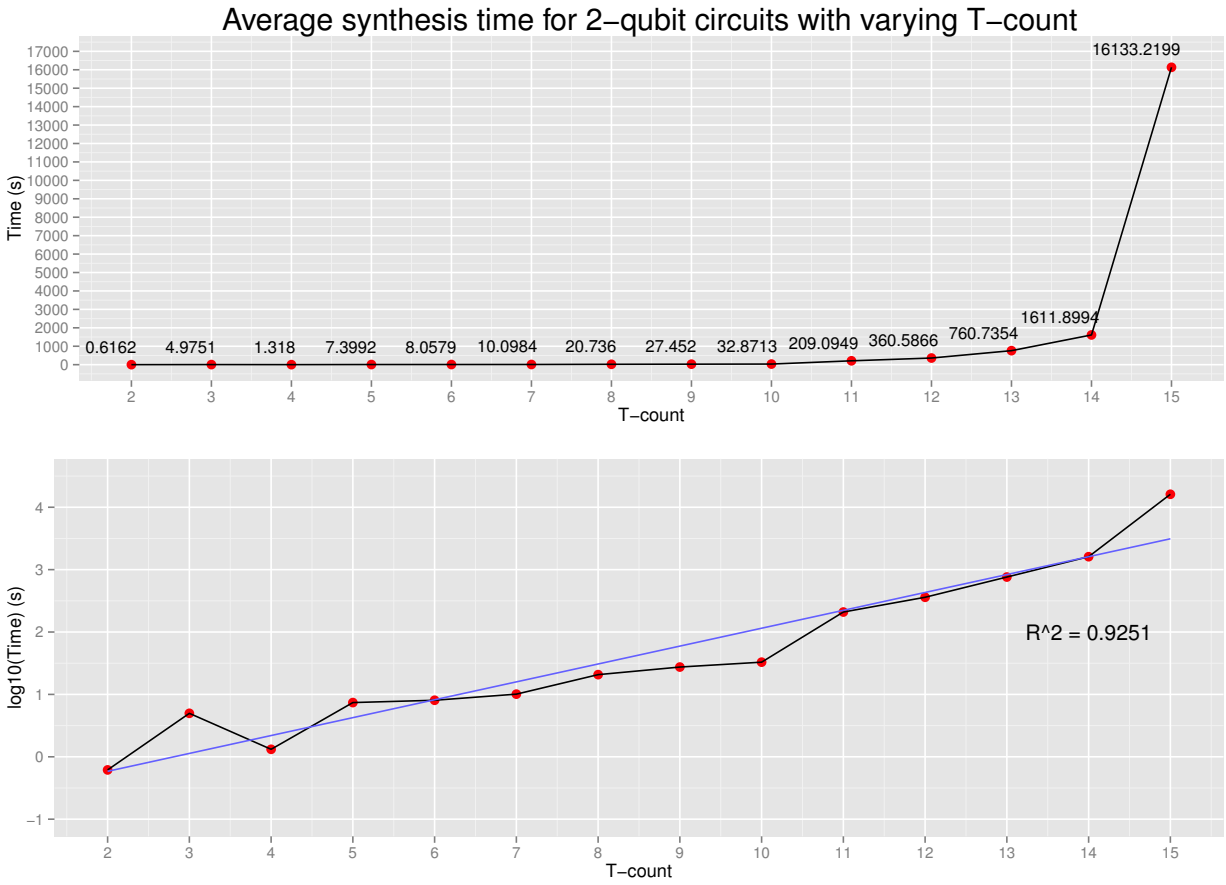
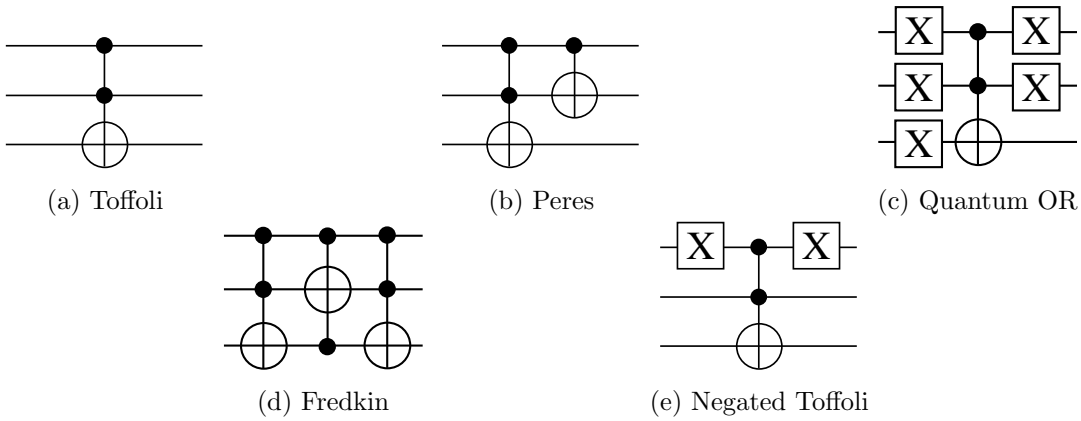


Figure 5.8: Average synthesis time with varying T-count for 2-qubit circuits. The upper plot displays the exponential increase in time taken as the T-count increases. This is clearer in the lower plot which shows the log transform of the data, producing an expected roughly linear trend with T-count. The equation of the line is $0.2867t - 0.8061$, where t is the T-count. As an indicator of how well it fits the data, we can compute its coefficient of determination (R^2 value). We find that $R^2 = 0.9251$, which is reasonably close to 1, indicating a satisfactory goodness of fit.

cores on Orca. All runs were done with 1/8 of the points designated as distinguished. RAM usage was about 1.3GB per run. Average times for each circuit are shown in Table 5.1.

Time-distribution histograms are shown in Figures 5.10-5.15. The results from these trials are intriguing - the distribution of times is much wider than that of the 2-qubit circuits, though in most cases there are distinguishable clusters of times towards the lower end of the scale. An unusual case was the Fredkin gate - there were 7 cases which took over 45 minutes to complete. Recall that for odd T-counts, the larger space is partitioned into chunks. For the Fredkin, there were possible solutions to the problem in the first chunk searched, as well as the sixth. The Pauli sequences found contained many commuting operators, which led to multiple solutions. A solution in the first chunk was found by 93 of the trials. The remaining 7 were unfortunate and did not find the solution initially, and ran until they found it in the sixth chunk, thus taking a significantly greater amount of time as they had to search sequentially through subsets of the space where there was no solution. As a result we show two histograms for the Fredkin gate: Figure 5.14 which contains all the data, and Figure 5.15 which contains the 93 points which found the solution in a reasonable amount of time.



(a) Toffoli (b) Peres (c) Quantum OR

(d) Fredkin (e) Negated Toffoli

Figure 5.9: Circuit diagrams for the five 3-qubit circuits with T-count 7 which we synthesized.

Circuit	**Average time (s)**	**Standard deviation (s)**
Toffoli	267.3684	174.6607
Negated Toffoli	280.2070	188.9637
Peres	270.3597	183.1228
Fredkin	387.9245	735.8697
Fredkin (without outliers)	191.2257	148.6447
Quantum OR	241.1220	165.3518

Table 5.1: Average runtimes for the five 3-qubit circuits. All trials were run using 16 threads on an Orca AMD Opteron node, and 1/8 of the points were designated as distinguished. The case of the Fredkin gate saw some extreme outlier times (over 3000s), hence we present the results both with and without these outliers.

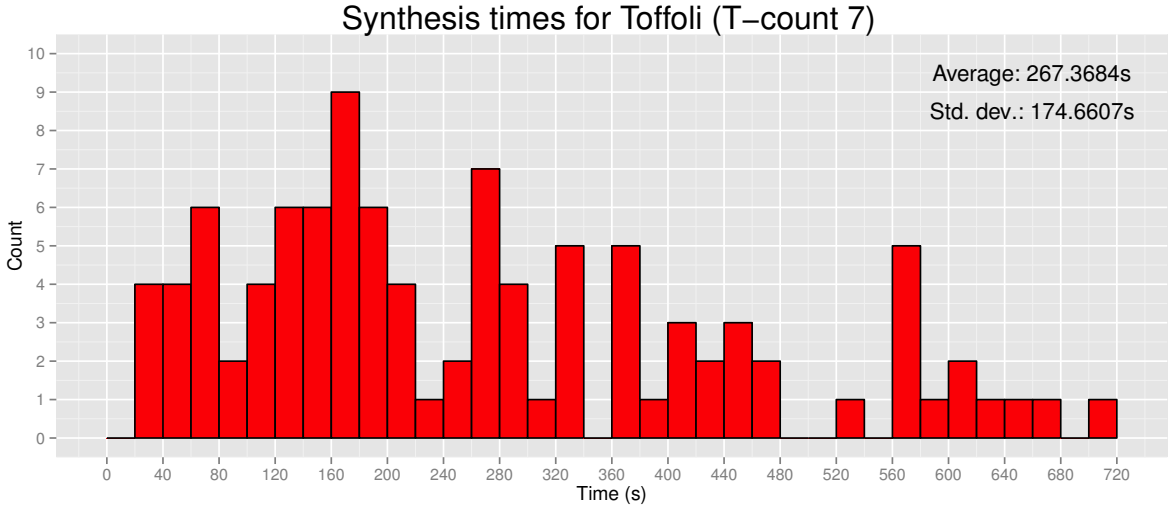


Figure 5.10: Distribution of synthesis times for the Toffoli (T-count 7, 16 threads, 1/8 points distinguished).

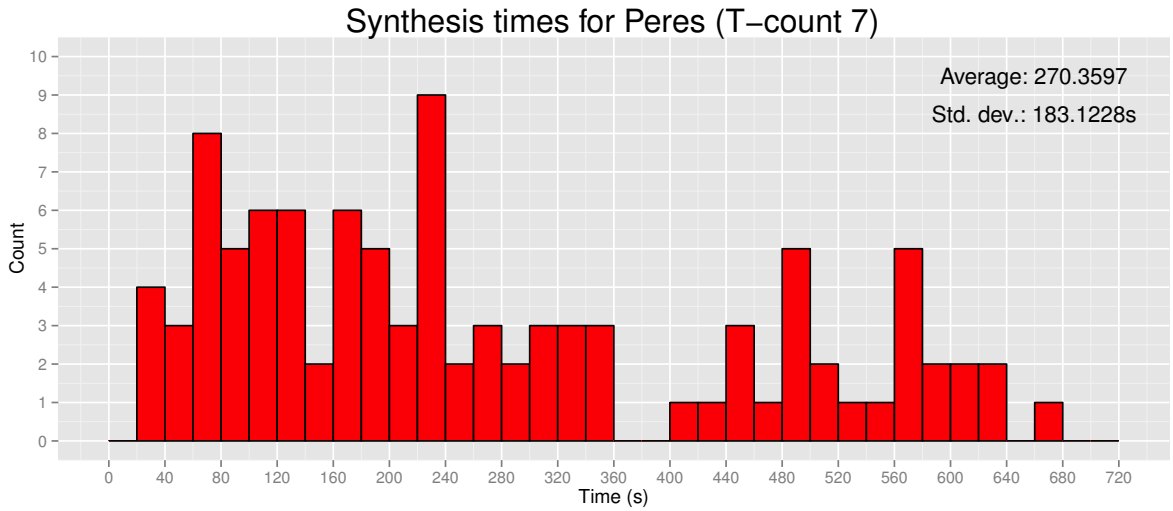


Figure 5.11: Distribution of synthesis times for the Peres (T-count 7, 16 threads, 1/8 points distinguished).

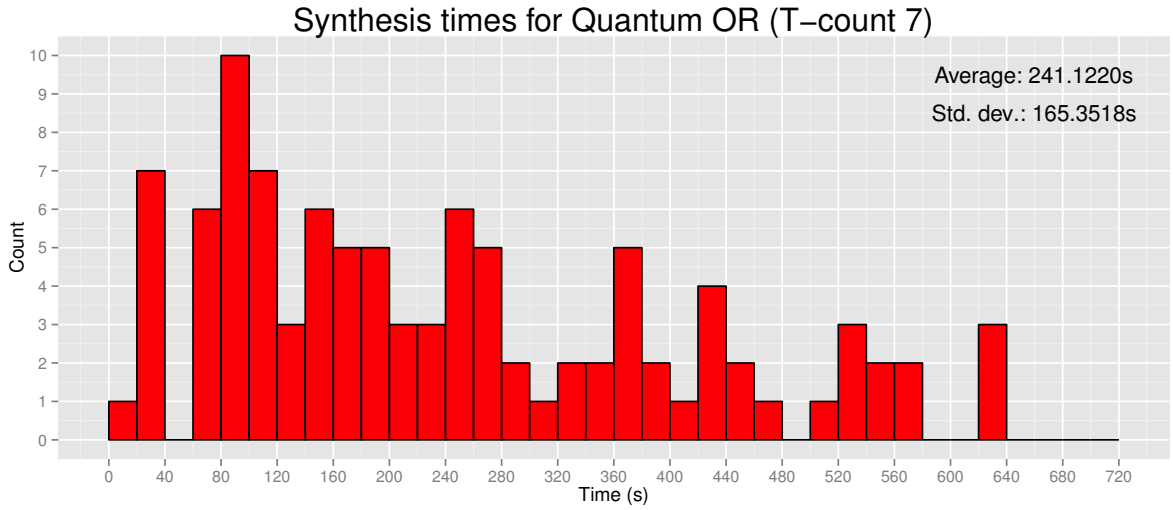


Figure 5.12: Distribution of synthesis times for the Quantum OR (T-count 7, 16 threads, 1/8 points distinguished).

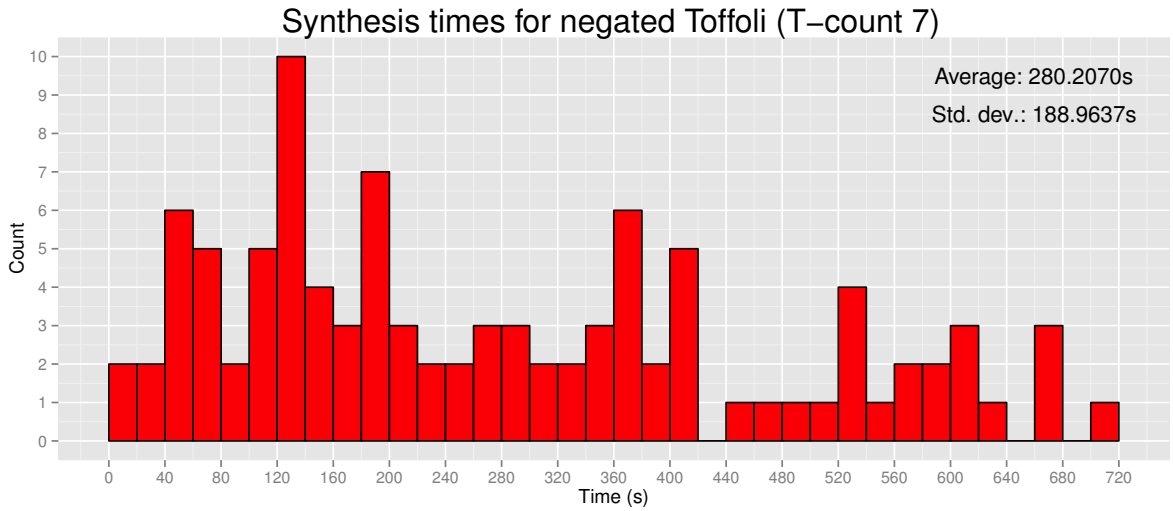


Figure 5.13: Distribution of synthesis times for the negated Toffoli (T-count 7, 16 threads, 1/8 points distinguished).

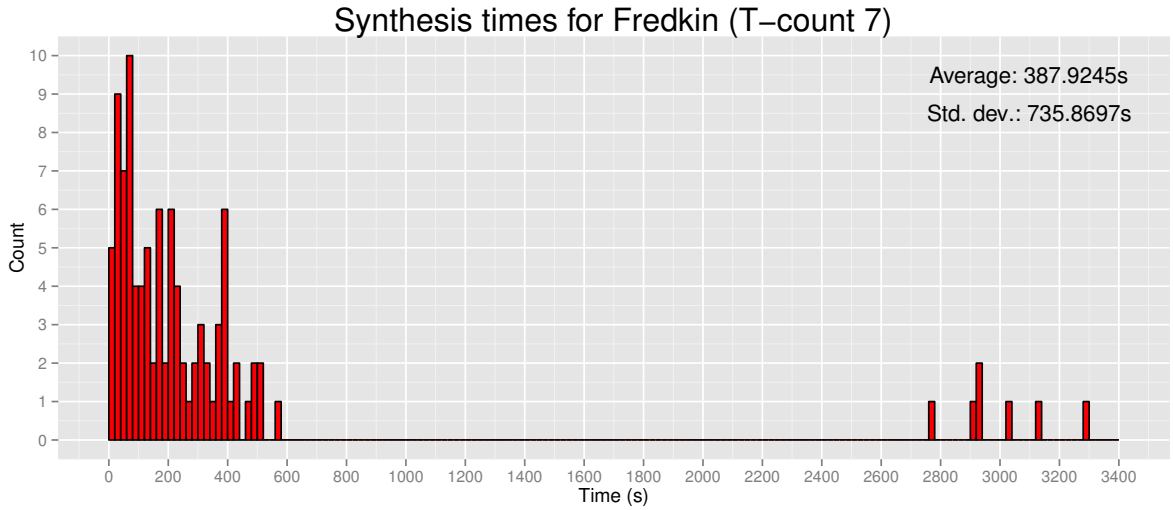


Figure 5.14: Distribution of synthesis times for the Fredkin gate (T-count 7, 16 threads, 1/8 points distinguished).

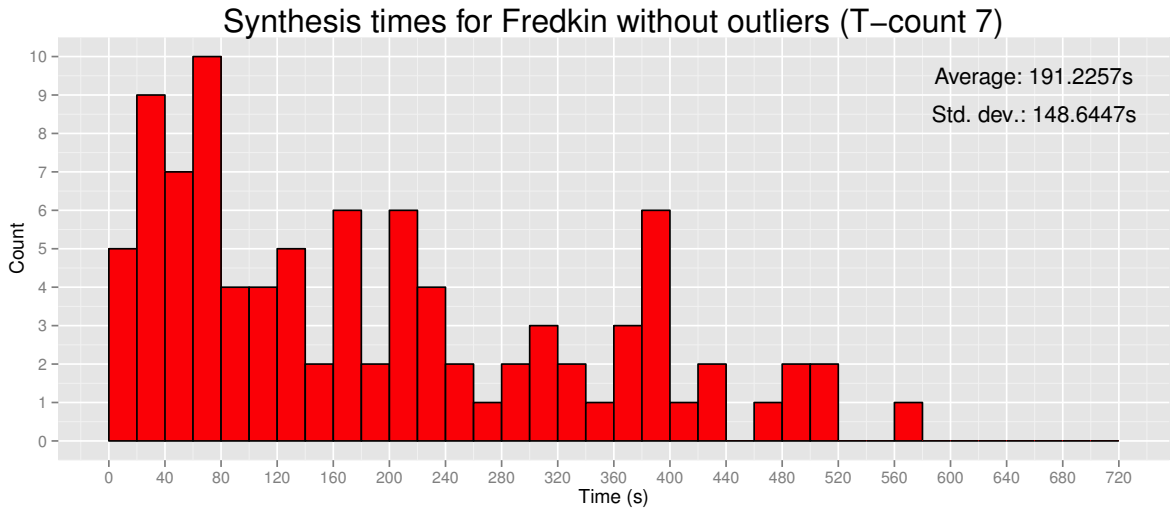


Figure 5.15: Distribution of synthesis times for the Fredkin gate with outliers removed (T-count 7, 16 threads, 1/8 points distinguished).

5.3.3.b A new regime: T-count 9 and T-count 11

The limit of the optimal T-count algorithm for 3-qubit circuits was T-count 7. Using the parallel algorithm, we have successfully synthesized a circuit with T-count 9, as well as T-count 11. The circuits consist of a Toffoli gate with appended controlled-Hadamards, as shown in Figure 5.16. As a consequence of the procedure for generation of "good" Pauli sequences as discussed in Section 5.2.2, successful synthesis of these algorithms is convincing evidence that these circuits do indeed have T-count 9 and 11, and not lower (otherwise, suitable sequences of Paulis may not have been found).

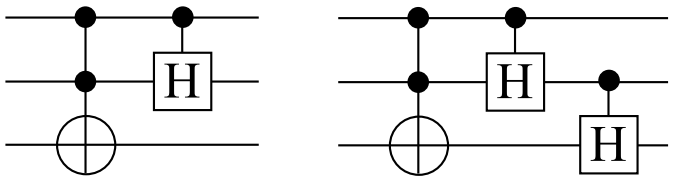


Figure 5.16: Our T-count 9 and 11 circuits of choice: a Toffoli (T-count 7), followed by a controlled-Hadamard (T-count 2), and a Toffoli followed by two controlled-Hadamards.

The mean synthesis time for the T-count 9 circuit was 2728.179s, or about 45 minutes. Virtual memory usage was reported as 1.3GB. The runtime varied greatly, as is depicted in Figure 5.17. We also see the time increase nearly 10-fold from the T-count 7 circuit, from less than 5 minutes to 45 minutes. This is not surprising, given the exponential runtime of this algorithm.

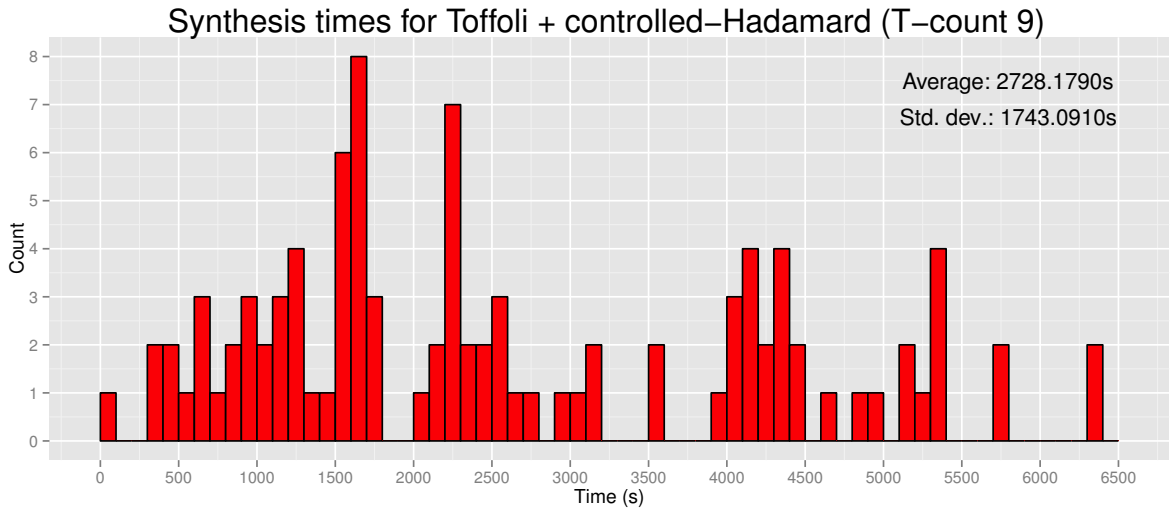


Figure 5.17: The distribution of synthesis times for a Toffoli and a controlled Hadamard (totaling T-count 9). The data was generated using 16 threads with 1/8 of the points distinguished.

Synthesis of the T-count 11 circuit was significantly more time-consuming. The average run time is over 15 hours; the fastest took just under 28 minutes, while the longest took over 41 hours. From Figure 5.18 we can see that the distribution of runtimes is fairly evenly spread over this range. To fully understand the distribution of runtimes we will have to generate a much larger set of data.

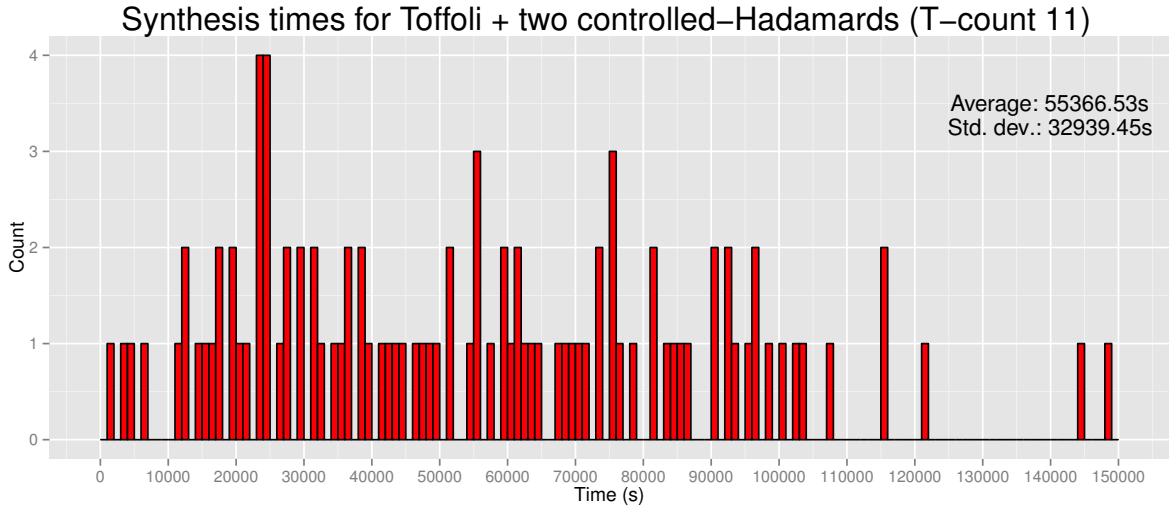


Figure 5.18: The distribution of synthesis times for a Toffoli and two controlled Hadamard (totaling T-count 11). The data was generated using 16 threads with 1/8 of the points distinguished. This plot shows the results of 97/100 separate trials which finished within the 48h allotted run time.

5.3.3.c Varying the number of threads

Another way of profiling the algorithm is to vary the number of threads. The runtime of the program should be roughly inversely proportional to the number of threads. In reality, memory access bottlenecks may reduce the possible benefits of adding more threads. We used 1, 2, 4, 8, and 16 threads to test the

synthesis of the Toffoli circuit. As usual, 100 trials were run for each different thread count. As shown in Figure 5.19, the synthesis is inversely proportional to the number of threads increases. This strongly indicates that the algorithm scales as we claimed.

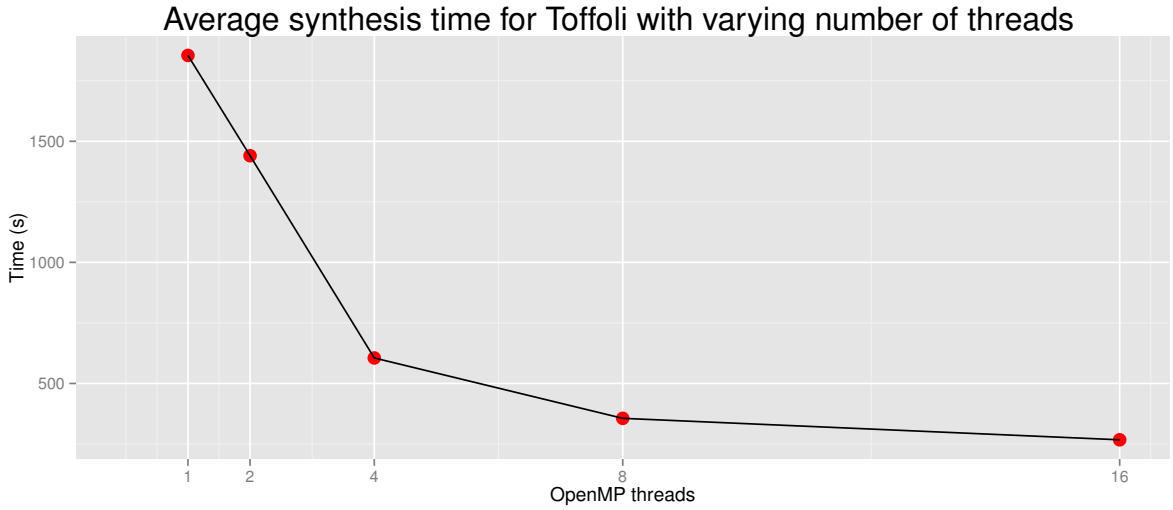


Figure 5.19: Variation in Toffoli synthesis times with the number of OpenMP threads. The runtime scales roughly inversely with the number of threads when all other parameters remain constant.

5.3.3.d Varying the fraction of distinguished points

The fraction of points designated as distinguished was the key parameter of the time-memory tradeoff utilized by our algorithm. The effect of varying this parameter was investigated for the Toffoli gate, for fractions of 1/2, 1/4, 1/8, 1/16, and 1/32. Each trial was run 100 times, and with 16 threads as usual. The results are plotted in Figure 5.20, and are somewhat intriguing.

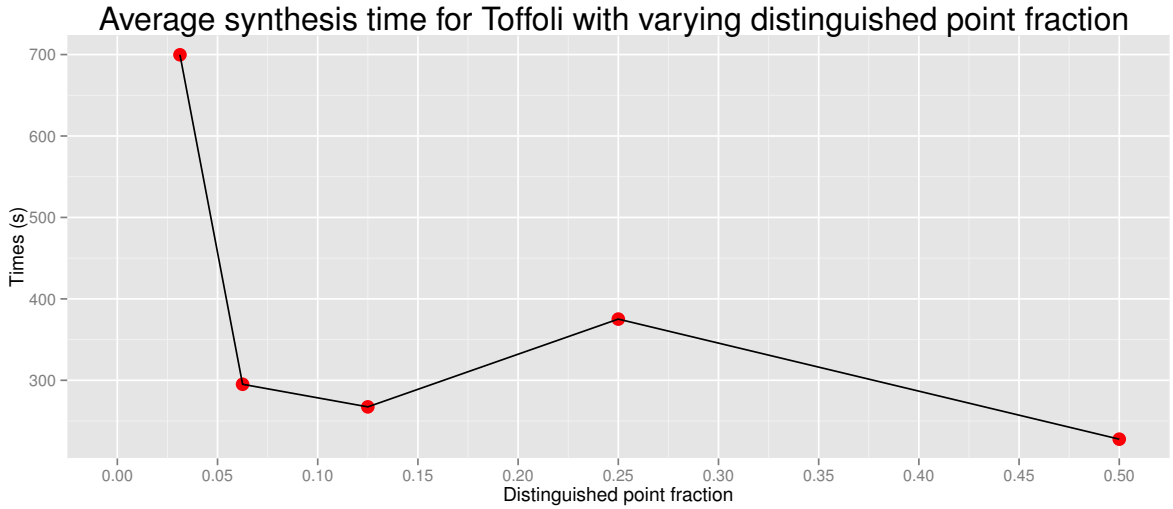


Figure 5.20: Variation in Toffoli synthesis average when the fraction of distinguished points is changed. We see that lower fractions lead to a definite increase in average synthesis time, while higher fractions show inconclusive variation.

We can see that lower fractions of distinguished points lead to a significant increase in average synthesis time. One can intuitively understand this by noting that more operations must be performed on average during each trail, and these operations are nontrivial multiplications of $N^2 \times N^2$ matrices. The increase in time at 1/4 suggests there may be an optimal fraction of distinguished points, however the decrease in average time when the fraction is 1/2 is difficult to explain.

One possibility is that the search space of the Toffoli is too small for there to be any detrimental effect from having too many distinguished points. More distinguished points should mean more storage space is required. However, the memory used, as reported by SHARCNET, was the same: 1.3GB regardless of the fraction of distinguished points set. It is plausible that the storage of distinguished points is fairly negligible, compared to storage of large channel representation matrices by all the threads. For larger circuits, though, choosing the distinguished fraction wisely will be much more important. Future work will involve fine-tuning the choice of this parameter based on the size of the problem.

5.4 Algorithm limitations

First and foremost, our algorithm is limited to the number of cores within a node. Orca has nodes with 16 and 24 cores; other machines have nodes with 8 cores, some even 32 cores. Either way, the level of parallelism needed to solve problems larger than 3 qubits and T-count 11 exceeds the resources of a single node. We propose a solution in Chapter 6.

The bottleneck at the distinguished point set is also a problem with the algorithm - only one thread can access the shared set at a time. Each time, the thread must search the set for a potential match, and add a distinguished point to it if one is not found. This could potentially result in many idle threads waiting for their turn to access the set, all of which is time that could have been spent generating new points or checking for a collision. A means to alleviate this issue is also suggested in Chapter 6.

Ultimately, the achieved runtime depends on the time-memory tradeoff chosen by the programmer. Even within such constraints however, the end result may still be highly variable. In the long run, our algorithm may outperform its counterparts [1, 24] due to a few reasons. First of all, it does not rely on precomputed databases. Databases take significant time to generate as well as many GB to store, in the case of [1, 24]. Though we could in theory make use of databases in the parallel algorithm, it is questionable whether they would be advantageous. They would take time to generate, and they would also have to be accessed by many threads at the same time. A possible database would have to consist of products of Paulis $\widehat{R(Q_i)}$ only, and not the coset label, as the computation of the RHS of Eq. (5.2) would still require multiplication by the circuit matrix $\widehat{U}$ which is different for every circuit we wish to synthesize. Computing the products of $\widehat{R(Q_i)}$ are not causing any bottleneck for the program as they use the sparse-like algorithm as discussed in Section 5.2.2 - thus there would likely be no significant benefit to using a database to store these. Furthermore, and this leads to our second point, the current implementation already synthesizes some circuits faster on average, and this is achieved without databases. Finally, the algorithm can synthesize optimal T-count circuits with a higher T-count than previously possible, even on only 16 threads. This shows great promise for the large-scale implementation of Chapter 6.

Chapter 6

Advanced implementation: hybrid parallel programming

As was mentioned in Chapter 5, using only OpenMP for parallelization limits the number of threads to the number of cores of a single node. In order to wrangle the hundreds, or thousands of cores needed to synthesize larger circuits, we must allow communication between nodes. In this chapter, we present the "advanced" version of the algorithm which uses the Message Passing Interface (MPI) to communicate between nodes, and OpenMP within each node.

6.1 Program flow

The general structure of the program is shown in Figure 6.1. All subroutines and procedures are identical to the threaded version, but the organization of processes is different. A number of MPI processes are spawned at the beginning of the program runtime. Each process should be attached to its own node in order to take full advantage of available resources. Processes are divided into three types: Workers, Collectors, and Verifiers.

Workers do the bulk of the work, as they continuously generate trails and distinguished points. Unless they are notified of a claw being found, a Worker executes the steps in Algorithm 4. Workers make use of the hybrid programming idea. In each step, the node's MPI process spawns a number of OpenMP threads which each generate a trail. The distinguished points that are found by Workers are sent to Collector processes. This is where the distinguished points are processed and stored. Each Collector stores a different subset of the distinguished points, so the Workers must send distinguished points to the right location.

Algorithm 4 MPI Worker subroutine

while claw has not been found **do**
  create an empty container for distinguished points
  spawn OpenMP threads
  each OpenMP thread runs a trail and produces a distinguished point
  each thread stores a distinguished point in the container and terminates
  the master MPI process of the current node sends distinguished points to appropriate Collectors
end while

When a Collector receives distinguished points, it attempts to insert them into the local set. As in the original algorithm, points which are not found in the set are added, and duplicates are tested for

Algorithm 5 MPI Collector subroutine

```
found_distinguished_points ← {}
while claw has not been found do
  receive distinguished points from Worker
  for each distinguished point do
    matching_point ← NULL
    search for distinguished point in found_distinguished_points
    if no match exists in found_distinguished_points then
      add the point to found_distinguished_points
    else
      matching_point ← match from found_distinguished_points
      Send pair of distinguished points to a Verifier
    end if
  end for
end while
```

claws. The steps executed by the Collectors are shown in Algorithm 5.

Two points which are candidates for a claw are sent by the Collector to a dedicated Verifier node - each Collector may have many Verifier nodes associated to it. The Verifiers step through the two trails and check if there is a claw. If a claw is found, the Verifier outputs the solution and signals to the other processors to abort execution. The steps of the Verifiers are shown in Algorithm 6.

Algorithm 6 MPI Verifier subroutine

```
while claw has not been found do
  receive pair of distinguished points from Collector
  if trails merge and claw is found then
    alert other processes that claw is found
    terminate
  end if
end while
```

6.2 Preliminary results

The algorithm is still in development, as a proper termination sequence has yet to be implemented. However we can, for the time being, test the synthesis time by using `mpi_abort` to terminate the execution once any Verifier finds a claw. We tested the algorithm using 64 MPI processes on Orca, using the AMD Opteron nodes. In each case, we tried differing numbers of Workers, Collectors, and Verifiers. MPI calls were made using Boost's compiled library Boost.MPI [32]. As it is very difficult to get access to entire nodes (see Section 6.3), it was not possible to spawn many OpenMP threads per MPI process on the Workers. The number of OpenMP threads was either 1 or 2 per process. We tried using 2 threads simply to investigate the possibility of hyperthreading on the Opteron nodes. The hybrid communication scheme used is *master only* [19, 20], meaning that when Workers send their distinguished points to Collectors, only a single process per node sends all the data at once (rather than each OpenMP thread sending data separately as it is generated).

The results are presented in Table 6.1. We can see that changing the number of Verifiers and Collectors has a noticeable impact on the average runtime. The best case was with 40 Workers, 8 Collectors, 16

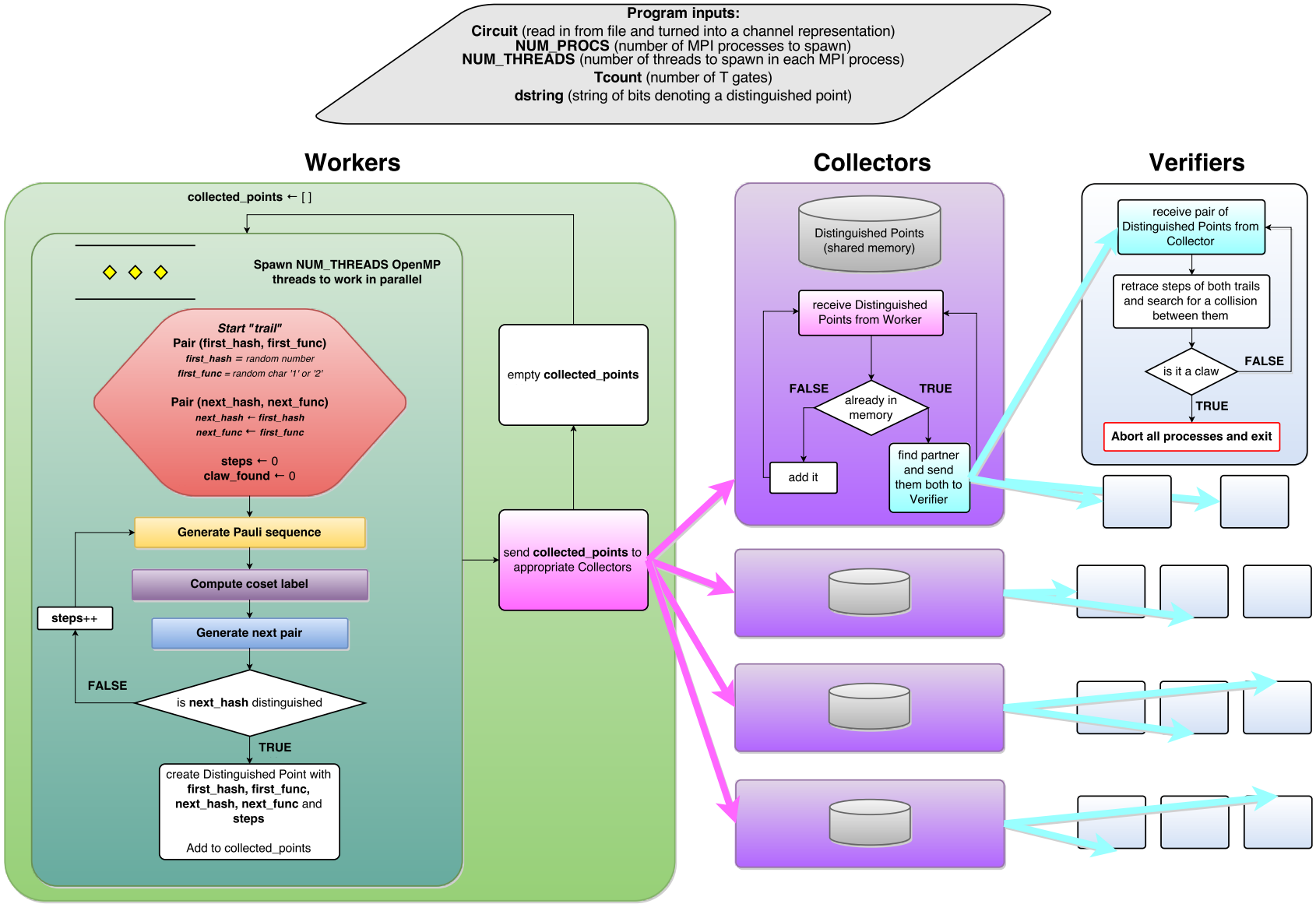


Figure 6.1: Program flow of the production-ready version of the algorithm. Multiple Worker nodes search for distinguished points. OpenMP is used within each node to take advantage of the multiple cores. Found points are distributed via MPI to various Collector nodes, each of which contains a local set of distinguished points, which are independent from those of the other Collectors. Each Collector has a relatively small number of Verifiers at its disposal. When matching pairs of distinguished points are found, the pair is sent to an idle Verifier to determine whether or not there is a claw.

Verifiers, and 1 thread each; this value was still higher than that using only OpenMP and 16 threads. This hybrid algorithm is not yet optimized in the sense that it may not send distinguished points from Worker to Collector in the most optimal way. It is also not taking full advantage of the multiple cores per MPI process, being limited to only 1 or 2 threads.

Workers send distinguished points to Collectors sequentially. On the Worker, all distinguished points meant for a single Collector are gathered and sent as one unit. The same is done for the next Collector, until we have sent out all the data. More threads would mean larger packages of distinguished points would be reaching a Collector with each message. This makes the messages less "wasteful", in a sense, as there is a higher throughput of distinguished points for the same amount of messages. On the other hand, the Collector will take longer to process each package of distinguished points, so messages for other Workers may have to wait longer before they can be handled.

One interesting point to note is that in some cases we see what looks like the negative consequences of hyperthreading. For example, in the case of 48 Workers and 40 Workers, using 1 thread rather than 2 appears to be more advantageous. However, what is more interesting is there are other cases when we see the opposite effect (such as for 28 Workers). Looking only at the variation for a single thread count, we can see that there is clearly an ideal number of Workers for a given number of Collectors and Verifiers. When we utilize a hybrid model, the multiple threads in a sense function as additional Workers. For the case of 28 Workers, it's possible that having $28 \times 2 = 56$ processors generating points is more balanced than simply having 28, when we have only 12 Collectors and 24 Verifiers. Similarly, for the case of 40 Workers, perhaps just 40 processors generating points is better than the $40 * 2 = 80$, for only 8 Collectors and 16 Verifiers. The number of Workers, Collectors, and Verifiers is thus a large tradeoff, the optimality of which must be explored in future work where hyperthreading does not occur.

Table 6.1: Average runtimes for Toffoli synthesis with varying numbers of Workers, Collectors, Verifiers, and OpenMP threads. Each run used a total of 64 MPI processes; the number of OpenMP threads per Worker was varied. All results are averages of 100 separate trials, with 1/8 of the points marked as distinguished.

Workers	**Collectors**	**Verifiers**	**Threads**	**Average time (s)**	**Standard deviation (s)**
48	4	12	2	793.1940	798.9989
40	8	16	2	551.2930	487.7952
28	12	24	2	697.9576	654.8881
16	16	32	2	1165.3257	812.2001
48	4	12	1	595.9559	547.3147
40	8	16	1	492.7831	457.8875
28	12	24	1	739.1256 [a]	564.2923
16	16	32	1	1122.4231 [b]	778.2605

[a] Due to an issue with the SHARCNET system, 3 of the jobs did not run at all. One job exceeded the 1hr maximum time limit it was allotted; thus, this result is the average of 96 trials rather than the standard 100.

[b] Also due to an issue with the SHARCNET system, only 89 trials were successful.

6.3 Advantages and limitations

Evidently, a major advantage of this more advanced implementation is the ability to use multiple nodes. The algorithm also scales appropriately for larger circuit scenarios (e.g. if we know we will have twice as many distinguished points, we can add twice as many Workers, and twice as many Collectors).

Having multiple Collectors should help ease the bottleneck normally encountered when a number of threads are using a single shared memory to store distinguished points. In addition, each shared set of distinguished points is smaller overall, so less time will be wasted searching through each set to find a match. Furthermore, adding in the Verifiers means that Collectors are always available to process distinguished points, in contrast to the previous algorithm where a single thread does all the work of finding a distinguished point, adds it to the set, and checks if there is a collision.

Our hybrid implementation does have some limitations. The first is the sheer complexity of the programming itself. One point of pain has been designing an effective termination sequence so that when a Verifier finds a claw, it can notify the other processes without causing a deadlock (leaving some processes waiting endlessly for messages that will never arrive because the processor that should have sent them has already terminated). Future work will involve using MPI debugging programs to analyze the flow of message passing in the program, and ensure that all processors terminate in a timely manner.

Though it is the simplest to implement, the *master only* model with OpenMP threading may not be the most efficient means of communication. In the Workers, threads are spawned, distinguished points are generated and stored, and then all threads are killed before any data is sent. This process happens repeatedly throughout the program execution, and is inefficient in a number of ways. First of all, there is a certain amount of overhead required in spawning OpenMP threads [19, 20]. Second, it may take time before distinguished points are able to be sent to a Collector. Many Workers are sending to the same Collector simultaneously, and these messages are queued up and received sequentially. If a node is waiting to send to a Collector, it is wasting valuable time that could be spent generating more distinguished points. It may be more efficient to use a hybrid model which balances the computation and communication, as suggested in [19, 20]. We could spawn a set of OpenMP threads only once and have them generate distinguished points continuously. Communication could be executed by a handful of designated threads (for example, when a certain number of distinguished points have been generated and stored in the shared memory of the node). In this model, there would never be an instance where all the threads on a Worker node are idle simultaneously, waiting to communicate with a Collector. The downside to this model, however, is its complexity to implement.

Another limitation is the amount of hardware required: a large set of full, dedicated nodes. To run this hybrid program in the best possible way requires each MPI process to be tied to a single node, with no other processes running on that node except for the set of OpenMP threads we spawn. It proved difficult to get access to individual nodes on the SHARCNET platform due to the large user base and long wait times. To that end, we are looking into obtaining an allocation on an IBM Blue Gene/Q, located at the University of Toronto [1]. The machine itself has over 65,000 cores, of which we are guaranteed at least 1,024.

[1] Documentation for the Blue Gene/Q can be found at `http://wiki.scinethpc.ca/wiki/index.php/BGQ`

Chapter 7

What else can we do with this framework?

Let us summarize the key results of this work. We have merged two algorithms (meet-in-the-middle circuit synthesis, and parallel collision finding) to develop a framework for parallel quantum circuit synthesis. An implementation using OpenMP and just 16 concurrent processors has already proved this technique to be beneficial. We synthesized many known circuits faster on average than the previous meet-in-the-middle algorithm. Furthermore, we synthesized two 3-qubit circuits with optimal T-count 9 and 11, which were previously not possible. We have high hopes for the hybrid OpenMP/MPI implementation which is under development, and future work will involve analyzing and improving this technique such that it becomes a scalable means of synthesizing even larger circuits than the ones shown in previous chapters.

Parallel quantum circuit synthesis is a framework that can be applied to several other quantum circuit synthesis methods, and there are many more different improvements one can make to the implementations presented in Chapters 5 and 6. We list a few of them here.

1. **Approximate circuit synthesis**
   Not every unitary matrix has elements over the ring $\mathbb{Z}[i, \frac{1}{\sqrt{2}}]$. Many commonly used quantum operations would not be synthesizable over our gate set, e.g. the rotations performed in the quantum Fourier transform, or the 2-qubit controlled-T gate [1].

   Although there are existing algorithms for approximate synthesis, in future work (i.e. my PhD work) we plan to investigate the possibility of building a new method that fits within the parallel framework. One can imagine the program execution to be very similar, but we could call a candidate circuit matrix distinguished if it is within ε of some finite set of target unitaries.

2. **Synthesis over arbitrary bases**
   We mentioned in Chapter 2 the work done to synthesize circuits of the V-basis. Depending on the physical implementation of a quantum computer, gate sets other than the Clifford+T set may be more conducive. It may thus be useful to extend the implementation of the synthesis algorithm to other gate sets, and perhaps let the user choose the one most pertinent to their application.

   In an ideal world, we would let the user specify a gate set rather than them choosing from a fixed set. However, much of the implementation currently depends on the nice properties of optimal

[1] Despite having its elements within the designated ring, the controlled-T gate does not satisfy the determinant conditions discussed in Chapter 2 [12]

T-count synthesis - the user would need to rewrite the routines to work within their own basis (e.g. determine the mapping from integers and hashes to elements from their sets). One possibility would be to release a template of the implementation, and have the user fill in the specific details of their gate set.

3. **Extending parallelism to harness GPUs**
   Recent advances in GPU technology have led to commercial sale of devices with thousands of cores [2]. Looking ahead to the future synthesis of very large circuits, this is on the order of cores we will need. Harnessing the capabilities of a GPU may be a useful area to investigate. We considered the possibility of using NVIDIA's CUDA libraries for the implementations presented above, but we did not implement it at this time for a number of reasons, stated below.

   The original idea was to launch one trail per GPU core. However, all cores may take varying amounts of time to run. When launching a CUDA program (called a kernel), we would have to copy all the memory from the CPU to the GPU at the beginning, and then the resultant distinguished points back at the end once *all* threads have terminated. Copying memory from CPU to GPU (and vice-versa) is a slow and thus "expensive" operation computationally, and should be minimized, so constantly copying distinguished points back and forth would likely hinder the performance of the algorithm. Furthermore, if we launch groups of many threads at a time, many threads will be idle at any given time, particularly if one of the trails goes into a self-loop and has to reach the maximum iteration limit before terminating. The program would have wait until all threads have completed before transferring the results back to the CPU.

   Another idea we considered was, instead of having one trail per GPU core, use standard MPI and just use CUDA to do the intensive matrix multiplication in parallel. However, this would be even more expensive memory wise, as we would have to copy memory back and forth to the GPU at each step after we multiply matrices. Furthermore, many of the matrices in the current implementation are class objects, and all their elements are class objects as well. This would make memory copying a difficult task, as we would have to copy the objects recursively, or layer by layer. CUDA 6.0 (and higher versions) have implemented a Unified Memory scheme which does this [33], but finding a powerful enough research cluster with both thousands of CPUs *and* a GPU running CUDA 6.0 or greater is not altogether easy.

   The final reason is hardware. Anyone with a handful of CPUs can run one of the current versions of the algorithm. However, using GPU acceleration would require researchers to have dedicated, expensive hardware. Furthermore, if we use CUDA, that hardware would have to be NVIDIA. This is an unnecessary limitation on the user base of such an algorithm. Using OpenGL instead of CUDA may be a way to circumvent this problem.

   Still, GPU acceleration is a possibility for the future. Currently, the best option would user-optional modularization of the matrix multiplication for the GPU.

Overall, the current prototypes show great promise for synthesis of very large, previously unmanageable circuits. We provided two implementations (OpenMP and hybrid OpenMP/MPI) of a single circuit

[2] See, for example, the NVIDIA GTX Titan Z, a powerful desktop graphics card with nearly 6,000 cores, just waiting to tackle a scientific computing problem if you have 3,340$ to spare. `http://www.nvidia.com/gtx-700-graphics-cards/gtx-titan-z/`.

synthesis problem (optimal T-count), but there are many more possibilites. Future work on this topic will focus on further developing a versatile tool: a packagable application that any other researcher can use out of the box to synthesize their circuits on hardware of their choice.

Bibliography

[1] D. Gosset, V. Kliuchnikov, M. Mosca, and V. Russo (2014) Quantum Info. Comput. **14** (15-16)

[2] M. A. Nielsen and I. L. Chuang *Quantum computation and quantum information.* Cambridge University Press, Cambride, 2000.

[3] A. Barenco et. al (1995) Phys. Rev. A **52** (5) 3457-3467

[4] D. P. DiVincenzo (1995) Phys. Rev. A **51** (2) 1015-1022

[5] P. O. Boykin, T. Mor, M. Pulver, V. Roychowdhury and F. Vatan (1999) `http://arxiv.org/abs/quant-ph/9906054`

[6] D. Gottesman (1998) Phys. Rev. A **57** (1) 127-137

[7] V. Kliuchnikov, D. Maslov and M. Mosca (2013) Phys. Rev. Lett. **110** 190502

[8] P. Selinger (2015) Quantum Info. Comput. **15** (1-2) 159-180.

[9] V. Kliuchnikov, D. Maslov, and M. Mosca (2012) `http://arxiv.org/abs/1212.6964`

[10] N. J. Ross and P. Selinger (2014) `http://arxiv.org/abs/1403.2975`

[11] V. Kliuchnikov, D. Maslov, M. Mosca (2013) Quantum Info. Comput. **13** (7-8) 607-630

[12] B. Giles and P. Selinger (2013) Phys. Rev. A **87** 032332.

[13] L. B. Levitin and T. Toffoli (2009) Phys. Rev. Lett. **103** 160502

[14] G. M. Amdahl (1967) AFIPS (Spring) joint computer conference (483-485)

[15] J. L. Gustafson (1988) Commun. ACM **31** (5) 532-533

[16] Udacity's *Intro to Parallel Programming* online course, built in conjunction with NVIDIA, provides an excellent overview of the ideas behind GPU programming (and many other common parallel programming techniques), and was a very useful resource. `https://www.udacity.com/course/cs344`

[17] `http://openmp.org/wp/`

[18] `http://www.mpi-forum.org/`

[19] R. Rabenseifner (2003) *Hybrid Parallel Programming: Performance Problems and Chances* in *Proceedings of the 45th CUG Conference. (CUG SUMMIT 2003),* May 12-16, Columbus, Ohio, USA.

[20] R. Rabenseifner, G. Hager, and G. Jost (2009) *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes* in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2009),* Feb 18-20, Weimar, Germany. pp. 427-236.

[21] C. M. Dawson and M. A. Nielsen (2005) Quantum Info. Comput. **6** (1) 81-95

[22] A. Bocharov, Y. Gurevich and K. M. Svore (2013) Phys. Rev. A. **88** 012313

[23] A. W. Harrow, B. Recht and I. L. Chuang (2002) J. Math. Phys. **43** 4445

[24] M. Amy, D. Maslov, M. Mosca, and M. Roetteler (2013) IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **32** (6), 818

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein *Introduction to Algorithms, 2nd edition.* The MIT Press (2001).

[26] D. Gottesman and I. L. Chuang (1999) Nature **402** 390-393

[27] S. Aaronson and D. Gottesman (2004) Phys. Rev. A **70** 052328

[28] P. C. van Oorschot and M. J. Wiener (1999) J. Cryptology **12** (1) 1-28

[29] P. C. van Oorschot and M. J. Wiener (1996) Advances in Cryptology - CRYPTO '96, Lecture Notes in Computer Science **1109** 229-236

[30] `http://www.boost.org/users/history/version_1_57_0.html`

[31] R. Koenig and J. A. Smolin (2014) `http://arxiv.org/abs/1406.2170`

[32] `http://www.boost.org/doc/libs/1_51_0/doc/html/mpi.html`

[33] `http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/`

[34] A. R. Calderbank, E. M. Rains, P. W. Shor, and N. J. A. Sloane (1997) `http://arXiv:quant-ph/9608006`

Appendix A

Binary symplectic representation

Binary symplectic representation is a conveninent tool used in quantum information to represent Pauli matrices as binary matrices (or strings) [34]. In general, a single-qubit Pauli can be written in the form $Z^a X^b$. For convenience, we note this as $(a|b)$. From this we obtain the correspondence

$$I \sim (0|0)\,, \quad X \sim (0|1)\,, \quad Z \sim (1|0)\,, \quad Y \sim (1|1)\,. \tag{A.1}$$

The left side of the bar represents the 'Z' component, and the right side the 'X' component.

We can represent multi-qubit Paulis in a similar way. A general n-qubit Pauli has the form $Z^{a_1} X^{b_1} \otimes \cdots \otimes Z^{a_n} X^{b_n}$. Its binary symplectic representation is $(a_1 \cdots a_n | b_1 \cdots b_n)$. For example, $X \otimes Y \otimes Z$ would be represented as $(011|110)$.

In the implementation of the synthesis algorithm, we used binary symplectic representation to map Paulis to the integers. For example, for the Pauli $X \otimes Y \otimes Z$ above, we would take its representation 011110 and convert it to an integer: 30. To execute a random walk through the space of possible products of $R(Q_i)$ then, it suffices to randomly walk over the integers, and the corresponding products of Paulis are easy to extract.