

A foundation for quantum programming and its
highly-parallel virtual execution

Yves Vandriessche

23th November 2012

Print: Silhouet, Maldegem

©2012 Yves Vandriessche

2012 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5718 226 6
NUR 989
Legal deposit D/2012/11.161/161

All rights reserved. No parts of this book may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

Abstract

The topic of this dissertation stands at the crossroads of two emerging technologies: highly-parallel computing and quantum computing. Both seek to go beyond the limits of current computing practice: in the absence of further miniaturization, highly-parallel computing seeks to increase the available computer power substantially. Quantum Computing looks beyond the classical approach to computation, leveraging quantum-mechanical effects to increase computing power in a fundamental way. In the absence and also in the presence of actual quantum computers it is crucial to support the development of quantum computer applications by creating an appropriate software framework. Creating such a framework currently faces two major research challenges. First, one needs to separate the realization of the quantum computer from the software running on it, whether it is an actual physical or a simulated realization. Our first contribution lies with defining a layered software architecture, ranging from a design tool at the higher abstraction levels to a simulated execution layer underneath. At its core sits a ‘Quantum Virtual Machine’ using the Measurement Calculus as the underlying quantum computational model. Such a virtual machine layer makes the application developed on top of it oblivious to how it is executed. The simulated execution of this quantum virtual machine is fundamentally a computationally intensive task. Parallel computing offers a way to throw more computational resources at the problem, in the light of the saturation of sequential performance of current computers. Simulation is the domain of the second challenge, as it has very high performance requirements. The simulation problem thus becomes: how does one expose the inherent parallelism of quantum computing simulation in a fundamental way, so as to be usable for the highly-parallel computers to come? Our second contribution is to provide a formal translation of quantum programs to highly-parallel dataflow computations, and to develop a virtual environment that compiles and executes such programs. This work demonstrates several properties of both the formal as well as the virtual level of computation that validate our approach.

Korte Inhoud

Het onderwerp van deze verhandeling ligt op het kruispunt van twee opkomende technologieën: massief parallel computers en kwantum computers. Deze domeinen bieden een uitweg aan om de rekenbeperkingen van huidige computersystemen te overstijgen. Gezien de recente afvlakking van de miniaturisatietrend, kunnen massief parallelle computers blijven de rekenkracht substantieel vergroten. Kwantum computers werpen een blik voorbij de huidige klassieke aanpak en gebruiken kwantummechanische effecten om de rekenkracht op een fundamentele wijze te verhogen. Bij het ontbreken, maar ook in aanwezigheid van zulke kwantum computers is het cruciaal om het ontwikkelen van nieuwe toepassingen te ondersteunen door middel van een bijpassend software raamwerk. Twee grote onderzoeksvragen dringen zich op bij het maken van een dergelijk raamwerk. Hierbij moet eerst de concrete verwezenlijking van de kwantum computer gescheiden worden van de software die er bovenop uitgevoerd wordt, zij het een fysieke of een virtuele verwezenlijking betreft. Onze eerste bijdrage ligt bij het definiëren van een gelaagde software-architectuur, gaande van het ontwerp op een hoger abstractieniveau tot een gesimuleerde uitvoeringsomgeving onderaan. Centraal bij onze aanpak is de ‘Quantum Virtual Machine’, die de *Measurement Calculus* gebruikt als model voor kwantum computer berekeningen. Deze virtuele machine zorgt dat de ervoor ontwikkelde toepassingen niet afhangen van de onderliggende uitvoeringsstrategie. De gesimuleerde uitvoering van deze kwantum virtuele machine is in wezen een rekenintensieve taak. Parallelle computers bieden een manier aan om meer rekenkracht aan te wenden, gezien de verzadiging van sequentiële rekenprestatie in huidige computersystemen. De tweede onderzoeksvraag wordt bijgevolg: hoe legt men op een fundamentele wijze het inherente parallelisme bloot in de gesimuleerde uitvoeringsomgeving, zodat deze toepasbaar is op de aankomende massief parallelle computers. Onze tweede bijdrage biedt een formele vertaling van kwantum programma’s naar massief parallelle *dataflow* programma’s. Daarbij ontwikkelen we een virtuele omgeving om dergelijke programma’s te vertalen en uit te voeren. Deze uitvoeringsomgeving legt verschillende theoretische en praktische eigenschappen aan de dag die onze aanpak verantwoorden.

Acknowledgements

First and foremost, I would like to thank my promoters Theo and Ellie D'Hondt. Much of the research work, presentation and validation in this dissertation only came into its own just before I started writing. I thank them for their patience and faith in my ability to pull through with this rather unusual combination of research domains.

This work would not have been possible without the support I enjoyed from family and friends. Jemma, my better half, who not only kept me sane this past year, but also helped in any way she could while suffering my PhD-mode lifestyle. My parents, whose unwavering belief and selfless support have carried me through college and the heavy PhD-writing period. I thank my friends, who have seen me drop off the radar for a while, for the many encouragements sorely needed distractions.

Finally, my colleagues at the SOFT (and old PROG) lab. I have learned from first hand experience the importance a good lab environment for fostering interesting research. This PhD wouldn't have been possible without the healthy balance of interest and knowledge, passion versus cynicism, the conceptual and the practical, amicability and professionalism, and of course work and play (with the occasional drink!)

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 1.1 | Context | 1 |
| 1.1.1 | Overview | 1 |
| 1.1.2 | Quantum Computing | 2 |
| 1.1.3 | Quantum Programming | 3 |
| 1.1.4 | Parallel Computing | 4 |
| 1.2 | Approach | 6 |
| 1.2.1 | Quantum programming | 6 |
| 1.2.2 | Parallel computing | 7 |
| 1.2.3 | Structure | 8 |
| 2 | Measurement-Based Quantum Computing | 9 |
| 2.1 | Requirements for a Quantum Programming Paradigm | 10 |
| 2.2 | Quantum Computing | 12 |
| 2.2.1 | Overview | 12 |
| 2.2.2 | Quantum Programming Languages | 14 |
| 2.3 | Operational Basics | 17 |
| 2.3.1 | Overview | 17 |
| 2.3.2 | Computation State and Preparation | 18 |
| 2.3.3 | Entanglement: E | 20 |
| 2.3.4 | Measurement: M | 22 |
| 2.3.5 | Corrections | 24 |
| 2.3.6 | Signals | 24 |
| 2.3.7 | Command Sequence | 25 |
| 2.4 | Measurement Calculus | 27 |
| 2.4.1 | Patterns | 27 |
| 2.4.2 | Rewrite rules | 29 |
| 2.5 | Measurement Calculus Properties | 31 |
| 2.5.1 | Conceptual framework | 31 |

| | | |
|----------|--|-----------|
| 2.5.2 | Practical framework | 32 |
| 2.5.3 | Matching the Requirements | 32 |
| 3 | Practical Foundation for a Quantum Programming Paradigm | 35 |
| 3.1 | Introduction | 36 |
| 3.2 | Architecture | 39 |
| 3.3 | Application Layer | 41 |
| 3.3.1 | Library Approach: First Class Patterns | 42 |
| 3.3.2 | Graphical Pattern Editor | 47 |
| 3.4 | Pattern Layer | 50 |
| 3.4.1 | Defining generalized compositions | 50 |
| 3.4.2 | Generalized Composition | 53 |
| 3.4.3 | Pattern assembly | 56 |
| 3.5 | Execution Layer | 57 |
| 3.5.1 | Machine model: vision of a Quantum Computer | 57 |
| 3.5.2 | The Quantum Virtual Machine | 58 |
| 3.5.3 | Interface and Syntax | 59 |
| 3.5.4 | Execution | 60 |
| 3.6 | Realization layer | 66 |
| 3.6.1 | Naive Linear Algebra approach | 66 |
| 3.6.2 | Correction and Entangle Operator optimization | 67 |
| 3.6.3 | Measurements | 68 |
| 3.6.4 | Storage optimization | 70 |
| 3.7 | Discussion | 72 |
| 4 | Landscape of Parallel Computing | 75 |
| 4.1 | Overview | 76 |
| 4.2 | von Neumann-style Parallel Computing | 77 |
| 4.2.1 | Sequential microprocessor adoption | 77 |
| 4.2.2 | The cost of complexity | 77 |
| 4.2.3 | The drive for parallelism | 79 |
| 4.2.4 | Parallelism limitations of von Neumann hardware | 81 |
| 4.2.5 | Impact on software | 83 |
| 4.2.6 | Status quo | 83 |
| 4.3 | Parallel Computing | 85 |
| 4.3.1 | Historical Context | 85 |
| 4.3.2 | Parallel Computational Models | 86 |
| 4.4 | Dataflow for Quantum Computing Simulation | 92 |
| 4.4.1 | Overview | 92 |
| 4.4.2 | Fine Grainedness | 93 |
| 4.4.3 | Asynchronicity | 95 |

| | | |
|----------|--|------------|
| 4.4.4 | Analyzability | 96 |
| 4.4.5 | Implicitness | 97 |
| 4.4.6 | Data-orientedness | 98 |
| 4.4.7 | Conclusion | 99 |
| 5 | Bridging QC and CC: mapping Measurement Calculus to Data-flow | 101 |
| 5.1 | Bridging measurement patterns and implementation using models | 101 |
| 5.1.1 | Goal | 101 |
| 5.1.2 | Requirements | 102 |
| 5.1.3 | Overview | 103 |
| 5.2 | Coarse grained graphs | 105 |
| 5.2.1 | Measurement Calculus Graph Model | 105 |
| 5.2.2 | Coarse Graph Model | 107 |
| 5.2.3 | Positional Coarse Graph (pCG) | 110 |
| 5.3 | Fine-grained graph model | 125 |
| 5.3.1 | Overview | 125 |
| 5.3.2 | Single qubit states | 126 |
| 5.3.3 | Notation | 130 |
| 5.3.4 | Permutation Operator | 131 |
| 5.3.5 | Multiple qubit states | 134 |
| 5.3.6 | Construction | 138 |
| 5.3.7 | Discussion | 138 |
| 5.4 | Conclusion | 141 |
| 6 | Execution, Implementation and Validation | 143 |
| 6.1 | Analytical Validation | 144 |
| 6.1.1 | Theoretical Parallelism | 144 |
| 6.1.2 | Average Parallelism | 146 |
| 6.2 | Implementation | 150 |
| 6.2.1 | Structure | 150 |
| 6.2.2 | Technology | 157 |
| 6.2.3 | Compiling for CnC | 162 |
| 6.3 | Experimental Validation | 171 |
| 6.3.1 | Goals | 171 |
| 6.3.2 | Approach | 172 |
| 6.3.3 | Experiments | 174 |
| 6.4 | Conclusion | 181 |

| | | |
|----------|--|------------|
| 7 | Conclusions and Future Work | 183 |
| 7.1 | Quantum Programming | 184 |
| 7.2 | Parallel Execution | 185 |
| 7.3 | Contributions | 187 |
| 7.4 | Future Work | 189 |
| 7.4.1 | Highly-parallel performance challenge | 189 |
| 7.4.2 | Expanding the quantum programming framework | 189 |
| 7.4.3 | Dissemination | 190 |
| A | Linear Algebra formulation of Quantum Computing | 191 |
| B | The von Neumann Architecture | 195 |
| C | Controlled-Phase gate decomposition | 197 |
| | Bibliography | 199 |

List of Figures

| | | |
|------|---|-----|
| 2.1 | Two typical quantum circuit examples. | 13 |
| 2.2 | Bloch sphere representation of diagonal basis vectors. | 19 |
| 2.3 | Bloch sphere representations of various measurement bases. | 22 |
| 3.1 | Overview of the layered architecture structure. | 40 |
| 3.2 | Recursive quantum circuit for QFT. | 41 |
| 3.3 | Quantum circuit for controlled-phase decomposition. | 42 |
| 3.4 | Editor tool screenshots showing step-by-step creation of QFT(3). | 48 |
| 3.5 | Graphical representation of controlled-NOT composition. | 52 |
| 3.6 | Schematic working model of a quantum co-processor. | 58 |
| 3.6 | Example command-line interactions with <code>qvm</code> application. | 62 |
| 4.1 | Categorization of parallel computing systems. | 87 |
| 4.2 | Simple dataflow graph example. | 93 |
| 5.1 | Schematic overview of the multiple models spanning quantum and classical computing. | 102 |
| 5.2 | Comparing the operations reshuffling amplitude positions in case of a cyclic shift (a) and transposition (b). | 116 |
| 5.3 | Transformation rule for correction operations $C_i = X_i$ or Z_i | 121 |
| 5.4 | Transformation rule for measurement operation M_i | 121 |
| 5.5 | Introducing explicit merge operation nodes. | 122 |
| 5.6 | Transformation rule for entanglement operation $E_{i,j}$ | 122 |
| 5.7 | Transformation rule contracting a chain of position changes. | 123 |
| 5.8 | Relation between coarse- and fine-grained representations. | 125 |
| 5.9 | Fine-grained graphs computing a general single-qubit unitary. | 127 |
| 5.10 | Large fine-grained graph example, combining all MC operations. | 139 |
| 6.1 | Average parallelism data points for different program sizes. | 148 |
| 6.2 | Measurement Calculus Compiler (<code>mcc</code>) structure overview. | 151 |

| | | |
|------|---|-----|
| 6.3 | Automatically generated coarse graph for a simple EMC-pattern. . . | 153 |
| 6.4 | Automatically generated coarse graph of the two-qubit Deutsch-Jozsa algorithm. | 154 |
| 6.5 | CnC graph for the vector tensor product. | 160 |
| 6.6 | CnC producer/consumer pipeline organization example. | 163 |
| 6.7 | The CnC graph for the pipelined vector tensor product. | 164 |
| 6.8 | Fast Bit-level realization of the X step's differentiating condition and index manipulation. | 168 |
| 6.9 | Visualization of the coarsening optimization that merges several coarse operations into one. | 170 |
| 6.10 | Parallel Speedup S_n values of multiple execution runs for the $QFT(16)$ measurement pattern. | 175 |
| 6.11 | Maximum memory use compared to runtime for different $QFT(n)$ parallel implementations. | 177 |
| 6.12 | Effect of introducing CnC tuner optimizations, reducing the overhead of individual parallel step instances. | 178 |
| 6.13 | Comparing runtime of original and optimized parallel implementation. | 179 |
| B.1 | The von Neumann architecture for a general purpose sequential computer, omitting Input/Output. | 196 |

Chapter 1

Introduction

1.1 Context

1.1.1 Overview

Quantum Computing promises astonishing applications that would be hard or even impossible to achieve with classical computers. Even though it is known how quantum computers work in principle, it is still hard to find new algorithms and applications. There is a need for an overarching set of tools, methodologies and languages for programming such quantum computers: a Quantum Programming Paradigm. The practical¹ foundation of such a quantum programming paradigm is, necessarily, an execution platform: a low-level virtual model of a quantum computer. This execution platform evidently needs to be automated, but the current lack of physical programmable quantum computers means that these need to be simulated by classical computers. Quantum computing simulators face fundamental performance problems; a linear increase in the quantum computation space translates into an exponential use of classical computation resources. Performance today means parallel performance; to get the most out of current computing hardware, the original problem needs to be expressed as a parallel computation. Sequential performance used to increase exponentially over time, driven by miniaturization of the microprocessor. But, this has hit a technical wall, putting the performance burden to produce parallel computations upon the software, rather than the processor hardware. We see this upheaval as an opportunity to investigate a highly-parallel formulation of a virtual quantum computing platform. This dissertation brings together both the quantum and the classical world by building a practical foundation for a quantum program-

¹We use the word *practical* to contrast with *formal*.

ming paradigm on the one hand, and investigating the parallel performance of its simulated execution on the other.

1.1.2 Quantum Computing

[...] because nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy.

—Richard P. Feynman

Current computers are based on classical physics and deterministic by nature. However, the size of a single transistor inside a modern computer has become small enough that quantum effects come into play; unpredictable, counter-intuitive and in certain circumstances undesirable. The research field of Quantum Computing (QC) harnesses the strange behavior of elementary particles to perform computations. Such research effort is not a purely scientific curiosity, QC has two important foreseen applications; quantum simulation and computation.

Quantum computers for simulation can be built as analog computers, hard-wired to emulate a certain quantum system. In other words, using an artificial quantum system to emulate a naturally occurring one. It has been shown [76] that only a quantum mechanical computer can efficiently and accurately simulate a quantum system. A large market exists for such simulations, as engineers and scientists get ever more involved into the very smallest of scales. Quantum simulation computers are likely to be the first quantum computers to hit the market, judging from the volume and practical results of publications in this field.

The second application of quantum computing is of more interest to the computer scientist: using a quantum computer to compute certain problems that are either too hard or impractical on a classical computer. This aspect of QC has received a lot of exposure after the publication of Shor's prime factorization quantum algorithm [171]. It is important to note: it has not been shown that a quantum computer is inherently faster than a classical computer. However, there are major indications that quantum computers can solve certain problems at a lower computational complexity than classical computers. Nevertheless, possible applications are highly valuable, enough to justify the current serious investigation from the broad research community and investments from private institutions. It is thus not unreasonable to expect further technological breakthroughs and the discovery of new applications. We envision that quantum computers will see a staggered introduction; first as analog computers, hard-wired to solve specific problems, and later as programmable quantum co-processors or accelerators.

The former is happening now, fixed-function quantum mechanical computers are used in optimization problems [116] and quantum simulation [155].

1.1.3 Quantum Programming

Beware of bugs in the above code; I have only proved it correct, not tried it.

– Donald Knuth

Mathematics is the art of reducing any problem to linear algebra.

– William Stein

Quantum algorithms and applications are still mainly developed on paper, by virtue of a person’s expert knowledge. Low-level physical effects, as described by quantum mechanics, are used to describe quantum algorithms with only little conceptual abstraction. Most commonly, the semi-formal quantum circuit model is used to express quantum computations. The circuit model is a simple framework, organizing reversible and deterministic quantum operations as one would organize logic gates in an electronic circuit. The classical computing elements often required in these quantum algorithms are kept informal or implicit. Many quantum computing simulators and imperative quantum programming languages follow the circuit model, each typically reformulating and formalizing the circuit model along the way. These approaches thus keep to a low-level formulation of quantum computing and integrate quantum computation with the classical programming world at this low level of abstraction.

In contrast, other research efforts focus on formulating quantum computing using higher-level frameworks from mathematics and computer science: type theory [86], linear logic [89], lambda calculus [167], functional languages [166], category theory [1, 70], etc. Most often, their low-level execution is a secondary concern; these formal frameworks and languages search for new insights and provide formal tools to facilitate proofs.

Formal frameworks can be powerful tools to aid the discovery and development of quantum computing applications. These can be more useful in a virtual environment, where operations that are tedious when performed by hand are automated, in which the formal tools can be used to analyze and verify executable quantum programs and where applications can be developed interactively. Broadly speaking, there is a need for a Quantum Programming Paradigm: a collection of quantum programming tools, models and approaches. However, a programming paradigm is not constructed, but is grown over time by accumulating the contributions of a diverse expert community [175]. Pragmatically speaking,

there is first and foremost a need for a practical quantum programming framework: an environment in which to interactively write, test and analyze quantum programs. In other words, there is a need for a programming framework that combines a low-level execution environment with the higher level formal tools and techniques.

In the Quantum Computing domain, there already exist executable languages, automated formal models and a great many low-level quantum computing simulators. Indeed, many quantum computing implementations have been developed over the years, both high and low-level. However, there has been a lack of an overarching approach or quantum computer model, one that is used for both higher-level formal work as well as low-level execution environments. The aforementioned circuit model typically fills the role of quantum computing model for practical low-level implementations. Although, because of the semi-formal nature of quantum circuits, many implementations have to create their own specific formalization and architecture. This creates a moving target for practical high-level frameworks and for the low-level implementation environments themselves. In summary, there is the need for an overarching formal model with a practical virtual implementation.

1.1.4 Parallel Computing

In order to provide a practical quantum programming framework, the issue of quantum simulation performance needs to be addressed. There is an existing body of work on this subject, these traditional approaches use known or novel techniques to help improve the performance issues with simulating elementary quantum operations on a classical computer. However, the world of classical computing is currently being perturbed. Parallel computing is disrupting the current dominant sequential model of computing. We regard the current state of affairs as an opportunity to investigate a more fundamental parallel approach to quantum computing simulation.

After more than sixty years of sequential computing, programmers are now forced to produce parallel programs, especially in cases where performance is an issue. In the past, technological advances brought phenomenal improvements to successive generations of sequential microprocessors. These improvements were mainly attributed to the successful translation of *more transistors*, afforded by Moore's law, into *more performance*. Moore's law is still in effect and, although showing signs of slowing down, is still expected to continue for the next few decades [114]. However, several forces have conspired to prevent processor manufacturers to translate more transistors effectively into more sequential performance. This has caused a move in mainstream processors towards so-called multicore processors: placing two or more processor cores inside the same microprocessor. For

the processor hardware industry, this was a fairly small and innocuous change; after all, multiprocessor systems have been in use for decades, using concurrency primitives and operating system features to make use of the additional processor resources. However, for the programmer this represented a sea-change, a multicore revolution [178]. Sequential programs today no longer benefit from new hardware improvements, creating a sequential performance ceiling. Breaking through this performance ceiling means turning to parallel processing.

Under the current circumstances, parallel computing is poised to change the direction of future processor architectures and how we program them. The current multicore processor design is a transitory phase, produced by incremental changes to an originally sequential processor architecture. The multicore design ensures that existing sequential software still works, but also offers additional parallel resources for software that is programmed to make use of it. The parallel computing model seems at first glance to make only a minor addition to the sequential model: a handful of low-level concurrency primitives. However, these fundamentally break the sequential computing model, discarding properties that programmers rely on to build large software, properties such as determinism and local reasoning. Parallel programming research dealing with multicore today constructs new parallel computing models and abstractions on top of this multicore model.

However, the multicore design has some fundamental issues that prevent it from effectively scaling in number of processing cores for general-purpose tasks. As we will see, the sequential design at the heart of the multicore architecture relies for its performance on low communication latencies and a globally consistent view on data. As the size of a parallel system increases, it becomes imperative for parallel approaches to tolerate higher latencies and deal with a decentralized data organization. Other parallel hardware and software approaches have been proposed in the past. Even today, the need for more performance has created a market for niche or special purpose parallel computing hardware, such as accelerators (GPUs, Adapteva [153]) or heterogeneous processors (Cell B/E, APUs [82]). These parallel hardware approaches all require different programming approaches, different still from the multicore programming model, but in return offer a more scalable parallelism. In short, the current multicore era for mainstream processors is very likely a transitory phase; the multicore design keeps backwards compatibility with older sequential software, but has fundamental issues that prevent it from scaling properly without breaking its programming model. We take this as an opportunity to break away from the current programming model when investigating the problem of parallel quantum computing simulation, using a highly-parallel programming model that does promise scalability.

1.2 Approach

1.2.1 Quantum programming

For our quantum programming virtual framework, we base ourselves on the formal framework of the Measurement Calculus (MC) by Danos et al. [53]. The Measurement Calculus brings together several interesting properties that make it highly suited as a formal basis. It introduces a compositional and modular quantum program abstraction, around which we build a practical and automated framework in which to create and combine quantum programs. More importantly, the MC is built upon a small, but universal set of low-level operations with simple semantics that form a ‘quantum assembly language’, as it were.

We use this quantum assembly language to build a practical virtual model of a Quantum Computer: a *Quantum Virtual Machine*. The quantum program abstractions mentioned above are automatically translated to low-level virtual machine instructions. This virtual machine abstraction allows changes to the above quantum program abstraction without impacting the virtual machine implementation. In the other direction, this allows changes to the virtual machine implementation without affecting the quantum programming framework built on top of it. In other words, the virtual machine enables us to experiment with different quantum programming approaches and multiple virtual machine implementations; be it a sequential or parallel quantum simulation and, in principle, even a physical quantum computer implementation.

In the first part of our approach, we do not simply build an ad hoc executable Measurement Calculus. Rather, we take the opportunity to design an extendable programming framework around the formal model, by separating each logical abstraction layer to form a layered architecture. Our vision is for this quantum programming framework to form a pragmatic basis on which a quantum programming paradigm can grow. In our framework, quantum programs are executed by a virtual machine, using the MC’s low-level operations as instruction set. Naturally, the programmer does not directly express quantum programs at this low level of abstraction. The formal MC framework introduces a modular and composable quantum program abstraction, which we implement in our practical framework. To improve the ease of use, this framework includes a graphical editor application to compose quantum programs in a visual style. The automated composition of quantum programs and their compilation into the low-level virtual machine instructions enables users of our framework to compose relatively large MC quantum computations that would be too tedious or error prone to produce by hand. These quantum programs can be executed by the virtual machine, after they have been compiled into a sequence of elementary operations.

1.2.2 Parallel computing

The simulated execution of the low-level Measurement Calculus operations contains a large amount of inherent parallelism: these operations have a relatively simple linear algebra formulation with highly repetitive structures. However, these quantum operations cannot be translated into parallel computations trivially. Traditional parallel approaches lend themselves well to problems where the problem space can be split and acted on in separation, but this is not the case here; each low-level quantum operation of the virtual machine acts on the entire state space, performing only few operations for comparably large amounts of data communication. This indicates that a different, more data-oriented, approach is needed.

We have found that the dataflow [125, 62, 61] model of computation fits this parallel simulation problem very well. In a dataflow program, operations are structured as a graph, in which operations are nodes and values are carried over the arcs connecting the nodes. Dataflow execution follows the availability of data; an operation is executed when the data it depends on is available. This exposes the inherent parallelism in the calculation, as the exact sequence in which the operations execute is left open. Such dataflow models of computation were popular in parallel research during the 70s and 80s, where actual dataflow architecture processors were developed. It has been shown in the past [98] that typical linear algebra problems can be efficiently expressed as dataflow programs, there are thus strong indications that MC's low-level operations can be naturally expressed as dataflow computations.

Dataflow processor architectures were researched in the past, but interest for them waned as sequential processors started to dominate even the high-performance computing market. Dataflow still survives in different forms; for instance, closest to its original form, as dataflow analysis techniques in optimizing compilers and as the Out of Order execution cores in nearly all modern stock processors. The latter requires in-hardware algorithms to extract limited dataflow programs from sequential instructions, but these are limited by hardware complexity cost and by the sequential nature of the input programs. We argue that the current race for parallelism will bring parallel software and hardware closer to a highly-parallel fine-grained model of computation, as exemplified by the dataflow model. First, fine-grained parallel models expose more of a problem's potential for parallelism. Second, dataflow's data-driven execution is better equipped to handle the increased latencies and data sharing issues that arise in a large scale parallel system.

In the second part of our approach, we build a conceptual dataflow model of the quantum virtual machine presented during the first part of the approach. We start by expressing a quantum program as a coarse dataflow model, in which each low-level MC operation is an atomic operation. This coarse model is further

refined until we can present a fine-grained dataflow model, in which elementary arithmetic operations form the atomic operations. As part of the validation, we implement a parallelizing compiler that transforms an instruction sequence for the quantum virtual machine into a fine-grained parallel program. The resulting parallel program is profiled to demonstrate significant parallel speedup.

1.2.3 Structure

We split our approach in two main parts. First, we design and build a virtual quantum programming framework in Chapter 3, based on the formal Measurement Calculus which we present in Chapter 2. Secondly, we investigate the formulation of the low-level MC operations in the dataflow model in Chapter 5, which is validated in Chapter 6 using the combination of theoretical analysis of the conceptual model and the empirical analysis of a practical implementation. We present our findings, conclusions and future work in Chapter 7.

Chapter 2

Measurement-Based Quantum Computing

In this chapter we establish Measurement-Based Quantum Computing and the Measurement Calculus as formal foundation on which to build the practical quantum programming framework in the next chapter. We start with listing the properties required of a formal framework to serve as basis for such a practical quantum programming paradigm. Then, we bring the reader a brief overview of the field of quantum computing, after which we discuss the related work in the field of quantum programming languages, checking existing frameworks against the formulated requirements. The main body of the chapter presents the Measurement Calculus in detail with a focus on its low-level operations, as we will come back to their semantics in later chapters. Finally, at the end of the chapter we discuss in more depth how the Measurement Calculus satisfies the requirements we formulate.

2.1 Requirements for a Quantum Programming Paradigm

Our requirements are based on Bettelli et al. [24]’s requirements for a similar quantum programming architecture proposal;

- **Completeness:** it should be possible to implement at least all quantum algorithms that can be expressed in the circuit model.
- **Integration:** support a way to integrate, extend or combine quantum programs with existing expressive classical programming paradigms.
- **Separability:** quantum state and operations should be clearly separate from classical ones. Furthermore, classical computation necessary for the realization of quantum operations should be separate from any other supporting classical computation.
- **Expressivity:** support a formulation of quantum programs in a way that puts human understanding and readability first. In other words support a programming paradigm with its high level constructs, abstractions, modularization, etc.
- **Hardware independence:** provide a fixed language or interface with the underlying implementation, giving room to implementors to optimize and change the underlying implementation as well as set a fixed target for language designers when experimenting with language features.

Any conceptual framework exhibiting the above properties forms a firm foundation for a practical quantum programming framework. The term ‘practical’ is left ambiguous with intent. By it we mean simultaneously; Practical: being useful and easy to use for its user, the quantum programmer, to solve problems and express algorithms. Practical: accommodating to implementation of both the supporting framework (languages, tools) as well as the actual execution (virtual and physical quantum computation). The above five requirements mainly cover the former. To cover the latter interpretation, we introduce four additional requirements for the formal framework;

- **Low-Level:** each elementary operation has a direct physical realization. Although, abstracting away concerns such as errors, timings and the actual implementation choice.
- **Executable:** formulation of operational semantics exposes the execution process and the internal machinery required for implementation.

-
- **Scalable:** physical scalability arguably poses the biggest challenge to the physical realization of quantum computers. A QC framework tackling this issue stands a better chance at remaining relevant in the future.
 - **Parallel:** virtualisation will remain the main method of executing QC programs for the foreseeable future. Increasingly, performance means parallel performance, even on today's stock computers. A framework supporting better parallel simulation can thus respond faster and simulate larger programs within the same time span.

2.2 Quantum Computing

Quantum Computing and Information is a relative young and interdisciplinary research field, with sub-disciplines in theoretical and experimental physics, mathematics, engineering, philosophy, cryptography and of course computer science. The domain of Quantum Programming Languages (QPL) has grown into a sub-domain. It can be distinguished from the rest of the field by its focus on formal and closed systems in which to express quantum computation, detached from implementation concerns and typically using techniques developed in computer science. While QPL is the closest related domain to our work, we feel it helpful to include a brief and chronological overview of the field of QC, which will help to place the related work. For further reading on Quantum Computing and related topics, we refer to the excellent textbooks by Nielsen & Chuang [150] and Gruska [97].

2.2.1 Overview

The earliest discussions on quantum-mechanical computers date from the early 1980s. The conceptual idea for quantum computers surfaced in articles about the physical limitations of classical computer hardware [21, 77] and the efficient simulation of quantum-mechanical systems [76, 77]. Bennett and Brassard [22] opened up the field of Quantum Cryptography with their quantum key distribution algorithm, which a decade later will prove to be quantum information's first commercial application. The Deutsch-Josza algorithm [65] provided in the early 1990s the first demonstration of a problem in which a classical solution requires more steps than the quantum algorithm. A few years later, Peter Shor published a polynomial algorithm for the prime factorization problem [172, 171]. Whereas the Deutsch-Josza algorithm was more of theoretical interest, prime factorization is a very useful problem to solve and in classical computing has no known polynomial algorithm. Due to this discovery, Quantum Computing experienced a large surge in popularity, helped by the fact that prime factorization is the cornerstone of the popular RSA encryption protocol. Soon after, Grover's search algorithm [96] demonstrated another useful quantum algorithm with an improved run time compared to classical computing. Since then, several other quantum algorithms have joined the complexity zoo¹: quantum annealing [54], quantum walks [146], quantum cryptography [38], etc. Many of these algorithms have been developed within the complexity theory field, with similar theoretical development still under way.

The main tool of the trade in much of the theoretical development mentioned above is the circuit model. Most textbooks use the circuit model as quantum com-

¹http://qwiki.stanford.edu/index.php/Complexity_Zoo

putational model to represent and explain quantum algorithms. The quantum circuit model invokes the familiar model of an electronic circuit using gates and wires, making *intermediate* computations deterministic and thus easier to reason about. Circuit model examples can be found in Figure 2.1 in order to give an impression on the concepts and notation. On these circuit, the quantum states flow along the wires (horizontal lines) from gate to gate (boxes). Measurements are depicted using the \mathcal{A} symbol, destroying the quantum state and thus terminating the wire. These measurements, non-deterministic and non-intuitive, are performed at the end of the computation, as a way to extract the results from the computation. Often, measurements are not present in the description of a circuit, but are part of an implicit and informal classical control structure. A qubit in the circuit-model starts its life in a certain state, which over its lifetime evolves by interacting via a certain sequence of gates with multiple qubits.

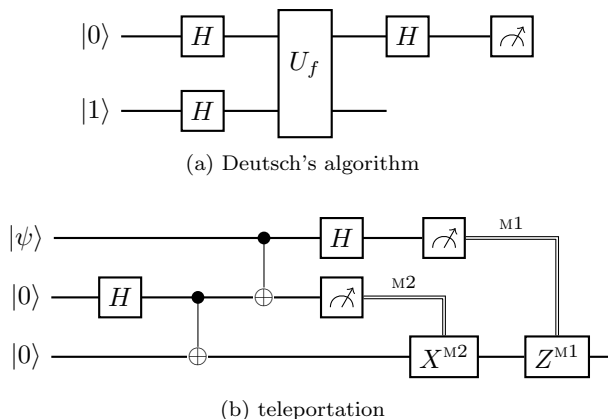


Figure 2.1: Some typical quantum circuit examples; the two-qubit special case of the Deutsch-Josza algorithm and the qubit teleportation circuit. The double line in the teleportation circuit carries classical data and thus denotes a measurement outcome control dependency.

In parallel with theoretical progress, experimental research has progressed as well. Experimental physicists have made important progress at the implementation side, producing physical demonstrations of known quantum algorithms such as Shor's factorization algorithm [194], implementing universal quantum gates [101] or working towards quantum-mechanical simulations [34]. Early experiments mainly strove to implement qubits, quantum gates and of course the then recently discovered theoretical algorithms. Many different approaches were explored: nuclear-magnetic resonance, ion traps, photonics, etc. Such experimental work demonstrated the basic principles, but also exposed serious scalab-

ility challenges. The search to overcome these challenges led to the development of new and radically different approaches to quantum computing such as adiabatic computing [25] and the one-way quantum computer [159]. Certain physical effects can produce larger and cheaper quantum resources, but the use of such resources requires quantum computational models that are fundamentally different from the circuit-based one. In summary, the insights gained from experimental research produced a new generation of quantum computing models. In such models, the commonly observed implementation pitfalls are avoided by construction.

The one-way quantum computer proposed by Raussendorf and Briegel [159] works in radically different ways than does a circuit-based one. The rationale behind the one-way computer is that a large amount of qubits with fixed entanglement patterns are cheap to create, but that it is expensive to keep a qubit alive for long due to *decoherence*. A qubit needs to be physically isolated to avoid losing the information it carries; each interaction with a qubit thus increases the chance of such decoherence. Measuring a qubit will evolve the state of other qubits it is entangled with and disentangle that qubit. After measurement, that qubit can thus be considered to be destroyed. This significantly reduces the lifetime and number of interactions on any one qubit, leading to a more scalable implementation. The one-way quantum computing model has since been demonstrated in experimental implementations [192, 183, 199].

Besides the physical realization benefits, the one-way computer has theoretical merits: it brings entanglement, measurement and measurement outcomes to the foreground. This stimulated theoretical research in what now falls under the name of Measurement-based [33] quantum computing (MQC). Measurement-based frameworks formed a popular basis for developing new theoretical tools and contributions: the Measurement Calculus [53], complexity analysis [36], new conceptual tools [55], circuit optimization [35], distributed QC [67], blind quantum computing [20], etc. Compared to the circuit model, the Measurement-based model is a radical paradigm shift; one-qubit measurements form the driving force of the computation and qubits are treated as disposable.

2.2.2 Quantum Programming Languages

The domain of Quantum Programming Languages(QPL) has grown into a sub-domain of Quantum Computing and Quantum Information. It can be distinguished from the rest of the field by its focus on more formal and closed systems in which to express quantum computations, detached from implementation concerns that typically use techniques developed in Computer Science. Simon Gay wrote an excellent survey of the QPL domain [85], categorizing the state of the art as imperative or functional languages, semantic techniques and compilers.

Rationale

A common criticism leveled against the domain of Quantum Programming Languages is that “*it is pointless to study languages for programming non-existent hardware*”. Gay refutes this criticism using three arguments. First, Quantum Computing has already produced practical and marketable results. Initially, using commercial fixed-function hardware such as random number generators and Quantum Cryptography solutions. Later, the company D-wave systems launched a commercial quantum computer² implementing a *quantum annealing* [116] processes that, although not a universal QC, addresses certain optimization and classification problems [54, 147]. Next, learning from the past, bottom-up programming paradigms developed for the early classical computers had certain avoidable issues that were only solved decades later with the introduction of languages with a better theoretical foundation; for example, early FORTRAN compared to Algol60 and Lisp. Co-developing QPL early on, using firm theoretical foundations can help avoid the issues observed in early purely hardware-oriented bottom-up programming paradigms. And lastly, the insights gained by the development of QPL, which are useful even in absence of practical QC.

In the context of our thesis, we add two additional arguments in Chapter 3 relating not to QPL as such, but to a working implementation thereof: supporting the development of new QC applications and interfacing with implementations.

Related work

The proverbial grandfather of QPL is Knill’s quantum pseudocode proposal [127] from 1996, operating on the *Quantum Random Access Machine* (QRAM) working model of a quantum computer. It is an early attempt to formalize the circuit model. The QRAM model mimics the conventional Random Access Machine; classical as well as quantum computational state stored in memory locations, a conventional imperative language manipulates the quantum locations using quantum gates. The QRAM model has formed the basis for most imperative quantum programming languages to date [152, 24, 64, 162, 112]. The first complete quantum programming language was Ömer’s *QCL* [152] in 2003, a procedural imperative language. SQRAM by Nagarajan et al. [145] describes and implements a low-level instruction set for a QRAM-based computer architecture. The focus of the imperative QPL lies on implementing and executing circuit-model quantum algorithms using existing programming tools. Some are even completely embedded in an existing language as a library [24, 64].

²The D-wave quantum annealing computer is not without controversy, mainly receiving criticism about its lack of definite proof that the quantum annealing process does not work strictly within classical mechanics. Recently, they did address some of the criticism by demonstrating quantum effects in the eight qubit case [116].

The imperative QRAM approaches suffer from the lack of formalization. For instance, comparing three QRAM-based languages [152, 24, 145] reveals a wide variety of basic operators and combinators, even if their approaches are fundamentally the same. By their nature, QRAM approaches construct circuits by using functionality of the classical host language. While this does provide some degree of integration and separability, these mesh together to the extent that it is hard to distinguish operations necessary for the larger algorithm from those that simply place a quantum gate within the circuit.

Another distinct camp in QPL are the functional languages. Selinger developed an influential functional language called *Quantum Programming Language* [166], incorporating a type system and two equivalent notations: textual and an ingenious graphical quantum flow charts. Another important functional language is QML by Grattage [94]. Functional languages typically have a different focus and purpose from their imperative counterparts, seeking to investigate unusual language features [94], higher-order abstractions or assist in proofs [70].

The Measurement Calculus by [53] provides the required formalization of a small but powerful low-level framework, a measurement-based *assembly language*, on which it builds higher-order concepts. The expressive power of the MC is evident from its use in the field as conceptual tool. It also has been advocated before as a basis for a bottom-up approach to a higher-level programming paradigm [66] in a distributed setting, for which we provided a virtual implementation [67] by extending our layered architecture covered later in Chapter 3. To this date, we know of only two other MC implementations besides our own. The calculus itself, a code rewriting process essentially, was implemented by D’Hondt [66] in the context of a PhD thesis. Independently from our own virtual implementation, Allcock [6] implemented as part of a master thesis an ad-hoc C++ implementation.

2.3 Operational Basics

The Measurement Calculus (MC) is a theoretical framework and a formalization of the measurement-based QC model. It defines a small and elegant core language with simple operational semantics. In addition, the MC also introduces a small calculus of equations and a pattern abstraction, supporting the design and analysis of MC programs. This pattern abstraction is presented in the next section. First, we present the operational principles of Measurement-based Quantum Computing (MQC), for which the MC's operational semantics provides us a realistic, powerful, low-level and universal instruction set. Such properties are important for the Quantum Virtual Machine, which we introduce in the next chapter.

2.3.1 Overview

The basic computational resource for MQC is a cluster state: a large and highly-entangled quantum state. In a cluster state, all qubits are prepared in a regularly-structured entanglement network, typically a mesh or 2D lattice. This structure describes the large quantum states that can be obtained by using certain physical effects, such as those described by the Ising interaction [49]. The conceptual generalization of such a cluster state is the *graph state*, in which nodes represent qubits and edges an entanglement relations. The MC starts its computation by forming a graph state using two-qubit entanglement operations. Computation then proceeds with a measuring phase, in which single-qubit measurement operations are applied to the graph state. These measured qubits are removed from the state, but the outcome of the measurement stored. In the third and final phase of execution, correction operations are applied when necessary to compensate for the non-deterministic effects of the measurement operations. The following MC formulation of the Hadamard transformation provides the smallest example which contains all three elements:

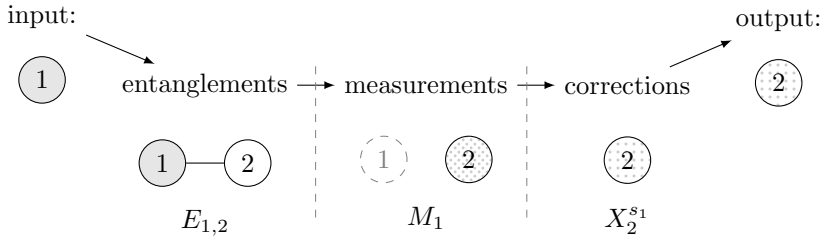
$$H := X_2^{s_1} M_1 E_{1,2} \tag{2.3.1}$$

where H is the operator

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.3.2}$$

In order, from right to left, we have an entanglement operation E , measurement M and correction X . The numbers in subscript are the target qubit names and s_1 denotes the measurement outcome of qubit 1. Before moving on to the detailed description of syntax and semantics, we show a rough schematic overview of the

effect of all three phases on the graph state. Therein, we visualize the graph state at each phase, using shaded circles for qubits and edges between them for entanglement.



Qubit 1 is taken as input state, combined with a fresh qubit 2 in the entanglement step and subsequently destroyed in the measurement phase. Qubit 2, which was modified as a side effect of the measurement, is modified by a correction operation to reach the desired output state.

We present the operational semantics of the Measurement Calculus from Danos et al. [53] by presenting the action of each operation upon the computational state in turn. Required notational conventions, linear algebra and QC fundamentals will be presented along the way. The Measurement Calculus offers three types of operations: *Entanglement* (E), *Measurement* (M) and *Correction* (X, Z). To distinguish between MC operations and the linear algebra operators more clearly, we will refer to MC operations as *Commands*.

2.3.2 Computation State and Preparation

Computational state is a pair (q, Γ) : a quantum state q and a classical state Γ . The classical part Γ is used to store the outcome of qubit measurements. The outcome of a measurement is either 0 or 1. Each qubit is destroyed after measurement and can thus be measured only once. It is thus sufficient to say that Γ maps *qubit names* to a Boolean value.

The classical state or *signal map* Γ fulfills two purposes. First, it stores measurement outcomes, which can be part of a quantum algorithm's result.³ However, the main purpose of the classical state is to control the execution of subsequent operations during computation, thus rendering an essentially non-deterministic computation deterministic.

Linear algebra is typically used for the mathematical formulation of quantum mechanics. The postulates of quantum mechanics tell us that the quantum state q is a vector in a specific type of vector space. More concretely q is a *unit*

³This is the case for Quantum Key Distribution protocols [73] that result in 0 and 1 outcomes that form encryption keys when concatenated.

vector in a *complex* vector space that has an *inner-product*. Its unique properties give rise to a geometric interpretation of a single qubit state. This geometric interpretation, called the *Bloch sphere* as in Figure 2.2, is a common intuitive aid that can be used to visualize a qubit state as a vector or point on the surface of a unit sphere. All non-input qubits needed by the MC computation are prepared in

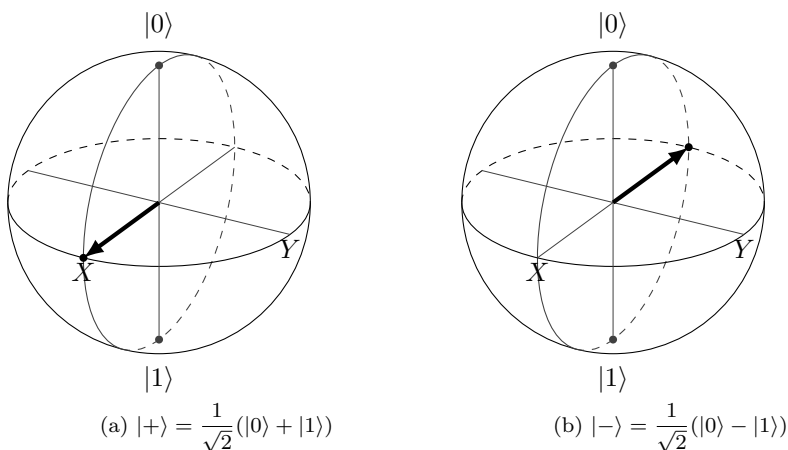


Figure 2.2: Bloch sphere representations of the diagonal basis vectors $|+\rangle$ in (a) and $|-\rangle$ in (b).

a so-called $|+\rangle$ state, which together with $|-\rangle$ forms the *diagonal basis*, as opposed to the *computational basis* formed by $|0\rangle$ and $|1\rangle$. The $|+\rangle$ and $|-\rangle$ states are in a superposition of the $|0\rangle$ and $|1\rangle$ basis states:

$$|+\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad , \quad |-\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} .$$

Both states, when visualized on the Bloch sphere, appear on the sphere's equator.

One of the quantum mechanical postulates state that composite systems are obtained by taking the tensor product of the separate systems. The computation's quantum state thus starts out as a product state of all qubits involved in the computation. Taking the Hadamard example above with the input qubit 1 in state $|\psi\rangle$, the computation would start out with $|\psi\rangle \otimes |+\rangle$ as computational state. To distinguish individual qubits, subscripts are often added to the notation. The example's initial state is thus often written as $|\psi\rangle_1 |+\rangle_2$. In general, the quantum

computational state q thus starts out as

$$q = |\psi\rangle \otimes \left(\bigotimes_{i \in Q} |+\rangle_i \right) \quad (2.3.3)$$

taking $|\psi\rangle$ combined input state and Q the set of all non-input qubit names used in an MC computation.

2.3.3 Entanglement: E

Qubits are said to be *entangled* when they cannot be represented as a product of individual qubit states. The MC introduces a special-purpose entanglement command E . This entanglement command targets two qubits, making it the only multi-qubit operation in the MC. The effect of the E command on a two-qubit state is described by the $\wedge Z$ operator, which has the following matrix representation:

$$\wedge Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

Its action is to apply the Pauli-Z operator $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ to the target (second) qubit when the control (first) qubit is set. For example, on the two-qubit initial state of the Hadamard example:

$$\begin{aligned} \wedge Z |\psi\rangle |+\rangle &= \wedge Z \frac{\alpha|00\rangle + \alpha|01\rangle + \beta|10\rangle + \beta|11\rangle}{\sqrt{2}} \\ &= \frac{\alpha|00\rangle + \alpha|01\rangle + \beta|10\rangle - \beta|11\rangle}{\sqrt{2}}, \end{aligned} \quad (2.3.4)$$

with $|\psi\rangle$ the single qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. As the quantum state is typically composed of more than two qubits, it is necessary to syntactically distinguish E commands that act on different qubits by writing $E_{i,j}$ where i and j are qubit names. Putting the target qubit names in subscript we see that for example $E_{2,3}$ acts on the target qubit 2 and control qubit 3. Due to linearity, when applying the operator to a larger state we obtain the following behavior

$$\wedge Z_{i,j} \sum \alpha_k |k\rangle = \sum \begin{cases} -\alpha_k |k\rangle & \text{if } |k\rangle = \dots |1\rangle_i \dots |1\rangle_j \dots \\ \alpha_k |k\rangle & \text{otherwise} \end{cases} \quad (2.3.5)$$

in which we use $|k\rangle = \dots |1\rangle_i \dots |1\rangle_j \dots$ to mean all basis states which have $|1\rangle$ at the positions of qubits i and j . Putting it all together, the action of the E command on the quantum state can be described as follows.

$$q, \Gamma \xrightarrow{E_{i,j}} \wedge Z_{i,j} q, \Gamma \quad (2.3.6)$$

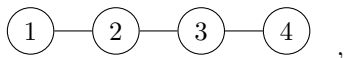
Some useful properties are that swapping target and control produces the same operation

$$E_{i,j} = E_{j,i} \quad (2.3.7)$$

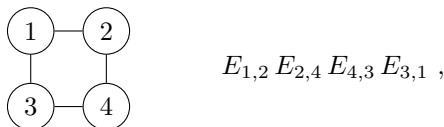
and that repeating an operation will cancel it out, as it is its own reverse

$$E_{i,j} E_{i,j} q = q . \quad (2.3.8)$$

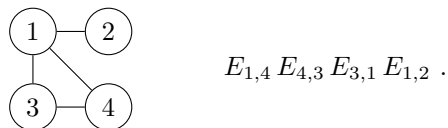
Entanglement operations can be chained together, such as for example in: $\wedge Z_{4,3} \wedge Z_{3,2} \wedge Z_{2,1} \bigotimes_{i \in \{1,2,3,4\}} q_i$. One can associate any graph with an entangled state, representing qubits as nodes and entanglement relations as edges. This associated state is a graph state. Reusing the above chaining example, the entanglement graph would be



which represents a different entangled state than



and different still from



In MQC terminology, the above describes a *graph state*. The lattice-shaped cluster states used by the original one-way quantum computer model are a special instance of graph states, for example the above case $E_{1,2} E_{2,4} E_{4,3} E_{3,1}$ forms a two-by-two cluster state. Early physical experiments use such cluster states [199], with some recent results indeed demonstrating that large-scale cluster states can be realized [34]. The graph state is a sensible abstraction, capturing essential qubit interactions. Graph states do have physical realizations, for instance by starting from a cluster state and disentangling certain qubits [104, 5].

2.3.4 Measurement: M

The measurement command M_i^α performs a single-qubit measurement with target qubit i . The *measurement angle* $\alpha \in [0, 2\pi)$ is a parameter that modifies the *measurement basis*. The angle α corresponds to the basis $\{|+\alpha\rangle, |-\alpha\rangle\}$ where

$$|+\alpha\rangle = \frac{1}{\sqrt{2}} (|0\rangle + e^{i\alpha}|1\rangle) \quad (2.3.9)$$

$$|-\alpha\rangle = \frac{1}{\sqrt{2}} (|0\rangle - e^{i\alpha}|1\rangle) . \quad (2.3.10)$$

without losing the universality property. The geometric interpretation of the above is evident when considering Euler's $e^{i\alpha} = \cos \alpha + i \sin \alpha$ describing a point on a circle in the complex plane. The measurement angle when omitted is assumed to be $\alpha = 0$, yielding the diagonal basis pair $\{|+\rangle, |-\rangle\}$ which is represented on the Bloch sphere in Figure 2.3a. Increasing α rotates this diagonal basis around the Bloch sphere's equator, within the XY -plane.

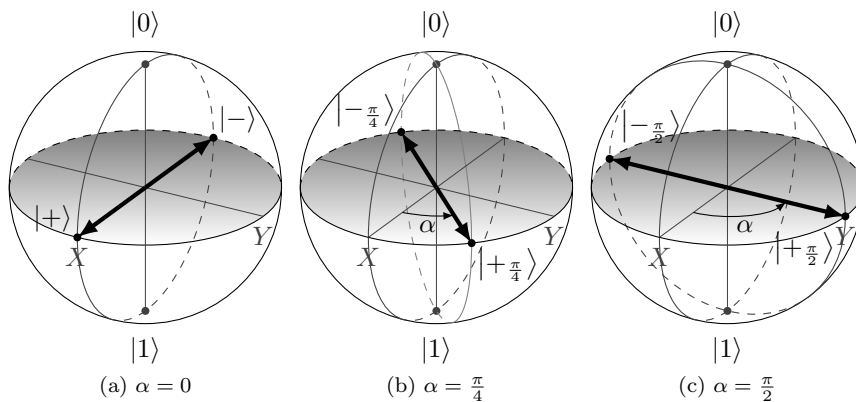


Figure 2.3: Bloch sphere representations of the $\{|+\alpha\rangle, |-\alpha\rangle\}$ measurement basis with some increasing values for angle parameter α : the diagonal or X basis $|\pm_0\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)$ in (a), an arbitrary basis $|\pm_\alpha\rangle$ in (b) and the Y basis $|\pm_{\pi/2}\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm i|1\rangle)$ in (c).

The measurement operation M_i^α will thus modify the quantum part q of the computational state such that the state of the qubit i is turned into either $|+\alpha\rangle$ or $|-\alpha\rangle$. This means qubit i 's state is now known and disentangled from any other qubits. The effect of M_i^α on the graph-state representation would be to remove all edges involving the node for qubit i . The quantum state of the computation after

measurement would thus be either the composed states $q' \otimes |+\alpha\rangle_i$ or $q'' \otimes |-\alpha\rangle_i$. Following the measurement-based approach, the state of qubit i is disregarded, removing the qubit's factor from the larger state to result in q' or q'' . To be more precise, using the inner product notation

$$q' = \langle +\alpha |_i q \quad (2.3.11)$$

$$q'' = \langle -\alpha |_i q . \quad (2.3.12)$$

The probabilities for either q' or q'' are calculated by comparing the norms of the pre- and post-measurement states. ⁴

$$p(0) = \frac{\|\langle +\alpha |_i q\|}{\|q\|} \quad (2.3.13)$$

$$p(1) = \frac{\|\langle -\alpha |_i q\|}{\|q\|} \quad (2.3.14)$$

Coming back to the Hadamard example, applying the measurement command M_1 on the post-entanglement state $\wedge Z |\psi\rangle_1 |+\rangle_2$ from Equation (2.3.4). The measurement command measuring in basis $\{|+\rangle, |-\rangle\}$ has a certain probability of applying either $\langle + |$ or $\langle - |$, resulting in the post-measurement state

$$\begin{aligned} & \langle + |_1 (\alpha|00\rangle + \alpha|01\rangle + \beta|10\rangle - \beta|11\rangle) \\ & = (\alpha + \beta) |0\rangle + (\alpha - \beta) |1\rangle \end{aligned} \quad (2.3.15)$$

or

$$\begin{aligned} & \langle - |_1 (\alpha|00\rangle + \alpha|01\rangle + \beta|10\rangle - \beta|11\rangle) \\ & = (\alpha - \beta) |0\rangle + (\alpha + \beta) |1\rangle \end{aligned} \quad (2.3.16)$$

respectively⁵.

The classical part Γ of the computational state is modified to store the *outcome* of the measurement. We use the convention that $s_i = 0$ if the outcome of measuring qubit i was basis $|+\alpha\rangle$. Otherwise, $s_i = 1$ if the qubit collapsed to $|-\alpha\rangle$. The command M_i^α modifies Γ to associate the qubit name i with the outcome. Dealing with such outcomes will be the domain of *signals*, a topic we will touch on shortly. In summary, the action on the quantum state of the measurement command can be either of two transitions

$$q, \Gamma \xrightarrow{M_i^\alpha} \langle +\alpha |_i q, \Gamma[0/i] \quad (2.3.17)$$

$$q, \Gamma \xrightarrow{M_i^\alpha} \langle -\alpha |_i q, \Gamma[1/i] . \quad (2.3.18)$$

in which we write $\Gamma[0/i]$ to mean the modification to Γ such that $\Gamma(i) = 0$.

⁴The $\|q\|$ in the norm is required because we do not normalize.

⁵As covered in the appendix, quantum states are equal up to a global phase. Not following normalization, we can leave out the $\frac{1}{\sqrt{2}}$ factors.

2.3.5 Corrections

By their very nature, measurements introduce non-determinism in the computation, transitioning probabilistically to either of two possible post-measurement states. Correction operations are typically applied after a measurement to merge these two computational branches. Taking the Hadamard example again, we can see that the $|+\rangle$ ($s_1 = 0$) measurement outcome produced the correct result by comparing Equation (2.3.15) to

$$\begin{aligned} H|\psi\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} (\alpha|0\rangle + \beta|1\rangle) \\ &= (\alpha + \beta)|0\rangle + (\alpha - \beta)|1\rangle . \end{aligned}$$

The same result can be obtained from the outcome $|-\rangle$ ($s_1 = 1$) branch from Equation (2.3.16) by applying the Pauli-X unitary operation $X := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

$$X((\alpha - \beta)|0\rangle + (\alpha + \beta)|1\rangle) = (\alpha + \beta)|0\rangle + (\alpha - \beta)|1\rangle .$$

Another correction operation used by the Measurement Calculus is the Pauli-Z unitary operator $Z := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$.

In summary, correction operations can turn inherently non-deterministic computation into a deterministic one by compensating for certain measurement outcomes. Their action on the quantum state is described by the transitions

$$q, \Gamma \xrightarrow{X_i} X_i q, \Gamma \tag{2.3.19}$$

$$q, \Gamma \xrightarrow{Z_i} Z_i q, \Gamma . \tag{2.3.20}$$

2.3.6 Signals

Signals are the mechanism used by the MC to control the conditional execution of commands. For example, the $X_2^{s_1}$ command applies the X_2 operation only if the signal s_1 evaluates to 1, doing nothing otherwise.

$$\begin{aligned} q, \{s_1 \rightarrow 0\} &\xrightarrow{X_2^{s_1}} q, \{s_1 \rightarrow 0\} \\ q, \{s_1 \rightarrow 1\} &\xrightarrow{X_2^{s_1}} X_2 q, \{s_1 \rightarrow 1\} \end{aligned}$$

The notation s_i evaluates to the outcome of qubit i in the given outcome map Γ . More formally:

$$\Gamma \vdash s_i \Downarrow \Gamma(i) . \tag{2.3.21}$$

Syntactically, the signal for corrections is specified in the exponent of the command, echoing the exponentiation of matrix operations $X_i^0 = I$ or $X_i^1 = X_i$. The

signal on a correction command is assumed to be 1 when omitted. Measurement commands can be equipped with two signals called s and t . These signals modify the measurement angle; the s -signal flips the sign of the angle and the t -signal adds π to the angle. This can be defined using the appropriate syntax as:

$${}^t[M_i^\alpha]^s := M_i^{(-1)^s\alpha+t\pi} \quad (2.3.22)$$

in which either signal s or t defaults to 0 when omitted.

A signal expression can contain a sum of multiple outcomes and values. Such sum is still evaluated to the Boolean value of 0 or 1 by summing the value of each term modulo two. For example $s = s_2 + s_3 + 1$ evaluates to 0 under $\Gamma = \{s_2 \rightarrow 0, s_3 \rightarrow 1\}$. Formally,

$$\frac{\Gamma \vdash s \Downarrow u \quad \Gamma \vdash t \Downarrow v}{\Gamma \vdash s + t \Downarrow u \oplus v} \quad (2.3.23)$$

where \oplus is sum modulo two or the logical XOR operation.

Evidently, a command equipped with a signal can only execute when the outcomes appearing in the signal are present in the signal map Γ . We call such constraints on the command execution order *signal dependencies*. Other such constraints are covered under the pattern abstraction in the next section.

2.3.7 Command Sequence

For the sake of clarity, we look at the action of each MC command as individual and atomic operations. However, an MQC computer ideally performs certain operations simultaneously. All entanglements at the front of the command sequence can be applied at the same time as part of the state preparation. Measurements that do not depend on one another and corrections to different qubits can also be applied simultaneously. But, the sequential view is semantically equivalent to this quantum parallel one, thanks to the linearity of the involved operators.

Putting all of the semantic rules together form the basis for the Measurement Calculus' operational semantics as presented by Danos et al. [53]:

$$q, \Gamma \xrightarrow{E_{i,j}} \wedge Z_{i,j} q, \Gamma \quad (2.3.24)$$

$$q, \Gamma \xrightarrow{X_i^{s\Gamma}} X_i^{s\Gamma} q, \Gamma \quad (2.3.25)$$

$$q, \Gamma \xrightarrow{Z_i^{s\Gamma}} Z_i^{s\Gamma} q, \Gamma \quad (2.3.26)$$

$$q, \Gamma \xrightarrow{{}^t[M_i^\alpha]^s} \langle +_{\alpha\Gamma} |_i q, \Gamma[0/i] \rangle \quad (2.3.27)$$

$$q, \Gamma \xrightarrow{{}^t[M_i^\alpha]^s} \langle -_{\alpha\Gamma} |_i q, \Gamma[1/i] \rangle \quad (2.3.28)$$

in which $\alpha_\Gamma := (-1)^s \alpha + t\pi$, and s_Γ refers to the value of signal s under Γ . These rules are applied to each command in a *command sequence* in right-to-left order.

2.4 Measurement Calculus

2.4.1 Patterns

In the example command sequence $X^{s_1}M_2E_{1,2}$, qubit 1 is implicitly used as an input qubit and qubit 2 as output qubit. For larger cases, it is useful to define what rule each qubit plays in the computation: input, output or temporary working qubits. This is the starting point of the following section, where we present the measurement pattern abstraction as defined by the MC.

A measurement pattern groups a command sequence with qubit sets. We take V to be the *computational space*, the set of all qubit names in the command sequence. The qubits in sets $I, O \subset V$ are respectively input and output qubits; qubits that are neither are called working qubits. As mentioned in the previous section, certain ordering constraints must be obeyed in the command sequence, these are distilled into the *definiteness conditions* [53].

Definition 1. A pattern $\mathcal{P} := (V, I, O, A)$ consists of the qubit sets V, I, O and a command sequence A obeying the four definiteness conditions:

- (D0) no command depends on an outcome not yet measured;
- (D1) no command acts on a qubit already measured;
- (D2) a qubit i is measured if and only if i is not an output.

The pattern for the Hadamard example above becomes:

$$\mathcal{H} := (\{1,2\}, \{1\}, \{2\}, X_2^{s_1}M_1E_{12})$$

Patterns are equivalent under a simple *qubit rewrite rule*: any qubit name can be substituted by another, if done so consistently. To refer to a pattern with specific names, the syntax $\mathcal{H}(3,4)$ is used for example as a shorthand for the pattern

$$\mathcal{H}(3,4) = (\{3,4\}, \{3\}, \{4\}, X_4^{s_3}M_3E_{34})$$

where the concrete qubits of V are explicit. Such rewriting becomes essential when combining patterns into larger computations.

Pattern combination is the merging of two patterns into a larger one. Two patterns $\mathcal{P}_1 := (V_1, I_1, O_1, A_1)$ and $\mathcal{P}_2 := (V_2, I_2, O_2, A_2)$ can be combined through one of two ways: composite and tensor combination. These are the pattern combinations as originally defined in Danos et al. [53]. When the qubit sets of both patterns are entirely disjoint,

$$V_1 \cap V_2 = \emptyset \tag{2.4.1}$$

we can perform a *tensor* or *parallel compose*:

$$\mathcal{P}_1 \otimes \mathcal{P}_2 := (V_1 \cup V_2, I_1 \cup I_2, O_1 \cup O_2, A_1 A_2) . \quad (2.4.2)$$

In other words, neither pattern interferes with the another and they can thus be merged trivially.

A *composite* or *sequential compose* is performed by connecting input with output qubits

$$\mathcal{P}_1 \circ \mathcal{P}_2 := (V_1 \cup V_2, I_1, O_2, A_1 A_2) \quad (2.4.3)$$

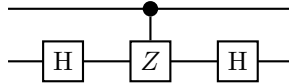
and can thus only be achieved on the condition that

$$V_1 \cap V_2 = O_1 = I_2 . \quad (2.4.4)$$

For example, connecting the input and output qubits of two Hadamard operations yields the composed pattern:

$$\mathcal{H}(2,3) \circ \mathcal{H}(1,2) = (\{1,2,3\}, \{1\}, \{3\}, X_3^{s_2} M_2 E_{23} X_2^{s_1} M_1 E_{12})$$

Note that the resulting pattern depends on qubit names, manual qubit renaming thus plays a part in composition. In the above example, qubit 2 is shared by both to connect input to output. In larger cases, the number of input and output qubits do not always match properly. For instance, the $\wedge \mathcal{X}$ pattern example from Danos et al. [53] implements the $\wedge X$ quantum gate based on the circuit:



by first introducing the two trivial patterns

$$\mathcal{I} := (\{1\}, \{1\}, \{1\}, \emptyset) \quad (2.4.5)$$

$$\wedge \mathcal{Z} := (\{1,2\}, \{1,2\}, \{1,2\}, E_{12}) \quad (2.4.6)$$

and combining composite and parallel composition with carefully selected qubit names:

$$\begin{aligned} \wedge \mathcal{X} &:= (\mathcal{I}(1) \otimes \mathcal{H}(3,4)) \circ \wedge \mathcal{Z}(1,3) \circ (\mathcal{I}(1) \otimes \mathcal{H}(2,3)) \\ &= (\{1,2,3,4\}, \{1,2\}, \{1,4\}, X_4^{s_3} M_3 E_{34} E_{13} X_3^{s_2} M_2 E_{23}) . \end{aligned} \quad (2.4.7)$$

The two pattern combination rules work well for small cases, but become a tedious and error-prone process on larger-scale pattern combinations. Later in Chapter 3 we introduce an alternative pattern combination rule in order to help automate this process.

Pattern combination is a simple but powerful abstraction as it supports the design of computations by enabling modularization and local reasoning. Because the pattern semantics is provably compositional [53], a component can be developed without having to take into account the entire system.

2.4.2 Rewrite rules

The MC defines a set of equations, a calculus, that acts as local rewrite rules, changing the ordering of a command sequence. The chief purpose of this calculus is to rewrite *wild patterns* into *standardized patterns*. A pattern in standard form performs first all entanglement operations, then all measurements and finally all corrections. This is an important and very useful element of the MC. For one, only standardized patterns can be executed by a physical measurement-based computer. However, in the context of virtual execution, the standardization of patterns can cause performance degradation. Putting all entanglement operations in front defeats some of the vital space optimization techniques for the simulation environment, which we cover in Section 3.5. Wild patterns typically destroy qubits right after introducing a new one, keeping the total number of entangled qubits down compared to the standardized pattern. In practice, we will thus keep patterns in their wild form due to performance reasons. Due to the importance of standardization for the MC, we do show their effect on a short example to impart some intuition. For further details, definitions, properties and proofs, we refer to Danos et al. [53].

Taking the wild pattern obtained by composition of the Hadamard pattern above, we can turn it into a *standard* pattern using the rewrite rules.

$$\mathcal{H}(2,3) \circ \mathcal{H}(1,2) = X_3^{s_2} M_2 E_{23} X_2^{s_1} M_1 E_{12} \quad E_{ij} X_i^s \Rightarrow X_i^s Z_j^s E_{ij} \quad (2.4.8)$$

$$= X_3^{s_2} M_2 X_2^{s_1} Z_3^{s_1} E_{23} M_1 E_{12} \quad E_{ij} A_k \Rightarrow A_k E_{ij} \quad (2.4.9)$$

$$= X_3^{s_2} M_2 X_2^{s_1} Z_3^{s_1} M_1 E_{23} E_{12} \quad M_i X_i^s \Rightarrow [M_i]^s \quad (2.4.10)$$

$$= X_3^{s_2} [M_2]^{s_1} Z_3^{s_1} M_1 E_{23} E_{12} \quad A_k Z_i \Rightarrow Z_i A_k \quad (2.4.11)$$

$$= X_3^{s_2} Z_3^{s_1} [M_2]^{s_1} M_1 E_{23} E_{12} \quad (2.4.12)$$

The commutation rules can be obtained and verified as matrix equalities. Other rules are somewhat more straightforward, such as that two commands operating on different target qubits and outcomes can be freely commuted, which can be seen in Equations (2.4.9) and (2.4.11). Correction commands can be absorbed in the s - or t -signal of measurement operations, which is the reason why these signals on measurements were introduced in the first place.

We stress again the importance of this standard *EMC-form*, the rewrite rules and the standardization process; Each pattern is guaranteed to have a unique standard form. The rewrite rules preserve the pattern's semantics, do not add signal dependencies, but can remove some. The standardization process is guaranteed to terminate. These patterns are more parallel from the quantum point of view, all entanglement operations can be performed at the start of the computation as part of the program preparation. Patterns in such form also have theoretical analytical benefits [37, 55, 36], but are also essential for promising a

scalable physical implementation [161].

2.5 Measurement Calculus Properties

At this point we can explain in more detail the arguments for and against our choice of the MC as quantum computational model, having discussed in the above the operational semantics and pattern abstraction. We separate *conceptual* from *practical* merits, matching them with the requirements presented at the start of the chapter. With the term *conceptual* we group the properties that are important intrinsic qualities for a quantum computing framework, regardless of how we intend to use it. With *practical*, we group properties that are relevant to our thesis topic and thus help in building a practical quantum programming paradigm.

2.5.1 Conceptual framework

The MC was conceived to be an *expressively powerful* and a *universal* conceptual framework for quantum computation, using conceptual and formal tools originally from the field of computer science. Its expressive power arises from the pattern abstraction, the composition and standardization of which adheres to the compositionality, locality and modularity properties. In addition to this expressiveness comes the improved computational power compared to the circuit model [36]. Explicit classical control and explicit non-local operations allow the MC to express certain distributed algorithms more elegantly, which can be observed in research results dealing with new algorithms such as computability [36], characterization [37], distributed computing [67] with in particular blind computing [20].

Universality means that the set of operations in the MC can be combined to express any quantum computation. This property is shown by defining a universal set. Analogously, in the context of classical computer circuits, the NAND logic gate implemented physically by a transistor forms a universal set for classical computing: any boolean function can be implemented using a combination of NAND gates. For quantum computing in the circuit model, arbitrary single-qubit unitary operations and the controlled-NOT ($\wedge X$) gate form a universal set. The universal set for the measurement-based model is shown [52] by the MC to be the following two measurement patterns:

$$\mathcal{J}(\alpha) := X_2^{s_1} M_1^{-\alpha} E_{12} \tag{2.5.1}$$

$$\wedge \mathcal{Z} := E_{12} \tag{2.5.2}$$

In other words, any conceivable quantum computation can be decomposed into a combination of these two measurement patterns.

2.5.2 Practical framework

The MC is rooted in the one-way computer model, which by construction provides a *scalable* framework for doing quantum computations. As discussed before, it does so by being designed to use an efficiently obtainable quantum resource [34, 173, 192]. Advances have also been made in making this one-way computer model more fault-tolerant [160].

The MC's command sequences form a *low-level* framework, each command has a direct physical realization. Naturally, each command is still abstracting away concerns such as errors, timing and the actual implementation. This low-level nature, small set of operations and universality make the MC an *assembly language* or '*Instruction Set Architecture*' for a virtual measurement-based quantum computer. The MC's *operational semantics* and its small instruction set nature of the command sequences offer a straightforward virtual implementation, in the form of a simple abstract machine or interpreter.

Measurement-based computations typically use more qubits than circuit-based ones, but the operations acting on the larger quantum state are simpler. Indeed, each command is realized by a diagonal matrix operation, which we will see in Chapter 5 plays a vital role in expressing the operations as *parallel computations*.

The MC possesses *explicit classical control* (signals) and an *explicit classical state* (outcome map). Both traits are important for a practical implementation, as it allows implementations to encapsulate the classical machinery required to execute the quantum program. This helps both the *integration* with existing programming languages as well as *separability*. As seen in our related work section, this is still often a perceived issue in practical QPL implementations. An example of the benefit of both properties can also be found in a distributed setting, in which a signal can contain outcomes communicated from a different machine. This last example is essential in the distributed measurement calculus, for which we developed a virtual implementation [67] within the layered architecture presented in the next chapter.

2.5.3 Matching the Requirements

In summary, we can distill from the above discussion all the features necessary as per our requirements towards building a Quantum Programming Paradigm. *Completeness*: is satisfied by the MC's universality proof. *Integration*: is aided by a clear instruction set and explicit classical machinery. *Separability*: quantum state is separate from classical state and explicit classical control in the MC's operational semantics. *Expressiveness*: can be claimed due to the expressive power of MC's pattern abstraction and a pattern algebra that is sound, compositional and context-free. *Hardware independence*: is achieved as the MC operations abstract away from how each is implemented. *Low-level*: MC's handful of opera-

tions map directly on physical operations for the one-way computer. *Operational semantics:* MC's semantics defined simple operational semantics with a small amount of state transformation steps. *Scalable:* cluster and graph states are used as quantum computation resource, which promises to be scale more easily, relatively speaking, in number of entangled qubits. *Parallel:* MC's simple elementary operations lead to a natural parallel computing realization of its virtual execution.

Chapter 3

Practical Foundation for a Quantum Programming Paradigm

This chapter continues on the formal basis established last chapter. Here, we work out a complete working proposal for a Quantum Programming Paradigm based around the measurement-based Quantum Computing model. The contents of this chapter is not a report on a Measurement Calculus implementation. Rather, we establish a architecture of multiple abstraction layers that conceptually follow the natural boundaries of the formal MC model. We do report on several implementations of these abstraction layers, occasionally going into details. Some of these implementation details resurface during Chapter 5 or Chapter 6, others specifics are offered in order to be precise. Important in this chapter are the various abstraction layers of the architecture, their interface and mutual interaction. After the introduction and an overview of the architecture, the structure of the chapter will follow various layers from applications down to the realization of quantum operators.

3.1 Introduction

Programming paradigms are collections of formal and informal methodologies, tools, abstractions, etc. to facilitate and support writing programs in a certain style. Each paradigm offers a different approach to programming a computer. An individual Quantum Programming Language is a vehicle or part of a Quantum Programming Paradigm (QPP). In the last chapter, we presented some of the common reasons for the pursuit of Quantum Programming Languages. For the case of a QPP, we can add two additional arguments: developing a Quantum Programming Paradigm supports development of quantum computing algorithms and applications, but can also help in the development of quantum computer hardware.

First, we regard application development support. Even when eventually practical quantum computers surface, it is likely they will remain an expensive computational resource, for a long while at least. A QPP can support the search, development and testing of both new and existing applications for quantum computers. Virtualisation is one obvious way to support this; it enables algorithms and applications to be developed for quantum computers not yet available or too expensive to operate, at least, if the working model and simulator are accurate enough. Another way to support QC development is by automating and formalizing the expression of quantum algorithms. Currently, quantum algorithms and applications are developed by virtue of the creativity, ingenuity and mathematical rigor of the creator. A programming paradigm seeks to facilitate much of this work, but always with the hope that a practical and well-developed QPP opens it up for other research domains and applications. The second argument for developing a QPP is to help the development of implementations. A practical QPL implementation can, when sufficiently hardware-independent, provide a common testing and benchmark system for implementations. Once quantum computers pass the current proof-of-concept and embryonic stage of development, it will become important to quantitatively compare and assess different physical implementation approaches. Today, this argument already applies to virtual implementations. For instance, our QPP prototype enables us in Chapter 6 to compare the two radically different virtual implementations of Chapters 3 and 5.

Approach Programming paradigms, as can be guessed from the definition above, are not designed and created as much as formed by accretion of new ideas, tools and technology over time. *Building* a programming paradigm is a misnomer, rather more fitting is the term *growing* a Quantum Programming Paradigm [175]. We currently see two main stakeholders in QC: designers and implementers. Designers are the mathematicians, physicists and computer scientists developing quantum computing algorithms and applications. Implement-

ors are the experimental physicists and engineers building the physical quantum computers. Also belonging to this latter group are the programmers building the languages, related tools as well as the virtual execution environment. For a Quantum Programming Paradigm (QPP) to be effective, it needs to integrate the concerns of both stakeholders.

For designers, expressiveness and tools are important. Algorithms and applications should be straightforward to express for an expert. Tools aid in the analysis, verification and transformation of programs; a good formal basis is required for the development of such tools, in order to represent, analyze and manipulate programs. Although programming paradigms often use abstraction layers to shield the programmers from underlying engineering issues, it is hard to develop an effective paradigm in isolation without regard for the supporting computer architecture, virtual or actual. Computer science has multiple examples of top-down designed paradigms that only got wide acceptance across the programming community after demonstrating a practical mapping to the reigning computer architecture: the von Neumann architecture computer or Random Access Machine model. Examples are functional programming [121], logic programming [200] and relational algebra [50].

Previous chapter, we presented the Measurement Calculus and argued that it possesses properties to satisfy both the designer and implementer stakeholders. Yet, an ad-hoc implementation of the MC would not fit the vision of growing a QPP. An ad-hoc implementation aims to make something work, but is not concerned with extensibility. Our strategy is to create a small kernel with all the necessary elements from top to bottom, but which is designed from the start to be extended and grown in multiple directions. To this purpose, we build our QPP around a quantum computer architecture built as a series of abstraction layers. Each abstraction layer hides underlying implementation concerns and focuses on other concerns. A layered architecture is a conversation between different stakeholders, allowing experts to focus on specific parts and problems without breaking or redesigning the entire architecture. Such an abstraction layer approach is ubiquitous in today's computer software architectures; an application running on a modern operating system will run through easily a dozen layers: a language layer, virtual machine, compiler, kernel, network layer, etc.

In this chapter we construct the practical foundations of a Quantum Programming Paradigm, based around the measurement-based model. We do not ourselves construct an entire set of methodologies and high-level programming abstractions. Rather, we focus on establishing a firm foundation in the form of a practical quantum computer architecture. Concretely, we design a layered architecture for a measurement-based quantum computer, with a complete and practical implementation of the multiple layers of abstraction, ranging from a virtual machine implementation to a graphical design tool.

Related Work Revisiting the related work in the domain of quantum programming languages, one can observe some practical language and architecture proposals that can be considered as a first step toward a QPP; for instance the more physically-oriented architecture proposal by Svore et al. [181] and the SGRAM model with its dual instruction set architecture proposal by [145]. These imperative low-level approaches directly or indirectly follow Knill’s QRAM circuit-based computational model. As mentioned before, these approaches suffer from a lack of formalization of QRAM and the circuit model; each implementation needs to first define its own set of low-level operations and then integrating them into a new or existing language. This still leaves open the opportunities to develop the conceptual framework and formal tools required to build a QPP. In this respect, the functional Quantum Programming Languages such as Grattage’s QML [94] and Selinger’s QPL [166] can be found to be the opposite. Functional QPLs are typically constructed with a strong focus on conceptual work, such as higher-level expression, type theory and category theory. Implementation, however, is a secondary concern. The functional QPL Closest to our requirements is Selinger’s QPL, which has described operational as well as denotational semantics, mapping the language to lower-level circuit-based gates. The Measurement Calculus, as we have seen in last chapter, offers a best of both worlds, combining a simple and practical low-level implementation side with a higher-level conceptual side. Additionally, the Measurement Calculus has been used, beyond its original research, to develop conceptual tools, forming what could be considered the start of a QPP. With the perceived issues in the circuit-based model of QC discussed last chapter, it is useful to provide an alternative quantum computing model based on the measurement-based model and the Measurement Calculus.

3.2 Architecture

A layered architecture is designed to separate different concerns into separate layers. The Measurement Calculus already satisfies the requirements for a Quantum Programming Language presented in Chapter 2: *completeness, integration, separation, expressiveness* and *hardware independence*. We have chosen the layers around the abstractions provided by the MC, so as to preserve the required properties. The structure of the proposed layered architecture is as follows.

- **Application layer:** *integration*. MC programs need to be integrated into classical programming languages in order to build useful applications. In the application layer, the QPP is connected to the programming environment at large. We envision and implement two ways in which this can happen: a graphical design tool for non-programmers and a pattern library extension for programmers.
- **Pattern layer:** *expressiveness*. The pattern layer provides pattern composition functionality, but also implements a pattern assembler to transform a given pattern into an concrete and executable command sequence.
- **Execution layer:** *completeness* and *separation*. Given a command sequence, the execution layer orchestrates the execution of its MC commands, taking care of the required classical computations.
- **Realization layer:** *hardware independence*. The realization layer virtually or physically executes an individual quantum operation.

Different aspects of the MC form the basis for the proposed layered architecture. Its operational semantics and command sequences are the basis for the execution layer. The pattern abstraction's qubit renaming and composition rules are embodied in the pattern layer, which automates this process while abstracting from the underlying semantics.

Each layer is developed and implemented separately, as a separate abstraction, library or even executable computer program. It is fundamental to the layered architecture design that each layer implementation can be changed, extended and even substituted without affecting any other layers. Our implementations of the different layers form a practical foundation on which to experiment with and extend the measurement-based quantum computing paradigm. In practice, our implementations of each layer have already seen several alternatives and extensions. For instance, we provide two parallel application layers: a pattern library and a graphical design tool. Both produce the same output for the pattern layer, but take a different approach. We also implemented two different execution layers: a sequential interpreter called `qvm` and a parallel compiler `mcc`. Although,

both use a different realization layer due to their radically different approaches. A distributed extension to several layers was implemented in the context of our work on the Distributed Measurement Calculus [67].

To summarize, we offer a visualization of the layered architecture structure in Figure 3.1. The structure of this chapter follows the structure of the layered architecture, going from application to realization layer. In each layer, we go in more detail for its related MC concepts and implemented functionality. The last section validates the layered architecture approach by revisiting the required properties in more detail.

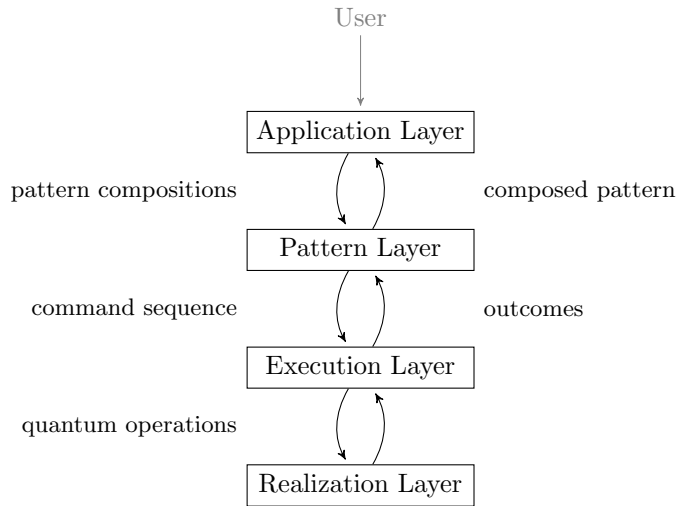


Figure 3.1: Overview of the layered architecture structure.

3.3 Application Layer

The application layer concerns itself with integrating basic quantum computations into larger concrete and useful applications. This abstraction level uses patterns as first-class entities, block boxes that can be passed around and manipulated only by certain composition operations provided by the pattern layer. The pattern layer underneath will perform the actual pattern creation, compositions and required transformations. We currently offer two different application layer approaches: the pattern library and graphical pattern editor applications. The former introduces pattern entities and operations into an existing programming language, highlighting the *integration* characteristic and its benefit in constructing complex patterns. The latter application we implemented is a Graphical User Interface (GUI) tool that exposes the pattern algebra as a small visual language, stressing *expressiveness*. The library approach requires programming expertise, but as we will see can be more powerful. The graphical approach allows non-programming experts to express and execute larger patterns without the tedium associated with doing the same manually.

The common example we will be using here is the construction of a pattern realizing the Quantum Fourier Transform (QFT), some familiarity of which is assumed during this discussion. The QFT is an interesting case; it is a well-known and often used quantum operation, requires a non-trivial composition of multiple measurement patterns and possesses a recursive structure: each n -qubit QFT can be defined in terms of an $n - 1$ -qubit version. We define the $QFT(n)$ pattern by following its circuit definition in Figure 3.2, which can be composed with Hadamard (H) and controlled-phase gates ($\wedge P(\alpha)$).

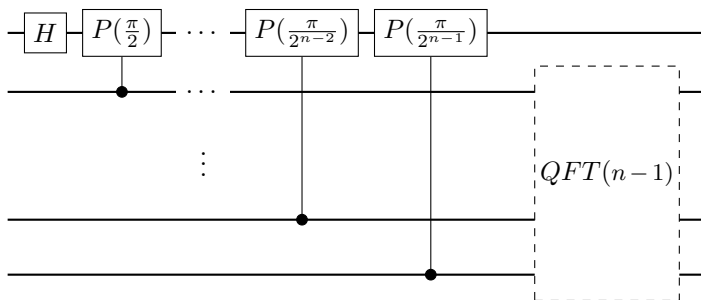


Figure 3.2: Recursive definition of $QFT(n)$: the Quantum Fourier Transform for n qubits using circuit notation.

The one-qubit phase gate

$$P(\alpha) := \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix} \tag{3.3.1}$$

performs a rotation of a qubit state by angle α over the Z -axis on the Bloch Sphere representation. To realize $P(\alpha)$ and its controlled version $\wedge P(\alpha)$, we decompose both into applications of the unitary operators $J(\alpha)$ and $\wedge Z$, for which we have already seen the measurement patterns $\mathcal{J}(\alpha)$ and $\wedge \mathcal{Z}$ in previous chapter. Any single-qubit unitary can be decomposed into a series of $J(\alpha)$ unitary operators, such that

$$P(\alpha) = J(0) J(\alpha) . \tag{3.3.2}$$

Similarly, any two-qubit controlled-unitary operator, such as $\wedge P(\alpha)$, can be decomposing in terms of $J(\alpha)$ and $\wedge Z$, using the general controlled-unitary decomposition from Danos et al. [52]. Further details on this decomposition and our realization of the $\wedge P(\alpha)$ can be found in the appendix. The decomposition after simplification can be found in Figure 3.3 in circuit notation for clarity. Composing a pattern realizing QFT thus consists of first composing $\wedge \mathcal{P}(\alpha)$ from $\mathcal{J}(\alpha)$ and $\wedge \mathcal{Z}$ patterns, then composing $QFT(n)$ from $\wedge \mathcal{P}(\alpha)$ and \mathcal{H} . We use these $\wedge \mathcal{P}$ and $QFT(n)$ patterns as cases for non-trivial pattern composition examples throughout this chapter.

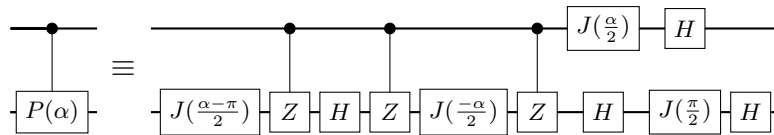


Figure 3.3: Circuit representation of $\wedge P(\alpha)$'s decomposition into $J(\alpha)$ and $\wedge Z$ gates, with $H = J(0)$.

3.3.1 Library Approach: First Class Patterns

The most straightforward way to integrate the components of the pattern layer in an existing *host* programming language is to provide the pattern functionality through a library. Central to this approach is to provide patterns as *first-class citizen* in the host programming language: patterns can be created, combined and stored and passed around as a regular program entity. This library approach enables the use of the *expressive* power of an existing programming language when creating and composing measurement patterns. It also facilitates the *integration* of quantum computations within an existing programming language.

Our prototype implementation integrates measurement patterns into the Common Lisp programming language. We use the realization of the $QFT(n)$ as working example using real code-fragment. This necessarily includes some programming and Common Lisp idiosyncrasies, for instance the use of alphabetic qubit names (a, b) instead of numeric ones (1,2). With the following code fragments, we wish to demonstrate how measurement patterns can be reified in practice within the application layer. Patterns entities are created either ex-nihilo or from combining two existing patterns.

Pattern definition A benefit of using Common Lisp is its syntactic macro system, which we make use of to extend the base language with a `defpattern` syntax to create new patterns. For instance, the $\mathcal{J}(\alpha)$ and $\wedge\mathcal{Z}$ patterns are defined in Listing 3.1. The arguments of `defpattern` are respectively the list

```
;;  $\mathcal{J}(\alpha) := (\{1,2\}, \{1\}, \{2\}, X_2^{s_1} M_1^{-\alpha} E_{12})$ 
(defpattern J (alpha) (a) (b) ()
  (X b (s a))
  (M a (- alpha))
  (E a b))

;;  $\wedge\mathcal{Z} := (\{1,2\}, \{1,2\}, \{1,2\}, E_{12})$ 
(defpattern CZ () (a b) (a b) ()
  (E a b))
```

Listing 3.1: Pattern library definition for the two patterns of the universal set: $\mathcal{J}(\alpha)$ and $\wedge\mathcal{Z}$.

of parameters and then the names for *input* (I), *output* (O) and *working qubits* ($V \setminus (I \cup O)$). Qubit names are treated as variables, strictly local to the pattern definition. The qubit sets are followed by the *command sequence* definition. This is specified using a Lisp-style rendering of the MC syntax, which is discussed in further detail for the execution layer below. The result of the first definition in Listing 3.1 is the creation of a new Lisp function called `J`. Invoking this function with e.g. `(J 0)` creates a pattern entity representing $\mathcal{J}(0)$. Similarly, a pattern entity representing $\wedge\mathcal{Z}$ is created by invoking `(CZ)`.

Pattern composition The two composition rules from the MC are realized by the functions `compose` and `tensor-compose`, named after the respective formal composition rules presented in Section 2.4.1. The `(compose p1 p2)` function can be seen to link the first output qubit of pattern `p1` to the first input qubit of `p2`, the second to the second and so on \dots ; a process that is defined in the

composite composition rule in Equation (2.4.3). As in the formal MC framework, both patterns need to have matching input and output qubit sets: $O_1 = I_2$. The function `(tensor-compose (list p1 p2 ...))` follows Equation (2.4.2), but can merge an entire list of patterns. Note that in both cases, unlike in the formal framework, no qubit explicit names need to be specified. This is a design choice; in practice, most patterns are chained together as with the above two compose functions. We will introduce a third composition function below to achieve more complex compositions.

The Z-axis rotation pattern

$$\mathcal{R}_z(\alpha) := \mathcal{H}(2,3) \circ \mathcal{J}(\alpha)(1,2)$$

can be defined as a regular Lisp function with the following code.

```
(defun Rz (alpha)
  (compose (J alpha)
           (H)))
```

Note that the composition functions in Lisp, e.g. the above `compose` function, combine patterns in left-to-right order, in contrast with the formal MC's right-to-left. A slightly more complex pattern composition, the pattern

$$\wedge \mathcal{X} := (\mathcal{I}(1) \otimes \mathcal{H}(3,4)) \circ \wedge \mathcal{Z}(1,3) \circ (\mathcal{I}(1) \otimes \mathcal{H}(2,3))$$

strings multiple composition functions together and is defined in code as:

```
(defun CNOT ()
  (compose (tensor-compose (list nil (H)))
           (compose (CZ)
                    (tensor-compose (list nil (H)))))) .
```

The `nil` stands in for the identity pattern \mathcal{I} , conforming more closely to Lisp's idiosyncrasies than would a `(I)`. The stringing together of compose functions, as in the above, occurs enough in practice to justify some convenience functionality. The `compose` function is extended to also accept as either argument a list of patterns, instead a single pattern. This list is passed to `tensor-compose`, creating a simple shorthand `(compose (list p1 p2) p3)` to mean `(compose (tensor-compose (list p1 p2)) p3)`. This first convenience simplifies the above example into the following.

```
(defun CNOT ()
  (compose (list nil (H))
           (compose (CZ)
                    (list nil (H))))))
```

In practice, some compositions have a long chain of such `compose` applications. The convenience function `compose-list` better captures the user's intent by taking a list of patterns and pairwise applying `compose`. This further simplifies the running example into:

```
(defun CNOT ()
  (compose-list (list nil (H))
                (CZ)
                (list nil (H))))
```

Combining these simple compose functions allows us to define the controlled-phase or $\wedge\mathcal{P}(\alpha)$ by following the circuit diagram in Figure 3.3.

```
(defun CP (angle)
  (compose-list
   (list nil (J (/ (- angle pi) 2)))
   (CZ)
   (list nil (H))
   (CZ)
   (list nil (J (- (/ angle 2))))
   (CZ)
   (list (compose (J (/ angle 2))
                  (H))
          (compose-list (H)
                        (J (/ pi 2))
                        (H)))))
```

The last composition functionality we introduce here aids in expressing compositions that combine a small pattern with a much larger one. This can be observed in the $QFT(n)$ circuit in Figure 3.2, where one or two-qubit gates are connected to the much larger $QFT(n-1)$. We introduce a `manual-compose` function that allows a more explicit handling of which input should be connected to what output. This function takes an additional argument, a list of *links*. A *link* takes two zero-based indexes, identifying a qubit name by position in the patterns' output and respectively input set; i.e. `(link 3 0)`, signifies that the fourth qubit output qubit of the first pattern should be connected to the first input qubit of the second pattern. Note that both patterns passed to `manual-compose` are not required to have the same input/output qubit set sizes, as was the case for `compose` and `tensor-compose`. Continuing with the $QFT(n)$ example, the pattern for $\wedge\mathcal{P}$ has a *control* and a *target* qubit. The pattern was constructed such that its first (index 0) input/output qubit is the control qubit and the second (index 1) the target. The two auxiliary functions `connect-H` and `connect-CP` defined in Listing 3.2 use `manual-compose` to compose the \mathcal{H} and $\wedge\mathcal{P}$ patterns to a given pattern by linking the correct qubits. Using two mutually recursive

```

(defun connect-H (pattern)
  (manual-compose (H) pattern
    (list (link 0 0))))

(defun connect-CP (pattern k)
  (let ((CP-pattern (CP (/ pi (expt 2 (- k 1)))))
    (manual-compose CP-pattern pattern
      (list (link 0 (- k 1))
        (link 1 0)))))

```

Listing 3.2: Two auxiliary functions used in the definition of QFT . `connect-H` connects a \mathcal{H} pattern entity's only qubit to the first qubit of the given pattern. `connect-CP` similarly links $\wedge\mathcal{P}$'s target qubit to the given pattern's first qubit and the control to the k 'th qubit.

functions, we obtain the following final definition for $QFT(n)$ in Listing 3.3. The k parameter is used to identify between individual controlled-phase gates in the circuit representation.

```

(defun QFT (n)
  (if (= n 1)
    (H)
    (recur-CP 1 n)))

(defun recur-CP (k n)
  (cond ((= k 1)
    (connect-H (recur-CP (+ k 1) n)))
    ((<= k n)
    (connect-CP (recur-CP (+ k 1) n)
      k))
    ((> k n)
    (tensor-compose (list nil (QFT (- n 1))))))

```

Listing 3.3: The QFT Lisp function creating of a $QFT(n)$ pattern entity by composing \mathcal{H} , $\wedge\mathcal{Z}$ and, recursively, $QFT(n-1)$ patterns. The stop condition is $QFT(1) = \mathcal{H}$

Execution Executing a measurement pattern means performing the low-level commands in the pattern's command sequence. The pattern entity in our library prototype has a specific internal representation to reify the formal measurement

patterns, a representation geared towards easier pattern combination rather than execution. It is also not the role of the application layer to implement the execution functionality, only to integrate and mediate with the pattern layer for composition and execution layer for execution. Therefore, the library prototype also includes the function `assemble` that turns a pattern entity into an executable form that complies with the execution layer's interface. This executable form has a simple plain text representation, allowing it to be simply saved to a file or sent to an execution layer implementation for execution.

3.3.2 Graphical Pattern Editor

Where the above library approach empowers a programmer, we offer a fundamentally different application layer approach for non-programmer experts by implementing a computer-aided pattern design tool. Its users can create and compose new patterns in a familiar graphical user interface environment. We use a visual boxes-and-lines language that captures the essentials to represent patterns and their composition constraints. While the visual notation evokes the circuit notation, the actual language and abstractions are different. The design tool exposes the same pattern layer to its users, but does it with different abstraction mechanisms than does the library approach.

To highlight some functionality of the design tool in its current state, we step through an example design process in which a user creates a 3-qubit QFT pattern. We provided a screen capture of each stage in Figure 3.4. New patterns can be added either ex-nihilo or by adding the current pattern composition in the *Pattern Composition Editor*, using the buttons at the top of the window. The agent pattern button is part of the functionality for distributed measurement patterns from D'Hondt and Vandriessche [67], not discussed here. The user is provided with some elementary patterns to start out with, which can be seen on the screen captures in Figure 3.4 as rectangular buttons under *Saved Patterns*. Clicking on such rectangular button will add a box to the *Pattern Composition Editor* pane representing a pattern instance. The circles left and right of the box represent respectively input and output qubits ports. A pattern composition is created by connecting these input and output ports, unconnected qubit ports become the input and output qubits of the new pattern. Note that for semantic reasons, covered in the pattern layer, no cycles can be created and qubit ports can only be used in one connection. One cannot connect a pattern's output to another pattern's input that already has a chained connection to the first pattern. The application will detect such cycles and refuse to proceed.

To realize the *QFT* pattern the user first constructs a Hadamard \mathcal{H} (a), Phase \mathcal{P} (b) and controlled-phase $\wedge\mathcal{P}$ (c) pattern. The \mathcal{H} pattern is realized by entering the angle parameter of the J pattern box (a), then saving this trivial composition

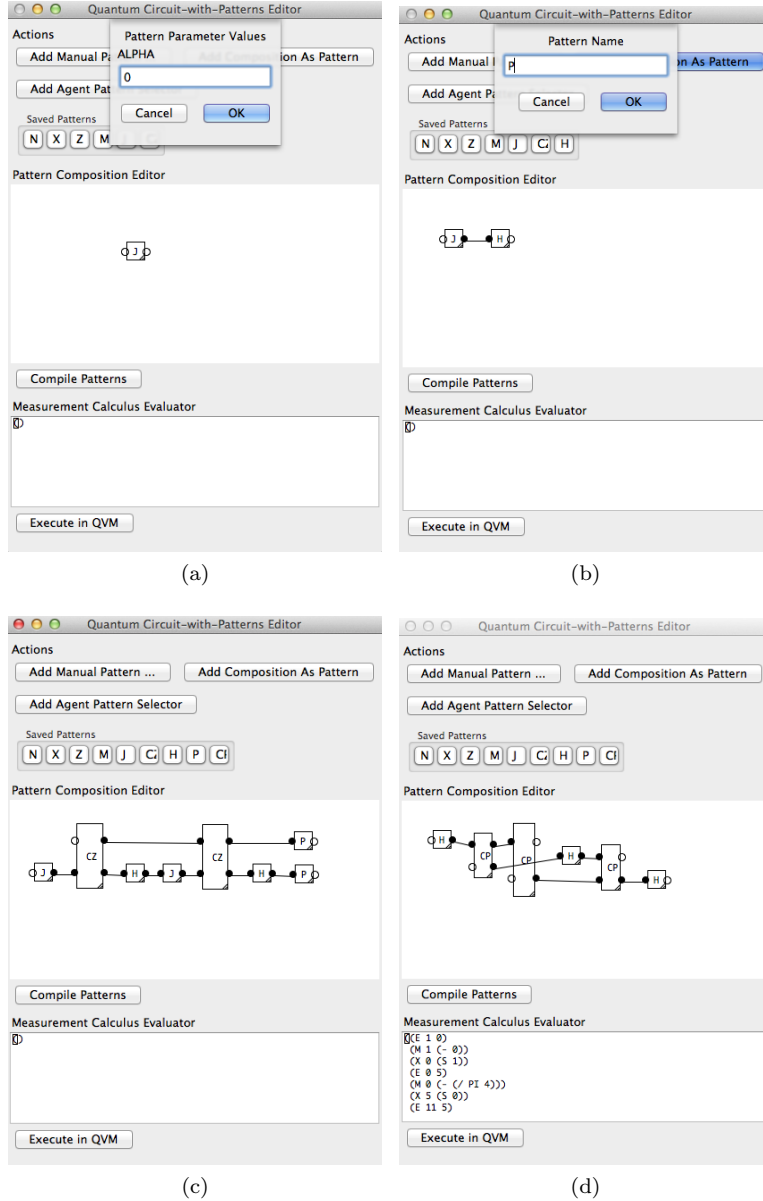


Figure 3.4: Step-by-step tool-assisted creation of a $QFT(3)$ pattern by definition of the Hadamard pattern \mathcal{H} in (a), phase pattern \mathcal{P} in (b), controlled-phase $\wedge\mathcal{P}(\alpha)$ in (c) and finally the modular composition thereof to obtain the $QFT(3)$ pattern in (d).

as the \mathcal{H} pattern. The phase pattern is composed by combining the \mathcal{J} and \mathcal{H} patterns, which is a matter of connecting J 's output to H 's input port (b). In the same fashion, we chain together \mathcal{J} , \mathcal{H} and \mathcal{P} patterns following Figure 3.3 to create the much larger controlled-phase or $\wedge\mathcal{P}$ pattern (c). Finally, we form the three-qubit QFT pattern (d) by connecting CP and H pattern boxes. This last example highlights the ease in which complex patterns can be expressed with the graphical pattern notation. However, it also highlights the expressive power and flexibility afforded by the library approach. Indeed, lacking recursion, the graphical notation cannot express the generalized $QFT(n)$, only build patterns for specific instances of n .

In summary, the design tool aids the design of measurement patterns in a user-friendly way, but does not add to the expressive power of the pattern layer. In contrast, the program library approach gives expert programmers more power by integrating the measurement pattern abstraction with an existing programming language. In the conclusion and future work chapter, we discuss several ways in which the design tool can be brought closer to the expressive power of the language integration approach.

3.4 Pattern Layer

The pattern layer concerns itself with the internal machinery required to compose and transform measurement patterns. Measurement patterns and their composition rules were already introduced by the MC as an abstraction mechanism, but require some changes in order to be automated. Performing and even defining larger pattern compositions by hand is a tedious and error-prone process. The MC can be a more useful conceptual tool in a context where the pattern algebra execution is not only reified and automated, but also expressed in a scalable way.

Conceptually, the pattern layer is also tasked to do pattern standardization. As shown in Chapter 2, the MC defines a process to transform wild patterns into a standardized form in which all entanglement operations happen first, then measurements and finally corrections. This process was already automated and implemented in [66]. Standardized patterns are a more efficient form for physical quantum computer implementations. However, as we show in the execution layer, a wild pattern (see Section 2.4.1) is typically more efficient in a virtual execution environment. We have therefore not implemented the standardization process in the current pattern environment. In the future work section we discuss this topic and suggest a standardization counterpart for a virtual environment. In principle, any pattern-level transformation and operation happens inside this pattern layer. Currently, the pattern layer implements two main operations: the *generalized composition* and *pattern assembly*.

The two rules for parallel and composite pattern composition in Equation (2.4.2) and (2.4.3) cannot be trivially automated as they stand. It is implicit that qubits get renamed in order to achieve the correct result and to satisfy the rule's preconditions. We replace qubit names by logic variables and replace the renaming process by variable matching. While this may seem more complex, it allows for easier automation of the composition rules and separates the way the composition is expressed from the underlying composition rules. Indeed, this separation is crucial to allow several alternative ways of expressing patterns and their composition, as was demonstrated in the application layer. This functionality is implemented by the *generalized composition*, which provides a consistent pattern definition and composition abstraction to the application layer above. Additionally, a *pattern assembly* process breaks down the pattern structures internal to the pattern layer into concrete command sequences, which can be passed to the execution layer for execution.

3.4.1 Defining generalized compositions

Before presenting the composition process, we first define some notation. The following pattern and composition structure notations are used to clearly sep-

arate the conceptual MC patterns \mathcal{P} from the internal structure $\bar{\mathcal{P}}$ used in the implementation.

Defining patterns We essentially replace the implicit renaming rule of the MC by logic variable names and variable matching, invoking some of the concepts used in logical programming languages. To distinguish clearly between both here, we will represent logical variables as letters prefixed by a question mark. For instance, the Hadamard pattern

$$\mathcal{H} := (\{1,2\},\{1\},\{2\},X_2^{s_1}M_1E_{1,2})$$

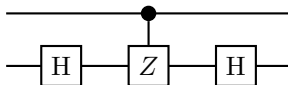
becomes

$$\bar{\mathcal{H}} := (\{?a,?b\},\{?a\},\{?b\},X_{?b}^{s_{?a}}M_{?a}E_{?a,?b}) .$$

A variable $?a$ stands for any possible qubit name, but only matching variable names within the same pattern definition take the same qubit name value. Variables are local to the pattern, meaning that a $?a$ in another pattern definition represents a different variable. It is evident that the use of variables is equivalent to and only subtly different from MC's concrete qubit name patterns with the free rewriting. However, it does lay the groundwork for a different and more general way of expressing pattern compositions to support an automatic renaming process.

Definition 2. A general pattern structure $\bar{\mathcal{P}}(?a, ?b, \dots)$ of a pattern \mathcal{P} represents the set of all possible $\mathcal{P}(a,b, \dots)$ with a,b, \dots distinct qubits.

Defining compositions We illustrate some of the inherent complexity in pattern composition by deconstructing a small but non-trivial example. Taking the $\wedge X$ gate example again, which in circuit notation is expressed



and formulated as pattern composition in [53] as

$$\wedge \mathcal{X} := (\mathcal{I}(1) \otimes \mathcal{H}(3,4)) \circ \wedge \mathcal{Z}(1,3) \circ (\mathcal{I}(1) \otimes \mathcal{H}(2,3)) . \quad (3.4.1)$$

The writer of such pattern composition has to take into account two main design constraints. The first is that the composite rule $(\mathcal{P}_1 \circ \mathcal{P}_2)$ rule requires both patterns to have the same number of output and input qubits respectively. The second design constraint is the choice of qubit names, all involved patterns need

new qubit names so as to string the correct input and output qubits together. Expressed more formally, the writer of the above composition is implicitly solving

$$\begin{aligned} & \{(\mathcal{I}(h) \otimes \mathcal{H}(f,g)) \circ \wedge \mathcal{Z}(d,e) \circ (\mathcal{I}(c) \otimes \mathcal{H}(a,b)) \\ & \text{choosing qubit names } a \dots h \\ & \text{such that } b = e, e = f, c = d, d = h\} \end{aligned} \tag{3.4.2}$$

by hand. For a human, this can, for large compositions, become a tedious and error-prone process. However, the process of choosing the correct qubit names is straightforward to automate for a computer, e.g. using simple constraint satisfaction. The constraint-based expression of a composition opens up a different way of composing patterns, one in which the patterns involved do not need to be of a same size. Such composition expression can avoid the now superfluous identity patterns and even forgo the two composition operations. Leaving the constraint satisfaction to an automated process, the pattern composition example can be expressed using *general pattern structures* as

$$\begin{aligned} & (\{ \bar{\mathcal{H}}(?a, ?b), \bar{\mathcal{CZ}}(?d, ?e), \bar{\mathcal{H}}(?f, ?g) \}, \\ & \{ (?b, ?e), (?e, ?f) \}), \end{aligned} \tag{3.4.3}$$

where the qubit variable pairs indicate equality constraints. The above composition structure has a natural graph visualization, shown in Figure 3.5, taking pattern structures as nodes and variable pairs as edges.

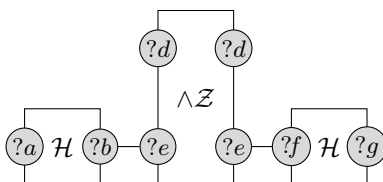


Figure 3.5: Graphical representation of Equation (3.4.3), the *general composition structure* expressing $\wedge \mathcal{X}$'s pattern composition

The following defines the structure used to express pattern compositions and will shortly be used in the automated composition process. We stress that this is not an operational expression as with the MC's composition operations \circ and \otimes . Rather, it is used to express the *intent* of a composition, to be used as input to the automated renaming process presented below.

Definition 3. A general composition structure (P, C) is a set P of general pattern structures \bar{P} and a set C of equality constraint pairs $(?o, ?i)$, in which $?o$ and

$?i$ are respectively output and input qubit variables of different patterns in P . Any $?o$ and $?i$ may only appear once as respectively first (output) element and second (input) element of any constraint pair $(?o,?i)$. Additionally, the graph constructed with the patterns from P as nodes and constraint pairs $(?o,?i)$ as edges must form a directed acyclic graph.

This structure together with the automated composition process subsumes the two MC composition rules. Both rules can be expressed as a general composition structure. Take in the following equations $\mathcal{P}_1 := (\bar{V}_1, \bar{I}_1, \bar{O}_1, \bar{A}_1)$ and $\mathcal{P}_2 := (\bar{V}_2, \bar{I}_2, \bar{O}_2, \bar{A}_2)$ to be the general pattern structure equivalents of patterns $\mathcal{P}_1 := (V_1, I_1, O_1, A_1)$ and $\mathcal{P}_2 := (V_2, I_2, O_2, A_2)$ respectively. Also take the functions $v_1 : V_1 \rightarrow \bar{V}_1$ and $v_2 : V_2 \rightarrow \bar{V}_2$ mapping a pattern's concrete qubit name to the respective general pattern structure's qubit variable, such that for instance $v_1(1) = ?a$.

$$\mathcal{P}_1 \otimes \mathcal{P}_2 = (\{\mathcal{P}_1, \mathcal{P}_2\}, \emptyset) \quad (3.4.4)$$

$$\mathcal{P}_2 \circ \mathcal{P}_1 = (\{\mathcal{P}_1, \mathcal{P}_2\}, \{(v_1(o), v_2(i)) \mid o \in O_1, i \in I_2, o = i\}) \quad (3.4.5)$$

With this generalized pattern and composition structure in place we can present the automatic renaming procedure.

3.4.2 Generalized Composition

The automated generalized composition takes a general composition structure (P, C) as input and ultimately produces a new general pattern structure \bar{P} as output. This composition process works in two stages. In a first stage, the *automated renaming process*, qubit variables of patterns in P are substituted with fresh qubit variables to create a new set of pattern structures P^* , such that the constraints in C are satisfied in P^* . In a second stage, all pattern structures in P^* are merged into a single the pattern structure \bar{P} using the *generalized merge process*.

$$(P, C) \xrightarrow{\text{renaming}} P^* \xrightarrow{\text{merge}} \bar{P}$$

The automated renaming process works by first generating a *substitution* map S . This map will be used to substitute qubit variables in each general pattern structure so as to satisfy the equality constraints. Starting empty $S = \emptyset$, the renaming process iterates over each constraint in the general composition structure's constraint set, adding new substitutions to S . When S contains a certain substitution $S[?a/?b]$, then $S(?b) = ?a$. To ensure that the construction of new bindings occurs in a well-defined and finite manner, elementary compositions in a composite structure are processed in topological order. To be precise, any

valid general composition structure can be, as discussed earlier, viewed as a graph. This graph is walked through in topological order, each edge a constraint pair that is applied to the rules below when walked over. Depending on the variables contained in the pair, one, two or no new substitutions are added to the substitution map S . The rules to add new substitutions per constraint pair are as follows.

$$\frac{S(?o) = S(?i) = \emptyset, ?f \text{ fresh}}{S \xrightarrow{(?o, ?i)} S[?f/?o][?f/?i]} \quad (3.4.6)$$

$$\frac{S(?o) = ?f, S(?i) = \emptyset}{S \xrightarrow{(?o, ?i)} S[?f/?i]} \quad (3.4.7)$$

$$\frac{S(?i) = ?f, S(?o) = \emptyset}{S \xrightarrow{(?o, ?i)} S[?f/?o]} \quad (3.4.8)$$

In other words, when neither names in the constraint pair appear in the substitution set S , rule (3.4.6) will trigger. A fresh qubit variable name $?f$ is chosen and added as substitution for both variable names in the pair. Rules (3.4.7) & (3.4.8) ensure that if a substitution already exists for one of the variable names in the pair, the other will use the same substitution. The topological ordering ensures that at all times only one of the three rules will execute; any variable may appear only once as first and once as second element in a constraint pair. After all edges have been visited by the walk, the substitution set S is complete. Finally, each pattern structure $(\bar{V}, \bar{I}, \bar{O}, \bar{A}) \in P$ of the input general composite structure (P, C) has its qubit variables substituted

$$(\bar{V}, \bar{I}, \bar{O}, \bar{A}) \xrightarrow{S} (\bar{V}', \bar{I}', \bar{O}', \bar{A}'), \quad (3.4.9)$$

such that

$$\forall ?a \in (\bar{V}, \bar{I}, \bar{O}, \bar{A}), ?b \in (\bar{V}', \bar{I}', \bar{O}', \bar{A}') : \begin{cases} ?b = ?a & \text{if } S(?a) = \emptyset \\ ?b = ?f & \text{if } S(?a) = ?f \end{cases} . \quad (3.4.10)$$

Applying the automated renaming process to the running example in Equation (3.4.3) will result in the expression

$$\begin{aligned} & (\{ \mathbb{H}(?a, ?b), \text{CZ}(?d, ?e), \mathbb{H}(?f, ?g) \}, \\ & \quad \{ (?b, ?e), (?e, ?f) \}) \\ & \quad \Downarrow \text{renaming} \\ & \{ \mathcal{H}(?a, ?z), \wedge \mathcal{Z}(?d, ?z), \mathcal{H}(?z, ?g) \} \end{aligned} \quad (3.4.11)$$

in which $?b$, $?e$ and $?f$ were substituted with a fresh $?z$. After this renaming process, all patterns are merged into a single pattern.

The generalized merge is applied pairwise to the set of all patterns P^* resulting from the renaming process. Command sequences are appended and qubit sets merged, similar but quite different from MC's composite rule. Essentially, all qubit variables that appear in both a pattern's output and input set are considered internal *working* qubits and removed from the new input and output sets. We describe this merge with a new merge rule that is more general but also more complex than MC's two composition rules. Concretely, we have the following new composition rule definition, where patterns are assumed to have already passed the renaming process.

Definition 4. *The generalized merge of general patterns structures*

$$(\bar{V}_1, \bar{I}_1, \bar{O}_1, \bar{A}_1) \quad \text{and} \quad (\bar{V}_2, \bar{I}_2, \bar{O}_2, \bar{A}_2)$$

is a new pattern structure

$$(\bar{V}_1 \cup \bar{V}_2, \bar{I}, \bar{O}, \bar{A}_2 \bar{A}_1)$$

where

$$\bar{I} = \bar{I}_1 \cup (\bar{I}_2 \setminus \bar{O}_1) \tag{3.4.12}$$

$$\bar{O} = (\bar{O}_1 \setminus \bar{I}_2) \cup \bar{O}_2 . \tag{3.4.13}$$

Indeed, qubit variables from the old input and output sets that were matched become auxiliary qubits and hence are only represented in the working set \bar{V} . Merging the running example from Equation (3.4.11) finally produces the desired general pattern structure

$$\{\mathcal{H}(?a, ?z), \wedge \mathcal{Z}(?d, ?z), \mathcal{H}(?z, ?g)\} \tag{3.4.14}$$

$$\Downarrow \text{ merge} \tag{3.4.15}$$

$$\begin{aligned} \wedge \bar{X} := \{ \{ ?a, ?z, ?d, ?g \}, \{ ?a, ?d \}, \{ ?g, ?d \}, \\ X_{?g}^{s?z} M_{?z} E_{?z, ?g} E_{?z, ?d} X_{?z}^{s?a} M_{?a} E_{?a, ?z} \} . \end{aligned} \tag{3.4.16}$$

We stress again that this pattern rule and the necessary renaming process are to be taken in the context of their automated execution. Pattern and composition structures are generated by the application layer, passed to the pattern layer for their automated composition, which in its turn returns a new pattern structure to the application layer. The graphical editor generates a composite structure by transforming the visible graph, the library approach offers composition operations that either implicitly specify constraints (`compose` and `tensor-compose`) or explicitly (`manual-compose`).

3.4.3 Pattern assembly

The general pattern structure is used as internal structure for the pattern layer. In order for a pattern represented as such to be executed, it needs to be assembled into a more directly executable form. The assembly step is very straightforward, it extracts the command sequence from pattern structure and chooses concrete qubit names for each qubit variable. The result of this simple assembly process is a concrete MC command sequence, without the pattern's qubit sets, that can be passed to the execution layer.

3.5 Execution Layer

The execution layer forms the border between the quantum and classical computing worlds. It concerns itself with orchestrating the execution of quantum operations and managing the quantum and classical states. In other words, it deals with implementing the classical execution part of MC's operational semantics. The execution layer itself does not perform any quantum computation, this is left to the realization layer. The execution layer takes care of the required classical control; storing measurement outcomes, computing signals and controlling the execution of individual commands.

The execution and realization layer are not always completely separable. Radically different realization layer implementations can require a different execution layer implementation. For instance, a virtual and physical realization of quantum operations necessarily require different classical programs to either simulate or drive the physical machinery. Although conceptually the interface between execution and realization layer cannot be fixed, we still make the distinction. Within a specific approach, a quantum operation can still be changed without affecting its interface or observable behavior. In practice, this refers to the tweaking, optimizing and debugging of quantum operations. We will call a specific execution and realization approach a *Quantum Virtual Machine*.

3.5.1 Machine model: vision of a Quantum Computer

Before presenting the implementation of a virtual measurement-based computer, is useful to establish a conceptual working model of such a quantum computer. The prevalent vision of a quantum computer in literature is one in the role of a co-processor [127, 145]. QC promises performance for certain classes of computation, classical computers will likely remain more effective in their current role for the foreseeable future. A quantum computer in the role of a co-processor enables quantum computation for the specific tasks it is good at, leaving much of the classical computing task to the controlling general-purpose processor. Special purpose co-processors are typically either completely controlled by a general-purpose central processor or by a limited embedded one.

In Chapter 2 we have established the use of the MC as a low-level 'assembly language'. The MC requires limited computational power to manage the classical state, computing signals and controlling command executions. We choose to embed this limited classical functionality within the quantum computer co-processor part of our conceptual working model. With such an approach, the precise inner workings and interface between the classical and quantum part of the quantum co-processor are encapsulated and abstracted away. The program input to the quantum processor can thus be a concrete MC command sequence.

This leaves the handling of measurement patterns and application integration to the general-purpose classical computer. We visualize our conceptual working model in Figure 3.6. In our layered architecture, the application and pattern layers are realized on the *classical computer* part. The execution layer represents the *quantum computer*, but can itself be completely classically implemented. This classical part of the quantum computer orchestrates the execution of the quantum operations, which are implemented by the realization layer.

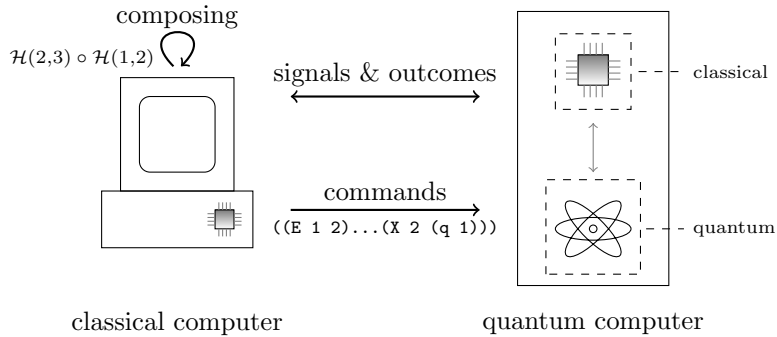


Figure 3.6: Schematic working model of a quantum co-processor.

3.5.2 The Quantum Virtual Machine

A virtual machine or VM is an implementation technique often used for high-level programming languages such as Smalltalk, Self, Java, C#, Lisp and Lua. The purpose of a VM is to alleviate a certain engineering problem: implementing a programming language, which can often change, on top of a variety of underlying computing platforms. In other words, a VM abstraction helps deal with changes to the abstraction layers both above and underneath. Traditional compilers directly translate a programming language to native processor instructions or a low-level language. Such direct translations to an execution platform require changes to the implementation for each change in both the top language and the bottom platform. A virtual machine meets the underlying platform half-way. Essentially an interpreter, a VM operates using a rather low-level set of instruction; low-level enough to be simple to implement on a real computing platform, but still general enough to still be applicable to a variety of platforms. Similarly, the VM's instruction set abstracts away enough low-level elements such that the programming language implementation on top takes less effort to (re-)implement than a machine-specific implementation. The VM abstraction fits our desired QPP approach: designing each layer with extensibility in mind.

The Measurement Calculus already suggests an obvious VM instruction set: the MC commands, as presented in Section 2.3. The commands already form a compact, complete and low-level set of operations, satisfying the typical VM instruction set requirements. While the QVM is essentially an executable Measurement Calculus, there are a few necessary differences with the formal model: abstraction, syntax and state implementation. As we have already seen above, the pattern abstraction is factored out and handled by a pattern layer. The QVM only accepts as input a concrete sequence of commands. The MC Syntax is turned into a machine-readable form. Rather than choosing a VM's traditional bytecode representation, we opt for the more flexible and human-readable s-expression syntax [140]. Both abstraction and syntax are part of the QVM interface, how the QVM implements its semantics does not have an impact on this interface.

Currently, we have three different QVM implementations. The first was a prototype built in the Common Lisp programming language, using QLisp [64] as realization layer. Much of the insights gained in its development were used towards developing a more optimized C implementation, which we refer to as `qvm`. This implementation step reuses parts of the a popular QC library for the realization layer, but its execution layer remains equivalent to the Lisp implementation. The third QVM implementation uses a parallelizing compilation approach and is the subject of Chapter 6. Here, we will deal with execution layer elements common to all our implementations: the interface and the managing of execution state and operations.

3.5.3 Interface and Syntax

Turning the mathematical notation of the MC into a machine-readable expression syntax is relatively straightforward. The most notable difference is the order in which commands in a sequence are applied. The MC follows the matrix application convention, which is right to left. For the machine-readable command sequence syntax, it is more traditional for commands in a list to be applied in left to right fashion. Thus, the command sequence

$$X_3^{s_2} Z_3^{s_1} [M_2]^{s_1} M_1 E_{23} E_{12}$$

becomes

```
((E 1 2) (E 2 3) (M 1 0) (M 2 0 (q 1)) (Z 3 (q 1)) (X 3 (q 2))) .
```

More formally, the new syntax is described (using BNF) as

$$\begin{aligned}
 \langle \text{sequence} \rangle & ::= (\{ \langle \text{command} \rangle \}) \\
 \langle \text{command} \rangle & ::= \langle \text{correction} \rangle \mid \langle \text{measurement} \rangle \mid \\
 & \quad \langle \text{entanglement} \rangle \\
 \langle \text{correction} \rangle & ::= (\mathbf{X} \langle \text{quref} \rangle [\langle \text{signal} \rangle]) \mid \\
 & \quad (\mathbf{Z} \langle \text{quref} \rangle [\langle \text{signal} \rangle]) \\
 \langle \text{measurement} \rangle & ::= (\mathbf{M} \langle \text{quref} \rangle \langle \text{angle} \rangle \quad (3.5.1) \\
 & \quad [\langle \text{s-signal} \rangle] [\langle \text{t-signal} \rangle]) \\
 \langle \text{entanglement} \rangle & ::= (\mathbf{E} \langle \text{quref} \rangle \langle \text{quref} \rangle) \\
 \langle \text{signal} \rangle & ::= 1 \mid \langle \text{outcome} \rangle \mid \\
 & \quad (+ \langle \text{signal} \rangle \{ \langle \text{signal} \rangle \}) \\
 \langle \text{outcome} \rangle & ::= (\mathbf{q} \langle \text{quref} \rangle)
 \end{aligned}$$

in which $\{ \}$ is used for repetition and $[]$ for option and qubit names $\langle \text{quref} \rangle$ are simple integers.

The interface of an external application with the QVM is exclusively using this syntax. We currently have not yet specified a application interface (API) for communicating the outcomes and other information such as final and intermediate quantum state. Typically, in the implementations of the QVM we provide, this information is offered to outside applications as simple data collections or output files. Examples of interactions with a QVM implementation can be found in Figure 3.6. The implementation used in the example is our fast sequential QVM implementation written in C, called `qvm`. It can be used from the command line as an interactive interpreter, as shown in the first example Figure 3.6 (a). The user is prompted on the lines with `qvm>` to input a single command which is directly executed, showing the resulting state and prompting for follow up commands. The second example (b) passes a file, generated by an application-layer program, containing a complete command sequence.

3.5.4 Execution

As mentioned before, the execution layer implementation often depends on the QVM approach. However, some basic elements are always required: the command sequence input in machine-readable expression syntax needs to be parsed, the classical and quantum states need to be managed, signals calculated and quantum operations controlled. Still, it is evident from MC's compact operational semantics that an execution layer implementation can be straightforward: an interpreter iterating front to back over each command in the given command sequence, evaluating the command's signal and, depending on the result, apply or skip the command. In the MC operational semantics, the entanglement and correction operators are trivial from an execution point of view: their respective operator is applied to the quantum state. The measurement command is slightly

```

soft85:qvm yvdriess$ ./qvm -i
Starting QVM in interactive mode.

qvm> (E 1 2)

qmem has 1 tangles:
  {[1, 2] ,
  {
    0.500000 +0.000000i|0> (2.500000e-01) (|00>)
    0.500000 +0.000000i|1> (2.500000e-01) (|01>)
    0.500000 +0.000000i|2> (2.500000e-01) (|10>)
    -0.500000 +0.000000i|3> (2.500000e-01) (|11>)
  }}
signal map: {
}

qvm> (M 1 pi/4)

qmem has 1 tangles:
  {[2] ,
  {
    0.146447 +0.353553i|0> (1.464466e-01) (|0>)
    0.853553 -0.353553i|1> (8.535534e-01) (|1>)
  }}
signal map: {
  1 -> 1,
}

qvm> (X 2 (+ 1 (q 1)))

qmem has 1 tangles:
  {[2] ,
  {
    0.146447 +0.353553i|0> (1.464466e-01) (|0>)
    0.853553 -0.353553i|1> (8.535534e-01) (|1>)
  }}
signal map: {
  1 -> 1,
}

qvm> (Z 2 (q 1))

qmem has 1 tangles:
  {[2] ,
  {
    0.146447 +0.353553i|0> (1.464466e-01) (|0>)
    -0.853553 +0.353553i|1> (8.535534e-01) (|1>)
  }}
signal map: {
  1 -> 1,
}

```

(a)

```

soft85:qvm yvdriess$ ./qvm cphase.mc
I have read:
((E 1 0) (M 1 (- -1.17809724))
 (X 0 (S 1)) (E 6 0) (E 0 7) (M 0 (- 0))
 (X 7 (S 0)) (E 6 7) (E 7 14)
 (M 7 (- -0.39269908)) (X 14 (S 7))
 (E 6 14) (E 14 21) (M 14 (- 0))
 (X 21 (S 14)) (E 21 26)
 (M 21 (- 1.57079632)) (X 26 (S 21))
 (E 26 31) (M 26 (- 0)) (X 31 (S 26))
 (E 6 36) (M 6 (- 0.39269908)) (X 36 (S 6))
 (E 36 41) (M 36 (- 0)) (X 41 (S 36)))

Resulting quantum memory is:
qmem has 1 tangles:
  {[31, 41] ,
   {
   0.500000 +0.000000i|2> (2.500000e-01) (|10>)
   0.353553 +0.353553i|3> (2.500000e-01) (|11>)
   0.500000 -0.000000i|0> (2.500000e-01) (|00>)
  -0.500000 +0.000000i|1> (2.500000e-01) (|01>)
  }}
signal map: {
  0 -> 1,
  1 -> 0,
  6 -> 0,
  7 -> 0,
  14 -> 1,
  21 -> 0,
  26 -> 1,
  36 -> 0,
}

```

(b)

Figure 3.6: Example command-line interactions with our QVM implementation written in the C language, called `qvm`. (a) is an artificial example showing step-by-step interaction with the user, (b) executes the controlled-phase pattern for $\alpha = \frac{\pi}{4}$ on the implicit $|++\rangle$ state.

more involved. It needs to calculate the measurement angle based on its two signals and add the measurement outcome to the outcome map Γ .

In practice, the need for optimizations and analysis adds additional complexity. Indeed, the parallelizing compiler implementation of the QVM covered in Chapter 6 builds several intermediate representations in order to find parallel workloads. We focus here on a state size optimization common to both our sequential and parallel QVM implementation. The adjectives *sequential* and *parallel* are used at the execution layer to differentiate between implementations issuing commands one by one or simultaneously. It does not matter at this point if the realization of the command's quantum operations is itself parallel or not. Indeed, it could well be possible to create a QVM with a sequential execution layer with a parallel realization layer and vice versa. So far, we have implementations of a completely sequential and a completely parallel QVM. The state optimization presented below is common to both practical QVM implementations, it introduces a smarter quantum state management that does not affect and is not affected by the realization layer below.

State optimization

An amplitude vector increases exponentially in size relative to the the number of entangled qubits in the quantum state it represents. This means that removing even a single qubit from an entangled state already halves the storage requirements. In practice, the entanglement graph of an MC computation is often not fully connected, meaning a qubit is not always entangled with all other qubits in a graph. This suggests an essential optimization for any virtual QC implementation: decomposing the quantum state in tensor factors when possible. Indeed, e.g. representing a 16-qubit quantum state q requires $2^{16} = 65536$ amplitudes. If that state can be decomposed as a product of two 8-qubit states $q = q_A \otimes q_B$, that factorizable state can be represented using only $2 \cdot 2^8 = 512$ stored amplitudes. Moreover, operators can be safely applied in parallel if they act on distinct factor states, something we will rely on in Chapter 5. Keeping track of factorizable quantum states is hardly new, it was already discussed by Knill in his quantum pseudo-code proposal [127] and was implemented by Ömer in QCL as Quantum Registers [152]. To our knowledge, this has not yet been applied to a measurement-based QC context. At first glance, it may seem that such size optimization would not benefit the measurement-based model, because of its use of highly entangled quantum states. However, the MC's semantics do not *require* patterns to be in the standardized form, in which all entanglements are performed at the start. Indeed, in practice, leaving patterns in a wild, un-standardized, form leaves room for state size optimizations.

In general finding tensor product factors of a given arbitrary quantum state is a hard problem [193]. We can however make a conservative approximation,

because of the MC's closed set of operations and the explicit entanglement operator. In essence, every qubit state is kept in a separate factor state or *tangle* (defined below), until it is entangled with an other. This simple analysis can be performed dynamically when applying each command, directly modifying the tangle's quantum state and qubit set to create, merge or shrink the tangle. The dynamic approach is taken by our QLisp and C implementations, who take an interpreter QVM approach. The same analysis can be performed statically, by checking the target qubits of all commands in the sequence before starting the actual execution. The latter static analysis is performed by the parallel compiler discussed in Chapter 5, which uses the information to incorporate tangle states in a graph.

We present one state size optimization here, based on the fly management of qubit state allocation. The first principle is to allocate qubits only when they first appear in any command, typically in an entanglement. Next, qubit state storage is shrunk after measurement as the qubit gets destroyed. Finally, the computational state no longer contains a single quantum state, but rather a collection of separately evolving states; each implementing a disjoint subset of qubit identifiers. We call the association of quantum state with a set of qubit identifiers a *tangle*.

Definition 5. A tangle T_Q is a pair (q, Q) where Q is the qubit set, a set of qubit names, and where $q \in \mathfrak{h}_Q$ is the quantum state of the qubits in Q , with \mathfrak{h}_Q the associated quantum state space over Q i.e. $\bigotimes_{i \in Q} \mathbb{C}^2$. We write $T_{i,j,\dots}$ for $T_{\{i,j,\dots\}}$, with the notational convention that the quantum state subscripts, when omitted, use the order in Q :

$$T_{i,j,\dots} = (\alpha_1 |0_i 0_j \dots\rangle + \dots + \alpha_{2^{|Q|}} |1_i 1_j \dots\rangle) .$$

As an example, take the two tangles

$$T_{1,3} = (|00\rangle + |11\rangle, \{1,3\}) \quad \text{and} \quad T_{4,2} = (|10\rangle + |01\rangle, \{4,2\}) .$$

The subscript convention is used to avoid ambiguities: does $|01\rangle$ mean $|0_2 1_4\rangle$ or $|0_4 1_2\rangle$? Tangles explicitly associate state with a qubit set, partitioning the quantum state space into a collection of disjoint state spaces requires such qubit sets to be explicit. To aid the discussion of tangle management, we introduce the *membership function* τ which maps a qubit name to the tangle it's being represented in:

$$\forall i : \tau(i) = \begin{cases} T_Q & \text{if } \exists T_Q : i \in Q \\ \emptyset & \text{otherwise} \end{cases} . \tag{3.5.2}$$

The changes to the operational semantics from Section 2.3.2 are relatively straightforward. First, the computational state is changed from (q, Γ) into

($\{T_Q, T'_Q, \dots\}, \Gamma$), i.e. the quantum state becomes a set of qubit tangles. Next, the initial quantum state is a singleton tangle $\{T_I\}$ containing all input qubits that we conservatively assume to be entangled. Then, each command is applied as before, but to the quantum state of the tangle $\tau(i)$ of its target qubit i , rather than the single global quantum state. In the case that $\tau(i) = \emptyset$, the command is applied as before to a fresh tangle $(|+\rangle, \{\mathbf{i}\})$. The correction commands X_i and Z_i are applied as before, as is the entanglement $E_{i,j}$ when $\tau(i) = \tau(j)$. For the case that $\tau(i) \neq \tau(j)$, we amend the operational semantics with the following rule:

$$(q_i, Q_i), (q_j, Q_j) \xrightarrow{E_{\mathbf{i}\mathbf{j}}} (\wedge Z_{\mathbf{i}\mathbf{j}}(q_i \otimes q_j), Q_i \cup Q_j) . \quad (3.5.3)$$

Measurement is as before, although the target qubit is removed from the tangle:

$$(q, Q) \xrightarrow{M_{\mathbf{i}}^\alpha} (\langle +_\alpha |_{\mathbf{i}} q, Q \setminus [\mathbf{i}]) . \quad (3.5.4)$$

As is apparent from the above amendments to the semantics, the individual operators and signal map manipulations are left as they were. Although the changes are only small, this simple state management technique enables the virtual execution of much larger measurement patterns. The storage requirement is still exponential, but it is exponential in the number of qubits in the largest tangle, not in the total number of qubits appearing in the measurement pattern.

3.6 Realization layer

The realization layer concerns itself with the evolution of the quantum state itself: the quantum operators. As with the execution layer, each implementation can have widely different approaches, but the basic and naive approach is very straightforward. Any extra complexity again arises from optimizations. The next two chapters will discuss parallel realization of quantum operators in detail, we will therefore keep to an overview of sequential optimization approaches. Many of the specifics overlap, we will thus provide an overview rather than a detailed report on the implementation.

We first present a simple but naive approach to the realization layer that directly implements linear algebra semantics, demonstrating that the realization layer can be compact but also quite wasteful. We cover a more optimized approach next; the MC's closed set of operations presents an optimization opportunity, arising from each operation's properties. This optimized approach was taken by the realization layer of the `qvm` implementation, using the C programming language with parts of the `libquantum`¹ library. Parts of the optimization techniques used are common even among circuit-based QC simulators, such as `libquantum`, `QCL` [152] or `QLisp` [64]. The purpose of our `qvm` implementation is to provide a reference implementation with good sequential performance. Later in this dissertation, the `qvm` implementation will serve as a benchmark baseline.

3.6.1 Naive Linear Algebra approach

The minimal functionality of a realization layer is the realization of each of the operators found in the MC's operational semantics: $\wedge Z$, X , Z and the measurement projection $\langle \pm_\alpha |$. A straightforward and common approach is to implement these operators directly by their linear algebra formulation, i.e. matrix operators acting on the state vector. Many programming languages include linear algebra extensions, such as the de facto standard BLAS, that can be used for this purpose. Recall that a single-qubit operator U_i is really a notational shorthand for some $I^m \otimes U \otimes I^n$, a direct linear algebra approach would thus have to create exponentially large matrix operators that match the state vector in size. Removing this wasteful behavior automatically brings us to the operator optimizations presented below. A less naive linear algebra approach exploits the regular block structure of the Kronecker product \otimes , the tensor product special case for vectors and matrices. This regular structure is further discussed in Chapter 5. These linear algebra approaches are often present in circuit-based simulators, even in optimized implementations employ it as a fallback to implement more general unitary

¹This quantum simulation library is commonly used in QC research and was included in the prominent SPEC benchmarks. More information can be found on <http://www.libquantum.de>

gate operations. Because the MC only requires a compact set of operators, it is worth examining each in detail and providing optimized implementations.

3.6.2 Correction and Entangle Operator optimization

Closer inspection of the basic MC operators in Section 2.3 does indeed reveal a relatively simple structure. All operators involved are linear operators with a simple diagonal transformation matrix. As a consequence, the X_i , Z_i and $\wedge Z_{i,j}$ operators can be implemented as a single *in-place iterative loop* over the state vector. To provide a compact and exact formulation, we will use code fragments written in the C programming language and thus assume some passing familiarity with C and some of its bit-level operations.

Recall the $\wedge Z$ semantics in Equation (2.3.5), from Section 2.3:

$$\wedge Z_{i,j} \sum \alpha_k |k\rangle = \sum \begin{cases} -\alpha_k |k\rangle & \text{if } |k\rangle = \dots |1\rangle_i \dots |1\rangle_j \dots \\ \alpha_k |k\rangle & \text{otherwise} \end{cases} . \quad (3.6.1)$$

This translates directly into the C subroutine

```
for (int k=0; k<size; k++) {
    // is |k> = ...|1>_i ...|1>_j ...?
    if ( (k & bitmask_i_j) == bitmask_i_j )
        state_vector[k] *= -1;
}
```

in which `state_vector` is an array of amplitudes with the convention that amplitude α_i resides at array index i . `bitmask_i_j` is the bit-level representation of $\dots |1\rangle_i \dots |1\rangle_j \dots$ with 0 bits in positions other than i and j . The technique to work with the binary representation of the qubit basis vectors is nearly ubiquitous in QC simulators, we use similar bit-level shortcuts in Chapter 6. The Z_i and X_i operators each have a similar efficient, iterative and in-place realizations, as with the above subroutine. Although with the obvious difference that their `bitmask_i` matches $\dots |1\rangle_i \dots$. To be complete, the semantics of

$$Z_i \sum \alpha_k |k\rangle = \sum \begin{cases} -\alpha_k |k\rangle & \text{if } |k\rangle = \dots |1\rangle_i \dots \\ \alpha_k |k\rangle & \text{otherwise} \end{cases} \quad (3.6.2)$$

and

$$X_i \sum \alpha_k |k\rangle = \sum \begin{cases} \alpha_k |\dots 1_i \dots\rangle & \text{if } |k\rangle = \dots |0\rangle_i \dots \\ \alpha_k |\dots 0_i \dots\rangle & \text{otherwise} \end{cases} \quad (3.6.3)$$

are respectively implemented as

```

for(int k=0; k<size; k++) {
    // is  $|k\rangle = \dots|1\rangle_i \dots$  ?
    if( (k & bitmask_i) == bitmask_i )
        state_vector[k] *= -1;
}
and
for(int k=0; k<size; k++) {
    // is  $|k\rangle = \dots|1\rangle_i \dots$  ?
    if( (k & bitmask_i) == bitmask_i )
        state_vector[k] = state_vector[k ^ bitmask_i];
} .

```

To summarize, each non-measurement MC operator can be realized by iterating over each amplitude and performing some manipulation depending on the associated basis vector.

3.6.3 Measurements

Measurement outcomes

Measurement operations are inherently probabilistic. In nature, this naturally happens automatically and obtaining the outcome is something that happens after the facts. In a virtual execution environment, the outcome of the measurement needs to be determined before applying the measurement operation. In virtual QC implementations, we have observed three basic ways of choosing a non-deterministic outcome.

- *Weighted Probabilistic*: calculate the probability for all possible measurement outcomes and select one based on a weighted (pseudo-)random probability.
- *Coin Toss*: select a measurement outcome by (pseudo-)random coin toss with *equal* probability.
- *Fixed*: determine measurement outcomes beforehand, either by always choosing the same outcome or by using a predetermined outcome map.

The probabilistic method emulates nature by emulating the expected probability for each outcome, probabilities that are calculated based on the pre-measurement state. While accurate this method comes with a performance penalty, as the probability needs to be calculated. In practice, this means another iteration pass over the state vector. Circuit-based QC relies on these probabilities, but MC patterns deal differently with measurements. Deterministic measurement

patterns rely on correction operations to obtain the same result regardless of the outcome. This opens the second and third option from the above list, both to avoid the performance hit of calculating probabilities. The coin toss option simply chooses an outcome based on a regular 50/50 coin toss. The fixed option determines the outcomes beforehand. Our `qvm` implementation can be configured to use either the fixed or probabilistic measurement outcome strategy.

Measurement operator

We present here an iterative algorithm for implementing the $\langle +_\alpha |$ or $\langle -_\alpha |$ actions on a quantum state. First, we need to introduce some notation to concisely express the required binary numeral manipulation. Recall that the index k of each basis vector $|k\rangle$ has a binary numeral representation of its index. We introduce the notation $b_1 \cdot b_0$ to concatenate binary numerals, such that

$$\begin{aligned} |k\rangle &= |k_{n-1}\rangle |k_{n-2}\rangle \dots |k_1\rangle |k_0\rangle \\ k &= k_{n-1}2^{n-1} + k_{n-2}2^{n-2} + \dots + k_12^1 + k_02^0 \\ &= k_{n-1} \cdot k_{n-2} \dots k_1 \cdot k_0 \end{aligned}$$

and taking

$$\begin{aligned} k &= k_{n-1} \dots k_j \dots k_0 = l \cdot k_j \cdot r \\ \bar{k} &= l \cdot r \quad . \end{aligned}$$

The action of $\langle +_\alpha |_j$ on an individual basis state can then be shown to be

$$\langle +_\alpha |_j |k\rangle = \begin{cases} \langle +_\alpha |0\rangle |\bar{k}\rangle = |\bar{k}\rangle & \text{if } k_j = 0 \\ \langle +_\alpha |1\rangle |\bar{k}\rangle = e^{-i\alpha} |\bar{k}\rangle & \text{if } k_j = 1 \end{cases} . \quad (3.6.4)$$

Applied on a quantum state, we can express the result as

$$\langle +_\alpha |_j \sum_{k=0}^{N-1} \alpha_k |k\rangle = \sum_l \sum_r (\alpha_{l \cdot 0 \cdot r} + e^{-i\alpha} \alpha_{l \cdot 1 \cdot r}) |l\rangle |r\rangle \quad (3.6.5)$$

$$= \sum_{\bar{k}=0}^{\frac{N}{2}-1} \left(\alpha_{\text{even}_j(\bar{k})} + e^{-i\alpha} \alpha_{\text{odd}_j(\bar{k})} \right) |\bar{k}\rangle \quad (3.6.6)$$

with $2^n = N$ and where the helper functions even_j and odd_j insert respectively a 0 and 1 at the binary numeral position j :

$$\begin{aligned} \text{even}_j(\bar{k}) &= l \cdot 0_j \cdot r \\ \text{odd}_j(\bar{k}) &= l \cdot 1_j \cdot r \quad . \end{aligned}$$

The corresponding C-style pseudo code

```

for (int k=0; k<N/2; k++) {
    even = state_vector[ even_j(k) ];
    odd  = state_vector[ odd_j(k) ];
    output_state_vector[k] = even + e-iα * odd;
}

```

is still a single iteration. Although the algorithm it is not an in-place one, as was the case in the other operations that could only modified the original state vector. The new quantum state, implemented by array `output_state_vector`, is filled by summing the matching two amplitudes from the input state `state_vector`. Note that the size of `state_vector` is double that of `output_state_vector`. The functions `even_index(k,i)` and `odd_index(k,i)` perform the necessary bit-level operations. The action of measurement operator $\langle -_{\alpha}|_i$ only differs in the above by the sign of a single constant $\langle -_{\alpha}|1\rangle = -e^{-i\alpha}$.

As an aside, we show that the MC's diagonal basis measurement can be performed in terms of the standard basis measurement by using the equality

$$\langle 0|HP^{-\alpha} = \langle +_{\alpha}|, \tag{3.6.7}$$

where P^{α} and H are respectively the phase shift and the Hadamard gate operators as defined earlier. MC measurement can thus be implemented on top of existing circuit-based QC simulators that typically only offer standard basis measurements. The standard basis measurements are typically optimized in these circuit-based simulators, taking advantage of the normalization condition: the probability to collapse to a specific basis can be read off its amplitudes directly. However, for a single-qubit measurement, all contributing probabilities still need to be summed. Our `qvm` implements both strategies: the direct diagonal measurement algorithm presented above and the transformation to the standard basis. The `libquantum` library used optimizes the H and P operators as well as the one-qubit standard basis measurement; the direct diagonal measurement was implemented manually as in the above code fragment. As a result, little difference is observed in practice between the performance of either.

3.6.4 Storage optimization

In the above operations, we have assumed storage of the amplitude vector as an array, in which the amplitude of basis vector $|k\rangle$ or $a_k|k\rangle$ is stored at array index k . i.e. storing the amplitude vector by a canonical basis. Following dictionary order, the state vector for three qubits

$$\alpha_{000}|000\rangle + \alpha_{001}|001\rangle + \alpha_{010}|010\rangle + \cdots + \alpha_{110}|110\rangle + \alpha_{111}|111\rangle$$

is then stored in memory as

| | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| α_{000} | α_{001} | α_{010} | α_{011} | α_{100} | α_{101} | α_{110} | α_{111} |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|

index: 0 1 2 3 4 5 6 7

It is observed in practice that many amplitudes are repeated or in some cases even zero. This allows for optimizations aimed at storing the quantum state in a more efficient way. Typically, saving on storage space will have the greatest effect, as it is the storage size's exponential explosion that brings the largest performance penalty. The simplest and by far most common optimization is at the level of the amplitude vector representation: using a sparse vector representation. The most interesting space optimization technique we have found is QuIDD [196], which uses binary decision diagrams – a technique used to great effect in classical circuit simulation – to capture the repeating block-structure typically resulting from the tensor product.

Our `qvm` implementation follows `libquantum` and `QLisp` in that it uses a sparse vector representation as simple space optimization technique. In a sparse vector, each element is stored together with its index such that only elements with a different value from the agreed upon default value are stored. i.e. only non-zero amplitudes of the quantum state vector are stored in a sparse vector, at the cost of management and storage overhead. Visualizing the above example and taking $\alpha_{001} = \alpha_{011} = \alpha_{110} = \alpha_{100} = 0$:

| | | | | |
|----------------|----------------|----------------|----------------|----------------|
| α_{000} | α_{010} | α_{100} | α_{101} | α_{111} |
| 0 | 2 | 4 | 5 | 7 |

The iterative operator implementations presented earlier stay essentially the same; i.e. looping over amplitude and index pairs rather than looping over an index and retrieving the amplitude.

3.7 Discussion

We have presented a layered architecture for quantum programming in a top-to-bottom manner. This layered design has been developed to enable further evolution or extension of each layer without affecting the entire software ‘stack’. We have already extended the application and pattern layers with distributed semantics in [67]. Similarly, we have multiple realizations of the execution layer that use different implementation approaches without affecting the layers above. We now validate our approach by revisiting the desired properties set out at the start of this chapter: *completeness, integration, separability, expressivity and hardware independence*.

The application layer deals with *integration* and in part *expressivity*. We have shown two different application layer approaches. The library approach provides a greater degree of *integration* of measurement patterns with an existing programming language. The design tool approach, while currently less powerful, enables non-programmers to design complex patterns in a more intuitive and *expressive* way through composition. The pattern layer vastly improves the *expressivity* of elementary measurement-based operations. We have automated the original MC composition rules, which removes much of the associated tedium in constructing large patterns. Beyond simple automatization, we have defined an equivalent and more general pattern representation and composition rule. In combination with automated execution, the pattern layer introduces multiple – arguably more expressive – ways to define patterns. The execution layer defines a Quantum Virtual Machine using the MC’s set of commands as instruction set; a set of commands proven to be universal and thus satisfying the *completeness* property inherited from the MC as is. This QVM sets a clear boundary, *separating* the quantum from the classical world, both internally and externally. Internally, it separates the limited required classical control to implement the MC’s operational semantics from the machinery required to realize the concrete quantum operation. Externally, classically-realizable features such as pattern composition are kept out of the execution layer. The realization of individual MC commands form the *hardware independent* interface between the execution layer and the tightly connected realization layer. This lowest-level layer houses the machinery required to realize the quantum operations, be it simulated or physical. In our virtual execution environment, this includes optimizations on operations and on quantum state representations.

Our proposed quantum computer architecture already demonstrates its usefulness by enabling the interactive development of large patterns otherwise too tedious or complex to create by hand. However, this architecture is intended as a starting point; a flexible foundation on which to build a quantum programming paradigm. We foresee three main ways in which to improve the usefulness and

potential adoption of this architecture. First, upwards expansion: This can happen by mapping existing higher-level quantum programming languages (such as QPL [166]) or tools to either the pattern layer or execution layer. The most obvious benefit to language designers is the re-use of the existing implementation and optimization work inside our QVM, this is particularly important in the context of further parallel optimizations discussed in Chapter 5 and Chapter 6. Another useful upwards expansion comes from building further upon the pattern abstraction, the first step of which is the visual design tool, for which we will suggest several new features in the future work section of our concluding chapter, such as recursive pattern composition. The second way to expand the scope of our architecture comes from horizontal expansions: re-using part of the layer infrastructure to introduce new features or concepts. An example of such horizontal expansion is our implementation of the Distributed Measurement Calculus [67]. The DMC implementation translates ‘agents’ in the design tool and pattern layer to CSP-like communication primitives in the execution layer. The third way we foresee this architecture evolving is by downward expansion: creating more sophisticated implementations of the QVM. With the current infrastructure in place, one can directly test the effect of QVM optimizations and alternative implementation approaches. For example, we can easily compare two alternative ways to perform quantum measurement by running patterns with a large number of commands, in the order of thousands or tens of thousands. Similarly, we can compare a parallel implementation approach to the reference sequential implementation, which is exactly what we do in the validation part of Chapter 6. The layered architecture design was aimed from the start designed to be extensible, in order to support the organic growth of a measurement-based Quantum Programming Paradigm. Any ad-hoc implementation, such as in [6], would require continuous re-implementation for each change or evolution within the paradigm. The layered design allows each separate layer to pursue extensions and optimizations while retaining the necessary cohesion to grow the QPP.

In the absence of actual practical quantum computers, the most pressing technical challenges we see for the development of measurement-based quantum programming paradigms lie with the performance of their virtual execution. Of course, there is no fundamental way to compensate completely for the exponential blowup associated with quantum simulation. Exacerbating the problem, measurement-based computing has the tendency to use a larger number of entangled qubits than the circuit model. However, this is balanced by its use of a handful of simple operations with interesting properties. This creates headroom to improve the absolute performance factor: bringing execution of a pattern down from one minute to one second can greatly improve productivity. Indeed, every efficiency improvement to the realization layer translates directly into being able to simulate and thus develop larger patterns. We see three directions in which

performance can be improved: keeping to smaller patterns during development and relying for larger applications on *pattern modularity*, *compression* schemes to reduce the quantum state storage requirements and *parallel* computing. A large complex pattern can be broken down in smaller and simpler patterns that can be developed and tested in isolation. While this doesn't help the execution performance of the larger pattern, *pattern modularity* allows for the development and testing of smaller patterns in isolation; their faster execution speed offer a more interactive design process. The context-free compositionality of patterns ensures that the composed whole is correct if its constituent patterns are too. As a side-effect, composing larger patterns from smaller ones leaves its measurement pattern in a non-standardized state, which we have seen allows for storage optimizations. The second way to increase performance is a series of techniques we put under the name *compression*. These techniques take advantage of two observations. First, while the quantum state vector might blow up exponentially in size depending on the number of qubits, MC commands can be implemented with a complexity linear in the state vector size. This is evident from their optimized implementation as a single iterative loop over the state vector, shown earlier this chapter. Second, during a quantum computation the state vector contains many repeating elements. Much performance improvement can thus be obtained by finding a more compact state vector representation. Our *tangle* optimization is already an example of such compression, decreasing the total storage requirement by pessimistically factoring the quantum state. Another common compression technique we already use is the sparse vector representation, which only stores non-zero values. Much more exotic compression techniques are available, the most notable being QuIDD [196] which represents a quantum state using binary decision diagrams, a compression technique popular in the domain of classical circuit simulation. In essence, these compression techniques exploit the regularities inherent in a typical quantum state in QC. Finally, parallel computing multiplies the computational resources that can be brought to bear. Similar – but different – regularities exploited by compression techniques can also be exploited by parallel implementations to separate state and perform parts of the computation simultaneously. In the next chapters we present a fundamental approach to a parallel implementation.

Chapter 4

Landscape of Parallel Computing

In this chapter we give an overview of the field of parallel computing. Parallel computing has a rich research history and is currently going through a renaissance, in that currently parallelism is brought into the everyday computer. We therefore first include some historical context and sketch the forces driving such change. In the second half of the chapter we formulate and argue the choice of the dataflow computational model to express quantum computing simulation as a parallel computation.

4.1 Overview

The field of parallel computing finds itself in a rather complex and interesting situation. Historically, it has existed as long as computing itself, but the parallel paradigm never superseded the simpler sequential one. The phenomenal advances of sequential processor performance in the last two decades have pushed parallel computing further into a niche field. But, in recent years, parallel computing has come back on the menu. This renaissance is not because of breakthroughs in the parallel field that have made it more popular as a programming paradigm. Rather, sequential performance has hit a plateau, forcing mainstream developers to look into parallelism. The term *performance* used in this context does not only mean execution speed, but can also mean: efficient use of computational resources and power, responsiveness, throughput, etc.

It is currently unclear how market forces and new technology will further shape the field of parallel computing. Mainstream processors have been incorporating explicit parallel technology for some years now with the introduction of multicore processor architectures. But, this is an incremental evolution, made by adapting existing sequential processor architectures. As we will see, this approach has some fundamental issue that will require a fundamental change across all computer abstraction layers to overcome. In other words, current parallel processor technology and parallel computing software are considered as stop-gap measures. This gives us cause to look back at computing history, considering how we got to the current mainstream processor technology and reconsider some of the fundamentally different parallel technology suggested in the past.

4.2 von Neumann-style Parallel Computing

4.2.1 Sequential microprocessor adoption

Microprocessors have become the stock computing hardware since their disruption of the processor market in the early 80s. Before, there was a variety of processors on the market, using different designs for each market segment. This processor market already showed a steady yearly performance growth of about 25% [107], attributed to both better design and technological improvements. In the period between the mid-1980s and 2002 this growth increased sharply for microprocessors, increasing the yearly performance growth to 52%. This growth is attributed mainly to the ability of the microprocessor design to translate Moore's law into performance. Microprocessors thus disrupted the existing processor market, growing into *the* dominant computer architecture. During the early to mid-1990s, even the large custom processors for supercomputers in the High-Performance Computing domain were supplanted by networked microprocessors, in a process dubbed *The rise of the killer micro* [14]. As a result, both computing hardware and software industry consolidated even more around the same computational model with the microprocessor as mainstream architecture. In other words, disregarding niche markets such as embedded devices and special-purpose hardware, the microprocessor supplanted all other computer architectures.

The performance gains in this microprocessor revolution has been in sequential performance. Each successive generation processor would increase both its operation frequency and the number of instructions it could execute in a single clock cycle. Under the hood, microprocessors incorporated a group of techniques to work around the von Neumann bottleneck by using resources afforded by continuous miniaturization, such as memory caches and Out of Order execution. However, the additional complexity required to implement these techniques on the processor chip themselves carry a certain cost, a cost that has recently reached a point of prohibitive diminishing returns.

4.2.2 The cost of complexity

This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.

–*The Landscape of Parallel Computing Research:
A View From Berkeley* [17].

Modern computers in essence still adhere to the von Neumann architecture. This makes sense as it is an efficient design for sequential computers. Microprocessors have increased their operating frequency spectacularly since their introduction. As illustration, Intel's 8086 processor was the first to use the currently ubiquitous x86 architecture and clocked at around 5 MHz. Twenty-two years later, the Intel Pentium 4 clocked above 3 GHz. Both processor and memory speed improved exponentially, although the rate for processors was significantly higher. This growing gap has forced chip makers to implement techniques to mitigate the effect of the *von Neumann bottleneck*. Caching, for example, exploits instruction and data locality by pre-fetching data from the main memory and putting it on fast but expensive memory inside the processor core. However, these techniques carry a cost; transistors spent on cache logic and memory do not directly contribute to the computation, but still draw power and produce heat. In addition, these bottleneck-avoiding techniques are not perfect; as the gap between memory and processor speed increases, so does the average memory access time. This phenomenon is dubbed the *memory wall* [203].

Transistor miniaturization technology has driven microprocessor performance. Higher transistor density means more transistor real estate can be packed in the same space, allowing more transistors to be reached inside the span of even shorter clock pulses. Put simply, more features can be added on a processor that runs faster with each generation. But, each extra transistor contributes to a higher power consumption and heat production through dissipation. The increasingly smaller feature sizes further add to the engineering problem of powering and cooling the processor. And finally, higher operating frequencies also require higher voltages to maintain a clear signal, thus turning a power limit into an operating frequency limit. Performance by simple virtue of miniaturization has in this way also reached a level of prohibitive diminishing returns. Its effect can be directly observed in the processor's stalling operating frequencies [107]. This is another limiting factor in microprocessor sequential performance, dubbed the *power wall*.

Both the power and memory wall can be seen as technical challenges, that may be further extended by breakthroughs in materials, production techniques and more efficient designs. A third factor exists that is more related to software: the *Instruction Level Parallelism (ILP) wall*. Parallelism is an attractive option in light of the above two diminishing returns, as it increases performance without increasing clock frequency or memory access time. ILP increases performance of a single processor by exploiting the inherent parallelism in the instruction stream of a sequential program. Using ILP-exploiting techniques, such as Out of Order execution, sequential performance is improved without changing the interface to the processor. These techniques were already at the basis of the rapid advances in microprocessor performance [107]. The amount of available ILP heavily depends on the type of computation being performed. And, even assuming a technically

perfect processor, the amount of ILP that can be exploited has an upper limit. Even before reaching this limit, one experiences diminishing returns.

In summary, while for decades miniaturization has continuously provided cheaper resources; shrinking the size of transistors such that more fit the same area. The power wall puts a physical limit to this process and conspires with the memory and ILP walls to create prohibitive diminishing returns. As a result, it is no longer worth the cost to increase the sequential performance of microprocessors. Or, quoting Berkeley's *The Landscape of Parallel Computing Research* [17]:

Power Wall + Memory Wall + ILP Wall = Brick Wall.

4.2.3 The drive for parallelism

Computers aren't getting faster, they're only getting wider.

– Steve Scott (Cray, NVIDIA)

The continuous hunger for processing power has forced processor designers and manufacturers to turn to more explicit forms of parallelism as a way to increase performance, while the exploitation of ILP was an implicit form of parallelism. Microprocessor designers started incorporating explicit forms of parallelism such as *Thread Level Parallelism (TLP)* and *Data Level Parallelism (DLP)*. The typical example of DLP is a vector operation, where a single operation acts on several data elements simultaneously. TLP occurs when a single problem is described as multiple programs; each running concurrently, but still sharing the same memory space. In terms of the original von Neumann architecture, presented in the appendix, a processor exploiting DLP duplicates the arithmetic unit (CA), while TLP-exploiting processors duplicates the entire core (C).

In Data Level Parallelism techniques, the processor's extra arithmetic units can only be used through specialized processor instructions, extending the basic instruction set used to control the microprocessor. As an illustration of the DLP that has been added over the years: the Intel Core i7 processor powering the laptop used to write this dissertation has seven different DLP instruction sets¹. Ideally, the burden falls to the compiler to automatically and transparently make use of these special data-parallel instructions where possible. In practice, programmers concerned with performance have to use specific compiler extensions explicitly or circumvent the compiler with inline assembly code to reliably make use of DLP.

¹By name and introduction date: MMX (1996), SSE (1999), SSE2 (2001), SSE3 (2004), SSE4 (2006), SSE4.1 (2007) and SSE4.2 (2008)

Large servers and supercomputers have been working with multiprocessing systems for decades, i.e. systems duplicating the entire processor. The operating system software layer already had concurrency primitives in place since the 60s to deal with multitasking, I/O issues and timesharing. Programmers were thus already familiar with concurrency primitives, such as *threads and locks*, when processor manufacturers started facing the performance brick wall. Processor manufacturers thus turned towards integrating multiple processor cores inside the same chip as a way to improve performance by exploiting TLP. The years between 2000-2005 formed a sea change in the microprocessor industry, frequently named the *multicore revolution* or the *end of the free lunch* [178]. The first multicore processor, integrating two processor cores, was launched by AMD in 2002. Hardware surveys² of client desktop computers show that today the vast majority carry dual- and quad-core processors, followed by a small fraction of single-core processors. The adoption of both TLP and DLP set the trend for parallelism in microprocessors. Hence, both are explicit forms of parallelism, the programmer has to be aware and adapt his program to utilize these hardware-afforded performance improvements. To conclude, the new TLP-exploiting multicore processors were brought about using repurposed concurrency primitives, rather than born out of parallel computing research.

We would like to stress again the sentiment of our earlier quotation taken from Asanovic et al. [17] at the start of the section: the drive for parallelism was brought about by hardware limitations, the failure to get more sequential performance economically. Multicore processors were introduced by reusing existing low-level concurrency primitives such as *threads*; from a hardware point of view, this constitutes a relatively small incremental step. But, from a software point of view, these concurrency primitives have a big impact. The issues they introduce are well documented [102, 180, 134], quoting Lee [134]:

[Threads] discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.

Multicore processors do show increased performance for applications built to expose TLP. This works especially well for *embarrassingly parallel* problems, such as image processing, which can be split into different parallel programs that share little to no data. However, in general, a program that uses multiple threads has no guarantee of improved performance. As we will see below, this is due to limitations inherent to the original sequential architecture design from which the multicore processor evolved. We therefore call the current multicore processor a *stopgap* parallel architecture: a temporary and incremental change to an existing solution, created to satisfy an immediate need for parallel performance.

²<http://store.steampowered.com/hwsurvey>

We can thus state the following two points:

- *Parallelism is here to stay* this time around [60], and
- the current multicore architecture parallel approaches are a *stopgap* measure, brought about without the required *fundamental rethinking*.

Next, we will argue the point that:

- Truly scaling general-purpose parallel performance requires a *fundamental change* in the current hardware and software approaches to parallelism.

For this, we base ourselves on the perceived limitations to multicore or *von Neumann-style parallel processors*.

4.2.4 Parallelism limitations of von Neumann hardware

[...] left to the multicore path, we may hit a “transistor utility economics” wall in as few as three to five years, at which point Moore’s Law may end, creating massive disruptions in our industry. [...] It promises to be an exciting time.

- Dark silicon and the end of multicore scaling, Esmailzadeh et al. [75]

The von Neumann architecture was originally not intended for parallel processing, actively sacrificing it for simplicity and thus sequential performance. The complex interplay between VNA bottleneck-avoiding techniques in modern processors introduces subtle but serious software errors and performance penalties, especially under thread-level parallel execution [3, 188]. This alone does not explain the observed lackluster scaling of the number of cores on commodity multicore processor; currently only specifically suited applications gain anything at all from running on more than four-core desktop processors. We make the case that *the von Neumann style of multiprocessing has fundamental issues* that prevent it from scaling effectively. Extrapolation of current multicore hardware trends corroborate this [75]. Parallel computing research in the past already reported on some issues to the VNA approach to parallel computing [16]. From a hardware point of view, two issues have to be dealt with in the construction of any parallel processor: *tolerating latency* and *data sharing*, issues that VNA-style multiprocessors have difficulties dealing with effectively.

Ability to tolerate memory latency: Latency is the delay between the selection of an instruction and its actual execution. During this delay, the data required for the instruction's operands are fetched from memory. As a processor scales in size and speed, so does the potential latency. This is not only due to the larger physical distance between parts: hardware trends show us that bandwidth scales an order of magnitude better than latency [107, ch. 1]. In other words, a processor architecture that can tolerate longer latency can more cost-effectively increase its bandwidth. A processor can thus more economically increase its total throughput (operations per second) by improving bandwidth, instead of latency. This principle can be observed in the design of modern GPUs (graphics accelerators), which trade latency for improved bandwidth; GPUs are currently popular in scientific computing due to their relatively cheap cost compared to the offered raw computing power. In sequential processors, latency is kept as small as possible by prefetching data using caches. With caches hitting the complexity brick wall, it has become an expensive way to reduce latency. Parallel processor designs can deal more effectively with latency when they can do *latency hiding*: overlapping data requests with other computations.

Ability to share data without constraining parallelism: Side effects are commonplace in sequential programming paradigms: instructions doing a read-modify-write cycle on the same memory location are not seen as a problem and even considered a boon in strictly sequential processors. However, in the context of a parallel VNA, such side-effects cause *data hazards*. For example, a write-write hazard is caused by two instructions simultaneously performing their read-modify-write cycle on the same memory location, where one instruction depends on the write result of the other. Another example is a read-write hazard, caused by an instruction reading a memory location that has yet to receive its proper value. To deal with data hazard situations, instructions have to be *synchronized* to keep their view of data in memory consistent. Such synchronization introduces additional delay or overhead, but also limits the available instruction-level parallelism in the program. Having to deal with hazards can be disastrous for a modern processor's performance, which rely on a program's ILP for their performance. Non-essential hazards can be avoided or become less frequent by changing the program: reordering operations, a functional style, immutable data-structures, non-blocking synchronization [59], using weak memory models [3], etc. This puts another parallel performance burden on the programmer's shoulder.

Mainstream processors have already incorporated small-scale parallel techniques for the past two decades, chiefly by exploiting ILP and more recently TLP. This forces these processors to deal with the above two issues. For instance, the execution core of modern processors achieves a degree of latency hiding and limiting

hazards doing a limited form of dataflow execution: micro-instructions are dynamically scheduled for execution, a scheduled instruction is only executed when their dependencies have been resolved. Only a limited execution window can be addressed as the dataflow dependencies have to be extracted from a von Neumann style instruction stream: the ILP wall. A multicore processor also needs to deal with data-sharing issues related to caching. Complex cache-coherency protocols have been devised to maintain a consistent view over a memory space shared by multiple processor cores, each having their own cache hierarchy. Although much progress has been made towards scalable cache coherency protocols [138], it remains a complex technique that adds even more hardware complexity, comes at a performance cost and requires software to be modified to deal with its quirks and pitfalls. New architectures seeking better scaling avoid hardware-based coherency (Tilera) or implement an explicit non-shared memory architecture (Cell B/E, GPUs, XMOS).

In conclusion, a von Neumann-style parallel processor faces some fundamental issues which prevents it from scaling in number of processing elements. Hardware-based techniques to avoid these issues exist, but cannot scale due to their complexity and associated hardware cost. Software thus bears the burden of dealing with complex latency and data-sharing issues.

4.2.5 Impact on software

A software developer with intimate knowledge of the underlying hardware architecture can modify his program to work on the two issues. From a software point of view, this comes at a great cost. First, there is the software engineering cost of modifying the program into a very fragile error-prone form [3]. Then, existing analysis tools and compilation technology often do not play well with such optimizations. Next, more modifications are required for each new processor generation or different architectures. And finally, the programmer himself needs to have a deep understanding not only of the complex hardware underneath, but also of the impact of his modifications on the entire software abstraction stack. In other words, now that processor hardware complexity and the carried cost of optimizing for it is at its peak, the programmer is handed the performance torch.

4.2.6 Status quo

It can be concluded that the current drive for parallelism is caused by a hardware engineering limit to the von Neumann style computational model. To overcome these limitations in practice, computer architecture designers have moved into parallel techniques decades ago. For market reasons, this was not paired with any change in the computational model, programmers still assumed the traditional VNA-based model. But, as all implicit ways to increase performance by

parallelism have been ‘mined out’ [179], so to speak, processors have been forced to move into more explicit forms of parallelism.

The drive for parallelism especially impacts software development today; On the one hand, software is forced to deal with complex parallel issues out of failure for hardware-based solutions. On the other hand, software development has been shaped by decades of a single sequential computational model. There is a vested interest in keeping the sequential paradigm status quo. This is a powerful factor that causes current parallel software approaches to favor small increments to the existing sequential model. We refer to this effect, by lack of a better name, as the *von Neumann gravity well*. The von Neumann gravity well means that any significant steps away from the sequential model requires a great investment of development and research to make the step worthwhile.

The stopgap parallel approach taken by the processor designers is a deceptively small step away from the sequential model: repurpose existing low-level concurrency primitives. Deceptively, because while this is a small and cheap change in hardware, it has an enormous impact on software development [134]. Many abstractions that keep the illusion of a sequential programming model are being suggested on top of these concurrency primitives, abstractions such as software transactions [102]. But, these have met limited success in practice [43]. There already exist high-level computation and coordination models that move away from von Neumann style software; such move has been advocated in the past and most notably by Backus [19], although parallelism was not always the chief argument. However, programming languages, tools and frameworks have still been shaped by the von Neumann style computing model. For instance, the popular purely-functional programming language Haskell relies for the execution of its semantics on an extremely imperative abstract machine [121].

All signs point towards a the need for a systemic change in the computational model in order to scale effectively and take advantage of highly-parallel hardware. This means there is a need for a coordinated effort across both the hardware and software industry to develop a fundamentally different parallel computing paradigm.

In the next section, we give a broad overview of past and present parallel approaches. In it, the effect of the von Neumann gravity well can be observed by the tight clustering of current parallel approaches around the same category. But, we will also see that several fundamentally different parallel approaches in both hardware and software that have been researched in the past, when the pull of the von Neumann gravity well was not quite so strong. From these different approaches we have selected the dataflow computational model, a choice we will defend in the next section using many of the terms and arguments broached in this past section.

4.3 Parallel Computing

First, we put parallel research in its historical context, sketching some of the original reasons why it was pursued and how it came to fall from grace. Then, we provide a categorization of parallel computing approaches in order to better place each parallel approach in context. We finish by giving a short overview of currently state of the art.

4.3.1 Historical Context

The parallel architecture research of the 1960s and 1970s solved many problems that are being encountered today.

-J. Dennis[60]

Rise. Prior to the microprocessor, computer processors were large, bulky and typically composed of discrete components. Furthermore they were relatively slow, making parallel processing a natural choice to increase computational power³. More concretely, in the pre-microprocessor era there were generally three reasons for doing highly parallel computing. First, an *economic* argument of *scale*: producing a unit in large quantities means that its hardware complexity becomes the main cost factor. In other words, it is easier to produce a multitude of small and simple computational units than it is to produce large complex ones. The performance of both being equal, the balance thus tips in favor of the multitude of simple units, which then have to run in parallel. The second reason for parallelism is one of *redundancy*. For several decades the failure rate of computer components was many times higher than today, enough to warrant the design of redundancies in the system. As an example of redundancy in practice: the earliest space shuttle flight had three computers on board performing the exact same tasks simultaneously, comparing the results with a majority vote. The third and most prominent reason for doing parallel computing is *performance*. If the fastest available uniprocessor cannot get a certain task done in the required time frame, one is forced to use parallelism to increase computation speed.

Fall. Parallel computing developments could not keep up during the two decades of fast-paced improvements to uniprocessors, invalidating the three reasons to do parallelism in various ways. Hardware error rates were improved through more reliable hardware and built-in error correction circuits. Microprocessors

³Most hardware techniques discussed in the last section were in fact first introduced in the bulky computers of the 50's and 60's [13, 60]. Only decades later were these techniques introduced in modern microprocessors.

obtained a economics of scale in a different way, by using the same processor to penetrate an entire range of markets that used to be populated by a multitude of different processor architectures. The expensive initial cost of launching a new processor product could be amortized by selling it simultaneously for personal computers, workstations, minicomputers mainframes and even supercomputers. The microprocessor saw a rapid technological development giving rise to phenomenal performance gains, leaving in its wake the remains of many parallel computer research projects and companies that could not keep up. Parallel programming languages and techniques slipped out of the common programmer's mind and curriculum. The High-Performance Computing domain became the only niche where parallel computing research continued, in a very specialized context.

Renaissance. Today, we are seeing the start of a *parallel renaissance*. The multicore revolution with its push to TLP has put parallelism back in the mainstream. But as seen above, fundamental issues make multicore a stopgap measure. It is increasingly obvious that to break away from the current status quo, it would take a change in both hardware *and* software approaches. We see such alternative approaches in niche or special-purpose markets that could evolve independently from the uniprocessor: digital signal processing, networking or audio/video stream domain computing. Particularly interesting are the Graphics Processing Units (GPU) that have evolved from fixed-function accelerators to the massively parallel general-purpose processors they are today [149]. Other processor architectures indicative of an impending change are the embedded multicore ARM Cortex or XMOS processors, hybrid processors such as the Cell B/E processor, the network-on-chip Tiler and the recently announced Intel Xeon Phi. Quoting Herb Sutter [179], “these are not separate trends, but aspects of a single trend.” Each requires programmers to take a different approach to writing their software; no compiler exists that can effectively parallelize a sequential program in the general case . Each parallel hardware or software system needs to deal with fundamental issues to achieve scalable parallel computing. While we cannot definitely predict the future, it is useful to investigate current trends and past approaches in order to formulate required properties to perceived fundamental obstacles.

4.3.2 Parallel Computational Models

The free lunch is over. Now welcome to the hardware jungle.

–Herb Sutter [179]

To give a better overview of the existing parallel approaches and computational models, we frame them in a broader category. We give a schematic overview in

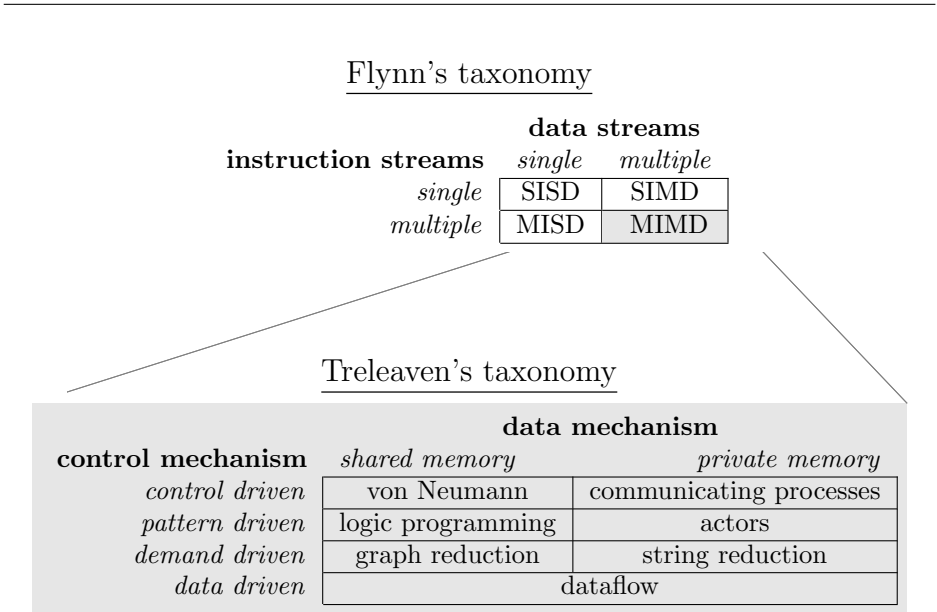


Figure 4.1: Categorization of parallel computing systems.
 Categorization of parallel computing systems in terms of two taxonomies.

Figure 4.1 of the categorization presented here. Flynn's taxonomy [78] is the most commonly seen categorization. It categorizes processor organization approaches by their sharing of instruction and data streams. The two most prominent parallel systems are either Single Instruction Multiple Data (SIMD) or Multiple Instructions Multiple Data (MIMD). Vector machines and Digital Signal Processing (DSP) processors are typical examples of SIMD. SIMD exploits data-level parallelism by having several processing elements (PE) execute the same instruction in lockstep on separate pieces of data. In MIMD, thread-level parallelism is exploited to feed each PE a different stream of instructions. While these terms are still often used, many computer systems have evolved to be too complex to be meaningfully covered by Flynn's taxonomy. Today, many processors mix both SISD and SIMD instructions inside each core, which are organized in a MIMD multicore fashion; it thus becomes difficult to label modern architectures as such. The vast majority of parallel approaches are situated in the MIMD category. To differentiate further within this category, we use Treleaven's taxonomy [189, 10] which provides a much richer categorization of complex parallel approaches. Treleaven's taxonomy considers the computational model's mechanisms that drive the execution of a program and the memory organization. The four control mechanism that can be observed in various approaches are:

- *Control driven*; Each statement in a control-driven program determines the next statement to be executed. The vast majority of sequential computing hardware is control-driven, as of course are the popular imperative and procedural programming languages. Two of the most practical and prominent parallel members of this category are MPI [79] and OpenMP [31]. Abstract models such as PRAM [74], BSP [191] and even CSP [110] also fit within this category.
- *Pattern driven*; The matching and unification of logical patterns against a database of facts drives the computation. Parallel logic programming systems such as Parlog [95] fit in this category. Also fitting this description are Actor-based systems based on the original formulation [4], such as the Erlang programming language [15].
- *Demand driven*; Computation is driven by the recursive substitution of expressions, also known as reduction. A given expression is simplified without changing its meaning, triggering the reduction of a sub-expression until the process terminates. Pure and lazy functional languages require such execution semantics, which mimics the lambda calculus' β -reduction. Such a reduction strategy can exploit parallelism enabled by the Church-Rosser theorem [48], which states that the order in which the substitutions are performed does not matter for the final result. Two different reduction execution strategies have been suggested and used: string reduction [72] and the far more popular graph reduction [121]. The main example in this category is Concurrent Haskell [120], because of the consolidation the FP languages community around Haskell. Reduction-based processors were planned [137], but only a few ever reached the prototype stage [10].
- *Data driven*; Operations execute as soon as their operands are available. Data-driven programs thus benefit from a program representation in which data dependencies are either explicit or trivially derived. Present-day examples of such execution models can be found in a wide variety of computer systems: the Out of Order execution in modern processors [107, ch. 3.6], stream processing [187, 176] or even Google's MapReduce [58] framework. More obviously within this category sits the family of Dataflow computational models, as originally popularized by Dennis [62], which have led after two decades of research to multiple dataflow machine architectures [195] and high-level programming languages [118].

Current state-of-the-art parallel systems are mainly within the control and data driven categories. Control driven systems integrate well with the current prevailing computer systems, they can re-use existing development tools and integrate with compiler and language technology. The observed trend going from control

to data driven is the loss of programmer control over the exact timing and ordering of execution. This makes sense in a parallel context where such *flexibility* is necessary to do more work simultaneously, helping to deal with the latency and sharing issues, which were covered last section. Data driven systems offer more parallel execution flexibility, but their programming model can be more restrictive to program in. Below, we sketch the landscape of recent and mainstream parallel approaches.

To create some distinction between these approaches, we use the terms *framework* and *library* approaches.

In *framework* approaches, the programmer divides the program in separate tasks explicitly; but, when and where each task is executed is only defined implicitly. Within each task, the programmer uses the host control-driven language as usual. The parallel runtime takes care of the task execution juggling act by using an efficient scheduler, both Intel TBB and Java F/J use a Cilk-style [30] work stealing scheduler [29]. In these Cilk-style frameworks, the programmer defines control dependencies between tasks. Examples of similar approaches using data dependencies are typically called stream [174, 103] or dataflow frameworks [113]. Dataflow framework approaches are often used for SaaS, BigData and Cloud applications on warehouse-sized distributed computers [57, 113]. Large parallel distributed systems⁴ such as warehouse-sized computers require even more flexibility, resulting in mainly data-driven frameworks; Popular instances are Google MapReduce [58] and its derivatives such as Hadoop. Dryad [113] has a more general dataflow flavor.

Library approaches are different from frameworks in that they do not require explicit task splitting. We define libraries as parallel approaches that use the invariants of operations on special datastructures to expose parallelism. For instance, calculating the sum of all elements in a vector can be expressed as a parallel computation; the user of the library only needs calls the sum functionality on the library's vector datastructure, which can then use this specific operation's mathematical properties to make assumptions on the *flexibility* of the computation. Other less trivial examples are the parallel prefix sum [132], the map and the reduce (or fold) operations. Some advanced implementations combine this library approach with dynamic compilation techniques to string together several such operations and produce optimized results, for example Microsoft Accelerator [184] and Intel Array Building Blocks [148]. In summary, frameworks exploit the execution flexibility between tasks for parallelism, providing the familiarity of control-driven execution within each task. In contrast, library approaches offer a set of parallel datastructures and operations without taking general control-flow away from the programmer.

⁴We mention distributed computing frameworks because of the frequent overlap with the parallel computing domain and because they are in essence parallel models of computation.

MPI and OpenMP are the main tools used by experts in the High Performance Computing domain. Both are very much control-driven, but offer a different type of memory mechanism and interface to the programmer. Both use a library approach, implemented in or hooked into current sequential programming languages. MPI offers low-level inter-process communication primitives, whereas OpenMP offers a thin structured abstraction over threads and locks. Both have well developed implementations, MPI gets the most performance out of a cluster and OpenMP from individual shared-memory multicore processors; today, both are typically combined. However, such a high-performance low-level approach comes at a high development cost. In light of the parallel computing resurgence, several initiatives seek to introduce new approaches to make parallel programming more accessible to programmers in general. Some examples of this in the HPC domain are the High-Productivity languages Chapel [45], X10 [143], Habanero [44] and Fortress [7]. In the spectrum between performance and productivity, low-level libraries and higher-level languages, sit many other approaches. At the higher end of the spectrum sit framework approaches such as Intel's Threading Building Blocks or Java Fork/Join; all featuring runtimes that are relatively heavy-weight compared to low-level approaches, but efficient relative to the current implementations of parallel languages.

It is clear that today there are a wide variety of parallel approaches. Not fully covered above are the fruits of parallel research in the past, which are not used or applicable in today's microprocessor computer landscape. For example, many reduction and pattern-driven parallel computers were investigated worldwide in the 80's during the Fifth Generation Project [108, 189]. Considering the size and history of the field, we can never exhaustively eliminate each parallel approach. We do however defend our choice of dataflow in the next section based on necessity conditions.

Before moving to these properties we want to formulate three very general arguments. First, the push towards higher degrees of parallelism naturally favors models with higher degrees of asynchronicity and more localized synchronization mechanisms. The pure dataflow model can be considered an extreme case: synchronization only happens between individual operations when they have a semantic data dependency. Our work presented in Chapter 5 can *carry over* to other parallel models that exhibit the listed properties, but perhaps do so in a lesser degree. Next, dataflow constitutes the most extensively researched, implemented, used and promising *non-von Neumann approach* to parallel computing; making it a prime choice as a vessel to escape the aforementioned von Neumann gravity well. In other words, using dataflow forces a change in the prevalent mindset, in which sequential is the default and parallel the special case. Lastly, the dataflow model has a *dual high-level low-level nature*; being both successful as a high-level abstract model and as a bottom-up physical machine model. Such

a dual nature is a benefit in the current context, in which experts advocate a concerted effort to develop a fundamentally different model for parallel computing, simultaneously across all levels of abstraction [18, 111, 139, 75, 41, 3].

4.4 Dataflow for Quantum Computing Simulation

4.4.1 Overview

We have chosen *dataflow model of computation* as foundation for our parallel approach, which is described at the end of this section. We first defend our choice of dataflow by using five necessary properties: *fine-grainedness*, *asynchronicity*, *analyzability*, *implicitness* and *data-orientedness*. These properties come from several angles. First, implicit and fine-grained parallel models scale better on highly-parallel machines. Next, MC operations are very sparse and fine-grained operations, with a natural data-oriented representation. Finally, dataflow is a clean and simple model with a clear mathematical formulation.

The dataflow model of computations was originally formulated in the 60s by Karp and Miller [123] as graph-theoretic model for the analysis of parallel computations. Although Karp first formalized the model, similar program organizations could already be found in earlier publications [164]. Most influential early on was the work by Jack Dennis [61, 63], who popularized the use of dataflow as basis for a parallel processor architecture during the '70s. Over the years, many different dataflow processor architectures have been proposed [195], with a similar array of dataflow languages and models. Dataflow has also been applied in domains other than parallel computing, such as signal processing [135] or process algebra [122]. Here, we will keep close to the original vision: dataflow as computational model of a dataflow computer.

At its heart, a dataflow computer works by executing operations whose operands are ready. A dataflow program is organized as a graph, with operations as nodes and data dependencies as directed edges. Operationally, data flows along the edges in the form of data tokens, typically containing a datum and destination information. An operation is selected for execution when all of its incoming edges contains a data token. After an operation has been executed, new data tokens containing the result are sent to their destinations. In contrast with a stored-program computer, e.g. with a von Neumann architecture, there is no concept of memory location nor sequence of operations. A dataflow program may at any point in time have multiple operations that are ready for execution, the order in which these are executed does not affect the final result of the computation [123]. The parallel execution of a dataflow program is thus simply a matter of simultaneously executing readied operations. To illustrate, we work out a simple dataflow computation in Figure 4.2.

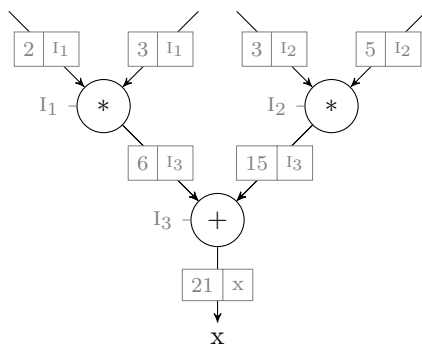


Figure 4.2: Graph representation of a simple dataflow program computing $x := 2 \cdot 4 + 3 \cdot 5$. Execution starts when the input data tokens are fed to the dataflow computer. A data token is represented here as rectangular boxes carrying a datum and destination.

For a more thorough treatment of dataflow architectures and languages we refer to the surveys by Treleaven et al. [189], Johnston et al. [119] and Veen [195], and the books by Sharp [169] and Almasi and Gottlieb [10].

4.4.2 Fine Grainedness

Conceptually: expose maximal parallelism. Every program has an essentially sequential part: control and data dependencies that impose a certain ordering or *critical path length*. These impose a minimum execution time, even given infinite parallel resources. The parallel speedup of a program is thus limited by the size of the sequential part of the program, as formulated by *Amdahl's law*. However, it has to be appreciated that a sequential part in a *program* does not imply a sequential part in the original *problem*. Using a different algorithm or even programming style has an impact on the parallel and sequential parts. In short, a programmer should be able to *expose as much parallelism inherent to the original problem*. Using a fine-grained dataflow model can expose any inherent parallelism to the level of individual instructions, making it singularly suited as a highly-parallel program representation.

Performance: balancing parallelism and overhead. The *granularity* of a parallel program is the ratio between the number of operations it performs in total and the number of times it has to communicate or synchronize. That is, the coarser the granularity, the more sequential operations are performed by each of the program's distinct parallel tasks. Expressing the same problem as a fine-grained or coarse-grained parallel program provides different trade-offs. The

fine-grained version contains more overhead because of the scheduling or synchronization of the more numerous parallel tasks. A coarse-grained version has larger blocks of consecutive sequential operations that do not require to communicate with the rest of the program. But, the coarse-grained version contains a larger sequential part than does the fine-grained version. As more tasks can be potentially executed in parallel, the more processing units can be kept busy. The optimal granularity is dependent on the underlying machine architecture. For example, today's multicore processors fare better with coarse-grained parallel computations: any context switch or synchronization may stall the fast and 'fat' sequential execution cores. GPUs on the other hand fare better with computations with an abundance of fine-grained parallelism. This is reflected in the GPUs entire architecture design. GPUs have slower memory latency, but by relying more on latency hiding through parallelism rather than caching, they can achieve a far greater bandwidth and throughput. A trend can be observed towards slower, 'thinner', simpler cores, but with a much larger number of cores, compared to the few 'fat' cores in current multicore processors. This trend towards *manycore* processors can be observed in parallel research [18, 32, 103], but also today's boundary-breaking parallel processors from companies such as the Tiler [201, 185], Adapteva [153] and lastly Intel with their Xeon Phi [165]. In the past, a similar trend could also be found in research in the experimental dataflow machines built in the 80s and 90s, but also again commercial processors with the Connection Machine [109] series and the Inmos Transputer [202]. It is thus not unreasonable to conclude that a scalable highly-parallel architecture requires a finer rather than coarser grained parallelism.

It need not be the case that fine granularity causes a disproportionately large overhead. Several techniques have been developed for dataflow machines to reduce the fine-grained parallelism overhead [170, 133], often by coarsening computations by extracting sequential code fragments [26, 163, 91]. It has been shown that sophisticated static compiler and dynamic hardware techniques can reduce the overhead associated with fine-grained parallelism. The SISAL [83] language, which compiles to a dataflow intermediate representation, matched the sequential performance of the then fastest available language: FORTRAN [42]. Its functional and dataflow nature were praised as being more conducive to analysis, optimization and implicit parallelization. On the flip side, extracting finer-grained computations from coarse-grained computations requires what is in essence a general parallelizing compiler: a notoriously difficult and in general still unsolved problem [100]. Furthermore, many of the techniques used in optimizing and parallelizing compilers consist of dataflow analysis [9], but these still need to extract the data-dependency graph from a sequential input program.

To conclude, a fine-grained parallel program is not fundamentally less practical than a coarse-grained one, on the contrary. The more parallel the underlying-

ing execution hardware, the more benefit is gained from a fine-grained program representation. Existing optimizations can reduce the overhead cost normally associated with running a fine-grained program on comparably coarser-grained parallel hardware.

Sparse nature of MC operators As seen in Chapter 2, the simulated execution of MC programs can be considered on two levels. Taking each vector state as a data element, we obtain in a natural way a coarse-grained computation. Considering a state’s individual amplitudes as data elements results in a fine-grained computation. The parallelism at a coarse level is heavily dependent on the number of factorizable states, achieving very low parallelism for non-trivial MC programs. Considering the computation at the level of amplitudes unlocks a vast amount of sparse and data-parallel computations, with a natural fine-grained dataflow representation. In practice, applications with similar such characteristics, such as the Fast Fourier Transform, have been successfully expressed and optimized using a fine-grained dataflow representations [186]. We show in Chapter 5 a similar result holds for MC.

4.4.3 Asynchronicity

It is self-evident that parallel models require to some degree to be asynchronous. SIMD models are relatively synchronous, all data elements being processed in lockstep based on the single program counter. MIMD models have more asynchronous elements, multiple tasks run independently and simultaneously. Shared-memory MIMD systems require in practice complex cache-coherency protocols and explicit synchronization to keep the semblance of a unified memory architecture. Private-memory MIMD systems such as MPI or Bulk Synchronous Processing-based systems [191] run independent processes, but require communication steps to share data.

Many parallel algorithms take a synchronous approach: parallel processes perform their data communication during globally synchronized steps. That is, in a synchronous approach, all parallel processes will wait for all other processes to finish their task before communicating to each other the data required for their next task. Some problems work well with such synchronous parallel approach, for instance Cannon’s matrix multiplication algorithm [93]. These problems typically have parallel tasks that perform a similar amount of work and only require limited local communication during the global synchronized communication step. Otherwise, the communication steps quickly dominates the computation when scaling the number of processing elements, considering the latency involved and the amount of idle hardware during such global synchronization overhead. Asynchronous parallel algorithms exist for problems that do not scale well with syn-

chronous approaches; to reduce the amount of communication [27] or to balance out the parallel load [23, 12].

The parallel quantum computing simulation state of the art [182, 56, 90] mainly takes a bulk synchronous approach. In each, we observe the entire quantum state being split and stored across all processes. A large global synchronization step between the application of each quantum operation takes care of data sharing. But, during this communication step, each process has to communicate with all other processes, causing a large communication step for relatively few computations. In our concrete case, parallelizing the virtual execution of the MC, it can be observed that each new amplitude after a unitary operation does not depend on all other amplitudes. In other words, in our case, a global synchronization is not semantically necessary. It is thus worth investigating an *asynchronous* parallel approach for parallel quantum computing simulation.

Expressing quantum computing simulation as a dataflow computation allows for a great degree of asynchronous computations. Operations on amplitudes can overlap based on the available data, rather than having to wait for all other operations to also have received their data. This becomes increasingly important as the size and scale of the parallel computer increases, as the increased amount of communication and latencies becoming the bottleneck in the computation.

4.4.4 Analyzability

In the bigger picture, there is still much scope for higher-level language designs which encourage programmers to think in a way which naturally encodes effectively on coming architectures – and even for new architectural features corresponding to programming innovations. Can we return to the comfort of 1985 when implementation languages and computer architecture matched?

–Alan Mycroft [144]

Mainstream software is built on a high stack of relatively thick abstraction layers. A productive programming language is surrounded with an entire ecology of tools to help developers. These tools require an in-depth knowledge of the programs they act on. Similarly, compilers and low-level runtimes need deep knowledge about the program to allow certain assumptions to be made to enable optimizations. There is already a mismatch between the simple random-access computer model assumed by many software abstraction layers and the reality of modern computer architectures [3, 107]. The switch to more parallel systems fundamentally disrupts this already mismatched stack. To illustrate, consider a programmer with a problem that can be naturally expressed as a parallel computation. He can express this problem in parallel in his favorite programming

language, for example by using a data-parallel library approach. This program is translated to a low-level language such as C, in which the parallelism is expressed using concurrency primitives. The C compiler in turn has no information about the original intent of the programmer, rather it needs to build on assumptions based on the sequential C programming model. Going even lower, the processor translates the instruction stream produced by the C compiler into machine instructions and turns to assumptions it can make about the assembly computational model to perform parallel execution. Modern processors, as we have seen, have evolved away from the initial sequential computational model. Weaker – less deterministic – memory models are used in order to more effectively share data over multiple cores [3]. This entire mismatched stack greatly complicates not only the task of optimizing for parallel computation explicitly, but also the analysis required for automatic optimization and verification tools.

Choosing a single overarching model for program representations across all abstraction levels has the benefit of providing optimization tools with the deeper knowledge they need about the application and programmer’s intent. Moreover, the more parallel the underlying system, the more assumptions are required for performance. The dataflow program representation in its various forms has simple mathematical formulations and properties that make it highly suited for analysis. Dataflow was originally [124] formulated as a more mathematically rigorous and verifiable model to express parallel programs. Dataflow has shown to be highly suitable for various static and dynamic optimizations, both in the past [163] and more recently [187]. Today, we can still find dataflow at nearly every analysis step on the software abstraction stack; from large frameworks [58, 148], domain-specific tools [47, 157] to optimizing compilers [100, 8, 111, 157]. Note, these dataflow graphs live only in the back-end or as temporary intermediate representation such as Single Static Assignment [51]. It may safely be said that dataflow has earned its spurs as analyzable model.

The analyzability of the dataflow graph itself will help us in Chapter 6 with the theoretical validation of our approach. Several metrics important to parallel performance can be read off the dataflow graph directly, metrics such as critical path and average parallelism. Such quantitative analysis can be powerful performance predictors, but also give us theoretical bounds on the amount of exposed parallelism and expected parallel execution behavior. In other words, the analyzability of the dataflow model helps us to quantify the amount of available parallelism in the simulation of the MC execution.

4.4.5 Implicitness

Expressing a parallel program is not fundamentally harder than expressing a sequential one. The complexity is mainly caused by having to optimize the parallel

program. Optimizations are unavoidable, as parallelism is mainly for increasing performance and there promises to be a much wider variety of parallel architectures than we ever saw for microprocessors. Parallel performance today means modifying the parallel program for the underlying hardware, keeping the abstraction layer as thin as possible to remove any unnecessary overhead and tweak the program around potential bottlenecks. Modifying a program by hand for performance requires an intimate knowledge not only about the various hardware components but also about the multitude of software abstractions between the program code and the hardware.

Explicit parallelism means the programmer explicitly divides the original problem space, distributes the data and orchestrates what operations have to be executed when. This offers the most manual way of optimizing a parallel program; an expert using explicit parallelism can with enough effort get the most out of the parallel hardware. This approach can often be found in the HPC application domain, which uses MPI as staple approach. The HPC domain is characterized by having a high hardware cost, which makes it logical to allow for a higher software development cost to use more of the costly hardware's potential. In other computing domains, the development cost dwarfs the hardware cost. Implicit parallelism provides the programmer with a relatively familiar abstraction and computational model: one that is not ostensibly parallel, but which can be easily transformed and analyzed into an optimized parallel computation. The main argument against explicit parallelism comes from the foreseen processing hardware changes. Different parallel architectures will require rewriting a large body of software written in an explicit parallel style. An implicit parallel style leaves room for a wider variety of underlying parallel architectures.

The dataflow computational model has a simple and intuitive implicit formulation. A more familiar approach to dataflow for the programmer would be through functional languages, which have been shown to map well to dataflow [84, 2, 204]. A dataflow graph does not impose a specific data storage or layout, does not have a notion of processing units or different tasks. The programmer only has to divide the problem into separate operations. Note that even this operation division does not guarantee the operations will not be grouped for optimized performance. A fine-grained dataflow program leaves much room for automatic optimizations, which it requires in practice for performance.

4.4.6 Data-orientedness

Communication in some form or another is the bottleneck for nearly every parallel computer architecture, from multicore processors to clusters. In shared-memory systems this is not always explicit, with cache-coherency hiding and automating the communication between processing elements. Even with such automatic

caches a programmer has to be aware of data movements in order to avoid disastrous performance penalties. PGAS models [205] provide primitives for the programmer to express locality or affinity to processing elements, providing the runtime with some information to optimize data movement. Highly parallel processors such as modern GPUs deal with data movement by forcing the user in an explicit memory hierarchy and by working with many overlapping memory requests; delaying dependent computations until data is available.

Efficient parallel execution of quantum computing simulation is mainly a problem of data-movement; because of the combination of a large number of amplitudes in each state, few computations per amplitude and potentially very wide strides over memory locations that ruin traditional data-locality. The straightforward approach to building a parallel quantum computing simulator is by distributing the quantum state data over multiple computation nodes. Typically, a one-qubit operation will access all of the amplitudes in a quantum state. When a state is distributed, each node stores only a subset of all amplitudes. For each operation, a node needs to pull individual amplitudes from other nodes [56]. As the quantum state grows exponential in size, it is easy to see that the same happens for the amount of communication. To optimize performance, it is important that data movements are explicit.

Dataflow requires data-dependencies to be explicit in the program. This enables flexible and overlapping memory requests, optimized data movement analysis and in general removing the direct coupling of processor performance and memory latency [16]. Other benefits result from the automated analysis made possible by the data-dependency graph, something we mainly covered under the ‘analysis’ argument. A less obvious benefit is a human-oriented one. Having the programmer explicitly work with data-dependencies also gives him a better view of the cost of a computation; today it is not instructions that are expensive, but data use [107]. It can also lead to a better understanding of the original problem. In our case, explicit data dependencies led to the connection between qubit positions in a tensor and the stride permutation, leading to the expression of a qubit position-changing operator in the next chapter. In other words, dataflow forces you in a style that gives you the tools to work on data movement problems.

4.4.7 Conclusion

We thus argue that dataflow is a parallel computational model that not only matches well to the problems faced by parallel quantum computing simulators, but also approaches parallel computing in a more fundamental and lasting way. Other computational models might match some of the above properties; the actor model is also an asynchronous model and MPI can also be used to implement an asynchronous approach. But, to our knowledge, only the dataflow model

combines all above properties. Its dual nature and origin as both mathematical and machine model is seen as an advantage in our context.

We have split our parallel approach for parallelizing the QVM into two distinct parts, first taking a conceptual approach followed by its practical implementation. The conceptual approach, presented in Chapter 5, acts as the blueprint, deconstructing and covering key aspects of the QVM's virtual execution. The mathematical simplicity of dataflow as a conceptual formalism allows us to present the parallelization of the QVM to the reader by abstracting away many hardware issues, but still staying within a dataflow model of computation. The practical implementation covered in Chapter 6 serves as validation for the practical feasibility of our approach.

Chapter 5

Bridging QC and CC: mapping Measurement Calculus to Dataflow

5.1 Bridging measurement patterns and implementation using models

5.1.1 Goal

In chapter 2 we have advanced a model and language for quantum computation, the Measurement Calculus. In chapter 4, the state of classical computing was presented with special attention to parallel computation hardware and models. In this chapter we bridge both domains, transforming the quantum virtual machine presented last chapter to an efficient classical, parallel computational model.

One way to approach classical simulation of the MC is to implement it as a computer program directly on top of linear algebra libraries. As we have seen in Chapter 3, this has been done several times for Quantum Computing in general and in a few cases for one-way QC and even MC specifically [6]. While ad hoc, these implementations do provide insight into the nature and qualitative properties of quantum computing. Trying to efficiently simulate or emulate QC computation drives home the core elements that make it computationally hard. This led to insights such as efficient subsets of QC [92] or non-obvious ways to compress quantum state [197].

In this chapter we bridge the QC and classical domain not by a direct implementation, but by step-wise decomposition and analysis of computational models.

Imagine a valley with the quantum computing domain on the left flank and the classical computing domain on the right, we build a bridge connecting both flanks by first planting several supporting pillars in between. Each pillar is a computational model or abstract machine that can be connected with the neighbouring pillars by way of equivalences and transformations. On the left sits the Measurement Calculus as the anchor for the Quantum Computing domain. On the right sits a fine-grained dataflow model for parallel classical computing. The choice for these two anchor point models has been thoroughly defended in the previous Chapters 2 and 4. A visual overview of this bridge metaphor is presented in Figure 5.1. All models presented in this chapter are graphs, which stems from the practical fact that both the MC and the dataflow model have an obvious and concise graph representation. The reasons for using computational models come from different directions. First, they allow us to discuss and analyse the problem at hand while abstracting over issues such as numerical accuracy and implementation platform details. Second, using multiple models allows us to tackle the separate individual features or issues more clearly, which would be lost in the context of a direct implementation. Third, by spreading over abstraction levels we ensure there is a common entry point for multiple quantum simulation strategies. In other words, it makes parts of the bridge reusable for different types of implementations. For example, a coarse graph could just as well represent a stabilizer calculus [92] computation.

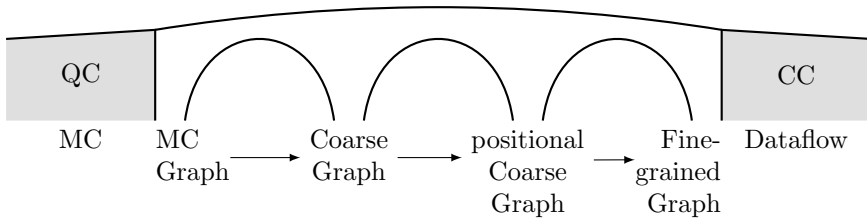


Figure 5.1: Schematic overview of the ‘bridge’ spanning the quantum and classical computing domains, using multiple intermediate models as pillars.

5.1.2 Requirements

The choice and development of the models below is guided by a set of requirements. These requirements arise from blending some of the practical requirements satisfied the MC from Chapter 2 on the one hand and those satisfied by the dataflow model on the other.

Executability. The model has to describe the computation with enough detail such that it can be expressed formally in terms of operational semantics. Models simplify, which helps to focus on important items. Models also abstract, which help a computational model to apply to a wider range of computing contexts. The purpose of the executability requirement is to avoid abstracting too much details important to execution. We have shown in Chapter 2 that the MC satisfies this requirement by virtue of its small and simple operational semantics. With this requirement we ensure that each consecutive model is detailed enough to preserve this property.

State-oriented. This second requirement forces each intermediate model to express its computation through explicit state transformations. It should be clear and explicit in the model for each operation which data elements it uses and how they are modified. In the current computing landscape it is the efficient use and movement of data that forms the main performance bottleneck, both for parallel and sequential computations. This requirement forces the model to make the manipulation of state explicit through data-dependencies, which is essential for analysis and efficient implementation.

Parallel. We require that when operations can be performed simultaneously, it should be explicit or obvious from the model. In other words, we require each model to be a parallel model of computation; or at least, a model where a set of simultaneous operations can be extracted non-trivially. This requirement is important because from the start it forces the computation to be expressed as a parallel one. Parallelism depends on the nature of the original problem and turning a sequential solution into a parallel one is non-trivial. With this requirement we force the modeled computations to expose a maximal degree of exploitable parallelism.

Simplicity. This final requirement seeks to reduce the general complexity of the various models. Each operation or feature should have a single and simple focus, upholding separation of concerns. Transforming a model to a more concrete one should not add more complexity, unless the added concept is essential. And finally, optimizations are introduced only if they simplify a model.

5.1.3 Overview

The structure of the rest of this chapter follows the various models from more abstract down to the more concrete. We start in Section 5.2 from essentially a graph representation of MC programs, which is used in the construction of the first state-oriented graph model in Section 5.2.2. In the following coarse-grained

model, the positional coarse graph in Section 5.2.3, the MC operations are made more concrete by referring to qubits by their tensor position rather than name. Finally, we present the fine-grained dataflow graph in Section 5.3, in which MC operations are completely described as a operations on individual amplitudes.

5.2 Coarse grained graphs

5.2.1 Measurement Calculus Graph Model

Overview

The Measurement Calculus defines its execution semantics using a set of low-level operations for a one-way quantum computer. As we have seen in Chapter 3, measurement patterns have a natural graph representation. In Chapter 3 we have used a graph representation a pattern composition as a way for quantum programmers to design and compile MC programs more easily and visually. Here we use a graph to represent individual measurement patterns as the starting point from which we derive other models. A measurement pattern forms a graph where each node is an MC command and edges represent qubit and signal dependencies, conform to the definiteness conditions. The definiteness conditions also ensure that such a measurement graph is a directed acyclic graph (DAG). We start by defining the graph's structure and abstract execution, then and conclude with a requirements check to clarify why we do not use the Measurement Calculus Graph directly as a computational model.

Structure

The Measurement Calculus Graph (MCG) is a Directed Acyclic Graph (DAG) in which nodes are associated with an operation and edges represent qubit or signal dependencies. The abstract machine for the MCG model works in two phases for each execution step: the select and application phases. The select phase determines which operation is ready to be executed. This selection phase is absent or at least trivial in the abstract machine for the MC, where measurement patterns are represented as a command sequence and selection is thus simply taking the next element of the sequence. With the MCG being a graph, this selection phase is somewhat more involved. An operation is selected when all its dependencies are resolved, thus it either had no incoming edges to begin with or each of its incoming edges come from operations that have already been applied. In the application phase, the selected nodes are *fired* in arbitrary order, executing the action of the operation they are associated with, transforming the computational state as described by the MC's operational semantics in Chapter 2. The order in which nodes are fired depends on the edges of the graph. There are two types of edges, one for qubit and another for signal dependencies. An edge (o_1, o_2) between two operations expresses the dependency: “ o_2 can only be executed after o_1 ”. A DAG imposes a partial order on its vertices: $o_1 \leq o_2$ if there is a path from o_1 to o_2 . Properties of DAGs guarantee there is always at least one minimum and maximum. Finding which operations in an MCG can be executed

is a matter of taking the graph's minimum; the nodes that do not have incoming edges and thus no dependencies. By repeatedly taking the graph minima and removing this set from the original graph we can decompose the graph into a sequence of sets containing nodes that can be executed simultaneously. We refer to this as a *schedule*, but is similar to the notion of *flow* [37] in measurement-based quantum computing can in other formal contexts be described as *greedy decomposition into anti-chains, layers of minima* or *skyline* [168]. This is similar to taking the topological sort of a DAG, which gives a possible total ordering of nodes, although several of such orderings can exist. While a topological sort describes an execution sequence, a schedule describes the parallel execution of an MCG; execution can be split in several rounds where in each round the machine can execute multiple operations simultaneously. The longer the schedule the more rounds required, the larger each operation set in the schedule, the more operations can be performed in parallel.

Requirements

Checking the first requirement, it is clear the MCG is *executable*, although one needs to resolve dependencies first by computing a topological sort of the graph for sequential execution or a schedule for parallel execution.

The MCG is *not state-oriented*. The MCG execution scheme is operation-driven; An operation can be executed only if all operations it has dependencies on have been executed.

The abstract machine we described for executing the MCG model can execute operations in parallel. However, this only holds on a physical measurement-based quantum computer. In a virtual execution environment, the concrete operators realizing the quantum operation need to operate one after the other, as they potentially modify the entire quantum state. As we target a classical simulation environment, the MCG *does not satisfy the parallel requirement* for our purpose.

Conclusion

The Measurement Calculus Graph is an intuitive way to express and compose MC patterns, it contains all required information to compute its patterns and has desirable practical and theoretical properties in a quantum computing environment. Below we show how we transform the MCG graph to different models that bring the expressed quantum computation closer to a representation that addresses our requirements. The first of such transformations still results in a coarse grained model; a node represents a complete operation on the quantum state. What does change is the process driving the execution of such operations. State is made explicit in the form of state nodes and the operation-driven execution of MCG is transformed into a state-driven one.

5.2.2 Coarse Graph Model

Overview

As we have seen above, quantum state is implicit in the MCG model. The Coarse Graph (CG) brings the quantum state to the foreground by using *state nodes* in addition to *operation nodes*. Naturally, the transformation from a MCG computation into a CG representation has to preserve its semantics. Concretely, the effect of firing an operation node in CG is still the transition $q \rightarrow q'$, but quantum states q and q' are now explicitly represented in the graph with state nodes. An operation node $\boxed{\sigma}$ in MCG thus becomes the chain $\textcircled{q}\text{--}\boxed{\sigma}\text{--}\textcircled{q'}$ in the Coarse Graph. Firing an operation node has the effect to *consume* the state in its input node and *produce* the state in its output node. Edges in the CG thus describe a produce/consume relationship of quantum state. Because the MC semantics only has the one global quantum state, a graph constructed in the above way would create a long sequential chain.

On the level of operators the most obvious parallelism is the simultaneous execution of quantum operations on separable states. Many of the parallel QC simulators in literature make use of this parallelism, sometimes even exclusively [151]. If a quantum state is factorizable, e.g. a q can be expressed as $q = q_1 \otimes q_2$, an operation A on q_1 and another operation B on q_2 can be truly performed independently and thus in parallel. That is,

$$(A \otimes B)q = (A \otimes B)(q_1 \otimes q_2) = Aq_1 \otimes Bq_2 = q_1' \otimes q_2' .$$

We already capitalized on factorizable states in Chapter 3 to reduce the storage requirement for quantum states. There, we introduced the concept of *tangles* in Section 3.5.4 in order to represent a factorizable quantum state $q = q_1 \otimes q_2$ as a set of states $\{q_1, q_2\}$. On such separate states, it follows that with

$$Aq_1 \otimes Bq_2 = q_1' \otimes q_2' = q' ,$$

we have

$$(A \otimes B)q = Aq_1 \otimes Bq_2 = \{Aq_1, Bq_2\} = \{q_1', q_2'\} = q_1' \otimes q_2' = q' .$$

Factorizable states can thus be modeled in the graph by using multiple state nodes. Each state node contains a disjoint part of the global quantum state space. Doing this requires keeping track of which qubits are in which factor state. We keep track of factorizable quantum states using *tangles*, as already discussed in Section 3.5.4. The amended operational semantics from Equations (3.5.3) and (3.5.4) show that the merging and shrinking of tangles can be determined statically, looking only at the operation name and target qubit.

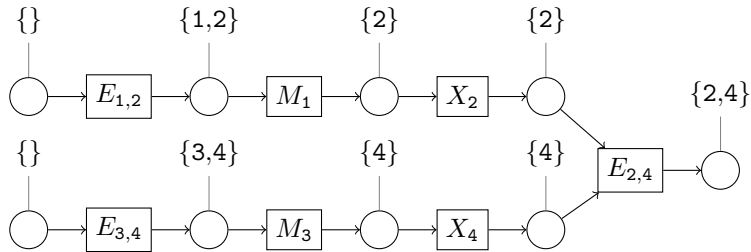
Thus, using information from the MCG, it is possible to determine how each intermediate state can be separated in multiple tangles.

To conclude, the Coarse Graph makes the quantum state explicit. In the MCG, this state was global and implicit, forming a graph in which nodes contained only one type of object: commands. The CG adds an explicit state, which also introduces an explicit execution order. The CG is *executable* and its execution is *state driven*. When applicable, multiple operations can be performed in parallel. This type of parallelism exhibited by the CG is very coarse however, relying on the nature of the computation.

Taking the example wild (non-standardized) command sequence

$$E_{24}X_4^{s_3}M_3X_2^{s_1}M_1E_{34}E_{12}$$

and following the construction process described below in Section 5.2.2 results in the following Coarse Graph.



Construction

Definition 6. *The Coarse Graph or CG is a directed bipartite graph over two types of nodes: states S and operations O .*

$$CG = \langle O, S, A \rangle \tag{5.2.1}$$

$$\text{where } A \subset (S \times O) \cup (O \times S) \tag{5.2.2}$$

State nodes contain computational state (a tangle) and operation nodes contain a transition function (an MC operation). An arc from a state to an operation node defines a *consume* relationship and from operation to state a *produce* relationship. By construction every state node has exactly one incoming and one outgoing arc, operation nodes also have only one output but certain nodes can have two inputs.

The coarse graph is constructed by transforming an MCG using an inductive process. We first define some notation and auxiliary functions to simplify presentation.

Proposition 1. *Merging two graphs unifies the sets of nodes and edges.*

$$CG_1 \cup CG_2 = \langle O_1, S_1, A_1 \rangle \cup \langle O_2, S_2, A_2 \rangle \quad (5.2.3)$$

$$= \langle O_1 \cup O_2, S_1 \cup S_2, A_1 \cup A_2 \rangle \quad (5.2.4)$$

State nodes are denoted s and operations o . A state node is a tangle, which in turn contains a qubit set and quantum state; during construction we often equate state nodes with qubit sets out of notational convenience, ignoring the tangle's quantum state component. For instance $s = \{\mathbf{i}, \mathbf{j}, \dots\}$ denotes a state node tangle $T_{\mathbf{i}, \mathbf{j}, \dots} = (q, \{\mathbf{i}, \mathbf{j}, \dots\})$. Similarly, $s' = s \cup \{\mathbf{k}, \mathbf{l}\}$ is a state node whose tangle contains the qubit sets merge of s 's tangle and $\{\mathbf{k}, \mathbf{l}\}$.

Initially the coarse graph is $CG_0 := s_{input}$, where s_{input} contains the tangle with all input qubits. The CG is constructed iteratively by walking through the MCG operations in topological order. For single qubit operations, the graph CG_i at iteration i is given by

$$CG_i := CG_{i-1} \cup \langle \{o_i\}, \{s_{out}\}, \{(s_{in}, o_i), (o_i, s_{out})\} \rangle \quad (5.2.5)$$

where at each iteration s_{out} is a newly created state node and o_i is the operation node for the operation currently visited by the iteration step. Furthermore, $s_{in} \in G_{i-1}$ and s_{in} contains the qubits used by the operation node o_i . The fresh state node s_{out} contains the same qubit set as s_{in} , except when the operation node is either a measurement or qubit creation operation, as described in Equations (3.5.3) and (3.5.4). When o is an entanglement operator such as $E_{\mathbf{i}, \mathbf{j}}$, it is possible that there is no single $s_{in} \in G_{i-1}$ such that $\mathbf{i}, \mathbf{j} \in s_{in}$. In this case there are two distinct state nodes $s_i, s_j \in G_{i-1}$ where $s_i = \{\mathbf{i}, \dots\}, s_j = \{\mathbf{j}, \dots\}$ and $s_{out} := s_i \cup s_j$.

Abstract Machine

The abstract machine for the coarse graph works by the same process as the MCG's. An execution step also works using two phases: *selection* and *application*. Selection determines which operation nodes can be executed by checking if they have no dependencies left. In the application phase, selected operation nodes are *fired*, causing their associated operation to be applied to the consumed state and in turn producing a new state. The application of operations is according to MC's operational semantics, but with the amended rules from Section 3.5.4 to work on tangle states. When multiple operation nodes are selected, all these operations can be executed simultaneously.

Conclusion

The CG is obtained by transforming the more operation-oriented model of the MCG into a state-oriented model. CG is, as the name suggests, a coarse grained

model of computation, exposing parallelism on the level of operator application. This parallelism depends on the nature of the computation, it will do well if there are large factorisable states in the computation. Like the MCG, CG still abstracts over how the quantum state and operations are modeled, leaving open if some underlying implementation would rather use stabilizer calculus¹, density matrix or state vector representation. Abstract models such as the CG are typically used as intermediate representations; we use it as such during compilation in our implementation in Chapter 6, never directly executing the coarse graph.

More parallelism is available when the model commits to a specific representation, by analysing and modeling what happens inside the individual operators. In the next sections we commit to the state vector representation, where quantum state is represented as a vector of complex amplitudes. A state node in the coarse graph representing a complete quantum state is refined into a multitude of state nodes representing individual amplitudes. Similarly, an operation node in CG is transformed into a number of amplitude-transforming operation nodes in the finer-grained graph.

5.2.3 Positional Coarse Graph (pCG)

Overview

The positional Coarse Graph or pCG makes a new implementation aspect explicit: taking into account the specific order in which a quantum state has been composed and is operated on. As we will see, this has an impact on the efficiency of the amplitude-vector implementation. For operators in particular, if an operation targets the 'last' qubit, e.g. 2 in $\{1, 2\}$, that operation can be computed in an embarrassingly parallel way. Moreover, by changing the state composition prior to an operation, any operation can target that last qubit.

The purpose of the pCG is to reflect the implementation-level concern that the same MC operation can be simulated more efficiently when the composition order of qubits is changed. If the pCG wants to take advantage of this efficiency, it will need to change the composition of qubit states. The pCG therefore imposes an explicit ordering on the composition of any qubit state. More practically, tangles in pCG impose a composition order by using an ordered qubit set or qubit list without repetition [...] rather than a qubit set {...}. This change allows to express more easily that for example qubit i is 'last' position and that operation X_5 is applied to the 'last' qubit of the ordered tangle $T_{135} = (q, [1, 3, 5])$, which makes for a more efficient computation than applying the same operation on T_{351} . Such a positional approach is something typically not considered in QC or MC, the pCG model therefore needs to introduce new nomenclature, naming and

¹Using the correct restrictions on the MC operations.

operations: ordered tangle, qubit position, positional operators, parallel position and position change operation. We show these before showing how positions affects efficiency in Section 5.2.3.

Notation

Ordered tangles are a simple derivation from tangles as used in the CG state nodes. Only a small semantic change is necessary: the qubit set is associated with a certain order. The tangle semantics as defined in Section 3.5.4 remain mostly unchanged, only now the order in which qubits are added matters, for example when merging two tangles during an entanglement operation. To make the distinction clear between ordered and unordered tangles we use a qubit list rather than a qubit set in both notation and semantics. When two tangles are merged, their qubit lists are concatenated. The order in which they are merged is now important; for example

$$\begin{array}{ll}
 T_1 \otimes T_{35} & T_{35} \otimes T_1 \\
 = (q_a, [1]) \otimes (q_b, [3,5]) & = (q_b, [3,5]) \otimes (q_a, [1]) \\
 = (q_a \otimes q_b, [1] + [3,5]) & = (q_b \otimes q_a, [3,5] + [1]) \\
 = (q_a \otimes q_b, [1,3,5]) & = (q_b \otimes q_a, [3,5,1]) \\
 = T_{135} & = T_{351}
 \end{array}$$

Composing tangles implies concatenation (using +). We call the difference between both such tangles a difference in qubit *composition* or *ordering*.

Qubits can be referred to by using their *position* in the qubit list. For example the qubit 5 is said to be in a different position in tangle T_{135} than in T_{351} .

Definition 7. *The position of a qubit named i in a qubit list Q is denoted by*

$$pos(i, Q) = \underline{i}$$

where \underline{i} is the index or ordinal for i 's position in the list.

For example $pos(5, [3,5,1]) = 2$, which we sometimes also write as $\underline{5} = 2$ when the qubit list is clear from the context. A position is thus a way to identify a qubit by its place in a tangle.

The MC is defined on a level of abstraction where the specific ordering or composition of the qubits does not matter, as it is handled by MC's notational conventions; qubits are referred to by name, and qubit subscripts are used to disambiguate combined systems such as $_1 \otimes q_3$. A differently combined $q_3 \otimes q_1$ still describes the same state; the specific order for such combined systems does not really matter, *as long as there is a consistent way to refer to qubits*. The

subscript qubit notation added to the vector state is a way to do this, another way is to maintain an ordering on qubits in such composite systems. An MC operation targets a qubit name rather than a qubit position, but the (implicit) state vector implementation of this operation depends on the target qubit's position. For example X_5 acting on T_{135} in the CG is implemented by the matrix operator $I \otimes I \otimes X$. The same X_5 acting on the differently composed tangle T_{351} is implemented by a different matrix operation: $I \otimes X \otimes I$. We therefore work in the pCG with operations that target qubits by their position, in what we call *positional operators*.

Definition 8. A positional operator U_i^n is a single-qubit operation acting on the i -th qubit of a quantum state of size n where

$$U_i^n = I^{\otimes i-1} \otimes U \otimes I^{\otimes n-i} . \quad (5.2.6)$$

Definition 9. The positional operator $\wedge Z_{i,j}^n$ is the two-qubit controlled-Z operator defined in terms of U_i^n as follows.

$$\wedge Z_{i,j}^n = |0\rangle\langle 0|_i^n + |1\rangle\langle 1|_i^n Z_j^n \quad (5.2.7)$$

In other words, a positional operator uniquely defines a matrix operator. For example $X_2^3 = I \otimes X \otimes I$ and $X_3^3 = I \otimes I \otimes X$. The positional entanglement operation is derived as a combination of single-qubit positional operators starting from

$$\wedge Z = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes Z$$

where by decomposing the tensor $|1\rangle\langle 1| \otimes Z = (|1\rangle\langle 1| \otimes I)(I \otimes Z)$ we get

$$|0\rangle\langle 0|_1^2 + |1\rangle\langle 1|_1^2 Z_2^2$$

which generalizes to n qubits yielding Equation (5.2.7).

The relation between an MC operation and it's positional equivalent is as follows.

Proposition 2. Any MC operation U_i applied to an ordered tangle $T_Q = (|\psi\rangle, Q)$ defines an equivalent positional operator

$$U_i T_Q = U_i^{|\mathbf{Q}|} T_Q \quad (5.2.8)$$

where $\underline{i} = \text{pos}(i, Q)$ the position of qubit i in Q .

In other words, every MC operation maps to a positional operator; which positional operator, depends on the position of the target qubit in the tangle.

For example, there exist permutations of qubit list $Q = [1,3,5]$ that map the MC operation X_5 respectively to X_1^3 or X_2^3 , rather than X_3^3 . In what follows we show why the latter is actually a more efficient operation than the former. Then we introduce a way to change the ordering of a single tangle, which is used in the pCG to change every positional operator to the efficient last-position version, which we call the parallel position.

Parallel Position

Single-qubit unitary operations have a simple structure that makes them easier to compute than general unitary operations on a state of the same dimensions. In the specific case of a last position operator $U_{\mathbf{n}}^{\mathbf{n}} = I^{\otimes n-1} \otimes U$ the structure of the operator exhibits a repeating pattern which makes it highly suitable for parallel computation. This can be demonstrated by showing the effect of such an operator on the amplitude vector. Taking the notational convention that $N = 2^n$ is the number of amplitudes in a state vector of n qubits; expanding the matrix operator on the amplitude-level shows the parallel-position operators can be expressed as a matrix in diagonal form:

$$U_{\mathbf{n}}^{\mathbf{n}} = (I^{\otimes n-1} \otimes U) = \begin{bmatrix} U & & \\ & U & \\ & & \ddots \end{bmatrix} = \text{diag}(U, \dots, U) \quad (5.2.9)$$

such that applying it to a state vector gives

$$U_{\mathbf{n}}^{\mathbf{n}}|\psi\rangle = \begin{bmatrix} U & & \\ & U & \\ & & \ddots \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \dots \\ \alpha_{N-2} \\ \alpha_{N-1} \end{bmatrix} = \begin{bmatrix} U \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} \\ U \begin{bmatrix} \alpha_2 \\ \alpha_3 \end{bmatrix} \\ \dots \\ U \begin{bmatrix} \alpha_{N-2} \\ \alpha_{N-1} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \alpha'_0 \\ \alpha'_1 \\ \alpha'_2 \\ \alpha'_3 \\ \dots \\ \alpha'_{N-2} \\ \alpha'_{N-1} \end{bmatrix}. \quad (5.2.10)$$

From a computing point of view an operation $I^{\otimes n} \otimes U$ can thus be seen as repeating 2^n times the application of U on contiguous parts of the input. The same holds for any unitary of any size as long as it can be expressed as $I^{\otimes n} \otimes U$. Other positional operators can in fact also be express in a diagonal form by considering

$$U_i^{\mathbf{n}} = I^{\otimes i-1} \otimes U \otimes I^{\otimes n-i} = I^{\otimes i-1} \otimes (U \otimes I^{\otimes n-i}),$$

such that indeed

$$\begin{aligned} U_{\mathbf{i}}^{\mathbf{n}} &= \text{diag}(U \otimes I^{\otimes n-i}, \dots, U \times I^{\otimes n-i}) \\ &= \text{diag}(U_{\mathbf{1}}^{\mathbf{n}-\mathbf{i}}, \dots, U_{\mathbf{1}}^{\mathbf{n}-\mathbf{i}}) \end{aligned} \tag{5.2.11}$$

where $U_{\mathbf{1}}^{\mathbf{n}-\mathbf{i}}$ is repeated 2^{i-1} times. It follows that the longer the distance of \mathbf{i} with respect to the last position n , the fewer the applications and the larger matrix operation. In other words, the computation has a repeating block structure, which is smallest for the positional operator $U_{\mathbf{n}}^{\mathbf{n}}$.

In an actual implementation, it is not always straightforward to determine which positional operator makes for a more efficient implementation than another. For example $2^{12} = 4096$ parallel applications of a 2×2 U operator might be too much parallelism for a computer with only two processing units. Many practical efficiency factors come into play: computer architecture, memory management, parallel strategy, etc. Because of our requirement to expose as much parallelism as possible, the pCG considers the last position $U_{\mathbf{n}}^{\mathbf{n}}$ to be the more efficient *parallel position*. It is trivial however to change the parallel position in the following discussion to a different qubit position depending on the context. To reiterate, the purpose of the pCG is to take advantage of the fact that the parallel position makes for a more efficient implementation. We now introduce an operation that changes the qubit ordering within a single tangle, so as to allow the pCG to express every positional operator as acting on the parallel position.

Changing Positions

Consider tangles $(|\psi\rangle, [1,2])$ and $(|\phi\rangle, [2,1])$ representing the same quantum state, but with a different qubit ordering, e.g. $|01\rangle + |00\rangle$ and $|10\rangle + |00\rangle$. As mentioned before, the difference between amplitude vectors $|\psi\rangle$ and $|\phi\rangle$ is the selected ordered basis for the vector representation. In other words, the column vector for $|\psi\rangle$ has the same entries as $|\phi\rangle$, only the positions in which they appear is different. There is thus a permutation matrix operator P such that $P|\phi\rangle = |\psi\rangle$ and its effect on the level of individual amplitudes is elaborated on in more detail in Section 5.3.4. On the level of tangles, we can thus introduce an operation

$$\begin{aligned} \mathbb{P}(T_Q) &= \mathbb{P}(|\psi\rangle, Q) \\ &= (P|\psi\rangle, \sigma Q) \end{aligned} \tag{5.2.12}$$

that changes a tangle's qubit order. Using the same example, there is an operation \mathbb{P} such that $\mathbb{P}(|\psi\rangle, [1,2]) = (P|\psi\rangle, \sigma[1,2]) = (|\phi\rangle, [2,1])$, with σ a qubit list permutation.

Whereas it is possible to do arbitrary reordering, there are good reasons to limit ourselves to a very specific type of permutation. Permutations of the qubit

list would be typically expressed as either series of *transpositions* or *circular shifts*; swapping two elements or shifting all elements a number of places respectively. Naturally, reordering the qubit list itself to put an arbitrary qubit at a specific position is not the computationally intensive part: reshuffling the entire amplitude vector is. Any change to the qubit list requires reshuffling the entire amplitude vector using a permutation operator P , so we have to look at the intrinsic properties of such P for each case. The circular shift of the qubit list uses a certain type of permutation operators that exhibits a recursive block structure that make it highly suited to an efficient and parallel implementation [117]. We will also see in Section 5.3.4 that this results in an elegant amplitude-level formulation of the permutation operator. The type of permutation matrix operators associated for a transposition of two qubits can also be recursively decomposed. Although this type of permutation operator leaves half of the amplitudes in place, it requires more parallel steps after its recursive decomposition. The two different types of permutation operators are visually compared with a simple example in Figure 5.2. Both permutations decompose in the same smallest possible local permutation: swapping two neighbouring position $\sigma_{i,i+1}$. However, the operator for a transposition permutation decomposes in more stages than the transposition. Expressing a swap of positions $\sigma_{i,j}$ is composed of two 'passes' of swaps, from i to j and back, whereas a cyclic shift σ_i is composed of a single 'pass' of swaps. Another reason to restrict ourselves to cyclic shifts of the qubit list is the frequent use of its type of permutation matrix operator in literature. To mathematicians and card-shuffling magicians² it is known as the *generalized perfect shuffle* [69] or *faro shuffle*. In multilinear algebra and statistics the same permutation is known as the *vec-permutation matrix* [106] or *(tensor-)commutation matrix* [136]. Engineers also call it *stride permutation* and use it for parallel processing algorithms of the Fourier transform [154, 115, 117], parallel divide-and-conquer matrix transpositions [190], or as a way to interconnect computer components in parallel [177].

By the definition of a tangle, some $T_Q = (q, Q)$ represents $q \in \mathfrak{h}_Q$. Consider $\mathfrak{h}_{K+L} = \mathfrak{h}_K \otimes \mathfrak{h}_L$ for any K and L partitions of Q . The circular shift $\sigma_{|K|}Q = \sigma_{|K|}(K + L) = L + K$ represents at the qubit level the effect of the tensor-commuting operation P on q 's ordered basis, achieving $Pq \in \mathfrak{h}_{L+K}$. Put plainly, the circular shift $\sigma_{\mathbf{k}}$ describes the position changes achieved by taking the first k -number of qubits and appending them behind the rest, such that element originally on position k finds itself in last position. More formally, this operation is described as follows.

Definition 10. *The circular shift operation $\sigma_{\mathbf{k}}$ maps each of the n positions in*

²In no way are the sets of mathematicians and magicians disjunct [68].

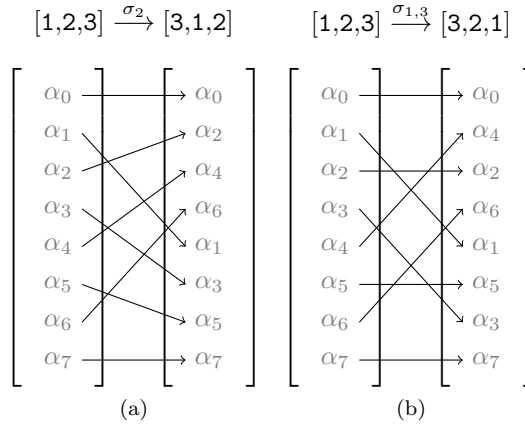


Figure 5.2: Comparing the permutation operators for reshuffling amplitude positions in case of a cyclic shift (a) and transposition (b).

a qubit list to a new position, such that for each position i is mapped to position

$$\sigma_{\mathbf{k}}(\mathbf{i}) \equiv i - k \pmod{n} .$$

That is, a circular shift of all elements to the left by k places.

For example $\sigma_3 [1,3,5,7] = [7,1,3,5]$ puts the third qubit last by circular shifting the complete list to the left by three places. Naturally, multiple circular shifts can be composed

$$\sigma_{\mathbf{k}}\sigma_{\mathbf{l}} = \sigma_{\mathbf{k+l}} \tag{5.2.13}$$

and decomposed

$$\sigma_{\mathbf{k}} = \sigma_{\mathbf{1}}^k . \tag{5.2.14}$$

Similarly, the inverse of a circular shift is always

$$\sigma_{\mathbf{k}}^{-1} = \sigma_{\mathbf{n-k}} \tag{5.2.15}$$

when applied to a qubit list of length n . In conclusion, we restrict ourselves with good reason to qubit position changes that can be achieved for the amplitude-vector by the tensor-commuting permutation as described in Section 5.3.4.

Proposition 3. *The position changing operation $\mathbb{P}_{\mathbf{k}}$ changes the composition of a tangle such that*

$$\mathbb{P}_{\mathbf{k}}(q, Q) = (Pq, \sigma_{\mathbf{k}}Q) \quad (5.2.16)$$

where $q' \in \mathfrak{h}_{\sigma_{\mathbf{k}}Q}$.

Like the circular shift, this operation is also (de-)composable

$$\mathbb{P}_{\mathbf{k}}\mathbb{P}_{\mathbf{l}} = \mathbb{P}_{\mathbf{k}+\mathbf{l}} = \mathbb{P}_{(\mathbf{k}+\mathbf{l}) \bmod \mathbf{n}} \quad (5.2.17)$$

and its inverse is easily shown to be

$$\mathbb{P}_{\mathbf{k}}^{-1} = \mathbb{P}_{\mathbf{n}-\mathbf{k}} \quad (5.2.18)$$

for a qubit list length n . We will often use the notation $\mathbb{P}_{\mathbf{i}}$, which uses the notation $\underline{i} = \text{pos}(i, Q)$ introduced earlier. $\mathbb{P}_{\mathbf{i}}$ denotes the position changing operation that moves qubit \mathbf{i} to the last position.

Targeting positional operators

The reason for changing qubit positions is to express arbitrary positional operators as parallel position ones. This serves the obvious purpose of expressing the computation in a more efficient and parallel form on the one hand and on the other it makes for a simpler and more uniform expression of qubit operations in the context of the pCG. The operators for $U_{\mathbf{1}}^{\mathbf{3}}$, $U_{\mathbf{2}}^{\mathbf{3}}$ and $U_{\mathbf{3}}^{\mathbf{3}}$ are all different. Similarly, all operators $U_{\mathbf{i}}^{\mathbf{n}}$ for a given \mathbf{i} vary with the size \mathbf{n} . However, the parallel position $U_{\mathbf{n}}^{\mathbf{n}}$ differs only in the number of applications of operator U . In other words, it is simpler to build a hypothetical machine implementing only $U_{\mathbf{n}}^{\mathbf{n}}$ because it simply needs to repeat U a variable number of times, while supporting arbitrary $U_{\mathbf{i}}^{\mathbf{n}}$ requires implementing a large range of different operations.

Our first step in making MC operations more concrete was to turn an MC operation $U_{\mathbf{i}}$, an operation on a qubit with name \mathbf{i} , into a positional operator $U_{\mathbf{i}}^{\mathbf{n}}$ in Proposition 2. Such operator still performs two tasks: looking up the qubit position and applying the appropriate positional operator. Here, we use the positional changing operator to separate out this lookup and to exclusively use the parallel position operator $U_{\mathbf{n}}^{\mathbf{n}}$.

Given a certain tangle T_Q there is always a position change operator $\mathbb{P}_{\mathbf{i}}T_Q = T_{\overline{Q}}$ such that

$$\text{pos}(\mathbf{i}, \overline{Q}) = \mathbf{n} , \quad (5.2.19)$$

from which follows, using Propositions 2 and 3, that

$$U_{\mathbf{i}}T_{\overline{Q}} = U_{\mathbf{n}}^{\mathbf{n}}T_{\overline{Q}} \quad (5.2.20)$$

$$= U_{\mathbf{n}}^{\mathbf{n}}\mathbb{P}_{\mathbf{i}}T_Q , \quad (5.2.21)$$

keeping the MC operations on the left hand side and their concretization on the right. Some operations have no influence on the composition of the quantum state, for example correction operations do not modify the qubit list. This means the position change on the left hand side of Equation (5.2.21) can be moved to the right hand side using its inverse.

$$U_i T_Q = \mathbb{P}_i^{-1} U_n^n \mathbb{P}_i T_Q \quad \text{by (5.2.18)} \quad (5.2.22)$$

$$= \mathbb{P}_{n-i} U_n^n \mathbb{P}_i T_Q \quad (5.2.23)$$

A MC operation on a qubit i can thus be replaced by a parallel position operator, if the qubit positions are shifted such that i finds itself at the last position, then doing the position shift to put qubit i in its original place. This yields a useful equation between two versions of the same correction operation; the MC operation on the left hand side and the more concrete parallel positional operator on the right.

Proposition 4.

$$U_i = \mathbb{P}_{n-i} U_n^n \mathbb{P}_i \quad (5.2.24)$$

for $U_i \in \{X_i, Z_i\}$.

The measurement operation does alter the qubit list by removing a qubit. Taking \mathbb{P}'_i to be \mathbb{P}_i operating on a smaller qubit set $Q \setminus [i]$, we can derive

Proposition 5.

$$M_i = \mathbb{P}'_i{}^{-1} M_n^n \mathbb{P}_i. \quad (5.2.25)$$

The case for the entanglement operator is more complex due to its two-qubit nature. The parallel position operator for the entanglement operation is $\wedge Z_{n-1, n}^n$, requiring the two target qubits to be in the last two positions. Under a cyclic shift the relative distance between two qubit positions remains the same. For example, take two qubits 1 and 5 that are two positions apart in $Q = [1, 3, 5, 7]$, there is no σ_k that will put both qubits next to each other. This can be achieved by applying the cyclic shift on only a part of the qubit list.

Proposition 6. *The partial shift operation $\sigma_{k,l}$ leaves the first k qubits in place and performs a circular shift σ_l on the remaining qubits. Given a qubit list $Q = K + L$ and $|K| = k$ and $|L| = l$,*

$$\sigma_{k,l} Q = K + \sigma_l L \quad (5.2.26)$$

where $+$ appends both lists, preserving their ordering.

The inverse of the partial shift is a matter of inverting the circular shift by applying Equation (5.2.15) in the above definition, yielding

$$\sigma_{k,l}^{-1} = \sigma_{k,n-k-l} . \quad (5.2.27)$$

On the amplitude vector, the permutation matrix operator associated with the shift $\sigma_{k,l}$ is the operator $(I^{\otimes k} \otimes P_{\sigma_l})$, where P_{σ_l} is the permutation operator associated with σ_l . As before, the concrete definition of the permutation operators P are presented later in Section 5.3.4.

Proposition 7. *The partial position change operation $\mathbb{P}_{\mathbf{k},\mathbf{l}}$ changes the composition of a part of a tangle's composition*

$$\mathbb{P}_{\mathbf{k},\mathbf{l}}(q,Q) = ((I^{\otimes k} \otimes P_{\sigma_{l-1}})q, \sigma_{k,l-1}Q) \quad (5.2.28)$$

such that qubits originally in positions k and l are brought into position k and $k+1$.

For example, $\sigma_{1,1}[1,3,5,7] = [1,5,7,3]$ brings 1 and 5 together by effectively performing $[1] + \sigma_1[3,5,7]$.

The (de-)composition of this partial position change is somewhat more complex; there is no general way to compose two partial position changes into a single one. In specific situations the regular position change's (de-)composition rule can be applied, for example when the subsets align for $\mathbb{P}_{\mathbf{k},\mathbf{l}}\mathbb{P}_{\mathbf{k},\mathbf{m}} = \mathbb{P}_{\mathbf{k},\mathbf{l}+\mathbf{m}}$. From the amplitude-vector implementation perspective this partial shift also performs a complete reshuffling of amplitude positions. The reshuffle pattern for the partial shift is in some ways more complex as the regular circular shift; for example two partial shuffles do not always compose into a single partial shuffle. We therefore use the partial shift only when it is necessary.

To formulate the MC entanglement operation $E_{\mathbf{i},\mathbf{j}}$ in terms of its parallel position operator $\wedge Z_{\mathbf{n}-\mathbf{1},\mathbf{n}}^{\mathbf{n}}$, we first need to bring the qubits \mathbf{i} and \mathbf{j} to adjacent position:

$$E_{\mathbf{i},\mathbf{j}} T_Q = \mathbb{P}_{\mathbf{i},\mathbf{j}}^{-1} \wedge Z_{\mathbf{i},\mathbf{i}+\mathbf{1}}^{\mathbf{n}} \mathbb{P}_{\mathbf{i},\mathbf{j}} T_Q . \quad (5.2.29)$$

Once both qubits are brought in adjacent positions, a regular cyclic shift position change can bring them both to the last two positions. Similarly to Proposition 4, we can derive:

Proposition 8. *The entanglement operator*

$$E_{\mathbf{i},\mathbf{j}} T_Q = \mathbb{P}_{\mathbf{i},\mathbf{j}}^{-1} \mathbb{P}_{\mathbf{i}+\mathbf{1}}^{-1} \wedge Z_{\mathbf{n}-\mathbf{1},\mathbf{n}}^{\mathbf{n}} \mathbb{P}_{\mathbf{i}+\mathbf{1}} \mathbb{P}_{\mathbf{i},\mathbf{j}} T_Q \quad (5.2.30)$$

In summary, each MC operation as they appear in the CG can be implemented as a series of position changing operations and parallel position operators. These handful of operations are chosen out of consideration for implementation efficiency. In the following we use these transformations to describe the construction of the pCG, transforming a CG's operation node into chains of efficient position changing and parallel position operations.

Construction

The construction of the pCG can be simply described as a transformation over each node of a CG. The structure of the graph remains similar, nodes only get expanded into a sequence of new operation and state nodes. The semantics and equations used for these expansions were covered in the above text. We break down the construction process in two phases: expansion and contraction.

The first step in the expansion phase is to choose an arbitrary order for all input state nodes, using an ordered tangle with a qubit list rather than a tangle with a qubit set. Note, what we call a qubit list is rather an ordered set, repetition is not allowed. Input state nodes are nodes that have no incoming edge and typically contain only one qubit. This step is to allow for arbitrary size input states. Other state nodes inherit their order from an upstream state node during the rest of the expansion phase. The transformation step is the second in the expansion phase, wherein every operation node is expanded into a chain of state and operation nodes. An operation node in the pCG contains either a parallel positional operator $\in \{X_n^n, Z_n^n, M_n^n, \wedge Z_{n-1,n}^n\}$, a position changing operation $\in \{\mathbb{P}_k, \mathbb{P}_{k,1}\}$ or a state merging operation \otimes . The merge operation \otimes was implicit in the CG, relying on the entanglement operation to merge two tangle nodes. We make it explicit here as part of the operation concretization effort. Each operation node in the CG contains either a correction, measurement or entanglement MC operation. Every such CG operation node is replaced by a chain of pCG nodes, a sequence of connected operation and state nodes. A correction operation is transformed using the rule visually represented in Figure 5.3 and makes use of Proposition 4, wrapping position changing operations around the parallel position operator. Note that the inverse position change can be denoted as both \mathbb{P}_i^{-1} or \mathbb{P}_{n-i} , we find the former to be more informative in the context of the transformation rules. Measurement operations use a similar rule in Figure 5.4, based on Proposition 5 and where $\mathbb{P}'_i^{-1} = \mathbb{P}_{n-i-1}$.

The case for the entanglement operation is somewhat more complex. Some entanglement operations perform a merge of two existing states, in which case the two input qubit lists are appended in a consistent way, as demonstrated in Section 5.2.3. The position changing operations are logically applied to the result of this merge, requiring the pCG to perform the merge using an explicit operation node, rather than seeing the merge as an implicit part of the entanglement

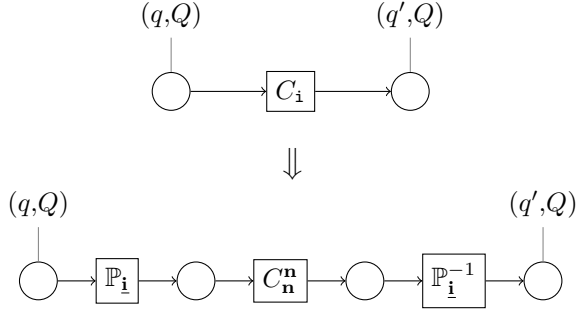


Figure 5.3: Transformation rule for correction operations $C_i = X_i$ or Z_i .

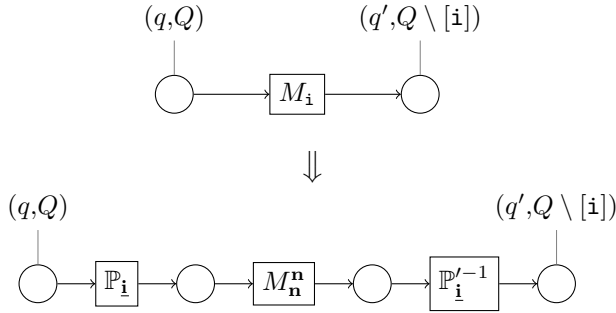
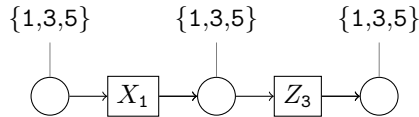


Figure 5.4: Transformation rule for measurement operation M_i .

operation. A simple expansion as shown in Figure 5.5 paves the way for the entanglement transformation rule in Figure 5.6, based on Proposition 8 and where $\mathbb{P}_{i,j}^{-1} = \mathbb{P}_{i,n-i-j}$.

Expanding all CG's operation nodes produces a large number of position changing operations. The second phase of construction seeks to contract these where obvious. Using as an example the following CG



which is transformed using the expansion rules into a pCG

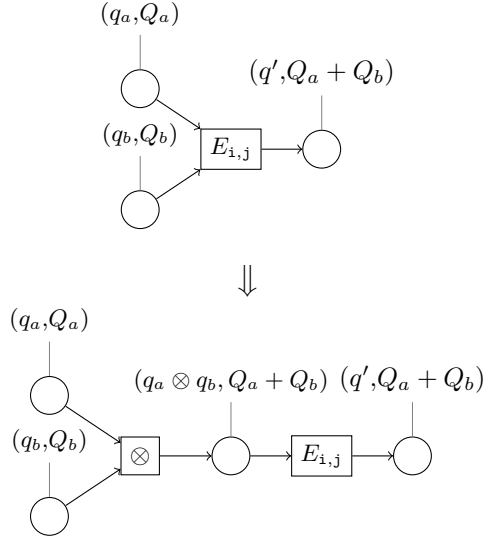


Figure 5.5: Introducing an explicit merge operation node for two-input entanglement nodes.

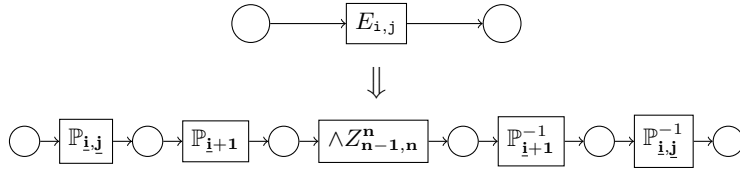
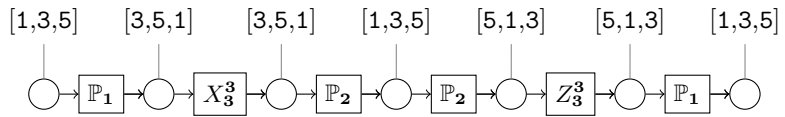
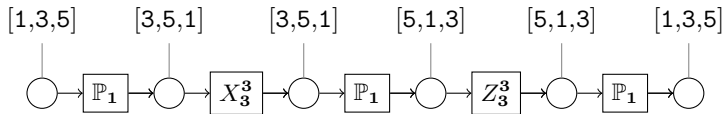


Figure 5.6: Transformation rule for entanglement operation $E_{i,j}$.



where by composing $\mathbb{P}_2\mathbb{P}_2 = \mathbb{P}_1$ by Equation (5.2.17) we arrive at the following contracted pCG.



This contraction rule is given more formally in Figure 5.7. The contraction phase

during construction of a pCG is not strictly necessary, but it does simplify the model and reduce the graph's overall number of operations.

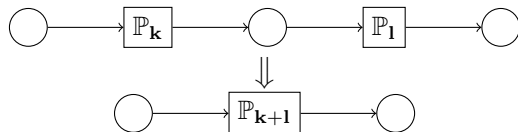


Figure 5.7: Transformation rule for contracting a chain of position change operations.

Conclusions

The positional Coarse Graph is a further refinement of the Coarse Graph, introducing elements that only make sense in an amplitude vector implementation. Where the CG dealt with a systemic change, introducing explicit and factorized state, the positional Coarse Graph focuses on refining what happens at the level of individual tangles and operations. New terminology such as positional tangles and operators gives us a vocabulary to express new concerns that stem from an underlying amplitude vector implementation. The various propositions introduced here express how to change around representations and operations to better suit implementation constraints. In this light, the pCG model acts as an intermediate qubit level language, between the MC and the abstract target computing platform. Working at the qubit-level liberates us from the specific way the underlying amplitude vector is represented and stored, but various concerns that arise across several such implementations can still be expressed in the pCG.

To come back to our requirements, the pCG results in improvements to the *executability* requirement and also greatly increases *parallelism*. From a rather abstract MC operation, we go to a concrete and executable form. The pCG reflects that MC operations are in many implementations not the same. An abstract machine for the MCG or CG needs to 'look up' the position of the target qubit of some U_i , to be able to construct a position-sensitive operator. We consider the pCG to be more *executable* because it does not need to make the conversion from name-based MC operations to positional operators, rather it makes it part of the construction process. The *parallelism* gains are a result of expressing MC operations, previously a monolithic black box operation, into a form that is known to be highly parallel. The position changing operations introduced with the pCG may appear at first glance an increase in complexity, a breach of the *simplicity* requirement. However, the explicit position changing operations serve both to elegantly express the MC operation's parallelism and to capture the difference between operations with a different target qubit.

For most purposes the pCG is already an adequate model for an efficient QCSim, ticking all boxes we put in the requirements. Most related work we discussed also remains at level of abstraction, barring some ad hoc optimizations. Even some apparent amplitude-level techniques in literature can be subsumed by the pCG. For instance, the permutation used in [56] to minimize communication, which can be expressed at the qubit level with qubit position changes. However, some amplitude-centric approaches at quantum simulation eschew the amplitude-vector altogether. The space-efficient approach by Frank et al. [80, 81] computes each amplitude by recursively recomputing its depending amplitudes. This type of strategy, among others, cannot be described by a coarse-grained model such as the pCG.

For a more complete analysis of quantum simulation computations, we have to work towards an amplitude-centric model. The following section explores such a model by further refining the pCG, creating a fine-grained graph whose state nodes contain individual amplitudes rather than complete quantum states.

5.3 Fine-grained graph model

5.3.1 Overview

All models presented so far used a coarse granularity: treating quantum state and operations as black boxes. In the fine-grained model of execution we dive into a more concrete implementation, peeling open the black box. The abstract machine for the fine-grained model executes simple arithmetic operations of a classical computer, where the level of granularity is at that of individual amplitudes. In the coarse grained graphs a quantum state $q = |\psi\rangle$ is represented by a single state node. The same quantum state in a fine-grained graph is represented by a group of state nodes, each containing a single amplitude; this is depicted in Figure 5.8.

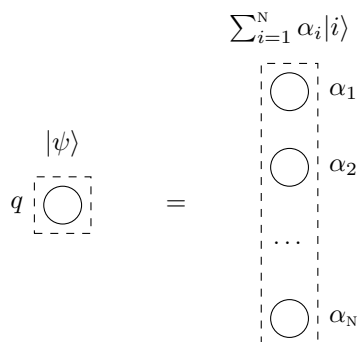
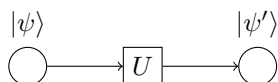
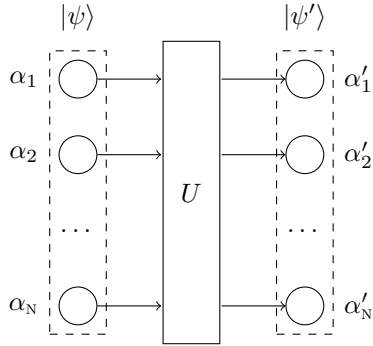


Figure 5.8: Relation between coarse- and fine-grained representation of a quantum state.

Large quantum states give rise to an exponential number of amplitudes and thus amplitude-transforming operations. This enables a fine-grained graph which exposes parallelism which is not obvious at the quantum-operator level. Put differently, a classical implementation of a single quantum operation is a collection of multiple amplitude operations; a fine-grained model of computation executes a single quantum operation in parallel by executing the amplitude operations in parallel. However, amplitude state nodes alone do not make the operations fine-grained. Indeed, the following coarse graph



with fine-grained amplitudes becomes



in which the unitary is a single operation. We base ourselves on the MC which only uses a small set of one-qubit and two-qubit operations. The properties specific to correction, measurement and entanglement operators allow us to 'cut up' the quantum operation in many smaller ones, each ideally consuming and producing only a single amplitude.

For the sake of clarity we first give an overview of each MC operation as applied to single qubits. Then, to express fine-grained quantum operations we introduce the fine-grained equivalents of the positional operators and qubit permutations.

5.3.2 Single qubit states

In the case of single qubit states the expression

$$U|\psi\rangle = |\psi'\rangle$$

expands to

$$U|\psi\rangle = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \begin{bmatrix} \alpha'_0 \\ \alpha'_1 \end{bmatrix} = |\psi'\rangle$$

and so

$$U|\psi\rangle : \begin{cases} \alpha'_0 = u_{00}\alpha_0 + u_{01}\alpha_1 \\ \alpha'_1 = u_{10}\alpha_0 + u_{11}\alpha_1 \end{cases} .$$

This computation is expressed as a graph in Figure 5.9, where operation nodes perform simple arithmetic operations and state nodes contain single amplitudes.

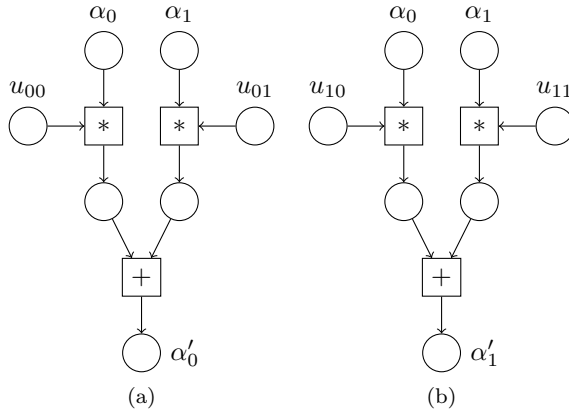


Figure 5.9: Two fine-grained graphs computing amplitudes α'_0 (a) and α'_1 (b).

Correction Operators

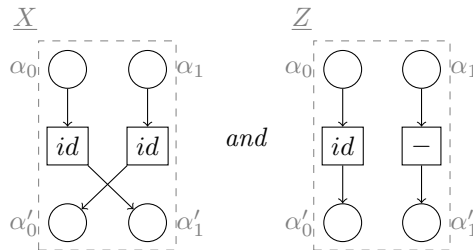
The Pauli-X and Pauli-Z operators, introduced in Chapter 2, have a much simpler structure. These operators perform the computations

$$X|\psi\rangle : \begin{cases} \alpha'_0 = 0.\alpha_0 + 1.\alpha_1 & = \alpha_1 \\ \alpha'_1 = 1.\alpha_0 + 0.\alpha_1 & = \alpha_0 \end{cases} \quad (5.3.1)$$

$$Z|\psi\rangle : \begin{cases} \alpha'_0 = 1.\alpha_0 + 0.\alpha_1 & = \alpha_0 \\ \alpha'_1 = 0.\alpha_0 - 1.\alpha_1 & = -\alpha_1 \end{cases} \quad (5.3.2)$$

which, as shown in the following proposition, require a much simpler graph than the general unitary operation.

Proposition 9. *The Pauli-X and -Z operations, applied to a single qubit $(\alpha_0|0\rangle + \alpha_1|1\rangle)$, are implemented in the fine-grained graph as respectively*



where the operation nodes containing id perform the identity operation.

Measurement

As we have seen in Chapter 2, the measurement operator M^α has two peculiarities: it is non-deterministic and destroys the target qubit. The effect on the quantum state is non-deterministic and in the case of the single-qubit $|\psi\rangle$ either

$$|\psi\rangle \xrightarrow{M^\alpha} \langle +_\alpha | \psi \rangle \tag{5.3.3}$$

$$\text{or } |\psi\rangle \xrightarrow{M^\alpha} \langle -_\alpha | \psi \rangle \tag{5.3.4}$$

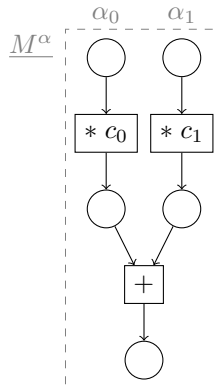
where $\langle +_\alpha | = 1/\sqrt{2} (|0\rangle + e^{-i\alpha}|1\rangle)$ and $\langle -_\alpha | = 1/\sqrt{2} (|0\rangle - e^{-i\alpha}|1\rangle)$, with probabilities depending on the contents of the quantum state. In Section 3.6.3 we have presented some of the ways used to dealing with the probabilistic measurement outcomes in a virtual execution environment. In computing terms, calculating the probability by way of computing the norm (see Equation (2.3.13)) is a *reduction* operation, which has an efficient parallel implementation and has been well-researched in the parallel computing community [28, 57]. Explicit control of the outcomes is a matter of adding control nodes or edges to the graph. For simplicity's sake we use the first way of dealing with measurements³, where the operation M^α always performs $\langle +_\alpha |$.

The effect of measurement on a single qubit quantum state is to destroy it. We see this in $\langle +_\alpha | \psi \rangle$ resulting in a scalar

$$\langle +_\alpha | (\alpha_0|0\rangle + \alpha_1|1\rangle) = \alpha_0 \langle +_\alpha | 0 \rangle + \alpha_1 \langle +_\alpha | 1 \rangle \tag{5.3.5}$$

the computation of which can be represented as a fine-grained graph.

Proposition 10. *The effect of the measurement operation on the amplitudes of one qubit is implemented by the following fine-grained graph*



³By doing this, we do restrict ourselves to MC patterns that realize unitary operations.

where $c_0 = \langle +_\alpha | 0 \rangle = \frac{1}{\sqrt{2}}$ and $c_1 = \langle +_\alpha | 1 \rangle = \frac{e^{-i\alpha}}{\sqrt{2}}$.

The computation for $\langle -_\alpha |$ only differs in the sign of the second constant $c_1 = -\frac{e^{-i\alpha}}{\sqrt{2}}$.

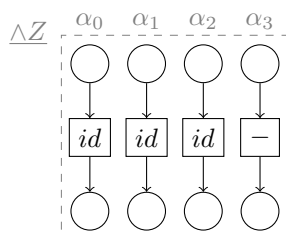
Entanglement

The entanglement operator $\wedge Z$ applied to a two-qubit state can be seen as simply switching the sign of the last amplitude

$$\begin{aligned} \wedge Z &= |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes Z \\ &= |00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 10| - |11\rangle\langle 11| \end{aligned} \tag{5.3.6}$$

which is straightforward to implement as a fine-grained graph.

Proposition 11. *The entanglement operator applied to a two-qubit state $\sum \alpha_i |i\rangle^4$ is implemented by the following fine-grained graph*

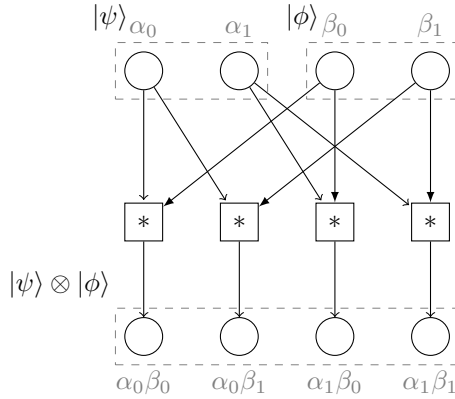


where operation nodes containing *id* perform the identity operation.

The above does not implement the complete entanglement operation however. The coarse graph introduced the concept of tangles to model separable quantum states, this means the entanglement operation carries an implicit operation combining qubit states when necessary: the tensor product. It is easy to see that the definition of the vector tensor product

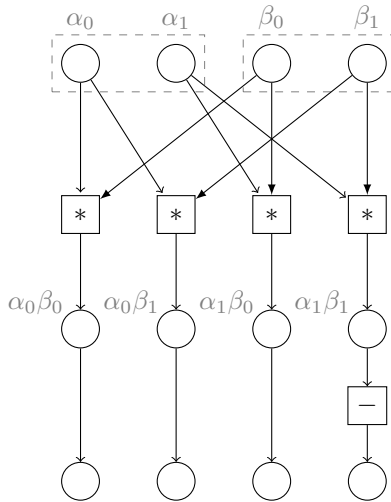
$$\begin{aligned} |\psi\rangle \otimes |\phi\rangle &= (\alpha_0|0\rangle + \alpha_1|1\rangle) \otimes (\beta_0|0\rangle + \beta_1|1\rangle) = \\ &= \alpha_0\beta_0 (|0\rangle \otimes |0\rangle) + \alpha_0\beta_1 (|0\rangle \otimes |1\rangle) + \alpha_1\beta_0 (|1\rangle \otimes |0\rangle) + \alpha_1\beta_1 (|1\rangle \otimes |1\rangle) \end{aligned} \tag{5.3.7}$$

is implemented by the following graph:



Hence, the entanglement operation applies the $\wedge Z$ after the tensor product if both target qubits have not already been combined.

Proposition 12. *The entanglement operator applied to a two-qubit tangle state $\sum \alpha_i |i\rangle \otimes \sum \beta_i |i\rangle$ is implemented by the following fine-grained graph*



where edges between state nodes imply the identity operation.

5.3.3 Notation

A word on the notation as used further in this section: because permutations only depend on the dimension of the inputs, we add it to the notation of the vector or

matrix as a superscript when the dimension of a vector is of importance but not obvious from context; $|v\rangle^N$ denotes a vector with dimension N . A quantum state of n qubits can thus be written as the sum of basis vectors $|\psi\rangle = \sum_{i=0}^{N-1} v_i |i\rangle^N$ with $N = 2^n$. We simplify the summation out of convenience to $\sum_i^N v_i |i\rangle^N$ or $\sum_i v_i |i\rangle^N$.

In a tensor product of two qubits

$$\alpha_0 \beta_0 (|0\rangle \otimes |0\rangle) + \alpha_0 \beta_1 (|0\rangle \otimes |1\rangle) + \alpha_1 \beta_0 (|1\rangle \otimes |0\rangle) + \alpha_1 \beta_1 (|1\rangle \otimes |1\rangle)$$

the last basis vector can be expressed, depending on notational convention, as $(|1\rangle \otimes |1\rangle)$, $|1\rangle|1\rangle$, $|11\rangle$ or even $|3\rangle$. For clarity's sake, we feel it is necessary to explain the origin and purpose of each notation to clearly distinguish between each in future formulations. By notational convention, all basis states of a vector space are identified by an index starting at 0, for a qubit this is trivially $|0\rangle$ and $|1\rangle$. The tensor product of two qubits creates a vector space with dimension four, thus with basis states $|0\rangle$, $|1\rangle$, $|2\rangle$ and $|3\rangle$. It is common in literature to write the tensor $(|1\rangle \otimes |1\rangle)$ as $|11\rangle$ by appending the *numerals* of both basis identifiers, which we refer to as the *numeral tensor notation*. The basis vector in the combined vector space is identified by reading out the numeral 11 as a binary number, thus $1 \cdot 2^1 + 1 \cdot 2^0 = 3$ forming the *tensor index notation* $|3\rangle$. For our purpose, the latter is more convenient, allowing us to express permutations on the state vector as a function on the tensor index number.

Proposition 13. *The tensor product of two basis vectors $|i\rangle^M$ and $|j\rangle^N$ with $i \leq M, j \leq N$ using the tensor index notation can be written as $|k\rangle^{MN}$ with $k \in \mathbb{N}^{MN}$ and $k = iN + j$, such that*

$$|i\rangle^M \otimes |j\rangle^N = |iN + j\rangle^{MN} . \tag{5.3.8}$$

5.3.4 Permutation Operator

The relation between different positional operators described in Section 5.2.3 proposes a position change operation \mathbb{P}_i on tangles, which produced a cyclic shift of the qubit list and a permutation operator P reshuffling the amplitude vector. The reason to constrain ourselves to cyclic shift position changes in the pCG was because the operator P is a specific kind of permutation operator that can lead to more efficient implementations. As already mentioned then, this permutation operator P arises from commuting the tensor product of two matrices—or vectors—under a certain ordering.

Proposition 14. *The permutation operator P commutes the tensor product of two vectors with respective dimensions M and N , such that*

$$P_{M,N} (|\psi\rangle^M \otimes |\phi\rangle^N) = |\phi\rangle^N \otimes |\psi\rangle^M \tag{5.3.9}$$

for quantum states $|\psi\rangle$ and $|\phi\rangle$ with respectively m and n qubits and with $M = 2^m$ and $N = 2^n$.

A quantum state $|\psi\rangle = \sum_i^n \alpha_i |i\rangle$ cannot in general be expressed as a tensor of two states. However, any basis vector $|i\rangle$ can be expressed as a tensor product of two basis vectors, as shown by Equation (5.3.8). Any integer can be split into a sum of its remainder and division; we know that for any integer i and n

$$i = n \left\lfloor \frac{i}{n} \right\rfloor + i \bmod n \quad (5.3.10)$$

using the integer functions as defined in L Graham et al. [131]. It thus follows from Equations (5.3.8) and (5.3.10) that any basis vector $|i\rangle^{MN}$ can be expressed as a tensor product

$$|i\rangle^{MN} = \left| \left\lfloor \frac{i}{N} \right\rfloor \right\rangle^M \otimes |i \bmod N\rangle^N \quad (5.3.11)$$

This may appear to be a convoluted way to express a basis vector tensor product, but it has a direct application for finding the permutation operation P .

We can find the amplitude-level expression for the permutation operator in $P|\psi\rangle$ using Equations (5.3.9) and (5.3.11) on the basis vectors of $|\psi\rangle$.

$$\begin{aligned} P_{M,N} |i\rangle^{MN} &= P_{M,N} \left(\left| \left\lfloor \frac{i}{N} \right\rfloor \right\rangle^M \otimes |i \bmod N\rangle^N \right) \\ &= |i \bmod N\rangle^N \otimes \left| \left\lfloor \frac{i}{N} \right\rfloor \right\rangle^M \\ &= |M(i \bmod N) + \left\lfloor \frac{i}{N} \right\rfloor\rangle^{MN} \end{aligned}$$

This results in the following definition for P as a map on basis states:

$$P_{M,N} : |i\rangle^{MN} \longrightarrow |p_{M,N}(i)\rangle^{MN} \quad (5.3.12)$$

in which p is the following function on a single index $i \in \mathbb{N}^{MN}$

$$p_{M,N}(i) = M(i \bmod N) + \left\lfloor \frac{i}{N} \right\rfloor . \quad (5.3.13)$$

One of the properties of the permutation operation is that

$$P_{N,M} = P_{M,N}^{-1} = P_{M,N}^T \quad (5.3.14)$$

It is useful to also express the permutation in an outer product notation

$$P_{M,N} = \sum_i^{MN-1} |p_{M,N}(i)\rangle \langle i| \quad (5.3.15)$$

$$= \sum_i^{MN-1} |i\rangle \langle p_{N,M}(i)| \quad \text{by (5.3.14)} . \quad (5.3.16)$$

which is applied to a state vector to find an amplitude-centric definition

$$\begin{aligned}
P_{M,N}|\psi\rangle &= \sum_i^{MN-1} |p_{M,N}(i)\rangle\langle i| \sum_j^{MN-1} \alpha_j |j\rangle \\
&= \sum_{i,j} \alpha_j |p_{M,N}(i)\rangle\langle i|j\rangle \\
&= \sum_i \alpha_i |p_{M,N}(i)\rangle \\
&= \sum_i \alpha_{p_{N,M}(i)} |i\rangle .
\end{aligned} \tag{5.3.17}$$

The same permutation operation can be used to commute a tensor of linear operators. Seeing that the permutation operator P is linear we can work out that

$$\begin{aligned}
&P_{M,N} (A^M \otimes B^N) (|\psi\rangle^M \otimes |\phi\rangle^N) \\
&= P_{M,N} (A^M |\psi\rangle^M \otimes B^N |\phi\rangle^N) \\
&= (B^N |\phi\rangle^N \otimes A^M |\psi\rangle^M) \\
&= (B^N \otimes A^M) (|\phi\rangle^N \otimes |\psi\rangle^M) \\
&= (B^N \otimes A^M) P_{M,N} (|\psi\rangle^M \otimes |\phi\rangle^N)
\end{aligned} \tag{5.3.18}$$

which implies that

$$(B^N \otimes A^M) P_{M,N} = P_{M,N} (A^M \otimes B^N) . \tag{5.3.19}$$

Here, the notation A^M denotes a square matrix of size M . Applying the inverse permutation on the right in the above Equation (5.3.19) yields the formula for commuting a matrix tensor product:

$$B^N \otimes A^M = P_{M,N} (A^M \otimes B^N) P_{N,M} . \tag{5.3.20}$$

An intuitive way of understanding the above permutations is by using a card game analogy. Imagine a dealer has a deck with MN cards, face down and ordered such that the top card is labeled 1 and the bottom MN . The permutation $P_{M,N}$ is achieved by playing the following card game.

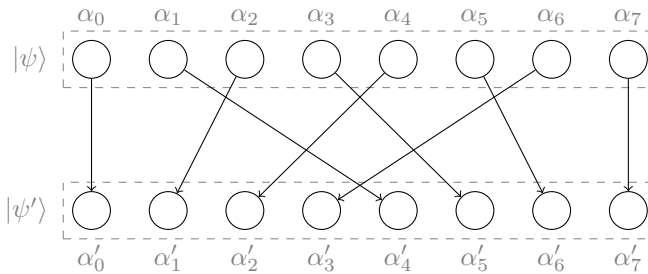
The dealer deals each of the M players one card, face up, from the top of the deck in succession, repeating the same process until the deck is exhausted and each player has N cards. The last player then gives the previous player his stack of cards, who puts them on top of his own. This second to last player repeats this process until the first player has all the cards. Finally the first player returns this stack face-down to the dealer.

The property $P_{M,N}P_{M,N}^{-1} = P_{M,N}P_{N,M} = I$ is easily checked by running the above game for $P_{M,N}$ and then using the result to play $P_{N,M}$ which returns the deck in the original order. The play for $P_{1,MN}$ is trivially checked to have no effect, each player only gets one card and stacks the deck back in the same order. Playing the game with one player $P_{MN,1}$ also keeps the original order, so $P_{1,MN} = P_{MN,1} = I$.

The permutation operator is implemented directly in our fine-grained graph by moving amplitude state node to state node. Note that such permutations in practice reality can take on several forms [117, 190], using the recursive decomposition and properties of the permutation operation to better suit the implementation hardware. These permutations can thus manifest themselves in different ways depending on the underlying execution framework, we abstract away from these underlying issues and represent the permutation directly in the fine-grained graph with nodes and edges. The construction of a graph for $P_{M,N}$ is straightforward in view of Equations (5.3.13) and (5.3.15).

Proposition 15. *Given the expression $P_{M,N}|\psi\rangle = |\psi'\rangle$ where the amplitudes of $|\psi\rangle$ and $|\psi'\rangle$ are indexed by i and j respectively⁴. Such a permutation operation is implemented in the fine-grained graph by a set of identity operation nodes connecting the state nodes representing α_i with α'_j where the indices $j = p_{N,M}(i)$.*

For example $P_{2,4}|\psi\rangle$ ⁸ is implemented by the following fine-grained graph



where an edge between two state nodes implies the identity operation.

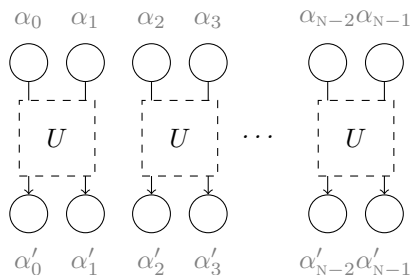
5.3.5 Multiple qubit states

Earlier in Section 5.2.3 we showed how an operation targeting a qubit in *parallel position* exhibits a simple execution pattern, repeating the one-qubit state version of the operator over contiguous parts of the amplitude vector. The practical benefit of this is that the fine-grained graph for a parallel position operator $U_{\mathbf{n}}^{\mathbf{n}}$

⁴Indexed by refers to the subscript of the amplitudes, such that for this case $|\psi\rangle = \sum_i^{MN} \alpha_i|i\rangle$ and $|\psi'\rangle = \sum_j^{MN} \alpha'_j|j\rangle$.

is a simple repetition of the graphs computing the single-qubit U , such as those presented in the previous section.

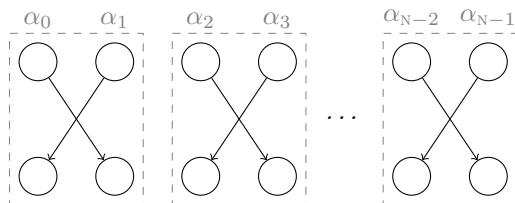
Proposition 16. *The fine-grained graph to compute the application of a parallel positional operator $U_{\mathbf{n}}^{\mathbf{n}}|\psi\rangle$ is obtained by repeating 2^{n-1} times the graph computing the single-qubit operator U on each two consecutive amplitudes of $|\psi\rangle$.*



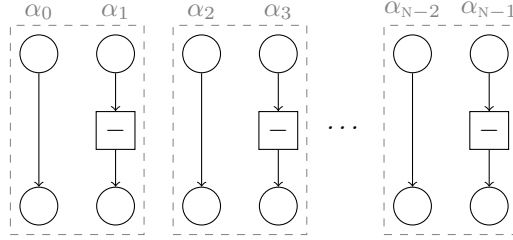
The above figure is somewhat schematic; the entanglement operator takes four amplitudes and the measurement produces one amplitude rather than two. Nevertheless, in all cases the same graph pattern is repeated a number of times over different parts of the amplitude vector. The measurement and entanglement operation cases will be discussed separately below. We present for each operation found in the pCG a proposition that refines it down to a fine-grained computation graph. These are the parallel position operators $X_{\mathbf{n}}^{\mathbf{n}}$, $Z_{\mathbf{n}}^{\mathbf{n}}$, $M_{\mathbf{n}}^{\mathbf{n}}$ and $\wedge Z_{\mathbf{n}-1, \mathbf{n}}^{\mathbf{n}}$; the position change operator $\mathbb{P}_{\mathbf{i}}$ and the merge operation \otimes . The position change operator is already covered by Proposition 15 for quantum states of arbitrary size.

Putting everything together, above we discussed both the simple one-qubit state Pauli-X and -Z, and the way to express positional operator in the parallel position $U_{\mathbf{n}}^{\mathbf{n}}$ as a repeating pattern of the more simple operator $U (= U_{\mathbf{1}}^{\mathbf{1}})$.

Proposition 17. *The fine-grained graph implementing a correction operation of the form $X_{\mathbf{n}}^{\mathbf{n}}$ is*

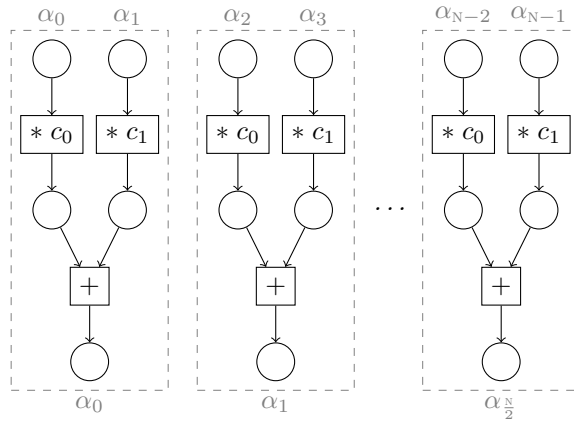


in which the dotted pattern repeats 2^{n-1} times and each implements the X operator on one qubit as can be seen in Proposition 9. The operation $Z_{\mathbf{n}}^{\mathbf{n}}$ is similarly implemented as



Similar to the above, the measure operation is a repetition of the single-qubit measurement in Proposition 10.

Proposition 18. *The fine-grained graph implementing the parallel position measurement operation $M_{\mathbf{n}}^{\mathbf{n}}$ is*

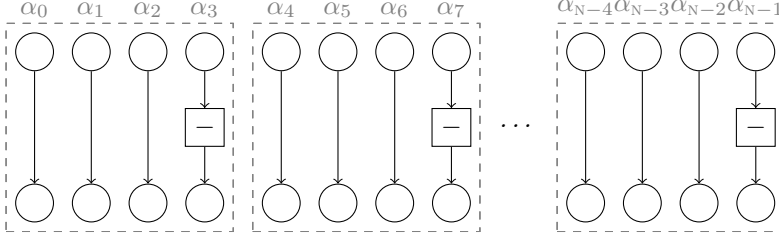


in which the dotted pattern is repeated 2^{n-1} times. The constants are as in Proposition 10 dependent on the measurement angle α , here suppressed in notation, such that $c_0 = \langle +_{\alpha} | 0 \rangle = \frac{1}{\sqrt{2}}$ and $c_1 = \langle +_{\alpha} | 1 \rangle = \frac{e^{-i\alpha}}{\sqrt{2}}$.

Entanglement

Two operations are involved in the entanglement operation: the merge operation \otimes and the parallel position controlled-Z $\wedge Z_{\mathbf{n}-1, \mathbf{n}}^{\mathbf{n}}$. The latter has a graph which is again a repetition of a pattern, in this case a two-qubit one, given in Proposition 11.

Proposition 19. *The parallel position entanglement operator $\wedge Z_{\mathbf{n}-1, \mathbf{n}}^{\mathbf{n}}$ is computed by the following fine-grained graph*



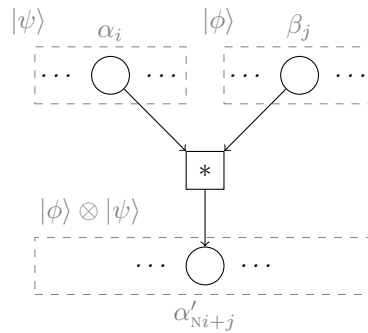
or in other words every amplitude α_i with index $i \equiv 3 \pmod{4}$ passes through an operation that flips its sign, all other amplitudes remain unchanged.

The merge operation node in the pCG performs the tensor product of two input amplitude vectors $|\psi\rangle \otimes |\phi\rangle$, in this contexts also sometimes called Kronecker product or tensor direct product. As a reminder to the notation used, we have seen earlier in Equation (5.3.8) that such a tensor product can be expressed as follows

$$\begin{aligned} \sum \alpha_i |i\rangle \otimes \sum \beta_j |j\rangle &= \sum \sum \alpha_i \beta_j |i\rangle |j\rangle \\ &= \sum \sum \alpha'_{Ni+j} |Ni+j\rangle \\ &= \sum \alpha'_k |k\rangle \end{aligned}$$

in which N is the number of amplitudes in $|\phi\rangle$. The fine-grained computation graph for the above thus has MN state nodes, which are connected to as many operation nodes multiplying each amplitude of $|\phi\rangle$ with each of $|\psi\rangle$.

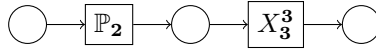
Proposition 20. *The fine-grained computation graph performing the vector tensor or Kronecker product on amplitude vectors $|\phi\rangle^M$ and $|\psi\rangle^N$ is given by repeating MN times the following pattern*



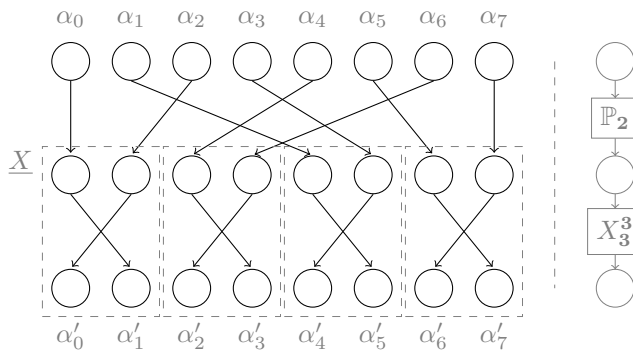
with each a different index pair $(i,j) : i < M, j < N$.

5.3.6 Construction

The pCG already expressed all operators as positional operators, at a cost of a permutation of the amplitudes prior to the operator application. In Section 5.3.4 above we have examined this permutation closely and expressed it as a computation graph. It thus suffices to directly expand operation nodes in the pCG, for example



expanding each operation by its fine-grained graph produces



where we put the fine-grained and positional coarse graph versions of the same computation in juxtaposition. As there are only identity operations between state nodes in the above example, the computation graph can be contracted by 'tracing through' the edges, for example directly connecting α_0 to α'_1 . Such a contraction does not change the produced result. We will however keep to the pCG's structure when it is practical, as regular repeating patterns are more important for the sake of simplicity than fewer nodes. Putting it all together, Figure 5.10 presents a large example that combines all MC operations; when tracing through the permutations on the left hand side, we absorb the permutation into positional operator on the right.

5.3.7 Discussion

The fine-grained graph describes the quantum computation at the amplitude level, using a dataflow computational graph. While we retain the broad structure of the pCG, each operation is refined down to individual arithmetic operations on amplitudes. The number of nodes and edges in the fine-grained graph is of the same rough order as the number of amplitudes in all states: an exponential

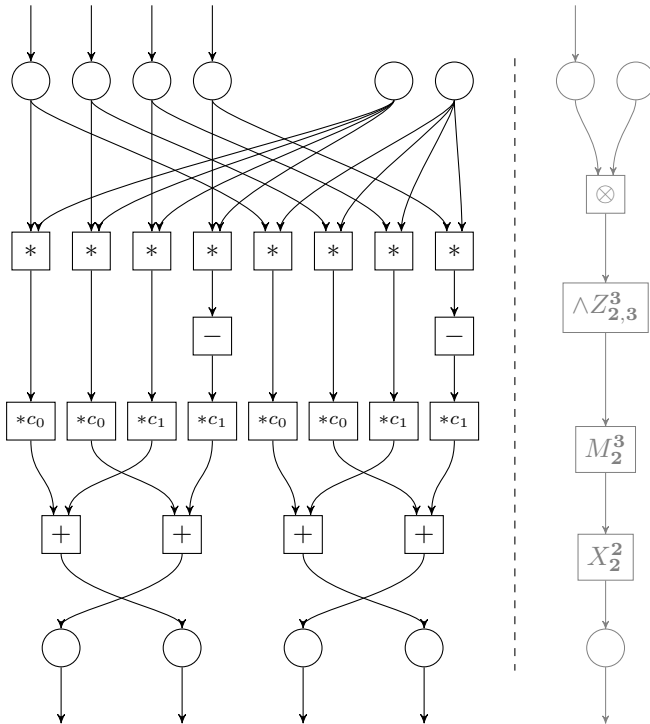


Figure 5.10: Fine-grained (left) and positional coarse (right) graphs of a $\mathcal{J}(\alpha)$ MC pattern acting on a two-qubit input tangle. For compactness, non-input amplitude and tangle nodes have been omitted and permutations have been absorbed into the positional operator.

complexity. However, such a complexity argument is moot. First, we define the fine-grained graph in the context of the pCG and retain its graph structure. As such only the insides of the operations have to be described in a fine-grained way. Second, the fine-grained graph is meant to describe the QCSim computation down to the finest level, not to be executed literally. It is at this finest level that one can see the repeating structures and optimization opportunities, but optimization mainly depends on the target computing platform. Such amplitude-centric definitions of all operations are also important in implementations that do not directly use an off the shelf programming library for linear algebra, as is the case for our implementation presented in the next chapter. For example in a demand-driven dataflow evaluation environment, one has to describe the dependent computations for each amplitude, which is trivial when a fine-grained

graph description is at hand.

5.4 Conclusion

We presented a set of intermediate models and transformations that bridge the gap between the Measurement Calculus and the dataflow models. Starting from a graph form of a MC program, state is introduced explicitly by following the amended operational semantics for the MC as presented earlier in Section 3.5.4. Using the original MC semantics would result in a strictly linear graph, due to the single global quantum state. The amended operational semantics factor the quantum state where possible into multiple *tangle* state nodes, which gives us a first degree of parallelism in the resulting Coarse Graph (CG) model. Committing to a column-vector representation for the quantum state forces us to deal with qubit positions within the quantum state. We therefore introduced the Positional Coarse Graph (pCG) model which turns each MC operation into a more concrete *positional operator*: a positional operator uniquely defines a matrix operator. Operations on the last qubit position exhibit a more natural parallel form. Each MC operation can be turned into such *parallel position operator* by introducing position-changing operators. The pCG model thus refines the MC operation nodes from the CG into several position changing operators and parallel position operators. In the final model, the fine-grained dataflow graph, each quantum state node in the pCG graph becomes a group of amplitude state nodes. By realizing the individual MC operations as simple amplitude-level dataflow graphs, we can repeat these graph structures to construct a fine-grained graph for multi-qubit parallel-position operators. This fine-grained model is a pure dataflow graph, in which each state node is a single amplitude data element and each operation node an elementary arithmetic operation.

Our fine-grained model is a pure dataflow model, execution of this graph will require an additional transformation to a hardware-specific form. For true dataflow machines, this translation can be straight forward. Steam-based or vector-based parallel machines would require more extensive transformations revolving around the parallel decomposition of the permutation operator [117]. In the next chapter we present a implementation artifact, which performs the above model transformations as multiple compiler stages, including an additional step to encode the fine-grained graph into the chosen parallel technology.

Chapter 6

Execution, Implementation and Validation

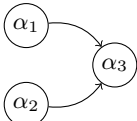
This chapter is split in three major parts. In the first section we offer an analytical validation that uses conceptual tools to show that the fine-grained graph does indeed expose a vast amount of parallelism, independent of the actual implementation. The second section goes over our implementation artifact: its structure, technology and target parallel platform. In the empirical validation section we demonstrate real-world parallel performance.

6.1 Analytical Validation

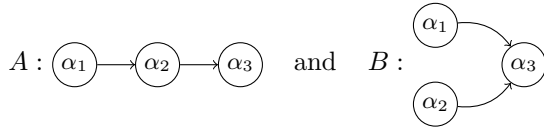
Our thesis contains the claim that expressing the Measurement Calculus as a dataflow computation exposes a large amount of parallelism. Supporting this claim solely on an empirical study is hard; real-world performance is very sensitive to specific implementation and machine architecture details, making it difficult to draw conclusions about the parallel approach itself. Therefore, we supplement the empirical results of our implementation with an analytical study of our fine-grained dataflow approach to quantum computing simulation. First, we use a formal definition of *more parallel* to sketch a theoretical proof. Second, we use a quantitative metric: average parallelism [98, 88]. Average parallelism is a machine-independent metric, observable from the program description itself and has in practice shown a high correlation with parallel speedup. In the first theoretical part of our analysis, we use a qualitative argument to show that our approach does indeed expose more of the inherent parallelism. In the second part, we provide a quantitative measure of parallelism exposed by our parallel programs. Both parts of this analysis support our claim in a way that is independent of implementation or parallel machine choices.

6.1.1 Theoretical Parallelism

The formal tools we use here are based on the work on parallel schemata, which has been used since the 60s by theoretical computer scientists to analyze parallel algorithms [124, 126]. A *program* is defined in Keller [126] as a way to achieve a *computation* and consists of a set of operations. Considering our specific case we name operations by $\alpha_1, \dots, \alpha_n$, roughly corresponding to operations on amplitudes of a quantum state. Two different *parallel program* schemata can both implement the same *computation*, meaning they both produce the same result in the end. Any valid linear arrangement of these operations for a parallel program we call a *schedule*. Any two operations in a program can have dependency constraints, imposing that one operation appears before the other in any schedule. For example,

the program schema:  has two schedules $(\alpha_1, \alpha_2, \alpha_3), (\alpha_2, \alpha_1, \alpha_3)$.

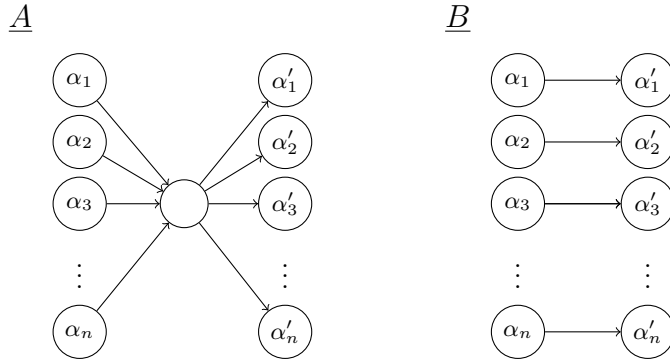
The number of different schedules of a program is used as a measure of parallelism, which we can use to show that one program is *more parallel* than the other. For instance, taking



as programs implementing the same computation. B is more parallel than A , because $\text{sch}(A) \subseteq \text{sch}(B)$ and there exists the schedule $(\alpha_2, \alpha_1, \alpha_3)$ of B that is not a schedule of A . That is, $\text{sch}(A) \subset \text{sch}(B)$.

Definition 11. *With two parallel programs A and B , both implementing the same computation, parallel program B is more parallel or exposes more parallelism than program A if $\text{sch}(A) \subset \text{sch}(B)$. That is, if all possible schedules of A are also schedules of B ($\text{sch}(A) \subseteq \text{sch}(B)$), and $\exists s \in \text{sch}(B) : s \notin \text{sch}(A)$.*

We prove that our dataflow approach exhibits more parallelism than existing approaches by constructing program schemata representing both in principle. As we already observed in Chapter 4, current parallel quantum simulators [182, 99, 56] use MPI to implement a communication step between each quantum operation. This communication step acts as global synchronization between quantum operations; a second quantum operation can start being computed only if the computation of the previous quantum operation has been completed. Our fine-grained dataflow model introduced in Chapter 5 does not exhibit such global synchronization. We take A and B as two different programs implementing the same computation. As part of this computation, both programs realize the successive application of the two quantum operations U and U' . We take the idealized case that all operations involved in computing U ($\alpha_1, \dots, \alpha_n$) can operate in parallel. The same holds for U' ($\alpha'_1, \dots, \alpha'_n$). Program A stands in for the approaches with global synchronization between quantum operations, which adds constraints such that no operation for U' may start before all operations of U have finished. Program B stands in for the approaches without global synchronization, each sub-operation for U' only depends on the part of U that is required for the computation; i.e. the operations computing U' can already start while U operations are still underway. We use A to represent related work approaches with global communication steps in between quantum operations, B represents a fine-grained dataflow approach. Note that for both programs, we still assume idealized parallel realization of U and U' . Schematically:



More formally, in all schedules of A holds $\forall \alpha_j, \alpha'_i : \alpha_j < \alpha'_i$, while all schedules of B holds only $\forall \alpha_i, \alpha'_i : \alpha_i < \alpha'_i$. To conclude the proof, all schedules of A also appear in B such that $sch(A) \subseteq sch(B)$. But, there exists schedules $(\dots \alpha'_i \alpha_j \dots)$ in B that are not in A . According to Definition 11, B is more parallel than A .

What we have shown above is that our dataflow parallel program can exploit not only available data parallelism, but also a form of *pipeline parallelism*. There are no reasons why an MPI-implementation could not implement program B as well, exploiting the extra parallelism. We do argue that doing so would require structuring and implementing a program that would in essence be an ad hoc and limited form of dataflow execution. We do not prove here that our fine-grained dataflow is *maximally parallel*, and thus exposing all of quantum computing simulation's naturally available parallelism. But, we can make this assumption, as long as it cannot be shown that another *program* implementing the same *computation* is *more parallel* by the same definition.

6.1.2 Average Parallelism

[...] This seems to indicate that this crude approximation to the overall average parallelism of a code is all that is necessary for an accurate prediction of its speedup curve.

–Gurd et al. [98]

Average parallelism is used as a metric to quantify the amount of parallelism exposed by a parallel program. It is a property or characteristic of a specific parallel program and can be calculated independently of underlying execution. To our knowledge, no other machine-independent measure combines simplicity with the strong predictive power for parallel performance that average parallelism has shown in practice [98, 30]. Indeed, one of the popularizing factors of the Cilk [30] approach is the predictable parallel performance based on this observable

parallelism metric. Average parallelism helps programmers during the design and implementation of parallel programs by giving them a measure of progress and expected performance.

Average parallelism is defined as the ratio between the total work in the program and the length of its critical path:

$$\pi_{av} = \frac{S_1}{S_\infty} \tag{6.1.1}$$

The *total work* S_1 of a program is the time it takes for a single processing element to execute the program. The *critical path* S_∞ is the time to execute the program with an unbounded number of processing elements. Take the A and B programs used in the proof above. Their work is respectively $2n + 1$ and $2n$, their critical paths are 3 and 2. The resulting average parallelism for A and B programs is thus

$$\pi_{av}(A) = \frac{2n + 1}{3} \qquad \pi_{av}(B) = \frac{2n}{2} \quad ,$$

in which we indeed observe a higher average parallelism for B , with a better scaling behaviour for an increasing n .

In parallel programs structured using DAGs, the S_1 and S_∞ properties are directly observable from the program graph. We proceed by first making the assumption that each elementary operation takes one time unit to execute. S_1 is then obtained by counting the number of operation nodes in the program graph. The critical path is obtained by counting the operations in the longest path from input to output node in the program graph.

Next, we use the fact that the fine-grained graph follows the structure of the coarse-grained graph, which only has a small set of distinct operations. The S_1 of an individual coarse operation can be obtained as a function of the tangle size and of its fine-grained graph. For example, the number of elementary operations and thus S_1 in the fine-grained graph of Z_n^n given in Proposition 17, is $\frac{N}{2}$. The critical path of the same operation is simply 1: the longest path only contains one operation node. Examining each coarse operation's fine-grained graph, as defined in Section 5.3, results thus in the following Table 6.1.

In this analysis, we use the simplification that permutations such as X_n^n or the stride permutation do not constitute work. In practice, this heavily depends on the parallel architecture and various compiler and runtime optimizations. Moving a single data element over a network is slow, but some compile-time or runtime optimizations can in practice remove some of the work associated of local permutations. We make abstraction of this issue by working under the assumption that a sufficiently high amount of parallelism is available, such that the latency involved in moving or accessing data elements can be completely hidden.

| | $\wedge Z_{i,j}^n$ | M_i^n | Z_i^n | \otimes |
|------------|--------------------|---------|---------------|-----------|
| S_1 | $\frac{N}{4}$ | N | $\frac{N}{2}$ | NM |
| S_∞ | 1 | 2 | 1 | 1 |

Table 6.1: An exhaustive list of the total work (S_1) and critical path (S_∞) of each coarse-grained graph operation; the X_n^n and position operations are considered to contribute no work.

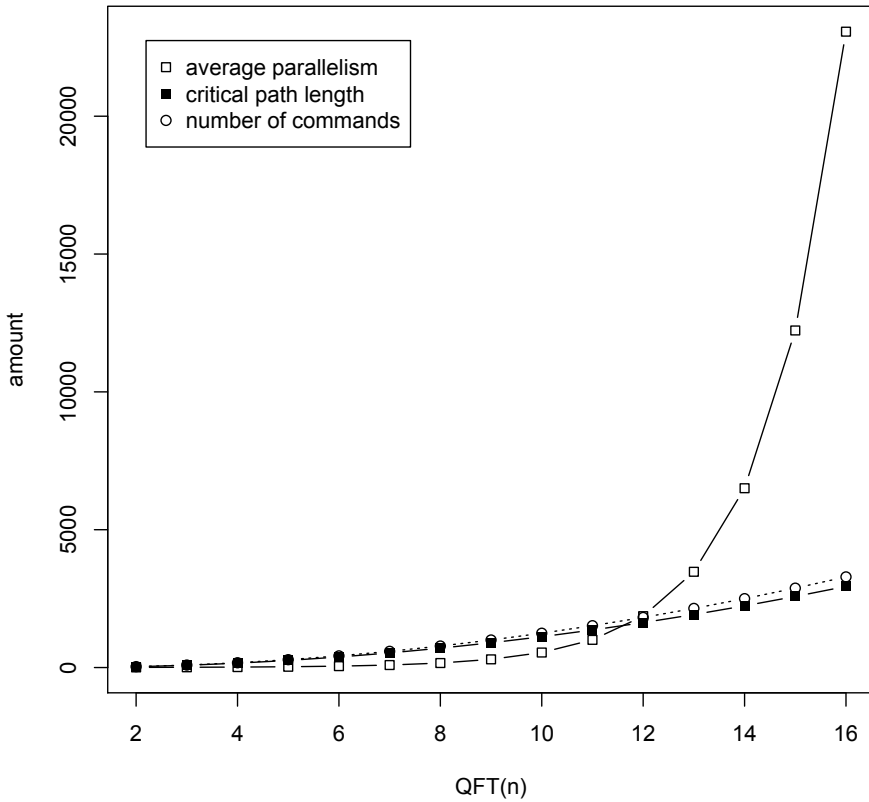


Figure 6.1: Average parallelism data points for different program sizes, indicating the exponential increase of work goes together with an exponential increase in exploitable parallel performance.

The S_1 of an entire MC program can then be calculated by taking the sum of all its coarse operations' S_1 . The S_∞ of an MC program is calculated by taking the critical path in the coarse graph, then summing the S_∞ of the coarse operations in that critical path. Rather than doing this by hand, we have automated this algorithm by integrating it in our parallel compiler (presented below in Section 6.2). The resulting metrics for a few interesting quantum programs are listed in Table 6.2 and plotted in Figure 6.1. The Quantum Fourier Transform (QFT) is interesting as indicator because of its practical relevance in QC, but also because it allows the step-wise incrementing of program length and qubit state width. We use $QFT(n)$ to refer to the n -qubit QFT program. The number of MC operations ($\#ops$) metric is included to compare program sizes. The π_{av} of the Quantum Fourier Transform can be observed to roughly double for each additional qubit, following the same increase in the 'width' of the quantum state. This is a predictable result, considering the wild pattern for QFT sequentially applies a long series of phase gates. The standardized pattern to create a W_3 state [71, 66] has only 40 MC operations rather than $QFT(16)$'s 3288, but affords a comparable measure of parallelism. The critical path length in every case reflects the number of MC operations, it is not significantly higher because of two reasons. First, the coarse graph enables the parallel execution of some MC operations. And second, the X operation is not counted as work.

| | $QFT(2)$ | $QFT(3)$ | $QFT(4)$ | $QFT(8)$ | $QFT(16)$ | W_3 |
|------------|----------|----------|----------|----------|------------|---------|
| $\#ops$ | 33 | 90 | 174 | 780 | 3,288 | 39 |
| S_1 | 165 | 779 | 2,595 | 114,699 | 67,862,667 | 920,682 |
| S_∞ | 30 | 82 | 158 | 702 | 2,942 | 42 |
| π_{av} | 5 | 9 | 16 | 163 | 23,066 | 21,921 |

Table 6.2: Work, critical path and average parallelism metrics of various MC programs from Section 6.2 benchmarks.

From the analysis of the average parallelism, as measure for parallelism, we can conclude that there is indeed evidence of a vast amount of exploitable parallelism. The average parallelism for larger quantum programs shows that there is indeed the potential parallelism to counteract the exponential increase in computational work with an exponential increase in computational resources. However, it is also the case that for computations like QFT there is also the effect of increasing critical path length, which is essentially a reflection of the highly entangled nature of such computations.

6.2 Implementation

The implementation artifact serves three goals; Validating our approach by empirically demonstrating good speedup on a parallel platform. Maintaining full QVM compatibility to ensure co-development and support for the measurement-based Quantum Programming Paradigm. And finally, provide a platform and a non-trivial case for further dataflow research, implementation and optimization. Both the *structure* and *technology* choices for the implementation artifact explained below are a result of balancing these three goals. The implementation artifact is structured as a stratified compiler, following the multiple model transformations in Chapter 5; staying as close as possible to the research we are validating (first and second goals) while providing enough hooks in the implementation for optimizations and alternatives (second and third goals). The parallel platform needs to be highly-parallel and follow dataflow execution semantics, staying close to our proposed parallel approach for the first goal: validation. But, the second and third goal also requires this platform to run on and cooperate with common development tools. Our choice of the Intel Concurrent Collection (CnC) framework [128] as parallel implementation platform reflects this trade-off.

This section first presents the general *structure* of the implementation artifact in Section 6.2.1, providing a frame of reference for the detailed discussions on *technology* and implementation in Section 6.2.2. Finally, with the implementation technology covered, we can treat the key technology-specific compilation phases in Section 6.2.3 that form the final executable program. These programs are then used for the empirical validation in Section 6.3.

6.2.1 Structure

The artifact we describe here acts as an alternative execution layer for the quantum programming paradigm proposed in Chapter 3. Before, we used and described a simple MC interpreter which uses a straightforward implementation that sequentially executes each instruction in turn. The implementation elaborated in this chapter provides an alternative execution platform for the QVM, effectively taking the same program input but executing the quantum simulation in parallel. Much of the analysis and transformations from Chapter 5 requires information on the complete quantum program. We have therefore opted to structure the implementation as a *compiler*. In the rest of this text we will refer to this artifact as `mcc`: *Measurement Calculus Compiler*.

The `mcc` is structured as an incremental compiler; using a progression of multiple compilation phases that roughly follow the structure of Chapter 5. An overview of its structure can be found in Figure 6.2. The input language for `mcc` is a QVM instruction sequence. The ultimate output of `mcc` is C++ code, which relies

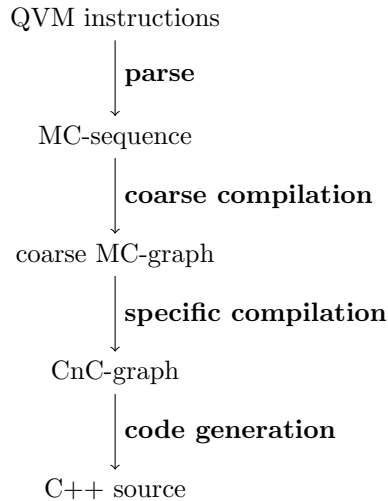


Figure 6.2: Measurement Calculus Compiler (`mcc`) structure overview.

on the Intel[®]CnC framework as parallel programming platform. Detailed discussion on CnC and other technology platform choices will follow in Section 6.2.2. As already mentioned, the intermediate representations within the `mcc` roughly follow the progression of abstract models from the previous chapter. Although, out of practical considerations, some models and features were combined. For example, the CG-equivalent representation in `mcc` uses ordered tangles from the start. The `mcc` has three internal representations; MC-sequence, coarse MC-graph and CnC-graph. There are thus four compilation phases, including the transformations to and from the two external representations: the QVM instruction sequence input and C++ source code output. The first two phases are relatively straightforward. The more interesting and implementation-dependent phases are the *specific compilation* and *code generation* steps. The many technology-specific choices and trade-offs of these last two phases are discussed after the technology section below.

Parse As the input to `mcc` is a textual or symbolic representation, a simple parse step is needed to transform it into an internal representation. The parse step retains the sequence structure of the QVM language, thus producing a sequence of abstract grammar objects. For example, the simple $\mathcal{J}(\frac{\pi}{2})$ -pattern as QVM instruction sequence is

(E 1 2) (M 1 pi/2) (X 2 (s 1))

is compiled into a sequence of three objects

(E M X)

where each program object contains the necessary information such as target qubit names, signals and measurement angle when applicable. For example the object **M** is a record containing:

```
{ qubit: 1 ; angle: 1.57 ;
  s-signal: {...} ; t-signal: {...} }
```

in which `s-signal` and `t-signal` are again records. These parsed operation objects are used as the abstract grammar of an MC program. In principle, this parse phase into an abstract grammar also happens within the sequential interpreter of the `qvm`, but on a per command basis. The `mcc` parses the complete command sequence before handing the abstract grammar objects as input to the next phase: coarse compilation.

Coarse compilation Recall that in Chapter 5, the construction of the coarse graph is described as a transformation of the MCG: a graph representation of an MC program. For the sake of explanation, it makes more sense in the theoretical work of Chapter 5 to start from the MCG as MC program representation. However, the coarse compilation step takes a sequential program MC program as input from the parser; the `mcc` is designed to fit in the proposed layered architecture of Chapter 3 as execution and realization layer, taking the same command sequence representation as input. Both graph and sequence representations of an MC program encode the same information and it is relatively trivial to move from one to the other. Generating an MCG from this serial MC representation would be superfluous, considering the equivalence or dual nature of both representations.

The coarse compilation step thus implements the construction of the abstract CG as discussed in Section 5.2.2, but with the trivial difference of the construction process walking through an MC sequence encoding a topological sort, rather than walking through an MC graph in topological order. A fresh node is created for each operation and connected with newly or previously created tangle nodes, as described by the semantics laid down in Section 5.2.2. The tangle nodes contain an abstract tangle object that contains static information about the tangle: qubit list and size. Note that a qubit list is constructed as in the positional Coarse Graph, rather than an unordered qubit set as in the CG; another practical difference in which the implementation diverges from the theoretical CG. Taking the example's sequence of abstract grammar objects:

(E M X)

is transformed by the coarse compilation step into the coarse graph shown in Figure 6.3. A more complex example for the Deutsch-Jozsa algorithm [65], for which the MC sequence can be found in Listing 6.1, is given in Figure 6.4. These coarse graphs were automatically generated by the implementation, the signal nodes are not discussed here and may be ignored.

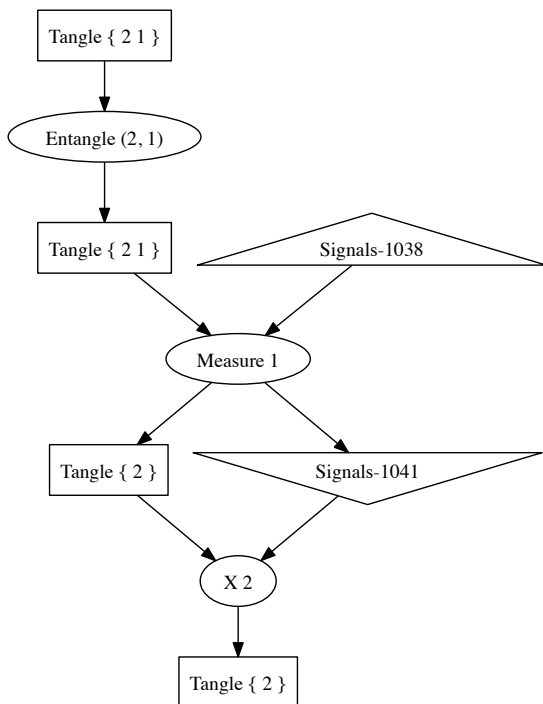


Figure 6.3: Coarse graph resulting from coarse-compilation of the input MC sequence: $((E\ 1\ 2)\ (M\ 1\ \pi/2)\ (X\ 2\ (s\ 1)))$.

Specific compilation This compilation step maps the coarse graph representation onto an abstract CnC program graph, the program representation of the chosen parallel platform: Intel Concurrent Collection (CnC). In the terminology introduced in Section 4.3.2, CnC is a hybrid framework approach in which intra-task execution is control-driven and inter-task execution is data-driven. For now, we keep to a structural overview of the specific compilation phase, without going into CnC-specific details.

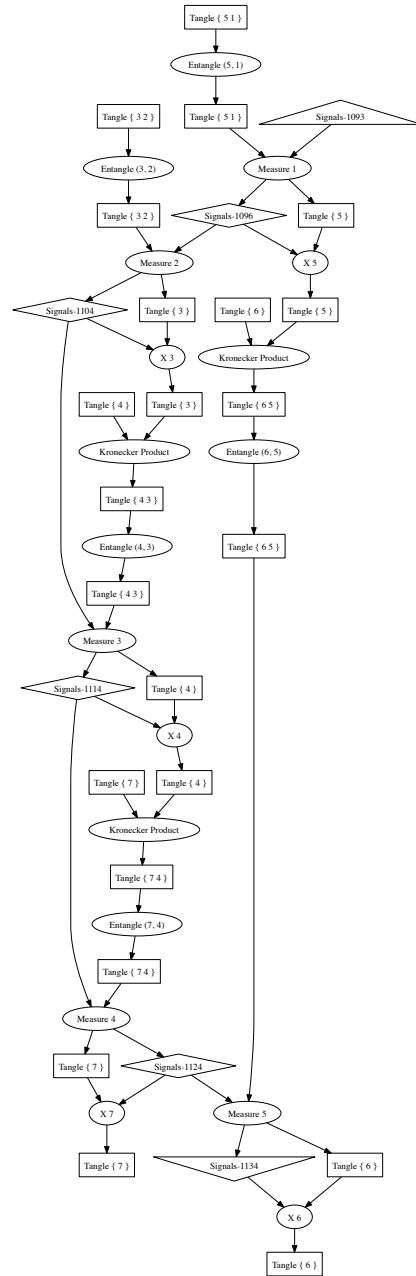


Figure 6.4: Automatically generated coarse graph for the two-qubit Deutsch-Jozsa MC pattern in Listing 6.1, showing some coarse level of parallelism.

```
((E 1 5) (M 1 0) (X 5 (q 1))
(E 2 3) (M 2 0) (X 3 (q 2))
(E 3 4) (M 3 -pi) (X 4 (q 3))
(E 4 7) (M 4 0) (X 7 (q 4))
(E 5 6) (M 5 0) (X 6 (q 5)))
```

Listing 6.1: The MC sequence for a wild pattern implementing a two-qubit Deutsch-Josza algorithm, for the function $f : \{0 \rightarrow 1\}, \{1 \rightarrow 1\}$ realized with $I \otimes X$.

CnC works with a vision in which a *domain expert* creates a CnC graph that captures the rough structure of the computation. Based on this graph, the CnC compiler then generates skeletal program code. A *tuning expert* fills in the details and optimizes the program for performance. We emulate this design organization in `mcc`; the *specific compilation* phase first creates a CnC graph, then the *code generation* phase generates platform-specific program code. However, not all necessary information about the CG can be encoded within the canonical CnC model. The code generation step requires information such as the type of MC operation, tangles sizes, stride permutation parameters and so on. The specific compilation phase therefore produces a `mcc`-specific CnC program graph, in which each CnC node hold records with additional information extracted from the CG. In other words, the specific compilation phase acts as the domain expert: transforming the coarse program graph into a domain-specific CnC program graph. This internal representation is still generic enough to be applicable to different CnC implementations and hardware platforms. But, it does holds all information required by the code-generation phase to produce efficient platform-specific program code.

Code generation The current code generation phase in the `mcc` produces C++ code that uses the Intel CnC++ implementation. The normal CnC++ use-case for programmers is to manually modify the code generated by running the canonical CnC graph through the CnC++ compiler. As we automate the entire code generation process, we bypass the provided compiler and use the CnC++ API directly. The CnC++ API, without going into too much detail, is based around a ‘context’ object that encodes the CnC graph. The program code for each node inside this context object is generated by making heavy use of templating techniques: program code with ‘holes’ for required constants. For example, a CnC node implementing the X_{8}^{16} operation uses the template for X , filling in the constants 8 and 16 to generate the correct code. The `mcc` compiler thus contains at least one template for each operation in the coarse graph. The code generation also produces code to handle the interface between the external environment and the CnC context object. This means feeding it input, starting the parallel execution and finally extracting the output. The final result of the code generation phase is C++ code that can be compiled into an executable for a variety of platforms. The code generation phase also adds code to interact with this executable after compilation, for example, to control the number of execution threads, which is used during the *empirical validation* in Section 6.3.

6.2.2 Technology

We make three main technology choices, cutting a slice from the large potential design space. Choosing the *infrastructural* platform: the implementation environment for the compiler itself. Choosing the *parallel software* platform: the framework, library or language used to implement the parallel program. And finally the choice of *parallel hardware* execution platform: the target parallel processor architecture.

The *infrastructure* of the Measurement Calculus Compiler or `mcc` is written in the general-purpose programming language Common Lisp, making generous use of its powerful object and macro system. The output of `mcc`'s code generator phase, as seen above, is C++ source code that can be compiled to a native binary on a large variety of computer platforms. C++ is used mainly because it is the host language of our target parallel software platform: *Concurrent Collections* or *CnC*. Other CnC implementations exist, but CnC++ has demonstrated better performance and is currently the most available, stable and practical implementation [46].

We choose Concurrent collections as *parallel software* platform owing to two theoretical and three practical characteristics. First, CnC uses *dataflow* execution semantics; it is the availability of data that will cause individual tasks to be scheduled for execution. The second theoretical characteristic is that CnC allows a more *fine-grained parallel approach*. In contrast, other dataflow frameworks are often focused on coarse-grained tasks [113]. The practical reasons for choosing CnC are performance, broad applicability and currency. CnC++ has demonstrable *performance* that matches the state of the art [46]. This gives us known performance bounds and a baseline for comparison. CnC is *broadly applicable*, by which we mean it is general purpose, can be applied across many problem domains and can be integrated well with existing programming techniques and tool-chains. Being *current* means that CnC has a reliable implementation, active developers and is still being used in research and industry settings. However, CnC remains a compromise. It is a hybrid dataflow framework in which many optimizations observed in pure dataflow systems are not automatically performed by CnC itself. Instead, this task is left to the user by way of CnC's tuners and various other manual optimizations. Still, to our knowledge no other current framework, language or runtime provides this combination of theoretical and practical characteristics.

The choice of *parallel hardware* platform is a balance between the fine-grained parallel model we seek to validate and real-world parallel performance. The ideal solution would be to run our program directly on a dataflow machine, using a dataflow language rather than the hybrid CnC. However, this is no option considering the current lack of such pure dataflow architectures. This leaves us with the set of parallel hardware available today. For the purpose of this discussion,

we cut up the current parallel hardware landscape as presented in Chapter 4 in three broad categories: stock, fringe and specialist hardware.

- Stock hardware is the mainstream processors in commodity computers: multicore processors with a uniform memory configuration.
- Fringe hardware are today's non-multicore parallel processors, which are typically oriented towards a different computing market niche: GPUs, Tiler, Cell B/E, FPGAs, Xeon Phi, etc.
- Specialist hardware is the amalgam of parallel hardware used in the High Performance Computing domain: cluster configurations, hybrid processor/co-processor nodes, Non Uniform Memory Architecture (NUMA) multiprocessor configurations, etc.

We choose stock hardware because of a combination of factors.

- Stock hardware is more readily available, making it more accessible and helping the *integration* goal.
- For the validation goal of our implementation, stock hardware offers better empirical *comparison* and a more straight-forward implementation approach.
- Although specialist and fringe hardware offer a larger amount of parallelism, this comes at a greater *implementation cost*, with substantial architecture-specific modification of the parallel program.

For a fairer empirical comparison, stock hardware offers generally the same architecture. A stock CPU implementation will have comparable performance characteristics across brands and processor generations. Fringe and specialist hardware performance varies widely depending on the nature of the workload. Stock hardware also makes it possible to compare the best sequential with the parallel implementation on the same processor. The main factor, however, in favor of stock hardware is the choice of the CnC software platform. The most mature CnC implementation supports stock multicore processors, and only a preliminary MPI implementation for clusters.

A word on GPUs. Arguably, GPUs could be considered stock hardware, seeing as such programmable graphics functionality are integrated in the vast majority of today's computers. A similar argument can be made to consider GPUs in the specialist hardware category. Indeed, GPUs are an increasingly common sight in HPC compute clusters, in an accelerator co-processor role. However, GPU hardware is still mainly designed to do graphics processing. This naturally shapes their architecture, programming model and preferred workload. Algorithms that

do well on GPUs are those that can be performed as simple stream computations with high computational density and unit-stride memory access patterns [141, 107]. We therefore put GPUs in the fringe hardware category, considering their clear affinity to specific computing problems. Even without considering the issues of implementation cost and comparison, we have reason to avoid committing to a GPU hardware target for our proof of concept implementation: in our case it is not certain that our simulated MC execution computation maps well to GPUs. For one, MC operations have a relatively low computational density, only few operations are performed for each data element. Its data access pattern, which we capture using the stride permutation operator, can be problematic for the banked memory accesses and small caches sizes of GPUs. Currently, it is unclear if stride permutations can be efficiently performed on modern GPUs, making it an interesting topic for future work.

Concurrent Collections

Concurrent Collections [128, 40, 39] is a parallel programming model that aims to be simple yet powerful. It is part of an ongoing effort that was in a previous life known as TStreams [129] from HP Cambridge, later implemented as CnC++ at Intel on top of their Threading Building Blocks (TBB) [156] work-stealing back-end in C++. A Habanero-Java implementation of CnC is also used as a research platform at Rice university [40]. The Concurrent Collections programming model is implicitly parallel, but with explicit task partitioning. CnC separates the *design* issues of expressing the computation in separate tasks from the *implementation* issues of distribution, scheduling and parallel execution. We use a running example to clarify the CnC programming model and execution semantics, it should be familiar enough to the reader due to its similarity – at a conceptual level – to the abstract dataflow models we have been using so far.

The CnC graph has three type of nodes; *item collections* encapsulate data, *step collections* encapsulate computation and *tag collections* are used to control the execution of steps. Conceptually, a CnC node or collection can contain a large number of element instances. In practice, these elements are not always present. Indeed, it is the availability of *tag* elements that will instantiate new *step* function instances which will each perform some computation that consumes and produces *items*. There are thus produce and consume relations between step and item collection nodes. Each step collection needs to be associated with exactly one tag collection in a *prescribe* relation.

One such CnC graph is given in Figure 6.5 for the vector tensor product example: $|a\rangle \otimes |b\rangle = |c\rangle$. The square is the graphic of an item collection, circle for a step and trapezium for a tag collection. Produce and consume relationships between nodes are represented by full arcs. Produce and consume relations with the external environment are represented using meandering edges. The contents

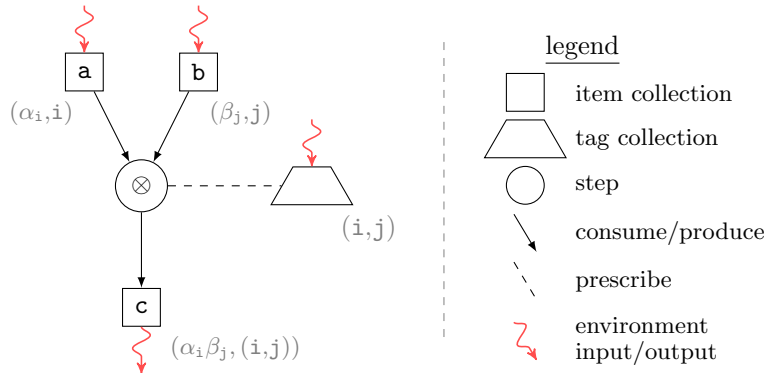


Figure 6.5: CnC graph for the vector tensor product. Annotations in gray comment on the content of each node, in which i, j are indices and α_i, β_j amplitudes.

of both item collections **a** and **b** come from the outside environment, one can conceptually imagine a separate process continuously feeding vector elements into both collections in arbitrary order. The step labeled \otimes multiplies an element from **a** with one from **b** and thus needs to be invoked once for every possible combination of both. A straightforward solution is thus to *prescribe* the step collection with a tag collection of all (i, j) tuples, where i and j are indices used as tags for respectively **a** and **b**. The step function itself is defined in the host programming language as a native function that takes a CnC context object and a tag. The context object conceptually contains the entire CnC graph and is used within the function body to interact with other CnC nodes. Typically, the context object is used for getting items from and putting them into collections. A tag is a user-defined datum that is used to differentiate between the multiple invocations of the step function; in our example the \otimes step function is called once for every different (i, j) tuple. Putting this all together, the pseudocode for the \otimes step function is given in the following listing.

```

function kronecker_product( context , [ i , j ] )
    size_b ::= size of vector b
    factor_1 := get i from context.a
    factor_2 := get j from context.b
    put ( factor_1 * factor_2 ) tagged ( size_b * i + j )
    into context.c
    
```

Execution CnC's execution semantics is conceptually simple and guarantees certain properties, such as determinism. For a more formal treatment of CnC's

execution semantics we refer to Budimlí et al. [40]. During execution, each tag or item can be in an *available* or *unavailable* state. When an item or tag is put in a collection, it receives the *available* attribute. Additionally, when such a tag is made available in a tag collection that prescribes a step collection, a step is instantiated with the tag and marked as *prescribed*. A prescribed step instance with *consume* relationships will check the availability of its dependent items. When all consume dependencies are satisfied, the step instance is marked as *enabled* and scheduled for execution. The step function retrieves (consumes) the necessary items itself and sends out (produces) the new items. To clarify, we go over two different execution scenarios for the same tensor product example in Figure 6.5.

In the first scenario, the external environment initially puts all items in collections **a** and **b** simultaneously. All elements of both collections are thus marked as available, this does not trigger any further actions. As a next step the external environment puts the tuple (7,2) in the tag collection, the trapezium in the schema. This triggers a cascade of actions. The tag (7,2) is marked as available in the tag collection. Then, because of the *prescribe* relation between the tag and step collection, a new step instance for \otimes is created. This new step instance, invoked with tag (7,2), is not executed right away because it has *consume* dependencies that need to be checked. The data dependency of this instance is known to be the item tagged by 7 in **a** and the item tagged 2 in **b**. Both items are available, thus the step instance is enabled. A scheduler, running in a continuous loop, detects \otimes for tag (7,2) as enabled and sends it to an idle processing element for execution. During execution, the step function retrieves the respective amplitudes from **a** and **b**, multiplies them and puts the result in the item collection **c** using a fresh tag, which marks the item as available. The environment, waiting for an item of **c** to be come available, retrieves it from the collection and terminates. Note that if the external environment adds each tag after waiting for the previous result, then the computation is strictly sequential. However, if all tags are added simultaneously, then all steps can potentially be executed in parallel.

The second scenario follows the somewhat more common case, in which input items are not always available. For example, the data is still getting computed or is still being read from a disk or network. Unlike the above scenario, a prescribed step instance will not immediately transition into the enabled state. A pool of prescribed steps will be waiting for their *consume* dependencies to become available, before they can be considered for execution. The environment retrieves items from the output item collection **c** as they become available.

Implementation Multiple implementations of CnC exist, using multiple implementation approaches to the above execution semantics. Our artifact is implemented using CnC++, which uses the C++ programming language as a host

and Intel's work-stealing scheduler in TBB as underlying execution platform. A complete CnC program consists of three elements; the CnC graph, step code and environment code. In C++, one defines the CnC graph as a context object that contains the necessary nodes and edges. This definition can be automatically generated, using a compiler and code generator bundled with CnC++. This compiler generates the context object and step function stubs based on a declarative textual notation for CnC graphs. The step code is defined as a typical C++ function, taking a tag as argument. Such a step function can use `get()` and `put()` methods on the context object to interact with item collections. The environment is a regular C++ program which creates the context object, adds all necessary input to it, starts the CnC execution process, waits for it to finish and finally retrieves the results.

The CnC++ implementation deviates from the regular CnC execution scenario in one important way: by default, steps are enabled the moment they are prescribed. A step can thus execute while the items it retrieves are not yet available. In this case the step will *deschedule* itself, waiting for the event that the required items become available to try again. In other words, CnC++ has a more optimistic scheduler. However, recent versions for CnC++ have added *tuning* primitives. Inside a tuner, a programmer can explicitly declare data dependencies and thus prevent the automatic enabling of step instances. We will explore the different effects of tuning as part of our validation section below.

6.2.3 Compiling for CnC

We revisit the `mcc` specific compilation and code generation phases. The topic can now be treated in more detail after having introduced CnC. We structure our discussion first on the coarse elements: the CnC graph with collections as nodes; produce, consume and prescribe relations as edges. Then we move to the fine-grained elements: the description of the step functions and the tuning primitives.

Coarse structure: CnC graph

The Coarse Graph model maps naturally upon the CnC graph structure. A tangle maps to an item collection and an operation to a step. Each amplitude in a tangle thus becomes an item in an item collection. The tag for each amplitude item is its index or vector position. A step *consumes* the item collection associated with the input tangle. And, a step *produces* the output tangle's item collection. The combination of both creates a *pipeline* of successive producer/-consumer nodes. This still leaves open the required *prescribe* relation between step and tag collections. A naive approach would be to include a prescribing tag collection for each step, and enable all tags from the external environment. This

however would cause a vast number of idle step instances to be created. A more sensible approach is to have steps themselves insert the tag for each item they *produce*. A step *consuming* such an item is thus *prescribed* at that moment. This creates a rippling *execution front* in which new amplitude items and their index tags continuously percolate through the pipeline. The CnC graph in Figure 6.6 is a simple example of such a pipeline organization. The coarse graph of non-trivial

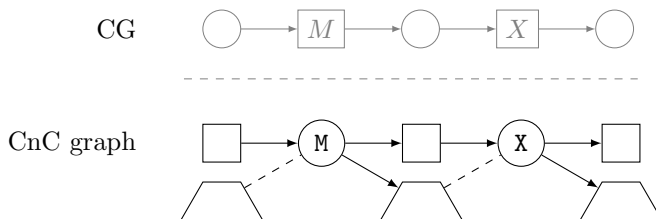


Figure 6.6: CnC producer/consumer pipeline organization example. Each step produces the tag to prescribe the next step in the pipeline.

QVM programs is typically not a straight sequence of nodes. Rather, the CG is structured as a tree with branches merging by way of the entanglement operation as the computation progresses. This structure can be observed for instance in Figure 6.4. The tensor product CnC graph from Figure 6.5, used as a running example earlier, uses a tuple of indices as prescribing tags for the \otimes step. In the pipeline structure, a step has to prescribe the tags for the next step in the pipeline. With two steps producing items for respectively **a** and **b**, it becomes hard to prescribe using tuple tags (i,j) . Conceptually, the \otimes step needs to be prescribed by joining both **a**'s i tag collection and **b**'s j tag collections. CnC currently cannot gracefully express such joined tag dependencies, but the developers plan to add the required functionality. Our practical solution to this is shown in Figure 6.7: each item collection **a** and **b** still has its separate tag collection, but the \otimes step is prescribed by only **a**'s tags. Each step instance, triggered by the availability of a single item α_i of **a**, thus needs to compute all $\alpha_i\beta_j$ for all j . The \otimes step in this case then consumes a single element of one collection and consumes all elements of the other. This stop-gap solution does sacrifice some of the available fine-grained parallelism in the implementation. We can only speculate about its impact, but as we will see below, the coarser parallelism likely improves the real-world performance in the current implementation.

Fine-grained structure: step functions

Having covered the coarse structure of the CnC graph, we now examine the fine-grained part of the computation: the step functions themselves. The hybrid

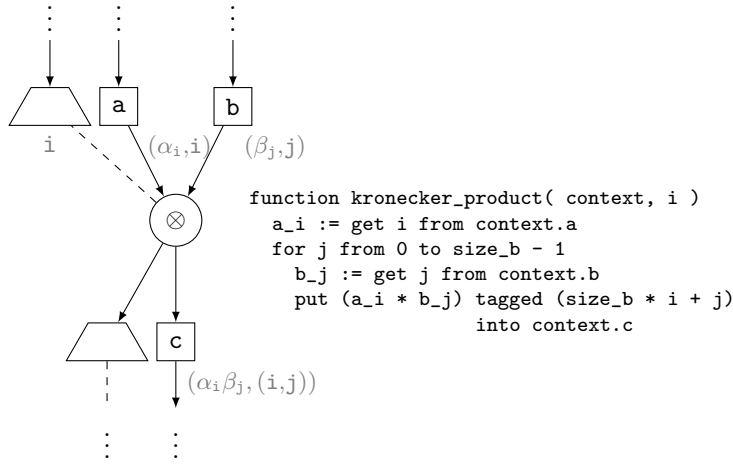
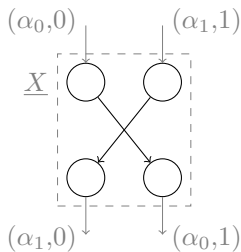


Figure 6.7: The CnC graph for the pipelined vector tensor product, also known as the Kronecker product. Each step instantiated by an i produces $\alpha_i \beta_j$ for all $\beta_j \in \mathbf{b}$.

nature of CnC requires that the execution inside the step functions is a sequential control-driven computation. The CnC graph structure leaves open if each step is fine- or coarse-grained. In order to more accurately demonstrate and validate the fine-grained dataflow approach, we would prefer to encode the fine-grained dataflow graph as presented in Section 5.3. But, for better efficiency on stock processors it is preferred to have coarser step functions. A coarse step function consumes and produces a range of amplitudes, rather than individual amplitudes, reducing CnC runtime overhead and better matching the coarse-grained parallel preference of stock multicore processors. We reconcile both roles for the empirical validation below. First, we express all steps as fine-grained, taking advantage where possible of all exposed parallelism. Afterwards, we introduce a coarse-grained step that is functionally compatible with the other steps. By ‘plugging in’ modular optimized step in the pipeline, we can compare the effect of the optimization. This is the corner stone of a quantitative approach to optimization and will be further explored in the empirical validation section below.

Single-qubit For the sake of clarity, we first present the single-qubit operation version of each step. The fine-grained graph of each operation was defined in Section 5.3 by Propositions 9, 10, 11 and 12. From these propositions, it can be observed that the operations X , Z and $\wedge Z$ can be implemented as one or more *monadic* operations: taking a single amplitude and directly producing a new

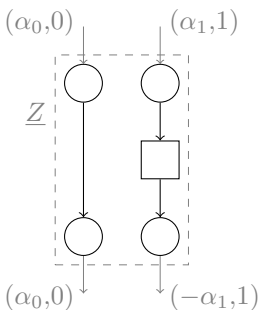
one. The M and \otimes operations are *diadic* in nature, requiring both dependent amplitudes to be available before producing a new amplitude. This distinction is important; the pipeline structure ensures that when a step is prescribed with a tag i , the amplitude α_i is available in the consumed item collection. The diadic operation step needs a CnC *step tuner* to guarantee the same. A fine-grained graph can be encoded as a control-driven step by branching the computation based on the incoming tag. For instance, the X and Z cases can be defined in pseudocode as follows.



```

step X( i )
  a := consume i
  if i even
    produce a tagged i+1
  else
    produce a tagged i-1

```

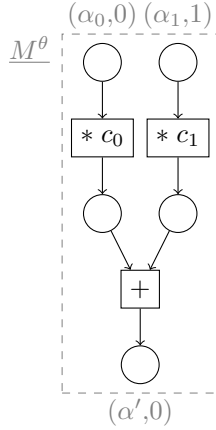


```

step Z( i )
  a := consume i
  if i even
    produce a tagged i
  else
    produce -a tagged i

```

The step for $\wedge Z$ is again similar to Z . The implementation for the dyadic operations \otimes and M are less trivial, as we have already seen for the \otimes case above. Take n the number of amplitudes in M 's input tangle. In the pipeline, the step M will be prescribed n times, although it should be only $n/2$ times where it consumes two items each time. As with \otimes , this type of joint dependency cannot be elegantly expressed in CnC. A simple solution is to do nothing on the 'odd' tags and perform the M computation on the 'even' tags, thus only producing a new item in the latter case. We put the fine-grained graph next to the M step pseudo code in the following:



```

step M( i )
  c0 ::  $\frac{1}{\sqrt{2}}$ 
  c1 ::  $\frac{e^{-i\theta}}{\sqrt{2}}$ 
  if i even
    a0 := consume i
    a1 := consume i+1
  produce (a0*c0 + a1*c1)
  tagged  $\lfloor \frac{i}{2} \rfloor$ 
    
```

Multi-qubit A central element in the M step above is the *differentiating condition*: the **if** condition in the above pseudocode. For one-qubit operations, checking for bit 0 or 1 is sufficient. Multi-qubit *parallel position* operations, such as Z_n^n , requires to check for even or odd tags, as in the pseudocode above. Previous chapter we introduced the stride permutation operator to realize any positional operation as a combination of parallel position and permutation operations. The same can be achieved in the CnC program, requiring the introduction of a monadic permutation function that consumes an item tagged i and produces the same item with the tag $p_{M,N}(i) = M(i \bmod N) + \lfloor \frac{i}{N} \rfloor$, using Equation (5.3.13). A fine-grained dataflow implementation does not require such an explicit permutation stage, we achieve the same effect in a different way. In practice, we realize the generalized position operation version of monadic steps by applying the p_{Mn} function to the tag of the consumed item. For instance, the step operator Z_{POS}^{SIZ} becomes:

```

step Z( i )
  a := consume i
  p_i := p(i, pos, siz)
  if p_i even
    produce a tagged i
  else
    produce -a tagged i
    
```

In other words, qubit position or size only has an impact on the *differentiating condition* of each step. From the point of view of a dataflow computational model, the permutation stage does not describe any operation, it describes a static property of the graph: how certain input/output edges are connected. In other words, true fine-grained dataflow execution performs the same operations

for instance for Z_2^3 as it does for Z_3^3 . In dataflow, there is no data stored in fixed addresses that need to be reshuffled. Data tokens conceptually just flow to a different destination. The differentiating condition is the way we encode this fine-grained dataflow into a control-driven execution.

Coarse-grained and completely control-driven implementations of the same operations require a separate stride permutation operation step, either as part of the communication or access pattern [56] or as a way to improve data-locality in memory storage by using the stride permutation's recursive properties [142, 158, 157]. The latter can often be observed in the context of the parallel Fast Fourier Transform algorithms [130], for which Pease [154] has demonstrated a stride-permutation formulation as early as the 60s.

Bit-pattern shortcuts A certain combination of features enables an alternative expression of index-manipulation operations such as the even/odd differentiating condition and stride permutation operations. The following elements are necessary for such a shortcut:

- integers as tags,
- a qubit quantum state representation,
- canonical vector ordering and
- an implementation environment with bit-level operations.

In Section 5.3.3 we made explicit the relation between a basis vector in *numeral tensor notation*, e.g. $|5\rangle$, and *tensor index notation* $|101\rangle$. The relation between both is naturally that the latter is the binary representation of the former. The tensor index notation is often used as it encodes the amplitude indices of the original qubits in the qubit tensor. We already encountered this in the realization layer in Section 3.6. Changing the position of the qubits has the effect of changing the indices of the amplitudes. Indeed, by using bit-level operations it becomes unnecessary to use the permutation operation directly. For example, the effect of the step X on its tag can completely and more efficiently be realized using the bit-wise NOT and bit-shift operations¹. The equivalence with the earlier defined step is illustrated in Figure 6.8. All other steps have a similar bit-pattern realization of their *differentiating condition*. This bit-pattern shortcut is to be seen as an optimization, not a starting point.

Coarse-grained optimization: the EMX step The coarseness of a parallel program was defined earlier as the amount of work performed between successive communication or synchronization. The fine-grained steps presented above

¹Respectively, these are \sim and \ll in the C programming language.

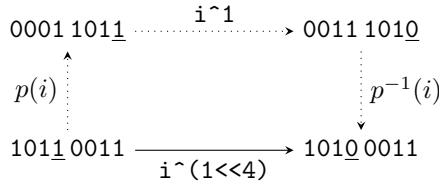


Figure 6.8: Fast Bit-level realization of the X step’s differentiating condition and index manipulation. Showing the equivalence with the permutation approach on a 8-qubit state example, targeting qubit position 4.

indeed only execute a handful of elementary operations for each amplitude, reflecting the sparse nature of the MC operators. Regarding the fine-grained steps as atomic parallel tasks, a coarsened step then logically bundles several of these step invocations. Such coarsening can happen in two directions: vertically and horizontally [11]. Vertical coarsening bundles multiple steps that depend on each other along the produce/consume pipeline. Horizontal coarsening bundles multiple invocations of the same step, consuming and producing multiple data elements. Vertical coarsening reduces overhead by increasing the amount of work that is performed for each produced item, but sacrifices pipeline parallelism. Horizontal coarsening trades-off data-level parallelism for overhead. We implemented a vertically coarsened step, for use in empirical performance analysis below and to demonstrate domain-specific optimization. The latter is made possible by the stratified compilation approach of the *mcc*. To our knowledge, only horizontal coarse computations have been used in parallel quantum computing simulation related work.

In the MC, the \mathcal{J} -pattern is a basic building block for creating larger patterns. Indeed, examining the Coarse Graph of non-trivial wild patterns reveals an often repeated produce/consume sequence: a tensor product with a fresh one-qubit tangle, followed by entanglement, measurement and lastly an X -correction operation, or in short: **EMX**. The *mcc* is used to detect the sequence during the specific compilation phase and replace all CnC nodes involved by the coarser **EMX** step collection. We have worked out a visualization in Figure 6.9 of the effect of this coarsening optimization on both a coarse and fine-grained graph example. The **EMX** step takes the same outward consume, produce and prescribe dependencies as the pipeline of steps it replaces. Each item prescribing the \otimes step at the front of the sequence will thus, after coarsening, prescribe the **EMX** step. The coarse step encodes the fine-grained graph of the entire sequence, producing at the end a single item. Both the tensor and measurement steps are dyadic, which would mean the coarse step encoding would require to consume four items. But, one of the two consumed item collections for the \otimes step is a fresh tangle that is

also included in the sequence. This reduces the number of consumed items to two because of the M step. Logically, any coarsening optimization decreases the exploitable parallelism. When the ratio between average parallelism and number of processing elements is still high enough after the coarsening, the parallel performance should remain unaffected.

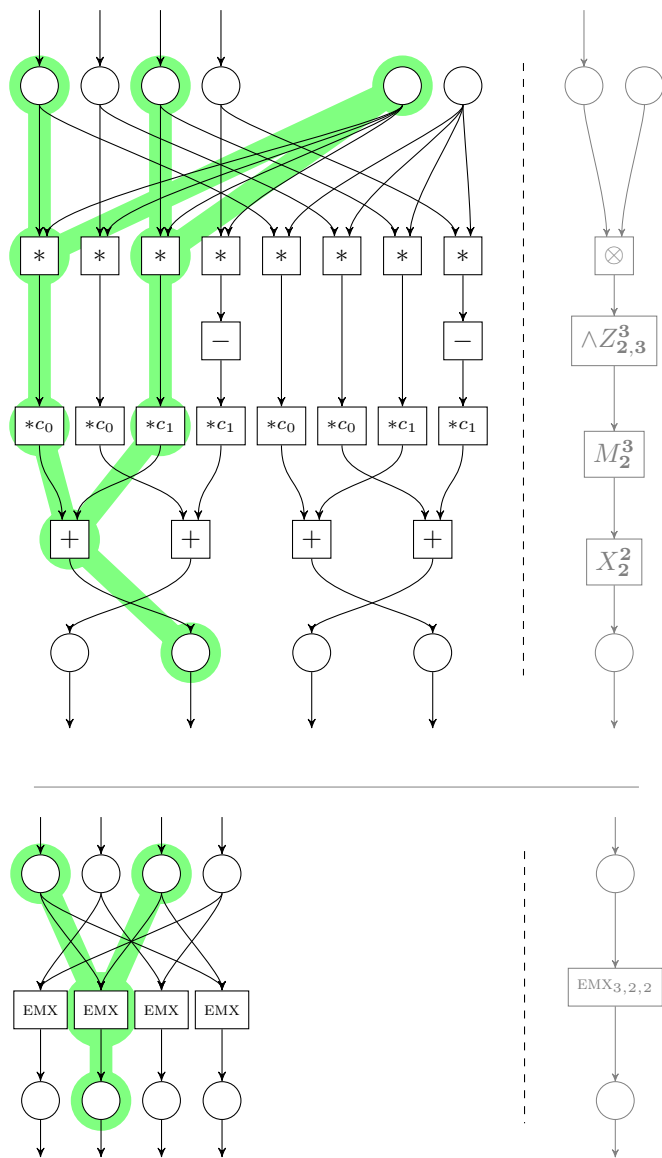


Figure 6.9: Visualization of the coarsening optimization that merges several coarse operations into one. Above: fine-grained (left) and coarse-grained (right) graphs of a \mathcal{J} -pattern acting on a two-qubit input tangle. Below: the same graph after the coarsening optimization. The highlighted nodes and edges follow the dataflow of a single output node; all nodes in such a single ‘trace’ are implemented by the coarsened EMX CnC step. For compactness, non-input amplitude and tangle nodes have been omitted.

6.3 Experimental Validation

6.3.1 Goals

The implementation artifact serves three main purposes.

- Demonstrate the feasibility of the parallel approach.
- Substantiate the results of the theoretical analysis with empirical analysis.
- And, provide a test bed for practical and theoretical developments.

Our implementation is a proof of concept, demonstrating by construction that indeed the MC can be realized as a fine-grained dataflow computation. This is not self-evident, the theoretical analysis from Chapter 5 can be used for a more traditional purely data-parallel implementation, as in parallel QC related work. Indeed, a traditional approach would likely perform better on today's stock processors. However, the goal of this implementation is not to get optimal performance on current stock hardware, but rather validate that our fine-grained dataflow approach is indeed feasible and exposes the promised parallelism. Therefore, our implementation approach follows the fine-grained dataflow semantics as closely as is practical. We do this to demonstrate the feasibility of our dataflow approach not just as a theoretical analysis tool but also as execution platform.

The theoretical analysis earlier this chapter reveals a vast amount of available parallelism, enough in theory to allow the simulation of additional qubits by an exponential increase in hardware resources. This abundant parallelism should be visible in a real world implementation in the form of parallel speedup, even in a high-overhead naive implementation. We show that our parallel MC implementation exposes and exploits inherent parallelism by using speedup as empirical quantitative measurement. Speedup is used as a relative performance measure, comparing different instances of the same program to measure parallelism. For a fair performance analysis, a parallel implementation should also be compared to a representative sequential algorithm. Our sequential implementation `qvm` was implemented for this purpose, using the low-level programming language C and *libquantum* to offer a fast sequential implementation as baseline comparison. As an aside, we show that having a fine-grained dataflow description of the computation exposes information that can be used for tuning optimizations. Furthermore do we demonstrate by introducing a coarsening optimization that that overhead is the dominant factor in the naive proof of concept implementation.

The `mcc` implementation artifact is designed to be an execution platform for the measurement-based Quantum Programming Paradigm. However, the goal is not simply to implement a QVM, but also to support experimentation with parallel computing. Developing practical parallel implementations requires a fair

quantitative performance analysis, measuring the effects of various optimizations and alternative approaches. The QVM offers a fixed implementation target: it described the functionality that needs to be implemented and enables multiple implementations to share the same input program, which is invaluable for comparing performance. This already allowed us to compare the sequential `qvm` implementation with the parallel `mcc`, but also to test variations on the `mcc` implementation and try alternative approaches and optimizations. Indeed, this proof of concept implementation artifact is to be seen as the starting point for developing a high-performance parallel QVM. Quantum computing simulation is an interesting case study from the parallel computing point of view, it contains a vast amount of parallelism, but its computational state cannot be trivially partitioned. We envision further research using the `mcc` as parallel computing case, which we further elaborate in the next section under future work.

For the demonstrator and test-bed goals above, the implementation itself forms the experiment and validation. The second goal, substantiating the theoretical analysis, still needs to be shown through empirical analysis.

6.3.2 Approach

We use experimental results to support two broader statements: our proof of concept implementation automatically exposes and exploits parallelism, and the current implementation has plenty of optimization headroom. We split these into several more concrete indicators supporting these statements. Indicators that can be verified experimentally.

- **Statement 1:** We automatically expose and exploit parallelism, with our proof of concept implementation already demonstrating good parallel performance.
 - **Indicator 1.1:** Parallel speedup scales positively with the number of processing elements.
- **Statement 2:** The current proof of concept implementation is wasteful, but contains a lot of optimization headroom. Given enough engineering effort, this can be exploited to increase absolute real-world performance.
 - **Indicator 2.1:** The execution runtime grows proportional with memory use.
 - **Indicator 2.2:** The naive implementation is careless with memory.
 - **Indicator 2.3:** Small tweaks that reduce the overhead of CnC steps offer a small parallel performance improvement.
 - **Indicator 2.4:** Increasing parallel task granularity has the largest impact on absolute performance.

The first statement is tied to the theoretical analysis above, which already shows that we expose a vast amount of average parallelism that *can* in practice lead to good parallel performance. By construction, our proof of concept does so automatically; the `mcc` takes the same program input as `qvm` and automatically produces parallel program code. We still need to show with experimental results that parallelism is exploited in practice. This is achieved with Indicator 1.1, by measuring the *parallel speedup* metric as an indicator for parallel performance. Speedup is the most commonly used parallel performance metric, it compares the execution time on a single processing element T_1 with the execution time on an n -number of processing elements T_n :

$$S_n := \frac{T_1}{T_n} . \tag{6.3.1}$$

Parallel speedup is typically graphed by scaling the number of processing elements n . From this graph, the scaling behavior of the parallel algorithm can be observed. An ideal parallel implementation has $S_n = n$ for any n , called linear speedup². In practice however, each processing element adds more overhead to the computation, leading to cannon ball trajectory graphs. In other words, the closer the speedup graph is to the diagonal, the better. As we will show below, our implementation already shows good speedup scaling.

Statement 2 uncovers why the ideal speedup is not obtained: overhead and other implementation factors. Recall from the discussion in Section 4.4.2 that the extreme fine-grained approach used in the current proof of concept implementation is expected to carry a very large overhead in various areas. However, we also mentioned that various optimizations exist that remove this overhead. With Indicators 2.1 and 2.2 we show that the maximum memory use metric is substantially higher in the parallel case compared to that of the sequential `qvm` implementation. Indicator 2.3 and 2.4 demonstrate that optimizations can indeed have a large impact, indicating that engineering effort can make the fine-grained approach competitive even on current stock hardware.

Experimental setup

Our proof of concept implementation is programmed to run on current multicore processors, today's stock hardware. The test-bed computer system we have used in the experiments uses two 2.26 Ghz Quad-Core Intel Xeon multicore processors, allowing us to scale up to eight effective hardware threads. These two processors

²Superlinear speedup can happen in reality [105], although rarely. Superlinear speedup effectively means that the total amount of work decreases when adding more processing elements. This happens in roughly three cases; most often because of an implementation mistake, in search algorithms such as random walk or backtracking, and because of the increased collective memory and cache size.

share the same main memory and are thus connected in a Uniform Memory Architecture (UMA) fashion. i.e. each processor can access any memory address within an equivalent time frame. The total system memory size is 8GB of RAM, with per processor cache sizes of 256KB (L2) and 8MB (L3).

As benchmark program, we use the $QFT(n)$ measurement pattern. The Quantum Fourier Transform is the basis of existing quantum computing algorithms, which means it often appears as a benchmark application. The main benefit of the $QFT(n)$ for our experiments is that $QFT(n)$ can be used for both strong and weak scaling performance measurements. *Weak scaling* scales the problem size while keeping the number of processing elements fixed. *Strong scaling* scales the number of processing elements while keeping the problem size fixed. For strong scaling we use $QFT(16)$, it constitutes a high enough workload in both sequential (`qvm`) and parallel (`mcc`) cases, with runtimes in the order of seconds. In weak scaling experiments, we can increase the workload by scaling the $QFT(n)$ from $n = 2$ to $n = 16$.

6.3.3 Experiments

Parallel speedup

- **Indicator 1.1:** Parallel speedup scales positively with the number of processing elements.

Parallel speedup, as already mentioned, is calculated by dividing the wallclock execution time of the program with one processing element by that of n processing elements. In practice, the execution time of the same program can vary between multiple runs, due to a wide range of factors. Each execution run can have small variations in thread scheduling, processor instruction scheduling, virtual memory use, etc. We therefore run each experiment a number of times in order to offer more statistically valid performance measurements [87]. We have experimentally established that twenty execution runs result in a high enough confidence interval ($> 90\%$) for all benchmarks presented here. The twenty wallclock runtimes of each experiment are presented using a violin plot, a variation of the boxplot that simultaneously visualizes the usual quartiles (black rectangle), mean (white dot) and probability density (violin shape). The results in Figure 6.10 were obtained from compiling the $QFT(16)$ pattern in our parallelizing compiler `mcc`, producing an executable that was run repeatedly. Each wallclock time T_n is measured at the operating system level with microsecond accuracy. The parallel speedup value $S_n = \frac{T_1}{T_n}$ for each run is calculated with T_1 being the average of all $n = 1$ runs, producing Figure 6.10.

It can be observed from the increasing parallel speedup in the figure that each additional processing element or thread does indeed improve performance.

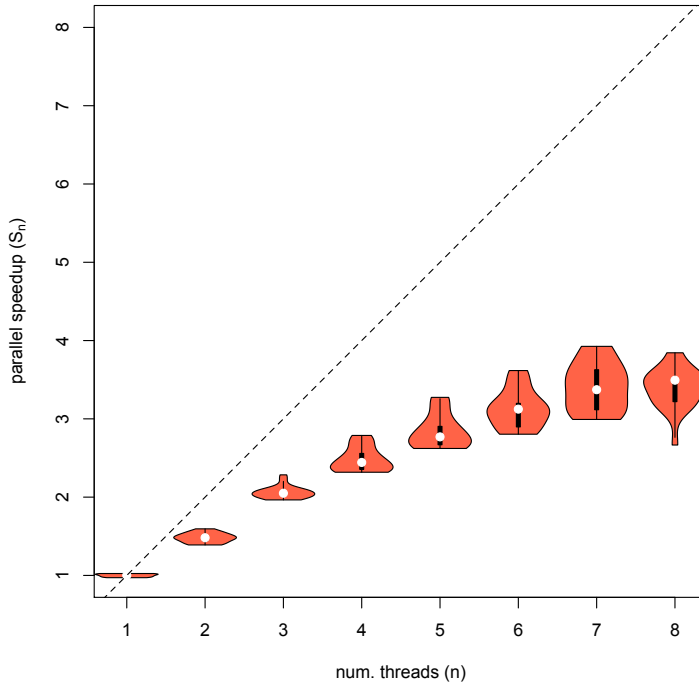


Figure 6.10: Parallel Speedup S_n values of multiple execution runs for the $QFT(16)$ measurement pattern, using the un-optimized fine-grained parallel implementation. The diagonal line represents ideal linear speedup.

But, as was expected, this increase tapers off. The factors contributing to this behavior of the speedup scaling graph can be varied and complex. Even for a proof of concept implementation, it is useful to analyze the underlying factors leading to the observed behavior. First, there is the task coarseness; each fine-grained task performs only a small amount of computation compared to the scheduling and task switching overhead. The effect of changing coarseness is measured below. Second, each processor has computing power and memory bandwidth limitations. Even under ideal circumstances, performance measures will grow close to some horizontal line as they come under the influence of a computing or memory bottleneck. We know from last section that the amount of computation per amplitude is relatively low, which means that memory bandwidth is the likely bottleneck. We measure the effect of memory use next.

Memory use

- **Indicator 2.1:** The execution runtime grows proportional with memory use.
- **Indicator 2.2:** The naive implementation is wasteful with memory.

Our sequential implementation `qvm` can explicitly manages the memory used to store the amplitude vector. This is a benefit of working with an imperative language where programmers can manually manage memory. Moving away from manual memory management means relying on the underlying implementation to manage storage effectively. For instance, saving memory by avoiding duplication and recycling memory that is no longer used. For the purpose of this discussion, we consider the memory use of the `qvm` to be the lower limit. Conceptually, parallel execution will use more memory on the whole, as it often needs duplication to perform simultaneous computations. The majority of the memory in our parallel implementation is used in CnC's item collections and step instantiating. The memory used locally by step instances is effectively recycled by CnC, it knows when a step instance finished executing. Item collections however retain the stored amplitudes even after they have been consumed. In other words, CnC holds the history of all intermediate states in memory in our unoptimized `mcc` implementation.

While memory use is not a direct measure of performance, it can be an indicator when comparing two different implementations. We compare absolute runtime measures with maximum memory usage of several workloads, both for the parallel version produced by the unoptimized `mcc` and the sequential `qvm`. Both wallclock runtime and maximum memory use measurements are taken at the operating system level. We graph both in Figure 6.11 on a linear scale, highlighting the order of magnitude difference. For Indicator 2.1 we indeed see the parallel runtime following memory use closely enough to be virtually overlapping. The gap of at least an order of magnitude between the sequential and parallel versions strongly demonstrate Indicator 2.2.

Effect of optimizations

- **Indicator 2.3:** Small tweaks that reduce the overhead of CnC steps offer a small parallel performance improvement.
- **Indicator 2.4:** Increasing parallel task granularity has the largest impact on absolute performance.

Having identified coarseness and memory use as two potential bottlenecks, we test this assumption by measuring the effect of optimizations for both bottlenecks.

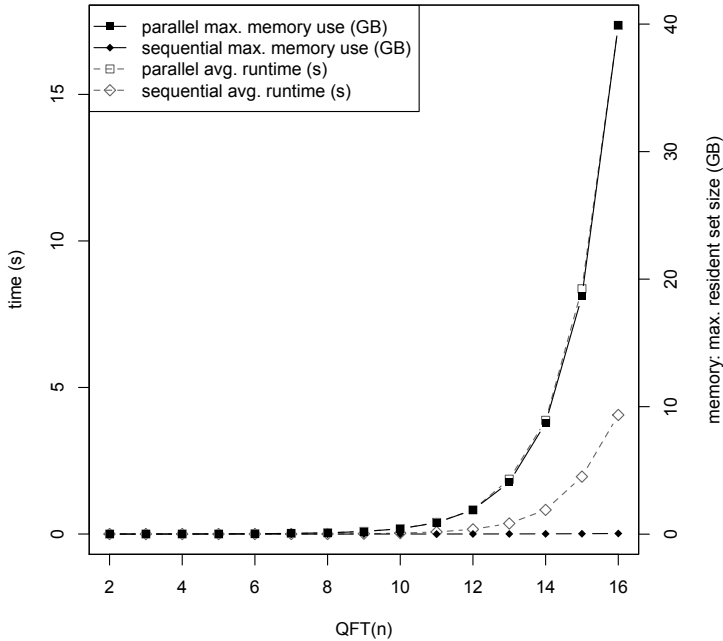


Figure 6.11: Maximum memory use compared to runtime for different $QFT(n)$ of both the sequential `qvm` and unoptimized parallel `mcc` implementation. Runtime is figured on the left axis and memory use on the right.

Optimizing for memory is hard to do directly in our implementation, as the Intel CnC and TBB libraries abstract away their memory management. CnC does offer *tuning* functionality, in which the programmer supplies additional information to the CnC runtime. We have added *step* and *item* tuners to the code generated by the `mcc`. A CnC *step tuner*, as mentioned earlier, declares the data dependency of each step instance to the scheduler. This information can be used by the scheduler to start executing a step instance when all its dependencies are ready, avoiding the situation in which a steps are suspended while waiting for their inputs. An *item tuner* on an item collection can declare how many times its items get consumed, in order to manage better the allocated memory. However, we notice in practice that no item collections currently get deallocated or downsized, which brings little benefit to our current use case. Any reported benefit of tuning

is chiefly due to step tuners. To demonstrate Indicator 2.3 we add both step and item tuner optimizations to the computation of $QFT(16)$. In Figure 6.12

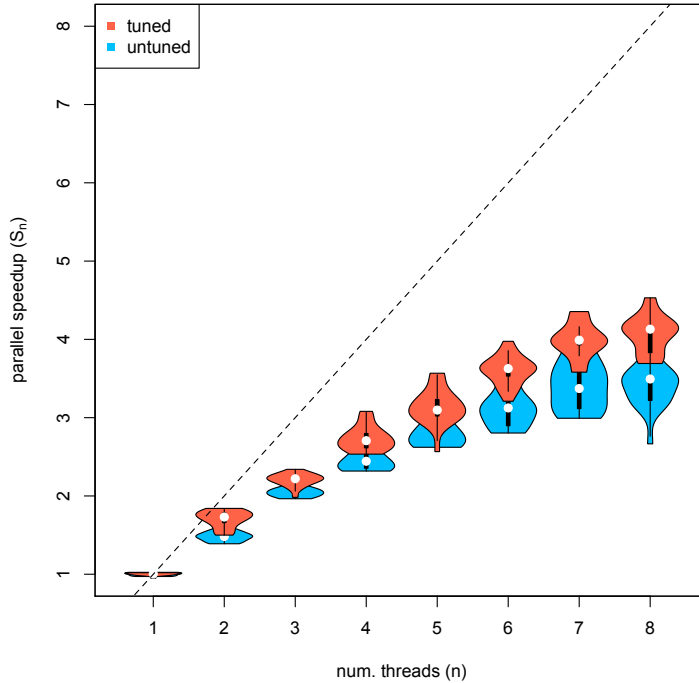


Figure 6.12: Effect of introducing CnC tuner optimizations, reducing the overhead of individual parallel step instances.

we superimpose the earlier reported speedup characteristics of the naive parallel implementation with the tuned version. We observe that tuners improve the speedup scaling behavior, as expected. Step tuners decrease the overhead of individual step instantiations, thus decrease the total parallel overhead. However, the parallel overhead is evidently not the main bottleneck; we observe a small but not radical shift of the parallel scaling graph towards linear speedup. This is likely an indicator that parallel overhead is dominated by the number of scheduled step instances, rather than the amount of overhead for each individual one. This is tested in the following by introducing a coarsening optimization, which reduces the overall number of step instances.

The coarsened step optimization was introduced at the end of the last sec-

tion. This coarsened step replaces several consecutive steps, effectively reducing the total amount of parallel step instances. Coarsening decreases the amount of parallel work and increases the amount of sequential work. Conceptually, the total amount of work remains the same, but in practice extra work is added in the form of parallel overhead. We thus get an indication of the amount of parallel overhead of our implementation by comparing the execution runtime of the unoptimized version with the coarsened versions. This is reported in Figure 6.13, in which we added the average sequential runtime speed of the sequential `qvm` implementation as reference point. This graphic clearly shows a large absolute

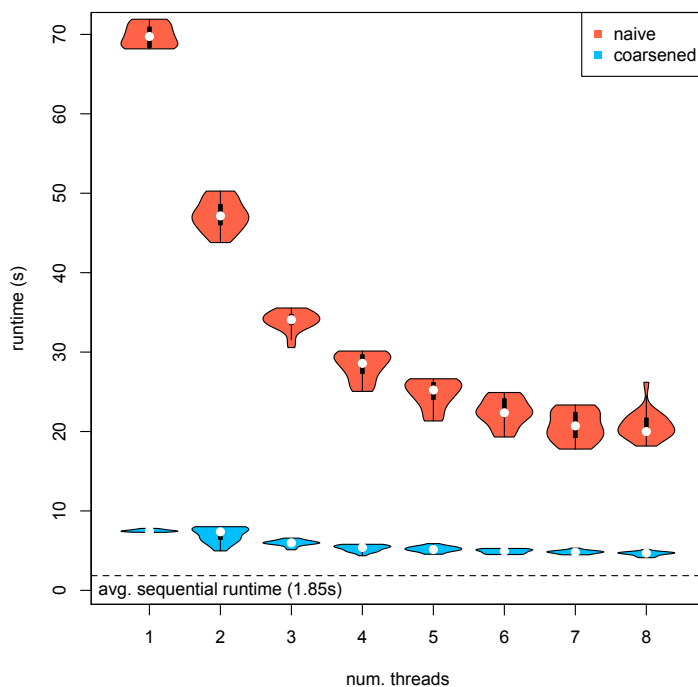


Figure 6.13: Comparing the absolute runtime speeds of the original unoptimized parallel implementation with the tuned and coarsened optimized version. Both execute the same $QFT(16)$.

performance win from introducing a small coarsening optimization. As we have discussed, this indicates parallel overhead from the fine-grained granularity to be the dominant performance factor in our proof of concept implementation. This

overhead is an issue of CnC, which is designed around coarser steps, but mainly it is an issue of the multicore execution platform.

6.4 Conclusion

We have approached the validation of the theoretical work from Chapter 5 on three fronts. First, we provided a theoretical analysis showing that the characteristics of the parallel program resulting from the fine-grained graph do indeed expose abundant parallelism. Next, we have built an implementation artifact, a parallelizing compiler `mcc` that implements a QVM: taking a command sequence and producing program code that executes in parallel. Last, we measure some common performance metrics. Although we can show good parallel performance, it cannot beat our best sequential implementation in absolute runtime performance. We want to stress again that this does not invalidate our approach. Behind the `qvm` implementation sits an entire set of optimized libraries, compilers and processors tuned to its type of workload. The `mcc` by comparison, sorely lacks in back-end optimization. We took a conceptual approach for the `mcc`; expressing the computation in a vastly different way to expose a maximum of parallelism, then worrying about optimization later. Essentially, we simulated a dataflow machine. As the shift towards more parallel software and hardware continues and compiler and framework technology matures, it will become more important to express as much parallelism as possible, rather than start from a sequential point of view and identify what parts can be parallelized. Indeed, the related work presented in Chapter 4 teaches us that many of the optimizations required to make parallel computation fast, such as our tuning and coarsening, achieve better results when left to a computer.

Some domain-specific parallelization projects such as StreaMIT [186], FORMLESS [103] and SPIRAL [157] are good examples of that. The SPIRAL framework is especially close to our work considering the related mathematics of their application domain (DSP) and their approach. A Digital Signal Processing problem, such as FFT, is expressed in SPIRAL using constructs in a high-level mathematical model. This model captures transformations rules similar to what we have used in the previous chapter: tensor product decomposition, commuting using stride permutation and parallel-position operators. These transformation rules are used to optimize the dataflow patterns of algorithms. The mathematical constructs are then code-generated into a Single Static Assignment (SSA) form to allow for and simplify common code-optimization algorithms. Although SSA is used as a medium that more easily exposes dataflow information, this information is only used to enable local and sequential optimizations. SPIRAL does not focus on parallel computing, but does take a first step in that direction by translating some constructs into a parallel loops. In contrast, we transform the same high-level mathematical constructs into an *explicit* dataflow graph. This graph is not used as a convenient intermediate representation, but as the final program representation. In other words, much like SPIRAL and FORMLESS,

we transform high-level mathematical constructs into a dataflow form to expose the parallelism in the program. However, we directly exploit this parallelism by way of executing the dataflow graph, whereas the related work uses the implicit dataflow information to produce more efficient sequential programs.

The `mcc` is to be seen as a starting point and proof of concept; built from the ground up to produce highly scalable and massively parallel programs that, while not competing with the best sequential implementations today, can with enough development effort take maximal advantage of highly parallel hardware.

Chapter 7

Conclusions and Future Work

In this dissertation, we have brought together two fascinating and rapidly developing research domains: quantum computing and parallel computing. By bridging these domains, we were able to make a number of contributions. In this concluding chapter, we give an overview of our work by domain in respectively Sections 7.1 and 7.2, highlighting our various contributions in Section 7.3. Much of the work presented here is a first step, providing proof of concept and laying down a foundation to support further research. We consequently propose in Section 7.4 further research based on the work presented here.

7.1 Quantum Programming

In this work, we proposed and deployed a quantum programming framework, based on the formal Measurement Calculus. Rather than directly implementing MC’s semantics, we proposed a layered architecture. The purpose of designing around multiple abstraction layers is to make the framework more robust to future changes. This was already helpful during the development of our practical¹ Framework, allowing us to experiment with different approaches and implementations. As presented in the future work section below, we wish to integrate various conceptual tools developed around the Measurement Calculus, with the ultimate goal of supporting the analysis interactive and development of new quantum computing applications.

As part of the development of our practical quantum programming framework, we made several contributions. At the application layer abstraction level, we used the insight that pattern composition has a natural graph representation to build a pattern editor graphical user interface. Using this application, large patterns can be intuitively and automatically combined, taking full advantage of the pattern abstraction’s modularity and compositionality. Automated pattern composition is not performed by the application itself, but rather by a separate pattern abstraction layer. We devised a declarative pattern composition structure and an automated composition process that subsumes the two original composition rules of the formal Measurement Calculus. Low-level operations of the Measurement Calculus form a natural ‘assembly language’. We put this into practice by building a separate Quantum Virtual Machine (QVM) layer that uses machine-readable expressions of MC operations as instructions. An optimized sequential implementation of this virtual machine in a standard programming language forms our reference implementation of the QVM. We introduced a number of common but effective optimizations, such as sparse matrices, bit-pattern functions and the representation of the quantum state as a set of separate factor states, which we call tangles. We observed that MC patterns in their standardized form, in which all entanglement operations are performed first, fail to take advantage of this crucial optimization.

¹Practical, as opposed to formal.

7.2 Parallel Execution

We have formulated the execution of the QVM as a parallel computation, to improve its efficiency on highly-parallel computing hardware that lays ahead. Our approach was to start from a relatively coarse parallel data-driven model and refine it through a series of model transformations into a fine-grained dataflow model of computation.

The operational semantics of the formal MC model is essentially sequential: its state transition rules operate on a single global quantum state. However, this global state can be split into several local states by representing a tensor-factorizable quantum state as a set of its factor states. As operations on factor states can act independently from each other, the MC transition rules can – with slight modification – simultaneously be applied to different factor states. Our coarse-grained parallel graph model is formed by building a graph of these state transitions.

In order to parallelize the action of MC operations on a quantum state, we first had to make them more concrete. In a concrete linear algebra approach, the positions of qubits matter more than their qubit name. To capture this sensitivity to position, we introduced the notion of a positional operator. An MC operation is turned into a positional operator by simply looking up the position of its target qubit. A positional operator acting on the ‘last’ qubit position, which we call the parallel position, has a simple concrete representation in which the qubit operator is continuously repeated. In other words, the action of such a parallel position operator on the entire quantum state vector can be described as a series of actions on smaller parts of the state vector. Closer inspection of positional operators reveals that different target positions result in the same quantum state vector, albeit with the elements in a different order. A particular class of recursively decomposable permutations captures this difference, such that by applying the correct permutation changes the target position. By introducing a permutation operator, we can formulate all MC operations as concrete parallel position operators. We obtain the fine-grained dataflow graph of a parallel positional operator by expressing the single-qubit version of the operator as a dataflow graph, with arithmetic operations on amplitudes as nodes, and by concatenating multiple instances of this graph. Combining this dataflow graph with the graph for the correct permutation achieves any other positional variant of the operation. A complete MC computation can be described as a fine-grained dataflow graph by connecting the various positional operators’ graphs following the coarse-grained graph structure.

We implemented the various transformation steps described by building a compiler. However, lacking a true dataflow execution platform, we turned to emulating the parallel execution of the fine-grained dataflow graph using a modern

parallel computing library: Intel Concurrent Collections (CnC) [128]. Because of CnC’s dataflow-like programming model, we were able to achieve true fine-grained dataflow execution semantics with only few compromises. The compiler artifact takes a quantum virtual machine instruction sequence and ultimately produces CnC-using C++ source code, which can then be compiled into a program executable on a multicore processor. This approach to parallel execution is a compromise, balancing at one end the need to demonstrate the conceptual fine-grained approach, and at the other end the notorious practical difficulties and hard to predict performance characteristics of parallel programming with respect to parallel hardware. Concretely, we avoided the risks associated with ad hoc parallel implementations and specialized parallel hardware architectures by choosing a parallel software framework that: encapsulates this complexity, offers us correctness guarantees and offers sufficient freedom in determining the parallel execution strategy. However, this limits the amount of parallel processing power, making purely experimental validation difficult.

As validation of our parallel approach, we performed both a theoretical and an experimental analysis of the fine-grained dataflow graph. Theoretical analysis of the average parallelism metric shows that the dataflow graph indeed exposes a vast amount of exploitable parallelism. We saw that for the Quantum Fourier Transform case, an exponential increase in total work is matched by an exponential increase of exposed parallelism. In the experimental analysis, we measured the real-world performance of executing large Quantum Fourier Transform computations. Analysis of the results show good parallel speedup. Although, as a consequence of our compromise in the parallel execution approach, the speedup is not ideal and the absolute performance falls short of the optimized sequential implementation. We show that this difference in absolute performance is indeed due to inefficiencies: experiments that introduce simple optimizations demonstrate dramatic performance improvements.

7.3 Contributions

For the sake of convenience, we summarize our main contributions in the following, separating the conceptual contributions from their concrete realization.

Conceptual contributions

- The formulation of Measurement Calculus' virtual execution as a fine-grained dataflow graph.

The mapping of the MC operations to a fine-grained dataflow graph exposes the vast parallel potential inherent in the classical execution of the MC's quantum operations.

- The formulation of the Measurement Calculus execution semantics into a coarse-grained parallel execution.

By representing quantum state as a set of factor states, the execution semantics of the MC can be modified to allow the simultaneous execution of certain operations.

- A layered architecture design for an MC-based quantum programming framework.

The organization of the MC as a stack of abstraction layers leads to a software organization more robust to implementation changes. Each abstraction layer is formulated such that it is independent and straightforward to automate.

- The formulation of a virtual machine for a measurement-based quantum computer.

The operational semantics of the MC is used to formulate the instruction set of a Quantum Virtual Machine.

Practical contributions

- The fine-grained and data-driven parallel execution of Measurement Calculus programs.

In this idealized parallel execution approach, arithmetic operations on individual amplitudes can be executed simultaneously. We demonstrated parallel speedup by profiling such parallel execution of a large MC program.

- A parallelizing multi-stage compiler framework for MC programs.

The parallelizing compiler artifact parses an MC program and transforms it over multiple increasingly-concrete intermediate representations. Ultimately, the compiler generates standard programming language source code which will execute the MC program as a parallel computation.

- A hand-crafted and optimized Quantum Virtual Machine implementation.

To represent the best available sequential execution of the Measurement Calculus, we produced a fast sequential implementation of the Quantum Virtual Machine.

- A programming framework for the Measurement Calculus; including a graphical pattern editor application, automated pattern composition and virtual execution through the Quantum Virtual Machine.

This framework is a concrete software realization of the layered architecture. The graphical pattern editor application acts as the framework's front-end. This application allows users to express complex pattern compositions by visually linking input and output qubits.

- The analytical and experimental profiling of a fine-grained highly-parallel formulation of the Measurement Calculus, indicating a large parallel performance potential.

We integrated analytical tools into the parallelizing compiler framework to collect parallel performance metrics: total work, critical path and average parallelism. These indicate that for large quantum algorithms, the exponential increase in work is matched by an exponential increase in exploitable parallelism.

7.4 Future Work

7.4.1 Highly-parallel performance challenge

The work we have presented here still has one open challenge, the question: *Does the parallel performance of our fine-grained dataflow approach scale for massively parallel computers of the future?* We have presented only indications that this is indeed the case: dataflow approaches have scaled well in the past [98] and a theoretical analysis that shows that the fine-grained dataflow graph indeed exposes a vast amount of parallelism. Our experimental validation however is mainly demonstrative in nature and runs on a parallel machine of modest scale. This was, as discussed earlier, a compromise in order to deal with several critical difficulties. Our next research step is thus to fully experimentally validate our fine-grained dataflow approach by demonstrating good scaling on a more highly-parallel and advanced parallel computer architecture. Using the existing artifact as a blueprint, we can more accurately plan new software implementations and adapt it for target highly-parallel architectures. This work can be seen as a large case-study, investigating a parallel approach off the beaten track. We are in contact with parallel computing peers and see interest in the dissemination of several facets of this work. In summary, having demonstrated the viability of a dataflow approach, we can take the next step and focus on achieving scalable parallel performance on cutting edge large-scale parallel system.

7.4.2 Expanding the quantum programming framework

With this work, we have sought to address the shortage of pragmatic quantum programming frameworks. Because of our pragmatic approach, we have left open important opportunities to further improve the usefulness of this framework. To goal of the following improvements is to aid in the dissemination of this quantum programming framework, by making it of practical use for researchers using the measurement-based quantum computing model.

The pattern abstraction layer currently automates the pattern composition and the pattern compilation to Quantum Virtual Machine instructions. However, other useful transformations and analyses tools at the level of measurement patterns exist. Most evidently, the Measurement Calculus' standardization process. Several other conceptual analysis and transformation tools have been suggested: *flow* analysis [55] and depth complexity [36]. These techniques can be included in the pattern layer abstraction to automate the optimization and analysis of newly created patterns.

The MC's standardization process was not included in the pattern layer out of practical considerations: a measurement pattern in standard form is harder to

execute in a simulated environment. We plan on developing a pattern transformation process with the goal of maximizing the potential for simulated execution. Rather than bringing all entanglement operations to the front, as in the standard form, this process will move each entanglement operation as close to the measurement operation that will destroy its qubits. This anti-standardization process will thus seek to minimize the quantum state size during execution.

At the application layer, we wish to improve the power of the visual language used to compose patterns. As we have seen in Chapter 3, some patterns have a repeating or even recursive structure. The library application approach can capture this recursion using the functionality of the host programming language, whereas the editor can only express an instance of such higher-order patterns. We expect that iterative or even recursive pattern compositions can be added as visual metaphors; for instance, by using special edges that loop backwards, creating a controlled violation of the acyclic graph rule.

7.4.3 Dissemination

Our contributions relating to quantum programming were in large part already presented to the relevant research community in [67]. The experimental and unorthodox nature of parallel computing contributions required a certain implementation and engineering maturity of our research artifacts (parallel compiler, runtime, etc.) in order to sufficiently demonstrate satisfactory results. We have now come to a point where we feel our contributions to the parallel computing domain can be disseminated. Indeed, we see interest in our the current parallel work as both an investigation of high-performance parallel quantum computing, and as an unconventional approach to parallel computing using fine-grained dataflow.

Appendix A

Linear Algebra formulation of Quantum Computing

This appendix serves to give readers unfamiliar with the topic a brief overview of the mathematical formulation and notational conventions as used in the Quantum Computing domain. We adhere to the notation as introduced in the textbooks by Nielsen & Chuang [150] and Gruska [97]. For a more detailed treatment of the subject, we refer to these standard works.

Quantum mechanics is typically formulated using linear algebra. The postulates of quantum mechanics, established over decades of experimentation, describe the behavior of elementary particles such as atoms, electrons, etc. For the purpose of quantum computing, it is sufficient to use a discrete formulation of these postulates. That is, we can use a simple operator formulation instead of more complex time-dependent functions.

The first postulate of quantum mechanics tell us that the state a quantum system is in can be completely described by a vector in a vector space. More concretely a quantum state is a unit vector in a complex vector space that has an inner-product. All possible states of a quantum system form a complex Hilbert space. Why this is the case, is outside the scope of this discussion, but we can impart a rough intuition on why these conditions are necessary. The *complex* coefficients are required to elegantly describe the quantum state and its evolution. The *inner-product* requirement provides, roughly speaking, a unit of distance or *angle* between vectors. The *unit vector* requirement isn't strictly necessary, but restricting state with this *normalization condition* to vectors of unit length or *norm* simplifies certain calculations.

The simplest non-trivial quantum state that can be represented is the *qubit*. It is used as the basic unit of information, much as 0 and 1 are used in classical

computers. Any qubit state can be described as a linear combination of two orthogonal vectors or basis. Typically, the computational basis ($|0\rangle, |1\rangle$) is used. Throughout this dissertation, we use the *Dirac* notation for vectors, using $|\psi\rangle$ to denote a column vector named ψ . The vectors of the computational basis are in the familiar column vector notation $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Formally, any arbitrary qubit ψ can be written as the linear combination:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{A.0.1}$$

with the *amplitudes* $\alpha, \beta \in \mathbb{C}$. We will use the Dirac and column vector notations interchangeably, for example expressing the above $|\psi\rangle$ vector as $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$. The Dirac notation $\langle v|w\rangle$ denotes the inner product of both vectors ($|v\rangle, |w\rangle$). The notation $\langle\psi|$ represents the complex conjugate and of the vector $|\psi\rangle$:

$$\langle\psi| = [\alpha^* \quad \beta^*] = \begin{bmatrix} \alpha^* \\ \beta^* \end{bmatrix}^T = (|\psi\rangle^*)^T ,$$

where $*$ stands for the complex conjugate. The normalization condition constrains the *norm* of any qubit such that

$$\| |\psi\rangle \| = \sqrt{|\alpha|^2 + |\beta|^2} = 1 . \tag{A.0.2}$$

A quantum state is actually associated with a ray in a Hilbert space, rather than a single vector, meaning that quantum states are equal up to a global phase. We use this occasionally to produce a more compact notation, for instance $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is essentially the same quantum state as $|0\rangle + |1\rangle$.

The second postulate formulates the evolution over time of a closed and isolated quantum state as a *unitary transformation*. A quantum state $|\psi\rangle$ that evolves over a discrete time step into $|\psi'\rangle$ can thus be described by

$$|\psi'\rangle = U|\psi\rangle$$

where U is a *unitary* matrix operation. The most common unitary operations we use in this work are the Pauli-X and Pauli-Z unitary operations, which operate on a single qubit:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{A.0.3}$$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} . \tag{A.0.4}$$

The X operation will flip a $|0\rangle$ state into $|1\rangle$ and vice versa and is therefore often called the quantum-NOT operation.

The third postulate deals with *measurement* of a quantum state. There is no classical analogue for such measurement operation, it is thus difficult to impart an intuitive understanding of the measurement process besides its mathematical description. A measurement always happens with respect to a certain orthonormal basis, for example $(|0\rangle, |1\rangle)$ called the standard basis or $(|0\rangle + |1\rangle, |0\rangle - |1\rangle)$ the diagonal basis. A state measured against such basis *collapses* to either basis state, non-deterministically. After measurement in the standard basis, the measured qubit finds itself in either the $|0\rangle$ or $|1\rangle$ state. The probability for either *measurement outcome* depends on the amplitudes of the quantum state.

The fourth and last postulate presented here deals with the composition of quantum systems. When two quantum systems interact, i.e. they are no longer completely isolated systems, their associated vector state space has to be combined in some way. Given the vectors $|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$ and $|\phi\rangle = \beta_0|0\rangle + \beta_1|1\rangle$ describing the state of two distinct quantum systems, then the state of the combined quantum systems is given by the *tensor product* of both vectors:

$$|\psi\rangle \otimes |\phi\rangle = \alpha_0\beta_0(|0\rangle \otimes |0\rangle) + \alpha_0\beta_1(|0\rangle \otimes |1\rangle) + \alpha_1\beta_0(|1\rangle \otimes |0\rangle) + \alpha_1\beta_1(|1\rangle \otimes |1\rangle) , \quad (\text{A.0.5})$$

for which the notational convention $(|0\rangle \otimes |1\rangle) = |0\rangle|1\rangle = |01\rangle$ is used for compactness, such that

$$|\psi\rangle \otimes |\phi\rangle = \alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle + \alpha_1\beta_1|11\rangle . \quad (\text{A.0.6})$$

The quantum state of two combined quantum systems is said to be *entangled* when this state cannot be represented as a combination of two states of the distinct systems. To illustrate this, take the above two-qubit state example and apply the *controlled-Z* or $\wedge Z$ unitary operator

$$\wedge Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

such that

$$\wedge Z (|\psi\rangle \otimes |\phi\rangle) = \alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle - \alpha_1\beta_1|11\rangle = |\chi\rangle .$$

The resulting state $|\chi\rangle$ can now no longer be described in general as a tensor product of two states, $|\chi\rangle$ is thus said to be *entangled*.

Appendix B

The von Neumann Architecture

It is hard to talk about parallel computing development without giving a brief account of the sequential processor design that has shaped the development of software and hardware over the years. Indeed, the von Neumann Architecture and the associated Random Access Machine (RAM) model of computation has dominated computer science. In the minds of many programmers scientists, the RAM model is how a computer is supposed to work.

The origin of today's most common computer designs date back to a series of papers published by J. von Neumann in 1945 [198], describing the requirements for building a stored-program computer that implements the abstract Turing Machine. This was dubbed the von Neumann architecture (VNA) and describes a machine in distinct subdivisions; using the original terminology these were central control (CC), central arithmetic (CA), memory (M) and Input/Output(I,O). The design sought simplicity and maximal utilization of each individual part; hardware for processing and memory were expensive and thus very precious. Parallel arithmetic operations were deliberately avoided to save on equipment and to simplify planning. The CC had the task to fetch the instruction to be executed and control the operation performed by the CA. There were instructions to load values from M into CA or conversely. A large breakthrough was the following, paraphrased from the original design document [198]:

The orders which are received by CC come from M, i.e. from the same place where the numerical material is stored.

A program was seen as a sequential feed of instructions residing in M, alongside with regular program data, without being able to distinguish one from the other.

Special instructions could change the source in M where CC would fetch its next instruction. The typical organization of a VNA machine is shown in Figure B.1.

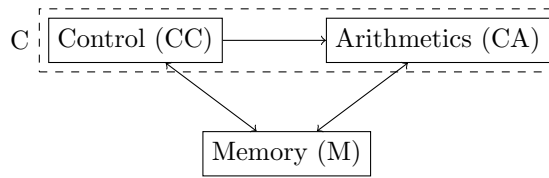


Figure B.1: The von Neumann architecture for a general purpose sequential computer, omitting Input/Output.

The benefit of such a design was a high utilisation of simple parts, efficiently implementing a sequential programmable stored-program computer. Its Achilles heel was already identified by its designer:

[The memory use estimate] shows in a most striking way where the real difficulty, the main bottleneck, of an automatic very high speed computing device lies: At the memory. [...] Clearly the practicality of a device as is contemplated here depends most critically on the possibility of building such an M, and on the question of how simple such an M can be made to be.

The feed between the central unit C and M is dubbed *the von Neumann bottleneck* (VNA) [19]; C can only run as fast as it can be fed from M. Another potential set of problems originates from side effects: instructions writing in already occupied memory locations. Both issues were avoided for a long while; Issues arising from undesirable side effects can be avoided when keeping to sequential execution. Memory speed and size grew tremendously, enjoying continuous technological advances. However, the faster the execution cores became, the more the von Neumann bottleneck became a source of diminishing returns. As parallel execution started being introduced in VNA processors to take up the slack, unwanted side effects have started to become a serious issue [3].

Appendix C

Controlled-Phase gate decomposition

The lemmas from Danos et al. [52], state that any single-qubit unitary operator U can be decomposed into a sequence of J -unitaries:

$$U = e^{i\alpha} J(0) J(\beta) J(\gamma) J(\delta) .$$

The $\wedge U$ or controlled-unitary version of this operator can then be obtained, which can also be decomposed as a sequence of J -unitaries:

$$\begin{aligned} \wedge U_{12} = & J_1(0) J_1\left(\alpha + \frac{\beta + \gamma + \delta}{2}\right) J_2(0) J_2(\beta + \pi) J_2\left(-\frac{\gamma}{2}\right) J_2\left(-\frac{\pi}{2}\right) J_2(0) \wedge Z_{12} \\ & J_2\left(\frac{\pi}{2}\right) J_2\left(\frac{\gamma}{2}\right) J_2\left(\frac{-\pi - \delta - \beta}{2}\right) J_2(0) \wedge Z_{12} J_2(-\beta + \delta - \pi) \end{aligned} \quad (C.0.1)$$

The phase-gate

$$P(\alpha) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix}$$

can be J -decomposed into

$$P(\theta) = J(0) J(\theta)$$

with $\alpha = \beta = \gamma = 0$ and $\delta = \theta$. This yields the following J -decomposition of the controlled-phase gate using Equation (C.0.1):

$$\begin{aligned}
 \wedge P_{12}(\theta) &= J_1(0) J_1\left(\frac{\theta}{2}\right) J_2(0) \underbrace{J_2(\pi) J_2(0) J_2\left(-\frac{\pi}{2}\right) J_2(0)}_{J_2\left(\frac{\pi}{2}\right) J_2(0) J_2\left(\frac{-\pi-\theta}{2}\right) J_2(0) \wedge Z_{12} J_2(-\theta - \pi)} \wedge Z_{12} \\
 &= J_1(0) J_1\left(\frac{\theta}{2}\right) J_2(0) \underbrace{J_2\left(\frac{\pi}{2}\right) J_2(0) J_2\left(\frac{\pi}{2}\right) J_2(0)}_{J_2\left(\frac{\pi}{2}\right) J_2(0) J_2\left(\frac{\pi}{2}\right) J_2(0) \wedge Z_{12} J_2\left(-\frac{\theta}{2}\right) J_2\left(\frac{\theta - \pi}{2}\right)} \wedge Z_{12} J_2\left(-\frac{\theta}{2}\right) J_2\left(\frac{\theta - \pi}{2}\right)
 \end{aligned} \tag{C.0.2}$$

where the underlined expressions are simplified using the

$$J(\alpha)J(0)J(\beta) = J(\alpha + \beta)$$

equality.

Bibliography

- [1] S Abramsky and B Coecke. A categorical semantics of quantum protocols. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 415–425, 2004.
- [2] Shail Aditya, Arvind, Jan-Willem Maessen, Lennart Augustsson, and Rishiyur S Nikhil. Semantics of pH: A parallel dialect of Haskell. In *FPCA 95*, pages 35–49. Computation Structures Group, 1995.
- [3] Sarita Adve and Hans-J Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8), August 2010.
- [4] G A AGHA. *Actors: A model of concurrent computation in distributed systems(Ph. D. Thesis)*. PhD thesis, MIT, 1985.
- [5] P Aliferis and DW Leung. Computation by measurements: a unifying picture. *Physical Review A*, 70(6):062314, 2004.
- [6] Joey Allcock. Emulating Circuit-Based and Measurement-Based Quantum Computation. Master’s thesis, Imperial College London, 2010.
- [7] E Allen, D Chase, J Hallett, and V Luchangco. The Fortress language specification, 2008.
- [8] F Allen, M Burke, P Charles, R Cytron, and J Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, 1988.
- [9] FE Allen and J Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.
- [10] George S Almasi and Allan Gottlieb. *Highly parallel computing*. Benjamin-Cummings Pub Co, 1994.

- [11] S Amarasinghe and M Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures*. PhD thesis, MIT, 2010.
- [12] P R Amestoy, I S Duff, J Y L'Excellent, and J Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
- [13] J.P. Anderson, S.A. Hoffman, J. Shifman, and R.J. Williams. D825-a multiple-computer system for command & control. *Proceedings of the December 4-6, 1962, fall joint computer conference*, pages 86–96, 1962.
- [14] T E Anderson, D E Culler, and D Patterson. A case for NOW (Networks of Workstations). *Ieee Micro*, 15(1):54–64, 1995.
- [15] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
- [16] Arvind and Robert A Iannucci. A critique of multiprocessing von Neumann style. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*. ACM, June 1983.
- [17] K Asanovic, R Bodik, BC Catanzaro, JJ Gebis, P Husbands, K Keutzer, DA Patterson, WL Plishker, J Shalf, and SW Williams. The landscape of parallel computing research: A view from berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.
- [18] K Asanovic, R Bodik, J Demmel, T Keaveny, K Keutzer, J Kubiatowicz, N Morgan, D Patterson, K Sen, and J Wawrzynek. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.
- [19] J Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
- [20] S Barz, E Kashefi, A Broadbent, J F Fitzsimons, A Zeilinger, and P Walther. Demonstration of Blind Quantum Computing. *Science*, 335(6066):303–308, January 2012.
- [21] P Benioff. Quantum mechanical Hamiltonian models of Turing machines. *Journal of Statistical Physics*, 29(3):515–546, 1982.
- [22] C Bennett and G Brassard. Quantum cryptography: Public key distribution and coin tossing. In *IEEE International Conference on Computers, Systems & Signal Processing*, 1984.

-
- [23] I Bethune, J M Bull, N J Dingle, and N J Higham. Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations. Technical report, Manchester Institute for Mathematical Sciences School of Mathematics, 2011.
- [24] S Bettelli, T Calarco, and L Serafini. Toward an architecture for quantum programming. *The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics*, 25(2):181–200, 2003.
- [25] Jacob D Biamonte and Peter J Love. Realizable Hamiltonians for universal adiabatic quantum computers. *Physical Review A*, 78(1):12352, July 2008.
- [26] Lubomir Bic. A process-oriented model for efficient execution of dataflow programs. *Journal of Parallel and Distributed Computing*, 8(1), January 1990.
- [27] K Blathras, D B Szyld, and Y Shi. Parallel Processing of Linear Systems Using Asynchronous Iterative Algorithms. Technical report, Temple University, 1996.
- [28] G E Blelloch. Scans as primitive parallel operations. *Computers, IEEE Transactions on*, 38(11):1526–1538, 1989.
- [29] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
- [30] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Request Permissions, August 1995.
- [31] OpenMP Architecture Review Board. OpenMP, May 2008.
- [32] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.
- [33] H J Briegel, D E Browne, W Dür, R Raussendorf, and M Van den Nest. Measurement-based quantum computation. *Nature Physics*, 5(1):19–26, 2009.
- [34] Joseph Britton, Brian Sawyer, Adam Keith, C-C Wang, James Freericks, Hermann Uys, Michael Biercuk, and John Bollinger. Engineered two-dimensional Ising interactions in a trapped-ion quantum simulator with hundreds of spins. *Nature*, 484(7395):489–492, April 2012.

- [35] A Broadbent and E Kashefi. Parallelizing quantum circuits. *Theoretical Computer Science*, 410(26):2489–2510, 2009.
- [36] Dan Browne, Elham Kashefi, and Simon Perdrix. Computational Depth Complexity of Measurement-Based Quantum Computation. *Theory of Quantum Computation*, 6519:35, 2011.
- [37] DE Browne, E Kashefi, M Mhalla, and S Perdrix. Generalized flow and determinism in measurement-based quantum computation. *New Journal of Physics*, 9:250, 2007.
- [38] D Bruß, G Erdélyi, T Meyer, T Riege, and J Rothe. Quantum cryptography: A survey. *ACM Computing Surveys (CSUR)*, 39(2):6, 2007.
- [39] Z Budimlic, A Chandramowlishwaran, K Knobe, G Lowney, V Sarkar, and L Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC'09: 14th International Workshop on Compilers for Parallel Computers*, 2009.
- [40] Z Budimlić, M Burke, V Cavé, K Knobe, G Lowney, R Newton, J Palsberg, D Peixotto, V Sarkar, and F Schlimbach. Concurrent collections. *Scientific Programming*, 18(3):203–217, 2010.
- [41] A Buttari, J Dongarra, J Kurzak, J Langou, P Luszczek, and S Tomov. The impact of multicore on math software. *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, pages 1–10, 2006.
- [42] D Cann. Retire Fortran? A debate rekindled. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 264–272, 1991.
- [43] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Communications of the ACM*, 51(11), November 2008.
- [44] V Cavé, J Zhao, J Shirako, and V Sarkar. Habanero-Java: the New Adventures of Old X10. *9th International Conference on the Principles and Practice of Programming in Java*, 2011.
- [45] BL Chamberlain, D Callahan, and HP Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007.

-
- [46] A Chandramowlishwaran, K Knobe, and R Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. *IEEE International Symposium on Parallel & Distributed Processing. Proceedings*, pages 1–12, April 2010.
- [47] C Chiw, G Kindlmann, J Reppy, L Samuels, and N Seltzer. Diderot: a parallel DSL for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 111–120. ACM, 2012.
- [48] Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.
- [49] B A Cipra. The Ising model is NP-complete. *SIAM News*, 2000.
- [50] EF Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [51] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [52] V Danos, E Kashefi, and P Panangaden. Parsimonious and robust realizations of unitary maps in the one-way model. *Physical Review A*, 72(6):064301, 2005.
- [53] V Danos, E Kashefi, and P Panangaden. The Measurement Calculus. *Journal of the ACM (JACM)*, 54(2):8, 2007.
- [54] Arnab Das and Bikas K Chakrabarti. Colloquium: Quantum annealing and analog quantum computation. *Reviews of Modern Physics*, 80(3):1061–1081, July 2008.
- [55] N de Beaudrap. Finding flows in the one-way measurement model. *Physical Review A*, 77(2):22328, 2008.
- [56] K de Raedt, K Michielsen, H de Raedt, B Trieu, G Arnold, M Richter, Th Lippert, H Watanabe, and N Ito. Massively parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, January 2007.
- [57] J Dean and S Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

- [58] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [59] D. Dechev and B. Stroustrup. Scalable nonblocking concurrent objects for mission critical code. *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 597–612, 2009.
- [60] PJ Denning and JB Dennis. The resurgence of parallelism. *Communications of the ACM*, 53(6):30–32, 2010.
- [61] J Dennis. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News*, 1974.
- [62] J Dennis. First version of a data flow procedure language. *Programming Symposium*, pages 362–376, 1974.
- [63] JB Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.
- [64] B Desmet, E D’Hondt, P Costanza, and T D’Hondt. Simulation of quantum computations in Lisp. In *3rd European Lisp Workshop, co-located with ECOOP*, 2006.
- [65] D Deutsch and R Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.
- [66] E D’Hondt. *Distributed quantum computation: a measurement-based approach*. PhD thesis, Vrije Universiteit Brussel, 2005.
- [67] Ellie D’Hondt and Yves Vandriessche. Distributed quantum programming. *Natural Computing*, 10(4):1313–1343, December 2011.
- [68] P Diaconis. Mathematics and magic tricks. *Clay Mathematics Institute*, 2006.
- [69] P Diaconis, R Graham, and W Kantor. The mathematics of perfect shuffles. *Advances in Applied Mathematics*, 1983.
- [70] Ross Duncan. A graphical approach to measurement-based quantum computing. *Compositional methods in Physics and Linguistics*, quant-ph, 2012.
- [71] W Dür, G Vidal, and J I Cirac. Three qubits can be entangled in two inequivalent ways. *Physical Review A (Atomic)*, 62(6):62314, December 2000.

-
- [72] R K Dybvig. *Three implementation models for scheme*. PhD thesis, University of North Carolina at Chapel Hill, June 1987.
- [73] A Ekert. Quantum cryptography based on Bell's theorem. *Physical Review Letters*, 1991.
- [74] D Eppstein and Z Galil. Parallel Algorithmic Techniques For Combinational Computation. *Annual Review of Computer Science*, 3(1):233–283, June 1988.
- [75] H Esmailzadeh, E Blem, RS Amant, K Sankaralingam, and D Burger. Dark silicon and the end of multicore scaling. *Proceeding of the 38th annual international symposium on Computer architecture*, pages 365–376, 2011.
- [76] RP Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6):467–488, 1982.
- [77] RP Feynman. Quantum mechanical computers. *Foundations of physics*, 16(6):507–531, 1986.
- [78] M.J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.
- [79] Message Passing Interface Forum. MPI: a message passing interface standard, 2009.
- [80] MP Frank, UH Meyer-Baese, I Chiorescu, L Oniciuc, and RA van Engelen. Space-efficient simulation of quantum computers. *Proceedings of the 47th Annual Southeast Regional Conference*, pages 1–6, 2009.
- [81] MP Frank, L Oniciuc, U Meyer-Baese, and I Chiorescu. A space-efficient quantum computer simulator suitable for high-speed FPGA implementation. *Imprint*, 9:12, 2009.
- [82] B R Gaster and L Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 2012.
- [83] JL Gaudiot, T DeBoni, J Feo, W Böhm, W Najjar, and P Miller. The Sisal model of functional programming and its implementation. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs '97)*, page 112, 1997.
- [84] JL Gaudiot, T DeBoni, J Feo, W Böhm, W Najjar, and P Miller. The Sisal project: real world functional programming. *Compiler optimizations for scalable parallel systems*, pages 45–72, 2001.

- [85] SJ Gay. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science*, 16(04):581–600, 2006.
- [86] SJ Gay and R Nagarajan. Types and typechecking for communicating quantum processes. *Mathematical Structures in Computer Science*, 16(03):375–406, 2006.
- [87] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM Request Permissions, October 2007.
- [88] D Ghosal and LN Bhuyan. Performance evaluation of a dataflow architecture. *Computers, IEEE Transactions on*, 39(5):615–627, 1990.
- [89] Jean-Yves Girard. Between logic and quantic: a tract. In *Linear logic in computer science*, pages 346–381. Cambridge Univ. Press, Cambridge, 2004.
- [90] I Glendinning and B Ömer. Parallelization of the qc-lib quantum computer simulator library. *Parallel Processing and Applied Mathematics*, pages 461–468, 2004.
- [91] M I Gordon, W Thies, M Karczmarek, J Lin, A S Meli, A A Lamb, C Leger, J Wong, H Hoffmann, and D Maze. A stream compiler for communication-exposed architectures. *ACM SIGPLAN Notices*, 37(10):291–303, 2002.
- [92] Daniel Gottesman. The Heisenberg Representation of Quantum Computers. *Audio, Transactions of the IRE Professional Group on*, pages –, June 1998.
- [93] Ananth Grama. *Introduction to parallel computing*. Addison Wesley, 2003.
- [94] J Grattage. *QML: A functional quantum programming language*. PhD thesis, Ph. D. thesis, The University of Nottingham, 2006.
- [95] Steve Gregory. *Parallel logic programming in PARLOG*. the language and its implementation. Addison-Wesley, 1987.
- [96] LK Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.
- [97] Josef Gruska. *Quantum Computing*. Mcgraw Hill Book Co Ltd, April 2000.

-
- [98] JR Gurd, CC Kirkham, and I Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.
- [99] E Gutierrez, S Romero, MA Trenas, and EL Zapata. Parallel Quantum Computer Simulation on the CUDA Architecture. *Lecture Notes in Computer Science*, 5101:700–709, 2008.
- [100] M. Hall, D Padua, and K. Pingali. Compiler research: the next 50 years. *Communications of the ACM*, 52(2):60–67, 2009.
- [101] D Hanneke, J P Home, J D Jost, J M Amini, D Leibfried, and D J Wine-land. Realization of a programmable two-qubit quantum processor. *Nature Physics*, 6(1):13–16, January 2010.
- [102] Tim Harris, Keir Fraser, Tim Harris, and Keir Fraser. *Language support for lightweight transactions*, volume 38. ACM, November 2003.
- [103] Matin Hashemi, Mohammad H Foroozannejad, Soheil Ghiasi, and Christoph Etzel. FORMLESS: scalable utilization of embedded manycores in streaming applications. In *LCTES '12: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM Request Permissions, June 2012.
- [104] M Hein, J Eisert, and H J Briegel. Multiparty entanglement in graph states. *Physical Review A*, 69(6):62311, June 2004.
- [105] DP Helmbold and CE McDowell. Modeling speedup (n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):250–256, 1990.
- [106] Harold Henderson and S Searle. The vec-permutation matrix, the vec operator and Kronecker products: a review. *Linear and Multilinear Algebra*, 9(4):271–288, 1981.
- [107] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach; 5th ed.* Elsevier Science, Burlington, 2011.
- [108] Carl Hewitt. Middle History of Logic Programming: Resolution, Planner, Edinburgh LCF, Prolog, Simula, and the Japanese Fifth Generation Project. *arXiv.org*, cs.LO, April 2009.
- [109] W Daniel Hillis. *The Connection Machine*. The MIT Press, February 1989.
- [110] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

-
- [111] Wen-mei Hwu. Top Five Reasons. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–31. University of Illinois, Urbana-Champaign, December 2006.
- [112] Mlnark Hynek. *Quantum Programming Language LanQ*. PhD thesis, Masaryk University, Faculty of Informatics, September 2007.
- [113] M Isard, M Budiu, Y Yu, A Birrell, and D Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):72, 2007.
- [114] ITRS. International Technology Roadmap for Semiconductors. Technical report, January 2012.
- [115] JR Johnson, RW Johnson, D Rodriguez, and R Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9(4): 449–500, 1990.
- [116] M Johnson, M Amin, S Gildert, and T Lanting. Quantum annealing with manufactured spins. *Nature*, 2011.
- [117] RW Johnson, CH Huang, and JR Johnson. Multilinear algebra and parallel programming. *The Journal of Supercomputing*, 5(2):189–217, 1991.
- [118] WM Johnston, JRP Hanna, and RJ Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.
- [119] WM Johnston, JRP Hanna, and RJ Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.
- [120] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Request Permissions, January 1996.
- [121] SLP Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(02): 127–202, 1992.
- [122] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information processing 74 (Proc. IFIP Congress, Stockholm, 1974)*, pages 471–475. North-Holland, Amsterdam, 1974.

-
- [123] RM Karp and RE Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.
- [124] RM Karp and RE Miller. Parallel program schemata: A mathematical model for parallel computation. *IEEE Conference Record of the Eighth Annual Symposium on Switching and Automata Theory*, pages 55–61, 1967.
- [125] KM Kavi, BP Buckles, and UN Bhat. A formal definition of data flow graph models. *Computers, IEEE Transactions on*, 100(11):940–948, 1986.
- [126] RM Keller. Parallel program schemata and maximal parallelism I. Fundamental results. *Journal of the ACM (JACM)*, 20(3):514–537, 1973.
- [127] E H Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.
- [128] K Knobe. Ease of use with concurrent collections (CnC). *Hot Topics in Parallelism*, 2009.
- [129] K Knobe and CD Offner. TStreams: How to Write a Parallel Program. Technical report, HP Laboratories Cambridge, 2004.
- [130] B Kumar, C.-H Huang, R.W Johnson, and P Sadayappan. A tensor product formulation of Strassen’s matrix multiplication algorithm with memory reduction. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 582–588, 1993.
- [131] Ronald L Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley, 1994.
- [132] Richard E Ladner and Michael J Fischer. Parallel Prefix Computation. *Journal of the ACM (JACM)*, 27(4), October 1980.
- [133] B Lee and A Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, 1994.
- [134] E A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- [135] EA Lee and DG Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, 100(1):24–35, 1987.
- [136] J Magnus and H Neudecker. The commutation matrix: Some properties and applications. *The Annals of Statistics*, 1979.

- [137] G A Mago. A network of microprocessors to execute reduction languages, Part I. *International Journal of Parallel Programming*, 8(5):349–385, 1979.
- [138] Milo M K Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7), July 2012.
- [139] T Mattson and M Wrinn. Parallel programming: can we PLEASE get it right this time? *Proceedings of the 45th annual Design Automation Conference*, pages 7–11, 2008.
- [140] J McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
- [141] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *PPoPP '12: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM Request Permissions, February 2012.
- [142] Peter A Milder. *A mathematical approach for compiling and optimizing hardware implementation of DSP transforms*. ProQuest, UMI Dissertation Publishing, September 2011.
- [143] Josh Milthorpe, V Ganesh, Alistair P Rendell, and David Grove. X10 as a Parallel Language for Scientific Computation: Practice and Experience. *IPDPS*, pages 1080–1088, 2011.
- [144] A Mycroft. Programming language design and analysis motivated by hardware evolution. *Static Analysis*, pages 18–33, 2007.
- [145] R Nagarajan, N Papanikolaou, and D Williams. Simulating and Compiling Code for the Sequential Quantum Random Access Machine. *Electronic Notes in Theoretical Computer Science*, 170:101–124, 2007.
- [146] Ashwin Nayak and Ashvin Vishwanath. Quantum Walk on the Line. *arXiv.org*, October 2000.
- [147] H Neven, V S Denchev, M Drew-Brook, J Zhang, W G Macready, and G Rose. NIPS 2009 demonstration: Binary classification using hardware implementation of quantum annealing. *Quantum*, pages 1–17, 2009.
- [148] C J Newburn, Byoungro So, Zhenying Liu, M McCool, A Ghuloum, S D Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel’s Array Building Blocks: A re-targetable, dynamic compiler and embedded language. *Audio, Transactions of the IRE Professional Group on*, pages 224–235, March 2011.

-
- [149] J Nickolls and W J Dally. The GPU Computing Era. *Micro, IEEE*, 30(2): 56–69, 2010.
- [150] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 10 anv edition, January 2011.
- [151] Kevin M Obenland and Alvin M Despain. A Parallel Quantum Computer Simulator. *arXiv.org*, quant-ph:4039, April 1998.
- [152] Bernhard Oemer. *Structured quantum programming*. PhD thesis, TU Vienna, 2003.
- [153] A Olofsson and R Trogan. A 25 GFLOPS/Watt Software Programmable Floating Point Accelerator. Technical report, Adapteva Inc., 2010.
- [154] Marshall Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the ACM (JACM)*, 15(2), April 1968.
- [155] Alejandro Perdomo-Ortiz, Neil Dickson, Marshall Drew-Brook, Geordie Rose, and Alán Aspuru-Guzik. Finding low-energy conformations of lattice protein models by quantum annealing. *Scientific Reports*, 2, August 2012.
- [156] C Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.
- [157] M Puschel, JMF Moura, JR Johnson, D Padua, MM Veloso, BW Singer, J Xiong, F Franchetti, A Gacic, and Y Voronenko. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- [158] M Püschel, P Milder, and J Hoe. Permuting streaming data using RAMs. *Journal of the ACM (JACM)*, 2009.
- [159] R Raussendorf and HJ Briegel. A one-way quantum computer. *Physical Review Letters*, 86(22):5188–5191, 2001.
- [160] R Raussendorf, J Harrington, and K Goyal. A fault-tolerant one-way quantum computer. *Annals of Physics*, 321(9):2242–2270, 2006.
- [161] Robert Raussendorf and Hans J Briegel. Quantum computing via measurements only. *arXiv.org*, page 10033, October 2000.
- [162] J W Sanders and P Zuliani. Quantum Programming. In *MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction*. Springer-Verlag, July 2000.

- [163] V Sarkar and D Cann. POSC—a partitioning and optimizing SISAL compiler. *ACM SIGARCH Computer Architecture News*, 18(3b):148–164, 1990.
- [164] R R Seeber and A B Lindquist. Associative logic for highly parallel systems. In *AFIPS '63 (Fall): Proceedings of the November 12-14, 1963, fall joint computer conference*. ACM, November 1963.
- [165] L Seiler, D Carmean, E Sprangle, T Forsyth, M Abrash, P Dubey, S Junkins, A Lake, J Sugerman, and R Cavin. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 27(3):18, 2008.
- [166] P Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(04):527–586, 2004.
- [167] Peter Selinger and Beno i t Valiron. Quantum lambda calculus. In *Semantic techniques in quantum computation*, pages 135–172. Cambridge Univ. Press, Cambridge, 2010.
- [168] M Sharifzadeh and C Shahabi. The spatial skyline queries. *Proceedings of the 32nd international conference on Very large data bases*, pages 751–762, 2006.
- [169] John A Sharp. *Data flow computing: theory and practice*. Ablex Publishing Corp., Norwood, NJ, USA, 1992.
- [170] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi. Evaluation of a prototype data flow processor of the sigma-1 for scientific computations. *ACM SIGARCH Computer Architecture News*, 14(2):226–234, 1986.
- [171] Peter W Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5): 1484–1509, 1997.
- [172] PW Shor. Algorithms for quantum computation: discrete logarithms and factoring. *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134, 1994.
- [173] S Simmons, R Brown, H Riemann, and N Abrosimov. Entanglement in a solid-state spin ensemble. *Nature*, 2011.
- [174] JH Spring, J Privat, R Guerraoui, and J Vitek. Streamflex: high-throughput stream programming in java. *ACM SIGPLAN Notices*, 42(10): 228, 2007.

-
- [175] GL Steele. Growing a language. In *Higher-Order and Symbolic Computation*, pages 221–236, 1999.
- [176] R Stephens. A survey of stream processing. *Acta Informatica*, 1997.
- [177] HS Stone. Parallel processing with the perfect shuffle. *Computers, IEEE Transactions on*, 100(2):153–161, 1971.
- [178] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3):202–210, 2005.
- [179] Herb Sutter. Welcome to the Jungle, January 2012. URL <http://herbsutter.com/welcome-to-the-jungle/>.
- [180] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7), September 2005.
- [181] KM Svore, AV Aho, AW Cross, I Chuang, and IL Markov. A layered software architecture for quantum computing design tools. *Computer*, pages 74–83, 2006.
- [182] F Tabakin and B Juliá-Díaz. QCMPI: A parallel environment for quantum computing. *Computer Physics Communications*, 180(6):948–964, 2009.
- [183] M S Tame, R Prevedel, M Paternostro, P Böhi, M S Kim, and A Zeilinger. Experimental realization of Deutsch's algorithm in a one-way quantum computer. *Physical Review Letters*, 98(14):140501, 2007.
- [184] D Tarditi, S Puri, and J Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, 2006.
- [185] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrati, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpfen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *Ieee Micro*, 22(2), March 2002.
- [186] W Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, MIT, 2009.
- [187] William Thies, Michal Karczmarek, and Saman P Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, April 2002.

-
- [188] J Torrellas, HS Lam, and JL Hennessy. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on*, 43(6):651–663, 1994.
- [189] PC Treleaven, DR Brownbridge, and RP Hopkins. Data-driven and demand-driven computer architecture. *ACM Computing Surveys (CSUR)*, 14(1):93–143, 1982.
- [190] CY Tsai, MH Fan, and CH Huang. VLSI circuit design of matrix transposition using tensor product formulation. *Proceedings of the International Conference on Informatics, Cybernetics, and Systems*, 2003.
- [191] LG Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
- [192] G Vallone, G Donati, N Bruno, A Chiuri, and P Mataloni. Experimental realization of the Deutsch-Jozsa algorithm with a six-qubit cluster state. *Physical Review A*, 81(5):050302, 2010.
- [193] CF Van Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1-2):85–100, 2000.
- [194] Lieven M K Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S Yannoni, Mark H Sherwood, and Isaac L Chuang. Experimental realization of Shor’s quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6):883–887, December 2001.
- [195] AH Veen. Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 18(4):365–396, 1986.
- [196] George F Viamontes, Igor L markov, and John P hayes. Graph-based simulation of quantum computation in the density matrix representation. *Quantum Information and Computation II. Edited by Donkor*, 5436:285–296, August 2004.
- [197] George F Viamontes, Igor L markov, and John P hayes. *Quantum Circuit Simulation*. Springer, 2009.
- [198] J Von Neumann and MD Godfrey. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.
- [199] P Walther, K J Resch, T Rudolph, E Schenck, H Weinfurter, V Vedral, M Aspelmeyer, and A Zeilinger. Experimental one-way quantum computing. *Nature*, 434(7030):169–176, March 2005.

-
- [200] D Warren. An abstract Prolog instruction set. Technical report, Digital Equipment Corporation, 1983.
- [201] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *Ieee Micro*, 27(5), September 2007.
- [202] C. Whitby-Strevens. The transputer. *ACM SIGARCH Computer Architecture News*, 13(3):292–300, 1985.
- [203] WA Wulf and SA McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–20, 1995.
- [204] Y Yamaguchi, K Toda, and T Yuba. A performance evaluation of a Lisp-based data-driven machine (EM-3). *ACM SIGARCH Computer Architecture News*, 11(3):363–369, 1983.
- [205] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM Request Permissions, July 2007.