Vrije Universiteit Brussel

Faculty of Sciences
Department of Computer Science
and Applied Computer Science

Multilinear Algebra for Multicore Architectures a Case Study in Parallel Quantum Virtualization

Bachelor Paper

Klaas Moerman

Advisor: dr. Yves Vandriessche

June 2013

Contents

Chapter 1

Introduction

This document reports on an educational research project, required by the *Vrije Universiteit Brussel* to obtain a Bachelor's Degree in Computer Science.

As an introductory chapter, we present a brief overview of the *quantum computing* and *multicore programming* domains, establishing the context of our work. Subsequently, we express the intent of our project and hint at the general approach taken.

1.1 Context

Quantum Computing Since its appearance [17] and development [8,18] in physics, quantum computing has grown into a research topic in computer science and engineering. However, we can still compare the current state of this field to the pre-electronic era of classical computing; a theoretical background for quantum computers and their physical principles exists, whereas a *quantum programming* paradigm has yet to emerge. This motivates research into building a quantum programming framework, which, in the absence of actual quantum computers, entails the need for simulating them.

Several competing models for quantum computing exist, often representing radically different ways of achieving quantum computations. The low-level *circuit model*, commonly used to express quantum computation [24,30], lacks conceptual abstraction and requires intimate knowledge of quantum mechanics. Conversely, more formal approaches at a higher level [22,33] lack a low-level execution model.

A more recent model, *measurement calculus* (MC) [15], combines the desirable properties of high-level expressiveness and formal low-level operational semantics, meeting the requirements [11] for a practical quantum architecture [42]. The model provides a modular and compositional abstraction for quantum programs based on a small set of primitives, it formalizes a physically realistic model for performing measurement based quantum computations and, finally, it separates the quantum and classical state of a program, yielding the realistic vision of a quantum computer as a coprocessor to a traditional CPU, similar to GPUs nowa-

days. From a computer scientists' perspective, the MC can serve as an assembly language for quantum computers.

Multicore programming As hardware engineers reach the performance limits of single core processors, multicore processors have become a mainstream commodity. This trend has considerable implications for the software community; hardware improvements no longer induce a performance gain in traditionally sequential programs, requiring a shift towards parallel programs [38]. Moreover, the traditional concurrency primitives (coarse-grained threads) used in parallel programming show scaling issues, both from a software engineering and performance perspective, as threads prove notoriously hard to deal with (in terms of understandability, predictability and determinism) and don't yield the desired speedup due to load imbalance, synchronization, and communication [27]. Consequently, the field of parallel computing has regained its status as a research topic [10].

Additionally, as microprocessor speed exceeds DRAM memory speed [44], memory traffic strongly constrains application performance [31]. The addition of multiple cache levels mitigates this problem, which occurs outside of the multicore realm as well. However, this *memory wall* effectively prevents the scalability of multicore architectures, inciting research into new architectural solutions [37].

This situation, commonly dubbed *the end of the free lunch* [38], will require new programming paradigms and languages, with fine-grained parallelism and higher flexibility [39]. In addition to classic threading libraries like `Pthreads` and `OpenMP`, solutions that existed in the *high-performance computing* domain before the current multicore period, for example the `Cilk` [1] or `NESL` [3] languages, now become mainstream in the form of vendor libraries or frameworks such as `Intel Cilk++`, `Intel Array Building Blocks` or `Intel Threading Building Blocks`.

1.2 Problem statement

Vandriessche [42] presented a framework for quantum programming in a layered architecture for composing and executing quantum programs. Ultimately, this framework compiles a quantum program down to a series of assembly language instructions based on the MC. In turn, a *quantum virtual machine* (QVM) interprets and executes these instructions. The QVM currently implements sequential interpreter with `libquantum` [13] as a back-end, however, for research purposes in quantum programming, this sequential interpreter performs poorly. The resurgence of parallel computing and the commercial availability of multicore hardware allow us to build a parallel implementation, bringing a better performing QVM to off-the-shelf desktop computers. However, the available development and analysis tools can only deal with the complexity of parallel programming to a limited extent, and often induce runtime overhead cancelling the parallel benefit. Thus, when aiming at *optimal performance*, meaning efficient usage of the

available computing resources, including memory, bandwidth, caches, processing time and active cores, the transformation form sequential to parallel turns out non-trivial.

1.3 Approach

Each quantum operation in the MC corresponds to a large matrix operation. By carefully exploring the mathematical properties of its linear algebra formulation, in particular the *tensor product*, we will obtain vector operations, reducing the workload and better matching the constraints of a multicore platform. As an implementation basis, we use an openly available parallel runtime library (Intel TBB), allowing relatively straightforward parallel code. We will validate our design decisions both theoretically and experimentally, primarily aiming at a scalable implementation of a *parallel quantum virtual machine* (PQVM).

Chapter 2

Quantum Operators in the Measurement Calculus

Before presenting the parallel adaptation of the QVM, we must first understand the mathematical properties of its underlying operations. In the ensuing chapter, we assume a basic understanding of linear algebra, specifically *linear operators* and *vector spaces*. We will first introduce the foundations of quantum computing, based on the formulation in [24]. In the subsequent sections we describe the actual MC primitives, constructing a formulation that can serve as an implementation basis.

2.1 Quantum Computing

Newton's laws, or equivalently, the formalisms of Lagrange or Hamilton, form an axiomatic foundation for the class of physical theories known as *classical mechanics*. Continuum mechanics, for instance, extends the basic mechanical axioms with the assumption of continuous matter, modelling the behavior of solids and fluids under stress.

Similarly, theories of *quantum mechanics* share an abstract set of postulates, first formulated in their current form by Dirac and Von Neumann. As computer scientists, we can interpret these postulates as a theoretical *framework* from which we can derive a quantum theory of *computation*, just as the Church-Turing thesis provides a theoretical basis for classical computation. Indeed, Deutsch [8] formulated a physical alternative of the Church-Turing thesis, thus suggesting a quantum mechanical foundation for computation.

We will now state the quantum mechanical postulates and construct the basic abstractions for our quantum computational model. The inquisitive reader will find a comprehensive survey of quantum computing and its history in the listed references [24, 30].

Postulate 1 *State space.* A unit vector in a complex Hilbert $\mathcal{H}$ space completely defines the state of an isolated quantum system. We call this vector the *state vector*,

and $\mathcal{H}$ its associated *state space.*

As the basic building block of quantum information, analogous to its classical counterpart, the *bit*, we define a *qubit* $|\psi\rangle$ as a vector in the two-dimensional complex vector space $\mathbb{C}^2$. For an appropriate basis $\{|0\rangle, |1\rangle\}$, we have

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

Specifically, we refer to the orthonormal basis

$$\mathsf{B} = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \tag{2.2}$$

as the *computational basis* and write

$$|\mathtt{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad |\mathtt{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

The unit condition of a state vector requires

$$\langle\psi|\psi\rangle = 1$$

from which we derive the normalization constraint on α and β

$$|\alpha|^2 + |\beta|^2 = 1$$

Postulate 2 *Time evolution.* Any time evolution of a closed quantum system corresponds to the application of a unitary transformation U to that system. Thus, for an initial state $|\psi\rangle$ and an evolved state $|\varphi\rangle$, we have

$$|\varphi\rangle = U|\psi\rangle$$

Lemma 2.1 *All and only the unitary operators preserve the inner product.*

$$\langle U\psi, U\varphi\rangle = \langle\psi, \varphi\rangle \iff UU^\dagger = I$$

Proof A direct proof based on the associativity of linear operators.

$$\begin{aligned} \langle U\psi, U\varphi\rangle &= \left(\langle\psi|U^\dagger\right) \cdot \left(U|\varphi\rangle\right) \\ &= \langle\psi|U^\dagger U|\varphi\rangle \\ &= \langle\psi|I|\varphi\rangle \\ &= \langle\psi||\varphi\rangle \\ &= \langle\psi, \varphi\rangle \end{aligned}$$

Lemma 2.1 establishes the importance of unitary operators, as the only norm-preserving operators. Using the computational basis (2.2), we can express a qubit

operator[1] U as a $\mathbb{C}^{2\times 2}$ matrix. For example, the X operator, the quantum analogy of the classical NOT bit-operation, acts on the basis vectors as

$$X : \mathbb{C}^2 \to \mathbb{C}^2 : \begin{cases} |\mathtt{0}\rangle \mapsto |\mathtt{1}\rangle \\ |\mathtt{1}\rangle \mapsto |\mathtt{0}\rangle \end{cases}$$

and has the matrix representation

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Recall from linear algebra that the action on the basis vectors completely specifies a linear operator; X therefore maps a qubit $|\psi\rangle = \alpha|\mathtt{0}\rangle + \beta|\mathtt{1}\rangle$ to

$$X|\psi\rangle = \alpha|\mathtt{1}\rangle + \beta|\mathtt{0}\rangle$$

Postulate 3 *Composition of systems.* The state vectors $|\psi_1\rangle$ and $|\psi_2\rangle$ of two systems in their respective state spaces $\mathcal{H}_1$ and $\mathcal{H}_2$ determine the state vector $|\psi_1\rangle \otimes |\psi_2\rangle$ of the combining system in state space $\mathcal{H}_1 \otimes \mathcal{H}_2$.

Inductively applying this postulate shows that the composition of n subsystems in state $|\psi_i\rangle \in \mathcal{H}_i$ will result in the state vector $\bigotimes_i |\psi_i\rangle \in \bigotimes_i \mathcal{H}_i$, where $i \in \{1, 2, \dots, n\}$.

Continuing the analogy with classical bits, we call the composition of n qubits $|\psi_i\rangle$ a *quregister*, with state vector

$$|\psi\rangle = \bigotimes_i |\psi_i\rangle \tag{2.3}$$

in state space

$$\mathcal{H} = \left(\mathbb{C}^2\right)^{\otimes n} = \mathbb{C}^{2^n}$$

When manipulating a quregister, we cannot always decompose it back into the tensor product of its constituent qubits (in the form of equation (2.3)). We call a quregister *separable* if we can express it as a tensor product of individual qubits, conversely, we call a quregister *entangled*, when we cannot decompose it into a tensor product. In general, the set of 2^n-dimensional state vectors formed by taking the tensor product of two 2^{n-k}-dimensional state vectors contains only a sparse subset of all the possible 2^n-dimensional state vectors [24].

Example 2.1 No combination of coefficients α_0, α_1 and β_0, β_1 exists such that the 2-qubit state $|\psi\rangle = \frac{1}{\sqrt{2}}|\mathtt{00}\rangle + \frac{1}{\sqrt{2}}|\mathtt{11}\rangle$ equals the tensor decomposition $(\alpha_0|\mathtt{0}\rangle + \alpha_1|\mathtt{1}\rangle) \otimes (\beta_0|\mathtt{0}\rangle + \beta_1|\mathtt{1}\rangle)$.

[1] The quantum computing literature often refers to qubit operators as *gates*, in the context of *quantum circuits*. In this report, adhering to the measurement calculus' standards, we avoid such terminology.

Proof *By contradiction.* Assume the existence of such coefficients α_0, α_1, β_0, β_1 and consider the resulting vectors

$$|\psi\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix} \quad \text{and} \quad \begin{pmatrix}\alpha_0\beta_0\\\alpha_0\beta_1\\\alpha_1\beta_0\\\alpha_1\beta_1\end{pmatrix}$$

By the first and the last rows, we must refute that any of the coefficients equal zero. However, by the second and third rows, we must conclude that at least two of the coefficients equal zero. This contradiction implies that no such coefficients can exist.

Postulate 4 *Measurement.* For a given orthonormal basis $\{|\varphi_i\rangle\}$ of a state space $\mathcal{H}$, one can perform a *Von Neumann measurement* of a state $|\psi\rangle = \sum_i \alpha_i|\varphi_i\rangle$, which outputs a label i with probability α_i^2 and leaves the system in state $|\varphi_i\rangle$.

Note that for the state $|\psi\rangle$ and basis vector $|\varphi_i\rangle$, the amplitude α_i equals

$$\alpha_i = \langle\varphi_i|\psi\rangle \tag{2.4}$$

2.2 Computational Basis

At its core, a QVM requires implementing unitary operators on vectors in a complex Hilbert space. To perform actual calculations, we need to choose a basis for this vector space. This basis, called the *computational basis*, will remain implicit throughout this report.

Definition 2.1 For the complex vector space $\mathbb{C}^2$, we define the computational basis $\mathtt{B}$ as the set of colums of I, the unit matrix.

$$\mathtt{B} = \left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\} \tag{2.5}$$

We use the compact Dirac notation

$$|\mathtt{0}\rangle = \begin{pmatrix}1\\0\end{pmatrix} \quad \text{and} \quad |\mathtt{1}\rangle = \begin{pmatrix}0\\1\end{pmatrix} \tag{2.6}$$

We can express a qubit $|\psi\rangle \in \mathbb{C}^2$ as a linear combination of these basis vectors

$$|\psi\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix} = \alpha|\mathtt{0}\rangle + \beta|\mathtt{1}\rangle$$

Definition 2.2 *Computational basis.* We define the computational basis $\mathtt{B}_n = \mathtt{B}^{\otimes n}$ spanning the complex vector space $\mathcal{H} = \mathbb{C}^{2^n}$ as the set of columns of the unit

matrix $I^{\otimes n}$

$$\mathsf{B}_n = \mathsf{B}^{\otimes n} = \{|\mathtt{0}\rangle, |\mathtt{1}\rangle\}^{\otimes n} = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}, \ \cdots \ , \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \right\} \tag{2.7}$$

We make extensive use of Iverson's bracket notation [25], mapping a true statement to 1 and a false statement to 0. Combining Dirac notation and Iverson brackets, we obtain a more compact representation for a basis vector

$$|i\rangle = \begin{pmatrix} [i = 0] \\ \vdots \\ [i = 2^n - 1] \end{pmatrix} \qquad 0 \leqslant i < 2^n$$

With respect to this basis, we can express the n-qubit quregister $|\psi\rangle$ as the linear combination

$$|\psi\rangle = \sum\nolimits_i \alpha_i |i\rangle \tag{2.8}$$

We call α_i the *probability amplitude* of $|i\rangle$ in $|\psi\rangle$.

Example 2.2 To show the relation between column vector (unary) and Dirac notation (binary or decimal), let us consider the state space of 2-qubit quregisters, which has four basis vectors

$$|0\rangle = |\mathtt{0}\rangle \otimes |\mathtt{0}\rangle = |\mathtt{00}\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad |2\rangle = |\mathtt{1}\rangle \otimes |\mathtt{0}\rangle = |\mathtt{10}\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$|1\rangle = |\mathtt{0}\rangle \otimes |\mathtt{1}\rangle = |\mathtt{01}\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad |3\rangle = |\mathtt{1}\rangle \otimes |\mathtt{1}\rangle = |\mathtt{11}\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The `binary` notation has our preference, as the number of qubits in the quregister corresponds to the number of digits in the ket; the decimal notation lacks this property. Given two qubits (using subscript to identify the qubit and the superscript to identify the basis vector)

$$|\psi_1\rangle = \psi_1^{\mathtt{0}} |\mathtt{0}\rangle + \psi_1^{\mathtt{1}} |\mathtt{1}\rangle \quad \text{and} \quad |\psi_2\rangle = \psi_2^{\mathtt{0}} |\mathtt{0}\rangle + \psi_2^{\mathtt{1}} |\mathtt{1}\rangle$$

the combining quregister $|\psi\rangle = |\psi\rangle_1 \otimes |\psi\rangle_2$ becomes

$$|\psi\rangle = \psi_1^{\mathtt{0}} \psi_2^{\mathtt{0}} |\mathtt{00}\rangle + \psi_1^{\mathtt{0}} \psi_2^{\mathtt{1}} |\mathtt{01}\rangle + \psi_1^{\mathtt{1}} \psi_2^{\mathtt{0}} |\mathtt{10}\rangle + \psi_1^{\mathtt{1}} \psi_2^{\mathtt{1}} |\mathtt{11}\rangle$$

The example above reveals some inherently binary properties of the computational basis and the tensor product. While familiar to the seasoned computer

scientist, we will develop these properties independently of integer representation and bitwise operations, maintaining mathematical rigor and providing a concise notation connecting the linear algebra formulation and the actual QVM implementation.

Definition 2.3 *Generalized parity.* Consider a non-negative integer and its canonical binary representation[2]. We call this number k-odd when the bit at position k equals `1` and k-even when the bit equals `0`. The function $\mathcal{P}_k$ maps k-odd numbers to 1 and k-even numbers to 0.

$$\begin{array}{rccc} \mathcal{P}_k: & \mathbb{N} & \to & \mathbb{Z}_2 \\ & n & \mapsto & n\backslash 2^k \bmod 2 \end{array} \tag{2.9}$$

where $a\backslash b = \left\lfloor \frac{a}{b} \right\rfloor$ denotes integer division. The reader may recognize the composition of a bit shift and a bit test operation. The base case $\mathcal{P}_0$ degenerates into a simple parity test, mapping odd numbers to 1 and even numbers to 0.

Example 2.3 Calculate $\mathcal{P}_k(13)$ for $k \in \{0, 1, 2, 3\}$

$$\begin{aligned} \mathcal{P}_0(13) &= 13\backslash 2^0 \bmod 2 \\ &= 1 \end{aligned} \qquad \begin{aligned} \mathcal{P}_2(13) &= 13\backslash 2^2 \bmod 2 \\ &= 1 \end{aligned}$$

$$\begin{aligned} \mathcal{P}_1(13) &= 13\backslash 2^1 \bmod 2 \\ &= 0 \end{aligned} \qquad \begin{aligned} \mathcal{P}_3(13) &= 13\backslash 2^3 \bmod 2 \\ &= 1 \end{aligned}$$

This corresponds to the binary representation $13 =$ `1101`. Note that $\mathcal{P}_k$ will correctly extend the binary numeral with zeros, for example

$$\mathcal{P}_4(13) = 13\backslash 2^4 \bmod 2 = 0$$

We prove some useful properties of this function, which we rely on for developing our implementation. These properties should come as no surprise when interpreting $\mathcal{P}_k(n)$ as function returning the bit at position k of integer n.

Lemma 2.2 The generalized parity function $\mathcal{P}_k(n)$ has a period $p_k = 2^{k+1}$.
Proof *By direct calculation.*

$$\begin{aligned} \mathcal{P}_k(n + p_k) &= \left\lfloor \frac{n + 2^{k+1}}{2^k} \right\rfloor \bmod 2 \\ &= \left\lfloor \frac{n}{2^k} + 2 \right\rfloor \bmod 2 \\ &= \left(\left\lfloor \frac{n}{2^k} \right\rfloor \bmod 2 \right) + (2 \bmod 2) \\ &= \mathcal{P}_k(n) + 0 = \mathcal{P}_k(n) \end{aligned}$$

[2] The least significant bit resides at position zero. The common two's complement representation would qualify.

Lemma 2.3 Within a single period $p_k = 2^{k+1}$, $\mathcal{P}_k$ equals 0 in the first half and 1 in the second half. We call the half-period $b_k = 2^k$ the even *block* and the second half the odd *block*. Symbolically

$$\forall \lambda \in \mathbb{N} : \begin{cases} \mathcal{P}_k(2\lambda b_k + \sigma) = 0 \\ \mathcal{P}_k((2\lambda+1)b_k + \sigma) = 1 \end{cases} \quad \text{where} \quad 0 \leqslant \sigma < b_k \tag{2.10}$$

Proof *By direct calculation.* We prove only the first equation of (2.10), the second follows by analogy.

$$\begin{aligned} \mathcal{P}_k(2\lambda b_k + \sigma) &= \left\lfloor \frac{2\lambda 2^k + \sigma}{2^k} \right\rfloor \bmod 2 \\ &= \left\lfloor 2\lambda + \frac{\sigma}{2^k} \right\rfloor \bmod 2 \end{aligned}$$

because $\lambda \in \mathbb{N}$ and $\sigma < 2^k$, we find

$$\begin{aligned} \mathcal{P}_k(2\lambda b_k + \sigma) &= 2\lambda \bmod 2 \\ &= 0 \end{aligned}$$

2.3 Positional Operators

As established by Postulate 2, the application of a unitary operator to a quantum state results in an evolved quantum state. These unitary operators will form the basis of quantum computations. We define a class of operators on quregisters, describing the application of an operator to one of its qubits.

Definition 2.4 *Positional operator.* For any unitary operator $U : \mathbb{C}^2 \to \mathbb{C}^2$, we define the positional operator $U_t : \mathcal{H} \to \mathcal{H}$ as

$$U_t = \bigotimes_i U^{[i=t]} \tag{2.11}$$

We prove the following lemma, ensuring the unitarity of a positional operator.

Lemma 2.4 Two unitary operators A and B have a unitary tensor product.
Proof We must show that the tensor product $A \otimes B$ satisfies the definition of a unitary operator $UU^\dagger = I$

$$\begin{aligned} (A \otimes B)(A \otimes B)^\dagger &= (A \otimes B)(A^\dagger \otimes B^\dagger) \\ &= AA^\dagger \otimes BB^\dagger \end{aligned}$$

Indeed, for unitary A and B, we find (omitting the dimensions of the identity)

$$(A \otimes B)(A \otimes B)^\dagger = I \otimes I = I$$

Example 2.4 Consider the X operator $\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$. The positional operators X_2 and X_3 on a quregister $|\psi\rangle$ of 3 qubits become

$$X_2 = I \otimes X \otimes I \qquad X_3 = I \otimes I \otimes X$$
$$= \begin{pmatrix}1&0\\0&1\end{pmatrix} \otimes \begin{pmatrix}0&1\\1&0\end{pmatrix} \otimes \begin{pmatrix}1&0\\0&1\end{pmatrix} \qquad = \begin{pmatrix}1&0\\0&1\end{pmatrix} \otimes \begin{pmatrix}1&0\\0&1\end{pmatrix} \otimes \begin{pmatrix}0&1\\1&0\end{pmatrix}$$
$$= \begin{pmatrix}0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\end{pmatrix} \qquad = \begin{pmatrix}0&1&0&0&0&0&0&0\\1&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{pmatrix}$$

Consider the tensor product's mixed product property

$$(A \otimes B)(C \otimes D) = (AC \otimes BD)$$

For a quregister in a separable state, we can distribute U_t over the individual qubits

$$U_t|\psi\rangle = \left(\bigotimes_i U^{[i=t]}\right)\left(\bigotimes_i |\psi_i\rangle\right) = \bigotimes_i \left(U^{[i=t]}|\psi_i\rangle\right)$$

applying the operator U to qubit $|\psi_t\rangle$ and the identity I to the others.

Example 2.5 The application of X_3 on a separable $|\psi\rangle$ of three qubits $|\psi_i\rangle = \left(\begin{smallmatrix}\alpha_i\\\beta_i\end{smallmatrix}\right)$ applies X to the third qubit

$$\begin{aligned} X_3|\psi\rangle &= \left(X^{[1=3]} \otimes X^{[2=3]} \otimes X^{[3=3]}\right)\left(|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle\right) \\ &= (I \otimes I \otimes X)\left(|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle\right) \\ &= I\begin{pmatrix}\alpha_1\\\beta_1\end{pmatrix} \otimes I\begin{pmatrix}\alpha_2\\\beta_2\end{pmatrix} \otimes X\begin{pmatrix}\alpha_3\\\beta_3\end{pmatrix} \\ &= \begin{pmatrix}\alpha_1\\\beta_1\end{pmatrix} \otimes \begin{pmatrix}\alpha_2\\\beta_2\end{pmatrix} \otimes \begin{pmatrix}\beta_3\\\alpha_3\end{pmatrix} \end{aligned}$$

In general, quantum entanglement prohibits the use of tensor decomposition to find the action of a positional operator on individual qubits. A direct matrix representation becomes unwieldy, as the number of dimensions of $\mathcal{H}$ increases exponentially with the number of qubits. We will therefore describe positional operators as acting on the computational basis vectors, thereby obtaining a complete yet compact notation using the generalized parity function.

Lemma 2.5 A basis vector $|i\rangle$ $(0 \leqslant i < 2^n)$ has the tensor decomposition

$$|i\rangle = \bigotimes_j |\mathcal{P}_{n-j}(i)\rangle \tag{2.12}$$

Thus, we find the action of a positional operator U_t on the basis

$$U_t|i\rangle = \left(\bigotimes_j U^{[j=t]}\right)\left(\bigotimes_j |\mathcal{P}_{n-j}(i)\rangle\right) = \bigotimes_j U^{[j=t]}|\mathcal{P}_{n-j}(i)\rangle \tag{2.13}$$

Example 2.6 Continuing examples 2.4 and 2.5, consider the state space $\mathcal{H} = \mathbb{C}^8$ of 3-qubit registers, and its computational basis $\mathtt{B}_3$.

$$\begin{aligned}
X_2|0\rangle &= X_2\big(|\mathcal{P}_2(0)\rangle \otimes |\mathcal{P}_1(0)\rangle \otimes |\mathcal{P}_0(0)\rangle\big) \\
&= I|\mathtt{0}\rangle \otimes X|\mathtt{0}\rangle \otimes I|\mathtt{0}\rangle \\
&= |\mathtt{0}\rangle \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes |\mathtt{0}\rangle \\
&= |\mathtt{0}\rangle \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes |\mathtt{0}\rangle \\
&= |\mathtt{0}\rangle \otimes |\mathtt{1}\rangle \otimes |\mathtt{0}\rangle \\
&= |2\rangle
\end{aligned}$$

$$\begin{aligned}
X_3|6\rangle &= X_3\big(|\mathcal{P}_2(6)\rangle \otimes |\mathcal{P}_1(6)\rangle \otimes |\mathcal{P}_0(6)\rangle\big) \\
&= I|\mathtt{1}\rangle \otimes I|\mathtt{1}\rangle \otimes X|\mathtt{0}\rangle \\
&= |\mathtt{1}\rangle \otimes |\mathtt{1}\rangle \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
&= |\mathtt{1}\rangle \otimes |\mathtt{1}\rangle \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&= |\mathtt{1}\rangle \otimes |\mathtt{1}\rangle \otimes |\mathtt{1}\rangle \\
&= |7\rangle
\end{aligned}$$

We can now fully express a quregister $|\psi\rangle$ as the result of the tensor product of n qubits $|\psi_j\rangle = \psi_j^{\mathtt{0}}|\mathtt{0}\rangle + \psi_j^{\mathtt{1}}|\mathtt{1}\rangle$ as a linear combination of the basis vectors $|i\rangle$

$$\begin{aligned}
|\psi\rangle &= \bigotimes_j \psi_j^{\mathtt{0}}|\mathtt{0}\rangle + \psi_j^{\mathtt{1}}|\mathtt{1}\rangle \\
&= \sum_i \prod_j \psi_j^{\mathcal{P}_{n-j}(i)}|i\rangle
\end{aligned} \tag{2.14}$$

Using equations (2.14) and (2.13), we can now generally express the application of a positional operator U_t on a quregister $|\psi\rangle$ of n qubits $|\psi_j\rangle = \psi_j^{\mathtt{0}}|\mathtt{0}\rangle + \psi_j^{\mathtt{1}}|\mathtt{1}\rangle$

$$\begin{aligned}
U_t|\psi\rangle &= \sum_i \prod_j \psi_j^{\mathcal{P}_{n-j}(i)} U_t|i\rangle \\
&= \sum_i \prod_j \psi_j^{\mathcal{P}_{n-j}(i)} \bigotimes_k U^{[k=t]}|\mathcal{P}_{n-k}(i)\rangle
\end{aligned} \tag{2.15}$$

As we will see in Chapter 4, this equation will prove useful as a basis for a parallel implementation. Before describing the actual operations in the measurement calculus, we need to construct a second class of operators.

Definition 2.5 *Controlled operator.* For any unitary operator $U : \mathbb{C}^2 \to \mathbb{C}^2$, we define the controlled operator $\widehat{U}_{c,t} : \mathcal{H} \to \mathcal{H}$, acting on the computational basis $\mathtt{B}_n$

$$\widehat{U}_{c,t}|i\rangle = U'_t|i\rangle \qquad \text{where} \quad U' = U^{\mathcal{P}_{n-c}(i)} \tag{2.16}$$

applying a positional operator U_t to all $(n-c)$-odd basis vectors and the identity to the others[3].

Example 2.7 Consider a quregister $|\psi\rangle$ of two qubits. The controlled operator $\widehat{X}_{1,2}$ acts on the second qubit with the first as a control qubit

$$\widehat{X}_{1,2} : \mathbb{C}^4 \to \mathbb{C}^4 : \begin{cases} |0\rangle \mapsto I \otimes X^{\mathcal{P}_1(0)}|0\rangle = I|\mathtt{0}\rangle \otimes I|\mathtt{0}\rangle & = & |0\rangle \\ |1\rangle \mapsto I \otimes X^{\mathcal{P}_1(1)}|1\rangle = I|\mathtt{0}\rangle \otimes I|\mathtt{1}\rangle & = & |1\rangle \\ |2\rangle \mapsto I \otimes X^{\mathcal{P}_1(2)}|2\rangle = I|\mathtt{1}\rangle \otimes X|\mathtt{0}\rangle & = & |3\rangle \\ |3\rangle \mapsto I \otimes X^{\mathcal{P}_1(3)}|3\rangle = I|\mathtt{1}\rangle \otimes X|\mathtt{1}\rangle & = & |2\rangle \end{cases}$$

Analoguous to equation (2.15), we use (2.14) to obtain the application of a controlled operator $\widehat{U}_{c,t}$ to a quregister $|\psi\rangle$

$$\widehat{U}_{c,t}|\psi\rangle = \sum_i \prod_j \psi_j^{\mathcal{P}_{n-j}(i)} \bigotimes_k U^{[k=t]\mathcal{P}_{n-c}(i)}|\mathcal{P}_{n-k}(i)\rangle \tag{2.17}$$

2.4 Measurement calculus

At its execution level, the measurement calculus performs three classes of operations on quregisters: *entanglement*, *measurement* and *correction* [42]. Four basic operators capture these operations.

Definition 2.6 *X and Z operators.* The X and Z correction operators, sometimes called σ_x and σ_z [24]

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \text{and} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.18}$$

From this, we can deduce their actions on the basis vectors $|\mathtt{0}\rangle$ and $|\mathtt{1}\rangle$

$$X : \mathbb{C}^2 \to \mathbb{C}^2 : \begin{cases} |\mathtt{0}\rangle \mapsto |\mathtt{1}\rangle \\ |\mathtt{1}\rangle \mapsto |\mathtt{0}\rangle \end{cases} \tag{2.19}$$

$$Z : \mathbb{C}^2 \to \mathbb{C}^2 : \begin{cases} |\mathtt{0}\rangle \mapsto \ \ |\mathtt{0}\rangle \\ |\mathtt{1}\rangle \mapsto -|\mathtt{1}\rangle \end{cases} \tag{2.20}$$

[3]Some notational ambiguity may arise. When $\mathcal{P}_c(i) = 0$, the operator U' equals I, the identity; the controlled operator $\widehat{U}_{c,t}$ then becomes I_t. While the notation I_t usually specifies the dimension t of the identity I, we denote a positional operator applying I at position t, resulting in an identity of dimension 2^n, with n the number of qubits. We avoid this confusion by consistently leaving the dimensions of identities implicit.

Using equation (2.13) we find the corresponding positional operators, acting on a basis vector $|i\rangle$ as

$$X_t|i\rangle = |i + \sigma_{n-t}(i)2^{n-t}\rangle \qquad \text{and} \qquad Z_t|i\rangle = \sigma_{n-t}(i)|i\rangle \tag{2.21}$$

where $\sigma_k(i) = (-1)^{\mathcal{P}_k(i)}$ equals -1 for a t-odd i and $+1$ otherwise.

Definition 2.7 *Controlled Z entanglement operator.* This operator entangles two qubits. Its definition follows from (2.20) and the definition of controlled operators.

$$\widehat{Z}_{c,t}|i\rangle = (-1)^{\mathcal{P}_{n-c}(i)\mathcal{P}_{n-t}(i)}|i\rangle \tag{2.22}$$

Definition 2.8 *Measurement operator.*

Chapter 3

Implementation Platform

In this chapter, we explore the constraints on the QVM implementation imposed by the targeted multicore architecture and introduce some of the available tools for developing parallel software. Furthermore, we show how to measure and analyze the results and decide on an appropriate experimental setup. To illustrate the properties of our platform, we interleave the sections of this chapter with the implementation of simple example algorithms, preluding the actual implementation process.

3.1 Computational Model

The high complexity of modern processors, especially multicore processors, makes it hard to make accurate or relevant predictions regarding algorithmic performance. The classic random access model, assuming a flat memory address space and unit-cost memory access, disregards the cache hierarchy, thereby underestimating the effect of data movement on execution speed [14, 35].

Cache effects and memory bandwidth exert an even bigger influence on multicore applications [35]. When multiple processing units operate on disjoint parts of the same cache line, data might needlessly move between cache levels, a problem known as *false sharing* [35, 37]. In symmetric multiprocessors (SMPs), main-memory access of all CPUs requires passing through the same front-side bus (FSB). This bottleneck limits memory bandwidth, resulting in latency increasing with the number of processors [16]. More advanced solutions attach different sections of the memory to different CPUs, known as non-uniform memory architecture (NUMA), where remote memory access comes at a higher cost than accessing local memory.

Several authors have identified *dwarfs* of scientific computing: classes of applications related by similarity in operations and data movement. The dwarfs allow for capturing the common requirements of applications while remaining reasonably independent of specific implementations. [10]. Considering the seven dwarfs defined by [10], a QVM qualifies as a Class 3 dwarf (spectral methods),

related to algorithms like Fast-Fourrier Transform [19]. It seems tempting to situate our QVM in Class 1 or 2 (dense or sparse linear algebra), however, we do not directly implement the linear algebra formulation of the MC; with the benefit of foresight, we anticipate on mainly performing data permutation and very little computation. It follows that memory access patterns will determine the performance of our PQVM.

3.2 Development Tools

The transformation of a single-threaded program into a multithreaded program requires a careful redesign. Ideally, we want to obtain a portable program[1] while maintaining optimal performance; concretely, we need to balance programming effort and performance loss [29]. Several libraries and languages for parallel programming exist, spanning different abstraction levels.

The POSIX threads library (`Pthreads`) provides a low level threading API. It requires explicit control of threads, leaving the programmer with great flexibility, at the expense of high complexity of manually managing synchronization primitives, such as locks, mutexes, semaphores, barriers and condition variables [29,35]. `OpenMP` (`OMP`), a basic multithreading tool for `C`, `C++` or `Fortran` programs, offers some basic abstractions of threads, in the form of compiler directives (`pragma`) marking parallel sections or loops [4, 29, 35].

`Intel Cilk++`, a language based on `Cilk` [1], extends the `C++` language with extra keywords telling the compiler which parts to make parallel. Divide and conquer algorithms perform well in this language. Additionally, `Cilk++` provides some code analysis tools, e.g. for detecting race condition [29].

`Intel Array Building Blocks` (`ArBB`), designed to use the features of existing and upcoming multicore and many core processors, extends codeC++ for complex data parallelism. `ArBB` provides a rich set of data containers allowing high-level expression of parallel programs [29]. Intel discontinued this project earlier this year [2].

`Intel Threading Building Blocks` (`TBB`) provides task-based parallelism; hiding actual threads from the programmer, `TBB` provides a library allowing the programmer to formulate a parallel program at a high level of abstraction, in the form of tasks executed by its runtime environment [23, 29].

Example 3.1 We will write an example algorithm[2] with `TBB` to normalize vectors of real and complex numbers. For arbitrary vectors, we need a two-step process: first calculate the norm (a reduce operation), then divide each component of the vector by this norm (a map operation). For brevity, the example below only includes the reduction step.

[1] from small multicore to huge manycore machines, different memory layouts, and different cache hierarchies

[2] complete source code at `http://github.com/kmoerman/pqvm`

Using TBB's `parallel_reduce` function requires writing a class with an overloaded `operator()` method encapsulating the work for a certain range of the vector, and a `join` method combining the results of two tasks. The TBB runtime will split a given range and assign a parallel task to each of its parts, or recursively split down to an appropriate size for a parallel task [23].

```
1  typedef tbb::blocked_range<size_t> range;
2  typedef double real;
3  typedef std::complex<double> complex;
4
5  inline real inner_product_term (const real a, const real b) {
6      return a * b;
7  }
8
9  inline complex inner_product_term(const complex a, const complex b) {
10     return a * std::conj(b);
11 }
12
13 template <typename number>
14 class inner_product {
15     vector<number>& vec;
16     number part;
17 public:
18     inner_product (std::vector<number>& v): vec(v), part(0) {}
19     inner_product (inner_product& ip, split): vec(ip.vec), part(0) {}
20
21     void operator() (const range& rng) {
22         for (size_t i = rng.begin(), n = rng.end(); i != n; ++i)
23             part += inner_product_term(vec[i], vec[i]);
24     }
25
26     void join (const inner_product& ipr) {
27         part += ipr.part;
28     }
29 };
30
31 template <typename number>
32 normalize_parallel (std::vector<number>& vec) {
33     tbb::parallel_reduce(range (0, vec.size()), inner_product (vec));
34     number norm = std::sqrt(ipr.part);
35     // ... divide vec by norm
36 }
37
38 template <typename number>
```

```
39 normalize_sequential (std::vector<number>& vec) {
40     number norm;
41     for (size_t i = vec.begin(), n = vec.end(); i != n; ++i)
42         norm += inner_product_term(vec[i], vec[i]);
43     norm = std::sqrt(norm);
44     // ... divide vec by norm
45 }
```

3.3 Testing Environment

We executed code on two testing machines, both in a standard C++ environment, with TBB version 4.1. We used the first, called `wilma` for exploratory work, including the examples in this chapter. This machine consists of 8 processors in a shared memory architecture.

```
Intel(R) Xeon(R) CPU X5687 @ 3.60GHz GenuineIntel GNU/Linux
Linux wilma 3.2.45-wilma #1 SMP Thu May 16 13:02:41 CEST 2013 x86_64

g++ (GCC) 4.7.1, -O2 optimization level
```

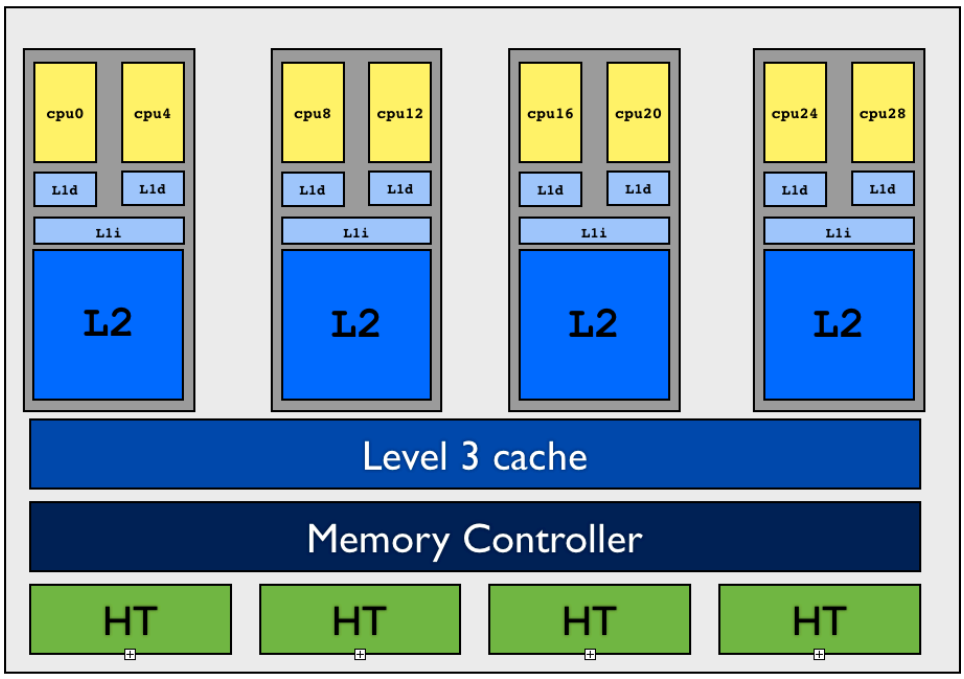

Figure 3.1 Serenity node model [41]

We used second machine, called `serenity`, provided by the computer science department, for the actual implementation.

```
AMD Opteron 6300 Abu Dhabi 16 core
Linux serenity 3.8.0-19-generic #29-Ubuntu SMP Wed Apr 17 18:16:28 UTC
2013 x86_64 x86_64 x86_64 GNU/Linux

g++ (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3, -O2 optimization level
```

This machine consists of 64 processors, distributed across 8 nodes in a NUMA architecture, giving each node its own memory controller. Figure 3.1 illustrates the structure of a node.

3.4 Experimental Analysis

We need to define some performance metrics for our software and hardware to allow for an objective evaluation of the results. Predominantly, we want to analyze the *scaling* behavior of a parallel program. *Weak scaling* characterizes the change in execution time as the problem size changes, while keeping the number of processors fixed. Conversely, *strong scaling* characterizes the change in execution time as the number of processors changes, while keeping the problem size fixed.

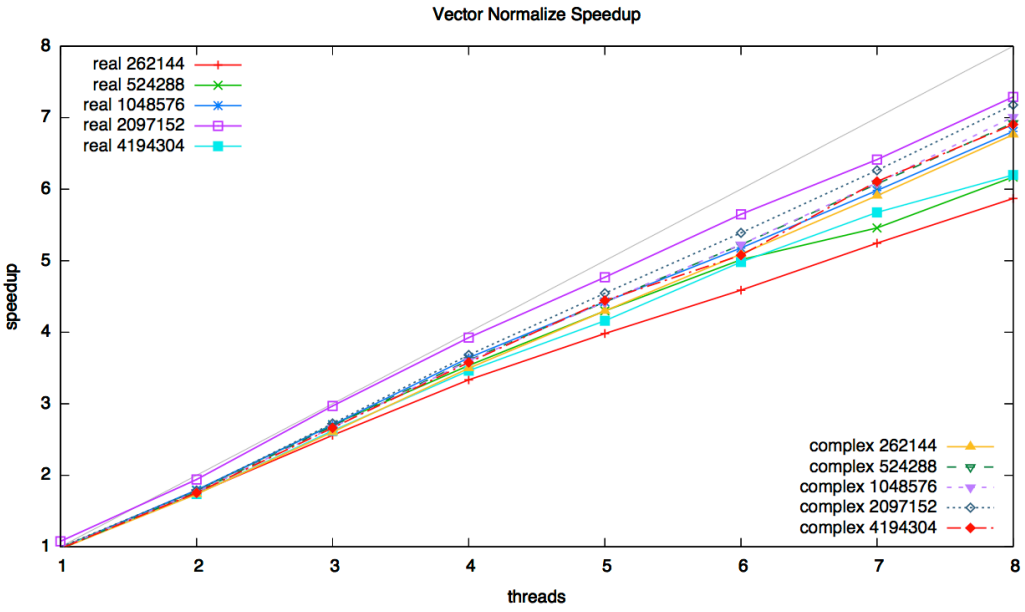

Figure 3.2 Speedup of normalizition of real and complex vectors of different sizes. The diagonal shows the ideal speedup.

Definition 3.1 *Speedup.* As a basic performance metric of parallel performance, we define the speedup S_p on p processors as the ratio of the sequential execution

time T_1 and the parallel execution time T_p.

$$S_p = \frac{T_1}{T_p} \tag{3.1}$$

We call $S_p = p$ the *ideal* or *linear* speedup, occurring when a program scales perfectly with the number of processors. In practice, *superlinear speedup* may occur; as the problem sizes for each processor diminish, more of the data can reside in the caches instead of main memory, speeding up memory access [35]. Note that in order to obtain a correct measure of speedup, accounting for parallel overhead, one must compare the parallel algorithm with the sequential algorithm, not just with the single-processor version of the parallel algorithm. Plotting the speedup for different numbers of processors results in a speedup graph, obviously, this graph strongly depends on the specific problem and implementation.

Definition 3.2 *Efficiency.* The efficiency E_p of a program running on p processors indicates the average efficiency of each processor.

$$E_p = \frac{S_p}{p} \tag{3.2}$$

An efficiency value of 1 corresponds to linear speedup. In general, the efficiency will amount to less than 1 and will diminish with the number of processors due to various sources of performance loss [35].

Example 3.2 We measure the speedup of the vector normalization algorithm above. Figure 3.2 displays the speedup graph for several vector sizes, overall showing a good result with efficiencies between 0.75 and 0.88.

Definition 3.3 *Amdahl's Law.* Ultimately, the presence of non-parallellizable code limits the maximally attainable speedup of a program. If s quantifies the total serial part of a program, then $1 - s$ quantifies the total parallel part. Assuming perfect speedup of the parallel part, Amdahl's Law gives us an upper bound $\overline{S_p}$ on the total speedup

$$\overline{S_p} = \frac{T_1}{sT_1 + (1-s)\frac{T_1}{p}} = \left(s + \frac{1-s}{p}\right)^{-1} \tag{3.3}$$

As the numbers of processors increases, the second term vanishes and $\overline{S_p} \to 1/s$. Clearly, for a program with a negligible serial part, $\overline{S_p} \to p$ as $s \to 0$; we call such a program *embarrassingly parallel*. Note that Amdahl's Law overestimates the upper bound in assuming limitless linear scaling of the parallel part.

While Amdahl's Law quantifies the influence of sequential bottlenecks in a program, it does not predict the actual parallel performance. As off-chip memory bandwidth constrains parallel performance of multicore programs and in particular of our application, we need a model to relate processor performance and memory traffic [43].

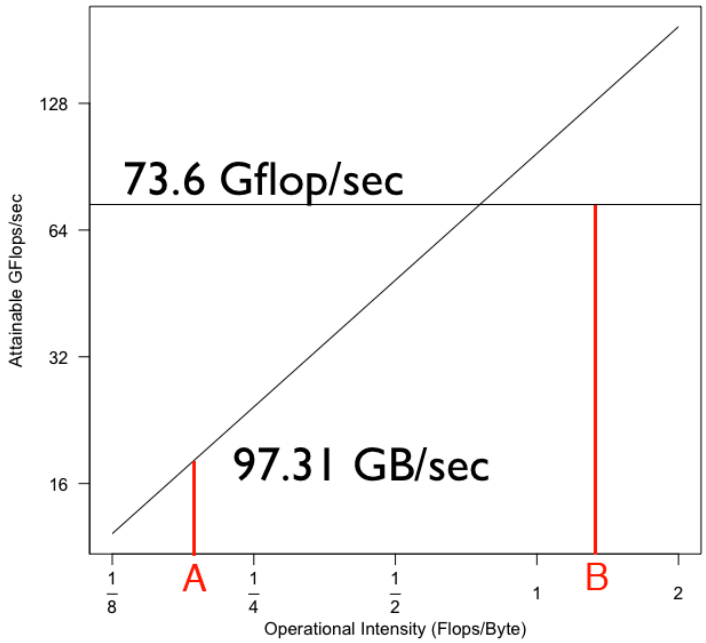

Figure 3.3 Roofline characteristics of `serenity`, our testing machine [41]. A memory-bound kernel *A* hits the slanted part of the roof, a compute-bound kernel *B* hits the flat part.

We use the therm *operational intensity*, meaning the operations per byte of DRAM traffic, as a measure of necessary DRAM bandwidth needed by a kernel (the core computation) on a particular machine. The *roofline* model, defined in [43], relates floating-point performance, operational intensity, and memory performance of a machine in a 2D graph. For a kernel with a given operational intensity, peak performance coincides either with peak memory performance, in which case we call the kernel *memory-bound*, or it coincides with maximal floating-point performance, which we call it a *compute-bound* kernel. Optimally, a computational kernel would hit the intersection, meaning that it maximally uses both memory bandwidth and processor speed.

Like Amdahl's law, this model provides only an upper bound, however, the authors suggest a series of *ceilings* an implementation must break through in order to achieve maximal performance. We will focus only on the memory bottlenecks, as our application will turn out strongly memory-bound due to a low operational intensity.

1. Optimizing for unit-stride memory access engages hardware prefetching, significantly reducing latency.

2. Ensure memory affinity, allocating memory and operating on data within the same memory-processor pair improves access times. This applies specif-

ically to the NUMA architecture of `serenity`.

3. Use software prefetching.

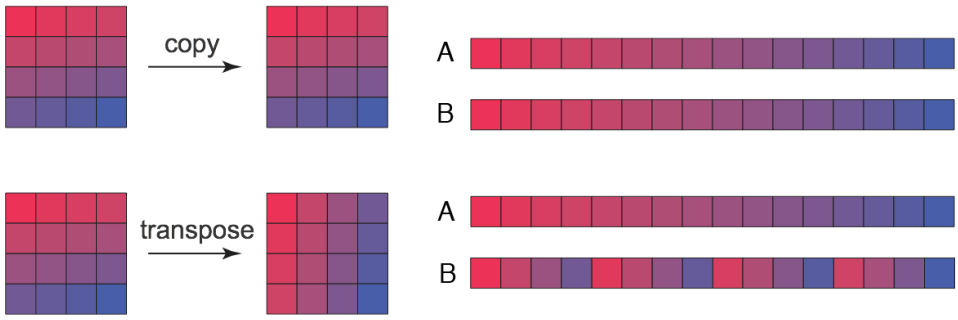

Figure 3.4 Transposition of a matrix in row-major storage format, compared to simple copying of the same matrix. Similar colours represent cache affinity.

Example 3.3 As an example of a memory-bound algorithm, we implement a matrix transposition algorithm. Consider the naive (sequential) implementation to transpose a row-major matrix A into a row-major matrix $B = A^\top$.

```
1 void transpose_naive (double** A, double** B, size_t m, size_t n) {
2     for (size_t i = 0; i < m; ++i)
3         for (size_t j = 0; j < n; ++j)
4             B[j][i] = A[i][j];
5 }
```

Due to the row-major storage scheme, the reading of matrix A uses unit stride, but the writing of B does not, as illustrated in Figure 3.4.

To make better use of the caches and hardware prefetching, we will implement an algorithm based on the scheme presented by Frigo et al. [19] For simplicity, we will limit ourselves to $n \times n$ (square) matrices, where $n = 2^k$ with k a non-negative integer. For a matrix A containing four submatrices A_i, we have

$$A^\top = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}^\top = B$$
$$\begin{pmatrix} A_1^\top & A_3^\top \\ A_2^\top & A_4^\top \end{pmatrix} = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

This relation suggests a divide and conquer approach: if we recursively divide A and B in four blocks, we will arrive at a point where, in principle, a source block from A and its corresponding destination block in B could fit in the cache. Copying the elements from the A block to the B block can then happen in an arbitrary order. Hence, we can simply continue splitting recursively, up to blocks

of unit size, ignorant of when the blocks become small enough to fit in the cache. We call this approach *cache-oblivious*, as we don't need any details regarding the cache or cache-line size and yet obtain an implementation that uses the caches effectively.

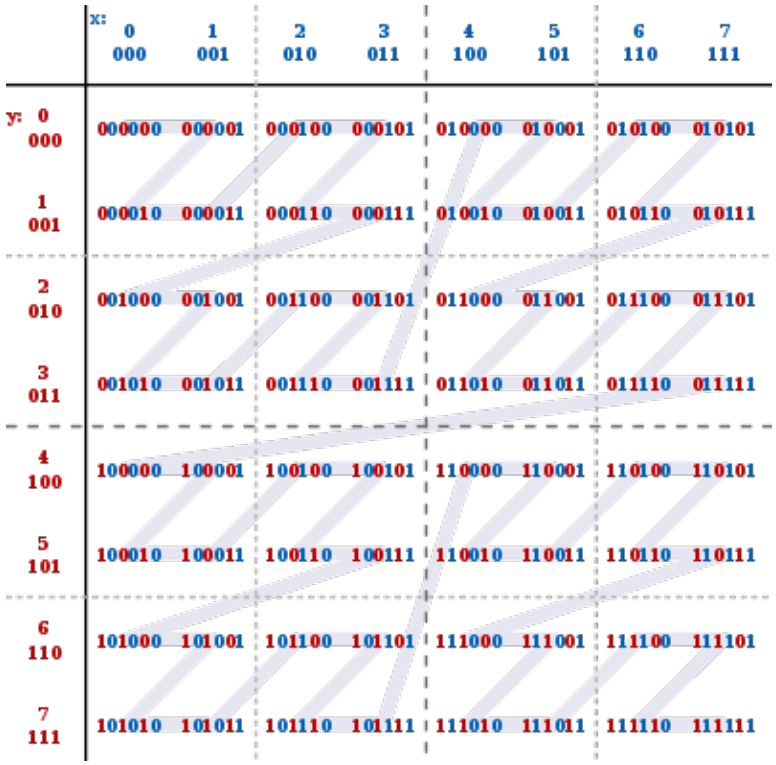

Figure 3.5 Z-order traversal of a matrix [7].

To avoid recursive function calls, we will use a *Z-order*[3] traversal of the matrix, illustrated in Figure 3.5. For an element of matrix A at row i and column j, we find the Z-value by interleaving the binary representations of i and j. The reverse operation, deinterleaving, turns a Z-value into a row and column index. By definition, the Z-order traversal results in the recursive block structure described above. Note that compared to the naive implementation, the divide and conquer method executes the same data movements (`B[j][i] = A[i][j]`), all we did was change the order of the operations.

```
1 inline void deinterleave (uint64_t z, uint64_t* row, uint64_t* col) {
2
3     z = (z & 0x9999999999999999) | ((z >>  1) & 0x2222222222222222)
4                                  | ((z <<  1) & 0x4444444444444444);
5     z = (z & 0xC3C3C3C3C3C3C3C3) | ((z >>  2) & 0x0C0C0C0C0C0C0C0C)
6                                  | ((z <<  2) & 0x3030303030303030);
```

[3] sometimes called *Morton order*

```
7       z = (z & 0xF00FF00FF00FF00F) | ((z >>  4) & 0x00F000F000F000F0)
8                                    | ((z <<  4) & 0x0F000F000F000F00);
9       z = (z & 0xFF0000FFFF0000FF) | ((z >>  8) & 0x0000FF000000FF00)
10                                   | ((z <<  8) & 0x00FF000000FF0000);
11      z = (z & 0xFFFF00000000FFFF) | ((z >> 16) & 0x00000000FFFF0000)
12                                   | ((z << 16) & 0x0000FFFF00000000);
13
14      *col = z & 0x00000000FFFFFFFF;
15      *row = z >> 32;
16  }
17
18  void transpose_dc (double** A, double** B, size_t n) {
19      size_t i, j;
20      for (size_t z = 0, z_max = n * n; z < z_max; ++z) {
21          deinterleave(z, &i, &j);
22          B[j][i] = A[i][j];
23      }
24  }
```

We omit the parallel transpose adaptations[4], as the sequential version already captures the essence. Figure 3.6 displays the speedup results. Note that superlinear scaling occurs for the smaller matrices, as they still fit perfectly in cache memory. For the larger matrices of 2048 by 2048 and 4096 by 4096 `double` elements, the divide and conquer method clearly scales better, with efficiencies of about 1, compared to the naive efficiency of about 0.38.

If we want to match the metrics introduced above with an actual implementation, we need methods of analyzing our code and the resulting program. The `libpapi` library [5] provides some performance analysis functions for accessing hardware counters, which we must manually include in the code. We provide a header file `performance.h` wrapping some of the functionality, which we use in some of our profiling programs, to select specific parts we want to measure.

The `perf` tools, a collection of profiler tools, periodically interrupts a running program to collect hardware counters and call-stack info, this allows for less obtrusive measuring of the codes performance. We use the command `perf stat` to quickly assess run time behaviour (running times, cache misses, CPU workload). With `perf record`, one can record events during the execution of a program; `perf report` provides an interface for analyzing the recorded events, `perf annotate` supplies the source code with counted events (function calls, cache misses, etc.) [6].

[4]The source code at `http://github.com/kmoerman/pqvm` contains the straightforward yet verbose parallel versions of both the naive and the divide and conquer methods.

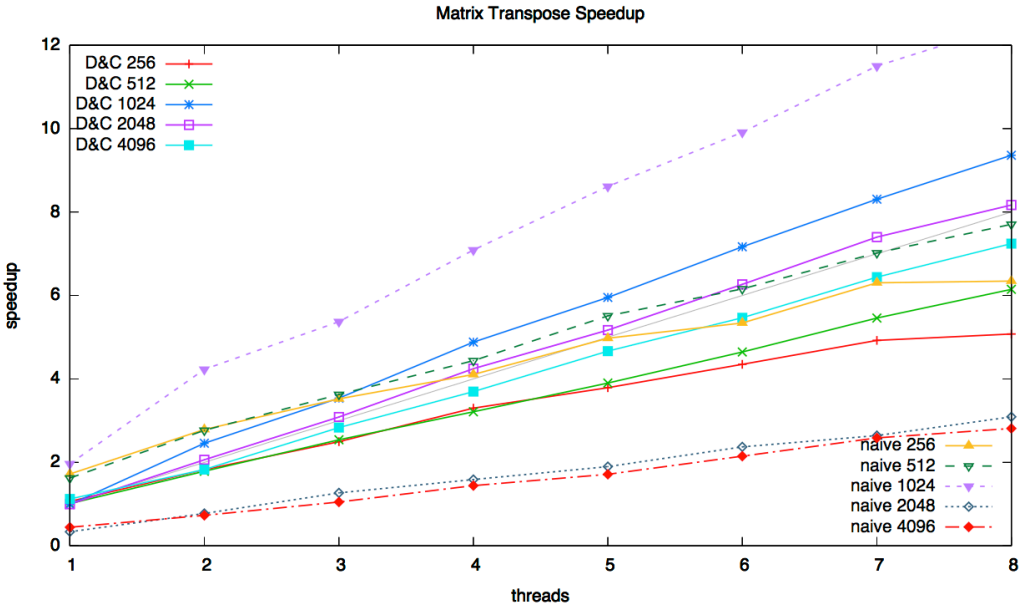

Figure 3.6 Matrix transpose using a naive and a divide and conquer algorithm for square matrices of different sizes.

Chapter 4

Implementing the Virtual Machine

We can now expose the work on `pqvm`, our implementation of the execution layer of the quantum programming framework introduced in the first chapter. We developed the mathematical model for the basic operations in MC in Chapter 2, from which we can derive our implementation, following the guidelines set out in Chapter 3. A `Git` repository on `https://github.com/kmoerman/pqvm` contains the full source code[1], we will only include the parts relevant to this document. To avoid the verbosity of `TBB` code, we will use pseudo `C++` code, especially **`parallel for`** loops, unless the exact parallel implementation requires scrutiny.

4.1 Dense State Vectors

The `libquantum` backend of `qvm`, the existing sequential VM, uses a sparse state vector implementation, saving memory by omitting the zero valued amplitudes. While bringing down the memory footprint, this approach leads to poor cache usage, resulting in a bad overall memory performance. This bottleneck corresponds to the first performance ceiling memory-bound applications must break trough (see Chapter 3). We will implement dense state vectors to improve cache usage by allowing unit stride access of a quregister. We implement both a sequential and a parallel VM, using the sequential version as a reference for speedup measurements. In this report, we include only a part of the implementation, as much of the operators share a similar structure and behaviour.

State vector We write our own `STL`-style vector class (see `vector.h`), as the classic `std::vector` class provides now way of allocating space before filling the vector concurrently. We define the `quregister` type as a `vector` of `std::complex <double>` numbers, the amplitudes of the quantum state. At index `i` of the vector, we store the amplitude α_i belonging to the computational basis state $|i\rangle$.

[1] Actually, the code can serve as a self-explanatory document on its own and contains extra implementation details that would otherwise needlessly obfuscate this report.

Tangle As we trade off memory for speed, the memory footprint of our implementation will inevitably increase compared to the original `qvm`. In general, a quregister of n qubits requires storing 2^n amplitudes, however, we retain an important optimization from the original implementation. When we decompose a quantum state as a tensor product of substates, we can effectively reduce the exponential factor and consequently the memory footprint. In fact, we do not even need to find the tensor decomposition of arbitrary state vectors, the small set of operations and the explicit entanglement operator allow us to keep track of the entangled and disentangled parts of the simulated quantum system. Thus, we split the entire quantum state space into disjoint parts called *tangles*. In the implementation, a `tangle` stores a lists of the qubits it contains and a state vector of the quantum system it represents.

Whenever the entanglement operator entangles two qubits residing in different tangles, we merge the tangles using the tensor product. As the measurement of a qubit entails its destruction, a tangle shrinks when the measuring one of its qubits; removing a qubit from a quantum state halves the number of stored amplitudes.

Kronecker product The kronecker product merges two quantum states (i.e. tangles), in accordance with Postulate 3. We split the parallel range along the left input vector; each task writes to its own contiguous part of the result vector and we obtain unit stride access for the right input vector and the result vector.

```
1  void kronecker (quregister& left, quregister& right, quregister&
       result) {
2      size_t n = left.size(),
3             m = right.size();
4
5      result.reserve(n * m);
6
7      parallel for (size_t i = 0, k = 0; i != n; ++i)
8          for (size_t j = 0; j != m; ++j; ++k)
9              result[k] = left[i] * right[j];
10 }
```

X operator Recall from Chapter 2 that applying the X_{n-k} operator on the basis vectors of a state space of n qubits will map k-even indices i to $i + 2^k$ and k-odd indices $j + 2^k$ to j, therefore, we simply need to exchange the amplitudes of k-odd and k-even indices. However, instead of implementing these exchange operations directly, the periodic behaviour of $\mathcal{P}_k$ allows us to perform block transfers Thus, we implement the X_t operator in three steps: allocate a new state vector, then copy the even blocks and, finally, copy the odd blocks. Consider the (simplified) parallel implementation of the even step

```
struct x_even {
    const size_t target;
    const iterator input, output;

    x_even (size_t t_, quregister& i_, quregister& o_) :
    target (t_), input (i_.begin()), output (o_.begin()) {}

    void operator () (range& rng) const {
        size_t stride = 1 << target,
               period = 2 * stride,
               i      = rng.begin(),
               block  = stride * sizeof(complex),
               blocks = rng.size() / period;

        iterator  ipt = input  + i,
                  opt = output + i + stride;

        while (blocks > 0) {
            memcpy(opt, ipt, block);
            opt += period;
            ipt += period;
            --blocks;
        }
    }
};
```

TBB's `parallel_for` will split the input state vector in ranges and assign an `x_even` task to each range. We then calculate the number of periods in the range and call `memcpy` to perform the data transfer of each even block.

Z operator The Z_t operator negates every odd-indexed amplitude. This task object iterates the amplitudes in a given range of the input vector and negates the odd-indexed amplitudes, skipping the even ones. To show the importance of data movement, we first show a version that copies the input vector while applying Z, in the next section we do away with the unnecessary copy.

```
struct z {
    const size_t mask;
    const iterator input, output;

    sigma_z (size_t t, quregister& in, quregister& out) :
    mask (1 << t), input (in.begin()), output (out.begin()) {}

    void operator() (const range& rng) const {
        for (size_t i = r.begin(), n = r.end(); i < n; ++i)
```

```
10              output[i] = (i & mask) ? -input[i] : input[i];
11      }
12  };
```

4.2 In Place Operations

While we cannot turn X into an in place operation, however, we can do so with Z and $\widehat{Z}$. Further reducing the workload will show a great performance benefit. The code below implements the Z operator transforming only the odd blocks.

```
1   struct z {
2       const size_t target;
3       const iterator input;
4
5       z (size_t t, quregister& in) :
6       target (t), input (in.begin()) {}
7
8       void operator() (const range& rng) const {
9           size_t i      = r.begin(),
10                 n      = r.end(),
11                 size   = n - i,
12                 block  = 1 << target,
13                 period = 2 * block;
14
15          //range contains at least one period
16          if (size >= period)
17              //skip even target blocks
18              for (i += block; i < n; i += block)
19              //loop amplitudes in odd target block
20              for (size_t j = 0; j < block; ++j, ++i)
21                  input[i] *= -1;
22
23          //range contained in a period
24          else
25              //skip if in even target block
26              if (i & block)
27              //loop ampliutdes in add target block
28              for (; i < n; ++i)
29                  input[i] *= -1;
30      }
31  };
```

Controlled Z operator The $\widehat{Z}_{c,t}$ operator applies Z_t to odd basis vectors (basis vector with a control qubit in state $|1\rangle$). The implementation resembles the Z operator, with one extra `for` nesting; for brevity we present only the in-place version.

```
struct control_z {
    const size_t control, target;
    const iterator input;

    control_z (size_t c, size_t t, quregister& in) :
    control (max(c, t)), target (min(c, t)), input (in.begin()) {}

    void operator() (const range& rng) const {
        size_t i        = r.begin(),
               n        = r.end(),
               size     = n - i,
               c_block  = 1 << control,
               c_period = 2 * c_block,
               t_block  = 1 << target,
               t_period = 2 * t_block;

        //range contains at least one control period
        if (size >= c_period)
            //skip even control blocks
            for (i += c_block + t_block; i < n; i += c_block +
                t_block)
            //skip even target blocks
            for (size_t j = 0; j < c_block; j += t_period, i +=
                t_block)
            //loop amplitudes in odd target block
            for (size_t k = 0; k < t_block; ++k, ++i)
                input[i] *= -1;

        //range contained in a control period
        else
            //skip if in even control block
            if (i & t_block);
                //...equivalent to normal z operator
};
```

4.3 Experimental Results

Individual operators We first present the speedup results of the individual operators. Each operator was executed 5 times, on random vectors of 20 qubits (2^20

complex amplitudes), allowing for ample parallelism. We only measure the time for the actual operation, the initialisation of the input state has no influence.

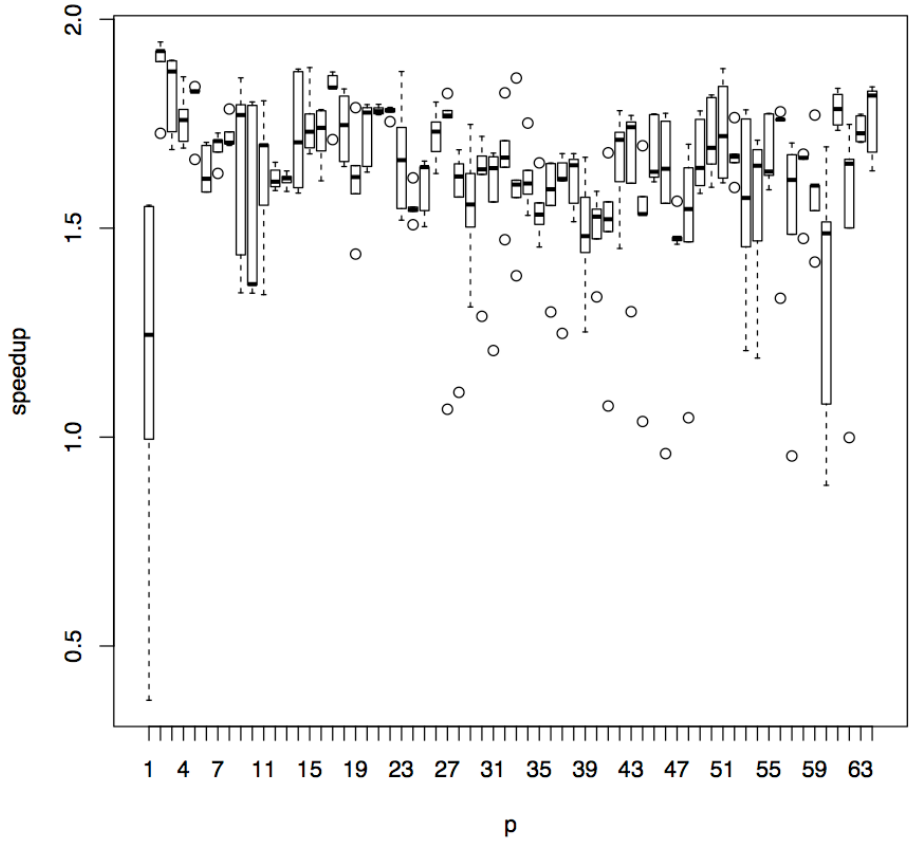

Figure 4.1 Speedup of the kronecker product. We observe no actual scaling beyond 4 processors.

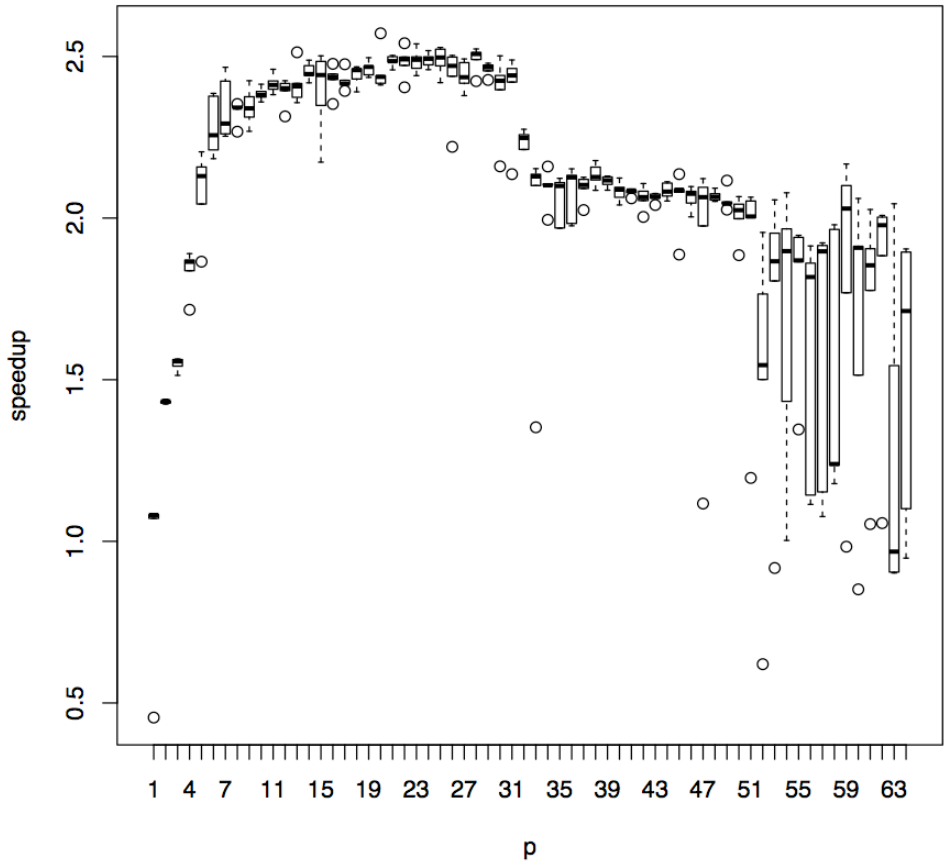

Figure 4.2 Speedup of the X operator. X exhibits sublinear scaling, which halts beyond 8 processors.

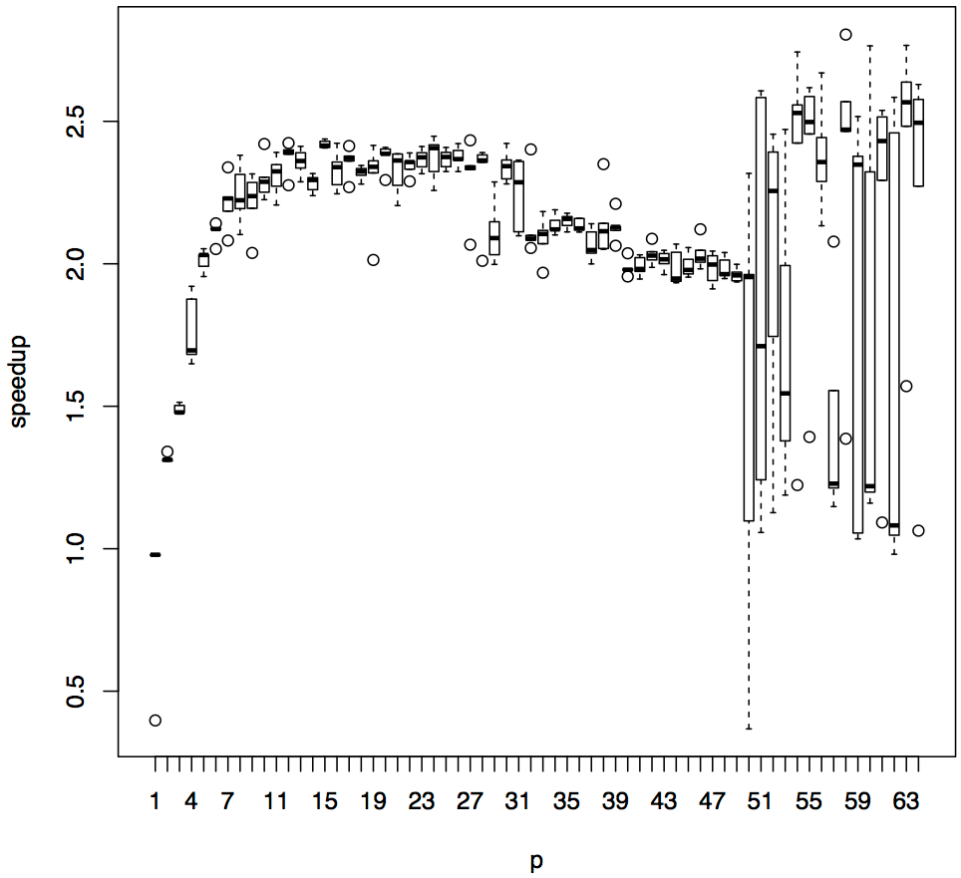

Figure 4.3 Speedup of the copying Z operator. The scaling behaviour of Z resembles that of X.

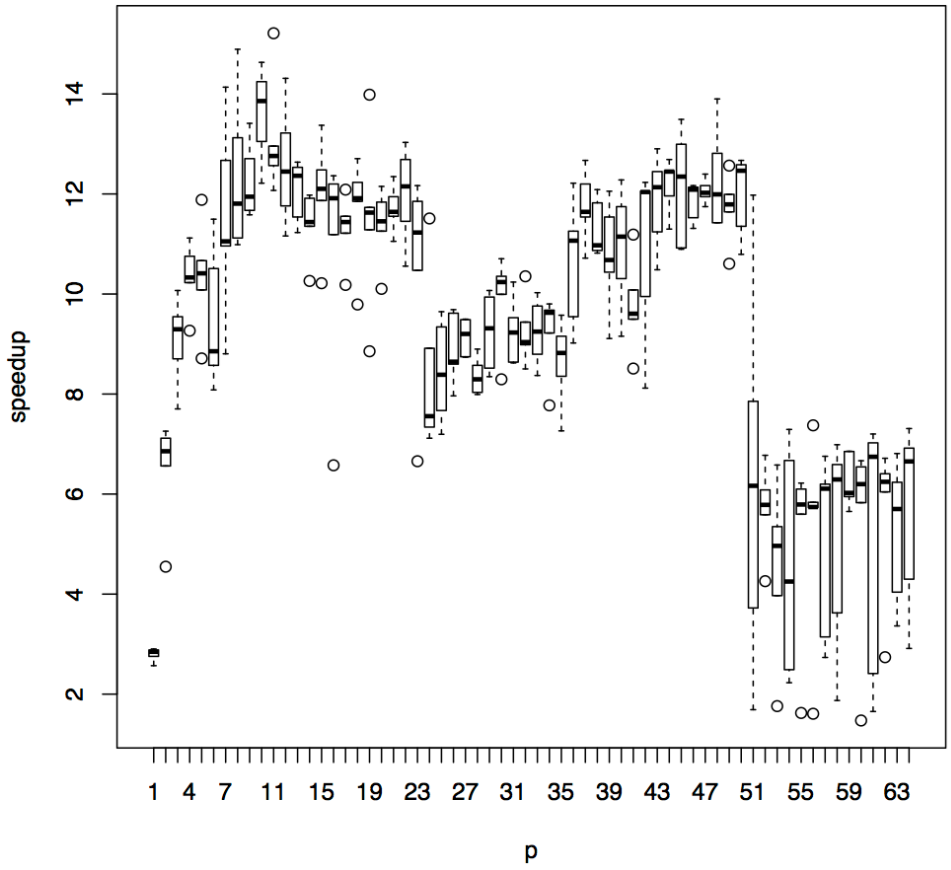

Figure 4.4 Speedup of the in-place Z operator. This version shows larger speedup, but again scaling stops beyond 8 processors.

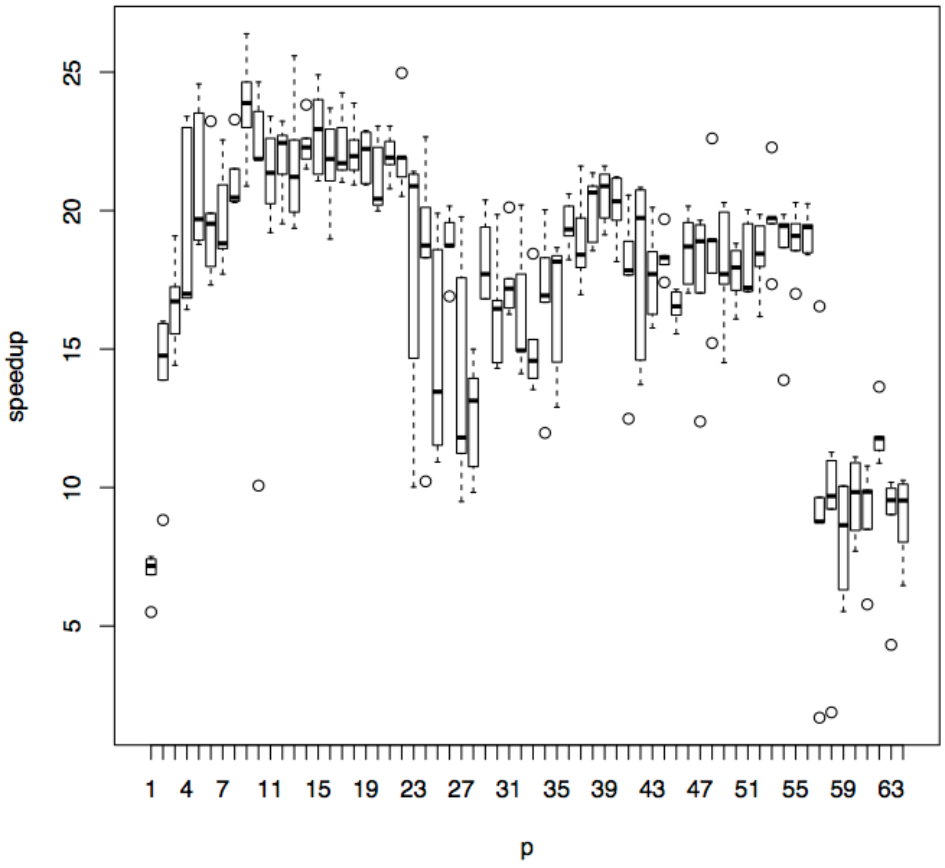

Figure 4.5 Speedup of the in-place $\widehat{Z}$ operator, similar the the Z operator.

Performance of pqvm. Next, we show the combined effect of these operations in `pqvm`. As a benchmark, we run `qft16`, a quantum fourrier transform of 16 qubits, the same benchmark used for testing the sequential version in [42]. Again we see no improvement beyond 8 processors. An inspection with `perf` on the X operator (Fgure 4.7), shows that execution spends the majority of its time in the actual tasks, with little overhead from `TBB`[2]. However, when we look at the execution of `pqvm` as a whole (Figure 4.8), we see a majority of the time spent in the `TBB` runtime (`libtbb.so.2`).

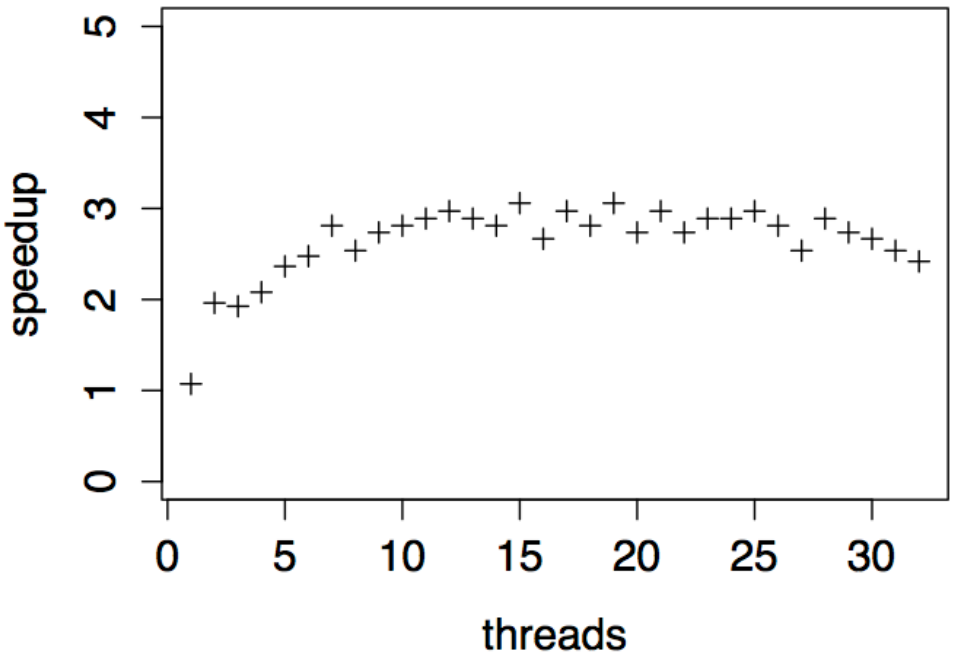

Figure 4.6 Speedup of the entire `pqvm` program,up to 32 processors. The shape of this graph follows the general behaviour observed in the previously shown experiments.

Overall result We now present the overall result of our project, the speedup of the final parallel implementation compared to the original sequential implementation. Measured with `GNU time`, running the `qft16.mc` benchmark, `qvm` takes 4.333 seconds; `pqvm` in sequential mode takes 1.206 seconds; `pqvm` running on 8 threads takes 0.405 seconds.

[2]With similar results for the other operators, not included for brevity.

```
Samples: 127K of event 'cycles', Event count (approx.): 66126401560
 41.38%       sigma-x  sigma-x                      [.] quantum::itbb::details::sigma_x_odd::operator()(tbb::block
 32.75%       sigma-x  sigma-x                      [.] quantum::itbb::details::sigma_x_even::operator()(tbb::bloc
  4.12%       sigma-x  libtbb.so.2                  [.] 0x0000000000016e41
  1.67%       sigma-x  [kernel.kallsyms]            [k] down_read_trylock
  1.66%          pqvm  libpthread-2.15.so (deleted) [.] 0x000000000000ed10
  1.45%       sigma-x  libc-2.17.so                 [.] __random
  1.35%       sigma-x  [kernel.kallsyms]            [k] __ticket_spin_lock
  1.06%       sigma-x  [kernel.kallsyms]            [k] up_read
  1.06%          pqvm  [kernel.kallsyms]            [k] system_call
  1.04%       sigma-x  sigma-x                      [.] main
  0.99%          pqvm  [kernel.kallsyms]            [k] system_call_after_swapgs
  0.87%       sigma-x  [kernel.kallsyms]            [k] clear_page_c
  0.73%          pqvm  [kernel.kallsyms]            [k] sysret_check
  0.62%          pqvm  pqvm                         [.] cparse_sexp
  0.47%       sigma-x  libtbbmalloc.so.2            [.] 0x00000000000049e5
  0.47%          pqvm  [kernel.kallsyms]            [k] fget_light
  0.44%       sigma-x  [kernel.kallsyms]            [k] __write_lock_failed
  0.43%       sigma-x  libc-2.17.so                 [.] __random_r
  0.37%       swapper  [kernel.kallsyms]            [k] acpi_idle_do_entry
  0.36%       sigma-x  [kernel.kallsyms]            [k] handle_mm_fault
  0.34%       sigma-x  [kernel.kallsyms]            [k] __schedule
  0.34%       sigma-x  [kernel.kallsyms]            [k] page_fault
  0.22%          pqvm  [kernel.kallsyms]            [k] sys_read
  0.21%       sigma-x  [kernel.kallsyms]            [k] system_call
  0.20%       sigma-x  [kernel.kallsyms]            [k] __mem_cgroup_commit_charge
  0.19%       sigma-x  [kernel.kallsyms]            [k] update_curr
  0.17%       sigma-x  [kernel.kallsyms]            [k] system_call_after_swapgs
  0.17%       sigma-x  [kernel.kallsyms]            [k] ktime_get_update_offsets
Press '?' for help on key bindings
```

Figure 4.7 The `perf` sampling of `X`; 74% time in the actual tasks `sigma_x_even` and `sigma_x_odd`.

```
Samples: 129K of event 'cycles', Event count (approx.): 44763885880
 77.85%  pqvm  libtbb.so.2          [.] 0x000000000001e254
 10.28%  pqvm  [kernel.kallsyms]    [k] 0xffffffff81043e6a
  2.34%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  2.30%  pqvm  libc-2.17.so         [.] __memcpy_ssse3
  2.10%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  1.22%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  1.14%  pqvm  libgcc_s.so.1        [.] __muldc3
  1.08%  pqvm  pqvm                 [.] void tbb::interface6::internal::partition_type_base<tbb::int
  0.52%  pqvm  libtbbmalloc.so.2    [.] 0x0000000000007d79
  0.38%  pqvm  libc-2.17.so         [.] __sched_yield
  0.11%  pqvm  pqvm                 [.] __muldc3@plt
  0.08%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.06%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.05%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.03%  pqvm  pqvm                 [.] memcpy@plt
  0.03%  pqvm  libc-2.17.so         [.] _int_free
  0.03%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.02%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.02%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.02%  pqvm  libm-2.17.so         [.] __hypot_finite
  0.02%  pqvm  libm-2.17.so         [.] __sin_fma4
  0.02%  pqvm  libc-2.17.so         [.] malloc
  0.02%  pqvm  pqvm                 [.] tbb::interface6::internal::signal_task::execute()
  0.02%  pqvm  libm-2.17.so         [.] __cos_fma4
  0.02%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.02%  pqvm  pqvm                 [.] tbb::interface6::internal::flag_task::execute()
  0.01%  pqvm  pqvm                 [.] tbb::interface6::internal::start_for<tbb::blocked_range<uns
  0.01%  pqvm  libpthread-2.17.so   [.] pthread_getspecific
  0.01%  pqvm  libtbb.so.2          [.] tbb::internal::allocate_continuation_proxy::allocate(unsigne
  0.01%  pqvm  libtbb.so.2          [.] tbb::internal::allocate_child_proxy::allocate(unsigned long)
  0.01%  pqvm  libtbb.so.2          [.] sched_yield@plt
  0.01%  pqvm  pqvm                 [.] tbb::interface6::internal::flag_task::~flag_task()
  0.01%  pqvm  libc-2.17.so         [.] malloc_consolidate
  0.01%  pqvm  libtbb.so.2          [.] tbb::task::note_affinity(unsigned short)
Press '?' for help on key bindings
```

Figure 4.8 The `perf` sampling of `pqvm` running `qft16`; 77% of time in the `TBB` runtime

Chapter 5

Conclusions and Future Work

The dense state vectors and efficient cache usage turn out as our main result. While our implementation proves no match for a complex machine like `serenity`, the results indicate that an implementation on smaller multicore machines (2, 4, 8 cores) can produce a reasonable speedup.

As `TBB` exhibits a high overhead, we might opt for another library in future implementations. The current memory model (large state vectors) seems unfit for NUMA machines; it might turn out better to split the quantum state across the nodes, and have each node operate mainly on its own part. However, as `TBB` hides the actual threads (and therefore the nodes and processors) in smaller grained tasks, other libraries (`ArBB` for instance) seem better suited.

As a hybrid between an interpreter and a compiler approach, we propose a deferred execution approach, grouping the execution of multiple operators into one; this would increase operational intensity, performing more on-chip work per byte of data, thus shifting the interpreter towards the roofline optimum. Furthermore, instead of storing each amplitude at an index corresponding to its basis state, we could easily store the basis state with along with it, yielding more numerical transformations and fewer data movement! Apart from the obvious gain in execution speed, we claim that such a model would work for sparse states as well, without losing unit stride access (note that we have never required random access). This claim needs more time and effort than we had available, however, it still seems within reach, given our current understanding of quantum simulation.

Finally, I would like to offer a word of thanks to my advisor Yves Vandriessche, for the many hours he has spent teaching and assisting me the past year.

Bibliography

[1] Cilk. http://supertech.csail.mit.edu/cilk/.

[2] Intel array building blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks.

[3] Nesl: A parallel programming language. http://www.cs.cmu.edu/~scandal/nesl.html.

[4] Openmp. http://www.openmp.org/.

[5] Papi: Performance application programming interface. http://icl.cs.utk.edu/papi/.

[6] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[7] Z-curve.svg. http://en.wikipedia.org/wiki/File:Z-curve.svg.

[8] Quantum theory, the church-turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London*, A 400, pages 97–117, 1985.

[9] Michael D. Adams and David S. Wise. Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 41–50, New York, NY, USA, 2006. ACM.

[10] R. Asanovic, R. Bodik, B. C. Catanzaro, P. Husband, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[11] S. Bettelli, Serafini L., and T. Calarco. Toward an architecture for quantum programming. *CoRR*, cs.PL/0103009, 2001.

[12] Aydın Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and

matrix-transpose-vector multiplication using compressed sparse blocks. In *IN SPAA*, pages 233–244, 2009.

[13] Björn Butscher and Hendrik Weimer. libquantum. `http://libquantum.de`, Februari 2013.

[14] S. Chatterjee and S. Sen. Cache-efficient matrix transposition.

[15] Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *Journal of the ACM*, 54(2), April 2007.

[16] U. Drepper. What every programmer should know about memory. November 2007.

[17] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7), 1982.

[18] Richard P. Feynman. Quantum mechanical computers. *Optics News*, 2:11–20, 1986.

[19] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[20] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. Technical report, 1991.

[21] Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 116–125, New York, NY, USA, 2007. ACM.

[22] J. Grattage. *QML: A functional quantum programming language*. PhD thesis, The University of Nottingham, 2006.

[23] Intel. Intel threading building blocks reference manual. `http://threadingbuildingblocks.org/docs/help/index.htm`, Februari 2013.

[24] Philip Kaye, Raymond Laflamme, and Michele Mosca. *An introduction to quantum computing*. Oxford University Press, Oxford, England, 2010.

[25] Donald E. Knuth, Oren Patashnik, and Ronald L. Graham. *Concrete Mathematics*. Addison-Wesley, Massachusetts, 1994.

[26] Piyush Kumar. Cache oblivious algorithms. In *Algorithms for Memory Hierarchies, LNCS 2625*, pages 193–212. Springer-Verlag, 2003.

[27] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences, University of California at Berkeley, January 2006.

[28] Norm Matloff. Programming on parallel machines. `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`.

[29] Panagiotis D. Michailidis and Konstantinos G. Margaritis. Performance study of matrix computations using multi-core programming tools. In *Proceedings of the Fifth Balkan Conference in Informatics*, BCI '12, pages 186–192, New York, NY, USA, 2012. ACM.

[30] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, 2000.

[31] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, October 2004.

[32] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachisetts Institute of Technology, june 1999.

[33] P. Selinger and B. Valiron. Quantum lambda calculus. *Semantic techniques in quantum computation*, pages 135–172, 2010.

[34] Viral Shah and John R. Gilbert. Sparse matrices in matlab*p: Design and implementation. In *In HiPC*, pages 144–155. Springer, 2004.

[35] Larry Snyder and Calvin Lin. *Principles of parallel Programming.* Pearson International, United States, 2008.

[36] Gerald J. Sussman and Jack Wisdom. *Structure and Interpretation of Classical Mechanics.* MIT Press, London, England, 2001.

[37] Herb Sutter. Welcome to the jungle. `http://herbsutter.com/welcome-to-the-jungle/`.

[38] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *D. Dobb's Journal*, 30(2), 2005.

[39] Herb Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7), September 2005.

[40] Dimitrios Tsifakis, Alistair P. Rendell, and Peter E. Strazdins. Cache oblivious matrix transposition: Simulation and experiment. In *In Lecture Notes in Computer Science 3037: Computational Science, ICCS*, pages 17–25.

[41] Y. Vandriessche. Serenity presentation. 2013.

[42] Yves Vandriessche. *A foundation for quantum programming and its higly-parallel virtual execution.* PhD thesis, Vrije Universiteit Brussel, 2012.

[43] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[44] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[45] A. N. Yzelman and Rob H. Bisseling. A cache-oblivious sparse matrix-vector multiplication scheme based on the hilbert curve. In *Progress in Industrial Mathematics at ECMI 2010*, volume 17 of *Mathematics in Industry*. 2010.