A highly-parallel formulation of quantum computing simulation through fine-grained dataflow.

Yves Vandriessche, Ellie D'Hondt, Tom Van Cutsem and Theo D'Hondt

Software Languages Lab
Vrije Universiteit Brussel
Email: {yvdriess, eldhondt, tjdhondt, tvcutsem}@vub.ac.be

***Abstract*—Quantum Computing lies at the frontier of computing, offering a radically different and unconventional model of computation. In the absence of practical quantum computers today, we must simulate their execution. This creates a performance problem, as quantum computing simulation is very costly. However, quantum computing simulation does contain an abundance of parallelism. The research question becomes: how to expose this parallelism? Although it is easy to show the problem inherently contains a large amount of parallelism, the type of parallelism is non-trivial to exploit in a scalable way using current mainstream parallelization techniques. This paper presents the formulation a virtual machine for quantum computing as a fine-grained dataflow schema, which aims to expose and exploit the abundance of parallelism in a way that avoids the observed scalability issues. We present the formal mapping from the linear algebra description of elementary quantum operations to dataflow graphs, analyze its theoretical parallel characteristics and present experimental performance results of a prototype implementation.**

I. Introduction

Quantum Computing started as a mathematics and physics curiosity, but has gradually shown to have interesting applications. This potential has spurred researchers to work towards actual physical quantum computers. The current state of this field is similar to the era before the electronic computer; the underlying principles are known, but the practical implementation techniques are still being developed. As computer scientists, it is interesting to consider how one would go about programming such computers. Early programming languages have been developed after hardware came along, discovering along the way what works and what does not. Quantum programming attempts to develop ways of working with a quantum computer, before these become available, guided by decades of experience building software systems and languages.

Building a practical quantum programming framework today means that: a) execution needs to be simulated and b) there is a large choice of possible hardware and software approaches. The problem with the first point is that this simulation grows in exponential time and space complexity, creating a performance barrier. The second point means that the programming framework needs to deal with heterogeneity on both the higher and lower levels of abstractions.

We approach the variation problem (b) using a Virtual Machine (VM) abstraction. Virtual machines for programming languages have become a staple technique in software language engineering to deal with variability on the level of the language it needs to support and on the level of the execution machinery it needs to run on. Facing similar variability problems in the domain of quantum programming languages, it was a logical step to apply the same virtual machine technique. In order to do this, it is important to choose an appropriate Instruction Set Architecture (ISA). In earlier work [1] we presented, to a quantum computing audience, a VM design and prototype based on the Measurement Calculus model of quantum computing, called the Quantum Virtual Machine (QVM). Here, we report on recent work which seeks to express the execution of such a VM as a highly-parallel computation.

The renewed interest today in parallel computing is understandable, considering we find ourselves in an era where sequential performance of computing hardware is stagnating, whereas parallel hardware performance continues to scale; a more performant sequential approach today will likely be outpaced tomorrow by the previously slower parallel approach. For the specific context of quantum computing simulation, we find that it is an interesting case for trying out novel parallel computing techniques. On paper, as we will show, there is an abundance of exploitable parallelism. In practice however, we observe[1] that mainstream parallel hardware architectures are ill-equipped to exploit the involved type of parallelism. Concretely, arithmetic intensity is low, the access pattern is global and hard to prefetch. For such workloads, common in-hardware techniques to reduce memory latency become ineffective or even counterproductive. In order to scale, a different paradigm of parallel computing is needed [2].

We have a two-pronged goal for this presented work. First, theoretically formulate quantum computing simulation as to fully express all available parallelism. Second, express the parallel computation in practice in a way that avoids the common issues encountered in mainstream parallel software and hardware approaches. For both goals, we make use of a Dataflow execution model: a formal dataflow model schema

[1]We observed this in an earlier experiments, in which we applied common parallelization techniques to our existing efficient sequential QVM implementation. This artifact has been made publicly available on http://github.com/yvdriess/qvm/. The experiment can be reproduced by comparing the code branches `dense` and `openmp`.

for the former and a dataflow execution framework for the latter.

A. Quantum Computing

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 'quantum gates' 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 [3], linear logic [4], lambda calculus [5], functional languages [6], category theory [7, 8], 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.

For our Quantum Virtual Machine, we base ourselves on the Measurement Calculus (MC) by Danos et al. [9]. This formal model brings together several properties we desire. First, it introduces a compositional and modular abstraction for quantum programs. Second, the Measurement Calculus is a formalization of a low-level model of quantum computations that is physically realistic: measurement-based quantum computing. Physicists have found that actual implementations of the circuit model run into fundamental scalability issues. Measurement-based quantum computing is one of the newer generations of approaches to quantum computing that use the insights gained from years of experimental research to avoid these scaling issues. Third, the MC elegantly separates quantum and classical state, allowing us to draw clear boundaries on what work can be performed on a classical system and which computations are inherently quantum. Lastly, the MC uses a small but universal set of operations: with it, any quantum operation can be expressed. The operations themselves are relatively simple, which simplifies the implementor's task. We have developed this QVM within the broader context of a quantum programming framework, which we have presented in prior work [1, 10]. The work here focuses specifically on the parallel formulation and execution of the QVM's ISA.

B. Dataflow model

We have chosen the *dataflow model of computation* as foundation for the parallel execution of our QVM. Quantum Computing is today mainly expressed in terms of linear algebra, such linear algebra problems have already been shown before to map well to dataflow models and hardware architectures [11, 12]. But more specifically to our case, dataflow exhibits certain properties that we desire for highly-parallel QC simulation: it is *fine-grained*, *data-oriented*, highly *asynchronous* (viz. local rather than global synchronization), *implicitly parallel* and easily *analyzable*. We assume here that the reader has a minimal familiarity formal dataflow models and dataflow of parallel execution. For a more thorough treatment of dataflow models, architectures and languages we refer to the surveys by Treleaven et al. [13], Johnston et al. [14] and Veen [15], and the books by Sharp [16] and Almasi and Gottlieb [17]. The formalization of our problem as a dataflow computation is expressed in a relatively simple, abstract and static dataflow model; similar to the Petri net-style formalization of dataflow by Kavi et al. [18].

We keep here to an abstract model formulation both for simplicity and to avoid having to commit to any specific dataflow computing platform. Recent research heavily favors hybrid dataflow/von Neumann architectures that strike a balance between the elegance of dataflow execution and some of the perceived inefficiencies and barriers to adoption of pure fine-grained dataflow. Some recent hybrid dataflow research architectures are the TRIPS [19], WaveScalar [20] and Apple-CORE [21] architectures. Each type of hybrid or macro dataflow model involves a different formulation of the same problem. A full survey and classification of hybrid dataflow architectures can be found in Yazdanpanah et al. [22].

In this text, we will start from a multilinear algebra formulation of the involved quantum operations. The expression of each QVM operation as a Kronecker product, a matrix- and vector-specific instance of the tensor product, allows us to establish a simple transformation (expansion) into a fine-grained dataflow model.

II. Dataflow formulation of the QVM

A. The Quantum Virtual Machine

The QVM presented here has to be considered in the context of a larger quantum programming framework. To give a brief overview: the framework's application layer enables its users to compose complex quantum programs in a structured way and integrate it into a traditional application, a compilation layer deals with performing the logical composition of the modular program parts and an assembly step will ultimately produce a sequence of concrete commands. Note that we will be using the term *commands* here as interchangeable with *instructions*. The command sequence thus forms a 'quantum assembly language' and is the program input for our quantum virtual machine, which functions as the execution layer for the framework. The operational semantics of this quantum assembly language are based on the Measurement Calculus by Danos et al. [9]. Each command of an input command sequence is executed one after the other, targeting only one or two *qubits*. A *qubit* is the smallest non-trivial quantum state, each qubit is identified in commands by a unique integer. Just

as the bit forms the elementary data element in a classical computer, so does the qubit for a quantum computer.

To get an impression of a quantum program as a command sequence, we give the following example. The command sequence performing the teleportation of the *qubit* identified by the integer 1 into the qubit 3 is expressed in MC's original mathematical notation as

$$X_3^{s_2} Z_3^{s_1} M_2^{s_1} M_1 E_{2,3} E_{1,2} \quad .$$

Read from right to left, analogous to matrix multiplication, the above contains a sequence of applications of the commands: E (Entanglement), M (Measurement), Z and X (Pauli-Z and -X operations). Subscripts denote which qubits the command acts upon. Superscripts denote predicated execution: $Z_3^{s_1}$ will be executed or not depending on the outcome of the measurement of qubit 1. As you can already observe, only the entanglement command acts on two qubits, the rest on only one. To explain the higher-level meaning of the above program would involve more than we have space available here, due to the radically different nature of the quantum computing paradigm. For our purposes, it is sufficient to only describe the small-step semantics, the effect of each individual command (E, M, Z and X) on the simulated quantum computational state. Each such command corresponds to a specific *quantum operation*, the effect of which we simulate[2] using a classical computer.

In this section, we will expose the effect of each individual command on the computational state during its simulated execution. By doing this, we focus on explaining only the concepts and semantics required to understand our core contribution: the mapping of the involved quantum operation to dataflow. We will start with the simplest formulation, one were each operation is applied on an atomic state. As we will see, the dataflow graph for each such atomic operation can be used to construct the dataflow graph for all involved operations.

B. Single qubit formulation

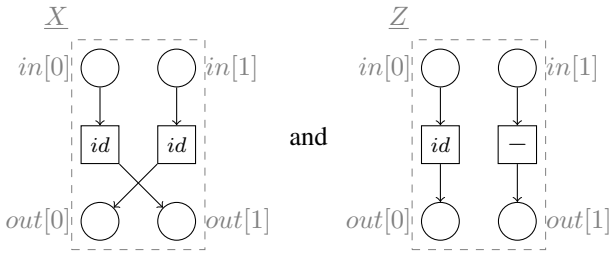
Each *command* of the Measurement Calculus manipulates the entire quantum state, even if logically only a single logical entity (a qubit) is manipulated. The quantum state can be represented as a vector of complex elements, which are for historical reasons called amplitudes. The simplest formulation of each involved quantum operation thus involves a computational state representing only a single qubit: a vector of two elements. In the case of the Entanglement command (E), the smallest applicable state is represents two qubits: a vector of four elements.

The operators realizing the X and Z commands on a single qubit are the following simple matrix operators.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{1}$$

[2]To avoid confusion, note that when we use the term quantum computing simulation, this has a different meaning than the term quantum simulation in for example molecular chemistry. Our type of simulation is more analogous to gate-level simulation of a processor chip, rather than the physical simulation of individual electrons.

Acting on a two-element vector, one can easily see that X swaps the position of the two elements and Z flips the sign of the second element. This behavior corresponds to the following simple dataflow graphs.

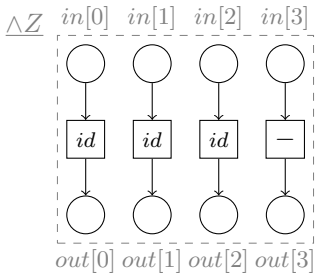


The operation nodes containing id are dummy nodes performing the identity operation and are used here for the sake of uniformity. The nodes are labeled here for clarity, conceptually linking these back to vector elements.

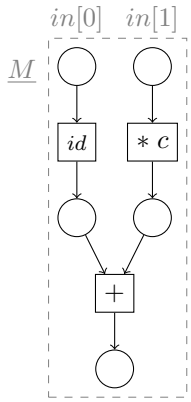
The quantum operation realizing the entanglement command E has a similarly simple matrix operator, called the *controlled-Z* or $\wedge Z$ operator.

$$\wedge Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{2}$$

Again, this corresponds to the following simple dataflow graph.



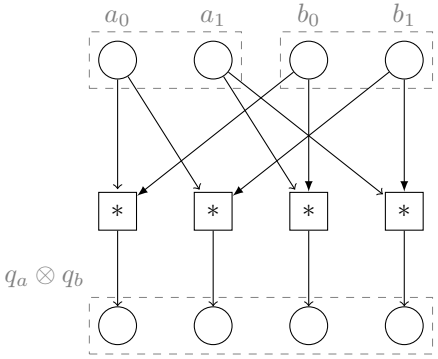
The measurement command M for a single-qubit state is normally a trivial corner case, as it completely destroys the input state. Nevertheless, it is useful to first consider the measurement of a single qubit state. This trivial case will leave a residual scalar, instead of a quantum state. In the Measurement Calculus, the M operation takes a *measurement angle* parameter α. The measurement of a single-qubit state vector $[a_0\ a_1]^T$ degenerates into the inner product: $\begin{bmatrix} 1 & e^{-i\alpha} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$. Taking the complex scalar $c = e^{-i\alpha}$, a simple dataflow graph can again be constructed as shown below.



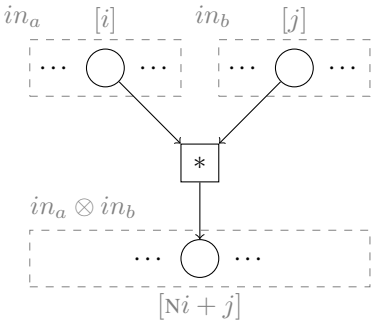
One essential operation has not been discussed so far, which forms an implicit part of the QVM's realization of the E command: the *Kronecker product*. In the formal semantics of the Measurement Calculus, on which the QVM is based, the quantum state is prepared beforehand in a state representation which combines all necessary qubits. A combined representation of two states is obtained by combining all amplitudes of one state with those of the other, thus creating a combinatorial explosion. In order to be practical, our QVM semantics amended the original MC operational semantics to combine two quantum states only when absolutely necessary, lazy state combination as it were. Only during the E command can such combination take place, as $\wedge Z$ is the only operator that acts on two qubits, and thus requires them to be in a combined state. In terms of linear algebra, the combination of two quantum state vectors q_a and q_b is achieved using the tensor product, specifically, the *Kronecker product*: $q_a \otimes q_b$. For instance, combining two single-qubit states results in the amplitude vector

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \otimes \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} a_0 \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \\ a_1 \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{bmatrix} \quad . \tag{3}$$

The above example results in the dataflow graph shown below.



This can be generalized for larger states by repeating the following graph pattern in which N is the size of the vector in_b:



C. Multi-qubit formulation

The first step is to consider the tensor product formulation of single-qubit operations. As quantum states get combined, the single-qubit operations need to reformulated as to act on this combination. This is realized by combining the 2×2 identity operator $I = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ with the single-qubit operator. The command X thus translates to some matrix operator

$$I^{\otimes m} \otimes X \otimes I^{\otimes n} \quad , \tag{4}$$

when applied on a quantum state representing $n+m+1$ qubits. Position within this tensor product is of prime importance: X needs to be in the same *tensor position* as the qubit it is targeting. We define the tensor position of a qubit as the position it was combined in. For example, take $q_1, q_2, \ldots$ to be single qubit states for qubits named $1, 2, \ldots$. Applying the MC command X_3, targeting qubit 3, to the quantum state combination $q_1 \otimes q_3 \otimes q_5$ is realized by the matrix operator $I \otimes X \otimes I$. In practice, we use a intermediate representation which associates with each quantum state a sequence of qubit names, in order to track in which position each qubit can be found.

Matrix operators with such a Kronecker product formulation have along the diagonal a certain repeating block structure [23]. The smaller the n in eq. (4), the smaller the repeating block's size and the more it is repeated. When X is completely at the last tensor position ($n = 0$), one obtains a perfectly data-parallel formulation of the operator [26]. That is, applying the $I^m \otimes X$ operator on a vector state is equivalent to applying m-times the two-by-two X matrix operator *in parallel* on contiguous two-element sub-vectors of the input vector state. For this reason, we will refer to this preferred tensor product position as the *parallel position*. As a parallel position operator is equivalent to the repeated parallel application of its atomic one- or two-qubit version, its dataflow graph is thus a repetition or concatenation of the atomic operator's graph.

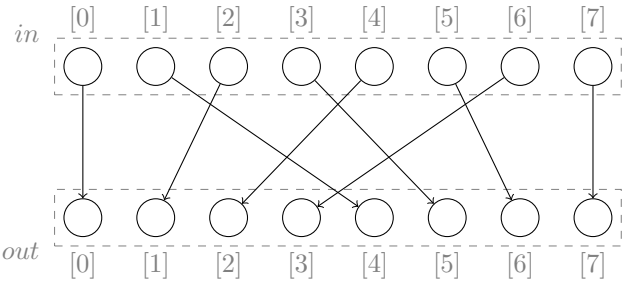
The key insight here, is that one can change the tensor positions of qubits in the quantum state vector. By itself, the Kronecker product is not commutative. However, there is a class of permutations that does permute any given Kronecker product [23]. There is thus a permutation matrix P_n such that

$$I^{\otimes m} \otimes X \otimes I^{\otimes n} \equiv P_n^{-1} \left(I^{\otimes m+n} \otimes X \right) P_n \quad .$$

Put in other terms, there is an operation which realizes a cyclical shift of qubit positions. The dataflow graph construction of this position changing permutation can be described as follows. Label each input state node from $in[0]$ to $in[\mathrm{N}-1]$ and do the same to for output nodes ($out[0]$ to $out[\mathrm{N}-1]$). Given the position changing operation $\mathbb{P}_\mathbf{s}$ which cyclically shifts all qubits s positions to the left and $\mathrm{S} = 2^s$, then connect each input node labeled $in[i]$ to an output node labeled $out[p_{\mathrm{S,N/S}}(i)]$, where the permutation function $p_{m,n}$ is defined as:

$$p_{m,n}(\mathrm{i}) = m\,(i \bmod n) + \left\lfloor \frac{\mathrm{i}}{n} \right\rfloor \ . \tag{5}$$

For instance, a $\mathbb{P}_\mathbf{1}$ position changing operation acting on a quantum state vector of eight elements is thus transformed into the following fine grained graph:



It is useful to note that this permutation function describes the data access pattern of imperative program implementations of the same operations. In such cases, it can be useful to perform the actual data movements, for instance to enable vectorization optimizations or to improve data locality. Indeed, the same tensor-commuting permutation can be observed in related work involving similar multilinear algebra, often the automatic parallelizing and optimization of Fourier transforms [25, 26]. In our case, this permutation is used to construct the static dataflow graph, as we have done here. But, the permutation can also be used in dynamic dataflow configurations, where the permutation function in (5) can be used to compute the destination of any given output element at runtime.

In conclusion, bringing everything together, a complete fine-grained dataflow graph can be constructed for any given sequence of commands. For the sake of compactness, this text has glossed over several details and mathematical formulations. A more full and rigorous treatment of the subject can currently be found in the doctoral dissertation of Vandriessche [10]. To clarify with a larger example, we have juxtaposed an example positional coarse graph with its fine grained graph in Figure 1.

III. Implementation and analysis

At the core of our contribution lies the claim that a command sequence expressed in our fine-grained dataflow graph model exposes a large amount of parallelism. Supporting this claim solely using an empirical study is hard, as real-world performance is very sensitive to implementation issues and the target parallel architecture. We therefore supplement the empirical results of our implementation, presented below, with a more theoretical quantitative analysis of the parallelism exposed by our fine-grained dataflow model.

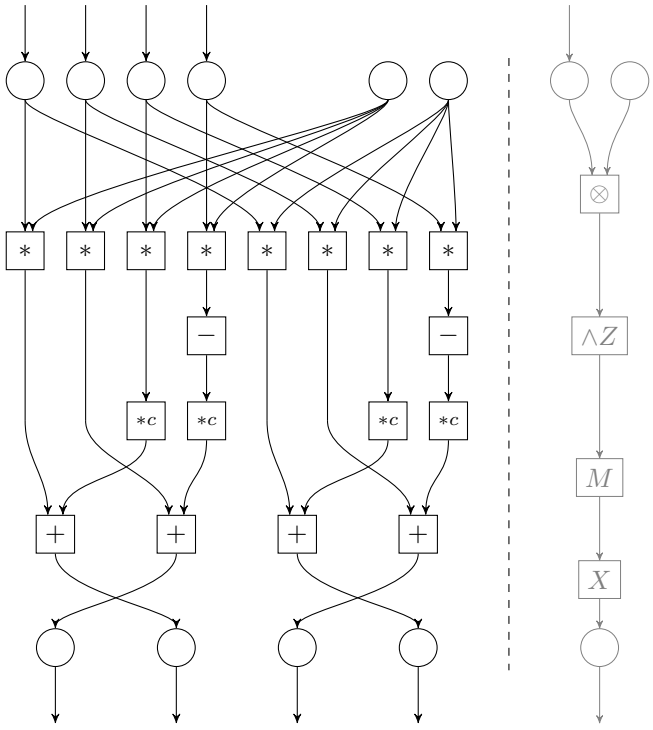


Figure 1. Fine-grained dataflow representation (left) of the command sequence $X_2^{s_1} M_1 E_{1,2}$ implementing the Hadamard transform, acting on a two-qubit input state. The schema on the right is a visual lead and a compact representation of the involved operators and quantum states.

A. Average parallelism

Average parallelism is a machine-independent metric, observable from the static program description that has in practice shown a high correlation with parallel speedup. To our knowledge, no other such metric combines simplicity with a strong predictive power for parallel performance [27, 28]. Indeed, one of the popularizing factors of the Cilk [29] 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. We thus use average parallelism as a quantitative measure for exploitable parallelism. This gives a basis for comparison for future work related to ours, but also reveals something about the parallel nature of quantum computations.

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

$$\pi_{av} = \frac{S_1}{S_\infty} \tag{6}$$

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. The average parallelism of

a program is the maximal parallel speedup one that can theoretically achieve. It is no coincidence that this relates to Amdahl's law: the critical path relates to the sequential part of a program, the average parallelism to the parallel part.

In parallel programs structured using DAGs, as is our case, 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. Rather than doing this by hand, we automated this by integrating it in our parallel compiler. The resulting metrics for a few interesting quantum programs are listed in Table I and plotted in Figure 2. 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 command sequence length and qubit state width. We use $QFT(n)$ to refer to the n-qubit QFT program. The number of QVM commands ($\#ops$) metric is included to compare program sizes. The π_{av} of the Quantum Fourier Transform can be observed to double roughly for each additional qubit, following the same increase in the 'width' of the quantum state. The program to create the W_3 state [30, 31] has only 40 commands, in contrast with $QFT(16)$'s 3288, but affords a comparable measure of parallelism. This is due to the nature of the quantum algorithm, but also because the W_3 program puts all entanglement commands in front of the sequence, creating a wide dataflow graph. The critical path length in every case reflects the number of commands in the given sequence, it is not significantly higher because of two reasons. First, the dataflow graph exposes some natural parallel execution of some commands. And second, the X operation does not perform work in a dataflow computing setting. Put in other terms; following the edges, one can 'compile away' the operation entirely.

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

Table I
WORK, CRITICAL PATH AND AVERAGE PARALLELISM METRICS OF THE DATAFLOW GRAPHS REALIZING VARIOUS QUANTUM COMPUTING BENCHMARK PROGRAMS.

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, the increasing critical path length reflects its highly entangled nature.

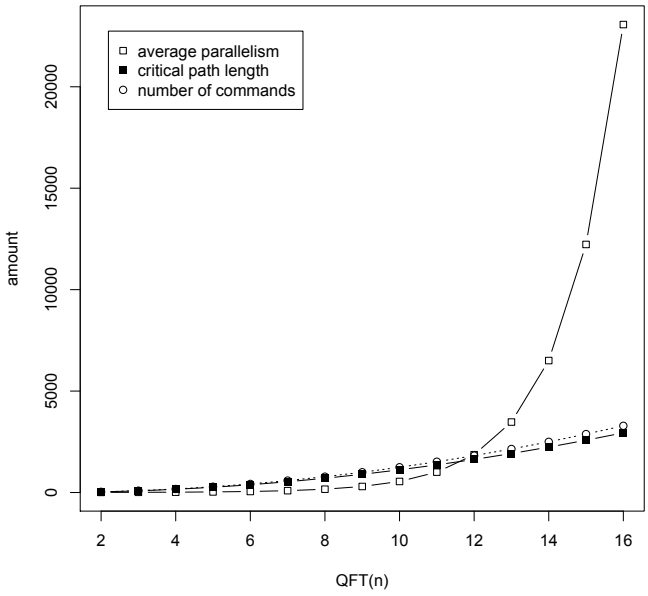


Figure 2. Average parallelism data points for different program sizes, indicating that the exponential increase of work goes together with an exponential increase in exploitable parallel performance.

B. Experimental Validation

The implementation used for the experimental validation of this work is a proof of concept, demonstrating by construction that the QVM can indeed be realized as a fine-grained dataflow computation, and that the described theoretical dataflow graph yields correct results. This is not self-evident, the same theoretical analysis can be used for a more traditional imperative-style data-parallel implementation, as in parallel QC related work [32, 33]. 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 formalization is indeed feasible and exposes the promised parallelism. Therefore, our implementation approach follows the fine-grained dataflow semantics as closely as possible.

The implementation artifact is structured as a stratified compiler, written in Common Lisp, using multiple intermediate model transformations; staying close to the research we are validating while providing enough hooks in the implementation for researching optimizations and alternative execution targets. For this initial experimental work, we choose Intel Concurrent Collection (CnC) [36] as parallel software platform. We thus make two main compromises in this prototype implementation. The first is in the choice of CnC as dataflow computation platform. The second is the implementation of the prototype as compiler rather than runtime virtual machine. Ideally, we want to directly emit instructions for a target

dataflow platform that can take advantage of the fine-grained dataflow formulation to perform the necessary dataflow optimizations. However, to our knowledge, there are no such off the shelf dataflow computing solutions. We thus compromise by choosing from the set of current and practical software frameworks that provides us the desired dataflow semantics on top of commodity multicore processors; frameworks such as Qthreads [34], Data-Driven Tasks [35] or Intel's CnC [36]. Our choice of CnC represents a trade-off between being able to achieve fine-grained dataflow execution semantics with an off the shelve parallel hardware and programming framework. Moreover, CnC allows for different back-ends [37, 38], which offers us prospects on testing our implementation under different execution strategies and hardware. The second compromise is our use of a compiler design for our prototype, instead of a runtime virtual machine as in the formal case and our reference sequential *qvm* implementation. This is out of technical considerations – a code generation step is required – and out of a formal consideration to stay close to the formal model transformations. Our prototype does remains identical in interface, function and results compared to the formal QVM and our reference sequential *qvm* implementation.

The theoretical analysis above 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 implementation exposes and exploits inherent parallelism by using speedup as an 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 the popular *libquantum* library to offer a fast sequential implementation as baseline comparison. Furthermore we demonstrate by introducing a coarsening optimization that that parallel scheduling overhead is the dominant factor in this proof of concept implementation.

We use the experimental results to support two statements.

- *Statement 1:* We automatically expose and exploit parallelism, with our proof of concept implementation already demonstrating good parallel performance.
- *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.

Statement 1 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; our artifact takes the same program input as `qvm` and automatically produces parallel program code based on the internal dataflow graph representation. However, we still need to show with experimental results that parallelism is exploited in practice. This is achieved by measuring the *parallel speedup* metric as for parallel performance. Speedup $S_n := \frac{T_1}{T_n}$ 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.

a) Experimental setup: The test-bed computer system we have used in the experiments uses two 2.26 Ghz quad-core Intel Xeon multicore processors in a uniform memory architecture configuration, allowing us to scale up to eight effective hardware threads. Processor cache sizes are 256KB (L2) and 8MB (L3). As benchmark program, we use the $QFT(n)$ quantum algorithm, as used in the theoretical analysis above. For demonstrating strong scaling we use $QFT(16)$, as it constitutes a high enough workload in both sequential (`qvm`) and parallel cases, with runtimes in the order of seconds. Each experiment consists of a number samples gathered over multiple execution runs, in order to offer more statistically valid performance measurements [39]. The multiple 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 (central dot) and probability density (violin shape).

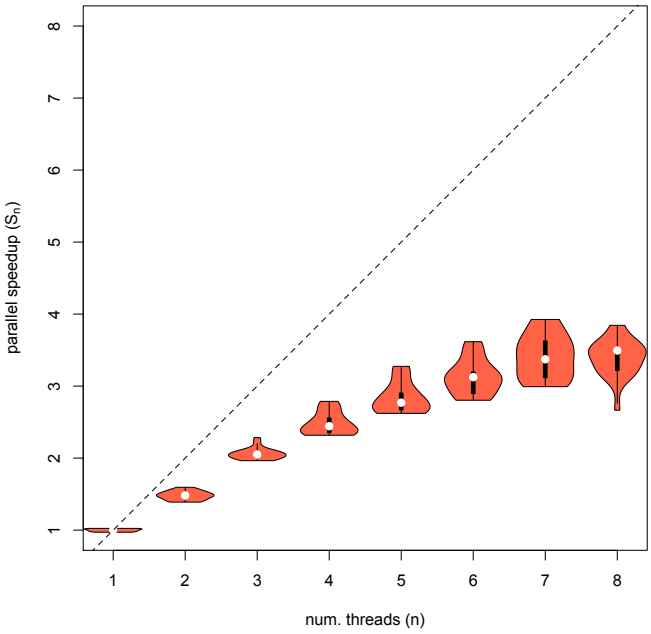


Figure 3. Parallel Speedup S_n values of multiple execution runs for the $QFT(16)$ command sequence, using the un-optimized fine-grained parallel implementation. The diagonal line represents ideal linear speedup.

b) Results: The speedup graph in Figure 3 shows a typical cannonball trajectory curve, where we see a speedup improvement whose increase gradually tapers off. The main factors contributing to this behavior is the task coarseness;

each fine-grained task performs only a small amount of computation compared to the scheduling, task switching and synchronization overhead. This first naive experiment does not take into account the ideal parallel task coarseness of the target hardware platform. We demonstrate that coarseness is indeed a dominant factor by showing the effect of a coarsening optimization below.

To test the amount of overhead, we measure the effect optimizations have on speedup. 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 optimizations in the form of *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 our artifact. A CnC *step tuner*, 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. In Figure 4 we superimpose the earlier reported

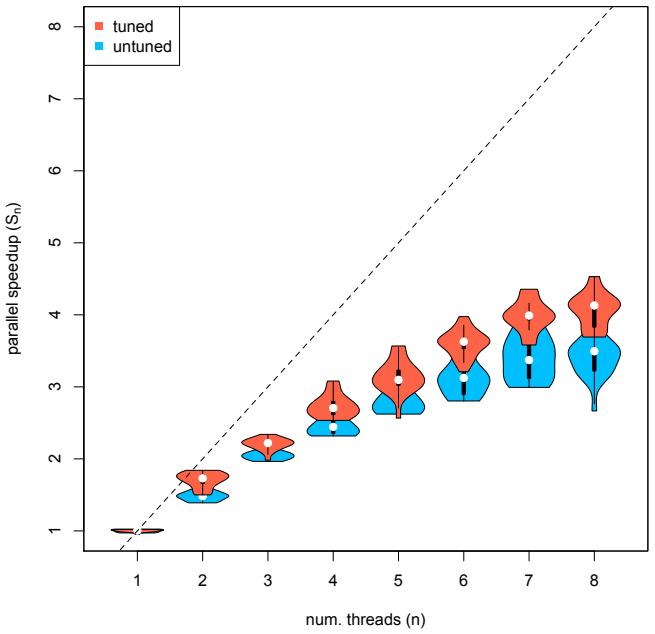


Figure 4. Effect of introducing CnC tuner optimizations, reducing the overhead of individual parallel step instances.

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 shift of the parallel scaling graph towards linear speedup. This is assumed to be an indicator that parallel overhead is dominated by the number of scheduled step instances, rather than the overhead for each individual one. This assumption is tested in the following by introducing a coarsening optimization, which reduces the overall number of step instances. The coarsening optimization is implemented at the level of our compiler artifact. By implementing parts of the fine-grained graph as a single CnC step, the amount of parallel work is decreased in favor of higher sequential work. 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 5, in which we added the average sequential runtime speed of the sequential `qvm` implementation as reference point. This graphic clearly

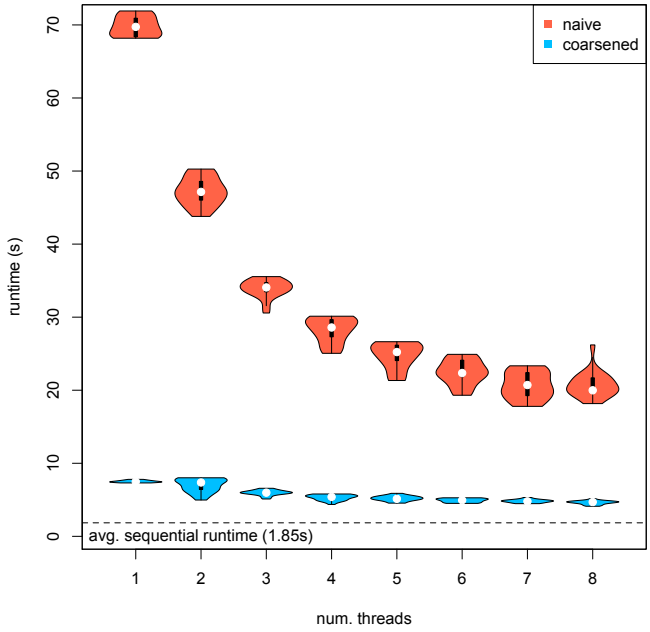


Figure 5. Comparing the absolute runtime speeds of the original unoptimized parallel implementation with the tuned and coarsened optimized version. Both execute the same $QFT(16)$ command sequence.

shows a large absolute performance win from introducing a small coarsening optimization. This indicates parallel overhead from the fine-grained granularity to be the dominant performance factor in our proof of concept implementation. This overhead does not only originate from CnC, which is designed around coarser steps, but this is also an issue of the multicore execution platform which still prefers large blocks of sequential workload.

IV. Conclusions

We have formulated the execution of simulated measurement-based quantum computing programs as a

highly-parallel dataflow computation, both to analyze its parallel potential and to express the problem independently from underlying parallel hardware. Our approach starts from a tensor product formulation of the involved linear operators to construct a fine-grained dataflow graph model.

We implemented the required transformation steps by building a dataflow compiler artifact. 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) [40]. Because of CnC's dataflow-like programming model, we were able to achieve true fine-grained dataflow execution semantics with only few compromises: 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.

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. We show that this difference in absolute performance is indeed due to inefficiencies: experiments that introduce simple optimizations demonstrate dramatic performance improvements.

With this research we seek to inspire further work that uses dataflow as model for parallel computing. We have used it effectively in the context of quantum computing simulation, both in its capacities as a simple and elegant conceptual model, but also as a practical way of executing fine-grained parallel computations. In addition, quantum computing simulation has also proven to be an effective parallel computing case, exhibiting a vast amount of parallelism that is hard to exploit with traditional parallel software techniques. For the future, we look forward to using the parallel QVM as a test bed for general-purpose fine-grained dataflow execution environments.

References

[1] E. D'Hondt and Y. Vandriessche, "Distributed quantum programming," *Natural Computing*, vol. 10, no. 4, pp. 1313–1343, Dec. 2011.

[2] Arvind and R. A. Iannucci, "A critique of multiprocessing von Neumann style," in *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*. ACM, Jun. 1983.

[3] S. Gay and R. Nagarajan, "Types and typechecking for communicating quantum processes," *Mathematical Structures in Computer Science*, vol. 16, no. 03, pp. 375–406, 2006.

[4] J.-Y. Girard, "Between logic and quantic: a tract," in *Linear logic in computer science*. Cambridge: Cambridge Univ. Press, 2004, pp. 346–381.

[5] P. Selinger and B. Valiron, "Quantum lambda calculus," in *Semantic techniques in quantum computation*. Cambridge: Cambridge Univ. Press, 2010, pp. 135–172.

[6] P. Selinger, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, no. 04, pp. 527–586, 2004.

[7] 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*, 2004, pp. 415–425.

[8] R. Duncan, "A graphical approach to measurement-based quantum computing," *Compositional methods in Physics and Linguistics*, vol. quant-ph, 2012.

[9] V. Danos, E. Kashefi, and P. Panangaden, "The Measurement Calculus," *Journal of the ACM (JACM*, vol. 54, no. 2, p. 8, 2007.

[10] Y. Vandriessche, "*A foundation for quantum programming and its highly-parallel virtual execution*," Ph.D. dissertation, Vrije Universiteit Brussel, Software Languages Lab, Nov. 2012.

[11] J. 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*, pp. 45–72, 2001.

[12] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2002.

[13] P. Treleaven, D. Brownbridge, and R. Hopkins, "Data-driven and demand-driven computer architecture," *ACM Computing Surveys (CSUR)*, vol. 14, no. 1, pp. 93–143, 1982.

[14] W. Johnston, J. Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[15] A. Veen, "Dataflow machine architecture," *ACM Computing Surveys (CSUR)*, vol. 18, no. 4, pp. 365–396, 1986.

[16] J. A. Sharp, *Data flow computing: theory and practice*. Norwood, NJ, USA: Ablex Publishing Corp., 1992.

[17] G. S. Almasi and A. Gottlieb, *Highly parallel computing*. Benjamin-Cummings Pub Co, 1994.

[18] K. Kavi, B. Buckles, and U. Bhat, "A Formal Definition of Data Flow Graph Models," *Computers, IEEE Transactions on*, vol. C-35, no. 11, pp. 940–948, 1986.

[19] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder, "Scaling to the End of Silicon with EDGE Architectures," *Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[20] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The WaveScalar architecture," *ACM Transactions on Computer Systems*, vol. 25, no. 2, pp. 4–es, May 2007.

[21] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, I. Uddin, and C. Jesshope, "Apple-CORE: harnessing general-purpose many-cores with hardware concurrency management," *Embedded Hardware Design (Microprocessors and Microsystems)(to appear)*, 2013.

[22] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid Dataflow/Von-Neumann Architectures," *Parallel and Distributed Systems, IEEE Transactions on*, no. 99, p. 1, 2013.

[23] H. Henderson and S. Searle, "The vec-permutation matrix, the vec operator and Kronecker products: a review," *Linear and Multilinear Algebra*, vol. 9, no. 4, pp. 271–288, 1981.

[24] R. Johnson and J. R. Johnson, "Programming schemata for tensor products," *Preprint*, 2007.

[25] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, and Y. Voronenko, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[26] R. Johnson, C. Huang, and J. Johnson, "Multilinear algebra and parallel programming," *The Journal of Supercomputing*, vol. 5, no. 2, pp. 189–217, 1991.

[27] J. Gurd, C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[28] D. Ghosal and L. Bhuyan, "Performance evaluation of a dataflow architecture," *Computers, IEEE Transactions on*, vol. 39, no. 5, pp. 615–627, 1990.

[29] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. 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, Aug. 1995.

[30] W. Dür, G. Vidal, and J. I. Cirac, "Three qubits can be entangled in two inequivalent ways," *Physical Review A (Atomic*, vol. 62, no. 6, p. 62314, Dec. 2000.

[31] E. D'Hondt, "Distributed quantum computation: a measurement-based approach," Ph.D. dissertation, Vrije Universiteit Brussel, 2005.

[32] K. de Raedt, K. Michielsen, H. de Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito, "Massively parallel quantum computer simulator," *Computer Physics Communications*, vol. 176, no. 2, pp. 121–136, Jan. 2007.

[33] K. M. Obenland and A. M. Despain, "A Parallel Quantum Computer Simulator," *arXiv.org*, p. 4039, Apr. 1998.

[34] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," *IEEE International Symposium on Parallel & Distributed Processing. Proceedings*, pp. 1–8, Apr. 2008.

[35] S. Tasirlar and V. Sarkar, "Data-Driven Tasks and Their Implementation," in *ICPP '11: Proceedings of the 2011 International Conference on Parallel Processing*. IEEE Computer Society, Sep. 2011.

[36] 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*, vol. 18, no. 3, pp. 203–217, 2010.

[37] A. Sbîrlea, Y. Zou, Z. Budimlic, and J. Cong, "Mapping a Data-Flow Programming Model onto Heterogeneous Platforms," *cs.rice.edu*, 2012.

[38] 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.

[39] A. Georges, D. Buytaert, and L. 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, Oct. 2007.

[40] K. Knobe, "Ease of use with concurrent collections (CnC)," *Hot Topics in Parallelism*, 2009.