MCBeth: A Measurement-based Quantum Programming Language

Aidan Evans, Seun Omonije, and Robert Soulé
Department of Computer Science
Yale University
New Haven, CT, USA
{aidan.evans,seun.omonije,robert.soule}@yale.edu

Robert Rand
Department of Computer Science
University of Chicago
Chicago, IL, USA
rand@uchicago.edu

***Abstract*—This work introduces MCBeth, a quantum programming language that bridges the gap between near-term and non-near-term languages. MCBeth allows users to directly program and simulate measurement-based computation by building upon the measurement calculus. While MCBeth programs are meant to be executed directly on hardware, to take advantage of current machines we also provide a compiler to gate-based instruction sets. We argue that MCBeth is more natural to use than common low-level languages, which are based upon the quantum circuit model, but still easily runnable in practice.**

***Index Terms*—quantum computing, programming languages, measurement-based quantum computing, one-way quantum computer, distributed computing**

I. Introduction

Existing languages for quantum computing fall into two categories: NISQ or non-NISQ. NISQ languages, such as OpenQASM [1], are languages aimed to interface with *Noisy Intermediate-Scale Quantum* technology, i.e., near-term hardware which supports fifty to several hundred qubits but which may also be prone to error [2]. They are analogous to assembly languages, in that they are low-level, and they describe quantum circuits, which can be verbose and difficult to understand. However, NISQ languages are runnable in practice. In contrast, non-NISQ languages, such as Q# [3] and Silq [4], describe an algorithm at a high-level and are easier to read and write; these languages, on the other hand, tend to not be runnable on existing devices because, e.g., there is no way to compile them to a runnable language or because the compilation is not optimized for the current hardware. An ideal quantum language would bridge the gap between NISQ and non-NISQ; it would be easy to read and write, but also easily runnable on existing devices.

Moreover, while there exists a plethora of software tools for dealing with gate-based models of quantum computation such as quantum circuits, few software tools currently exist for dealing with alternative models such as measurement-based quantum computing (MBQC) models – despite promising experimental results of running measurement-based algorithms on real quantum computers [5]. At the moment, most tools available for effectively working with measurement-based quantum algorithms are primarily theoretical frameworks, like *the measurement calculus* [6]. The measurement calculus (§II) has proven to be a powerful tool for reasoning about MBQC algorithms, but there are currently few practical ways to create and simulate programs written in this language.

In this paper, we present MCBeth, a programming language and framework based on the measurement calculus which allows one to easily read, write, and simulate quantum programs (§III, §IV). We argue that by basing MCBeth on MBQC models, it avoids pitfalls present in the quantum circuit model with respect to ease of use, while retaining the ability to run on real devices.

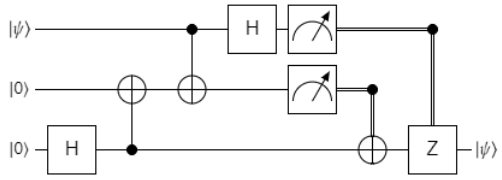
We provide a library containing tools for creating MCBeth programs via primitive instructions and non-primitive instructions which can be compiled down to the primitives. We provide utility functions for converting the primitive measurement calculus instructions to a standard form and checking program validity. We provide functions for the weak and strong simulation of MCBeth programs and implementations of several quantum algorithms written in MCBeth. Additionally, we provide a translation framework from MCBeth to other quantum programming frameworks such as OpenQASM and Cirq [7], giving programmers the ability to test their programs on popular quantum devices. We provide functions for constructing distributed MCBeth programs, converting non-distributed MCBeth programs into a distributed form, and provide data on the simulation runtime of different size systems (§V, §VI).

II. Preliminaries

We assume the reader is familiar with the basics of quantum computing and will refer to a standard text, such as [8], as required. Here we discuss the basics of MBQC.

A. Measurement-Based Quantum Computing

MBQC is a promising alternative to the standard circuit model for quantum computation. One important model of MBQC is the *cluster state model* [9]. The cluster state model involves: (1) the creation of a collection of entangled qubits called a *cluster state*, (2) the measurement of qubits in different bases, and, in some variations, (3) the application of X and Z *corrections* using the Pauli operators. Cluster states are typically obtained by applying CZ gates between qubits to



(a) Traditional Circuit

$J(\alpha)(q_0, q_1); J(\beta)(q_1, q_2)$ where $\alpha, \beta = 0$

(b) Cluster and Commands

$$|q_2\rangle = |q_0\rangle \text{ — } \boxed{R_z(-\alpha)} \text{ — } \boxed{R_x(-\beta)} \text{ —}$$

$$|q_2\rangle = R_x(0)R_z(0)\,|q_0\rangle = |q_0\rangle$$

(c) Cluster Effect on Original Quantum State

Fig. 1: Teleportation

entangle them. Because the entanglement of the cluster only decreases in the second stage, the cluster state model is often referred to as *one-way quantum computation*. The basis of each measurement and whether corrections are performed may be dependent on the outcome of earlier measurements performed during the computation. In measurement-based computation, this ordering of entanglement, measurement, and correction operations is enforced.

The *measurement calculus* by Danos et al. [6] provides a clear syntax and semantics to cluster state operations/instructions, also known in the measurement calculus as *commands*, along with rewriting rules for simplifying sequences of instructions. The measurement calculus, like the cluster state model, is universal for quantum computation. Although there has been much work on the measurement calculus, it has lacked an implementation. MCBeth aims to rectify this with a practical and usable language for the measurement calculus.

III. MCBeth by Example: Teleportation

We now introduce MCBeth and the measurement calculus by walking through the implementation of the quantum teleportation algorithm. To highlight how MCBeth differs from the quantum circuit model, we provide implementations of teleportation using both the gate-based model and MCBeth.

Teleportation is fundamental to any form of distributed quantum computing and quantum networking. Teleportation is important because it provides a way to transfer a quantum state between two qubits without using a quantum channel or violating the *no-cloning theorem*. The no-cloning theorem states that it's not possible to *copy* a quantum state from one qubit to another. Teleportation, however, allows us to at least *transfer* the quantum state to another qubit via entanglement and sending of information over classical channels.

Gate-based Teleportation. Figure 1a depicts the traditional quantum circuit for this process. The first four gates – the two Hadamard and CNOT gates – entangle the qubits together. Measurement is then performed on the first and second qubits. Finally, X and Z gates are applied to the last qubit based on the measurement outcomes of the first and second qubits. One may not find it clear at first glance, however, how the qubits of this circuit actually affect one another and why the state ends up being transferred from the first qubit to the last qubit. The measurement calculus, on the other hand, does help us reason about how the qubits interact with each other and about the overall computation in general.

Using Clusters to Construct Measurement-based Programs. We can start by approaching the problem diagrammatically. We want to find a *cluster* that will render the desired effect on the input qubits' states during computation; in this case, we wish for our initial qubit's state to be transferred identically to some other qubit. We will ultimately use the cluster depicted in Figure 1b. A cluster consists of qubits arranged in a grid so that there are rows and columns of qubits; for example, Figure 1b consists of one row and three columns. Each qubit may be considered a node in a graph and then entanglement between qubits is represented via edges.

Each distinct cluster has an associated program and can be interpreted as performing a computation starting on the leftmost qubits in a cluster, i.e., the leftmost column, and propagating the computation through the cluster until the final state is remaining in the rightmost qubits of the cluster. This relationship between the leftmost initial qubits and rightmost final qubits can be mapped to a quantum circuit acting on n qubits, where n is the number of rows in our cluster. Thus, reading the graph of a cluster from left to right, nodes in the first column represent input qubits and nodes in the last column represent output qubits. We display several example clusters with their corresponding circuits and commands collectively in the online appendix.[1]

Edges between nodes in different columns represent the application of the command $J(\alpha)(q_i, q_j)$ in the measurement calculus. The $J(\alpha)(q_i, q_j)$ command is a non-primitive command which compiles down to several lower-level primitive commands; specifically, it entangles qubits q_i and q_j, measures q_i, and then corrects q_j as necessary, where q_i is the left qubit, q_j is the right qubit, and $\alpha \in [0, 2\pi]$ is the *angle of measurement*. The angle of measurement determines what basis we measure q_i in. We explain this further below.

Edges between nodes in the same column represent the application of the $CZ(q_i, q_j)$ command, which just entangles q_i and q_j – as seen in Figure 3b.

We can also concatenate two clusters together by merging the nodes in the last column of one cluster with the nodes in the first column of another cluster.

Writing the Measurement-based Teleportation Program in MCBeth. For our purposes, we want the three qubit cluster

[1] https://www.aidantevans.com/pubs/mcbeth-appendix.pdf

depicted in Figure 1b. Here we have three qubits positioned in a horizontal line with entanglement between q_0 and q_1 and between q_1 and q_2. Computation then proceeds by measuring q_0 in a certain basis, impacting q_1's state, and then measuring q_1, impacting q_2's state. Specifically, we get the following program written in the measurement calculus:

$$J(\alpha)(q_0, q_1); J(\beta)(q_1, q_2)$$

For teleportation, because we wish for the final state to equal the initial state, we set $\alpha, \beta = 0$ and now get

$$J(0)(q_0, q_1); J(0)(q_1, q_2)$$

Figure 1c shows the effect this will have on the original state of qubit q_0. Note that this corresponds to performing an R_z rotation of 0 and R_x rotation of 0; i.e., the state is not changed yet because the computation was propagated through the clusters, we end up with our state from q_0 in q_2 as desired.

In MCBeth, we would represent these commands using the following program:

```
Input(0);
PrepList([1, 2]);
J(0.0, 0, 1);
J(0.0, 1, 2);
```

Here, we have a list of commands. The "J(0.0, 0, 1);" command is the command for $J(0)(q_0, q_1)$.

This program can then be both decomposed into the primitive commands, which we describe in detail below.

Decomposing into Primitive Commands. The commands

$$J(0)(0, 1); J(0)(1, 2)$$

help us make sense of the circuit in Figure 1a because in the measurement calculus, non-primitive commands such as J are actually slight abstractions of lower-level quantum operations we call *primitive commands*. These include, preparation (N_i), entanglement (E_{ij}), measurement (M_i^α), and Pauli-X and Pauli-Z corrections (X_i and Z_i). We now discuss these commands individually and their associated MCBeth counterparts; further detail of the MCBeth language is provided in Section IV.

The preparation command, N_i, initializes qubit i to the $|+\rangle$ state. The $|+\rangle$ and $|-\rangle$ states are two common states used in quantum computation:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$
$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

In MCBeth, the initialization command is represented by the command `Prep(`q`)` where q is a qubit; thus, to initialize a qubit 0, we write `Prep(0)`. For convenience, we also provide a command which allows one to initialize a list of qubits at once: `PrepList([0; 1; 4])`. This initializes qubits 0, 1, and 4 to the $|+\rangle$ state.

Since some qubits in a program may be passed in as input as opposed to initialized, we signify this distinction by having an input command: `Input(`q`)`. During simulation, a mapping from qubits to states can then be passed into the simulator which will set each input qubit q to its desired state at the start of the simulation.

All qubits must either be declared using the `Prep` or `Input` commands at the start of the program.

The measurement calculus's entanglement command, E_{ij}, entangles qubits i and j by performing a controlled-Z gate between them with i the control and j the target. In MCBeth, we write this as `Entangle(`q_1`, `q_2`)`.

The measurement command, M_i^α, measures qubit i with the following basis states:

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

where $\alpha \in [0, 2\pi]$; α is the *the angle of measurement*. Notice that when $\alpha = 0$, measurement is performed with the $|+\rangle$ and $|-\rangle$ basis states. In MCBeth, we write `Measure(`α, q, $signals_s$, $signals_t$`)` where α is a float and $signals$ are lists of qubits – signals are explained further below.

The correction commands, X_i and Z_i, apply the Pauli X or Z gates, respectively, to qubit i. In MCBeth we write, `XCorrect(`$q, signals$`)` and `ZCorrect(`$q, signals$`)`.

The measurement and correction commands may depend on the results of past measurements performed. The outcome of a measurement on qubit i is denoted as s_i where

$$s_i = \begin{cases} 0 & \text{if the qubit collapsed to } |+_\alpha\rangle \\ 1 & \text{if the qubit collapsed to } |-_\alpha\rangle \end{cases}$$

Dependent commands use *signals* to determine the operation performed. Signals are the exclusive-or of measurement outcomes. In other words, signal $s = (\oplus_{i \in Q} s_i)$ where Q is the set of qubits the commands depends on. Q is referred to as the *domain* of a signal.

Dependent corrections are written X_i^s and Z_i^s. The operation is performed if $s = 1$ and not performed if $s = 0$. For example, in the case of X_i^s, $X_i^1 = X_i$ and $X_i^0 = I_i$ where I is simply the identity matrix.

Dependent measurements are written as

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

where t and s are signals. In MCBeth, these signals are included in the measurement and correction commands as lists of qubits.

A measurement calculus program consists of a finite sequence of commands and three sets of qubits: V, I, and O. V is the computational space, i.e., the set of all qubits used in the computation. I is the set of input qubits and O is the set of output qubits. Moreover, in this paper, a sequence of commands, $A_1; \ldots; A_n$ is read from left to right.[2] Note that

[2]In Danos et al.'s measurement calculus, commands are written from right to left so mathematical operations, i.e., matrix multiplication, are easier expressed. Since we are designing a language for programming, we choose the standard representation for sequencing.

$I \cup O \subseteq V$ and the sets I and O may share the same qubits – $I \cap O$ may be non-empty.

Programs must abide by the following constraints:

1) no command depends on an outcome not yet measured
2) no command acts on a qubit already measured
3) no command acts on a qubit not yet prepared, unless it is an input qubit
4) a qubit i is measured by some M_i^α if and only if i is not an output qubit

This enforces a physically realizable execution order.

The given set of non-primitive commands are universal for quantum computation; i.e., any quantum algorithm can be implemented using only these commands:

$$CZ(i,j) := E_{ij}$$
$$J(\alpha)(i,j) := E_{ij}; M_i^{-\alpha}; X_j^{s_i}$$

where $V = \{i,j\}$ for both commands; for CZ, $I = O = \{i,j\}$; for $J(\alpha)(i,j)$, $I = \{i\}$ and $O = \{j\}$.

Standardizing Command Order. We also want all entanglement to occur before measurement; this process is known as *standardization*. Standardizing a program takes an arbitrary, well-formed sequence of commands and rewrites the program to put commands in the following order: preparations, entanglement, measurement, corrections. The program is rewritten using a set of rewriting rules laid out by [6]. Once standardized, the program is simplified and in the format required for cluster state computation. This is easily done in MCBeth by passing a list of commands into the `standardization` function; given a list of commands, it will return an equivalent standardized list of commands.

Importantly, the rewriting rules preserve the semantics of the program; therefore, the standardized program will result in the same computation as the original, non-standardized one. In interest of space, we defer the full set of rewriting rules to the online appendix.

Returning to our example of teleportation, we can decompose our program

$$J(0)(0,1); J(0)(1,2)$$

into

$$E_{01}; M_0; X_1^{s_0}; E_{12}; M_1; X_2^{s_1}$$

Here, $V = \{0,1,2\}$, $I = \{0\}$, and $O = \{2\}$; in other words, the quantum state of qubit 0 is transferred to qubit 2. Because 1 and 2 are non-input, they are initialized to the $|+\rangle$ state.

Then, using the rewriting rules, we can obtain the equivalent program below:

$$E_{01}; E_{12}; M_0; [M_1]^{s_0}; Z_2^{s_0}; X_2^{s_1}$$

Thus, in MCBeth, we would obtain the standardized program shown in Figure 2.

Note that standardization now allows us to perform all of the entanglement at once. Furthermore, if we create the initial entanglement of the qubits on one computer, we can distribute the qubits over a quantum network to two other nodes such that qubits 0 and 1 are on one node and qubit 2 is on another. Thus, we can transfer the quantum state of qubit 0 to qubit 2 over any distance, as long as we can transfer the signals s_0 and s_1 over a classical channel. Further application of this distributed architecture is explained in Section VI-A.

```
Input(0);
PrepList([1, 2]);
Entangle(0, 1);
Entangle(1, 2);
Measure(0, 0.0, [], []);
Measure(1, 0.0, [0], []);
ZCorrect(2, [0]);
XCorrect(2, [1]);
```

Fig. 2: Teleportation in MCBeth after Standardization

Comparison to the gate-based version. By using the measurement calculus, we can see how a circuit such as the one in Figure 1a could be constructed to teleport a quantum state between two qubits. We first constructed a cluster state which allowed us to have the desired effect on our input state – in this case, no effect. We then constructed a program representing this cluster, decomposed the program into low-level primitive commands, and standardized the program. This process gave us a sequence of quantum operations where entanglement happens first, then measurement, and finally corrections – similar to that of our circuit in Figure 1a.

In the gate-based model, using circuit identities, the $CNOT$ and Hadamard gates can be converted to two CZs. Remember that our entanglement command, E_{ij}, is equivalent to a CZ; therefore, the $CNOT$ and Hadamard gates entangle the three qubits together. The circuit then measures two qubits and uses this outcome to apply dependent X and Z gates, just like in the measurement calculus.

In the measurement calculus, however, we had a clear process of how to approach setting up the cluster and the effects it would have on our final qubit; we were easily able to reason about how the state would be transferred among qubits using the J operator. We did not have this ability with the gate-based circuit.

MBQC's forced ordering of operations clearly distinguishes computation into three phases which highlight the true role that entanglement and measurement have in quantum programs. Because the ordering requires all entanglement to be realized upfront, instead of buried in the computation, MBQC shows how entanglement is really a resource for a quantum computation. An initial amount of entanglement is provided upfront in the form of an entangled cluster. Computation proceeds by using measurement to continually remove qubits from the cluster; thus, disentangling the cluster. This underscores how the computation itself is primarily performed by the measurement operations. Thus, with MBQC, it is clear what quantum operation the computation itself primarily depends on. This is in direct contrast to the quantum circuits used by popular NISQ languages where every step of the program (i.e.,

every gate) is just said to simply perform a computation.

IV. Programming in MCBeth

We now discuss programming in MCBeth. We first informally present the syntax. A BNF specification and denotational semantics may be found in the online appendix. We then provide an implementation of the Deutsch-Jozsa algorithm in MCBeth to highlight features of the language beyond what was covered in Section III. Additionally, a 2-bit version of Grover's algorithm may be found in the online appendix.

A. Syntax

A MCBeth program consists of a list of commands separated by semicolons. The supported commands break down into three general categories: input preparation, core execution, and readout. When writing a program, qubits are referenced via natural numbers.

The input preparation phase consists of two types of commands: "Input" commands and "Prep" commands. "Input" allows one to declare which qubits should be passed in as input during the start of the computation, while "Prep" initializes the qubit to $|+\rangle$. "InputList" and "PrepList" commands are added to allow for the easy initialization of a list of qubits.

The main execution phase consists of applying the commands for entanglement, measurement, and Pauli corrections to the quantum system initialized in the input preparation phase. For the applicable commands, signals are passed in as lists of qubits.

Finally, an optional readout phase performs a final measurement on specified qubits in a custom basis and stores measurement outcomes as output reported at the end of execution. This is done using the `ReadOut(...)` command.

B. The Deutsch-Jozsa Algorithm

Deutsch's problem [10] motivated an early example of the power of quantum computing. The question is as follows: given a black box function that implements $f : \{0,1\} \to \{0,1\}$, determine if that function is constant ($f(0) = f(1)$) or balanced ($f(0) \neq f(1)$). A quantum computer can solve this using a single query to the oracle.

The Deutsch-Jozsa [11] algorithm generalizes Deutsch's algorithm and is widely considered the starting point for understanding quantum algorithms. This is the case due to the straightforwardness of the problem, and the relative simplicity of implementation.

The algorithm solves the following problem: provided a quantum oracle which implements a function $f : \{0,1\}^n \to \{0,1\}$ and the knowledge that this function is either constant (that is, all outputs are 0, or all are 1) or balanced (half the input domain returns 1, the other half returns 0), determine if f is constant or balanced using the quantum oracle. Classically, this problem takes $2^{n-1}+1$ queries to the oracle in the worst case. Using a quantum computer, however, we only need one query to the oracle.

We convert the MBQC-based algorithms for Deutsch-Jozsa developed by [12] and [13] into MCBeth programs. We present a 6-qubit cluster state for this in Figure 3b; this cluster handles the case where the oracle being queried is balanced. In Figure 3c, we provide the MCBeth program for this; we also show here how taking advantage of different measurement bases allows us to remove two qubits from the cluster state (q_4 and q_5) and achieve the same algorithm. We also compare it to the circuit version displayed in Figure 3a to demonstrate the differences in readability and operation count for programs with the same output.

The measurement calculus program for this cluster is below, and can be extended to accommodate an additional bit x by adding a CZ operator from qubit 1 to qubit x, and a J operator from qubit x to an additional qubit y:

$$J(\alpha)(q_0, q_1); CZ(q_1, q_2); CZ(q_1, q_3);$$
$$J(\alpha)(q_2, q_4); J(\alpha)(q_3, q_5);$$

As displayed in the MCBeth program outlined in Figure 3c, performing readout measurements on qubits 1, 2, and 3 in the bases shown instead of all in the same base allows us to remove the need for qubits 4 and 5.

Furthermore, removing the CZ operations and leaving the rest of the cluster intact will implement the constant oracle for the problem.

V. Simulation and Execution

In this section, we discuss how we simulate and execute programs in MCBeth through our own simulator implementation, and how we also compile them down for use on general gate-based quantum systems. The MCBeth codebase is available online[3] under an open source license.

A. Simulation Implementation

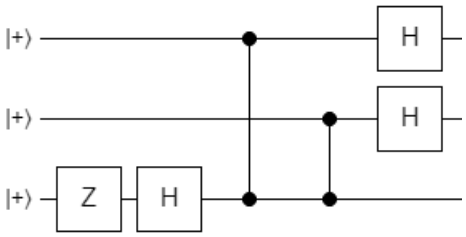
We implement MCBeth using OCaml. Given the lack of available libraries for quantum computing in OCaml, we created our own from scratch.[4] We define each individual MCBeth command as an inductive data type and represent a program as a list of these commands. Importantly, qubits are specified as integers; a constraint on the current implementation is that an initial system must start the qubit number at 0 and then continue numbering successively so that if a system uses n qubits, it will have a qubit associated with each integer between 0 and $n-1$, inclusively. We use the Lacaml matrix library[5] to simulate the execution of a MCBeth program.

We provide two simulators for MCBeth programs: one for weak simulation and one for strong simulation. The key difference between weak and strong simulation is that weak simulation uses randomization to determine which basis to collapse to during measurement and only computes that computation path, while strong simulation computes both computation paths and uses the result to construct a probability distribution. While more computationally intense, strong simulation gives a whole picture of the possible outcomes while

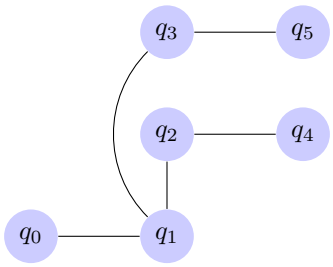
[3] https://github.com/seunomonije/mcbeth

[4] https://github.com/seunomonije/mcbeth/tree/master/mcbeth/lib/qlib

[5] https://github.com/mmottl/lacaml



(a) Deutsch-Jozsa circuit with a balanced oracle; for a constant oracle, remove the two CZ gates in the middle.



(b) Cluster with balanced oracle.

```
PrepList([0, 1, 2, 3]);
J(pi, 0, 1);
CZ(1, 2);
CZ(1, 3);
ReadOut(1, Z);
ReadOut(2, X);
ReadOut(3, X);
```

(c) MCBeth with Balanced Oracle and Read-out Optimizations

Fig. 3: Deutsch-Jozsa in three different representations

weak simulation only gives one possible outcome; therefore, in order to extract useful information from a weak simulation, the simulation must be run for many iterations to create a sample of possible outcomes. Nonetheless, weak simulation is more faithful to actual execution on a quantum computer and is more appropriate for deterministic programs like the Deutsch-Jozsa algorithm.

A weak simulation of an MCBeth program is executed by passing the program into the `rand_eval` function contained in the `Backend.Run` module; similarly, strong simulation is executed with the `simulate` function.

These functions return three objects: a matrix representing the output quantum system, an ordered list of qubits corresponding to those qubits in the output quantum system, and a table containing entries for the readout qubits. Because the quantum system during the computation gets partially destroyed by the measurement commands, the ordered list of qubits is required in order to properly know the position of each qubit left in the system.

The return output matrix differs in type for weak and strong simulation. For weak simulation, because we can simply collapse to one basis randomly during the measurement operation, we can perform the simulation using the state vector representation; therefore, a state vector of the output system is returned by `rand_eval`. For strong simulation, because we need to combine multiple possibilities of measurement, we perform the simulation using the *density matrix* representation; therefore, a density matrix of the output system is returned by `simulate`. A density matrix is another way to represent a quantum system mathematically; for unmeasured (*pure*) states, instead of simply having the system represented as a state vector $|\psi\rangle$, we now represent the system as $|\psi\rangle\langle\psi|$. For probability distributions over measured (*mixed*) states, the corresponding density matrix is the sum of the unscaled outcomes: For instance, if we measure $\rho = |+\rangle\langle+|$ in the computational basis, we get $|0\rangle\langle 0|\rho|0\rangle\langle 0| + |1\rangle\langle 1|\rho|1\rangle\langle 1| = \frac{1}{2}I_2$.

B. Removing Measured Qubits

Because measurement is a destructive operation, instead of keeping the result as a component in the vector or matrix representing the system, during the measurement operation we can remove it by projecting down into a subspace of a lower dimension. The projection into a smaller subspace removes the qubit from the system while also decreasing the size of a state vector by one-half and the size of a density matrix by three quarters. This allows for future computations to multiply significantly smaller matrices, allowing for a faster simulation of the quantum circuit. For example, in the case of our teleportation example in Figure 2, each measurement command will remove a qubit from the system; therefore, we no longer need to keep track of it in our matrix.

C. Serialization and cross compatibility

In order to enable exploration of programs written in MCBeth to other formats, MCBeth serializes down to a standard JSON format which can be parsed and translated to other frameworks as needed.

Serialization of the language is straightforward. Each MCBeth command maps to a JSON object with the name of the command as the key, and a nested JSON object as a value. This nested JSON object contains relevant information about the command following the structure of the *cmd* type in our language implementation. For example, the MCBeth command:

Entangle(0, 1)

translates to the JSON object:

{'Entangle': {'on_qubits': [0, 1]}}

As programs get larger, we represent them with a list of these objects, which can be parsed and converted into the desired language. In our work, we provide a translator from MCBeth to Cirq with the *CirqBuilder* class. With this translation implementation, we can translate MCBeth programs into OpenQASM by running them through Cirq first. The compiled OpenQASM representation of MCBeth programs makes the language compatible with almost every major quantum programming framework or language, and also allows us to run programs written in MCBeth on popular quantum hardware, e.g., the deployment of programs submitted via IBM's cloud platform. The programs that execute on these quantum devices will not be MCBeth programs – as measurement-based quantum computers aren't available to access and submit programs to online – but they will correspond semantically to the original program.

VI. MCBeth-specific Architectures

In this section, we outline some systems and computer architectures that can take advantage of measurement-based computation's unique benefits, and how MCBeth can be used to realize and work with these systems.

A. Distributed Computation

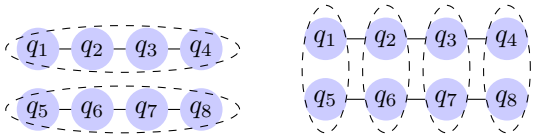
One of the hardest challenges in achieving scalable quantum computation is the issue of having quantum computers handle large numbers of qubits. By using MCBeth to create programs that can be distributed to multiple quantum computers, the issue of scaling the number of qubits a computer can handle is isolated to a single node which only needs to focus on entanglement.

Because MCBeth standardizes the program to have entanglement first, MCBeth qubits can be initialized on a central entangling node and sent to different nodes via quantum channels (for instance, the photonic channels proposed by [14]). The entangling node would be made specifically for the purpose of entanglement and, thus, its hardware could be optimized for performing entanglement. Because the qubits are entangled, operations on one entangled qubit will still affect the qubits with which it is entangled, regardless of where these qubits are. Moreover, because the signals are classical pieces of information, they may be shared across classical channels during the computation. This would allow us to distribute both the measurement and the corrections sections of the computation across nodes, removing the need for each node to have the ability to store and operate on large numbers of qubits. While this could be done in the gate-based model by converting non-suitable circuits into a distributed form, programs in MCBeth are created by default in a distributed form; therefore, there is no need to convert the program; This makes MCBeth more natural to program in for this purpose.

Not only does distributed computation allow for the decrease in qubits per edge node, but it also allows for the execution of certain steps of a program in parallel. Figure 4a displays just this case; there are two subsystems: $S_1 = [q_1, q_2, q_3, q_4]$ and $S_2 = [q_5, q_6, q_7, q_8]$. Because no qubit in S_1 is dependent on a qubit in S_2, and vice versa, the computation of each subsystem can be done perfectly in parallel.

We provide a module `Backend.Distributed` to split a program up into distributed subsystems by passing it into `build_dist_prog` along with a list of lists containing how the qubits should be partitioned. This module contains utility functions to split a program into isolated subprograms for each network node. It is up to the designers of the architecture to implement the communication between nodes.

We cannot simulate accurately all distributed computations while also retaining distributed subsystems, because we cannot simulate the entanglement between nodes when entangled qubits are split between different nodes. However, if we split a computation up in such a way where each node in the network has only qubits entangled with other qubits in the same node, such as that in Figure 4a as opposed to Figure 4b, then entanglement does not exist between nodes. Thus, an accurate distributed simulation can be performed in such cases. To measure simulation performance, we ran weak simulations of arbitrary programs utilizing n qubits per node with m nodes. Each node, therefore, contained one subsystem of the computation, akin to Figure 4a. Results from the weak simulation of distributed programs of various configurations on a single processor, with each node simulated on separate threads, is shown in Figure 5. As expected, the running time of the simulation increases exponentially as the number of qubits increases. Then, as the number of subsystems increase, the runtime increases linearly since the subsystems were run on separate threads of the same processor.



(a) Optimal (b) Non-Optimal

Fig. 4: Partitioning

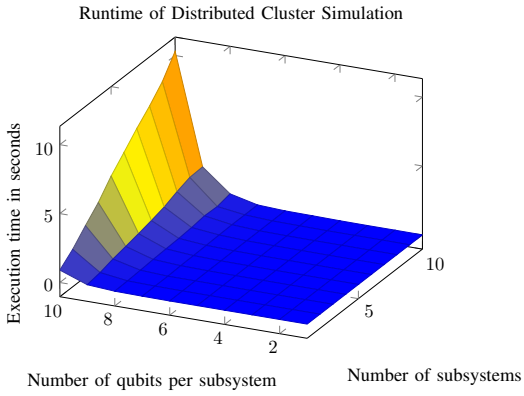


Fig. 5: Distributed Simulation

B. MCBeth for MBQC-Specific Architecture

In gate-based models, mapping between the program operations and the hardware is achieved by compilation, which maps input circuits to the topology of the hardware, and *pulse scheduling*, which configures where and when pulses are sent to the individual qubits. With a variety of gates and operations, compiling and scheduling gate-based circuits can become a complex and involved process. In measurement-based models and specifically cluster-state computing, compiling the circuits down is straightforward.

The separation of entanglement and measurement in cluster-state, one-way computing models makes it easier for architectures to delegate hardware specifically for entangling qubits and creating the cluster-state, then send pulse schedules to that cluster without needing to worry about re-entangling or any other non-measurement operations. An example of this architecture can be seen in a 2014 experimental implementation of Simon's algorithm [5]. The group was able to

modify an existing photonic architecture used for a quantum error correction code to generate a linear 1D cluster state by producing photon pairs and entangling them using a polarizing beam splitter (PBS). They then sent pulses to these entangled photons to measure in the X, Y, or Z basis, with distinct pulse paths for each basis.

In theory, all quantum algorithms that could be represented through the cluster state, one-way model could be run on this architecture, assuming that the hardware could produce and entangle enough photons for an n qubit cluster state. Compiling all quantum algorithms in this model becomes simple: prepare the cluster state, and schedule each measurement one after another. This allows the programmer to be able to develop algorithms at a high-level while also understanding the program at the pulse level. In many cases, pulse-level understanding would not be necessary, however; this shows that from the programmer's point of view, understanding the computation at all levels in MCBeth is significantly easier than doing so in the gate-based model.

VII. Related Work

Compiling Non-NISQ Languages. Some non-NISQ languages, like Q# [3], can be compiled to a NISQ language, allowing them to be run on existing devices. These languages don't give programmers a good model for realizable quantum hardware, which potentially limits their utility because no intuition for the core of quantum computation is built. In contrast, with MCBeth, the programmer gets higher-level simplicity with hardware-level awareness.

ZX-calculus. Like MCBeth, the ZX-calculus [15] offers an alternative to the quantum circuit abstraction. It introduces both graphical representations and computations that aren't possible or apparent in the quantum circuit model. However, while the ZX calculus is a powerful and extremely useful tool to reason and work with MBQC programs, it is not intended to perform the role of a programming language.

Paddle Quantum. Paddle Quantum is a Python framework that offers a sub-module for measurement-based quantum computing [16]. Paddle Quantum compiles from gate-based programs to measurement-based programs, while MCBeth allows for measurement-based to gate-based translation. The MBQC functionality of Paddle Quantum is a sub-module of a larger Python package, rather than a standalone package like MCBeth, and it is more verbose in its command names. Paddle does, however, provide a more flexible naming convention for qubits and their MBQC module performs additional optimizations for weak simulation. Finally, Paddle Quantum supports weak simulation, while MCBeth supports both weak and strong simulation.

VIII. Conclusion

Our experience with MCBeth point to two benefits. First, a measurement-based language leads to a more intuitive algorithm composition procedure compared to quantum circuits, which current NISQ languages are based on. Second, MCBeth is more easily adapted to scalable, distributed quantum computing and in the design of quantum hardware.

Prior to this work, measurement-based quantum computing was known to be a powerful alternative model to gate-based quantum computation, but it had rarely been used for practical quantum programming. In developing MCBeth, we were able to explore the advantages and disadvantages of this model from a programmer's perspective. We created an open source repository containing a simulator for the easy creation and simulation of MCBeth programs; we also provide a compiler from MCBeth to a gate-based model, allowing MCBeth programs to be run on common, real quantum hardware, or imported into other quantum programming languages. We argued that programming in MCBeth allows one to naturally create distributed quantum algorithms, which could be executed on a network of machines where each machine is specialized to handle a different type of quantum operation. Therefore, MCBeth is easily runnable in practice both in simulation and on different types of quantum hardware.

Acknowledgment

This work is funded in part by EPiQC, an NSF Expedition in Computing, under Award Number CCF-1730449.

References

[1] A. W. Cross *et al.*, "Open quantum assembly language," *arXiv:1707.03429*, 2017.
[2] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 2018.
[3] K. Svore *et al.*, "Q# enabling scalable quantum computing and development with a high-level DSL," in *RWDSL*, 2018.
[4] B. Bichsel *et al.*, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *PLDI*, 2020, pp. 286–300.
[5] M. S. Tame *et al.*, "Experimental realization of a one-way quantum computer algorithm solving simon's problem," *Phys. Rev. Lett.*, vol. 113, p. 200501, Nov 2014.
[6] V. Danos *et al.*, "The measurement calculus," *J. ACM*, vol. 54, no. 2, pp. 8–es, Apr 2007.
[7] Cirq Developers, "Cirq," Dec 2022. [Online]. Available: https://github.com/quantumlib/Cirq/
[8] M. A. Nielsen and I. L. Chuang, "Quantum computation and quantum information," *Phys. Today*, vol. 54, no. 2, p. 60, 2001.
[9] M. A. Nielsen, "Cluster-state quantum computation," *Reports on Mathematical Physics*, vol. 57, no. 1, pp. 147–161, feb 2006.
[10] D. Deutsch, "Quantum theory, the Church-Turing principle and the universal quantum computer," *Proc. R. Soc. A: Math. Phys. Sci.*, vol. 400, no. 1818, pp. 97–117, 1985.
[11] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," *Proc. R. Soc. A: Math. Phys. Sci.*, vol. 439, no. 1907, pp. 553–558, 1992.
[12] G. Vallone *et al.*, "Experimental realization of the Deutsch-Jozsa algorithm with a six-qubit cluster state," *Phys. Rev. A*, vol. 81, no. 5, May 2010.
[13] M. S. Tame and M. S. Kim, "Scalable method for demonstrating the deutsch-jozsa and bernstein-vazirani algorithms using cluster states," *Phys. Rev. A*, vol. 82, p. 030305, Sep 2010.
[14] C. Monroe *et al.*, "Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects," *Phys. Rev. A*, vol. 89, no. 2, p. 022317, 2014.
[15] J. van de Wetering, "ZX-calculus for the working quantum computer scientist," *arXiv:2012.13966*, 2020.
[16] "Paddle quantum," 2020. [Online]. Available: https://github.com/PaddlePaddle/Quantum