Implementing NChooseK on IBM Q Quantum Computer Systems

Harsh Khetawat[1][0000−0003−2121−0958], Ashlesha Atrey[1], George Li[1], Frank Mueller[1][0000−0002−0258−0294], and Scott Pakin[2][0000−0002−5220−1985]

[1] North Carolina State University, NC, USA
{hkhetaw,amatrey,gpli,fmuelle}@ncsu.edu
[2] Los Alamos National Laboratory, NM, USA
pakin@lanl.gov

Abstract. This work contributes a generalized model for quantum computation called NChooseK. NChooseK is based on a single parametrized primitive suitable to express a variety of problems that cannot be solved efficiently using classical computers but may admit an efficient quantum solution. We implement a code generator that, given arbitrary parameters for N and K, generates code suitable for execution on IBM Q quantum hardware. We assess the performance of the code generator, limitations in the size of circuit depth and number of gates, and propose optimizations. We identify future work to improve efficiency and applicability of the NChooseK model.

Keywords: IBM Q · Quantum Computing · NChooseK.

1 Introduction

Despite a number of quantum-computing hardware platforms that have recently become available and their theoretical potential to more efficiently solve problems that are of high computational complexity [12,23], few computational scientists have embraced these novel platforms other than to demonstrate how very small problems may be solved. A short-term challenge to adoption is hardware immaturity (low qubit counts, rapid decoherence, poor gate fidelities, etc. [21]). However, a longer-term impediment to using quantum computing as a practical resource for computational scientists is the difficulty of *programming* such systems. Several programming paradigms and languages have been proposed in prior work to address this issue but they are all variants of the same, low level of abstraction over the underlying hardware [13].

We address the quantum programmability issue by designing a new high-level quantum programming model that reduces the challenge for programmers to express their computational problems. We implement the software tools for generating programs expressed in our model to target contemporary quantum hardware. More specifically, we develop the NChooseK model that constrains "N bits such that K of those bits must be TRUE" (where K can be a set of possibilities). This is of inerest since one can express NP-complete problems as NChooseK.

There are two unique aspects to our approach. First, the programming model we propose has a *classical* semantics, which makes it not only approachable by computational scientists who are not trained in quantum information theory but also easy to integrate into existing classical workflows. Second, the same program can be compiled *unmodified* on both gate-model quantum computers and quantum annealers. The model represents computational problems as satisfiability problems.

We first discuss our proposed programming model, NChooseK, and how it can be used to represent computational problems. We then provide an implementation via a code generator that generates code for IBM Q quantum computer systems [16] for any arbitrary parameters in the NChooseK programming model. We present results for the characteristics of the generated IBM Q circuit representation in terms of both circuit depth and gate count. Finally, we discuss the limitations of the code generator and explore future work to optimize and extend NChooseK to express more complex computation.

2 Background

A quantum Turing machine (or universal quantum computer) [11] is an abstract machine that models the behavior of a quantum computer. It can be used to formally express any quantum algorithm. A quantum circuit, which is computationally equivalent, is more widely used to model quantum algorithms rather than a quantum Turing machine. In the quantum circuit model, computation is described as a sequence of quantum gates on quantum registers. The model necessitates that any computation be reversible as quantum gates are unitary.

The code generator introduced in this work generates code for IBM Q quantum systems, which uses this model for computation. IBM Q systems use superconducting Josephson junctions [5] to implement the state of qubits. Other technologies, such as trapped ions [4] and optical lattices [2], have also been used to realize quantum computers in hardware. While these technologies realize quantum bits and gates through different substrates (materials) that exhibit quantum effects and operations, they follow a common quantum circuit model for operation.

Figure 1 depicts a 5 qubit IBM Q processor with the Josephson junctions for qubits, measuring circuits and interconnection between qubits. The image shows that there is no all-to-all connection between qubits as only certain qubits are connected to and may thus directly interact with one another.

3 The NChooseK Programming Model

We first describe the NChooseK programming model and then present our implementation on IBM Q systems. NChooseK is based on a single parameterized primitive, which can be used to express a wide variety of problems that quantum computer programmers might be interested in solving. The single NChooseK primitive constrains k of n Boolean variables to TRUE. More precisely, given

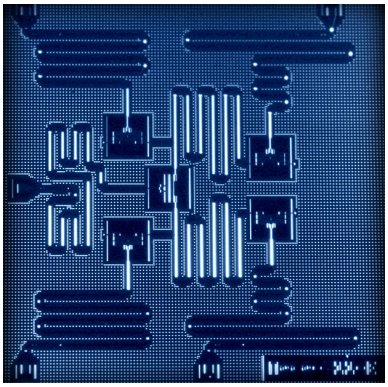


Fig. 1. IBM Q processor. Photo: IBM Research

n Boolean variables and a set of K integers in the range $[0, n]$, executing the primitive sets exactly k of those Boolean values to TRUE for some $k \in K$.

Executing an entire NChooseK program results in the system assigning Boolean values that honor all of the program's constituent primitives. For example, using the notation "nck(V,K)" to indicate that of the n variables in the set $V, k \in K$ of them must be set to TRUE, Figure 2 presents a trivial example of an NChooseK program. The program expresses the constrains that either 0 or 1 of the set of variables {a,b,c} must be TRUE, either 2 or 3 of the set of variables {b,c,d} must be TRUE, and exactly 1 of the set of variables {c,d,e} must be TRUE. Execution of this program amounts to computationally finding an assignment of variables that satisfies all three constrains. In this case, the sole solution is {b,d} = TRUE, {a,c,e} = FALSE.

$$\boxed{\begin{array}{l} nck(\{a,b,c\},\{0,1\}) \\ nck(\{b,c,d\},\{2,3\}) \\ nck(\{c,d,e\},\{1\}) \end{array}}$$

Fig. 2. Trivial example of an NChooseK program

3.1 Implementing the NChooseK model

The objective of this work is to convert the entire NChooseK program into a quantum black box (i.e., a unitary operator U_ω expressed as a quantum circuit) suitable for use in Grover's search algorithm [15]. Given a total of n Boolean variables in an NChooseK program, an exhaustive (classical) search for a satisfying assignment takes time $O(2^n)$. Grover's algorithm reduces the time to $O(\sqrt{2^n})$.

Consider the example in Figure 3 depicting a quantum black box that corresponds to $nck(\{b, c, d\}, \{2, 3\})$. It maps a quantum state $|bcd\rangle\,|x\rangle$ to $|bcd\rangle\,|x \oplus 1\rangle$ when exactly 2 or 3 of $|b\rangle$, $|c\rangle$, and $|d\rangle$ are $|1\rangle$ and to $|bcd\rangle\,|x\rangle$ otherwise, as required by Grover's algorithm.

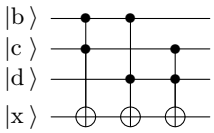


Fig. 3. A quantum black box for $nck(\{b, c, d\}, \{2, 3\})$

It is always possible to generate a circuit of the form used in Figure 3 from an NChooseK expression of a problem by following the approach described by Younes [24]. Although the gates required by Younes's approach—CNOT, CCNOT, CCCNOT, etc.—are not provided natively by modern hardware (with the occasional exception of CNOT), standard transformations can be applied to map these gates onto the available gate set. Assuming a typical gate set of single-qubit gates plus CNOTs, these transformations would normally realize the circuit shown in Figure 3 as a large (∼40-qubit) circuit. To keep the depth more manageable for current hardware, which exhibits relatively short decoherence times, one could employ the techniques developed by Cincio et al. [3] to find shorter-depth equivalents. In the case of $nck(\{b, c, d\}, \{2, 3\})$, Cincio et al. find the 17-qubit circuit shown in Figure 4.

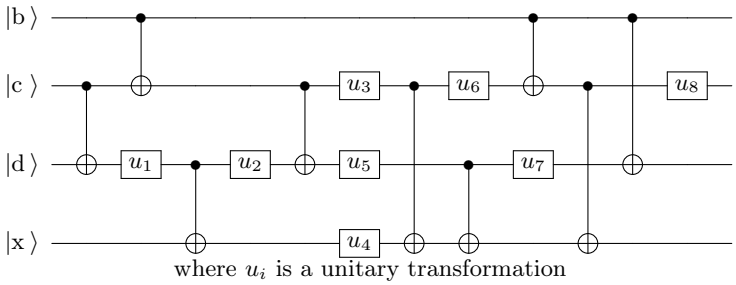


Fig. 4. A short-depth implementation of $nck(\{b, c, d\}, \{2, 3\})$ using only single-qubit gates and CNOTs

3.2 Generality of the NChooseK Model

The NChooseK model is based on the single, easy-to-understand constraint of "K of N bits must be TRUE" (where K can be a set of possibilities). The key advantages of this model are that

1. it is sufficiently high-level as to abstract away the underlying hardware architecture so compilers and optimizers can target gate-model quantum computers, quantum annealers, and even classical computers and supercomputers;
2. it enables programs written to that model to be formally specified and exhibit a unique interpretation, even across disparate architectures; and
3. as a classical programming model, it can integrate easily into existing, classical, scientific workflows.

Let us next demonstrate that the NChooseK model is useful. Specifically, we show how one can express NP-complete problems [6]—loosely, problems that cannot efficiently be solved classically—using NChooseK.

Circuit Satisfiability Given a Boolean expression, the goal of the circuit-satisfiability problem is to find a set of inputs for which the expression evaluates to TRUE or report that no such set exists. Figure 5 shows how one can construct the primitive operations needed to express circuit-satisfiability problems in terms of the NchooseK model.

The figure illustrates various NchooseK primitives as rectangles and the variables upon which they act as circles. The simplest primitives are shown in Figures 5a and 5b. The former illustrates that variable A can be biased towards TRUE by expressing, "1 of out 1 input should be TRUE". Likewise, the latter illustrates that variable A can be biased towards FALSE by expressing, "0 of out 1 input should be TRUE". Figure 5c shows that an inverter can be expressed as "1 out of 2 inputs should be TRUE", which leads one of variables A and $\neg A$ to be TRUE and the other FALSE. Expressing OR and AND requires a modicum of creativity. For a 2-input OR, Table 1a indicates that $K = \{0, 2, 3\}$ corresponds to valid rows and $K = \{1, 2\}$ corresponds to invalid rows.

Table 1. Adapting the truth table for OR for expression with NchooseK

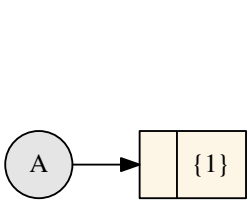
A	B	$A \vee B$	Valid?	#TRUE
F	F	F	✓	0
F	F	T		1
F	T	F		1
F	T	T	✓	2
T	F	F		1
T	F	T	✓	2
T	T	F		2
T	T	T	✓	3

(a) Truth table for Boolean OR

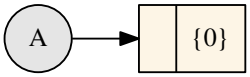
A	B	$A \vee B$	$A \vee B$	Valid?	#TRUE
F	F	F	F	✓	0
F	F	T	T		2
F	T	F	F		1
F	T	T	T	✓	3
T	F	F	F		1
T	F	T	T	✓	3
T	T	F	F		2
T	T	T	T	✓	4

(b) Truth table for Boolean OR with the third column repeated

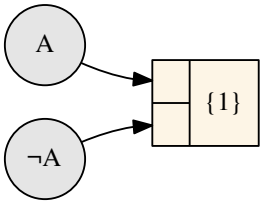
Because 2 appears in both the valid and invalid sets, one cannot use $nck(\{A, B, A \vee B\}, \{0, 2, 3\})$ to express OR. However, if one repeats the third col-



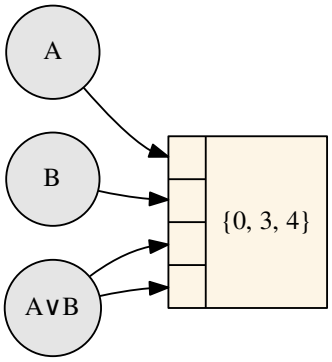
(a) $nck(\{A\}, \{1\})$: Favor A being TRUE



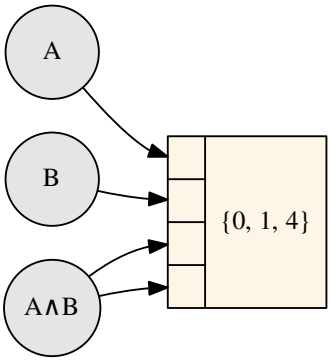
(b) $nck(\{A\}, \{0\})$: Favor A being FALSE



(c) $nck(\{A, \neg A\}, \{1\})$: Favor A and $\neg A$ having different truth values



(d) $nck(\{A, B, A{\vee}B, A{\vee}B\}, \{0, 3, 4\})$: Favor $A{\vee}B$ being TRUE if and only if at least one of A or B is TRUE

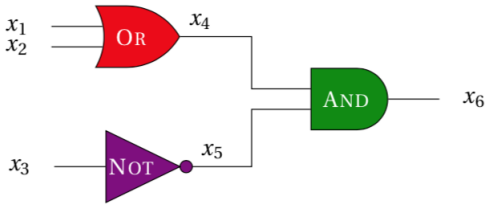


(e) $nck(\{A, B, A{\wedge}B, A{\wedge}B\}, \{0, 1, 4\})$: Favor $A{\wedge}B$ being TRUE if and only if both A and B are TRUE

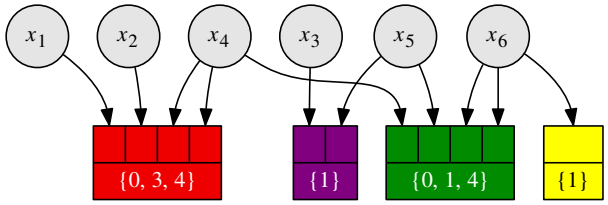
Fig. 5. NchooseK building blocks for circuit satisfiability

umn of the truth table as in Table 1b, then $K = \{0, 3, 4\}$ corresponds to valid rows and $K = \{1, 2\}$ corresponds to invalid rows. Because these are disjoint sets, OR can be expressed as in Figure 5d. One can employ the same trick to find that AND can be expressed with $nck(\{A, B, A{\vee}B\}, \{0, 1, 4\})$ as in Figure 5e.

A trivial circuit-satisfiability problem, corresponding to the function $x_6 = (x_1 \vee x_2) \wedge \neg x_3$, is illustrated in Figure 6. Figure 6a depicts this function as a



(a) Circuit expression



(b) NChooseK expression

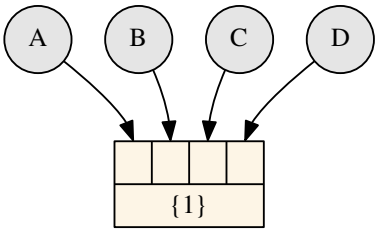
Fig. 6. Example of expressing a circuit-satisfiability problem with NChooseK

digital circuit, and Figure 6b demonstrates how to find values of inputs x_1, x_2, and x_3 in the NChooseK model. x_4 is constrained to x_1 OR x_2 using the OR primitive defined in Figure 5d; x_5 is constrained to the negation of x_3 using the inverter primitive defined in Figure 5c; x_6 is constrained to x_4 AND x_5 using the AND primitive defined in Figure 5e; and x_6 is further constrained to TRUE using the TRUE primitive defined in Figure 5a.

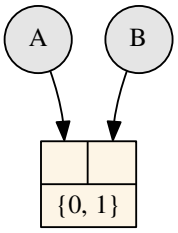
Because AND, OR, and NOT constitute a universal (classical) gate set, the implication is that *any* Boolean function can be expressed in the NChooseK model, demonstrating its universality.

Map Coloring Map coloring is another NP-complete problem. The goal is to color a map (a planar graph) using at most c colors, such that no two adjacent regions share a color, where is c is a constant, e.g., $c = 4$ to color a map of states or countries. Here, we show that the map-coloring problem, like the circuit-satisfiability problem, is easily expressed in the NChooseK model.

An NChooseK version of map coloring relies on only two primitives, which are illustrated in Figure 7. Following the approach taken by Dahl [10] we use a



(a) $nck(\{A, B, C, D\}, \{1\})$: Favor coloring the current region with exactly one of colors A, B, C, or D



(b) $nck(\{A, B\}, \{0, 1\})$: Favor coloring at most one of regions A and B with the current color

Fig. 7. NChooseK building blocks for map coloring

unary encoding of each region of the map: one Boolean for each of red, orange, green, and blue. In NChooseK, this is expressed as "1 out of 4 inputs should be TRUE" and is illustrated in Figure 7a. The other primitive ensures that for two adjacent regions, at most one of them is red—and likewise for each of orange, green, and blue. As Figure 7b illustrates, an "either 0 or 1 of 2 inputs must be TRUE" NChooseK primitive expresses that constraint.

Figure 8 illustrates the construction of a two-region map-coloring problem using the building blocks from Figure 7. The two regions are dubbed P and

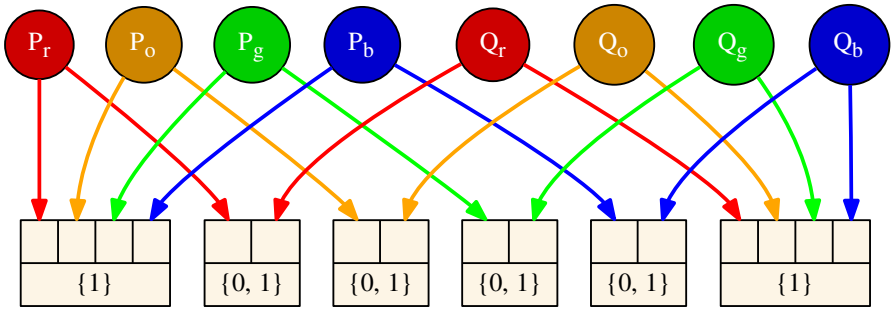


Fig. 8. Coloring two adjacent regions using NChooseK

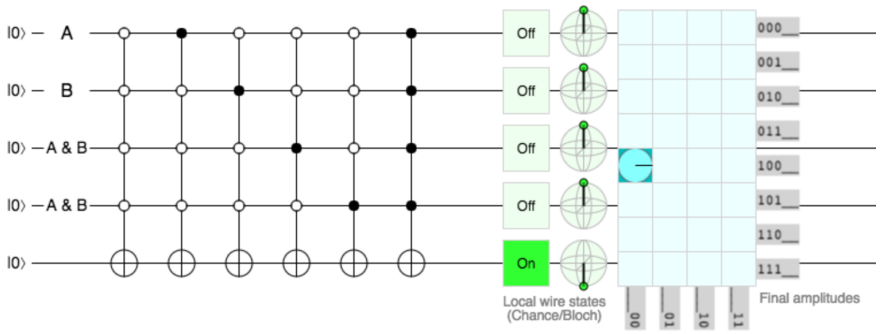
Q, and each is represented by four variables, one per color, yielding the eight variables P_r, P_o, P_g, P_b, Q_r, Q_o, Q_g, and Q_b. The four P variables connect to a block in Figure 7a while the four Q variables connect to a block in Figure 7a.

The P and Q "red" variables connect to a block in Figure 7b, and likewise for each of "orange", "green", and "blue".

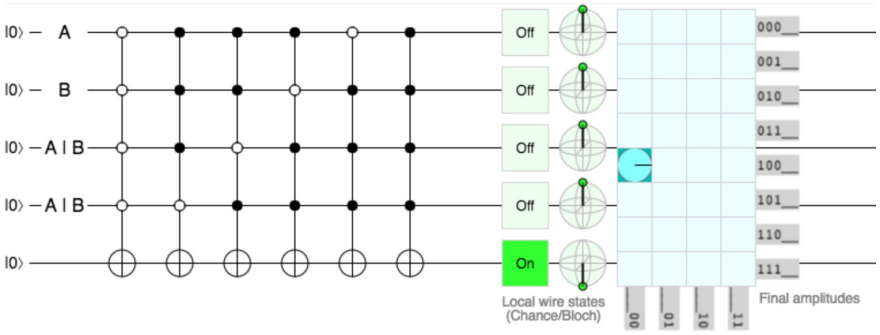
4 Implementation of the Code Generator

We implement a code generator for IBM Q Quantum Systems. It generates code for the IBM Qiskit API [17] given arbitrary N and K parameters for the NChooseK model. Our code generator then generates a complete program that can be executed on the IBM quantum simulator or actual quantum computing hardware.

To demonstrate how the code generator works, we first implement the basic logic gates, AND and OR, in Quirk [14]. Figure 9 depicts the implementation of the gates using Quirk for all $|0\rangle$ inputs (equivalent to FALSE in the following). Because Quirk allows the use of NOT gates with multiple controls and anti-controls, a circuit and its behavior can easily be visualized.



(a) AND of 6 NChooseK Combinations in Quirk



(b) OR of 6 NChooseK Combinations in Quirk

Fig. 9. Implementation of basic gates on Quirk

Figure 9a shows 6 conditions that would need to be addressed for the AND circuit. The first condition (all anti-controls) specifies that the output is TRUE if all inputs are FALSE. The next 4 conditions set the output to TRUE if one and only one of the inputs is TRUE (while the other 3 inputs are FALSE). The last condition sets the output to TRUE if all 4 inputs are TRUE. These 6 conditions correspond to the conditions required for the NChooseK primitive shown in Figure 5e. So when the output is TRUE, this circuit represents an AND circuit of NChooseK combinations. Similarly, Figure 9b uses 6 conditions to represent a multi-bit OR circuit of NChooseK combinations. In this case, the conditions represent those for the NChooseK primitive shown in Figure 5d. For both Figures 9a and 9b we need two qubits to represent A $\wedge$ B and A $\vee$ B, respectively. The truth table and intuition for this is described in Table 1.

4.1 Code Generation Example

In this section we present an example code generated by our code generator. We choose the example of $nck(\{A, B, C\}\{0, 2\})$, which implements the XOR function $C = A \oplus B$ using the NChooseK primitive. The generated code is shown below. We exclude the initialization and measurement code for brevity.

```
q = QuantumRegister(3)
qoutput = QuantumRegister(1)
c = ClassicalRegister(3)
coutput = ClassicalRegister(1)
qc = QuantumCircuit(q, qoutput, c, coutput)

def andInner(t, qx, qz, m, qc):
        if m == 1:
                qc.ccx(t[0], qx[0], qz[0])
        else:
                tmp = QuantumRegister(1)
                qc.add(tmp)
                qc.ccx(t[0], qx[m-1], tmp[0])
                andInner(tmp, qx, qz, m-1, qc)
                qc.ccx(t[0], qx[m-1], tmp[0])
        return qc

def and_nway(qx, qz, n, qc):
        if n == 1:
                qc.cx(qx[0], qz[0])
        else:
                if n == 2:
                        qc.ccx(qx[1], qx[0], qz[0])
                else:
                        t = QuantumRegister(1)
                        qc.add(t)
                        qc.ccx(qx[n-1], qx[n-2], t[0])
                        andInner(t, qx, qz, n-2, qc)
                        qc.ccx(qx[n-1], qx[n-2], t[0])
```

```
        return qc

#Creating equal superposition.
qc.h(q)

qc.x(q)
and_nway(q, qoutput, 3, qc)
qc.x(q)

qc.x(q[0])
and_nway(q, qoutput, 3, qc)
qc.x(q[0])

qc.x(q[1])
and_nway(q, qoutput, 3, qc)
qc.x(q[1])

qc.x(q[2])
and_nway(q, qoutput, 3, qc)
qc.x(q[2])

qc.measure(q, c)
```

Listing 1. XOR using NChooseK

The Python code creates a 3-qubit register, q for the inputs, a single qubit register, *qoutput* to represent the output. During measurement these registers map to their classical counterparts, c and *coutput*. The *andInner* and *and_nway* functions create the necessary circuit required to implement an n-input AND gate using CCNOT gates. Finally, we add the gates for the necessary conditions of each k, where $k \in \{0, 2\}$, i.e., the first condition for $k = 0$ and the last three conditions for $k = 2$.

4.2 Evaluation

Because contemporary quantum hardware, including the IBM Q, does not support controlling a single gate by multiple controls, we use CCNOT and X gates (provided by IBM's Qiskit API) to create complex, multi-control gates. While complex logical circuits can be created using these previously described AND and OR circuits, our code generator supports more expressive NChooseK circuits by combining simpler ones. A programmer can thus more effectively describe their computational problem. The CCNOT operation is an expensive operation because it is composed of 9 single qubit and 6 two-qubit gates [22]. Because the cost of the circuit is dominated by CCNOT operations, we focus on the number of CCNOT gates required for a particular NChooseK computation. We also assess the depth of the circuit for different values of N and K.

Figure 10 shows the number of CCNOT gates required for different combinations of N and k. We can see from the plot that the number of gates required is maximal when $k = \frac{N}{2}$, where $k \in K$. Also, the number of gates required

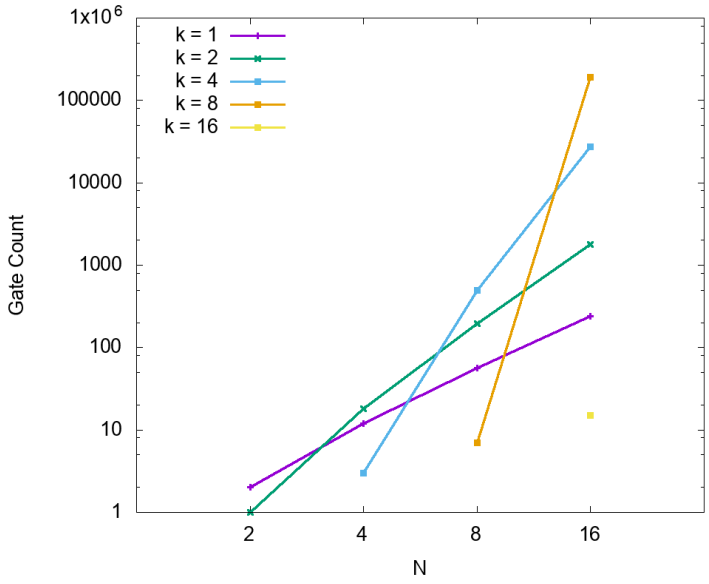


Fig. 10. Number of CCNOT gates required for arbitrary N and k

increases exponentially with N. So while more complex computation can be expressed with larger values of N, programmers need to establish a trade-off between the using simpler circuits, such as AND and OR, and expressing more complex computation with larger values of N.

Figure 11 indicates the depth of CCNOT gates required for different combinations of N and k. Similar to the previous figure, the depth of the circuit is maximal when $k = \frac{N}{2}$, where $k \in K$. These results further confirm the need for programmers to establish a trade-off between expressing computation in high-level NChooseK primitives vs. using several small NChooseK primitives to express the same computation.

5 Related Work

It is projected that the number of qubits will approach 50 or more in the next few years, yet we are still addressing the quantum programmability issue to reduce the challenge for programmers to express quantum computational problems effectively and effortlessly.

Several attempts were made to address this issue. The first attempt was made by Deutsch [11] to define notations of quantum Turing machines (QTM). A formalized quantum programming language proposal given by Knill [18] defined pseudo code for implementation on a quantum random access machine (QRAM) but was not precise enough to be implementable as a quantum programming language. The first real quantum computing language, QCL [20], was developed

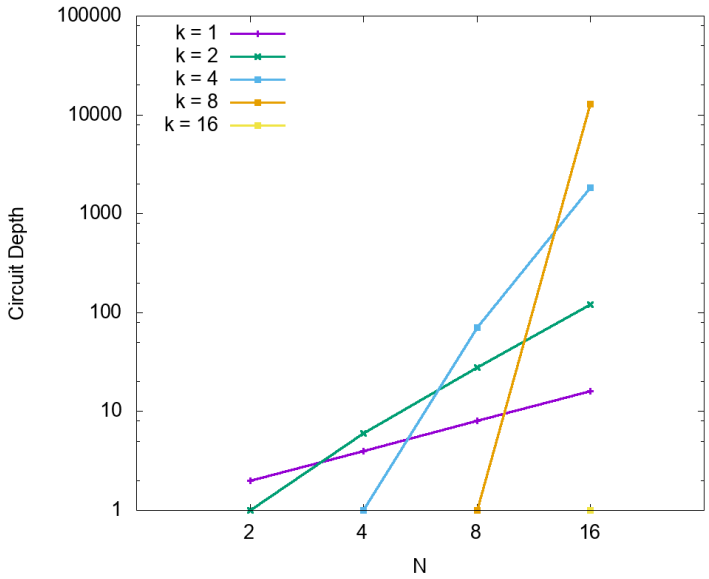


Fig. 11. Depth of circuit in CCNOT gates for arbitrary N and k

by Omer with syntax similar to C and provides a range of high level quantum programming features such as memory management and automatic derivation of conditional versions of operators. Another high-level language based on C++ was developed by Bettelli [1]. A quantum programming language based on probabilistic predictive programming, guarded-command language, quantum lambda calculus with operational semantics and an equational theory were defined. A first-order functional programming language, QML in which control as well as data suitable for quantum was defined. Quantum programming with Haskell by defining basic elements of quantum mechanics as data types and functions was also implemented. These solutions still failed to reduce the challenges faced to solve quantum computational problems efficiently [13].

Recently, many open-source quantum software projects are being developed and many major companies are trying to develop their own solutions. An open source framework Qiskit [17] was developed by IBM Research in 2017 for creating and manipulating quantum programs [7]. Qiskit uses the Python programming language to eventually translate a quantum programs to the OpenQASM [8] representation of circuits of quantum gates. Microsoft has defined a new programming language, Q#, with simulators working either on local systems or a cloud platform [19]. D-Wave Systems's Qbsolv solves QUBO problems on quantum processors as well as classical hardware architecture [9]. Our solution is different from these solutions as we are providing a way to express a variety of problems in a generalized model of computation called NChooseK. It makes the process of describing a problem efficient by requiring users to define problems

in terms of the model so that software can generate the code to execute the problem.

6 Conclusion and Future Work

In this work we present a novel model, NChooseK, for expressing quantum computation. We show how this model can be used to express computation for a circuit based universal quantum computer like the IBM Q. We demonstrate the generality of the programming model using 2 important applications, circuit satisfiability and map coloring. Finally, we describe the implementation of our code generator, which can generate Qiskit code for arbitrary inputs of N and K along with an example of the XOR gate. Our evaluation shows how the gate count and circuit depth is affected by different input parameters. In this context we discuss the trade-offs involved in using a single NChooseK primitive for more expressiveness vs. several smaller primitives to keep the gate count and circuit depth low.

We would like to further extend our code generator to combine multiple NChooseK primitives to express complex computational problems. The code generator/compiler can even explore the aforementioned trade-off space to automatically break down large NChooseK primitives into smaller more efficient subprimitives allowing the programmer to use larger, more expressive constructs. We are also looking at code generation of NChooseK primitives for quantum annealing systems such as the D-Wave.

Acknowledgments

Research presented in this article was supported in part by NSF grants 1525609 and 1813004 and by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project numbers 20160069DR and 20190065DR. This work was also supported by the U.S. Department of Energy through Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy (contract no. 89233218CNA000001).

References

1. Bettelli, S., Calarco, T., Serafini, L.: Toward an architecture for quantum programming. The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics **25**(2), 181–200 (2003)
2. Brennen, G.K., Caves, C.M., Jessen, P.S., Deutsch, I.H.: Quantum logic gates in optical lattices. Physical Review Letters **82**(5), 1060 (1999)
3. Cincio, Ł., Subaşı, Y., Sornborger, A.T., Coles, P.J.: Learning the quantum algorithm for state overlap. arXiv preprint arXiv:1803.04114 (2018)

4. Cirac, J.I., Zoller, P.: Quantum computations with cold trapped ions. Physical review letters **74**(20), 4091 (1995)
5. Clarke, J., Wilhelm, F.K.: Superconducting quantum bits. Nature **453**(7198), 1031 (2008)
6. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing. pp. 151–158. ACM (1971)
7. Cross, A.: The IBM Q experience and QISKit open-source quantum computing software. Bulletin of the American Physical Society **63**(1), BAPS.2018.MAR.L58.3 (2018)
8. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open quantum assembly language. arXiv:1707.03429 (2017), `http://arxiv.org/abs/1707.03429`
9. D-Wave Systems, Inc.: qbsolv, `https://docs.ocean.dwavesys.com/projects/qbsolv/`
10. Dahl, E.D.: Programming with D-Wave: Map coloring problem. D-Wave Official Whitepaper (2013)
11. Deutsch, D.: Quantum theory, the Church–Turing principle and the universal quantum computer. Proc. R. Soc. Lond. A **400**(1818), 97–117 (1985)
12. Feynman, R.P.: Simulating physics with computers. International Journal of Theoretical Physics **21**(6-7), 467–488 (1982)
13. Gay, S.J.: Quantum programming languages: Survey and bibliography. Mathematical Structures in Computer Science **16**(4), 581–600 (2006)
14. Gidney, C.: Quirk: A drag-and-drop quantum circuit simulator. `http://algassert.com/quirk`
15. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. pp. 212–219. ACM (1996)
16. IBM: IBM Q Experience. `https://quantumexperience.ng.bluemix.net/qx`
17. IBM: IBM Qiskit (2019), `https://qiskit.org/`
18. Knill, E.: Conventions for quantum pseudocode. Tech. Rep. LA-UR-96-2724, Los Alamos National Laboratory (Jun 1996)
19. Microsoft Research: Microsoft quantum development kit samples (2019), `https://github.com/Microsoft/Quantum`
20. Ömer, B.: A Procedural Formalism for Quantum Computing. Master's thesis, Department of Theoretical Physics, Technical University of Vienne (Jul 1998)
21. Schneider, S., Milburn, G.J.: Decoherence and fidelity in ion traps with fluctuating trap parameters. Physical Review A **59**(5), 3766 (1999)
22. Shende, V.V., Markov, I.L.: On the cnot-cost of toffoli gates. arXiv preprint arXiv:0803.2316 (2008)
23. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. pp. 124–134. IEEE (1994)
24. Younes, A.: Using Reed-Muller expansions in the synthesis and optimization of Boolean quantum circuits. In: Inspired by Nature, pp. 113–141. Springer (2018)