Fast quantum modular exponentiation

Rodney Van Meter* and Kohei M. Itoh

Graduate School of Science and Technology, Keio University and CREST-JST 3-14-1 Hiyoushi, Kohoku-ku, Yokohama-shi, Kanagawa 223-8522, Japan

We present a detailed analysis of the impact on quantum modular exponentiation of architectural features and possible concurrent gate execution. Various arithmetic algorithms are evaluated for execution time, potential concurrency, and space trade-offs. We find that to exponentiate an n-bit number, for storage space $100n$ (20 times the minimum $5n$), we can execute modular exponentiation 200–700 times faster than optimized versions of the basic algorithms, depending on architecture, for $n = 128$. Addition on a neighbor-only architecture is limited to $O(n)$ time, whereas non-neighbor architectures can reach $O(\log n)$, demonstrating that physical characteristics of a computing device have an important impact on both real-world running time and asymptotic behavior. Our results will help guide experimental implementations of quantum algorithms and devices.

I. INTRODUCTION

Research in quantum computing is motivated by the possibility of enormous gains in computational time [1–4]. The process of writing programs for quantum computers naturally depends on the architecture, but the application of classical computer architecture principles to the architecture of quantum computers has only just begun.

Shor's algorithm for factoring large numbers in polynomial time is perhaps the most famous result to date in the field [1]. Since this algorithm is well defined and important, we will use it as an example to examine the relationship between architecture and program efficiency, especially parallel execution of quantum algorithms. Shor's factoring algorithm consists of main two parts, quantum modular exponentiation, followed by the quantum Fourier transform. In this paper we will concentrate on the quantum modular exponentiation, both because it is the most computationally intensive part of the algorithm and because arithmetic circuits are fundamental building blocks we expect to be useful for many algorithms.

Fundamentally, quantum modular exponentiation is $O(n^3)$; that is, the number of quantum gates or operations scales with the cube of the length in bits of the number to be factored [5–7]. It consists of $2n$ modular multiplications, each of which consists of $O(n)$ additions, each of which requires $O(n)$ operations. However, $O(n^3)$ *operations* do not necessarily require $O(n^3)$ *time steps*. On an abstract machine, it is relatively straightforward to see how to reduce each of those three layers to $O(\log n)$ time steps, in exchange for more space and more *total* gates, giving a total running time of $O(\log^3 n)$ if $O(n^3)$ qubits are available and an arbitrary number of gates can be executed concurrently on separate qubits. Such large numbers of qubits are not expected to be practical for the foreseeable future, so much interesting engineering lies in optimizing for a given set of constraints.

This paper quantitatively explores those trade-offs.

This paper is intended to help guide the design and experimental implementation of actual quantum computing devices as the number of qubits grows over the next several generations of devices. Depending on the postquantum error correction, application-level effective clock rate for a specific technology, the choice of exponentiation algorithm may be the difference between hours of computation time and weeks or between seconds and hours. This difference, in turn, feeds back into the system requirements for the necessary strength of error correction and coherence time.

The Schönhage-Strassen multiplication algorithm is often quoted in quantum computing research as being $O(n \log n \log \log n)$ for a single multiplication [8]. However, simply citing Schönhage-Strassen without further qualification is misleading for several reasons. Most importantly, the constant factors matter:[1] quantum modular exponentiation based on Schönhage-Strassen is only faster than basic $O(n^3)$ algorithms for more than ~ 32 *kilobits*. In this paper, we will concentrate on smaller problem sizes, and exact, rather than $O(\cdot)$, performance.

Concurrent quantum computation is the execution of more than one quantum gate on independent qubits at the same time. Utilizing concurrency, the latency, or circuit depth, to execute a number of gates can be smaller than the number itself. Circuit depth is explicitly considered in Cleve and Watrous' parallel implementation of the quantum Fourier transform [9], Gossett's quantum carry-save arithmetic [10], and Zalka's Schönhage-Strassen-based implementation [11]. Moore and Nilsson define the computational complexity class QNC to describe certain parallelizable circuits and show which gates can be performed concurrently, proving that any circuit composed exclusively of controlled NOT gates (CNOTs) can be parallelized to be of depth $O(\log n)$ using $O(n^2)$ ancillae on an abstract machine [12].

---

[1]Shor noted this in his original paper, without explicitly specifying a bound. Note also that this bound is for a Turing machine; a random-access machine can reach $O(n \log n)$.

---

*Electronic address: rdv@tera.ics.keio.ac.jp
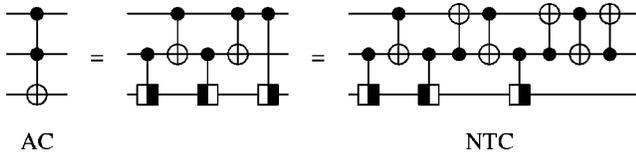
FIG. 1. CCNOT constructions for our architectures AC and NTC. The box with the bar on the right represents the square root of X, and the box with the bar on the left its adjoint. Time flows left to right, each horizontal line represents a qubit, and each vertical line segment is a quantum gate.

We analyze two separate architectures, still abstract but with some important features that help us understand performance. For both architectures, we assume any qubit can be the control or target for only one gate at a time. The first, *abstract concurrent* (AC) architecture, is our abstract model. It supports CCNOT (the three-qubit Toffoli gate, or controlled-controlled-NOT), arbitrary concurrency, and gate operands any distance apart without penalty. It does not support arbitrary control strings on control operations, only CCNOT with two ones as control. The second, the *neighbor-only, two-qubit-gate, concurrent* (NTC) architecture, is similar but does not support CCNOT, only two-qubit gates, and assumes the qubits are laid out in a one-dimensional (1D) line, and only neighboring qubits can interact. The 1D layout will have the highest communications costs among possible physical topologies. Most real, scalable architectures will have constraints with this flavor, if different details, so AC and NTC can be viewed as bounds within which many real architectures will fall. The layout of variables on this structure has a large impact on performance; what is presented here is the best we have discovered to date, but we do not claim it is optimal.

The NTC model is a reasonable description of several important experimental approaches, including a one-dimensional chain of quantum dots [13], the original Kane proposal [14], and the all-silicon NMR device [15]. Superconducting qubits [16,17] may map to NTC, depending on the details of the qubit interconnection.

The difference between AC and NTC is critical; beyond the important constant factors as nearby qubits shuffle, we will see in Sec. III B that AC can achieve $O(\log n)$ performance where NTC is limited to $O(n)$.

For NTC, which does not support CCNOT directly, we compose CCNOT from a set of five two-qubit gates [18], as shown in Fig. 1. The box with the bar on the right represents the square root of X,

$$\sqrt{X} = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$$

and the box with the bar on the left its adjoint. We assume that this gate requires the same execution time as a CNOT.

Section II reviews Shor's algorithm and the need for modular exponentiation, then summarizes the techniques we employ to accelerate modular exponentiation. Section II A introduces the best-known existing modular exponentiation algorithms and several different adders. Section III begins by examining concurrency in the lowest level elements, the

adders. This is followed by faster adders and additional techniques for accelerating modulo operations and exponentiation. Section IV shows how to balance these techniques and apply them to a specific architecture and set of constraints. We evaluate several complete algorithms for our architectural models. Specific gate latency counts, rather than asymptotic values, are given for 128 bits and smaller numbers.

II. BASIC CONCEPTS

A. Modular exponentiation and Shor's algorithm

Shor's algorithm for factoring numbers on a quantum computer uses the quantum Fourier transform to find the order r of a randomly chosen number x in the multiplicative group (mod N). This is achieved by exponentiating x, modulo N, for a superposition of all possible exponents a. Therefore, efficient arithmetic algorithms to calculate modular exponentiation in the quantum domain are critical.

Quantum modular exponentiation is the evolution of the state of a quantum computer to hold

$$|\psi\rangle|0\rangle \rightarrow |\psi\rangle|x^\psi \bmod N\rangle. \tag{1}$$

When $|\psi\rangle$ is the superposition of all input states a up to a particular value $2N^2$,

$$|\psi\rangle = \frac{1}{N\sqrt{2}} \sum_{a=0}^{2N^2} |a\rangle. \tag{2}$$

The result is the superposition of the modular exponentiation of those input states,

$$\frac{1}{N\sqrt{2}} \sum_{a=0}^{2N^2} |a\rangle|0\rangle \rightarrow \frac{1}{N\sqrt{2}} \sum_{a=0}^{2N^2} |a\rangle|x^a \bmod N\rangle. \tag{3}$$

Depending on the algorithm chosen for modular exponentiation, x may appear explicitly in a register in the quantum computer or may appear only implicitly in the choice of instructions to be executed. In general, quantum modular exponentiation algorithms are created from building blocks that do modular multiplication,

$$|\alpha\rangle|0\rangle \rightarrow |\alpha\rangle|\alpha\beta \bmod N\rangle, \tag{4}$$

where β and N may or may not appear explicitly in quantum registers. This modular multiplication is built from blocks that perform modular addition,

$$|\alpha\rangle|0\rangle \rightarrow |\alpha\rangle|\alpha + \beta \bmod N\rangle, \tag{5}$$

which, in turn, are usually built from blocks that perform addition and comparison.

Addition of two n-bit numbers requires $O(n)$ gates. Multiplication of two n-bit numbers (including modular multiplication) combines the convolution partial products (the one-bit products) of each pair of bits from the two arguments. This requires $O(n)$ additions of n-bit numbers, giving a gate count of $O(n^2)$. Our exponentiation for Shor's algorithm requires $2n$ multiplications, giving a total cost of $O(n^3)$.

Many of these steps can be conducted in parallel; in clas-

TABLE I. Parameters for our algorithms, chosen for 128 bits.

Algorithm	Adder	Modulo	Indirect	Multipliers (s)	Space	Concurrency
Concurrent VBE	VBE	VBE	N/A	1	897	2
Algorithm **D**	CSUM($m=4$)	$p=11$, $b=1024$	$w=2$	12	11969	$126 \times 12 = 1512$
Algorithm **E**	QCLA	$p=10$, $b=512$	$w=2$	16	12657	$128 \times 16 = 2048$
Algorithm **F**	CDKM	$p=10$, $b=512$	$w=4$	20	11077	$20 \times 2 = 40$
Algorithm **G**	CDKM	Fig. 7	$w=4$	1	660	2

sical computer system design, the *latency* or *circuit depth*, the time from the input of values until the output becomes available, is as important as the total computational complexity. *Concurrency* is the execution of more than one gate during the same execution time slot. We will refer to the number of gates executing in a time slot as the concurrency or the concurrency level. Our goal through the rest of the paper is to exploit parallelism, or concurrency, to shorten the total wall clock time to execute modular exponentiation and, hence, Shor's algorithm.

The algorithms as described here run on logical qubits, which will be encoded onto physical qubits using quantum error correction (QEC) [19]. Error correction processes are generally assumed to be applied in parallel across the entire machine. Executing gates on the encoded qubits, in some cases, requires additional ancillae, so multiple concurrent logical gates will require growth in physical qubit storage space [20,21]. Thus, both physical and logical concurrency are important; in this paper we consider only logical concurrency.

B. Notation and techniques for speeding up modular exponentation

In this paper, we will use N as the number to be factored and n to represent its length in bits. For convenience, we will assume that n is a power of two and the high bit of N is one. x is the random value, smaller than N, to be exponentiated, and $|a\rangle$ is our superposition of exponents, with $a < 2N^2$ so that the length of a is $2n+1$ bits.

When discussing circuit cost, the notation is (*CCNOTs*; *CNOTs*; *NOTs*) or (*CNOTS*; *NOTs*). The values may be total gates or circuit depth (latency), depending on context. The notation is sometimes enhanced to show required concurrency and space, (*CCNOTs*; *CNOTs*; *NOTs*)#(*concurrency*; *space*) (where # is used as a separator). t is time, or latency to execute an algorithm, and S is space, subscripted with the name of the algorithm or circuit subroutine. When t or S is superscripted with AC or NTC, the values are for the latency of the construct on that architecture. Equations without superscripts are for an abstract machine assuming no concurrency, equivalent to a total gate count for the AC architecture. R is the number of calls to a subroutine, subscripted with the name of the routine.

m, g, f, p, b, and s are parameters that determine the behavior of portions of our modular exponentiation algorithm. m, g, and f are part of our carry-select and conditional-sum adders (Sec. III B). p and b are used in our

indirection scheme (Sec. III E). s is the number of multiplier blocks we can fit into a chosen amount of space (Sec. III C).

Here we summarize the techniques, which are detailed in following sections. Our fast modular exponentiation circuit is built using the following optimizations: (i) Select correct qubit layout and subsequences to implement gates, then hand optimize (no penalty) [22–28]. (ii) Look for concurrency within addition and/or multiplication (no space penalty, maybe noise penalty) (Secs. III A). (iii) Select multiplicand using table and/or indirection (exponential classical cost, linear reduction in quantum gate count) ([29], Sec. III E). (iv) Do multiplications concurrently (linear speedup for small values, linear cost in space, small gate count increase; requires quantum-quantum (Q-Q) multiplier, as well as classical-quantum (C-Q) multiplier) (Sec. III C). (v) Move to e.g., carry-save adders (n^2 space penalty for reduction to log time, increases total gate count)([10], Sec. II C 4) conditional-sum adders (Sec. III B 2), or carry–look-ahead adders (Sec. II C 5). (vi) Reduce modulo comparisons, only do subtract N on overflow (small space penalty, linear reduction in modulo arithmetic cost) (Sec. III D).

C. Existing algorithms

In this section we will review various components of the modular exponentiation that will be used to construct our parallelized version of the algorithm in Sec. III. There are many ways of building adders and multipliers, and choosing the correct one is a technology-dependent exercise [30]. Only a few classical techniques have been explored for quantum computation. The two most commonly cited modular exponentiation algorithms are those of Vedral *et al.* [7], which we will refer to as VBE, and Beckman *et al.* [5], which we will refer to as BCDP. Both BCDP and VBE algorithms build multipliers from variants of carry-ripple adders, the simplest but slowest method; Cuccaro *et al.* [31] have recently shown the design of a smaller, faster carry-ripple adder. Zalka proposed a carry-select adder; we present our design for such an adder in detail in Sec. III B. Draper *et al.* [32] have recently proposed a carry–look-ahead adder and Gossett a carry-save adder. Beauregard [33] has proposed a circuit that operates primarily in the Fourier transform space.

Carry–look-ahead (Sec. II C 5), conditional-sum (Sec. III B 2), and carry-save (Sec. II C 4) adders all reach $O(\log n)$ performance for addition. Carry–look-ahead and conditional-sum adders use more space than carry-ripple adders, but much less than carry-save adders. However, carry-save adders can be combined into fast multipliers more

easily. We will see in Sec. III how to combine carry–look-ahead and conditional-sum adders into the overall exponentiation algorithms.

1. VBE carry-ripple adder

The VBE algorithm [7] builds full modular exponentiation from smaller building blocks. The bulk of the time is spent in $20n^2 - 5n$ ADDERs.[2] The full circuit requires $7n + 1$ qubits of storage: $2n+1$ for a, n for the other multiplicand, n for a running sum, n for the convolution products, n for a copy of N, and n for carries.

In this algorithm, the values to be added in, the convolution partial products of x^a, are programed into a temporary register (combined with a superposition of $|0\rangle$ as necessary) based on a control line and a data bit via appropriate CCNOT gates. The latency of ADDER and the complete algorithm are

$$t_{ADD} = (4n - 4; 4n - 3; 0) \tag{6}$$

$$t_V = (20n^2 - 5n)t_{ADD} = (80n^3 - 100n^2 + 20n; 96n^3 - 84n^2 + 15n; 8n^2 - 2n + 1). \tag{7}$$

2. BCDP carry-ripple adder

The BCDP algorithm is also based on a carry-ripple adder. It differs from VBE in that it more aggressively takes advantage of classical computation. However, for our purposes, this makes it harder to use some of the optimization techniques presented here. Beckman *et al.* [5] present several optimizations and trade-offs of space and time, slightly complicating the analysis.

The exact sequence of gates to be applied is also dependent on the input values of N and x, making it less suitable for hardware implementation with fixed gates (e.g., in an optical system). In the form we analyze, it requires $5n + 3$ qubits, including $2n+1$ for $|a\rangle$. Borrowing from their Eq. (6.23),

$$t_B = (54n^3 - 127n^2 + 108n - 29; 10n^3 + 15n^2 - 38n + 14; 20n^3 - 38n^2 + 22n - 4). \tag{8}$$

3. CDKM carry-ripple adder

Cuccaro *et al.* have recently introduced a carry-ripple circuit, which we will call CDKM, which uses only a single ancilla qubit [31]. The latency of their adder is $(2n - 1; 5; 0)$ for the AC architecture.

The authors do not present a complete modular exponentiation circuit; we will use their adder in our algorithms **F** and **G**. This adder, we will see in Sec. IV C 1, is the most efficient known for NTC architectures.

4. Gossett carry-save and carry-ripple adders

Gossett's arithmetic is pure quantum, as opposed to the mixed classical-quantum of BCDP. Gossett does not provide

a full modular exponentiation circuit, only adders, multipliers, and a modular adder based on the important classical techniques of *carry-save arithmetic* [10].

Gossett's carry-save adder, the important contribution of the paper, can run in $O(\log n)$ time on AC architectures. It will remain impractical for the foreseeable future because of the large number of qubits required; Gossett estimates $8n^2$ qubits for a full multiplier, which would run in $O(\log^2 n)$ time. It bears further analysis because of its high speed and resemblance to standard fast classical multipliers.

Unfortunately, the paper's second contribution, Gossett's carry-ripple adder, as drawn in his Fig. 7, seems to be incorrect. Once fixed, his circuit optimizes to be similar to VBE.

5. Carry–look-ahead adder

Draper *et al.* have recently proposed a carry–look-ahead adder, which we call QCLA [32]. This method allows the latency of an adder to drop to $O(\log n)$ for AC architectures. The latency and storage of their adder is

$$t_{LA}^{AC} = (4 \log_2 n + 3; 4; 2)\#(n; 4n - \log n - 1). \tag{9}$$

The authors do not present a complete modular exponentiation circuit; we will use their adder in our algorithm **E**, which we evaluate only for AC. The large distances between gate operands make it appear that QCLA is unattractive for NTC.

6. Beauregard-Draper QFT-based exponentiation

Beauregard has designed a circuit for doing modular exponentiation in only $2n+3$ qubits of space [33], based on Draper's clever method for doing addition on Fourier-transformed representations of numbers [34].

The depth of Beauregard's circuit is $O(n^3)$, the same as VBE and BCDP. However, we believe the constant factors on this circuit are very large; every modulo addition consists of four Fourier transforms and five Fourier additions.

Fowler *et al.* [35] and Devitt *et al.* [36] have simulated Shor's algorithm using Beauregard's algorithm, for a class of machines they call *linear nearest neighbor* (LNN). LNN corresponds approximately to our NTC. In their implementation of the algorithm, they found no significant change in the computational complexity of the algorithm on LNN or an AC-like abstract architecture, suggesting that the performance of Draper's adder, like a carry-ripple adder, is essentially architecture independent.

III. RESULTS: ALGORITHMIC OPTIMIZATIONS

We present our concurrent variant of VBE, then move to faster adders. This is followed by methods for performing exponentiation concurrently, improving the modulo arithmetic and indirection to reduce the number of quantum multiplications.

A. Concurrent VBE

In Fig. 2, we show a three-bit concurrent version of the VBE ADDER. This figure shows that the delay of the concur-

---

[2]When we write ADDER in all small capital letters, we mean the complete VBE n-bit construction, with the necessary undo; when we write adder in small letters, we are usually referring to a smaller or generic circuit block.
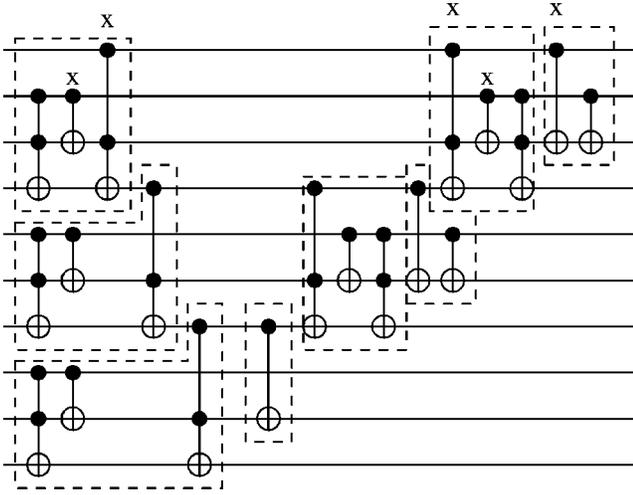
FIG. 2. Three-bit concurrent VBE ADDER, AC abstract machine. Gates marked with an x can be deleted when the carry in is known to be zero.

rent ADDER is $(3n-3)t_{CCNOT}+(2n-3)t_{CNOT}$, or

$$t_{ADD}^{AC} = (3n-3;2n-3;0),\qquad(10)$$

a mere 25% reduction in latency compared to the unoptimized $(4n-4;4n-3;0)$ of Eq. (6).

Adapting Eq. (7), the total circuit latency, minus a few small corrections that fall outside the ADDER block proper, is

$$t_V^{AC} = (20n^2 - 5n)t_{ADD}^{AC} = (60n^3 - 75n^2 + 15n;40n^3 - 70n^2$$
$$+ 15n;0).\qquad(11)$$

This equation is used to create the first entry in Table II.

B. Carry-select and conditional-sum adders

Carry-select adders concurrently calculate possible results without knowing the value of the carry in. Once the carry in becomes available, the correct output value is selected using a multiplexer (MUX). The type of MUX determines whether the behavior is $O(\sqrt{n})$ or $O(\log n)$.

1. $O(\sqrt{n})$ carry-select adder

The bits are divided into g groups of m bits each, $n=gm$. The adder block we will call CSLA, and the combined adder,
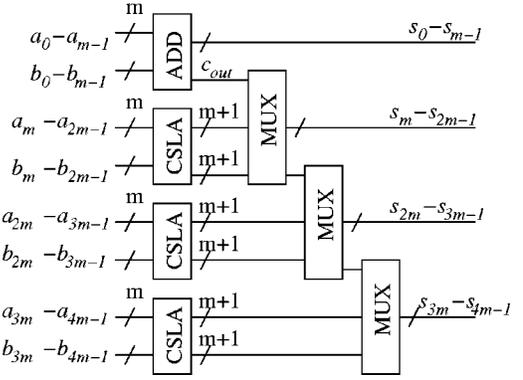
FIG. 3. Block-level diagram of four-group carry-select adder. a_i and b_i are addends, and s_i is the sum. Additional ancillae not shown.

MUXes, and adder undo to clean our ancillae, CSLAMU. The CSLAs are all executed concurrently, then the output MUXes are cascaded, as shown in Fig. 3. The first group may have a different size, f, than m, since it will be faster, but for the moment we assume they are the same.

Figure 4 shows a three-bit carry-select adder. This generates two possible results, assuming that the carry in will be zero or one. The portion on the right is a MUX used to select which carry to use, based on the carry in. All of the outputs without labels are ancillae to be garbage collected. It is possible that a design optimized for space could reuse some of those qubits; as drawn a full carry-select circuit requires $5m-1$ qubits to add two m-bit numbers.

The larger m-bit carry-select adder can be constructed so that its internal delay, as in a normal carry-ripple adder, is one additional CCNOT for each bit, although the total number of gates increases and the distance between gate operands increases.

The latency for the CSLA block is

$$t_{CS}^{AC} = (m;2;0).\qquad(12)$$

Note that this is not a "clean" adder; we still have ancillae to return to the initial state.

The problem for implementation will be creating an efficient MUX, especially on NTC. Figure 3 makes it clear that the total carry-select adder is only faster if the latency of MUX is substantially less than the latency of the full carry-

TABLE II. Latency to factor a 128-bit number for various architectures and choices of algorithm: AC, abstract concurrent architecture; NTC, neighbor-only, two-qubit gate, concurrent architecture; and Perf, performance relative to VBE algorithm for that architecture, based on CCNOTs for AC and CNOTs for NTC.

	AC		NTC	
Algorithm	Gates	Perf.	Gates	Perf.
Concurrent VBE	$(1.25\times10^8;8.27\times10^7;0.00\times10^0)$	1.0	$(8.32\times10^8;0.00\times10^0)$	1.0
Algorithm **D**	$(2.19\times10^5;2.57\times10^4;1.67\times10^5)$	569.8	N/A	N/A
Algorithm **E**	$(1.71\times10^5;1.96\times10^4;2.93\times10^4)$	727.2	N/A	N/A
Algorithm **F**	$(7.84\times10^5;1.30\times10^4;4.10\times10^4)$	158.9	$(4.11\times10^6;4.10\times10^4)$	202.5
Algorithm **G**	$(1.50\times10^7;2.48\times10^5;7.93\times10^5)$	8.3	$(7.87\times10^7;7.93\times10^5)$	10.6
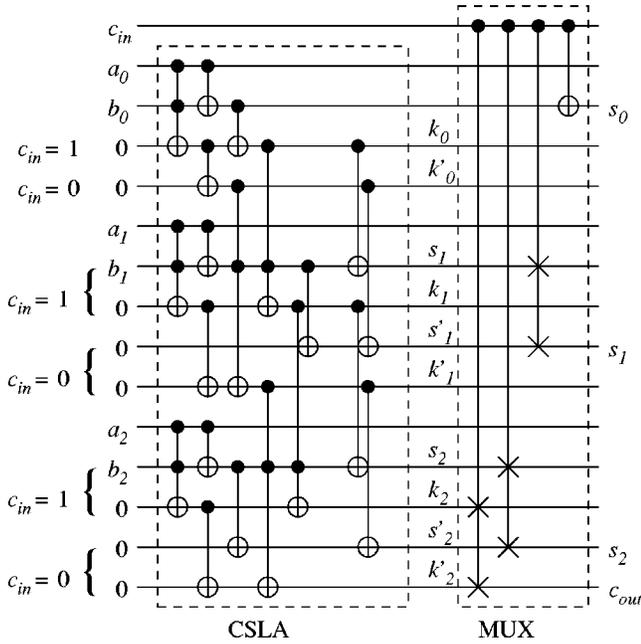
FIG. 4. Three-bit carry-select adder (CSLA) with multiplexer (MUX). a_i and b_i are addends. The control-SWAP gates in the MUX select either the qubits marked $c_{in}=1$ or $c_{in}=0$, depending on the state of the carry-in qubit c_{in}. s_i qubits are the output sum, and k_i are internal carries.

ripple. It will be difficult for this to be more efficient that the single-CCNOT delay of the basic VBE carry-ripple adder on NTC. On AC, it is certainly easy to see how the MUX can use a fanout tree consisting of more ancillae and CNOT gates to distribute the carry in signal, as suggested by Moore [12], allowing all MUX Fredkin gates to be executed concurrently. A full fanout requires an extra m qubits in each adder.

In order to unwind the ancillae to reuse them, the simplest approach is the use of CNOT gates to copy our result to another n-bit register, then a reversal of the circuitry. Counting the copy out for ancilla management, we can simplify the MUX to two CCNOTs and a pair of NOTs.

The latency of the carry ripple from MUX to MUX (not qubit to qubit) can be arranged to give a MUX cost of $(4g+2m-6;0;2g-2)$. This cost can be accelerated somewhat by using a few extra qubits and "fanning out" the carry. For intermediate values of m, we will use a fanout of 4 on AC, reducing the MUX latency to $(4g+m/2-6;2;2g-2)$ in exchange for three extra qubits in each group.

Our space used for the full, clean adder is $(6m-1)(g-1)+3f+4g$ when using a fanout of 4. The total latency of the CSLA, MUX, and the CSLA undo is

$$t_{SEM}^{AC} = 2t_{CS}^{AC} + t_{MUX}^{AC} = (4g + 5m/2 - 6; 6; 2g - 2). \quad (13)$$

Optimizing for AC, based on Eq. (13), the delay will be the minimum when $m \sim \sqrt{8n/5}$.

Zalka was the first to propose use of a carry-select adder, though he did not refer to it by name [11]. His analysis does not include an exact circuit, and his results differ slightly from ours.
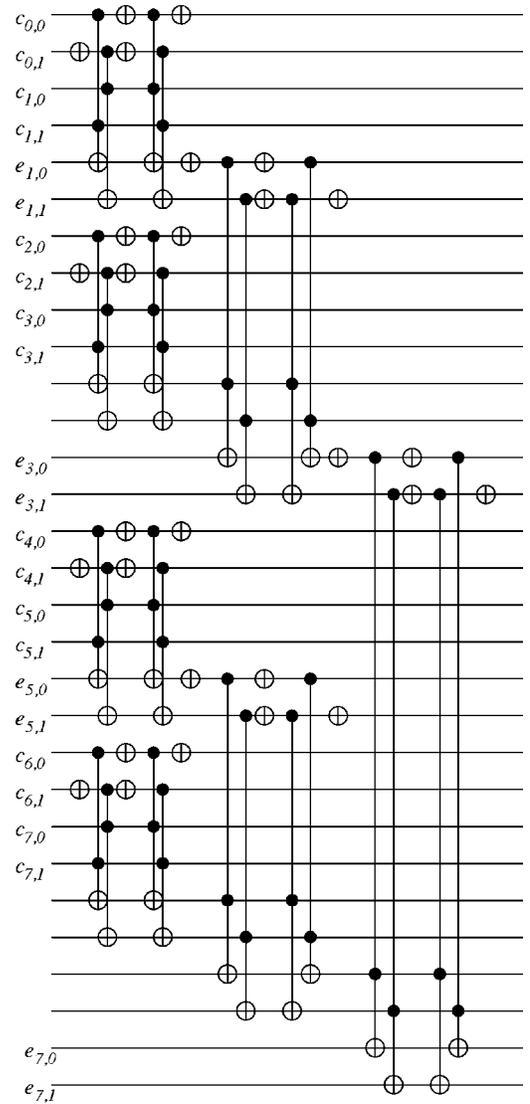
FIG. 5. $O(\log n)$ MUX for conditional-sum adder, for $g=9$ (the first group is not shown). Only the $c_{i,j}$ carry out lines from each m-qubit block are shown, where i is the block number and j is the carry in value. At each stage, the span of correct effective swap control lines $e_{i,j}$ doubles. After using the swap control lines, all but the last must be cleaned by reversing the circuit. Unlabeled lines are ancillae to be cleaned.

2. O(log n) conditional-sum adder

As described above, the carry-select adder is $O(m+g)$, for $n=mg$, which minimizes to be $O(\sqrt{n})$. To reach $O(\log n)$ performance, we must add a multilevel MUX to our carry-select adder. This structure is called a conditional-sum adder, which we will label CSUM. Rather than repeatedly choosing bits at each level of the MUX, we will create a multilevel distribution of MUX select signals, then apply them once at the end. Figure 5 shows only the carry signals for eight CSLA groups. The e signals in the figure are our effective swap control signals. They are combined with a carry in signal to control the actual swap of variables. In a full circuit, a ninth group, the first group, will be a carry-ripple adder and will create the carry in; that carry in will be distributed concurrently in a separate tree.
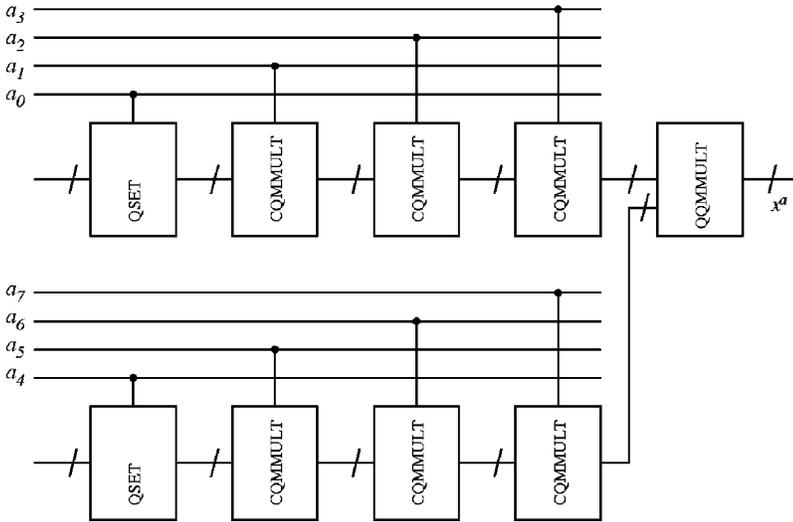
FIG. 6. Concurrent modular multiplication in modular exponentiation for $s=2$. QSET simply sets the sum register to the appropriate value.

The total adder latency will be

$$t_{\text{CSUM}}^{\text{AC}} = 2t_{\text{CS}}^{\text{AC}} + [2\lceil\log_2(g-1)\rceil - 1] \times (2;0;2) + (4;0;4)$$
$$= (2m + 4\lceil\log_2(g-1)\rceil + 2; 4; 4\lceil\log_2(g-1)\rceil + 2),$$

(14)

where $\lceil x \rceil$ indicates the smallest integer not smaller than x. For large n, this generally reaches a minimum for small m, which gives asymptotic behavior $\sim 4\log_2 n$, the same as QCLA. CSUM is noticeably faster for small n, but requires more space.

The MUX uses $\lceil 3(g-1)/2\rceil - 2$ qubits in addition to the internal carries and the tree for dispersing the carry in. Our space used for the full, clean adder is $(6m-1)(g-1)+3f +\lceil 3(g-1)/2 - 2 + (n-f)/2\rceil$.

C. Concurrent exponentiation

Modular exponentiation is often drawn as a string of modular multiplications, but Cleve and Watrous pointed out that these can easily be parallelized, at linear cost in space [9] (see Fig. 6). We always have to execute $2n$ multiplications; the goal is to do them in as few time delays as possible.

To go (almost) twice as fast, use two multipliers; for four times, use four. Naturally, this can be built up to n multipliers to multiply the necessary $2n+1$ numbers, in which case a tree recombining the partial results requires $\log_2 n$ quantum-quantum (Q-Q) multiplier latency times. The first unit in each chain just sets the register to the appropriate value if the control line is 1, otherwise, it leaves it as 1.

For s multipliers, $s \leqslant n$, each multiplier must combine $r = \lfloor(2n+1)/s\rfloor$ or $r+1$ numbers, using $r-1$ or r multiplications (the first number being simply set into the running product register), where $\lfloor x \rfloor$ indicates the largest integer not larger than x. The intermediate results from the multipliers are combined using $\lceil\log_2 s\rceil$ Q-Q multiplication steps.

For a parallel version of VBE, the exact latency, including cases where $rs \neq 2n+1$, is

$$R_V = 2r + 1 + \lceil\log_2[\lceil(s - 2n - 1 + rs)/4\rceil + 2n + 1 - rs]\rceil$$

(15)

times the latency of our multiplier. For small s, this is $O(n)$; for larger s,

$$\lim_{s \to n} O(n/s + \log s) = O(\log n).$$

(16)

D. Reducing the cost of modulo operations

The VBE algorithm does a trial subtraction of N in each modulo addition block; if that underflows, N is added back in to the total. This accounts for two of the five ADDER blocks and much of the extra logic to compose a modulo adder. The last two of the five blocks are required to undo the overflow bit.

Figure 7 shows a more efficient modulo adder than VBE, based partly on ideas from BCDP and Gossett. It requires only three adder blocks, compared to five for VBE, to do one modulo addition. The first adder adds x^j to our running sum. The second conditionally adds $2^n - x^j - N$ or $2^n - x^j$, depending on the value of the overflow bit, *without* affecting the overflow bit, arranging it so that the third addition of x^j will overflow and clear the overflow bit if necessary. The blocks pointed to by arrows are the addend register, whose value is set depending on the control lines. Figure 7 uses n fewer qubits than VBE's modulo arithmetic, as it does not require a register to hold N.

In a slightly different fashion, we can improve the performance of VBE by adding a number of qubits p to our result register and postponing the modulo operation until later. This
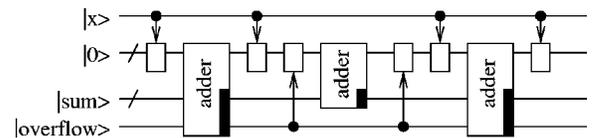
FIG. 7. More efficient modulo adder. The blocks with arrows set the register contents based on the value of the control line. The position of the black block indicates the running sum in our output.
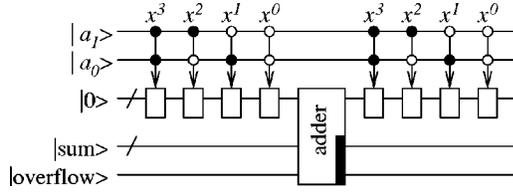
FIG. 8. Implicit indirection. The arrows pointing to blocks indicate the setting of the addend register based on the control lines. This sets the addend from a table stored in classical memory, reducing the number of quantum multiplications by a factor of w in exchange for 2^w argument setting operations.

works as long as we do not allow the result register to overflow; we have a redundant representation of modulo N values, but that is not a problem at this stage of the computation.

The largest number that does not overflow for p extra qubits is $2^{n+p}-1$; the largest number that does not result in subtraction is $2^{n+p-1}-1$. We want to guarantee that we always clear that high-order bit, so if we subtract bN, the most iterations we can go before the next subtraction is b. The largest multiple of N we can subtract is $\lfloor 2^{n+p-1}/N \rfloor$. Since $2^{n-1} < N < 2^n$, the largest b we can allow is, in general, 2^{p-1}.

For example, adding three qubits, $p=3$, allows $b=4$, reducing the 20 ADDER calls VBE uses for four additions to nine ADDER calls, a 55% performance improvement. As p grows larger, the cost of the adjustment at the end of the calculation also grows and the additional gains are small. We must use $3p$ adder calls at the end of the calculation to perform our final modulo operation. Calculations suggest that p of up to 10 or 11 is still faster.

Equation (17) shows the number of calls to our adder block necessary to make an n-bit modulo multiplier.

$$R_M = n(2b+1)/b. \qquad (17)$$

E. Indirection

We have shown elsewhere that it is possible to build a table containing small powers of x, from which an argument to a multiplier is selected [29]. In exchange for adding storage space for 2^w n-bit entries in a table, we can reduce the number of multiplications necessary by a factor of w. This appears to be attractive for small values of w, such as 2 or 3.

In our prior work, we proposed using a large quantum memory, or a quantum-addressable classical memory (QACM) [37]. Here we show that the quantum storage space need not grow; we can implicitly perform the lookup by choosing which gates to apply while setting the argument. In Fig. 8, we show the setting and resetting of the argument for $w=2$, where the arrows indicate CCNOTs to set the appropriate bits of the 0 register to 1. The actual implementation can use a calculated enable bit to reduce the CCNOTs to CNOTs. Only one of the values x^0, x^1, x^2, or x^3 will be enabled, based on the value of $|a_1 a_0\rangle$.

The setting of this input register may require propagating $|a\rangle$ or the enable bit across the entire register. Use of a few extra qubits (2^{w-1}) will allow the several setting operations to propagate in a tree
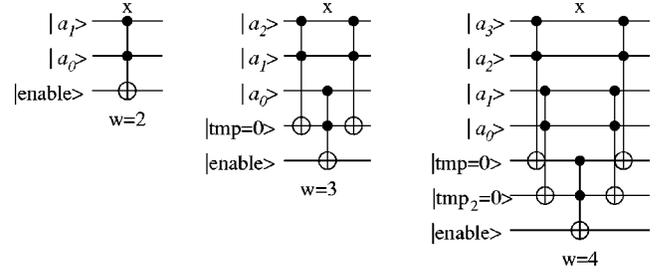
FIG. 9. Argument setting for indirection for different values of w, for the AC architecture. For the $w=4$ case, the two CCNOTs on the left can be executed concurrently, as can the two on the right, for a total latency of 3.

$$t_{\text{ARG}}^{\text{AC}} = \begin{cases} 2^w(1;0;1) = (4;0;4) & w=2 \\ 2^w(3;0;1) & w=3,4 \end{cases}. \qquad (18)$$

For $w=2$ and $w=3$, we calculate that setting the argument adds $(4;0;4)\#(4,5)$ and $(24;0;8)\#(8,9)$, respectively, to the latency, concurrency and storage of each adder. We create separate enable signals for each of the 2^w possible arguments and pipeline flowing them across the register to set the addend bits. We consider this cost only when using indirection. Figure 9 shows circuits for $w=2,3,4$.

Adapting Eq. (15) to both indirection and concurrent multiplication, we have a total latency for our circuit, in multiplier calls, of

$$R_I = 2r + 1 + \lceil \log_2[\lceil(s-2n-1+rs)/4\rceil + 2n+1-rs] \rceil, \qquad (19)$$

where $r = \lceil \lceil(2n+1)/w\rceil/s \rceil$.

IV. EXAMPLE: EXPONENTIATING A 128-BIT NUMBER

In this section, we combine these techniques into complete algorithms and examine the performance of modular exponentiation of a 128-bit number. We assume the primary engineering constraint is the available number of qubits. In Sec. III C we showed that using twice as much space can almost double our speed, essentially linearly until the log term begins to kick in. Thus, in managing space trade-offs, this will be our standard; any technique that raises performance by more than a factor of c in exchange for c times as much space will be used preferentially to parallel multiplication. Carry-select adders (Sec. III B) easily meet this criterion, being perhaps six times faster for less than twice the space.

Algorithm **D** uses $100n$ space and our conditional-sum adder CSUM. Algorithm **E** uses $100n$ space and the carry-lookahead adder QCLA. Algorithms **F** and **G** use the CDKM carry-ripple adder and $100n$ and minimal space, respectively. Parameters for these algorithms are shown in Table I. We have included detailed equations for concurrent VBE and **D** and numeric results in Table II. The performance ratios are based only on the CCNOT gate count for AC, and only on the CNOT gate count for NTC.
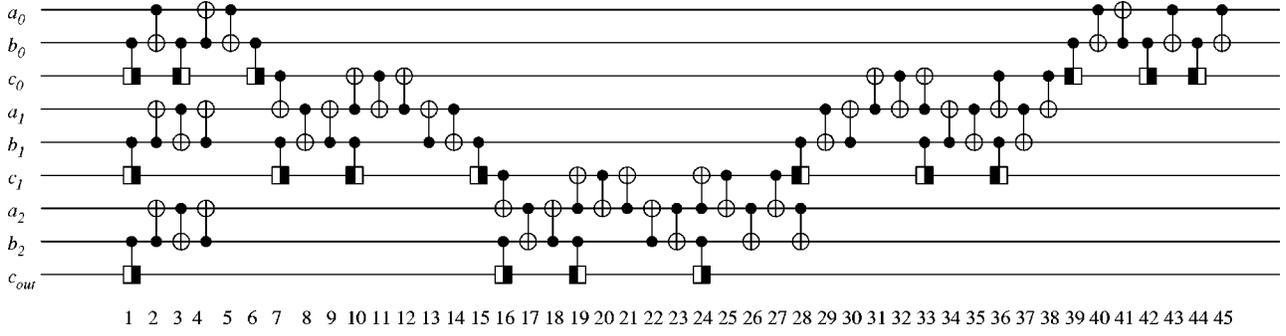
FIG. 10. Optimized, concurrent three bit VBE ADDER for the NTC architecture. Numbers across the bottom are time steps.

A. Concurrent VBE

On AC, the concurrent VBE ADDER is $(3n-3;2n-3;0)$ $=(381;253;0)$ for 128 bits. This is the value we use in the concurrent VBE line in Table II. This will serve as our best baseline time for comparing the effectiveness of more drastic algorithmic surgery.

Figure 10 shows a fully optimized, concurrent, but otherwise unmodified version of the VBE ADDER for three bits on a neighbor-only machine (NTC architecture), with the gates marked x in Fig. 2 eliminated. The latency is

$$t_{ADD}^{NTC} = (20n - 15;0)\#(2;3n + 1) \tag{20}$$

or 45 gate times for the three-bit adder. A 128-bit adder will have a latency of $(2545;0)$. The diagram shows a concurrency level of three, but simple adjustment of execution time slots can limit that to two for any n, with no latency penalty.

The unmodified full VBE modular exponentiation algorithm, consisting of $20n^2 - 5n = 327\,040$ ADDER calls plus minor additional logic, is

$$t_V^{NTC} = (20n^2 - 5n)t_{ADD}^{NTC} = (400n^3 - 400n^2 + 75n;0). \tag{21}$$

B. Algorithm D

The overall structure of algorithm **D** is similar to VBE, with our conditional-sum adders instead of the VBE carry-ripple, and our improvements in indirection and modulo. As we do not consider CSUM to be a good candidate for an algorithm for NTC, we evaluate only for AC. Algorithm **D** is the fastest algorithm for $n=8$ and $n=16$.

$$t_D = R_I R_M \times (t_{CSUM} + t_{ARG}) + 3pt_{CSUM}. \tag{22}$$

Letting $r = \lfloor \lceil (2n+1)/w \rceil / s \rfloor$, the latency and space requirements for algorithm **D** are

$$t_D^{AC} = 2r + 1 + \lceil \log_2[\lceil (s - 2n - 1 + rs)/4 \rceil + 2n + 1 - rs] \rceil n(2b$$
$$+ 1)/b \times [2m + 4\lceil \log_2(g-1)\rceil + 2;4;4\lceil \log_2(g-1)\rceil$$
$$+ 2) + (4;0;4)] + 3p(2m + 4\lceil \log_2(g-1)\rceil$$
$$+ 2;4;4\lceil \log_2(g-1)\rceil + 2) \tag{23}$$

and

$$S_D = s(S_{CSUM} + 2^w + 1 + p + n) + 2n + 1 = s[7n - 3m - g + 2^w$$
$$+ p + \lceil 3(g-1)/2 - 2 + (n-m)/2 \rceil] + 2n + 1. \tag{24}$$

C. Algorithm E

Algorithm **E** uses the carry-lookahead adder QCLA in place of the conditional-sum adder CSUM. Although CSUM is slightly faster than QCLA, its significantly larger space consumption means that in our $100n$ fixed-space analysis, we can fit in 16 multipliers using QCLA, compared to only 12 using CSUM. This allows the overall algorithm **E** to be 28% faster than **D** for 128 bits.

1. Algorithms F and G

The Cuccaro carry-rippler adder has a latency of $(10n + 5;0)$ for NTC. This is twice as fast as the VBE adder. We use this in our algorithms **F** and **G**. Algorithm **F** uses $100n$ space, while **G** is our attempt to produce the fastest algorithm in the minimum space.

D. Smaller n and different space

Figure 11 shows the execution times of our three fastest algorithms for n from eight to 128 bits. Algorithm **D**, using CSUM, is the fastest for eight and 16 bits, while **E**, using QCLA, is fastest for larger values. The latency of 1072 for $n=8$ bits is 32 times faster than concurrent VBE, achieved with $60n=480$ qubits of space.

Figure 12 shows the execution times for $n=128$ bits for various amounts of available space. All of our algorithms have reached a minimum by $240n$ space (roughly $1.9n^2$).

E. Asymptotic behavior

The focus of this paper is the constant factors in modular exponentiation for important problem sizes and architectural characteristics. However, let us look briefly at the asymptotic behavior of our circuit depth.

In Sec. III C, we showed that the latency of our complete algorithm is

$$O(n/s + \log s) \times \text{latency of multiplication} \tag{25}$$

as we parallelize the multiplication using s multiplier blocks. Our multiplication algorithm is still
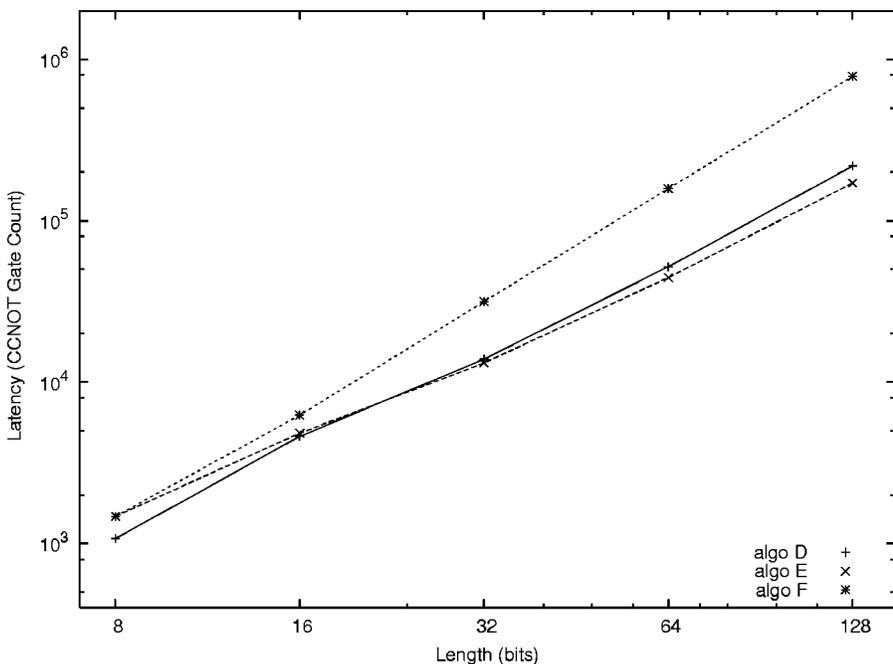
FIG. 11. Execution time for our algorithms for space $100n$ on the AC architecture, for varying value of n.

$$O(n) \times \text{latency of addition.} \qquad (26)$$

Algorithms **D** and **E** both use an $O(\log n)$ adder. Combining Eqs. (25) and (26) with the adder cost, we have asymptotic circuit depth of

$$t_D^{AC} = t_E^{AC} = O((n \log n)(n/s + \log s)) \qquad (27)$$

for algorithms **D** and **E**. As $s \rightarrow n$, these approach $O(n \log^2 n)$ and space consumed approaches $O(n^2)$.

Algorithm **F** uses an $O(n)$ adder, whose asymptotic behavior is the same on both AC and NTC, giving

$$t_F^{AC} = t_F^{NTC} = O((n^2)(n/s + \log s)) \qquad (28)$$

approaching $O(n^2 \log n)$ as space consumed approaches $O(n^2)$.

This compares to asymptotic behavior of $O(n^3)$ for VBE, BCDP, and algorithm **G**, using $O(n)$ space. The limit of performance, using a carry-save multiplier and large s, will be $O(\log^3 n)$ in $O(n^3)$ space.

V. DISCUSSION AND FUTURE WORK

We have shown that it is possible to significantly accelerate quantum modular exponentiation using a stable of tech-
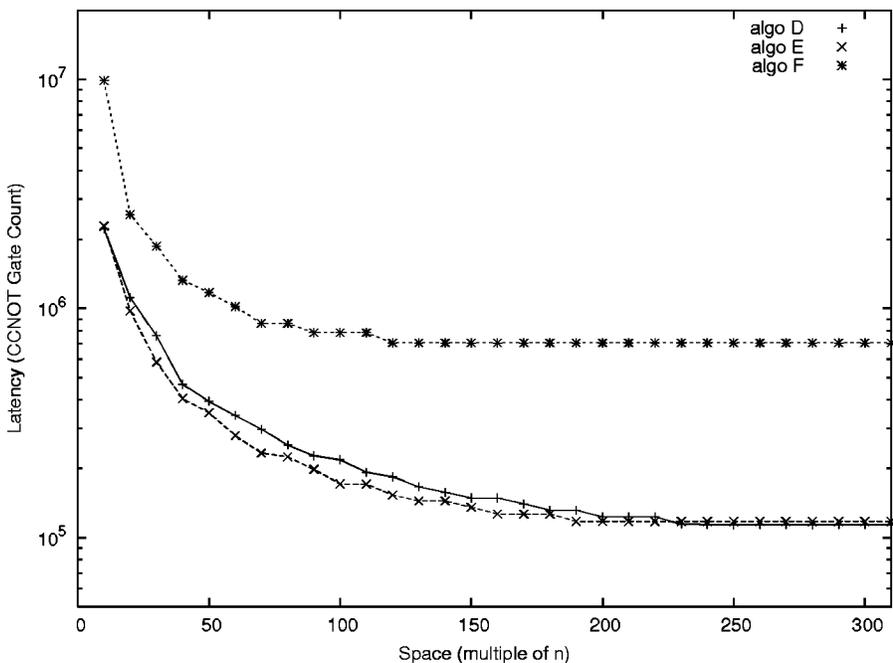
FIG. 12. Execution time for our algorithms for 128 bits on the AC architecture, for varying multiples of n space available.

niques. We have provided exact gate counts, rather than asymptotic behavior, for the $n=128$ case, showing algorithms that are faster by a factor of 200–700, depending on architectural features, when $100n$ qubits of storage are available. For $n=1024$, this advantage grows to more than a factor of 5000 for non-neighbor machines (AC). Neighbor-only (NTC) machines can run algorithms such as addition in $O(n)$ time at best, when non-neighbor machines (AC) can achieve $O(\log n)$ performance.

In this work, our contribution has focused on parallelizing execution of the arithmetic through improved adders, concurrent gate execution, and overall algorithmic structure. We have also made improvements that resulted in the reduction of modulo operations, and traded some classical for quantum computation to reduce the number of quantum operations. It seems likely that further improvements can be found in the overall structure and by more closely examining the construction of multipliers from adders [30]. We also intend to pursue multipliers built from hybrid carry-save adders.

The three factors that most heavily influence performance of modular exponentiation are, in order, concurrency, the availability of large numbers of application-level qubits, and the topology of the interconnection between qubits. Without concurrency, it is, of course, impossible to parallelize the execution of any algorithm. Our algorithms can use up to $\sim 2n^2$ application-level qubits to execute the multiplications in parallel, executing $O(n)$ multiplications in $O(\log n)$ time steps. Finally, if any two qubits can be operands to a quantum gate, regardless of location, the propagation of information about the carry allows an addition to be completed in $O(\log n)$ time steps instead of $O(n)$. We expect that these three factors will influence the performance of other algorithms in similar fashion.

Not all physically realizable architectures map cleanly to one of our models. A full two-dimensional mesh, such as neutral atoms in an optical lattice [38], and a loose-trellis topology [39] probably fall between AC and NTC. The behavior of the scalable ion trap [40] is not immediately clear. We have begun work on expanding our model definitions, as well as additional ways to characterize quantum computer architectures.

The process of designing a large-scale quantum computer has only just begun. Over the coming years, we expect advances in the fundamental technology, the system architecture, algorithms, and tools, such as compilers, to all contribute to the creation of viable quantum computing machines. Our hope is that the algorithms and techniques in this paper will contribute to that engineering process in both the short and long term.

[1] P. W. Shor, SIAM J. Comput. **26**, 1484 (1997).
[2] L. Grover, *Proc. 28th Annual ACM Symposium on the Theory of Computation* (ACM, New York, 1996), pp. 212–219.
[3] D. Deutsch and R. Jozsa, Proc. R. Soc. London, Ser. A **439**, 553 (1992).
[4] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, 2000).
[5] D. Beckman, A. N. Chari, S. Devabhaktuni, and J. Preskill, Phys. Rev. A **54**, 1034 (1996).
[6] P. W. Shor, *Proc. 35th Symposium on Foundations of Computer Science* (IEEE Computer Society Press, Los Alamitos, CA, 1994), pp. 124–134.
[7] V. Vedral, A. Barenco, and A. Ekert, Phys. Rev. A **54**, 147 (1996).
[8] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 3rd ed. (Addison-Wesley, Reading, MA, 1998).
[9] R. Cleve and J. Watrous, in *Proc. 41st Annual Symposium on Foundations of Computer Science* (ACM, New York, 2000), pp. 526–536.
[10] P. Gossett, e-print quant-ph/9808061.
[11] C. Zalka, e-print quant-ph/9806084.
[12] C. Moore and M. Nilsson, SIAM J. Comput. **31**, 799 (2001).
[13] D. Loss and D. P. DiVincenzo, Phys. Rev. A **57**, 120 (1998).
[14] B. E. Kane, Nature (London) **393**, 133 (1998).
[15] T. D. Ladd, J. R. Goldman, F. Yamaguchi, Y. Yamamoto, E. Abe, and K. M. Itoh, Phys. Rev. Lett. **89**, 017901 (2002).
[16] Y. A. Pashkin, T. Yamamoto, O. Astafiev, Y. Nakamura, D. V. Averin, and J. S. Tsai, Nature (London) **421**, 823 (2003).
[17] J. Q. You, J. S. Tsai, and F. Nori, Phys. Rev. Lett. **89**, 197902 (2002).
[18] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, Phys. Rev. A **52**, 3457 (1995).
[19] P. W. Shor, in *Proc. 37th Symposium on Foundations of Computer Science* (IEEE Computer Society Press, Los Alamitos, CA, 1996), pp. 56–65.
[20] A. M. Steane and B. Ibinson, e-print quant-ph/0311014.
[21] A. M. Steane, Quantum Inf. Comput. **2**, 297 (2002).
[22] A. V. Aho and K. M. Svore, e-print quant-ph/0311008.
[23] G. Ahokas, R. Cleve, and L. Hales, in [41].
[24] Y. Kawano, S. Yamashita, and M. Kitagawa (private communication); Phys. Rev. A (to be published).
[25] N. Kunihiro (private communication).
[26] Y. Takahashi, Y. Kawano, and M. Kitagawa, in [41].
[27] L. M. K. Vandersypen, Ph.D. thesis, Stanford University, 2001, (unpublished).
[28] A. Yao, in *Proceedings of the 34th Annual Symposium on Foundations of Computer Science* (Institute of Electrical and Electronic Engineers Computer Society Press, Los Alamitos, CA, 1993), pp. 352–361.
[29] R. Van Meter, in *Proc. Int. Symp. on Mesoscopic Superconduc-*

tivity and Spintronics (MS+S2004) (2004).

[30] M. D. Ercegovac and T. Lang, *Digital Arithmetic* (Morgan Kaufmann, San Francisco, CA, 2004).

[31] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, e-print quant-ph/0410184.

[32] T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore, e-print quant-ph/0406142.

[33] S. Beauregard, Quantum Inf. Comput. **3**, 175 (2003).

[34] T. G. Draper, e-print quant-ph/0008033.

[35] A. G. Fowler, S. J. Devitt, and L. C. Hollenberg, Quantum Inf. Comput. **4**, 237 (2004).

[36] S. J. Devitt, A. G. Fowler, and L. C. Hollenberg, e-print quant-ph/0408081.

[37] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, England 2000), pp. 266–268.

[38] G. K. Brennen, C. M. Caves, P. S. Jessen, and I. H. Deutsch, Phys. Rev. Lett. **82**, 1060 (1999).

[39] M. Oskin, F. T. Chong, I. L. Chuang, and J. Kubiatowicz, in *Computer Architecture News, Proc. 30th Annual International Symposium on Computer Architecture* (ACM, New York, 2003).

[40] D. Kielpinski, C. Monroe, and D. J. Wineland, Nature (London) **417**, 709 (2002).

[41] *Proc. ERATO Conference on Quantum Information Science (EQIS2003)* (unpublished).