Quantum Information and Computation, Vol. 5, No. 2 (2005) 113–130
© Rinton Press

GRAPH-BASED SIMULATION OF QUANTUM COMPUTATION IN THE DENSITY MATRIX REPRESENTATION

GEORGE F. VIAMONTES[a]
Department of Electrical Engineering and Computer Science, University of Michigan
1301 Beal Avenue, Ann Arbor, Michigan 48109-2122, USA

IGOR L. MARKOV[b]
Department of Electrical Engineering and Computer Science, University of Michigan
1301 Beal Avenue, Ann Arbor, Michigan 48109-2122, USA

JOHN P. HAYES[c]
Department of Electrical Engineering and Computer Science, University of Michigan
1301 Beal Avenue, Ann Arbor, Michigan 48109-2122, USA

Received June 11, 2004
Revised December 17, 2004

Quantum-mechanical phenomena are playing an increasing role in information processing, as transistor sizes approach the nanometer level, and quantum circuits and data encoding methods appear in the securest forms of communication. Simulating such phenomena efficiently is exceedingly difficult because of the vast size of the quantum state space involved. A major complication is caused by errors (noise) due to unwanted interactions between the quantum states and the environment. Consequently, simulating quantum circuits and their associated errors using the density matrix representation is potentially significant in many applications, but is well beyond the computational abilities of most classical simulation techniques in both time and memory resources. The size of a density matrix grows exponentially with the number of qubits simulated, rendering array-based simulation techniques that explicitly store the density matrix intractable. In this work, we propose a new technique aimed at efficiently simulating quantum circuits that are subject to errors. In particular, we describe new graph-based algorithms implemented in the simulator QuIDDPro/D. While previously reported graph-based simulators operate in terms of the state-vector representation, these new algorithms use the density matrix representation. To gauge the improvements offered by QuIDDPro/D, we compare its simulation performance with an optimized array-based simulator called QCSim. Empirical results, generated by both simulators on a set of quantum circuit benchmarks involving error correction, reversible logic, communication, and quantum search, show that the graph-based approach far outperforms the array-based approach for these circuits.

Keywords: Quantum circuits, quantum algorithms, simulation, density matrices, quantum errors, graph data structures, decision diagrams, QuIDDs

Communicated by: D. Wineland & B. Terhal

[a]E-mail: gviamont@eecs.umich.edu
[b]E-mail: imarkov@eecs.umich.edu
[c]E-mail: jhayes@eecs.umich.edu

1 Introduction

Practical information-processing applications that exploit quantum-mechanical effects are becoming common. For example, MagiQ Technologies markets a quantum communications device that detects eavesdropping attempts and prevents them [1]. The act of eavesdropping can be modeled as both making a quantum measurement and corruption by environmental noise [2]. Additionally, quantum computational algorithms have been discovered to quickly search unstructured databases [3] and to factor numbers in polynomial time [4]. Implementing quantum algorithms has proved to be particularly difficult, however, in part due to errors caused by the environment [5, 6]. Another related application is the design of reversible logic circuits. Since the operations performed in quantum computation must be unitary, they are all invertible and allow re-derivation of the inputs given the outputs [2]. This phenomenon gives rise to a host of potential applications in fault-tolerant computation. In addition, reversible logic gates can be used to completely specify search predicates in unstructured quantum search and modular exponentiation in quantum number factoring [3, 4]. Since reversible logic, quantum communication, and quantum algorithms can be modeled as quantum circuits [2], quantum circuit simulation incorporating errors could be of major benefit to these applications. In fact, any quantum-mechanical phenomenon with a finite number of states can be modeled as a quantum circuit [2, 7]. It may seem at first glance that efficient simulation of quantum circuits diminishes their value since they no longer offer a computational speed-up over classical computation. However, such simulation is extremely useful in that it offers insight into quantum circuit design and provides a way to study error behavior and new ideas for error correction, *which may be applicable to other quantum circuits that cannot be simulated efficiently.* Furthermore, quantum communication circuits are intended to provide secure forms of communication rather than computational improvements over classical computation. Similarly, reversible circuits have applications in quantum algorithms that provide speed-ups over classical computation *and* in fault-tolerant computing. Thus, efficient simulation of quantum communication and reversible logic circuits can be particularly valuable.

We present a new technique that facilitates efficient simulation of the density matrix representation of a class of quantum circuits. The density matrix representation is crucial in capturing interactions between quantum states and the environment, such as noise. In addition to the standard set of operations required to simulate with the state-vector model, including matrix multiplication and the tensor product, simulation with the density matrix model requires the outer product and the partial trace. The outer product is used in the initialization of qubit density matrices, while the partial trace allows a simulator to differentiate qubit states coupled to noisy environments or other unwanted states. The partial trace is invaluable in error modeling since it facilitates descriptions of single qubit states that have been affected by noise and other phenomena [2]. Our techniques specifically target well-defined quantum circuit applications which only require unitary evolutions of a state initialized in the computational basis, and we are not dealing with explicitly specified Hamiltonians. Errors can be injected into the unitary operations directly. As will be demonstrated later, the main motivation for using the density matrix in this work is that the partial trace can be used to isolate the effect of these errors on the data qubits, whereas this cannot be done in the state-vector model.

Unfortunately, like the state-vector model, simulation with the density matrix is computationally challenging on classical computers. The size of any density matrix grows exponentially with the number of qubits or quantum states it represents [2]. Thus, simulation techniques which require explicit storage of the density matrix in a series of arrays are inefficient and generally intractable. However, the new simulation technique we propose is founded in graph-based algorithms which can represent and manipulate density matrices very efficiently in many important cases. A key component of our algorithms is the *Quantum Information Decision Diagram* (QuIDD) data structure, which can represent and manipulate a useful class of matrices and vectors commonly found in quantum circuit applications using time and memory resources that are *polynomial* in the number of qubits [8, 9]. A limitation of our previous QuIDD algorithms, and other graph-based techniques [10], is that they simulate the state-vector representation of quantum circuits. In this work, we present new algorithms to perform the outer product and the partial trace with QuIDDs. These algorithms enable QuIDD-based simulation of quantum circuits with the density matrix representation.

We also describe a set of quantum circuit benchmarks that incorporate errors, error correction, reversible logic, quantum communication, and quantum search. To empirically evaluate the improvements offered by our new technique, we use these benchmarks to compare QuIDDPro/D with an optimized array-based density matrix simulator called QCSim [11]. Performance data from both simulators show that our new graph-based algorithms far outperform the array-based approach for the given benchmarks. It should be noted, however, that not all quantum circuits can be simulated efficiently with QuIDDs. A useful class of matrices and vectors which can be manipulated efficiently by QuIDDs has been formally described in previous work [8] and is restated below. For some matrices and vectors outside of this class, QuIDD-based simulation can be up to three times slower due to the overhead of following pointers in the QuIDD datastructure.

The paper is organized as follows. Section 2 provides background on decision diagram data structures and previous simulation work. In Section 3 we present our new algorithms along with a description of the QuIDDPro/D simulator. Section 3.3 describes the quantum circuit benchmarks and presents performance results on each benchmark for QuIDDPro/D and QCSim. Finally, in Section 4 we present our conclusions and ideas for future work.

2 Background and Previous Work

The simulation technique proposed in this work relies on the QuIDD data structure, which is a type of graph called a decision diagram. This section presents the basic concepts of decision diagrams, assuming only a rudimentary knowledge of computational complexity and graph theory. It then reviews previous research on simulating quantum circuits.

2.1 Binary Decision Diagrams

Many decision diagrams are ultimately based on the binary decision diagram (BDD). The BDD was introduced by Lee in 1959 in the context of classical logic circuit design [12]. This data structure represents a Boolean function $f(x_1, x_2, ..., x_n)$ by a directed acyclic graph (DAG) as shown in Fig. 1. By convention, the top node of a BDD is labeled with the name of the function f represented by the BDD. Each variable x_i of f is associated with one or more nodes, each of which have two outgoing edges labeled *then* (solid line) and *else* (dashed line). The *then* edge of node x_i denotes an assignment of logic 1 to the x_i, while the *else* edge denotes an assignment of logic 0. These nodes are called *internal* nodes and are labeled by the corresponding variable x_i. The edges of the BDD point downward, implying a top-down assignment of values to the Boolean variables depicted by the internal nodes.

At the bottom of a BDD are *terminal* nodes containing the logic values 1 or 0. They denote the output value of the function f for a given assignment of its variables. Each path through the BDD from top to bottom represents a specific assignment of 0-1 values to the variables $x_1, x_2, ..., x_n$ of f, and ends with the corresponding output value $f(x_1, x_2, ..., x_n)$.

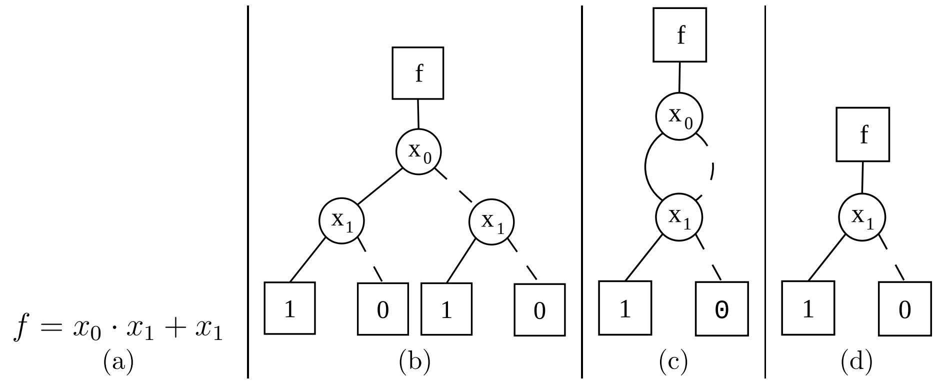

Fig. 1. (a) A logic function, (b) its BDD representation, (c) its BDD representation after applying the first reduction rule, and (d) its ROBDD representation.

The memory complexity of the original BDD data structure conceived by Lee is exponential in the number of variables for a given logic function. Simulation of many practical logic circuits with this data structure was therefore impractical. To address this limitation, Bryant developed the Reduced Ordered BDD (ROBDD) [13], where all variables are ordered, and assignment of values to variables are made in that order. A key advantage of the ROBDD is that variable-ordering facilitates an efficient implementation of reduction rules that automatically eliminate redundancy from the basic BDD representation. These rules are summarized as follows:

(i) There are no nodes v and v' such that the subgraphs rooted at v and v' are isomorphic
(ii) There are no internal nodes with *then* and *else* edges that both point to the same node

An example of how the rules distinguish an ROBDD from a BDD is shown in Fig. 1. The subgraphs rooted at the x_1 nodes in Fig. 1b are isomorphic. By applying the first reduction rule, the BDD in Fig. 1b becomes the BDD in Fig. 1c. Note that, in Fig. 1c, the *then* and *else* edges of the x_0 node now point to the same node. Applying the second reduction rule eliminates the x_0 node, resulting in the ROBDD in Fig. 1d. Intuitively it makes sense to eliminate the x_0 node since the output of the original function is determined solely by the value of x_1. In many Boolean functions, this type of redundancy is eliminated with varying success depending on the order in which variables in the function are evaluated. Finding the optimal variable ordering is an NP-complete problem, but efficient ordering heuristics have been developed for specific applications. Moreover, it turns out that many practical logic functions have ROBDD representations that are polynomial (or even linear) in the number of input variables [13]. Consequently, ROBDDs have become indispensable tools in the design and simulation of classical logic circuits.

2.2 BDD Operations

Even though the ROBDD is often quite compact, efficient algorithms are also needed to manipulate ROBDDs for circuit simulation. Thus, in addition to the foregoing reduction rules, Bryant introduced a variety of ROBDD operations with complexities that are bounded by the size of the ROBDDs being manipulated [13]. Of central importance is the *Apply* operation, which performs a binary operation with two ROBDDs, producing a third ROBDD as the result. It can be used, for example, to compute the logical AND of two functions. *Apply* is implemented by a recursive traversal of the two ROBDD operands. For each pair of nodes visited during the traversal, an internal node is added to the resultant ROBDD using the three rules depicted in Fig. 2. To understand the rules, some notation must be introduced. Let v_f denote an arbitrary node in an ROBDD f. If v_f is an internal node, $Var(v_f)$ is the Boolean variable represented by v_f, $T(v_f)$ is the node reached when traversing the *then* edge of v_f, and $E(v_f)$ is the node reached when traversing the *else* edge of v_f.

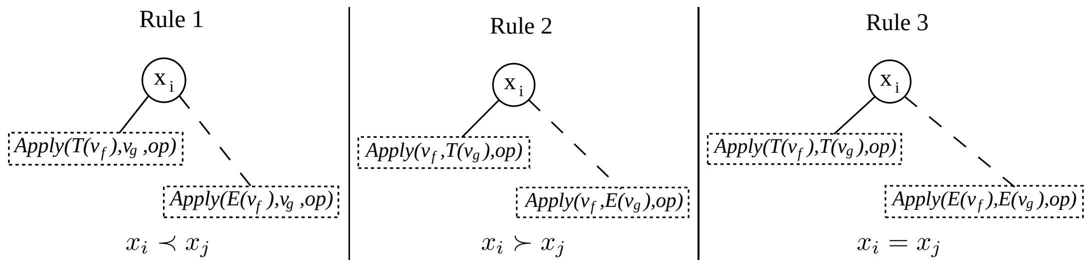

Fig. 2. The three recursive rules used by the *Apply* operation which determine how a new node should be added to a resultant ROBDD. In the figure, $x_i = Var(v_f)$ and $x_j = Var(v_g)$. The notation $x_i \prec x_j$ is defined to mean x_i precedes x_j in the variable ordering.

Clearly the rules depend on the variable ordering. To illustrate, consider performing *Apply* using a binary operation *op* and two ROBDDs f and g. *Apply* takes as arguments two nodes, one from f and one from g, and the operation *op*. This is denoted as $Apply(v_f, v_g, op)$. *Apply* compares $Var(v_f)$ and $Var(v_g)$ and adds a new internal node to the ROBDD result using the three rules. The rules also guide *Apply*'s traversal of the *then* and *else* edges (this is the recursive step). For example, suppose $Apply(v_f, v_g, op)$ is called and $Var(v_f) \prec Var(v_g)$. Rule 1 is invoked, causing an internal node containing $Var(v_f)$ to be added to the resulting ROBDD. Rule 1 then directs the *Apply* operation to call itself recursively with $Apply(T(v_f), v_g, op)$ and $Apply(E(v_f), v_g, op)$. Rules 2 and 3 dictate similar actions but handle the cases when $Var(v_f) \succ Var(v_g)$ and $Var(v_f) = Var(v_g)$. To recurse over both ROBDD operands correctly, the initial call to *Apply* must be $Apply(Root(f), Root(g), op)$ where $Root(f)$ and $Root(g)$ are the root nodes for the ROBDDs f and g.

The recursion stops when both v_f and v_g are terminal nodes. When this occurs, *op* is performed with the values of the terminals as operands, and the resulting value is added to the ROBDD result as a terminal node. For example, if v_f contains the value logical 1, v_g contains the value logical 0, and op is defined to be $\oplus$ (XOR), then a new terminal with value

$1 \oplus 0 = 1$ is added to the ROBDD result. Terminal nodes are considered *after* all variables are considered. Thus, when a terminal node is compared to an internal node, either Rule 1 or Rule 2 will be invoked depending on which ROBDD the internal node is from.

The success of ROBDDs in making a seemingly difficult computational problem tractable in practice led to the development of ROBDD variants outside the domain of logic design. Of particular relevance to this work are Multi-Terminal Binary Decision Diagrams (MTBDDs) [14] and Algebraic Decision Diagrams (ADDs) [15]. These data structures are compressed representations of matrices and vectors rather than logic functions, and the amount of compression achieved is proportional to the frequency of repeated values in a given matrix or vector. Additionally, some standard linear-algebraic operations, such as matrix multiplication, are defined for MTBDDs and ADDs. Since they are based on the *Apply* operation, the efficiency of these operations is proportional to the size in nodes of the MTBDDs or ADDs being manipulated. Further discussion of the MTBDD and ADD representations is deferred to Sec. 3 where the general structure of the QuIDD is described.

2.3 Previous Simulation Techniques

Quantum circuit simulators must support linear-algebraic operations such as matrix multiplication, the tensor product, and the projection operators. Simulation with the density matrix model additionally requires the outer product and partial trace [2]. Many simulators typically employ array-based methods to facilitate these operations and so require exponential computational resources in the number of qubits. Such methods are often insensitive to the actual values stored, and even sparse-matrix storage offers little improvement for quantum operators with no zero matrix elements, such as Hadamard operators. Previous work on these and other simulation techniques is reviewed in this subsection.

One popular array-based simulation technique is to simulate k-input quantum gates on an n-qubit state-vector ($k \leq n$) without explicitly storing a $2^n \times 2^n$-matrix representation [11, 16]. The basic idea is to simulate the full-fledged matrix-vector multiplication by a series of simpler operations. To illustrate, consider simulating a quantum circuit in which a 1-qubit Hadamard operator is applied to the third qubit of the state-space $|00100\rangle$. The state-vector representing this state-space has 2^5 elements. A naive way to apply the 1-qubit Hadamard is to construct a $2^5 \times 2^5$ matrix of the form $I \otimes I \otimes H \otimes I \otimes I$ and then multiply this matrix by the state-vector. However, rather than compute $(I \otimes I \otimes H \otimes I \otimes I)|00100\rangle$, one can simply compute $|00\rangle \otimes H|1\rangle \otimes |00\rangle$, which produces the same result using a 2×2 matrix H. The same technique can be applied when the state-space is in a superposition, such as $\alpha|00100\rangle + \beta|00000\rangle$. In this case, to simulate the application of a 1-qubit Hadamard operator to the third qubit, one can compute $|00\rangle \otimes H(\alpha|1\rangle + \beta|0\rangle) \otimes |00\rangle$. As in the previous example, a 2×2 matrix is sufficient.

While the above method allows one to compute a state space symbolically, in a realistic simulation environment, state-vectors may be much more complicated. Shortcuts that take advantage of the linearity of matrix-vector multiplication are desirable. For example, a single qubit can be manipulated in a state-vector by extracting a certain set of two-dimensional vectors. Each vector in such a set is composed of two probability amplitudes. The corresponding qubit states for these amplitudes differ in value at the position of the qubit being operated on but agree in every other qubit position. The two-dimensional vectors are then multiplied by matrices representing single qubit gates in the circuit being simulated. We refer to this technique as *qubit-wise multiplication* because the state-space is manipulated one qubit at a time. Obenland implemented a technique of this kind as part of a simulator for quantum circuits [16]. His method applies one- and two-qubit operator matrices to state vectors of size 2^n. Unfortunately, in the best case where $k = 1$, this only reduces the runtime and memory complexity from $O(2^{2n})$ to $O(2^n)$, which is still exponential in the number of qubits.

Another implicit limitation of Obenland's implementation is that it simulates with the state-vector representation only. The qubit-wise technique has been extended, however, to enable density matrix simulation by Black et al. and is implemented in NIST's QCSim simulator [11]. As in its predecessor simulators, the arrays representing density matrices in QCSim tend to grow exponentially. This asymptotic bottleneck is demonstrated experimentally in Sec. 3.3.

Gottesman developed a simulation method involving the *Heisenberg representation* of quantum computation which tracks the commutators of operators applied by a quantum circuit [17]. With this model, the state need not be represented explicitly by a state-vector or a density matrix because the operators describe how an arbitrary state-vector would be altered by the circuit. Gottesman showed that simulation based on this model requires only polynomial memory and runtime on a classical computer in certain cases. However, it appears limited to the Clifford group of quantum operators, which do not form a universal gate library. A recent extension to this technique enables simulation with any quantum operators, but the complexity grows exponentially with every operator introduced that is not in the Clifford group [18].

Other advanced simulation techniques including MATLAB's "packed" representation, apply data compression to matrices and vectors, but cannot perform matrix-vector multiplication without first decompressing the matrices and vectors. A notable exception is Greve's graph-based simulation of Shor's algorithm which uses BDDs [10]. Probability amplitudes of individual qubits are modeled by single decision nodes. Unfortunately, this only captures superpositions where every participating qubit is rotated by ± 45 degrees from $|0\rangle$ toward $|1\rangle$.

3 Graph-Based Algorithms for Density Matrix Simulation

QuIDDPro/D utilizes new simulation algorithms with unique features that allow it to have much higher performance than naive explicit array-based simulation techniques. These algorithms are the subject of this section. In addition, we provide some implementation details of the QuIDDPro/D simulator.

3.1 QuIDDs and New QuIDD Algorithms

Our density matrix simulation technique relies on the QuIDD data structure. Previous work reported the use of QuIDDs in simulating the state-vector model of quantum circuits [8, 9]. We present new algorithms which use QuIDDs to efficiently perform the outer product and the partial trace, both of which are needed to simulate density matrices. Before discussing the details of these algorithms, we briefly review the QuIDD data structure.

The QuIDD was born from the observation that vectors and matrices which arise in quantum computing contain repeated structure. Operators obtained from the tensor product of smaller matrices exhibit common substructures which certain ROBDD variants can capture.

The QuIDD can be viewed as an ADD [15] or MTBDD [14] with special properties [8, 9].

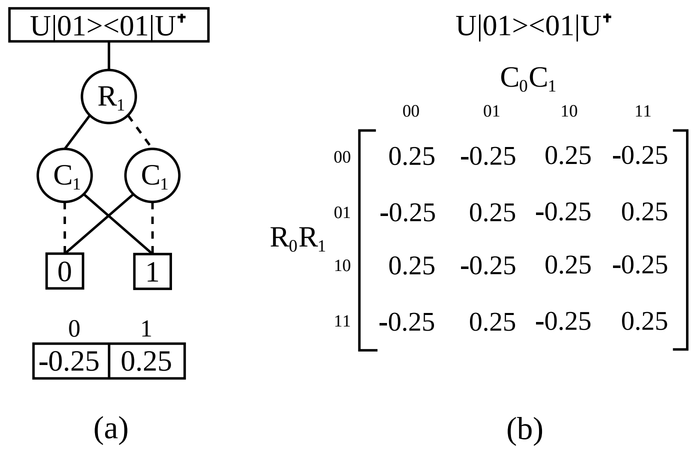

Fig. 3. (a) QuIDD for the density matrix resulting from $U|01\rangle\langle 01|U^\dagger$, where $U = H \otimes H$, and (b) its explicit matrix form.

Fig. 3a shows the QuIDD that results from applying U to an outer product as $U|01\rangle\langle 01|U^\dagger$, where $U = H \otimes H$. The R_i nodes of the QuIDD encode the binary indices of the rows in the explicit matrix. Similarly, the C_i nodes encode the binary indices of the columns. Solid lines leaving a node denote the positive cofactor of the index bit variable (a value of 1), while dashed lines denote the negative cofactor (a value of 0). Terminal nodes correspond to the value of the element in the explicit matrix whose binary row/column indices are encoded by the path that was traversed.

Notice that the first and second pairs of rows of the explicit matrix in Fig. 3b are equal, as are the first and second pairs of columns. This redundancy is captured by the QuIDD in Fig. 3a because the QuIDD does not contain any R_0 or C_0 nodes. In other words, the values and their locations in the explicit matrix can be completely determined without the superfluous knowledge of the first row and column index bits.

Measurement, matrix multiplication, addition, scalar products, the tensor product, and other operations involving QuIDDs are variations of the well-known *Apply* algorithm discussed in Sec. 2.2 [8, 9]. Vectors and matrices with large blocks of repeated values can be manipulated in QuIDD form quite efficiently with these operations. In addition, it has been proven that by interleaving the row and column variables in the variable ordering, QuIDDs can represent and operate on a certain class of matrices and vectors using time and memory resources that are *polynomial* in the number of qubits. This class includes, but is not limited to, any equal superposition of n qubits, any sequence of n qubits in the computational basis states, n-qubit Pauli operators, and n-qubit Hadamard operators. Specifically, this class includes any vector or matrix created from the tensor product of vectors or matrices whose elements are in a *persistent* set. Informally, a persistent set is a set of complex numbers whose set of all-pairs products is the same size as the original set. Persistent sets have been explicitly characterized in previous work [8] and can include, for example, arbitrary roots of unity, zero, and other complex numbers.

```
$\textbf{Outer\_Product}(Q, num_q ubits)$ {
    $Q\_cctrans = \textbf{Swap\_Row\_Col\_Vars}(Q)$;
    $Q\_cctrans = \textbf{Complex\_Conj}(Q\_cctrans)$;
    $R = \textbf{Matrix\_Multiply}(Q, Q\_cctrans)$;
    $R = \textbf{Scalar\_Div}(Q\_cctrans, 2^{num\_qubits})$;
    $return\ R$;
}
```

(a)

```
$\textbf{Complex\_Conj}(Q)$ {
    $if\ (\textbf{Is\_Constant}(Q))$
        $return\ \textbf{New\_Terminal}(\textbf{R}eal(Q),$
                                    $-1 * \textbf{I}mag(Q))$;
    $if\ (\textbf{Table\_Lookup}(computed\_table, Q, R)$
        $returnR$;
    $v = \textbf{Top\_Var}(Q)$;
    $T = \textbf{Complex\_Conj}(Q_v)$;
    $E = \textbf{Complex\_Conj}(Q_{v'})$;
    $R = \textbf{ITE}(v, T, E)$;
    $\textbf{Table\_Insert}(computed\_table, Q, R)$;
    $return\ R$;
}
```

(b)

Fig. 4. Pseudo-code for (a) the QuIDD outer product and (b) its complex conjugation helper function **C***omplex_Conj*. The code for **S***calar_Div* is the same as **C***omplex_Conj*, except that in the terminal node case it returns the value of the terminal divided by a scalar. Other functions are typical ADD operations [15, 19].

Since QuIDDs already have the capability to represent matrices and multiply them [8, 9], extending QuIDDs to encompass the density matrix requires algorithms for the outer product and the partial trace. The outer product involves matrix multiplication between a column vector and its complex-conjugate transpose. Since a column vector QuIDD only depends on row variables, the transpose can be accomplished by swapping the row variables with column variables. The complex conjugate can then be performed with a DFS traversal that replaces terminal node values with their complex conjugates. The original column vector QuIDD is then multiplied by its complex-conjugate transpose using the matrix multiply operation previously defined for QuIDDs [8, 9]. Pseudo-code for this algorithm is given in Fig. 4. Notice that before the result is returned, it is divided by 2^{num_qubits}, where num_qubits is the number of qubits represented by the QuIDD vector. This is done because a QuIDD that only depends on n row variables can be viewed as either a $2^n \times 1$ column vector or a $2^n \times 2^n$ matrix in which all columns are the same. Since matrix multiplication is performed in terms of the latter case [8, 9, 15], the result of the outer product contains values that are multiplied by an extra factor of 2^n, which must be normalized.

Although QuIDDs enable efficient simulation for a class of matrices and vectors in the state-vector paradigm, it must be shown that the corresponding density matrix version of this class can also be simulated efficiently. Since state-vectors are converted to density matrices via the outer product, this can be shown by proving that the outer product of a QuIDD vector in this class with its complex-conjugate transpose results in a QuIDD density matrix with size polynomial in the number of qubits.

Theorem 1: Given an n-qubit QuIDD state-vector whose terminal values are in a persistent set, the outer product of this QuIDD with its complex-conjugate transpose produces a QuIDD matrix with polynomially many nodes in n.

Proof: Since the given QuIDD state-vector's terminal values are in a persistent set, the number of nodes in the QuIDD is $O(n)$ [8]. Consider the pseudo-code for the QuIDD outer product shown in Fig. 4a. The first operation is to create a transposed copy of the QuIDD state-vector. Transposition only requires remapping the internal variable nodes to represent column variables instead of row variables. This can be done in one pass of all the nodes in the QuIDD state-vector [8]. Since the number of nodes is $O(n)$, this operation has $O(n)$ runtime complexity and creates a transposed copy with $O(n)$ nodes. The next operation is to complex-conjugate the transposed QuIDD copy. As evidenced by the pseudo-code for complex conjugation of QuIDDs in Fig. 4b, this involves a single recursive pass over all nodes. All internal nodes are returned unchanged with the $O(1)$ ADD *ITE* operation [15], whereas the complex-conjugate of the terminals are returned when they are reached. Since the number of nodes in the transposed QuIDD copy is $O(n)$, the runtime complexity of this operation is $O(n)$ and results in a new QuIDD with $O(n)$ nodes. Next, QuIDD matrix multiplication is performed on the QuIDD state-vector and its complex-conjugate transpose to produce the QuIDD density matrix. It has been proven that QuIDD matrix multiplication of a QuIDD with A nodes and a QuIDD with B nodes has runtime complexity $O((AB)^2)$ and results in a QuIDD with $O((AB)^2)$ nodes [8]. Since the QuIDD state-vector and its complex-conjugate transpose each have $O(n)$ nodes, the matrix multiplication step has runtime complexity $O(n^4)$. The final normalization step of the outer product is a scalar division of the terminal values. Like QuIDD complex conjugation, this operation is a single recursive pass over the QuIDD, but when the terminals are reached the scalar division result is returned. Since the QuIDD density matrix has $O(n^4)$ nodes, this operation has runtime complexity $O(n^4)$. Based on the complexity of all steps in the QuIDD outer product algorithm, the overall runtime complexity of the QuIDD outer product is $O(n^4)$ and results in a QuIDD density matrix with $O(n^4)$ nodes. $\square$

To motivate the QuIDD-based partial trace algorithm, we note how the partial trace can be performed with explicit matrices. The trace of a matrix A is the sum of A's diagonal elements. To perform the partial trace over a particular qubit in an n-qubit density matrix, the trace operation can be applied iteratively to sub-matrices of the density matrix. Each sub-matrix is composed of four elements with row indices $r0s$ and $r1s$, and column indices $c0d$and $c1d$, where r, s, c, and d are arbitrary sequences of bits which index the n-qubit density matrix.

Tracing over these sub-matrices has the effect of reducing the dimensionality of the density matrix by one qubit. A well-known ADD operation which reduces the dimensionality of a matrix is the *Abstract* operation [15]. Given an arbitrary ADD f, abstraction of variable x_i eliminates all internal nodes of f which represent x_i by combining the positive and negative cofactors of f with respect to x_i using some binary operation. In other words, $Abstract(f, x_i, op) = f_{x_i} \; op \; f_{x'_i}$.

For QuIDDs, there is a one-to-one correspondence between a qubit on wire i (wires are labeled top-down starting at 0) and variables R_i and C_i. So at first glance, one may suspect that the partial trace of qubit i in f can be achieved by a performing $Abstract(f, R_i, +)$ followed by $Abstract(f, C_i, +)$. However, this will add the rows determined by qubit i independently of the columns. The desired behavior is to perform the diagonal addition of

```
$\mathbf{Ptrace}(Q, qubit\_index)$ {
  $if(\mathbf{Is\_Constant}(Q))$
    $return\ Q;$
  $top\_q = \mathbf{Top\_Var}$
  $if\ (qubit\_index < \mathbf{Index}(top\_q))$ {
    $R = \mathbf{Apply}(Q, Q, +);$
    $return\ R;$
  }

  $if\ (\mathbf{Table\_Lookup}(computed\_table, (Q, qubit\_index), R)$
    $return\ R;$
  $T = Q_{top\_q};$
  $E = Q_{top\_q'};$

  $if\ (qubit\_index == \mathbf{Index}(top\_q))$ {
    $if\ (\mathbf{Is\_Constant}(T)\ ||\ \mathbf{Index}(T) > \mathbf{Index}(Q) + 1)$
      $r_1 = T;$
    $else$ {
      $top\_T = \mathbf{Top\_Var}(T);$
      $r_1 = T_{top_T};$
    }

    $if\ (\mathbf{Is\_Constant}(E)\ ||\ \mathbf{Index}(E) > \mathbf{Index}(Q) + 1)$
      $r_2 = E;$
    $else$ {
      $top\_E = \mathbf{Top\_Var}(E);$
      $r_2 = E_{top\_E'};$
    }
    $R = \mathbf{Apply}(r_1, r_2, +);$
    $\mathbf{Table\_Insert}(computed\_table, (Q, qubit\_index), R);$
    $return\ R;$
  }

  $else$ {   /* $(qubit\_index > \mathbf{Index}(top\_q))$ */
    $r_1 = \mathbf{Ptrace}(T, qubit\_index);$
    $r_2 = \mathbf{Ptrace}(E, qubit\_index);$
    $R = \mathbf{ITE}(top\_q, r_1, r_2);$
    $\mathbf{Table\_Insert}(computed\_table, (Q, qubit\_index), R);$
    $return\ R;$
  }
}
```

Fig. 5. Pseudo-code for the QuIDD partial trace. The index of the qubit being traced-over is *qubit_index*.

sub-matrices by accounting for both the row and column variables due to i *simultaneously*. The pseudo-code to perform the partial trace correctly is depicted in Fig. 5. In comparing this with the pseudo-code for the *Abstract* algorithm [15], the main difference is that when R_i corresponding to qubit i is reached, we take the positive and negative cofactors *twice* before making the recursive call. Since the interleaved variable ordering of QuIDDs guarantees that C_i immediately follows R_i [8, 9], taking the positive and negative cofactors twice simultaneously abstracts both the row and column variables for qubit i, achieving the desired result of summing diagonals. In other words, for a QuIDD f, the partial trace over qubit i is $Ptrace(f, i) = f_{R_i C_i} + f_{R'_i C'_i}$. Note that in the pseudo-code there are checks for the special case when no internal nodes in the QuIDD represent C_i. Not shown in the pseudo-code is book-keeping which shifts up the variables in the resulting QuIDD to fill the hole in the ordering left by the row and column variables that were traced-over.

As in the case of the outer product, the QuIDD partial trace algorithm has efficient runtime and memory complexity in the size of the QuIDD being traced-over, as we now show.

Theorem 2: Given an n-qubit QuIDD density matrix with A nodes, any qubit represented in the matrix can be traced-over with runtime complexity $O(A)$ and results in a QuIDD density matrix with $O(A)$ nodes.

Proof: Consider the pseudo-code for the QuIDD partial trace algorithm in Fig. 5. The algorithm performs a recursive traversal over the nodes in the QuIDD density matrix and takes certain actions when special cases are encountered. If a node is encountered which corresponds to a qubit preceded by the traced-over qubit in the variable ordering,[d] then recursion stops and the sub-graph is added to itself with the ADD *Apply* algorithm [13]. This operation has runtime complexity $O(A)$ and results in a new sub-graph with $O(A)$ nodes. Next, if the partial trace of the current sub-graph has already been computed, then recursion stops and the pre-computed result is simply looked up in the computed table cache and returned. This operation has runtime complexity $O(1)$ and returns a sub-graph with $O(A)$ nodes [13]. If there is no entry in the computed table cache, the algorithm checks if the current node's variable corresponds to the qubit to be traced-over. If so, *Apply* is used to add the node's children or children's children, which again has $O(A)$ runtime and memory complexity. If the current node does not correspond to the qubit being traced-over, then the partial trace algorithm is called recursively on the node's children. Since all the other special cases stop recursion and involve an *Apply* operation, then the overall runtime complexity of the partial trace algorithm is $O(A)$ and results in a new QuIDD density matrix with $O(A)$ nodes. $\square$

3.2 QuIDDPro/D

QuIDDPro/D is the implementation of our simulation technique [20]. It was written in C++, and the source code is approximately 17,000 lines long. The density matrix is represented by a QuIDD class with terminal node values of type *complex* $<$ *long double* $>$. Gate operators are also represented as QuIDDs. QuIDDPro/D utilizes our earlier QuIDDPro source code [8, 9] and extends it significantly with an implementation of the outer product and partial trace pseudo-code of Fig. 4 and Fig. 5. Additionally, the technique of using an epsilon to deal with precision problems in QuIDDPro [8, 9] has been replaced with a technique that rounds

[d]Recall that there is a one-to-one correspondence between a qubit on wire i and variables R_i and C_i

complex numbers to 25 significant digits. This enhancement allows an end-user to avoid having to find an optimal value of epsilon for a given quantum circuit input. A front-end parser was also created to accept a subset of the MATLAB language, which is ideal for describing linear-algebraic operations in a text format. In addition to incorporating a set of well-known numerical functions, the language also supports a number of other functions that are useful in quantum circuit simulation. For example, there is a function for creating controlled-U gates with an arbitrary configuration for the control qubits and user-defined specification of U. Functions to perform deterministic measurement, probabilistic measurement, and the partial trace, among others, are also supported. The current version of QuIDDPro/D contains over 65 functions.

3.3 Experimental Results

We consider a number of quantum circuit benchmarks which cover errors, error correction, reversible logic, communication, and quantum search. We devised some of the benchmarks, while others are drawn from NIST [11] and from a site devoted to reversible circuits [21]. For every benchmark, the simulation performance of QuIDDPro/D is compared with NIST's QCSim quantum circuit simulator, which utilizes an explicit array-based computational engine. The results indicate that QuIDDPro/D far outperforms QCSim. All experiments are performed on a 1.2GHz AMD Athlon workstation with 1GB of RAM running Linux.

3.4 Reversible Circuits

Here we examine the performance of QuIDDPro/D simulating a set of reversible circuits, which we define as quantum circuits that perform classical operations [2]. Specifically, if the input qubits of a quantum circuit are all in the computational basis (i.e. they have only $|0\rangle$ or $|1\rangle$ values), there is no quantum noise, and all the gates are NOT variants such as CNOT, Toffoli, X, etc, then the output qubits and all intermediate states will also be in the computational basis. Such a circuit results in a classical logic operation which is reversible in the sense that the inputs can always be derived from the outputs and the circuit function. Reversibility comes from the fact that all quantum operators must be unitary and therefore all have inverses [2].

The first reversible benchmark we consider is a reversible 4-bit ripple-carry adder which is depicted in Fig. 6. Since the size of a QuIDD is sensitive to the arrangement of different values of matrix elements, we simulate the adder with varied input values ("rc_adder1" through "rc_adder4"). This is also done for other benchmarks. Two other reversible benchmarks we simulate contain fewer qubits but more gates than the ripple-carry adder. One of these benchmarks is a 12-qubit reversible circuit that outputs a $|1\rangle$ on the last qubit if and only if the number of $|1\rangle$'s in the input qubits is 3, 4, 5, or 6 ("9sym1" through "9sym5") [21]. The other benchmark is a 15-qubit reversible circuit that generates the classical Hamming code of the input qubits ("ham15_1" through "ham15_3") [21].

Performance results for all of these benchmarks are shown in Tab. 1. QuIDDPro/D significantly outperforms QCSim in every case. In fact for circuits of 14 or more qubits, QCSim requires more than 2GB of memory. Since QCSim uses an explicit array-based engine, it is insensitive to the arrangement and values of elements in matrices. Therefore, one can expect QCSim to use more than 2GB of memory for *any* benchmark with 14 or more qubits, regardless of the circuit functionality and input values. Another interesting result is that even though

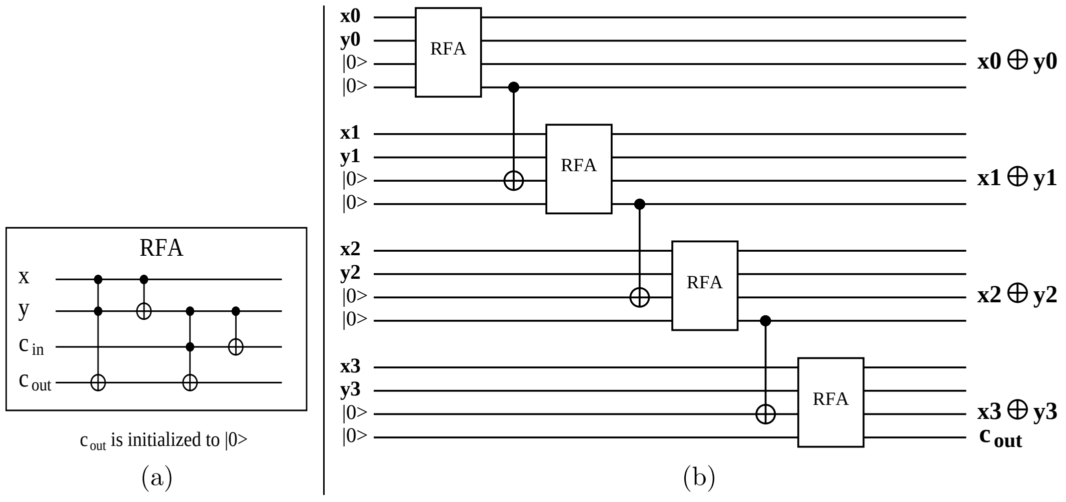

Fig. 6. (a) An implementation of a reversible full-adder (RFA), and (b) a reversible 4-bit ripple-carry adder which uses the RFA as a module. The reversible ripple-carry adder circuit computes the binary sum of two 4-bit numbers: $x_3x_2x_1x_0 \oplus y_3y_2y_1y_0$. c_{out} is the final carry bit output from the addition of the most-significant bits (x_3 and y_3).

Table 1. Performance results for QuIDDPro/D and QCSim on the reversible circuit benchmarks. MEM-OUT indicates that a memory usage cutoff of 2GB was exceeded.

Benchmark	No. of Qubits	No. of Gates	QuIDDPro/D		QCSim	
			Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
rc_adder1	16	24	0.44	0.0625	MEM-OUT	MEM-OUT
rc_adder2	16	24	0.44	0.0625	MEM-OUT	MEM-OUT
rc_adder3	16	24	0.44	0.0625	MEM-OUT	MEM-OUT
rc_adder4	16	24	0.44	0.0625	MEM-OUT	MEM-OUT
9sym1	12	29	0.2	0.0586	8.01	128.1
9sym2	12	29	0.2	0.0586	8.02	128.1
9sym3	12	29	0.2	0.0586	8.04	128.1
9sym4	12	29	0.2	0.0586	8	128.1
9sym5	12	29	0.2	0.0586	7.95	128.1
ham15_1	15	148	1.99	0.121	MEM-OUT	MEM-OUT
ham15_2	15	148	2.01	0.121	MEM-OUT	MEM-OUT
ham15_3	15	148	1.99	0.121	MEM-OUT	MEM-OUT

QuIDDPro/D is, in general, sensitive to the arrangement and values of matrix elements, the data indicate that QuIDDPro/D is insensitive to varied inputs on the same circuit for error-free reversible benchmarks. However, QuIDDPro/D still compresses the tremendous amount of redundancy present in these benchmarks.

3.5 Error Correction and Communication

Now we analyze the performance of QuIDDPro/D on simulations that incorporate errors and error correction. We consider some simple benchmarks that encode single qubits into Steane's 7-qubit error-correcting code [22] and some more complex benchmarks that use the Steane code to correct a combination of bit-flip and phase-flip errors in a half-adder and Grover's quantum search algorithm [3]. Secure quantum communication is also considered here because eavesdropping disrupts a quantum channel and can be treated as an error.

The first set of benchmarks, "steaneX" and "steaneZ," each encode a single logical qubit as seven physical qubits with the Steane code and simulate the effect of a probabilistic bit-flip and phase-flip error, respectively [11]. "steaneZ" contains 13 qubits which are initialized to the mixed state $0.866025|0000000000000\rangle + 0.5|0000001000000\rangle$. A combination of gates apply a probabilistic phase-flip on one of the qubits and calculate the error syndrome and error rate. "steaneX" is a 12 qubit version of the same circuit that simulates a probabilistic bit-flip error.

A more complex benchmark that we simulate is a reversible half-adder with three logical qubits that are encoded into twenty one physical qubits with the Steane code. Additionally, three ancillary qubits are used to track the error rate, giving a total circuit size of twenty four qubits. "hadder1_bf1" through "hadder3_bf3" simulate the half-adder with different numbers of bit-flip errors on various physical qubits in the encoding of one of the logical qubit inputs. Similarly, "hadder1_pf1" through "hadder3_pf3" simulate the half-adder with various phase-flip errors.

Another large benchmark we simulate is an instance of Grover's quantum search algorithm. Grover's algorithm searches for a subset of items in an unordered database of N items. Allowed selection criteria are black-box predicates, called oracles, that can be evaluated on any database record. This particular benchmark applies an oracle that searches for one element in a database of four items. It has two logical data qubits and one logical oracle ancillary qubit which are all encoded with the Steane code. Like the half-adder circuit, this results in a total circuit size of twenty four qubits. "grover_s1" simulates the circuit with the encoded qubits in the absence of errors. "grover_s_bf1" and "grover_s_pf1" introduce and correct a bit-flip and phase-flip error, respectively, on one of the physical qubits in the encoding of the logical oracle qubit.

In addition to error modeling and error correction for computational circuits, another important application is secure communication using quantum cryptography. The basic concept is to use a quantum-mechanical phenomenon called entanglement to distribute a shared key. Eavesdropping constitutes a measurement of the quantum state representing the key, disrupting the quantum state. This disruption can be detected by the legitimate communicating parties. Since actual implementations of quantum key distribution have already been demonstrated [1], efficient simulation of these protocols may play a key role in exploring possible improvements. Therefore, we present two benchmarks which implement BB84, one of the earliest quantum key distribution protocols [23]. "bb84Eve" accounts for the case in which an eavesdropper is present (see Fig. 7) and contains 9 qubits, whereas "bb84NoEve" accounts for the case in which no eavesdropper is present and contains 7 qubits. In both circuits, all qubits are traced-over at the end except for two qubits reserved to track whether or not the legitimate communicating parties successfully shared a key (BasesEq) and the error due to eavesdropping (Error).

Performance results for all of these benchmarks are show in Tab. 2. Again, QuIDDPro/D significantly outperforms QCSim on all benchmarks except for "bb84Eve" and "bb84NoEve." The performance of QuIDDPro/D and QCSim is about the same for these benchmarks. The reason is that these benchmarks contain fewer qubits than all of the others. Since each additional qubit doubles the size of an explicit density matrix, QCSim has difficulty simulating

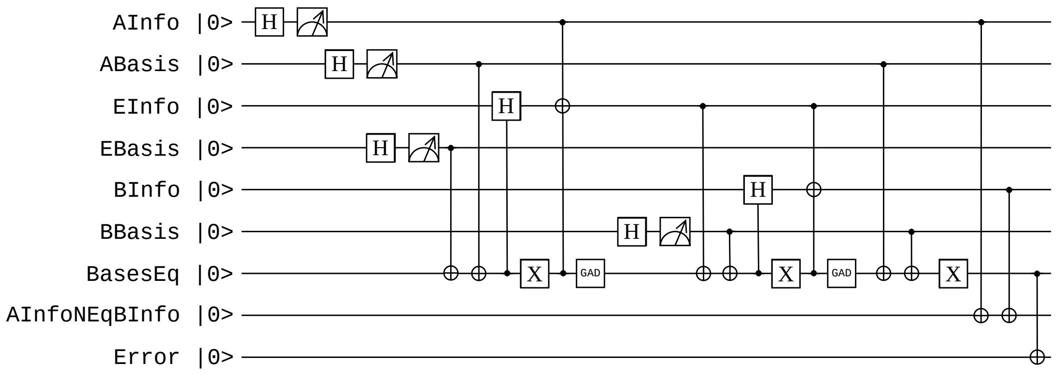

Fig. 7. Quantum circuit for the "bb84Eve" benchmark.

Table 2. Performance results for QCSim and QuIDDPro/D on the error-related benchmarks. MEM-OUT indicates that a memory usage cutoff of 2GB was exceeded.

Benchmark	No. of Qubits	No. of Gates	QuIDDPro/D		QCSim	
			Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
steaneZ	13	143	0.6	0.672	287	512
steaneX	12	120	0.27	0.68	53.2	128
hadder_bf1	24	49	18.3	1.48	MEM-OUT	MEM-OUT
hadder_bf2	24	49	18.7	1.48	MEM-OUT	MEM-OUT
hadder_bf3	24	49	18.7	1.48	MEM-OUT	MEM-OUT
hadder_pf1	24	51	21.2	1.50	MEM-OUT	MEM-OUT
hadder_pf2	24	51	21.2	1.50	MEM-OUT	MEM-OUT
hadder_pf3	24	51	20.7	1.50	MEM-OUT	MEM-OUT
grover_s1	24	50	2301	94.2	MEM-OUT	MEM-OUT
grover_s_bf1	24	71	2208	94.3	MEM-OUT	MEM-OUT
grover_s_pf1	24	73	2258	94.2	MEM-OUT	MEM-OUT
bb84Eve	9	26	0.02	0.129	0.19	2.0
bb84NoEve	7	14	<0.01	0.0313	<0.01	0.152

the larger Steane encoded benchmarks.

3.6 Scalability and Quantum Search

To test scalability with the number of input qubits, we turn to quantum circuits containing a variable number of input qubits. In particular, we reconsider Grover's quantum search algorithm. However, for these instances of quantum search, the qubits are not encoded with the Steane code, and errors are not introduced. The oracle performs the same function as the one described in the last subsection except that the number of data qubits ranges from five to twenty.

Performance results for these circuit benchmarks are shown in Tab. 3. Again, QuIDDPro/D has significantly better performance. These results highlight the fact that QCSim's explicit representation of the density matrix becomes an asymptotic bottleneck as n increases, while QuIDDPro/D's compression of the density matrix and operators scales extremely well.

Table 3. Performance results for QCSim and QuIDDPro/D on the Grover's quantum search benchmark. MEM-OUT indicates that a memory usage cutoff of 2GB was exceeded.

No. of	No. of	QuIDDPro/D		QCSim	
Qubits	Gates	Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
5	32	0.05	0.0234	0.01	0.00781
6	50	0.07	0.0391	0.01	0.0352
7	84	0.11	0.043	0.08	0.152
8	126	0.16	0.0586	0.54	0.625
9	208	0.27	0.0742	3.64	2.50
10	324	0.42	0.0742	23.2	10.0
11	520	0.66	0.0898	151	40.0
12	792	1.03	0.105	933	160
13	1224	1.52	0.141	5900	640
14	1872	2.41	0.125	MEM-OUT	MEM-OUT
15	2828	3.62	0.129	MEM-OUT	MEM-OUT
16	4290	5.55	0.145	MEM-OUT	MEM-OUT
17	6464	8.29	0.152	MEM-OUT	MEM-OUT
18	9690	12.7	0.246	MEM-OUT	MEM-OUT
19	14508	18.8	0.199	MEM-OUT	MEM-OUT
20	21622	28.9	0.203	MEM-OUT	MEM-OUT

4 CONCLUSIONS AND FUTURE WORK

We have described a new graph-based simulation technique that enables efficient density matrix simulation of quantum circuits. We implemented this technique in the QuIDDPro/D simulator [20]. QuIDDPro/D uses the QuIDD data structure to compress redundancy in the gate operators and the density matrix. As a result, the time and memory complexity of QuIDDPro/D varies depending on the structure of the circuit. However, we demonstrated that QuIDDPro/D exhibited superior performance on a set of benchmarks which incorporate qubit errors, mixed states, error correction, quantum communication, and quantum search. This result indicates that there is a great deal of structure in *practical* quantum circuits that graph-based algorithms like those implemented in QuIDDPro/D exploit.

We are currently seeking to further improve quantum circuit simulation. For example, algorithmic improvements directed at specific gates could enhance an existing simulator's performance. With regard to QuIDDPro/D in particular, we are also exploring the possibility of using "read-k" ADDs and edge-valued diagrams (EVDDs) in an attempt to elicit more compression. We are also studying technology-specific circuits for quantum-information processing. Optionally incorporating technology-specific details may lead to simulation results that are more meaningful to physicists building real devices, particularly with regard to error modeling.

Lastly, QuIDD terminal values are computed to within available machine precision. However, the computational complexity of a number of other physical simulation techniques has been greatly reduced by introducing approximations. We are exploring the possibility of introducing approximation into the terminal values such that values which are equal to within some small epsilon are compressed into the same terminal. Such an approximation could serve to reduce the size of QuIDDs quite substantially in the face of small errors.

Acknowledgements

This work is funded by the DARPA QuIST program, an NSF grant, and a DOE HPCS fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements of employers and funding agencies.

References

1. *Start-up makes quantum leap in cryptography*, CNET News.com, November 6, 2003.
2. M. A. Nielsen and I. L. Chuang (2000), *Quantum Computation and Quantum Information*, Cambridge Univ. Press.
3. L. Grover (1997), *Quantum mechanics helps in searching for a needle in a haystack*, Phys. Rev. Lett., 79, pp. 325-328.
4. P. W. Shor (1997), *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM J. of Computing, 26, p. 1484.
5. D. Kielpinski, C. Monroe, and D. J. Wineland (2002), *Architecture for a large-scale ion-trap quantum computer*, Nature, 417, pp. 709-711.
6. C. Monroe (2002), *Quantum information processing with atoms and photons*, Nature, 416, pp. 238-246.
7. B. M. Boghosian and W. Taylor (1997), *Simulating quantum mechanics on a quantum computer*, quant-ph/9701019.
8. G. F. Viamontes, I. L. Markov, and J. P. Hayes (2003), *Improving gate-level simulation of quantum circuits*, Quantum Inf. Processing, Vol. 2, pp. 347-380.
9. G. F. Viamontes, M. Rajagopolan, I. L. Markov, and J. P. Hayes (2003), *Gate-level simulation of quantum circuits*, in Proc. of ACM/IEEE Asia and South-Pacific Design Automation Conf. (ASPDAC), pp. 295-301.
10. D. Greve, *QDD: a quantum computer emulation library*, `http://thegreves.com/david/QDD/qdd.html`.
11. P. E. Black et al., *Quantum compiling and simulation*, `http://hissa.nist.gov/~black/Quantum/`.
12. C.Y. Lee (1959), *Representation of switching circuits by binary decision diagrams*, Bell System Tech. J., 38, pp. 985-999.
13. R. Bryant (1986), *Graph-based algorithms for Boolean function manipulation*, IEEE Trans. on Computers, C35, pp. 677-691.
14. E. Clarke et al. (1996), *Multi-terminal binary decision diagrams and hybrid decision diagrams*, in T. Sasao and M. Fujita, eds, *Representations of Discr. Functions*, pp. 93-108, Kluwer.
15. R. I. Bahar et al. (1997), *Algebraic decision diagrams and their applications*, J. of Formal Methods in System Design, Vol. 10, No. 2/3.
16. K. M. Obenland and A. M. Despain (1998), *A parallel quantum computer simulator*, High Performance Computing.
17. D. Gottesman (1998), *The Heisenberg representation of quantum computers*, quant-ph/9807006.
18. S. Aaronson and D. Gottesman (2004), *Improved Simulation of Stabilizer Circuits*, to appear in Phys. Rev. A, quant-ph/0406196.
19. F. Somenzi (1998), *CUDD: CU Decision Diagram Package*, release 2.3.0, Univ. of Colorado at Boulder.
20. `http://vlsicad.eecs.umich.edu/Quantum/qp/`.
21. D. Maslov, G. Dueck, and N. Scott, *Reversible logic synthesis benchmarks page*, `http://www.cs.uvic.ca/~dmaslov/`.
22. A. M. Steane (1996), *Error-correcting codes in quantum theory*, Phys. Rev. Lett., 77, p. 793.
23. C. H. Bennett and G. Brassard (1984), *Quantum cryptography: public key distribution and coin tossing*, in Proc. of IEEE Intl. Conf. on Computers, Systems, and Signal Processing, pp. 175-179.