Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SIMULATING QUANTUM COMPUTING: QUANTUM EXPRESS

Kareem S. Aggour
Renee Guhde
Melvin K. Simmons

GE Global Research
One Research Circle
Niskayuna, NY 12309, U.S.A.

Michael J. Simon

Lockheed Martin Space Systems Company
P.O. Box 179
Denver, CO 80201, U.S.A.

ABSTRACT

Quantum Computing (QC) research has gained a lot of momentum recently due to several theoretical analyses that indicate that QC is significantly more efficient at solving certain classes of problems than classical computing. While experimental validation will ultimately be required, the primitive nature of current QC hardware leaves practical testing limited to trivial examples. Thus, a robust simulator is needed to study complex QC issues. Most QC simulators model ideal operations, and thus cannot predict the actual time required to execute an algorithm or quantify the effects of errors in the calculation. We have developed a novel QC simulator that models physical hardware implementations. This simulator not only allows the accurate simulation of quantum algorithms on various hardware implementations, but also takes an important step towards providing a framework to determine their true performance and vulnerability to errors.

1 INTRODUCTION

As transistors get smaller, conventional computation will encounter fundamental limits of size, time, and energy. It is anticipated that the effects of statistical thermodynamics and quantum mechanics will be encountered in hardware designed in the next decade. Some efforts have already been made to reduce these effects in conventional architectures. However, Quantum Computing (QC) has significant advantages that other researchers are seeking to employ in new computational devices. QC is an emerging technology that could overcome conventional hardware size and speed limitations, but with hardware, algorithms, and program designs completely unlike those in use today.

Enthusiasm for quantum computing has exploded over the past few years. An important driving force behind this enthusiasm has been proof of the theoretical capabilities of quantum computers to solve problems generally believed to be too compute-intensive for conventional computing approaches. These theoretical capabilities are potentially important to security, where a quantum computer could break cryptography systems in wide use today. However, apart from certain problems of interest to theoretical scientists, there is little understanding of benefits that quantum computers could offer to other problem domains. Gaining such understanding will require the invention and evaluation of new algorithms for those domains.

The objective of this work is to design and develop a simulator in software that can test and quantify the performance of such quantum computing algorithms. This includes simulating both the input of data and the reading of the output of a quantum computer. In addition, it is necessary to be able to determine the amount of time required to complete an operation based on the physical properties of the hardware (as opposed to performing idealized operations) to allow for performance studies. Since no single quantum hardware implementation has been identified as the best, it is important to understand the trade-offs between different implementations. The simulator must also be capable of simulating noise and quantum *decoherence* (state deterioration due to environmental factors), to support error robustness analysis.

Section 2 provides a background of related work in quantum computing and simulation. Section 3 then reviews how a quantum algorithm is executed. Section 4 provides an overview of the simulator, named *Quantum eXpress* (QX). Section 5 gives some initial results from a quantum algorithm simulated using QX. Finally, Section 6 describes future research efforts.

2 BACKGROUND AND PRIOR ART

Quantum computing can theoretically solve certain problems much more efficiently than classical computing because of the intrinsic parallelism of quantum phenomena, parallelism that does not require hardware duplication. For example, while a classical N-bit register can hold any *one* integer between 0 and 2^N–1, an N-quantum bit (qubit) regis-

ter can simultaneously hold *all* (or any given subset) of these 2^N integers in the form of a quantum *superposition* (combination) of states. However, there is a catch—the result of the computation is also in the form of a superposition and any attempt to read out (measure) the result causes the superposition to collapse into just one of the 2^N integer states.

The measurement of a particular quantum state will not deterministically collapse to the same state. The collapse to different states is probabilistic. Therefore, to get more information on the quantum state being measured, one must freshly prepare the superposition many times and perform the measurement each time. The outcomes of each of these "trials" will allow one to build a detailed understanding of the likelihood of the quantum state to collapse to each of the different possible states. For many applications, the inefficiency of these repetitions outweighs the efficiency of the original quantum parallelism. However, for some applications, there is a net gain, even a tremendous gain, in efficiency.

So far, only a few problems are known for which QC offers a net gain in efficiency. One class of these is the Abelian hidden subgroup problem (Lomonaco & Kauffman 2002), which can be solved by QC using exponentially fewer operations than the best known classical algorithm. This class of problem includes Shor's famous code-breaking algorithm (Shor 1997), which is the motivation for much of the current funding for QC research. See Rieffel & Polak (2000) and Ekert et al. (2001) for additional information on quantum computing.

2.1 Quantum Simulation

Even where QC is theoretically more efficient, it is important to understand just how much time a quantum computation would require in practice. Not only does each operation take time, but there is also overhead before and after each computation. Especially important is the overhead due to quantum error correction, which protects the quantum superposition from decoherence. In the absence of decoherence, a state vector (i.e., a general superposition) $|\psi\rangle$ evolves in time during a single operation according to a Schrödinger equation (Griffiths 1995):

$$i\hbar \frac{d}{dt}|\psi\rangle = H|\psi\rangle \tag{1}$$

where the matrix H is known as a *Hamiltonian*, which is some linear Hermitian operator, and $\hbar$ is a physical constant known as Planck's constant. An operator (represented as a matrix) is called *Hermitian* if it is equal to its own transposed complex-conjugate. The vector $|\psi\rangle$, known as a 'ket', is the complex vector associated with state ψ. In the presence of decoherence, and with some approximations, the evolution is described more generally by a "master equation" such as (Louisell 1973):

$$\hbar \frac{d}{dt}\rho = -i[H,\rho] - [V,[V,\rho]] \tag{2}$$

where square brackets denote commutators (the commutator of two operators A and B is denoted [A,B] and is defined as [A,B] = AB – BA) and ρ is a *density matrix*, a natural way of representing a statistical distribution of states (see section 3.2 for details). For a pure state (a completely known superposition), the density matrix has the form (Louisell 1973):

$$\rho = |\psi\rangle\langle\psi| \tag{3}$$

where $\langle\psi|$ is the complex conjugate (also referred to as the 'bra') of $|\psi\rangle$. $|\psi\rangle\langle\psi|$ denotes the outer product of the ket and bra. A state remains pure (no decoherence) if *V=0* in Equation 2, in which case the master equation is equivalent to the Schrödinger equation (Equation 1). Otherwise, the master equation describes, in a statistical sense, the decohering influence of the environment.

Most QC simulators deal only with pure states and thus cannot accommodate evolution according to a master equation. Our simulator uses the density-matrix representation, allowing us to naturally simulate the effects of decoherence. The simulator's ability to accommodate decoherence does come at a price, however. In the density-matrix representation, a state of N qubits is represented by a 2^Nx2^N square matrix instead of a 2^N-element vector. Because our simulator uses the density-matrix representation, it cannot handle as many qubits as a pure-state simulator could. Nevertheless, the density-matrix representation is the most straight-forward way to properly accommodate decoherence in the simulation. See Nielsen & Chuang (2000) for additional information on quantum simulation.

2.2 Existing Simulators

A number of quantum simulators exist that vary in complexity, purpose, state representation and implementation. This is by no means an exhaustive list. The study of quantum simulation began when Deutsch (1985) introduced the notion of a Quantum Turing Machine (QTM). Many QTM simulators have been implemented, including the Quantum Turing Machine Simulator (QTS) developed by Hertel (1999) using Mathematica. The major drawback of using QTMs is finding an appropriate step operator T. The step operator of a QTM is similar to the transition function of a classical TM. Also, the runtime complexity of QTMs can be devastatingly large in comparison to the problem size. Therefore, QTMs are useful when studying quantum complexity theory, but have little importance outside this area.

Most quantum simulators use complex numbers to represent quantum states. Other approaches have been used. QDD, a C++ library developed by Greve (1999), uses binary states which are represented using a Binary Decision Diagram (BDD). This allows QDD to model relatively large quantum states, although this factor limits QDD to representing a "digital" quantum computing model versus an "analog" model. Quantum Bayesian Nets are another common representation of quantum states and are used by Quantum Fog and Qubiter (Tucci 1998). Using Bayesian Nets, quantum systems can be graphically represented. Quantum Fog is used to write quantum computer programs in a high level visual language and Qubiter translates this language to qubit-level instructions. Simulators such as Quantum Fog and Qubiter were built to study quantum Bayesian Nets and explore the possible use of such systems in AI applications on quantum computers.

Quantum Computing Language, developed by Ömer (1998), was the first architecture-independent programming language for quantum computers. It is a quantum computer simulation language designed to work with any qubit-based quantum computer architecture. It is useful in studying quantum computing theory, but cannot capture hardware-specific phenomena.

The Parallel Quantum Simulator, developed by Obenland and Despain (1997), was specifically designed to examine the effects of errors during quantum computation. The simulator was built to analyze the feasibility of quantum computation, as well as its scalability. It only simulates Shor's and Grover's algorithms, and was specifically designed to model an Ion Trap quantum computer as proposed by Cirac and Zoller (1995), making this simulator hardware-specific.

Most quantum computing simulators are designed to simulate a single algorithm on a single type of hardware, most commonly Shor's quantum factoring algorithm and the algorithms needed to implement it (such as the Fourier Transform). Quantum eXpress, on the other hand, can be used to implement any quantum algorithm running on any type of hardware, and can report projected algorithm execution times on a quantum computer. Due to its flexible architecture and use of the density-matrix state representation, QX can easily be augmented to simulate hardware-specific decoherence effects.

3 QUANTUM ALGORITHM EXECUTION

The execution of any algorithm can be divided into three steps: input, evaluation, and output.

3.1 Input

Quantum eXpress requires two primary inputs: (1) a state file and (2) an algorithm file. In the state file a 'base' must be specified, indicating whether the states of the system represent qubits (base 2), qutrits (base 3), or more. While this document will always refer to qubits (2^N), it should be understood that QX can also handle qutrits (3^N) and other higher base states, at the user's discretion. The initial state of the system is represented by a vector of 2^N elements (base 2), where N is the number of distinct qubits.

The base and initial states of Quantum eXpress are specified in an eXtensible Mark-up Language (XML) file using the World Wide Web Consortium's (W3C 2001) Mathematical Mark-up Language (MathML) specification. This file contains sets of vectors defining both the initial states and 'states of interest'. These states are effectively identical in construction, except the initial states also have probability values associated with them indicating the probability that the initial system is in that state. States of interest are defined for the purpose of allowing QX to 'watch' certain states. At any time during the execution of an algorithm, the system can be evaluated to determine the probability of it being in each of these 'watched' states. At the end of the execution of an algorithm, the probabilities of each of the states of interest are displayed to give an indication of the final superposition of the system.

The other required input is a second XML file that describes the quantum algorithm to be executed. The algorithm includes what gate operations to run and on which qubits those operations are performed. This file is kept separate from the initial state file, so that a single algorithm can be easily executed with various initial states.

3.2 Evaluation

Quantum simulators need a succinct method for describing quantum systems and their operations. Since a state is represented as a vector (ket), a statistical ensemble of states is naturally represented as a matrix, referred to as a (probability) density matrix. The density matrix describes the current state of a quantum system. The execution of a quantum algorithm can be viewed as the multiplication of a system's density matrix with other matrices that represent quantum operations.

The initial states and their probabilities determine the initial density matrix of the system using the equation:

$$\rho = \sum_{k=1}^{\#init\ states} p(k) \mid k\rangle\langle k \mid \qquad (4)$$

where $p(k)$ is the probability of state k. Equation 4 allows us to define the initial density matrix ρ of the system.

A typical quantum algorithm can be seen graphically in Figure 1. Each of the lines in Figure 1 represent a distinct qubit, and each box represents an operation performed on those qubits. Each of the operations can also be referred to as a quantum 'gate'.

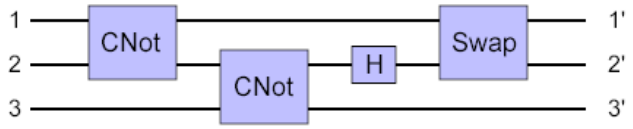

Figure 1: Graphical Representation of an Algorithm

A third input into the system is a set of 'gate' XML files that define the structure of these operations. Each gate is a *unitary operator*, which is defined by a Hamiltonian matrix and a time Δt over which it is applied. This is described by the following equation:

$$U(\Delta t) = e^{-iH\Delta t/\hbar} \tag{5}$$

where U is the unitary operator and H is the Hamiltonian for the gate. As the algorithm is executed, these operators are applied to the density matrix ρ according to the following equation (ignoring decoherence for simplicity):

$$\rho(t+\Delta t) = U(\Delta t)\rho(t)U(\Delta t)^{\mp}. \tag{6}$$

Here $U^{\mp}$ is the Hermitian conjugate of U. The gate XML file contains the matrix H and Δt in MathML format. Each gate may act on a different number of possible qubits, as some apply to single qubits (ex., Not), some apply to two (ex., CNot {Conditional Not} and Swap), and some apply to more. The exact Hamiltonian to apply and for how long depend on (a) the type of gate operation and (b) the type of hardware. For example, a 'Not' gate may have different Hamiltonians depending on the type of hardware modeled.

The exponentiation of the matrix H in Equation 5 is evaluated using the Taylor Series expansion of e^x:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^k}{k!} + \ldots \tag{7}$$

Combining Equations 5 and 7, the unitary operator U may be written as:

$$U(\Delta t) = e^{-iH\Delta t/\hbar} \approx I - \frac{iH\Delta t}{\hbar} - \frac{1}{2!}\left(\frac{H\Delta t}{\hbar}\right)^2 + \frac{i}{3!}\left(\frac{H\Delta t}{\hbar}\right)^3. \tag{8}$$

Note that the approximation of $e^{-iH\Delta t/\hbar}$ uses the Taylor Series expansion of the exponent up to the third (cubic) element. This could be increased to improve the numerical accuracy of the simulator (though it would negatively impact its efficiency). Using the cubic expansion produces numerical errors on the order of 10^{-5}, which for most evaluations is quite sufficient. Equations 4 through 8 illustrate, using the no-decoherence case for simplicity, how the simulator evaluates quantum algorithms.

3.3 Output

At the completion of the evaluation of an algorithm, we wish to understand the final superposition that the system is in. The states of interest are measured against the final density matrix to determine the probability that the system is in each state using the following equation:

$$p(k) = trace \quad (|k\rangle\langle k|\rho) \tag{9}$$

where $p(k)$ is the probability that the final superposition is in state k described by ket $|k\rangle$.

4 SYSTEM ARCHITECTURE

Quantum eXpress has been developed entirely in Java 1.3.01 using Object-Oriented design paradigms. It is platform independent, and has been successfully executed in Windows and UNIX environments. The architecture is divided into four modules: (1) Gate, (2) Software Interface, (3) Quantum Algorithm and (4) Simulator Engine. Figure 2 shows these components and how they interact. Each of the components, described in detail in the following sections, is responsible for a subset of QX functionality.

4.1 Gate Module

The Gate module, (1) in Figure 2, is responsible for reading quantum gates (unitary operators) from XML files and for maintaining those gates for the Software Interface Unitary Operator class to access. When a gate is specified in an algorithm XML file, that gate's name is passed to the Gate Manager class, which is responsible for maintaining in memory all of the available gates. Gate Parser loads the content from gate XML files, interprets them, and creates individual Gate objects based on those file's contents. The Gate classes, initialized by Gate Parser and maintained in Gate Manager, store each gate's Hamiltonian matrix and the time necessary to apply the Hamiltonian for it to have the effect of the desired unitary operator.

When a request for a gate is made, Gate Manager operates as follows:

```
if the requested gate is not in local memory
  read the gate from XML using Gate Parser
  store the new Gate object in local memory
return the Gate from local memory
```

The Gate object that is returned supplies interfaces to the Hamiltonian matrix in two parts: a two-dimensional array containing the real part of the matrix and another two-dimensional array containing the imaginary part.

The Gate module is independent of the other modules. Gate Manager can be executed stand-alone, in which case it will use Gate Parser to read the structure of a gate speci-

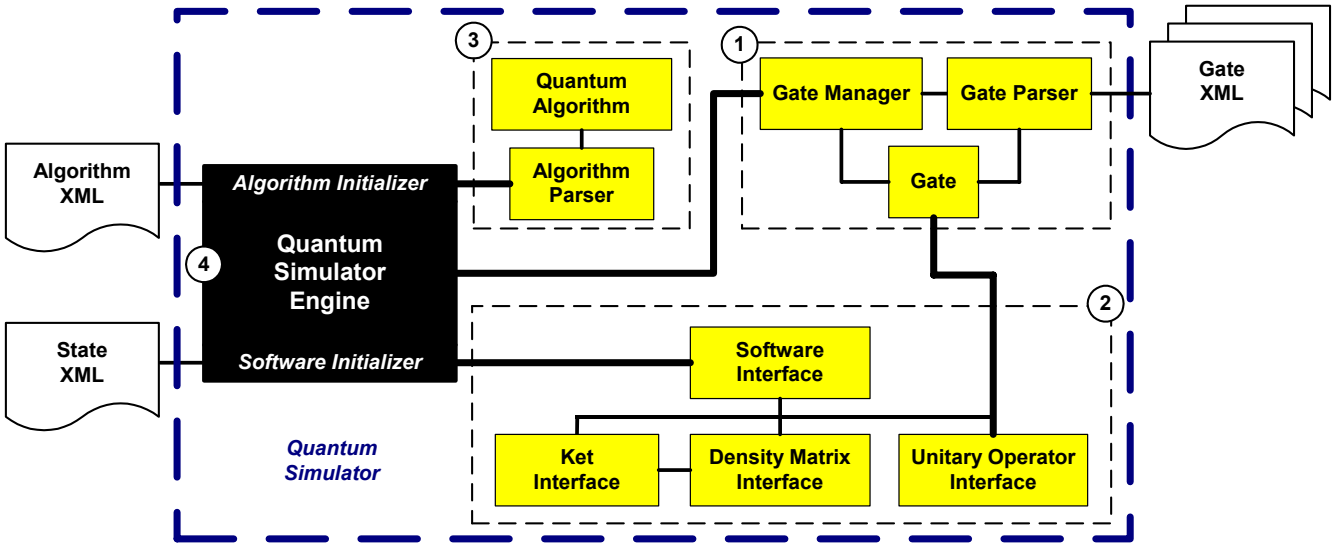

Figure 2: Quantum Simulator Architecture

fied in an XML file, display the composition of the created Gate class, and then exit.

If a new gate is required (either due to implementing a new operator or due to modeling a different type of hardware), a new gate XML file must be written. If this file is written in accordance with the standards specified for Gate Parser, then the new gate will automatically be available to the rest of the simulator through the Gate Manager class. Thus, adding new gates to the simulator requires no code writing or compilation.

4.2 Software Interface Module

The Software Interface module, (2) in Figure 2, defines common interfaces to be used by all classes that define the structure for Density Matrices, Kets, and Unitary Operators in QX. It also provides stub methods for executing unitary operations on a system (Equation 6) and for measuring the probabilities of certain states in the system (Equation 9). The software interface is dependent only upon the Gate module. This dependency is tied through the Unitary Operator interface. The gates define the structures of the unitary operators that act on the density matrix, so the Software Interface module is responsible for converting gates to usable unitary operators (Equation 8).

Unlike the other modules, the Software Interface module defines an interface and not a specific class implementation. Since the majority of the simulator's computational effort is expended in this portion of the code (primarily through large matrix-vector and matrix-matrix multiplications), it is valuable to be able to try out multiple versions of these classes to attempt to construct more efficient implementations without impacting the rest of the code. Therefore, the primary module, the Simulator Engine, is only aware of the interface of the Software Interface module. The class name of the specific Software Interface instance to be used by the simulator is specified in the state XML configuration file and the class is loaded dynamically using Java's reflection (runtime class loading) API.

We have implemented pure Java instances of the Software Interface components. In the future we can easily implement other instances, such as classes that connect to third party applications like Matlab or Mathematica. These tools, which are better optimized than Java for performing large matrix multiplications, could be used to improve the performance of the simulator without having to impact the other modules.

The current pure Java implementation of the Software Interface components required the creation of a number of common mathematical data objects, including complex numbers, vectors, and matrices, as well as a wide array of methods to manipulate those objects.

4.2.1 Complex Numbers

Due to the heavy use of complex numbers in the simulator (every matrix and vector element is a complex number), it was decided not to create a complex number class to represent these objects. Instead, arrays of two Java 'double' values are used (one value for the real part, one for the imaginary), as there is significantly less overhead for Java to create an object of a known type. A custom, static Complex class was developed to simplify the manipulation of these objects by providing a set of methods for adding, subtracting, and multiplying complex numbers, as well as other required methods, but this class is not instantiated and thus does not add overhead to the ubiquitous complex number manipulation.

4.2.2 Vectors

Vectors are required to represent the initial states (kets) of the system to produce the initial density matrix, and to represent the states of interest of the system. A custom Vector class was developed to store and manipulate vectors, including taking inner and outer products between vectors, and multiplying vectors with matrices and scalars.

4.2.3 Matrices

Matrices are required to represent the density matrix as well as unitary operators. A custom Matrix class has been developed to store and manipulate matrices, including adding and multiplying with other matrices and vectors.

Matrix multiplication can take a significant amount of time. The Strassen Algorithm described by Cormen et al. (1989) replaces expensive multiplication operations that occur during matrix multiplication with less-expensive addition and subtraction operations. Therefore, the Strassen Algorithm with Winograd's Variant was implemented in QX to improve the speed of matrix multiplication over the standard row-column multiplication algorithm. Because of the structure of Strassen's Algorithm, it is only really effective for matrices with dimensions greater than 64x64. Also, a performance improvement may not be experienced on all systems, as the algorithm requires more memory than the standard multiplication procedure and so systems with limited or shared memory resources may not experience any performance improvement.

4.3 Quantum Algorithm Module

The Quantum Algorithm module, (3) in Figure 2, is divided into two components—Algorithm Parser and Quantum Algorithm. Together, these classes are responsible for reading in an algorithm XML file and storing the defined algorithm. The gates to execute are defined in this file, along with their order and the qubits they operate on, which altogether make up the algorithm. Algorithm Parser is responsible for parsing the XML file and then creating an instance of the Quantum Algorithm class, which is used to store the information found in the algorithm.

By specifying another algorithm filename in place of a gate name in an algorithm file, that algorithm will be executed as a sub process of the primary algorithm. In this way, separate algorithm files can be written to implement certain functions, which then can be flexibly incorporated into larger algorithms. For example, if algorithm A operates on N qubits and algorithm B operates on M ($M \geq N$), algorithm B can invoke A and cause it to operate on any combination N of the M qubits in algorithm B. Thus, any algorithm of the same qubit size or smaller can be invoked as a sub process of any another algorithm. Figure 3 shows the algorithm that executes a quantum Fourier Transform on three qubits. The dark rectangles represent various one- and two-qubit transformations (Hadamards and Conditional-Conditional-Phase Shifts), and the light rectangle represents another algorithm invoked within this Fourier Transform algorithm. The included algorithm (Swap-3) reverses the order of three qubits. As shown in the file menu in Figure 3, both gates and algorithms can be inserted into an algorithm.

The Quantum Algorithm module is independent of the other modules. Algorithm Parser can be run stand-alone, in which case it will read in a quantum algorithm specified via an XML file, display that algorithm, and then exit.

If a new quantum algorithm is to be implemented, a new algorithm XML file must be written. If this algorithm XML file is written in accordance with the standards specified for Algorithm Parser, then the new algorithm will automatically be available for the simulator to execute. Thus, testing new algorithms in the simulator requires no code writing or compilation.

4.4 Simulator Engine Module

The Simulator Engine module's role, (4) in Figure 2, is to piece the other modules together. It is the main class that is executed, and is responsible for loading the XML configuration files and passing their contents to the appropriate classes for initialization. Once all of the initialization is complete, the engine executes the quantum algorithm.

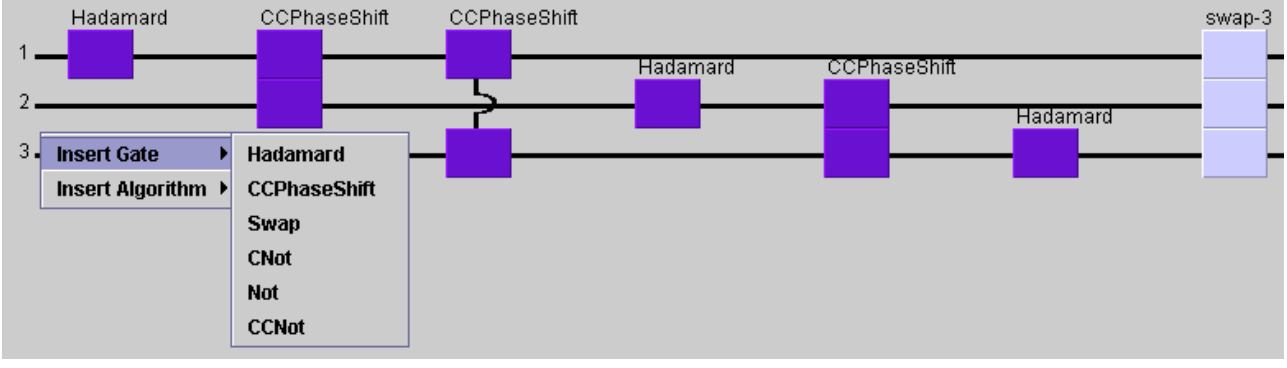

Figure 3: Three Qubit Fourier Transform Algorithm

The simulator engine is dependent upon all of the other modules in Quantum eXpress. It takes as input arguments the state and algorithm XML files, and then calls the Software Interface module to initialize the states of the system and the Quantum Algorithm module to initialize the algorithm. The engine operates by performing the following initialization procedure:

```
If missing a required input parameter
                    (algorithm or state file)
  Exit
Attempt to open each file
If either file is not valid
  Exit
Use the state file to initialize the software
interface
Use the algorithm file to initialize the
quantum algorithm
```

After the initialization process is completed properly, the Simulator Engine module executes the algorithm utilizing the Gate module to load the gates identified in the algorithm. The following procedure is followed to execute the quantum algorithm:

```
Generate the initial density matrix from the
initial states and their probabilities
Display the density matrix
1: While there are steps remaining in the
quantum algorithm
  If the present step is a gate
    Read the required Gate specified in the
    algorithm from the Gate Manager
    Initialize a Unitary Operator for the Gate
    Apply the Unitary Operator to the density
    matrix
    Sum the amount of time spent on the algo-
    rithm so far plus the amount of time spent
    performing the current operation
  Else If the present step is an algorithm
    Open and initialize the specified algo-
    rithm file
    GOTO 1 and iterate over the steps in the
    new algorithm file
Display the final density matrix
Display the final amount of time required to
perform the entire algorithm
Evaluate and display the final probabilities
of all of the states of interest
```

4.5 Graphical User Interface

Also written purely in Java, a Graphical User Interface (GUI) was created to facilitate the development and testing of quantum algorithms. The QX GUI can be used to write both state and algorithm configuration files, and run the engine. It contains four basic components. The first is an editor panel, a screen capture of which can be seen in Figure 4, used to build state configurations. Here users can create a set of initial states, a set of interest states and can specify a base and the number of qubits to use. The next component allows users to build gate configurations by

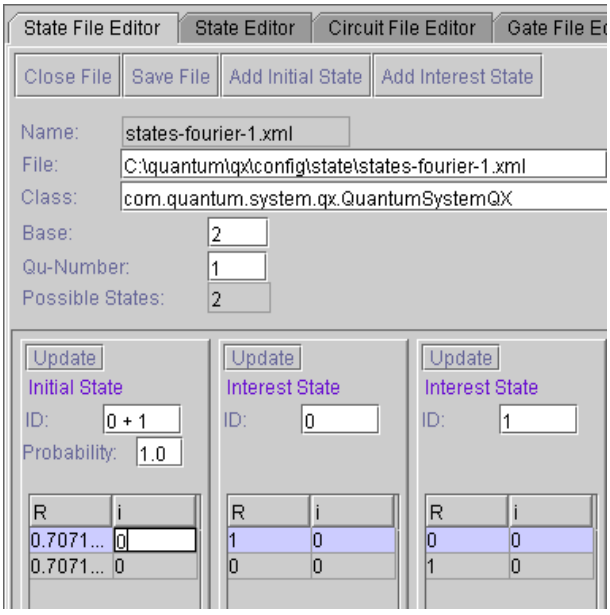

Figure 4: GUI State Editor Panel

specifying Hamiltonian matrices and operating times. The third component, shown in Figure 3, allows users to build an algorithm by inserting gates and other algorithms into a quantum algorithm. The last component allows for the execution of a quantum algorithm, where the user can specify both a state and algorithm configuration.

The QX GUI can directly invoke a local instance of the simulator engine or it can connect to a remote server via the Simple Object Access Protocol (SOAP) to execute the simulator engine on a shared server. Figure 5 displays these two alternatives. Invocation of the engine is configured via an XML properties file read by the GUI at initialization. After executing the algorithm, the simulator engine returns a result set containing the final density matrix, the time required to perform the entire quantum algorithm, and the final probabilities of all of the states of interest. These results are then displayed to the user through the GUI.

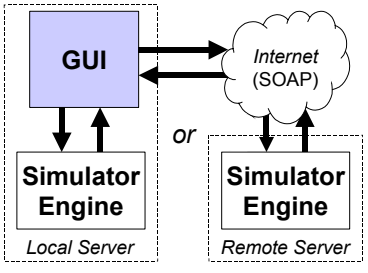

Figure 5: Alternatives for GUI Connecting to Engine

5 SIMULATOR EXPERIMENTS

We have run several experiments with no decoherence to confirm that the simulator functions properly under ideal

quantum circumstances. These tests included simulations of the quantum Fourier Transform (a subroutine in Shor's code-breaking algorithm) on both 3 and 7 qubits (the 3 qubit algorithm can be seen in Figure 3). One of the more interesting and complicated algorithms we have implemented is a generalization of the Fourier Transform, simulating a Nuclear Magnetic Resonance (NMR) device (Nielsen & Chuang 2000). This algorithm involved 333 gate operations (from 5 distinct gates) acting on 7 qubits. In the simulator, this algorithm was logically subdivided into 4 separate algorithm files and one driver file, resulting in a total of 5 XML files used.

In this example, the input was the pure signal shown in Figure 6 with a significant amount of noise added. The signal and noise, which together formed the experiment's input, can be seen in Figure 7.

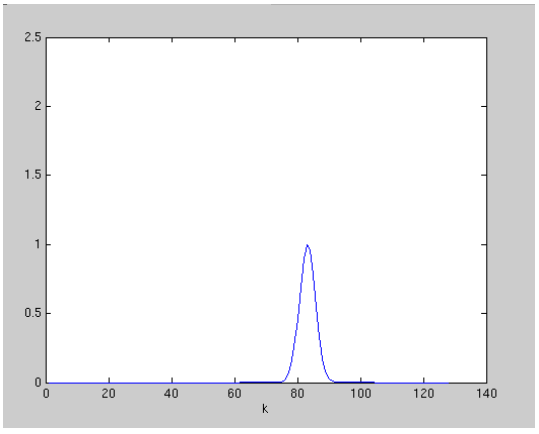

Figure 6: Pure Signal

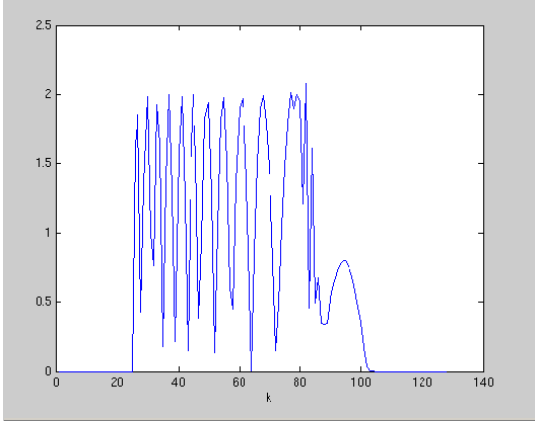

Figure 7: Signal with Noise as Input

To verify the output of the simulator, we first implemented the algorithm in Matlab using ideal unitary operators. The Matlab results can be seen in Figure 8. In Quantum eXpress we used the Hamiltonians and operation times to approximate the unitary operators. The QX output can be seen in Figure 9. Clearly, Figures 8 and 9 are nearly identical, indicating the extremely low numerical error produced by the simulator. This and other examples were implemented to validate QX's execution, and all indicate that the simulator is extremely accurate in its modeling of quantum computing algorithms under ideal circumstances.

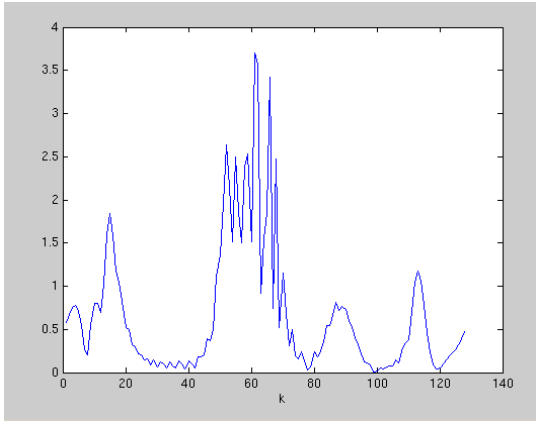

Figure 8: Matlab Output

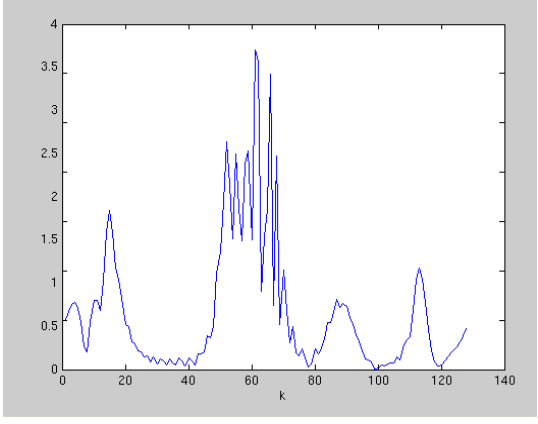

Figure 9: Simulator Output

This work prepares us to extend Quantum eXpress to incorporate the effects of quantum decoherence in algorithms, allowing us to explore the effects of decoherence that are harder to anticipate in quantitative detail.

6 CONCLUSIONS AND FUTURE WORK

Quantum eXpress is a novel simulator that incorporates elements of Object-Oriented software design with principles from quantum computing. Some of its key features are allowing (a) the quantum system to be described via a density matrix representation and (b) quantum operations to be described by physical Hamiltonians. These capabilities allow the execution times of quantum algorithms to be accurately determined. They also allow the study of the impact of errors and decoherence on the algorithms. Quantum eXpress already has the capability to insert errors into a quantum algorithm operation, simulating imperfections in the hardware implementation and gate application dura-

tion. We will soon be extending QX to effectively simulate quantum decoherence within this framework.

QX has a flexible architecture that can be configured entirely through XML files. This enables researchers to explore new algorithms and gate architectures in-silico before they can be physically realized, without having to write any code. To our knowledge, no other quantum simulator has these capabilities. In the future, we will explore porting Quantum eXpress to a reconfigurable computing architecture. This might dramatically decrease simulation run-times and allow for the possibility of processing algorithms in excess of 15 qubits. Future releases of QX will also include visualization capabilities. This will allow for intuitive analysis of quantum information and easy interpretation of algorithm execution results.

We are also in the process of making Quantum eXpress freely available to the public. For information on gaining access to the simulator, please contact the authors.

ACKNOWLEDGMENTS

The authors would like to thank GE Global Research and Lockheed Martin Space Systems Company for their support of this research. We would especially like to thank Randall Schnathorst of Lockheed Martin for his advice and direction. The authors would also like to thank the conference organizers and anonymous reviewers.

REFERENCES

Cirac, J.I. and P. Zoller. 1995. Quantum Computations with Cold Trapped Ions, *Physical Review Letters* 74(20):4091-4094.

Cormen, T.H., C.E. Leiserson and R.L. Rivest. 1989. *Introduction to Algorithms*. New York: The MIT Press.

Deutsch, D. 1985. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer, *Proceedings of the Royal Society of London A* 400:97-117.

Ekert, A., P. Hayden, and H. Inamori. 2001. Basic Concepts in Quantum Computation [online]. Available via <http://lanl.arXiv.org/abs/quant-ph/0011013> [accessed July 1, 2003].

Greve, D. 1999. QDD: A Quantum Computer Emulation Library [online]. Available via <http://qdd.sourceforge.net/> [accessed July 1, 2003].

Griffiths, D.J. 1995. *Introduction to Quantum Mechanics*. New Jersey: Prentice Hall.

Hertel, J. 1999. Quantum Turing Machine Simulator, *The Mathematica Journal* 8(3):440-457.

Lomonaco, S.J. and L.H. Kauffman. 2002. Quantum Hidden Subgroup Algorithms: A Mathematical Perspective [online]. Available via <http://arxiv.org/abs/quant-ph/0201095> [accessed July 1, 2003].

Louisell, W.H., 1973. *Quantum Statistical Properties of Radiation*. New York: John Wiley and Sons.

Nielsen, M. and I. Chuang. 2000. *Quantum Computation and Quantum Information*. New York: Cambridge University Press.

Obenland, K. and A. Despain. 1997. A Parallel Quantum Computer Simulator [online]. Available via <http://arxiv.org/abs/quant-ph/9804039> [accessed July 1, 2003].

Ömer, B. 1998. A Procedural Formalism for Quantum Computing [online]. Available via <http://tph.tuwien.ac.at/~oemer/doc/qcldoc/> [accessed July 1, 2003].

Rieffel, E.G. and W. Polak. 2000. An Introduction to Quantum Computing for Non-Physicists, *ACM Computing Surveys* 32(3):300-335.

Shor, P. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, *SIAM Journal of Computing* 26:1484-1509.

Tucci, R. 1998. How To Compile a Quantum Bayesian Net [online]. Available via <http://lanl.arXiv.org/abs/quant-ph/9805016v1> [accessed July 1, 2003].

W3C. 2001. MathML version 2.0 [online]. Available via <http://www.w3c.org/TR/MathML2/> [accessed July 1, 2003].

AUTHOR BIOGRAPHIES

KAREEM S. AGGOUR is a computer engineer at GE Global Research. His research interests include quantum computing and simulation, and the design of soft computing and artificial intelligence systems. He holds a Masters in Computer and Systems Engineering from Rensselear Polytechnic Institute (RPI) in Troy, NY. He can be reached by e-mail at <aggour@research.ge.com>

RENEE GUHDE is a software engineer at GE Global Research. She holds a Masters in Computer Science from RPI. Her research interests include quantum simulation and artificial intelligence. She can be reached by e-mail at <guhde@research.ge.com>

MELVIN K. SIMMONS is a physicist at GE Global Research. His interests include the physical limitations of advanced computation, biological signaling pathways and simulation, and quantum computation. He holds a PhD in physics from the University of California Berkeley. He can be reached at <simmons@research.ge.com>

MICHAEL J. SIMON is a software and multimedia engineer at Lockheed Martin Space Systems Company. His research interests include quantum computation, reconfigurable computing, scientific visualization and physical simulation. He is pursuing a dual Masters in Computer Science and Engineering at the University of Denver. He can be reached at <michael.j.simon@lmco.com>