Parallel **Si**mulation of **Q**uantum **Co**herent Evolution by **P**arameterized **A**rbitrary **T**ime-dependent **H**amiltonians (PSiQCoPATH)

Eric Fellheimer and Mark Rudner

May 12, 2005

1 Introduction

PSiQCoPATH is the product of the authors' work in Parallel Computing 6.338J at MIT in the Spring of 2005. Our goal was to write a scalably parallel program to efficiently compute the time evolution of finite dimensional quantum systems. In particular, PSiQCoPATH is designed to compute the time evolution of finite dimensional quantum systems with arbitrary time-dependent Hamiltonians in a discrete time simulation to accuracy $\mathcal{O}\left[(\Delta t)^n\right]$. The following section contains an introduction to the theory behind the methods used by PSiQCoPATH, an overview of the program's structure, and finally a few motivating physical examples of systems that can be studied by PSiQCoPATH.

1.1 Physical Background

In quantum mechanics, the state of an isolated physical system is fully characterized by a single unit vector $|\,\psi\,\rangle$ in a complex vector (Hilbert) space. The "ket" symbol $|\quad\rangle$ was originally introduced by Dirac, and plays the role of the vector arrow, i.e. $\vec{v}$, commonly used to denote vectors in linear algebra. Elements of the dual space are denoted by the "bra" symbol $\langle\quad|$. Bras act on kets according to the standard Hermitian inner product, denoted by the contracted symbol $\langle\,\phi\,|\,\psi\,\rangle$.

Physically observable properties of the system are represented by the action of Hermitian operators on the Hilbert space. Following standard convention, linear operators on the Hilbert space will be denoted by the "hat" symbol $\hat{O}$. If a measurement of observable quantity O is made on the system, the only possible results are those corresponding to eigenvalues of $\hat{O}$. The probability of obtaining the result corresponding to the eigenvalue λ_k of $\hat{O}$ is given by $|\langle\,\lambda_k\,|\,\psi\,\rangle|^2$, where $|\,\lambda_k\,\rangle$ is the eigenvector of $\hat{O}$ with eigenvalue λ_k.

An operator of particular importance is the system's Hamiltonian operator $\hat{H}$. The Hamiltonian operator is the quantum generalization of system's "energy" operator. Additionally, as in classical analytical mechanics, the Hamiltonian plays a special role as the generator of time evolution.

In the Schrödinger picture of quantum mechanics, the Hermitian operators corresponding to physical observables are *static*, while the state vector $|\,\psi(t)\,\rangle$ evolves in time. Because $|\,\psi(t)\,\rangle$ remains a unit vector for all time, its evolution is *unitary.* This unitary evolution is governed by the Schrödinger equation

$$i\frac{d}{dt}|\,\psi(t)\,\rangle = \hat{H}|\,\psi(t)\,\rangle \tag{1}$$

where $\hat{H}$ is the Hamiltonian operator mentioned above.

In general, $\hat{H}$ can be time-dependent or time-independent. If $\hat{H}$ does not explicitly depend on time, then (1) can be trivially solved by exponentiating $\hat{H}$. When $\hat{H}$ does explicitly depend on time, the situation is much more complicated.

As a first step toward solving the Schrödinger equation in this case, we introduce the unitary time evolution operator $\hat{U}(t,\, t_0)$ through the relation

$$|\,\psi(t)\,\rangle = \hat{U}(t,\, t_0)\,|\,\psi(t_0)\,\rangle \tag{2}$$

Additionally, $\hat{U}(t,\, t_0)$ satisfies

$$\hat{U}(t,\, t_0) = \hat{U}(t,\, t_1)\,\hat{U}(t_1,\, t_0) \tag{3}$$

and

$$\hat{U}(t_0,\, t) = \hat{U}^{-1}(t,\, t_0). \tag{4}$$

Property (3) leads to a natural way to divide the evolution into a series of discrete time intervals (without approximation):

$$\hat{U}(t_N,\, t_0) = \hat{U}(t_N,\, t_{N-1})\cdots\hat{U}(t_2,\, t_1)\,\hat{U}(t_1,\, t_0) \tag{5}$$

This discretization has not bought us anything yet, as each $\hat{U}(t_{k+1},\, t_k)$ must be found by solving the Schrödinger equation over the time interval $t_k \leq t \leq t_{k+1}$. However, if $\hat{H}(t)$ has analytic time dependence then $\hat{U}(t_k + \Delta t,\, t_k)$ can be expanded in a Taylor series

$$\hat{U}(t_k + \Delta t;\, t_k) = \mathbb{1} + \left.\frac{d\hat{U}}{dt}\right|_{t_k} \Delta t + \frac{1}{2!}\left.\frac{d^2\hat{U}}{dt^2}\right|_{t_k} (\Delta t)^2 + \frac{1}{3!}\left.\frac{d^3\hat{U}}{dt^3}\right|_{t_k} (\Delta t)^3 + \cdots. \tag{6}$$

For sufficiently small Δt, it is a good approximation to truncate this series after only a few terms.

The key to solving the problem is now to relate the derivatives $\left.\frac{d^n\hat{U}}{dt^n}\right|_{t_k}$ to functions of $\hat{H}(t_k)$, $\dot{\hat{H}}(t_k)$, etc. It is a simple matter to check that as a result of definition (2), $\hat{U}(t, t_0)$ obeys a Schödinger equation of its own:

$$i\frac{d}{dt}\hat{U}(t, t_0) = \hat{H}(t)\hat{U}(t, t_0). \tag{7}$$

This relation can be applied successively to replace the derivative terms of all orders in (6). Up to second order, this gives

$$\hat{U}(t_k + \Delta t;\, t_k) = \mathbb{1} \,-\, i\,\hat{H}(t_k)\Delta t \,-\, \frac{1}{2}\left[i\left.\frac{d\hat{H}}{dt}\right|_{t_k} + \hat{H}^2(t_k)\right](\Delta t)^2 \,+\, \mathcal{O}\left[(\Delta t)^3\right]. \tag{8}$$

We have obtained the expression for $\hat{U}(t_k + \Delta t;\, t_k)$ up to third order, but the current implementation of PSiQCoPATH only uses up to second order.

With this expression in hand, we can now calculate the time evolution operator for any finite dimensional quantum system undergoing unitary evolution by *any* possible time-dependent Hamiltonian. In light of our goal to keep PSiQCoPATH as general as possible, we did not want to give up any of this generality. Thus we sought a way for the user to specify his/her particular time-dependent Hamiltonian of interest in a completely general form.

The solution to this problem comes from the fact that for an N-dimensional quantum system, the set of Hermitian operators is spanned by a basis of $d_H = N^2$ linearly independent Hermitian matrices. Because of this, it is possible to expand any arbitrary time dependent Hamiltonian into a linear combination of at most d_H fixed "basis Hamiltonians" with time-dependent coefficients

$$\hat{H}(t) = \sum_{j=1}^{d_H} \alpha_j(t) \hat{H}_j \tag{9}$$

where the operators $\{\hat{H}_j\}$ do *not* depend on time.

While in principle it is possible to construct a Hamiltonian with all d_H coefficients non-zero, in practice it is rarely necessary to require more than 3 or 4. As an additional benefit, this decomposition leads to a computational speedup in the calculation of $\hat{U}(t_{k+1},\, t_k)$ as will be described later.

1.2 High-level System Overview

PSiQCoPATH's work is divided into three phases — input, computation, and output. In the input phase, all necessary data to fully specify the problem are read in to the program from a user-created input file. This file also includes any relevant flags/parameters needed to specify how the calculation should be performed. The specific information contained in the input file will be discussed in the section on implementations below.

In the computation phase, PSiQCoPATH calculates the time evolution of the desired system according to the second order method described above. Depending on the user's intentions, PSiQCoPATH can calculate either the time evolution of a given initial state, i.e. $|\,\psi(t)\,\rangle$ given $|\,\psi(t_0)\,\rangle$, or it can calculate the full time evolution operator (matrix) $\hat{U}(t,\, t_0)$ of the system over a specified time interval. Single state evolution is calculated using a row-distributed data parallel matrix-vector multiplication algorithm, while time evolution matrix calculations are performed using a matrix-multiply version of the parallel prefix algorithm.

Single state evolution can be performed with significantly lower computational cost than the cost of calculating the time evolution matrix calculation for the same system, but yields considerably less information. The time evolution matrix contains *complete* information about the evolution of the system. Once this operator is known, the evolution of any initial state can be found by a single matrix-vector multiplication.

As a special case of the aforementioned modes of operation, PSiQCoPATH can also be used to simulate so-called "quantum circuits." In the quantum circuit model of quantum computation, a quantum system can be used to perform computations by applying a series of unitary "gates" to the system. In reality, the action of each unitary gate arises from evolution under some Hamiltonian as described above. However, if one is not interested in the specifics of the Hamiltonian evolution itself then he/she can compress this information into a sequence of unitary operations exactly as in (5). In quantum circuit mode, PSiQCoPATH calculates the evolution of a system under the influence of a supplied series of unitary gates.

Although it is not currently set up this way, with a small amount of modification this quantum circuit mode can be used to combine the results of a series of time evolution operator calculations. For example, if the desired Hamiltonian is not globally smooth but rather is piecewise smooth, PSiQCoPATH can be used to calculate and save the time evolution matrices over each of the smooth pieces. These results could then be combined in a final parallel prefix operation through the quantum circuit mode apparatus.

Finally, in the output phase, PSiQCoPATH produces several files containing the results of the calculation. This stage of the program is still fairly rough, and only working in detail for full time evolution matrix calculations. Because PSiQCoPATH can in principle be applied to any type of finite dimensional quantum system, it does not perform any data analysis on its results. However, the output files are created in a format that is easily read in by Matlab. We have developed Matlab scripts to analyze the results produced by PSiQCoPATH for each of the test systems we have run. These scripts will be described in more detail in the section on results.

1.3 Applications

While in principle PSiQCoPATH can be used to study any finite dimensional quantum system, finite computer memory, processor speed, and available time place limitations on the size of system that can be studied. Within these limitations, PSiQCoPATH can be applied to any system of choice. As a slightly degenerate case, PSiQCoPATH can be used to simulate quantum circuits as described above. More interestingly, PSiQCoPATH is particularly well suited for studying the nearly adiabatic evolution of quantum systems where $\hat{H}(t)$ is a *slowly varying* function of time.

The adiabatic theorem states that if $\hat{H}(t)$ varies slowly enough then a system initially prepared in the instantaneous ground state[1] of $\hat{H}(t_0)$ will remain in a state close to the instantaneous ground state of $\hat{H}(t)$ for all $t > t_0$. The question of "how slow is slow enough?" is not always easy to answer, but roughly speaking the variations are slow enough when the time scale over which $\hat{H}(t)$ changes by a significant amount is long compared to the timescale τ associated with the minimum energy gap between the ground state and first excited state of the instantaneous Hamiltonian over the course of the evolution, i.e. $\tau \ll \hbar/\Delta E_{min}$.

In addition to the purely intellectual benefit of learning more about quantum dynamics near the adiabatic regime, recent developments in the theory of quantum computation have brought adiabatic evolution into the spotlight of physics research in a more practical context. Farhi et al. (arXiv:quant-ph/0104129) proposed an algorithm for the solution of NP-complete problems via quantum adiabatic evolution in 2001. If this algorithm can be proven to work in all cases, then it will constitute a very significant example of the computational power of quantum systems.

The main idea behind the algorithm is that the solution to the problem can be encoded as the ground state of a cleverly constructed "problem" Hamiltonian. At the outset, it is not easy to construct the ground state as this requires knowledge of the problem's solution. However, it is possible to start the system in the ground state of a different Hamiltonian, the ground state of which is known and can easily be constructed. By slowly "morphing" the initial Hamiltonian into the final "problem" Hamiltonian, the state of the system can be coaxed into the ground state of the problem Hamiltonian. The solution to the problem is then simply read out by measuring the final state of the system in the computational basis.

This is all well and good, but the question of whether or not this algorithm can be used to *efficiently* solve all possible instances of NP-complete problems is still a topic of active research. The problem is that

[1]The ground state is the eigenstate (eigenvector) corresponding to the lowest eigenvalue of $\hat{H}$.

a rigorous proof of the algorithm's efficiency would require a proof that it is always possible to construct an adiabatic switching Hamiltonian with a minimum energy gap that shrinks only algebraically with the number of quantum bits (qubits) involved in the problem. If the gap becomes exponentially small as the number of qubits becomes large, then the running time needed to achieve successful adiabatic evolution to the problem's solution will become exponentially long. As of this time, neither proof of nor counterexample to this gap condition have been found.

2 Implementation

PSiQCoPATH is written in C++ with the MPI library of communication directives to control inter-processor communication. Throughout the design process, we tried to maintain the robustness and adaptability of our code through the techniques of object oriented programming (OOP). To this end, we invested a significant amount of time to the design of proper abstractions in our system. The subject of object oriented programming in the development of PSiQCoPATH will be discussed further in section 5.

2.1 Input Files

The input file format for PSiQCoPATH is simple and straightforward. Whitespace is used as the delimiter between entries. Complex numbers such as $4 + 5i$ are supplied as ordered pairs of their real and imaginary parts in parentheses, i.e. $(4, 5)$.

Although there may be more efficient ways to store the input data, taking the most straightforward approach left us more time to concentrate on the algorithms and parallelization of our code. Furthermore, the simple whitespace delimited text file approach should not lead to any problems if PSiQCoPATH is ported across platforms. Figure 1 is an example of an input file for a very simple test run.

The entry of the first line is used to set the value of the `isAlpha` flag in the code. The value of this flag determines whether the program will run in Hamiltonian evolution mode or quantum circuit mode. In this example, the input "alpha" is used to set isAlpha to 1 and hence to instruct PSiQCoPATH to run in Hamiltonian evolution mode. In this case, the rest of the input is a used to set up the parameters of the desired time-dependent Hamiltonian.

If "qga" is instead specified, the program will run in quantum circuit mode. In this case, the remainder of the input file contains a series of unitary quantum gates to be applied to the system.

The next three lines specify the size of the Hilbert space, the number of terms in the parametric expansion of the Hamiltonian, and T, the number of time steps to be calculated. In the example, the Hilbert space has 2 dimensions, the problem is parametrized by 1 hamiltonian, and 5 time steps are to be run.

In the line that follows, the user specifies the values of t at each of the T time steps. Note that this allows the possibility of non-uniform time steps. In the example, however, the time steps are equally sized.

The next three lines are used to specify the values of $\alpha(t)$, $\dot{\alpha}(t)$, and $\ddot{\alpha}(t)$ for the first basis Hamiltonian at each time step. In the example $\alpha(t) = 1$, and $\dot{\alpha} = \ddot{\alpha} = 0$.

Next, the matrix representation of the first "basis" Hamiltonian $\hat{H}_1$ is supplied. In the example, the two-dimensional identity matrix is used. In the general case where more than one basis Hamiltonian is required, the coefficients and basis Hamiltonians of the remaining pieces of the total Hamiltonian supplied next. The format for each piece is the same – a list of values for the coefficients and their derivatives followed by the matrix representation of the basis Hamiltonian.

The final few lines determine the parameters of the output. First, an output filename is specified. Next, the starting point of the output and frequency at which to store the state of the system are given. In the example, the code is instructed to start the output at the first time step, and only save one state vector.

Finally, the value of the `outputType` flag is set. The value "evolution" is used to instruct the code to perform a full time evolution matrix calculation. Alternatively, the value "state" is used to instruct the code to perform a single state evolution. In this case, the code will then read in the column vector corresponding to the desired initial state, the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ in this example.

```
alpha

2
1
5

1 2 3 4 5


1 1 1 1 1
0 0 0 0 0
0 0 0 0 0

1 0
0 1


alphaOut.out
1
1
1
state
1
0
```

Figure 1: Example of a simple input file. The Hamiltonian is the identity in this case.

Figure 1 is a very simple example. For more realistic simulations of larger systems over longer time intervals, it was necessary to write additional scripts to generate the input files automatically. Examples of such scripts are grover.m (written in Matlab) and genAdQCInFile.cpp (written in C++). These scripts are included in the source code archive available on the PSiQCoPATH website.

2.2 Output

We also tried to maintain simplicity when creating output files. To aid in data analysis, the output files are created in a format easily read in by Matlab.

In Hamiltonian evolution mode, three output files are created. In the first file, the calculated states or matrices at the desired number of output checkpoints are written. A coefficient file is also created, containing the values of t, $\alpha(t)$, $\dot{\alpha}(t)$, and $\ddot{\alpha}(t)$ for each time step. The last file contains the relevant parameters of the calculation and total running time.

2.3 Parallel Prefix

Using decomposition (5) of the introduction, a system's time evolution can be represented by the sequence of "incremental" time evolution operators

$$\left\{\mathbb{1},\ \hat{U}(t_1, t_0),\ \hat{U}(t_2, t_1)\hat{U}(t_1, t_0),\ \hat{U}(t_3, t_2)\hat{U}(t_2, t_1)\hat{U}(t_1, t_0),\ \ldots\right\} \tag{10}$$

The incremental time evolution operator of time step k, $\hat{U}(t_{k+1}, t_k)$, depends only on the local values of $\hat{H}(t_k)$, $\dot{\hat{H}}(t_k)$, etc. As a result, the calculation of each time step's incremental evolution operator can be performed independently of all others. This fact makes the task of generating the set of operators

$$\left\{\hat{U}(t_1, t_0),\ \hat{U}(t_2, t_1),\ \hat{U}(t_3, t_2),\ \ldots\right\}$$

"embarassingly parallel."

Expression (8) shows the explicit form of the incremental evolution operators in terms of the Hamiltonian and its derivatives. In general to any order, this expression involving products of $\hat{H}$ and itself/its derivatives. On the surface, this seems to indicate that calculating the incremental evolution operators is an expensive operation requiring many matrix multiplication operations, each scaling as $\mathcal{O}(N^3)$. However, expansion (9) makes it possible to push the $\mathcal{O}(T \cdot N^3)$ cost of the matrix multiplication operations needed to calculate the incremental evolution operators for the entire run into a single group of matrix multiplications at the beginning of the computation. The number of matrix multiplications at this step depends on the order of the calculation and the number of basis Hamiltonians employed. Under most circumstances, only a few basis Hamiltonians are required and this step comes with minimal computational cost compared to the rest of the calculation.

The advantage comes from the observation that, up to second order for example, $\hat{U}(t_k + \Delta t, t_k)$ is just a linear combination of the static basis Hamiltonians $\{\hat{H}_j\}$ and their binary products $\{\hat{H}_i\hat{H}_j\}$. The coefficients of this linear combination are just combinations of the (real) scalars $\alpha_j(t_k), \dot{\alpha}_j(t_k), \ldots$. Thus as the first "pre-computation" step of the calculation, PSiQCoPATH calculates the set of matrices

$$\hat{B}_{ij} = \hat{H}_i\hat{H}_j. \tag{11}$$

For a third order calculation, we would additionally need to calculate an analogous quantities $\hat{T}_{ijk} = \hat{H}_i\hat{H}_j\hat{H}_k$ and so forth. Once these multiplications have been performed, the calculation of each incremental evolution operator involves only matrix addition operations.

In our code, the simulation is divided into p contiguous blocks of time of roughly equal size, where p is the number of processors in use. Each processor generates and stores the incremental time evolution operators for all time steps within its allotted block of time.

Once the incremental time step evolution operators have been calculated, the final step is to combine them through matrix multiplication into the sequence (10). This is a straightforward application of the parallel prefix algorithm with matrix multiplication playing the role of the associative binary operator. Because matrix multiplication is non-commutative, it is critical to maintain proper operator ordering throughout the calculation.

In general, the number of time steps is much larger than the number of processors available. In this case, the parallel prefix algorithm begins with each processor performing a serial scan operation on its own block of data. Once these local scans are complete, the standard parallel prefix binary tree ascent/descent steps are performed on the top-most (latest time) elements of each processor's data. Finally, each processor other than the root performs a second serial update of its data by right-multiplying each of its own matrices by the top-most element of its earlier time neighbor.

When the number of time steps is much larger than the number of processors, the local serial scan steps dominate the running time to give $\mathcal{O}(2T \cdot N^3/p)$ scaling, where T is the number of time steps, N is the dimension of the system, and p is the number of processors in use. Using an improvement discussed in the section on future work, the prefactor 2 can be reduced to $1+f$, where f is the ratio of output steps requested by the user divided by the total number of time steps. Typically, this number is much less than 1, leading to a nearly 2-fold speedup.

2.4 Row Distributed Multiplication

If the user is interested only in the evolution of a particular initial state, calculation of the full time evolution operator calculation is unnecessarily computationally expensive. That is, a particular initial state $|\,\psi(t_0)\,\rangle$ can be evolved to the state $|\,\psi(t)\,\rangle$ at a time $t > t_0$ at significantly lower computational cost than that of a full time evolution matrix calculation.

Using the method described above, we can calculate the incremental time evolution operators $\hat{U}(t_k + \Delta t,\, t_k)$ to any desired order. The initial state can then be evolved by successively applying the incremental evolution operators to it. In terms of linear algebra, this is simply the problem of calculating: U_1x, U_2U_1x, $U_3U_2U_1x$, $\ldots$, $U_TU_{T-1}\cdots U_1x$ where x is an $N \times 1$ column vector and each U_i is an $N \times N$ unitary matrix.

Rather than using serial/parallel scan operations to first compute all of the matrix products and then multiply by the initial state vector to get the final result, a method that scales as $\mathcal{O}(N^3)$, the calculation can be performed using exclusively matrix-vector multiplication operations that scale as $\mathcal{O}(N^2)$. That is, the calculation starts by computing U_1x. This result can the be used to calculate $U_2U_1x = U_2(U_1x)$, and so on.

Because of its N-fold better scaling properies, we used a data-distributed version of this latter technique of successive matrix vector multiplication to parallelize the calculation of single state evolution. Let p be the the number of processing units and N be the dimension of the Hilbert Space. Let T be the number of matrices in the operation, i.e. the length of the simulation. The algorithm is quite simple:

1. Each processor stores approximately N/p rows of each of the T matrices U_i. Let U_i^k be the "local" matrix stored by processor k. That is, U_i^k contains N/p rows of U_i.
2. Let the vector $curRes = x$, where x is the initial state of the system.
3. for $i = 1$ to T:
    i. Each processor k calculates $localRes = U_i^k \times curRes$
    ii. Each processor broadcasts $localRes$ to all others. The next state vector is reconstructed from each processor's $localRes$ vector and stored in $curRes$

2.5 Runtime analysis of Row Distributed Matrix Vector Multiplications

The running time associated with this algorithm will be:

$$\mathcal{O}\left(\frac{N^2}{p} + C(N,p)\right) T \tag{12}$$

where $C(N,p)$ is the time associated with sharing the local result $localRes$ with all other processors. No MPI documentation we found could give a good estimate on running time of `MPI_Allgather`. Nonetheless, a safe upper bound is that $C(N,p) = \mathcal{O}(p(p-1)N/p) = \mathcal{O}(Np)$. That is, in the worst case, each processor sends $\mathcal{O}(N/p)$ data elements to the $(p-1)$ other processes.

Thus, if N is large, then $C(N,p)$ is dominated by the matrix vector multiplications. In this case, the running time is close to:

$$\mathcal{O}\left(\frac{N^2 T}{p}\right) \tag{13}$$

This result is highly desirable because it as a p-fold speedup over the "ideal" serial algorithm based on matrix-vector multiplication.

3 Results

PSiQCoPATH's first sanity check came in the form of the simplest possible quantum system: a spin-1/2 magnetic moment in a static applied magnetic field. In this situation, the well known exact solution is that the spin precesses about the field with angular frequency ω_L, the Larmoor frequency. Figure 2 shows the Bloch sphere representation of the trajectory of a spin initially aligned in the z-direction in the presence of a magnetic field parallel to the y-direction. From this plot it is clear that the spin exhibits precession as we know it should.

This case is perhaps too simple to use as a test case, though, as the Hamiltonian is independent of time. The second test was to have PSiQCoPATH simulate the behavior of a spin-1/2 magnetic moment in a time-varying magnetic field. The magnetic field varied in time according to the equation $\vec{B}(t) = \sin(\Omega_B t)\,\vec{j} + \cos(\Omega_B t)\,\vec{k}$, where $\vec{j}$ and $\vec{k}$ are unit vectors in the y and z directions, respectively. The Hamiltonian for this system is

$$\hat{H}(t) = \sin(\Omega_B t)\,\hat{\sigma}^y + \cos(\Omega_B t)\,\hat{\sigma}^z \tag{14}$$

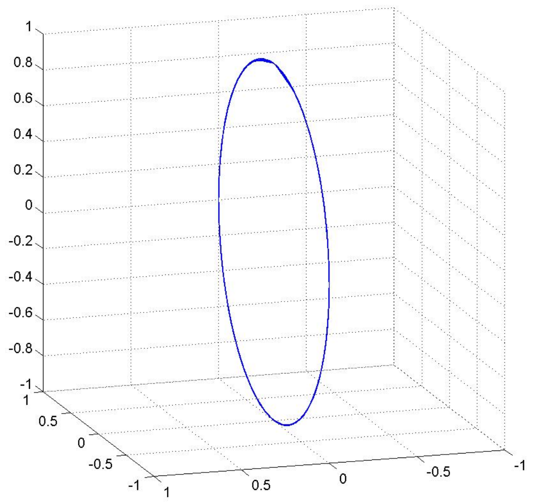

Figure 2: Bloch sphere representation of spin trajectory with $\hat{H} = \hat{\sigma}^y$

where $\hat{\sigma}^y$ and $\hat{\sigma}^z$ are the Pauli spin operators. In the standard basis, the Pauli operators are represented by the matrices

$$\sigma^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma^y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{15}$$

As the initial condition, the spin was aligned with the field along the z-axis. Some of the results of these calculations are shown in figure 3.

In the case where $\Omega_B/\omega_L = 0.01$, the motion is very nearly adiabatic. That is, the spin direction very closely follows the field direction. As Ω_B increases, the trajectory acquires increasingly large cycloid-like wiggles. Although this behavior was not expected beforehand, in retrospect it is easy to understand.

The key to this understanding is a mapping that we discovered between this problem and the trajectory of a point on the rim of a cone rolling on a flat surface (see figure 4). This mapping comes from the fact that a magnetic field generates rotations about its direction with frequency ω_L. In our case, this rotation axis is itself rotating with frequency Ω_B in the yz-plane.

The rolling motion of a cone on flat surface consists of two combined rotations – the cone rotates about its own symmetry axis with frequency ω_a and about the vertical axis through its tip with frequency Ω. Under the condition of rolling without slippage, the combined effect of these two angular velocities is a net angular velocity along the line of contact between the cone and the surface. As the cone rolls, the direction of this instantaneous axis of rotation rotates about the vertical direction with frequency Ω.

Thus the instantaneous axis of rotation in the cone problem has exactly the same behavior as the instantaneous axis of rotation (the magnetic field) in our spin problem. At a given instant, all points on the cone are rotating about the instantaneous axis of rotation, just as at any given instant the spin is precessing about the instantaneous direction of the magnetic field.

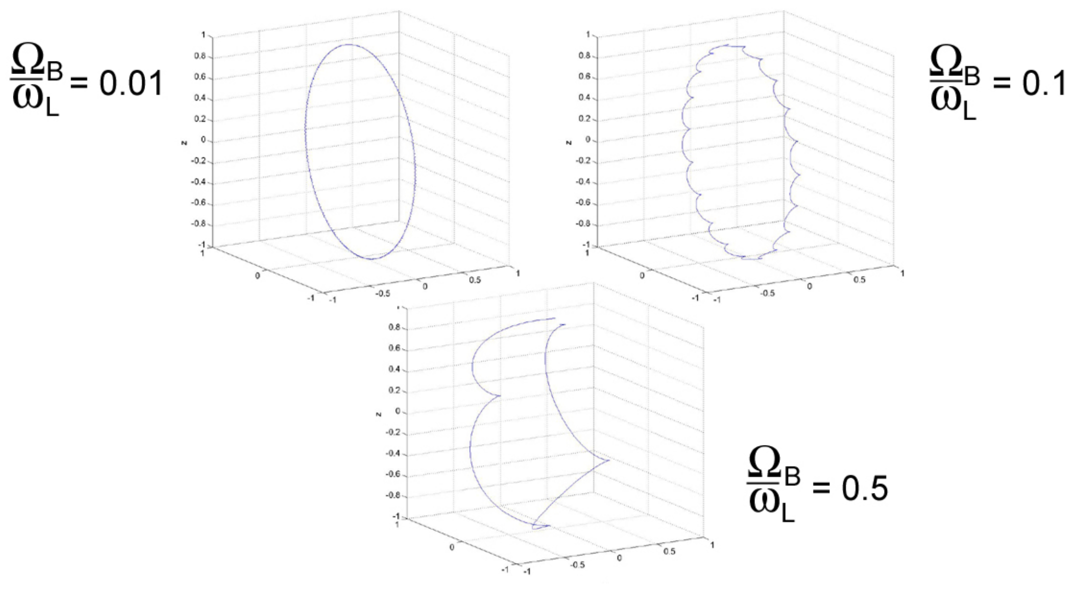

Figure 3: Bloch sphere representation of spin trajectories with $\hat{H} = \sin(\Omega_B t)\hat{\sigma}^y + \cos(\Omega_B t)\hat{\sigma}^z$

The half-angle of the cone corresponding to a particular choice of ω_L and Ω_B in the quantum spin problem can be found by simple trigonometry, and is given by the relation

$$\tan\alpha = \frac{\Omega_B}{\omega_L} \tag{16}$$

Note that $\alpha \to 0$ as $\Omega_B/\omega_L \to 0$. This means that for very slowly changing fields, the maximum amplitude of the spin's deviations from the field direction goes to zero as expected in the adiabatic limit. From this analysis, we see that any initial condition that starts the quantum spin at a point on the rim of the cone associated with the particular values of Ω_B and ω_L of that system will lead to a spin trajectory equivalent to that of the corresponding point on the rim of the associated rolling cone.

Aside from the intellectual interest of this result, it also turns out to be one of the rare cases of a Hamiltonian with non-trivial time dependence for which we have an exact answer to compare with the simulation. The agreement with our results appears to be quite good, though we have not explored it in quantitative detail.

Once this method validation was complete, we used PSiQCoPATH to simulate the solution of a few instances of NP-Complete problems using the method of quantum computation by adiabatic evolution described in the introduction and the reference given there. The particular problem for which we had easy access the proper Hamiltonians was the so-called exact cover problem. Exact cover is a version of satisfiability involving 3 bit clauses of the form

$$z_i + z_j + z_k = 1 \tag{17}$$

where z_i, z_j, and z_k are bits that take on the values 0 or 1.

This problem is described in detail in the paper by Farhi et al. As in that paper, we use a linear

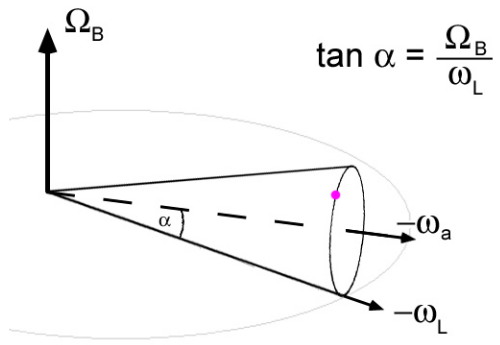

Figure 4: Equivalent problem – the trajectory of a point on the rim of a cone rolling on a flat surface

interpolation between the initial and final Hamiltonians of the form

$$\hat{H}(t) = \left(\frac{t}{T}\right)\hat{H}_p + \left(1 - \frac{t}{T}\right)\hat{H}_0 \tag{18}$$

where $\hat{H}_p$ is the "problem" Hamiltonian and $\hat{H}_0$ is an initial Hamiltonian with ground state

$$|\,\psi_0\,\rangle = \frac{1}{\sqrt{2^N}}\left(|\,0\,\rangle + |\,1\,\rangle\right)^{\otimes N}, \tag{19}$$

and T is the length of the run. Larger values of T correspond to longer runs, and hence slower evolution. Thus the evolution should become increasingly adiabatic as T becomes large.

We would like to thank Daniel Nagaj for supplying us with these Hamiltonians. An example of our results for a 6 qubit instance of Exact Cover are shown in figure 5.

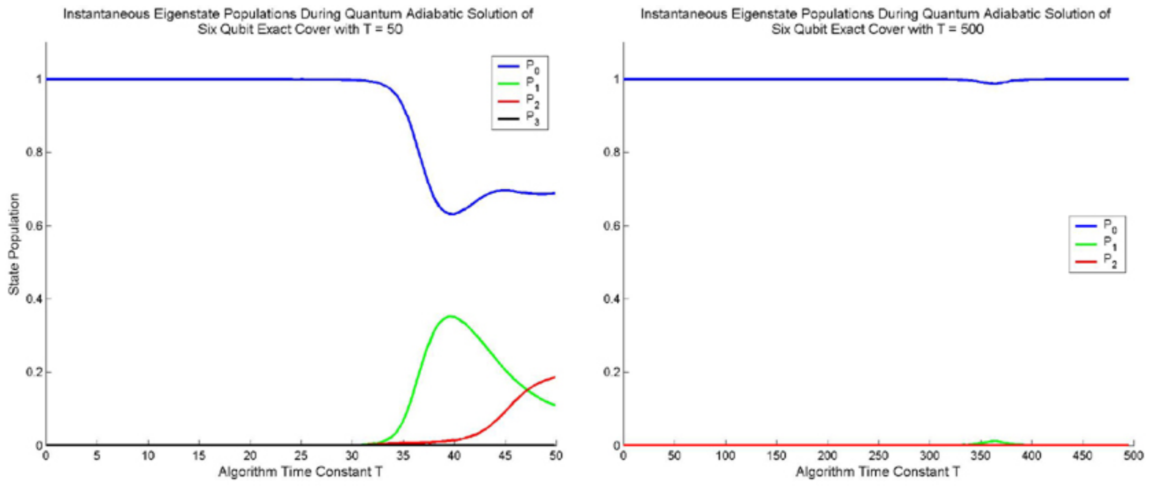

Figure 5: Eigenstate populations vs time for 6 qubit Exact Cover for sweep rates $T = 50$ and $T = 500$

These plots were generated by a Matlab script written by us to parse the PSiQCoPATH output files and perform the desired analysis. The script diagonalizes the system's Hamiltonian at each output time step

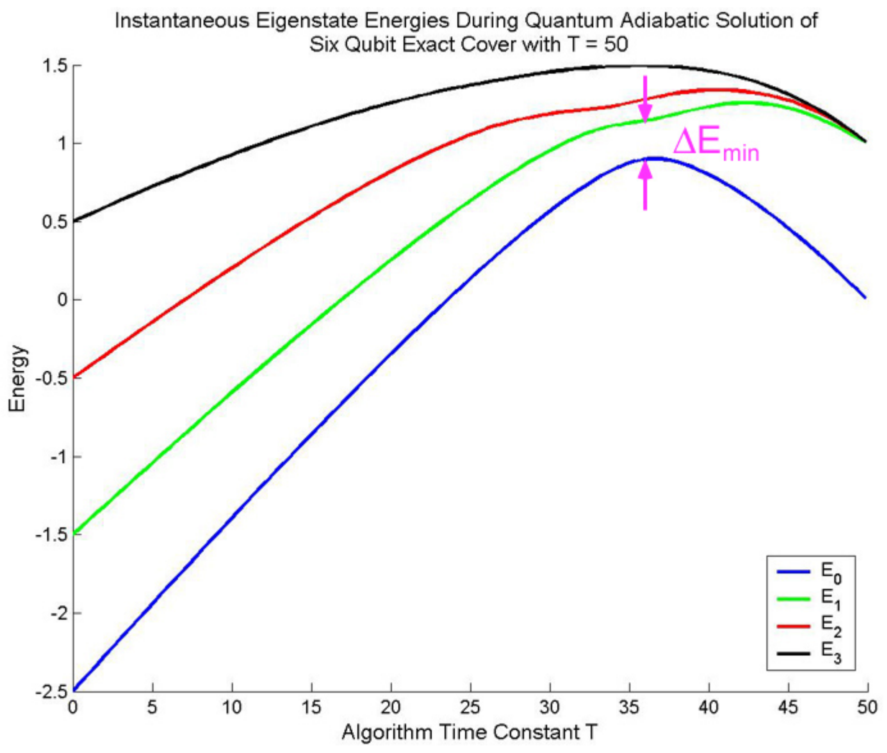

Figure 6: Energy levels vs time for 6 qubit Exact Cover

and transforms the evolved state at the corresponding time step into this eigenbasis. The n^{th} eigenstate population is equal to the square magnitude of the n^{th} component of the evolved state in the instantaneous eigenbasis. For $T = 50$, we see that the probability of finishing in the ground state, i.e. of obtaining the correct solution to the problem, is approximately 60%. When $T = 500$, this probability is very nearly 1.

In figure 6, the energy levels (eigenvalues) of the instantaneous Hamiltonian are plotted over the course of the evolution. Notice that the position of the minimum energy gap is precisely where the ground state population gets "lost" in the fast run. This is what is expected from the considerations of the adiabatic theorem, and makes for a nice confirmation of the theory.

In the end we were only able to test up to 8 qubits. This is really not enough to make progress over the current state of the art in research on this topic, but with the improvements described in the section on future work we should be able to scale up to much higher dimension. All results were obtained from full time evolution operator calculations. Although this calculation is in a sense overkill for what we have used it for in the analysis, the full time evolution matrix could be used to find the success probability in the case where the initial state is actually a "mixed-state" due to thermal noise and/or uncertain preparation. This is an interesting situation to look at from a practical point of in view as this is a more realistic picture of what the situation will be like in real physical implementations.

With the distributed data approach of the matrix-vector multiplication single state evolution algorithm, we should be able to reach even larger systems. Memory usage is significantly lower in that case, and the distribution across processors should allow us to handle much larger matrices without having to reach beyond

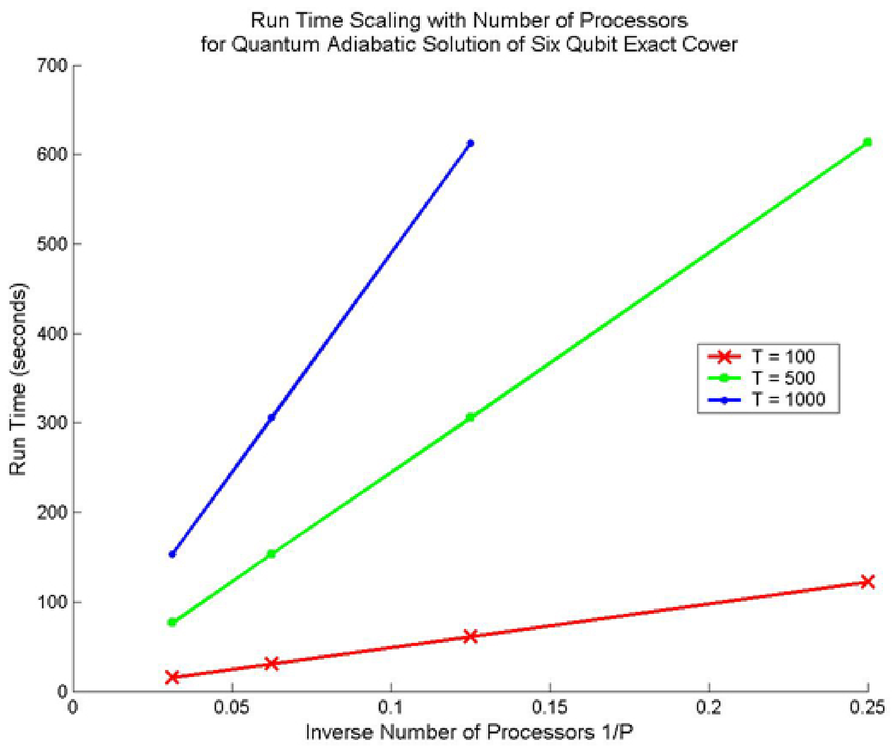

Figure 7: Running time vs inverse number of processors

the cache/fast memory. This code did not become operational until after the tests described, so we only have detailed results for the full time evolution operator calculations.

Throughout these runs we also kept track of PSiQCoPATH's performance in terms of running time. Running time as a function of simulation length and number of processors is plotted in figure 7. The trends are very nearly linear in both $1/p$ and T, confirming our projected scaling rules.

4 Future Work

One area where PSiQCoPATH could be improved siginificantly is in the area of memory management during parallel prefix runs. In its current form, the program first calculates and stores the incremental time evolution operators for *all* time steps. Each processor then performs a serial scan operation on its own block of data. The storage of all the incremental time evolution operators accounts for the vast majority of PSiQCoPATH's memory usage. This memory usage, in turn, is what limits the size of problem we are able to run.

We recently realized that it is not necessary to ever have all of the incremental time evolution operators stored at one time. In general, the number of output timesteps requested by the user is much less than the number of actual timesteps performed in the evolution (by a factor of perhaps 1000). Rather than storing all of the incremental operators, all we really need is that fraction of them corresponding to the much coarser output time step.

A much more efficent procedure would be to partially combine the first two steps in the following way. Let $\mathcal{N}$ be ratio of the total number of time steps to the number of output steps requested. Instead of storing every $\hat{U}(t_k + \Delta t,\, t_k)$, we really only need to store each $\hat{U}_{\mathcal{N}}(t_k + \mathcal{N}\Delta t,\, t_k) = \hat{U}(t_k + \mathcal{N}\Delta t,\, t_k + (\mathcal{N} - 1)\Delta t) \cdots \hat{U}(t_k + \Delta t,\, t_k)$. We can build up $\hat{U}_{\mathcal{N}}$ by multiplying by each successive incremental time evolution operator as it is generated. Once $\mathcal{N}$ time steps have been calculated and combined, that $\hat{U}_{\mathcal{N}}$ can be stored in memory and the next one started. In this way, the memory requirements of the program will be cut roughly by a factor of $\mathcal{N}^{-1}$.

As an additional benefit, the final local update step will also be shortened by a factor of $\mathcal{N}^{-1}$. Overall this leads to a speedup of approximately $2/(1 + \mathcal{N}^{-1})$ in the projected running time.

Also, we did not fully explore alternative basic arithmetic algorithms that could speedup the system. For instance, Strassen's algorithm for matrix multiplication runs asymptotically faster than n^3. However, this algorithm has significantly different memory usage. Also, its overhead is much larger than standard matrix multiply. Thus, simply using Strassen's algorithm would not necessarily be an improvement. Possible changes of this sort are worth considering, and can easily be integrated and tested in our code due to its object-based construction.

5 Case Study on Object Oriented Parallel Programming

Throughout the course on parallel computing, most of the skeletal code we were given was not object-oriented. We made it a point to make our code as object-oriented as possible. Over the course of the project we found that many aspects of object oriented programming carry over directly to the parallel setting, but we also encountered some new challenges unique to the playing field of parallelism.

At a high level, the big advantage of object-oriented programming is the power of abstraction. We employed such abstraction with objects that we knew would be parallelized. For instance, in the `ComplexMatrix` class, we have methods such as `send`, `receive`, and `rowDist` to communicate matrix objects between processors.

The `send` and `receive` methods were relatively simple and worked well. These routines simply send entire matrices to/from other processors via the MPI. An alternative approach would have been to define a new MPI datatype for objects of the ComplexMatrix class, but we found it much simpler to simply add these communication methods in the class itself.

The `rowDist` method was somewhat awkward. Its goal was to distribute the data of a matrix stored completely on a single processor to all other processors in row-wise fashion. Similar to an `MPI_Bcast` command, every processor calls `rowDist`. Before calling this command, however, each processor had to determine how many rows it would store. This added an additional step to the method, but was not an insurmountable challenge.

Overall, we found using objected oriented techniques to be very useful in maintaining abstractions in an MPI-driven parallel program. However, classes should be designed *with paralellism in mind* to achieve maximum robustness. It is not always so easy to simply sprinkle in some MPI routines into a class after it has already been designed.

6 Conclusion

Due to the growing importance of quantum systems to the information technology industry as well as their intrinsic scientific interest, it is important to be able to simulate quantum dynamics as accurately and efficiently as possible. We have developed PSiQCoPATH in an attempt to apply the power of parallel computing to the accurate simulation of a very wide class of quantum problems. Validation was performed on several test systems and showed excellent agreement with analytical predictions.

While there are certainly issues we could have considered in greater detail, this project has been a huge success as a proof of concept. Quantum systems can indeed be simulated on parallel machines efficiently as a means for learning more about the quantum mechanics of various physical systems.