Parallelization of the QC-lib Quantum Computer Simulator Library

Ian Glendinning*

March 19, 2003

*work carried out jointly with Bernhard Ömer

Outline

- Introduction
- QC-lib
- Parallelization
- Performance Measurements
- Conclusions and Future Work

Introduction

- Why Quantum Computer Simulators?
- Why Parallelize?
- Representation of Qubits, Registers and Gates

Why Quantum Computer Simulators?

- Quantum computer hardware is not currently available outside physics research labs
- Simulators are currently the only way to experiment with more than 7 qubits
- Debugging quantum programs
  - Can examine quantum machine state

Why Parallelize?

- Problem
  - The execution time and memory requirements of simulators increase exponentially with the number of qubits
- Parallelization
  - Allows more qubits to be simulated in the same time
  - Or the same number of qubits to be simulated in less time
  - Only a modest increase, due to the exponential increase in resources required, but nevertheless could be useful

Representation of Qubits, Registers and Gates

- The state of a qubit can be represented as a two-dimensional complex vector of length 1
- The state of an n-qubit register can be represented as an 2^n-dimensional complex vector of length 1
- Any n-qubit gate (operator) can be represented as a $2^n \times 2^n$ unitary matrix, i.e. a complex matrix U with the property that $U^\dagger U = I$
- The operation of a gate on a quantum register is implemented by matrix multiplication

Representation of Qubits

It is conventional to define the computational basis vectors $|0\rangle$ and $|1\rangle$ as:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

So a general qubit state is:

$$\alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

subject to the *normalization* condition $|\alpha|^2 + |\beta|^2 = 1$ where $\alpha, \beta \in \mathbf{C}$

Representation of Quantum Registers

The state of a two qubit register can be represented as a four-dimensional complex vector of length 1. The computational basis vectors are:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Representation of Quantum Registers

So a general two-qubit state is of the form:

$$\sum_{i=0}^{3} \alpha_i |i\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix}$$

subject to the normalization condition:

$$\sum_{i=0}^{3} |\alpha_i|^2 = 1$$

where $\alpha_i \in \mathbf{C}$

Representation of Quantum Registers

An n-qubit register has 2^n computational basis states. If we call the ith basis state $|i\rangle$, where $0 \leq i \leq 2^n - 1$, then a general n-qubit state has the form:

$$\sum_{i=0}^{2^n-1} \alpha_i |i\rangle$$

where α_i is the ith component of the complex vector representing the state, $|\alpha_i|^2$ is the probability that measurement will give the value i, and the normalization condition is:

$$\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$$

Representation of Quantum Gates

- The Quantum NOT Gate
- The Hadamard Gate
- The Controlled-NOT Gate

The Quantum NOT Gate

The only non-trivial classical single-bit gate is the NOT gate, defined by its 'truth table' in which $0 \to 1$ and $1 \to 0$.

We can define an analagous quantum NOT gate, conventionally written as an operator X, such that $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$

This does not tell us what happens to superpositions of the states $|0\rangle$ and $|1\rangle$. In fact, the quantum NOT gate acts *linearly*, i.e.:

$$\begin{aligned} X(\alpha|0\rangle + \beta|1\rangle) &= X\alpha|0\rangle + X\beta|1\rangle \\ &= \alpha X|0\rangle + \beta X|1\rangle \\ &= \alpha|1\rangle + \beta|0\rangle \end{aligned}$$

Matrix Representation

Linearity is a general property of quantum mechanics, and allows quantum gates to be represented in matrix form. For X we have:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

In matrix-vector notation, applying X to state $\alpha|0\rangle + \beta|1\rangle$ we have:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$$

Notice that X preserves the normalization, as do *all* quantum gates.

The Hadamard Gate

In contrast to the classical case, there are many non-trivial single-qubit gates. One of the most useful is the *Hadamard* gate

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Acting on the Basis states $|0\rangle$ and $|1\rangle$:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

and

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

The Controlled-NOT Gate

The prototypical multi-qubit gate is the *controlled*-NOT or CNOT gate.

It has two inputs, known as the *control* and *target* qubits, and two outputs.

If the control qubit is set to 0, the target qubit is unchanged, and if the control qubit is set to 1, the target qubit is flipped ($|c, t\rangle$):

$$|00\rangle \rightarrow |00\rangle; \ |01\rangle \rightarrow |01\rangle; \ |10\rangle \rightarrow |11\rangle; \ |11\rangle \rightarrow |10\rangle$$

CNOT is a generalization of the classical XOR gate, since its action may be summarized as $|x, y\rangle \rightarrow |x, y \oplus x\rangle$, where $\oplus$ is addition modulo two, which is the same as XOR.

Matrix Representation

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

QC-lib

- A C++ library for the simulation of quantum computers at an abstract functional level, written by Bernhard Ömer
- QC-lib's main features are:
  - Basis vectors of arbitrary length (not limited to word length)
  - Efficient representation of quantum states using hash tables
  - Nesting of substates and arbitrary combinations of qubits
  - Composition and tensor product of operators (gates)
  - Easy extendibility by class inheritance

QC-lib Classes

The top-level classes in QC-lib are:

- **bitvec** - arbitrary length bit vectors which represent basis states
- **term** - a basis vector with a complex amplitude
- **termlist** - internal representation of a quantum state
- **quState** - user class for quantum states
- **opOperator** - user class for quantum operators

Class bitvec

- Data structure

```
class bitvec {
  int len;
  union {
    word *pvec;
    word vec;
  };
```

- Member functions
  - (contructor, destructor), assignment, length, getbits, setbits, concatenation, bitwise logical operations etc...

Class term

- Data structure

```
class term {
  bitvec _vect;
  complx _ampl;
```

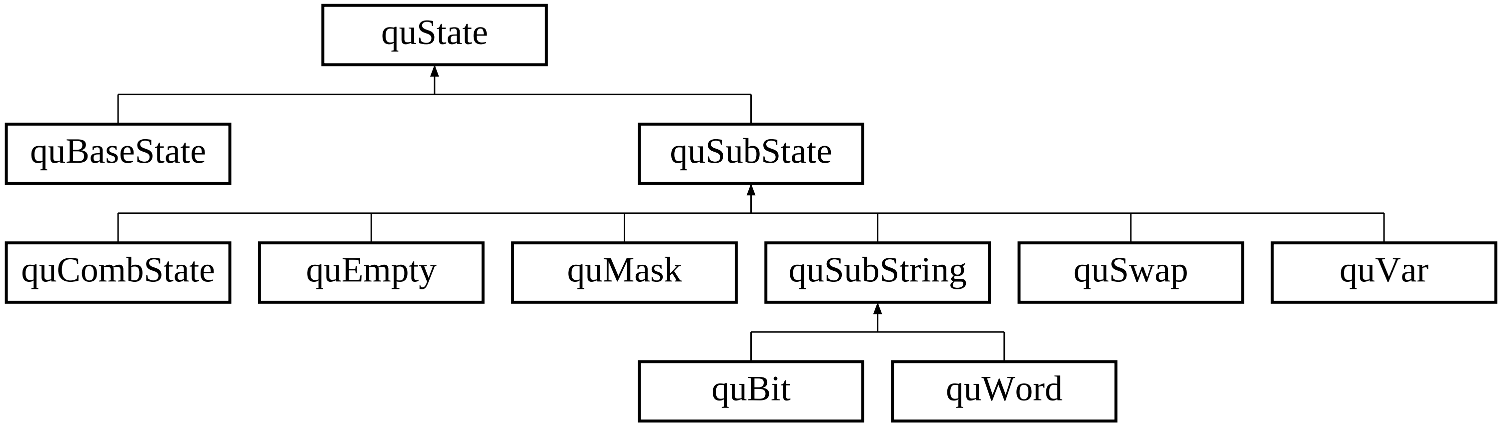
- Member functions
  - assignment, no. of bits, get and set vector and its amplitude

Class termlist

- Data structure
  - Linear array in combination with a hash table
  - The array and hash table are dynamically doubled in size if the array fills up
  - Only terms with non-zero amplitudes are stored
- Member functions
  - assignment, no. of terms, add terms, i^{th} term from list, clear list, and a couple of others

Class quState

- Inheritance Diagram



- A `quBaseState` object represents the state of quantum memory
- Subregisters of class `quSubState` etc. can be allocated from it

Class quBaseState

- Data structure

```
class quBaseState : public quState {
  termlist *pterms;
  termlist *ptermbuf;
```

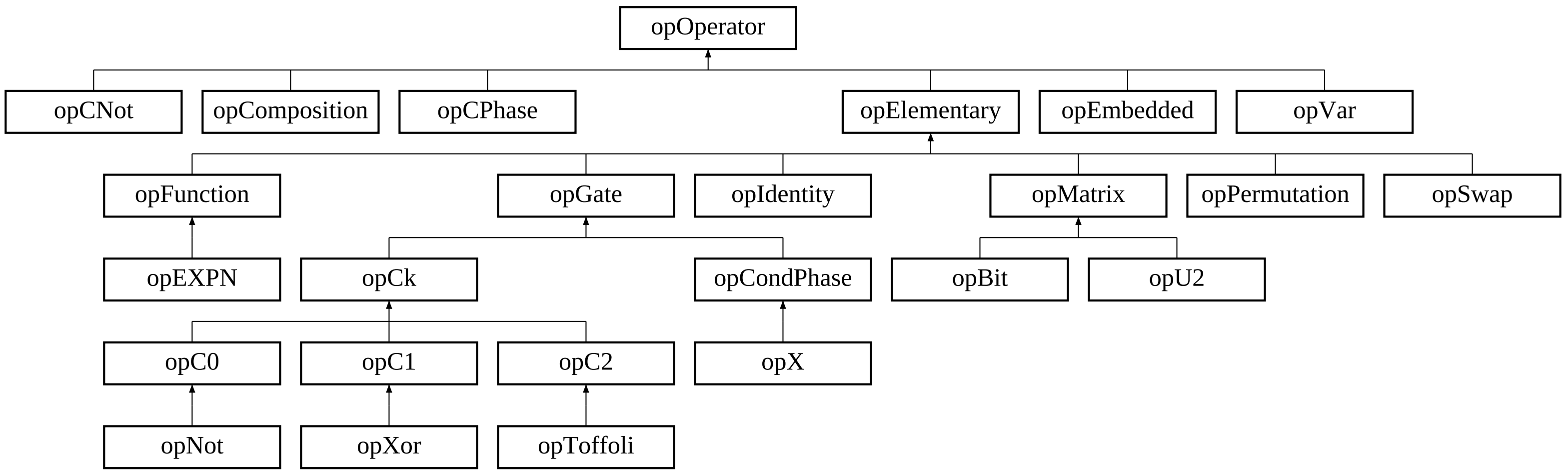
- One termlist for the current state and one to accumulate the result of an operation on it

Operators

- Any n-qubit operator can be represented as a $2^n \times 2^n$ complex matrix
- Class `opMatrix` implements this, storing the non-zero elements of each row in an array of lists
- However, most operators are simpler, only working on a few qubits, or substituting one base vector for another, with or without a phase factor
- QC-lib provides classes for a number of such special cases

Class opOperator

- Inheritance Diagram



Some Subclasses of opOperator

- opEmbedded
  $OP = I(n) \otimes U \otimes I(m)$
- opPermutation
  $U|i\rangle = c_i|j_i\rangle$
- opFunction
  $U|i, 0\rangle = |i, f(i)\rangle$
- opGate
  $U|i\rangle = c(i)|f(i)\rangle$

Example C++ Progam using QC-lib

```
// Implement Hadamard transform
void H(quState& qs) {
  opBit H1(1,1,1,-1,sqrt(0.5)); // define Hadamard gate H
  // loop over qubits in register
  for(int i=0; i<qs.mapbits(); i++) {
    quBit qbit(i,qs);           // create one-qubit substate
    H1.apply(qbit);             // apply H to the qubit
  }
}
quBaseState qs(20); // allocate 20-qubit register
H(qs);              // apply the Hadamard transform to qs
```

The Same Program in QCL

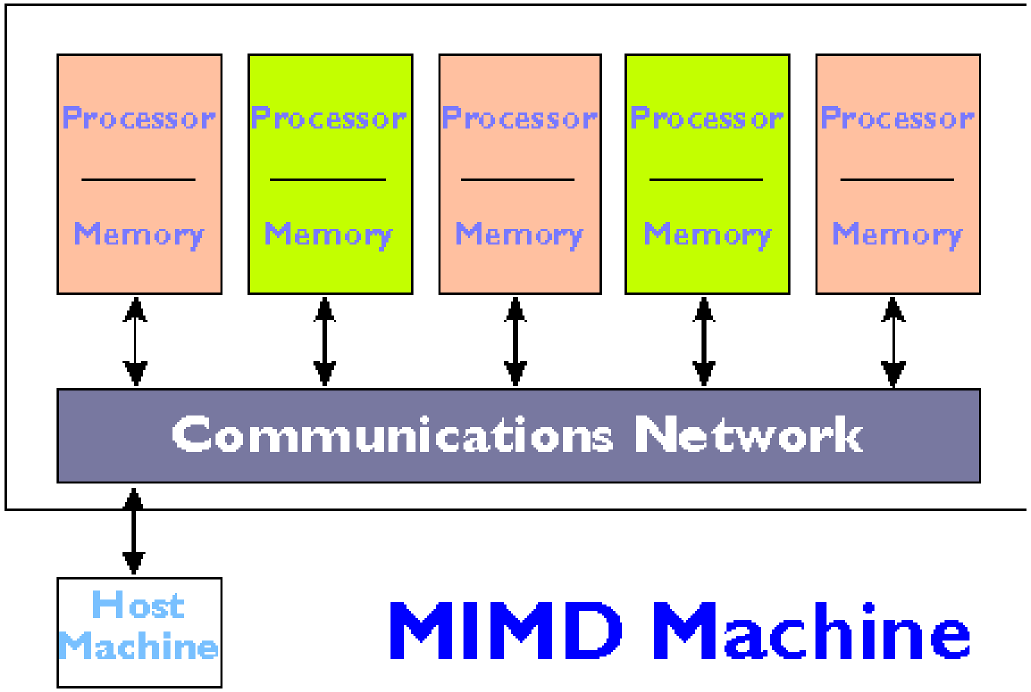
```
qureg qs[20];
H(qs);
```

Parallelization

- Target Architecture
- Parallelization Strategy
- Single-Qubit Operators
- C++ and MPI
- Current Status - Operators Parallelized
- Implementation of Grover's Algorithm

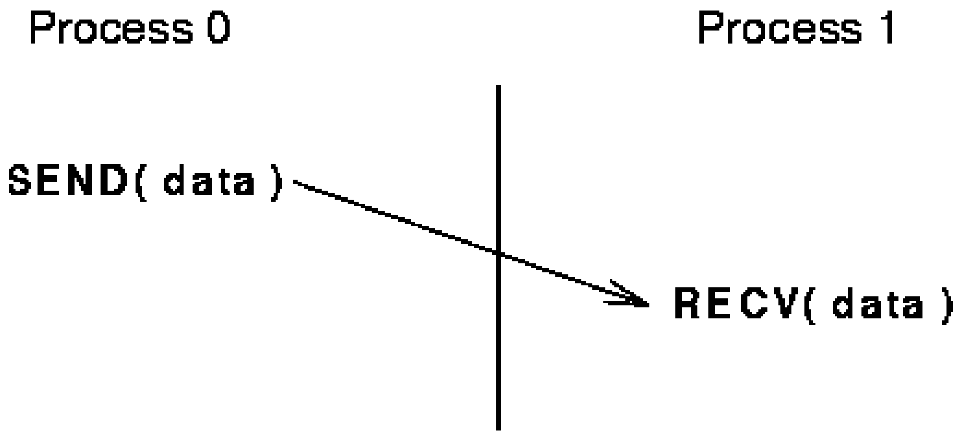
Target Architecture

- Hardware: Distributed Memory MIMD



Target Architecture

- Software: Message-Passing
  - Exchange of data is cooperative
  - Data is both explicitly sent and received



  - Use the MPI Message Passing Interface

Parallelization Strategy

- Distribute the representation of the quantum memory
- Each processor stores a subset of the terms
- SPMD - each processor running the same program
- Operators are applied only to local terms (owner computes)
- The result of the local operations in general includes terms that are not owned by the processor, which need to be communicated to their owning processors

Data Distribution Scheme

Least significant bits determine processor. E.g. for four processors:

Basis State	Processor
$\|0\mathbf{00}\rangle$	0
$\|0\mathbf{01}\rangle$	1
$\|0\mathbf{10}\rangle$	2
$\|0\mathbf{11}\rangle$	3
$\|1\mathbf{00}\rangle$	0
$\|1\mathbf{01}\rangle$	1
$\|1\mathbf{10}\rangle$	2
$\|1\mathbf{11}\rangle$	3

Single-Qubit Operators

Consider a 3-qubit register and four processors, with the register initially in the state $|000\rangle$, which is stored on processor 0:

$$\begin{aligned}(H|0\rangle)|00\rangle &= (1/\sqrt{2})(|0\rangle + |1\rangle)|00\rangle \\ &= (1/\sqrt{2})(|000\rangle + |100\rangle)\end{aligned}$$

Both new states are still on processor 0 - no communication needed.

$$\begin{aligned}|0\rangle(H|0\rangle)|0\rangle &= |0\rangle(1/\sqrt{2})(|0\rangle + |1\rangle)|0\rangle \\ &= (1/\sqrt{2})(|000\rangle + |010\rangle)\end{aligned}$$

$|010\rangle$ is on processor 2, so it's amplitude needs communicating.

Communication Pattern

For single-qubit operators:

- For each term in the initial state, at most two terms are created, one local, and one remotely owned
- For each processor, all the remotely owned terms are owned by one other processor
- Symmetrically, the remote processor potentially creates terms that are owned by the local one
- Communication is needed between disjoint pairs of processors

Parallelization of Single-Qubit Operators

- Accumulate local and remote terms separately, then exchange remote terms with remote processor, and accumulate the received terms
- Representation of the distributed quantum memory encapsulated in class `quBaseState`

Operator Invocation mechanism

- All operators have an `apply()` member function that takes a `quState` argument
- `apply()` is implemented differently for various operator subclasses
- The largest subclass is `opElementary`, all of whose subclasses apart from `opIdentity` use the same `apply()` function
  - `opElementary::apply()` uses three member functions in class `quState`, namely `opbegin()`, `opadd()` and `opend()`

Operator Invocation mechanism

```
void quBaseState::opbegin() {
  ptermbuf->clear();
}
void quBaseState::opadd(const bitvec& v,const complx& z) {
  ptermbuf->add(v,z);
}
void quBaseState::opend() {
  termlist *pl;
  pl=pterms; pterms=ptermbuf; ptermbuf=pl;
}
```

Changes to Class `quBaseState`

- `quBaseState` constructor allocates new `mp_ptermbuf` termlist to list of remote terms
- `reset()` changed to initialize quantum memory to $|0\rangle$ only on process zero, its owner. Other processes initially own no terms
- New `dump()` function to print out the state of quantum memory
- `opbegin()` takes new parameter defining the bit to be operated on, and sets new variable `mp_remote_proc` to remote process id
- New member function `mp_owner_proc()` to return the owner processor of a basis state (bitvec object)

New Version of opadd()

```
void quBaseState::opadd(const bitvec& v,const complx& z) {
  if (mp_owner_proc(v) == mp_my_proc) {
    ptermbuf->add(v,z);
  }
  else {
    mp_ptermbuf->add(v,z);
  }
}
```

New Version of opend()

Calls new function `mp_sendrecv` to exchange lists of remote terms

```
void quBaseState::opend() {
  termlist *pl;
  if (mp_remote_proc != mp_my_proc) {
    mp_sendrecv(mp_ptermbuf, mp_remote_proc,
                ptermbuf, mp_remote_proc);
  }
  pl=pterms; pterms=ptermbuf; ptermbuf=pl;
}
```

C++ and MPI

- C++ classes were sometimes a hinderance to efficient use of MPI
- E.g. an array of complex number objects cannot be portably sent or received, as the address of the data is hidden by the object
- Similarly, class `termlist` internally stores terms in a linear array, but the terms are objects
- In this first implementation, portability was achieved at the cost of copying the data out of the objects into a buffer
- It would be more efficient to re-implement `termlist` with a list not using objects, and a `mp_sendrecv` member function

Current Status - Operators Parallelized

- Zero qubit
  - opCphase
    (Cond. phase: $V(\phi)|\epsilon\rangle \rightarrow e^{i\phi}|\epsilon\rangle$ if $\epsilon = 111\ldots$ otherwise $\rightarrow |\epsilon\rangle$)
- Single qubit
  - opBit
  - opNot
- Multiple qubit
  - opCNot

Implementation of Grover's Algorithm

```
void grover(int n) {
  int l=int(log2(n)+1);                       // no. of qubits
  double m = ceil(pi/8*sqrt(pow(2,l)));  // no. of iterations
  quBaseState qs(l+1);
  quSubString q(l,0,qs);
  quBit f(l,qs);
  H(q);
  for (int i=1; i<=m; ++i) {
      query(q,f,n);             // calculate C(q)
      CPhase(pi,f);             // negate |n>
      query(q,f,n);             // undo C(q)
      diffuse(q);               // diffusion operator
```

Implementation of Grover's Algorithm

```
void diffuse(quState& q) {
  H(q);           // Hadamard Transform
  Not(q);         // Invert q
  CPhase(pi,q); // Rotate if q=1111..
  Not(q);         // undo inversion (Not is own inverse)
  H(q);           // undo Hadamard (H is own inverse)
}
```

Implementation of Grover's Algorithm

```
void query(quState& x, quBit& f, int n) {
  int i;
  for (i=0; i<=x.mapbits()-1; ++i)
    if (((n>>i) & 1) == 0) { // if i-th bit of n is zero
      quBit xi(i,x);
      Not(xi);                // then flip i-th qubit of x
    }
  CNot(f,x);                  // flip f if x=1111...
  for (i=0; i<=x.mapbits()-1; ++i) // restore x
    if (((n>>i) & 1) == 0) {
      quBit xi(i,x);
      Not(xi);
```

Performance Measurements

grover(16384) and grover(4096) - NB: no measurement yet

Qubits	No. Procs	Time (s)
15	1	132
15	2	69.3
15	4	39.3
15	8	25.9
13	1	16.3
13	2	11.2
13	4	8.8
13	8	17.2

Storage Requirements

- 64 bytes are needed per term for the sequential code
- So on a single processor with 2GB of memory, assuming 1GB ($= 2^{30}$) available, if all possible basis vectors had non-zero amplitudes, the maximum number of qubits that can be simulated is about 24
- With 16 nodes each with 2GB, the parallel version could represent up to four more qubits, i.e. 30, though there is an extra memory overhead so it's probably more like 3 more
- However, not all algorithms use such a large number of non-zero terms

Conclusions

- Hard to understand C++ code without documentation, even with access to the author!
  - Doxygen tool helped - class hierarchy and code browsing
- C++ classes can make efficient use of MPI harder than for C or Fortran
- A parallel version of Grover's algorithm has been implemented
- Performance is promising
- Much more to do!

Future Work

- Static control of distributed qubits by programmer
- Multi-qubit operators
- Shor's algorithm (need opSwap)
- Optimized parallel `termlist` class with `mp_sendrecv`
- Dynamic redistribution of qubits

Further Information

- http://www.stack.nl/~dimitri/doxygen/
- http://www.qubit.org/
- http://arXiv.org/archive/quant-ph
- http://tph.tuwien.ac.at/~oemer/qcl.html
- http://www.vcpc.univie.ac.at/~ian/