Parallelization of the General Single Qubit Gate and CNOT for the QC-lib Quantum Computer Simulator Library*

Ian Glendinning and Bernhard Ömer

VCPC
European Centre for Parallel Computing at Vienna
Liechtensteinstraße 22, A-1090 Vienna, Austria
{ian,oemer}@vcpc.univie.ac.at

Abstract. This paper reports on work to parallelize QC-lib, a C++ library for the simulation of quantum computers at an abstract functional level. After a brief introduction to quantum computing, we give an outline of QC-lib, then describe the parallelization using MPI of the general single qubit gate and the controlled-NOT gate (CNOT). Although QC-lib also implements other gates, this combination is already universal for quantum computing, and has allowed us to implement Grover's quantum search algorithm as a test case. We present some performance measurements made on a Beowulf cluster, which show reasonable speedups for Grover's algorithm.

1 Introduction

Quantum computers are devices that process information using physical phenomena unique to quantum mechanics, and which have the potential to be able to solve certain problems such as prime factorization spectacularly faster than any conventional computer [1]. In a classical computer the basic unit of information is the bit, a two-state device that can represent the values 0 and 1. The quantum analogue of the bit is a two-state quantum system, such as an electron's spin or a photon's polarization, which has come to be known as a *qubit*. The difference between a qubit and a bit is that a qubit can exist not only in the states 0 and 1, but also in a mixture of both of them, called a superposition state. Furthermore, whereas a register of n bits can be in any one of 2^n states, storing one of the numbers 0 to $2^n - 1$, a register of n qubits can be in a superposition of all 2^n states, and a function applied to a quantum register in a superposition state acts on all 2^n values at the same time! This is known as quantum parallelism, and it is one of the key ingredients in the power of quantum computers.

Unfortunately, when a quantum register in a superposition state is measured, the result obtained is only *one* of the 2^n possible values, at random. However all is

* This work was partially supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF.

not lost, as the probabilities of measuring the different values can be manipulated by operating on a quantum register with quantum gates, which are the quantum analogue of logic gates. Quantum algorithms consist of sequences of quantum gate operations and optionally measurements, and it turns out that algorithms exist that are able to exploit quantum parallelism, and to leave an output register in a state where the probability of obtaining the value that is the answer to the problem is very close to one, giving an advantage over classical algorithms.

However, building quantum computers is a huge technological challenge, and quantum computing hardware is not currently available outside physics research labs, so simulators present an attractive alternative for experimenting with quantum algorithms. Furthermore, at present the world's most powerful quantum computer only has seven qubits, which simulators can do better than. Simulators also help debugging of quantum programs, allowing direct examination of the quantum state, which is not possible in physical quantum computers. Simulators do suffer from a problem, which is that their execution time and memory requirements increase exponentially with the number of qubits. Parallelization alleviates this problem, allowing more qubits to be simulated in the same time or the same number to be simulated in less time.

2 Representation of Qubits, Registers and Gates

The state of a qubit can be represented by a two-dimensional complex vector of length 1. The states that are the quantum analogues of 0 and 1 are called the computational basis vectors, and they are written $|0\rangle$ and $|1\rangle$, in a notation due to Dirac. In terms of vectors, they are conventionally defined to be:

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

And a general qubit state is:

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

where α and β are complex numbers called *amplitudes*. Measurement of the state always gives either $|0\rangle$, with probability $|\alpha|^2$ or $|1\rangle$ with probability $|\beta|^2$, which is consistent with the *normalization* condition that the vector's length is 1, which is $|\alpha|^2 + |\beta|^2 = 1$.

The state of a two qubit register can be represented as a four-dimensional complex vector of length 1. The computational basis vectors are the quantum analogues of the binary values 00, 01, 10 and 11:

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

and a general two-qubit state is a linear combination (superposition) of them:

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

where the α_i are complex, and the normalization condition is:

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

The state of an n-qubit register can be represented as a 2^n-dimensional complex vector of length 1. 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 complex component of the vector representing the state, $|\alpha_i|^2$ is the probability that measurement will give the value i, and the normalization condition is:

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

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. 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 analogous 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.:

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

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

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

This gate is useful because applying it to either of the Basis states $|0\rangle$ and $|1\rangle$ produces an equal mixture of both of them:

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

and

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

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 ($|c,t\rangle$):

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

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

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

3 QC-Lib

QC-lib is a C++ library for the simulation of quantum computers at an abstract functional level [2], and it is used as the back end of an interpreter for the QCL Quantum Computation Language [3]. Its 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), and easy extendibility by class inheritance. 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

The data structure of class `termlist` is a linear array in combination with a hash table. Only terms with non-zero amplitudes are stored, and the array and hash table are dynamically doubled in size if the array fills up.

The class `quState` contains two sub-classes, `quBaseState` which contains actual state information, and `quSubState` which represents substates. An object of class `quBaseState` represents the state of the whole quantum memory, and the `quSubState` class can be used to allocate subregisters. A `quBaseState` object contains two `termlist` objects. One contains the terms in the current state, and the other is a term buffer to accumulate the result of an operation on the state.

A sub-class of `opOperator` called `opMatrix` implements the representation of an n-qubit operator as a $2^n \times 2^n$ complex matrix, 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 basis vector for another, with or without a phase factor, and `opOperator` has sub-classes for a number of such special cases, such as permutation of qubits, quantum functions, general single qubit operations, and the CNOT gate.

Figure 1 shows an example C++ Program using QC-lib, which implements the Hadamard transform, which is the Hadamard gate applied to every qubit in a register.

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

Fig. 1. The Hadamard transform implemented using QC-lib

4 Parallelization

The parallelization is being carried out in the message-passing style of programming, using the MPI message passing interface [4]. The parallelization strategy is to distribute the representation of the quantum memory. Each processor stores a subset of the terms, and operators are applied only to local terms. The result of a local operation in general includes terms that are not owned by the processor, which must be communicated to their owning processors. The program is SPMD, with each processor running a copy of the same code.

4.1 Data Distribution Scheme

The data distribution scheme is that on 2^n processors, the n least significant bits of a basis state form a distribution key, which determines which processor its corresponding term is stored on. For example, figure 2 shows the processor allocation of basis states for a four-qubit register on the four processors P_0–P_3.

P_0	$\|0000\rangle$, $\|0100\rangle$, $\|1000\rangle$, $\|1100\rangle$
P_1	$\|0001\rangle$, $\|0101\rangle$, $\|1001\rangle$, $\|1101\rangle$
P_2	$\|0010\rangle$, $\|0110\rangle$, $\|1010\rangle$, $\|1110\rangle$
P_3	$\|0011\rangle$, $\|0111\rangle$, $\|1011\rangle$, $\|1111\rangle$

Fig. 2. Data distribution for four qubits on four processors

4.2 Communication Pattern for Single Qubit Operators

Consider a general single qubit operator $U = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix}$ operating on a single qubit with state $\alpha|0\rangle + \beta|1\rangle$. For simplicity, assume that there are just two processors, and the qubit in question is the least significant one, and so determines the data distribution. The operation of U is as follows:

$$\begin{array}{l} P_0 \\ P_1 \end{array} \begin{array}{|c|} \hline \alpha|0\rangle \\ \hline \beta|1\rangle \\ \hline \end{array} \xrightarrow{U} \begin{array}{|c|} \hline \alpha U|0\rangle \\ \hline \beta U|1\rangle \\ \hline \end{array} = \begin{array}{|c|} \hline \alpha u_{11}|0\rangle + \alpha u_{21}|1\rangle \\ \hline \beta u_{12}|0\rangle + \beta u_{22}|1\rangle \\ \hline \end{array} \xrightarrow{\text{Comm}} \begin{array}{|c|} \hline (\alpha u_{11} + \beta u_{12})|0\rangle \\ \hline (\alpha u_{21} + \beta u_{22})|1\rangle \\ \hline \end{array}$$

After the operation of U locally on each processor, terms are created that are not owned by the processor, and so communication is necessary. Specifically, $\beta u_{12}|0\rangle$ has to be sent from $P_1 \to P_0$ and $\alpha u_{21}|1\rangle$ from $P_0 \to P_1$.

When a single qubit operator is applied to the ith qubit of a register, it is applied to the ith qubit of every term in the superposition state of the register, leaving the other qubits in each term unchanged. For example, applying U to the least significant qubit of a three qubit register that is initially in the state $\alpha|010\rangle + \beta|101\rangle$:

$$\begin{array}{l} P_0 \\ P_1 \end{array} \begin{array}{|c|} \hline \alpha|010\rangle \\ \hline \beta|101\rangle \\ \hline \end{array} \xrightarrow{U} \begin{array}{|c|} \hline \alpha U|010\rangle \\ \hline \beta U|101\rangle \\ \hline \end{array} = \begin{array}{|c|} \hline \alpha u_{11}|010\rangle + \alpha u_{21}|011\rangle \\ \hline \beta u_{12}|100\rangle + \beta u_{22}|101\rangle \\ \hline \end{array} \xrightarrow{\text{Comm}} \begin{array}{|c|} \hline \alpha u_{11}|010\rangle + \beta u_{12}|100\rangle \\ \hline \alpha u_{21}|011\rangle + \beta u_{22}|101\rangle \\ \hline \end{array}$$

For each term in the initial state, at most two terms are therefore created (if one or more of the u_{ij} are zero, less terms will be created). If the ith qubit is not one of the qubits that determines the data distribution, then no communication is necessary, as both new terms are owned by the original processor. Otherwise, one of the new terms is locally owned and the other one is remotely owned and must be communicated. For each processor, all the remotely owned terms are owned by the same other processor, as a single bit has been flipped in the distribution key. Symmetrically, the remote processor potentially creates terms that are owned by the local one. In general therefore, bi-directional communication is needed between disjoint pairs of processors.

If the basis vectors of an n-qubit register are thought of as the coordinates of the corners of an n-dimensional hypercube, such that the ith qubit represents a coordinate of 0 or 1 in the ith dimension of the hypercube, then the communication pattern generated by an operation on the ith qubit can be visualized as being along parallel edges of the hypercube with data movement in its ith dimension.

4.3 Parallelization of Single Qubit Operators

The representation of the distributed quantum memory has been encapsulated in class `quBaseState`, without altering the operator invocation mechanism in the sequential version of QC-lib. In the parallel version of the code, each object of class `quBaseState` has a second term buffer (a `termlist` object), to accumulate terms destined for remote processors. The local and remote terms that result from the action of single qubit operators are accumulated separately, and when the accumulation is complete, the remote term buffer is sent to the remote process, and reciprocally, the buffer that the remote process sent is received. This exchange of data between pairs of processes is implemented using the MPI function `MPI_Sendrecv()`. The received terms are then merged into the local term buffer, accumulating amplitudes if there is more than one term with the same basis state. Finally, the term buffer is swapped with the termlist for the current state, so that the new terms become the current state, and the old one is ready to be used to accumulate the result of the next operation.

4.4 Current Status

Currently only a few of the operator sub-classes of `opOperator` have been parallelized, but they include the general single qubit operator and the CNOT gate, which together are universal for quantum computation, though they don't necessarily offer the most efficient way of achieving a particular operation. Although the CNOT gate is a two qubit gate, it only modifies a single qubit, so we were able to use the same parallelization strategy as for single qubit gates.

The current version of the program was implemented using the MPI-1.1 language bindings, which do not include support for C++ datatypes, specifically the class `complex<double>`, which is used by QC-lib to represent complex amplitudes. In order to guarantee portability, the real and imaginary parts of these amplitudes were copied into a buffer before being communicated. The next version of the program will eliminate this overhead by using the MPI-2 language bindings, which support this datatype.

5 Performance Measurements

Performance measurements were made on a Beowulf cluster with 16 compute nodes, each having a 3,06 GHz Pentium 4 processor and 2 GByte of 266 MHz dual channel DDR-RAM FSB, running Linux 2.4.19. The nodes were connected by two independent networks, Gigabit Ethernet for message-passing communication, and Fast Ethernet for job submission, file access, and system administration tasks.

Figure 3 shows how the run time for the Hadamard transform varies with the problem size (number of qubits) for different numbers of processors.[1] The run time for the sequential code is also shown, and it can be clearly seen from the semi-logarithmic plot that it increases by a constant factor for every extra qubit. Each extra qubit corresponds to a doubling of the problem size, which is consistent with the slope of the line, which corresponds to an increase in run time by a factor of 2.05 for each extra qubit. The parallel version of the program run on one processor has very little overhead compared with the sequential version, which is not surprising, as it does not perform any communication. When larger numbers of processors are used for smaller problem sizes there is more of an overhead, but as the problem size increases, the scaling behaviour of the parallel code becomes similar to that of the sequential version.

Figure 4 shows how the speedup varies with the number of processors, for various problem sizes. The speedup is relative to the run time of the sequential code for the same problem size, but since the sequential code ran out of memory for the larger problem sizes, the sequential run time for those cases had to be estimated, which was done by by linearly extrapolating a least squares fit to the the logarithm of the run time versus the number of qubits. For small problem sizes there is actually a decrease in speedup for larger numbers of processors, which is to be expected for any problem size if enough processors are used. For larger problem sizes reasonable speedups were obtained, up to 9.4 on 16 processors for the largest problem size.

The Hadamard transform is a sufficiently simple operation, that it was possible to make make runs that used all the system's available memory within a reasonable run time. It was found that both the sequential version of the program and the parallel version on one processor could simulate a maximum of 25 qubits, and that each doubling of the number of processors increased the

[1] Because by coincidence the amount of memory needed for the largest problem sizes was almost exactly the same as the amount available, if a significant amount of memory was already in use by the system when the run started, the program was sometimes forced to use swap memory, increasing the run times by a factor of about five. Even with exclusive use of the nodes, it was found that as much as 600 MByte could sometimes be in use by the system. This phenomenon is being investigated.

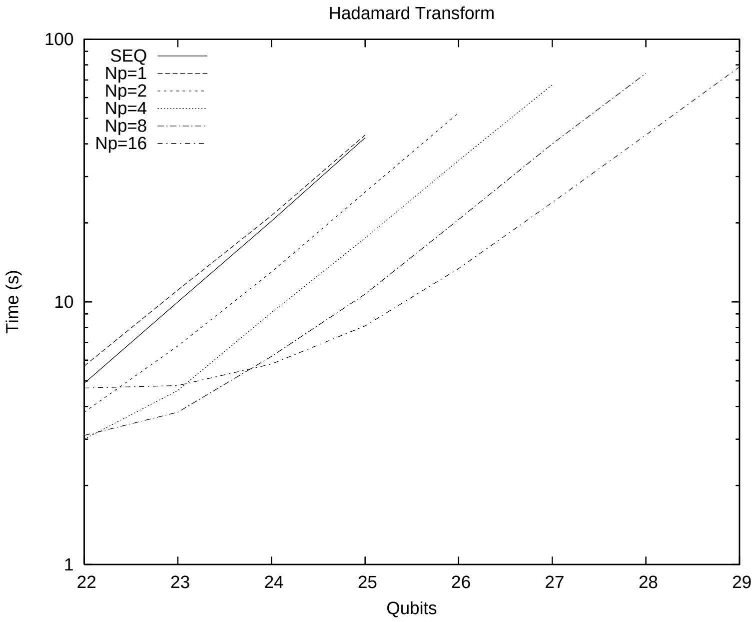

Fig. 3. Run time for Hadamard transform

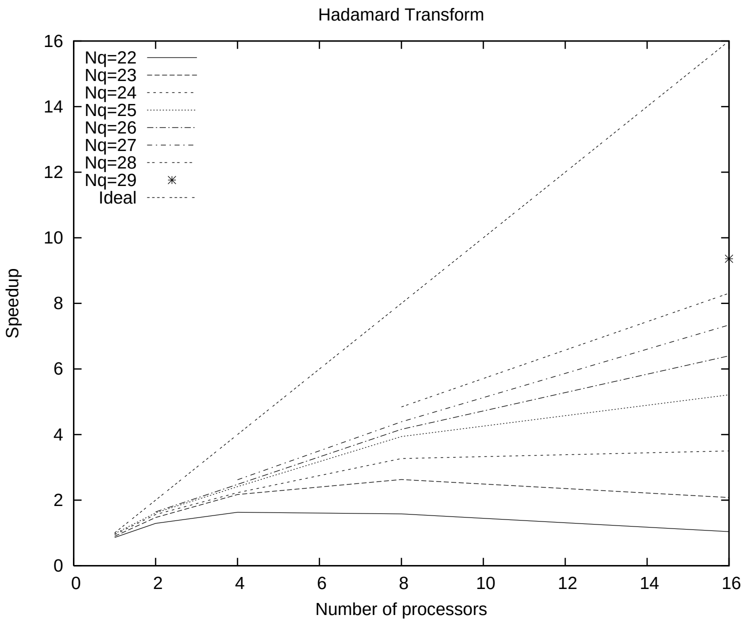

Fig. 4. Speedup for Hadamard transform

maximum number of qubits by one, up to a maximum of 29 qubits for all 16 processors. This is consistent with the amount of memory needed by the program to store quantum states, which in the sequential version of the code, for quantum states with no more than 2^{32} terms, is 64 bytes per term. The parallel version of the code contains an extra `termlist` to buffer terms that need to be communicated, which means that it potentially needs up to 50% more memory, but in the Hadamard transform very few terms need to be communicated, as only changes to the least significant qubits cause communication, and these are processed first, but large numbers of terms are not created until the most significant qubits are processed. Operating on n qubits, the Hadamard transform ultimately produces a state with 2^n terms, so the amount of memory needed to represent it is approximately $64(2^n) = 2^{n+6}$ bytes, and 25 qubits require 2 GByte, which was the amount of memory available. Not all algorithms use such a large number of terms with non-zero amplitudes, and as only these are explicitly represented by QC-lib, some algorithms can be run for still larger numbers of qubits.

Grover's quantum search algorithm [5] was also implemented using QC-lib (Fig. 5), and the run time was measured for the portion of the algorithm up to the point when the first measurement is made. This was done to allow meaningful comparison of run times, as in the full version of the algorithm, there is a finite probability that the search will fail, and that it has to be repeated, so the run time is non-deterministic. Figure 6 shows how the run time for Grover's algorithm varies with the problem size, for different numbers of processors. The behaviour is similar to that in the case of the Hadamard transform, but in this case the slope of the line for the sequential run time corresponds to an increase in run time by a factor of 3.10 for each extra qubit. This is partly accounted for by the fact that the number of iterations in the main loop of the algorithm increases by a factor of $\sqrt{2}$ for each extra qubit, and taken together with the doubling of the problem size, that would imply an increase in run time by a factor of 2.82.

Figure 7 shows how the speedup varies with the number of processors, for various problem sizes. Significantly better speedups are obtained than for the Hadamard transformation. For just 17 qubits, which is less than the minimum number of qubits considered for the Hadamard transform, a speedup of 11.2 was obtained on 16 processors, and it is to be expected that with more qubits the performance would be even better.

6 Related Work

There are many quantum computer simulators [6], but we are only aware of two of them that have been implemented for parallel systems. Obenland and Despain's simulator [7] runs on distributed-memory parallel computers, like our simulator, but it is based on a physical model of a particular experimental realization of a quantum computer. Like our simulator, the one by Niwa et al. [8] does not depend on any physical model, but it runs on shared-memory multicomputers rather than distributed-memory ones.

```
void grover(int n) {
  int l=int(log2(n+0.5)+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
  }
}

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)
}

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);
    }
}
```

Fig. 5. Grover's algorithm implemented using QC-lib

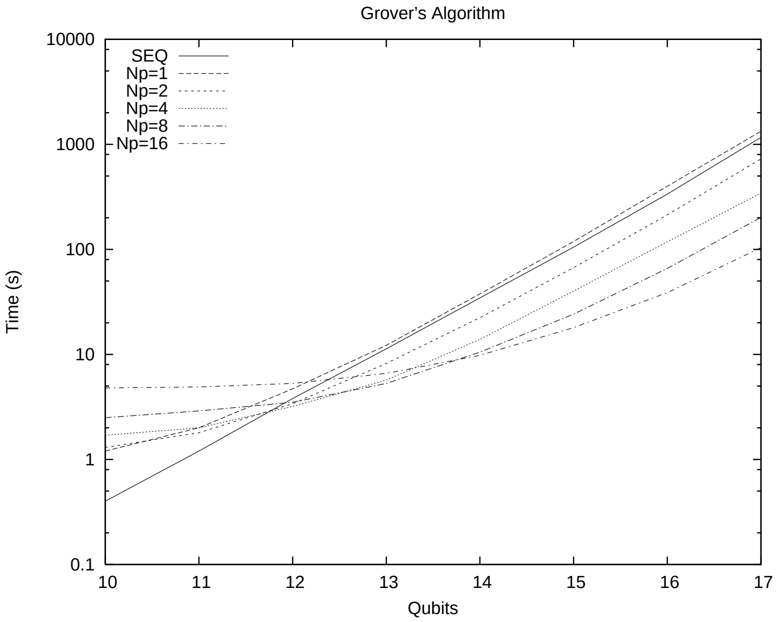

Fig. 6. Run time for Grover's algorithm

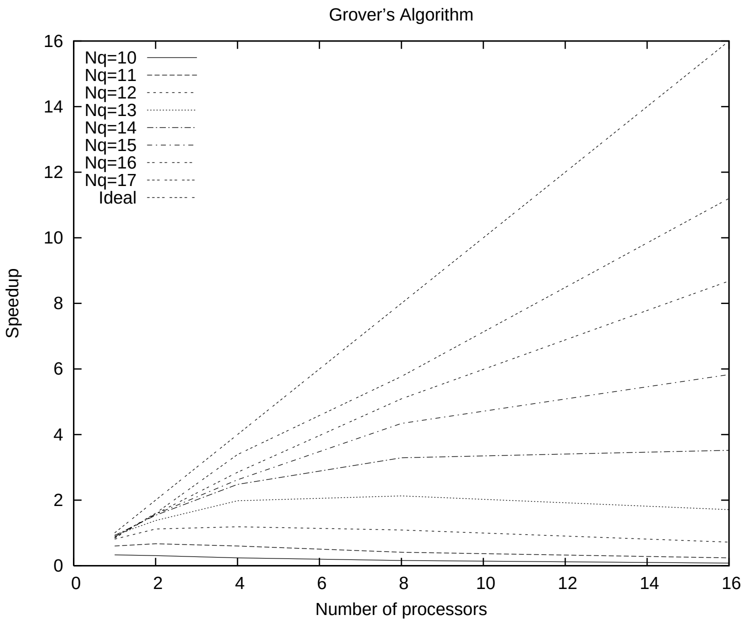

Fig. 7. Speedup for Grover's algorithm

7 Conclusion

Sufficient functionality of QC-lib has been implemented to simulate universal quantum computation. We have implemented the Hadamard transform and Grover's algorithm using our library, and have made performance measurements for these codes. Promising speedups were obtained.

Future work will include re-coding to use the MPI-2 C++ binding for the complex datatype, implementing static control of the distribution of qubits by the programmer, implementing more operators and Shor's prime factorization algorithm [9], and investigating its performance. In the longer term, dynamic redistribution of qubits will be implemented and a load-balancing strategy will be developed.

References

1. M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.
2. B. Ömer. Simulation of quantum computers, October 1996. http://tph.tuwien.ac.at/~oemer/doc/qcsim.ps.
3. B. Ömer. Quantum Programming in QCL. Master's thesis, Vienna University of Technology, 2000. http://tph.tuwien.ac.at/~oemer/doc/quprog/index.html.
4. Message Passing Interface Forum. MPI: A message-passing interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org/.
5. L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. of the 28th annual ACM Symposium on the Theory of Computation (Philadelphia, Pennsylvania)*, pages 212–219, New York, 1996. ACM Press.
6. J. Wallace. Quantum computer simulators, 2001. http://www.dcs.ex.ac.uk/~jwallace/simtable.htm.
7. K. M. Obenland and A. M. Despain. A parallel quantum computer simulator, April 1998. http://arXiv.org/abs/quant-ph/9804039.
8. H. Imai J. Niwa, K. Matsumoto. General-purpose parallel simulator for quantum computing, January 2002. http://arXiv.org/abs/quant-ph/0201042.
9. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comp.*, 26(5):1484–1509, 1997.