Efficient Quantum Computing Simulation in GRID Environments

V. Pavlov

Key Words: *Quantum Computing Simulation; Shor's Algorithm*

Abstract. *We describe an implementation of Shor's quantum algorithm for prime number factorization on a portable universal quantum computing simulator that allows large-scale entirely quantum computation to be carried out in parallel environments. The quantum computing simulator in question is a result of our previous work in which we speculated that its special design will not only allow running efficient simulations of various quantum algorithms for large problem sizes, but will also allow reducing the simulation time further by parallelizing its execution among multiple computing elements. We test these expectations by implementing and running Shor's algorithm for different problem sizes — **up to 59 qubits** — in single - and multi-processing environments, consisting of mainstream computing elements participating in a GRID infrastructure. The results indicate ideal scaling of the simulator as a function of the number of the computing elements involved and thus show that it can be used standalone or in GRID environment as a valuable research tool for performing large scale quantum algorithm simulations.*

1. Introduction

This paper is a follow up of a previous work [1], in which the design of a powerful and portable universal quantum computing simulator has been presented. The simulator is specially designed so that the major factor determining the simulation time be the „amount" of superposition and entanglement of the state of the quantum register. Instead of blindly representing the n-qubit register as a 2^n x1 matrix of complex numbers, the software tries to keep the qubits that comprise the quantum register in independent sub-registers and only combine them in case entanglement between qubits of different sub-registers is necessary. This allows the simulator to handle large quantum registers. In the present paper, the realization of an entirely quantum simulation of Peter Shor's factorization algorithm [2] is given. The simulation is then run several times for input of different sizes. In the last run the size of the quantum register was 59 qubits, in contrast to the state of the art systems in which the absolute upper limit for the size of the register is set to about 36 qubits. This last limit is imposed by the exponential growth of resource requirements, which amount to 1TB for 36 qubits in a straightforward 2^nx1 matrix representation. If our simulator worked with such representation, over 8 million TB (!) would be necessary to represent the 59 qubits in question. Moreover, each elementary transformation would require the multiplication of the 2^n x2^n matrix of the transformation by the 2^n x1 column-vector of the quantum register, and there are possibly thousands and millions elementary transformations in every non-trivial quantum algorithm. Still, due to its design, the simulator was able to finish the 59-qubit quantum computation in less than 1 day using the processing power of 8 ordinary mainstream workstations.

This paper is organized as follows:

- Section 2 describes briefly the quantum simulator and the Application Programming Interface API it exposes.
- Section 3 describes Shor's algorithm.
- Section 4-7 describes our implementation of the quantum transformations that are the building blocks of Shor's algorithm.
- Section 8 shows the results of the simulation.
- Section 9 discusses means for parallelizing the execution and the results obtained via parallel runs.
- Section 10 concludes the paper.

2. About the Simulator

The quantum simulator described in [1] is realized in Common Lisp and more specifically using the GNU CLISP Implementation (http://clisp.cons.org). One of the reasons for preferring Lisp over other alternatives (e.g. C) is that it offers great extensibility, including the ability to construct program code in run-time (through program-writing macros) and its natural ability to implement embedded languages easily. The former is very useful for designing quantum algorithms, since in most of them the structure of the program to be executed depends on some of the input values (for example, the structure of Shor's algorithm depends upon two of its inputs). On the other hand, due to the second ability, instead of designing a language in which to express quantum algorithms and then implementing a parser/compiler for it, we let Lisp manage this. Both classical and quantum code are thus written in the same language, Lisp.

In contrast with GUI-based quantum algorithm construction sets, which are quite limited, our approach enables the quantum algorithm developers to use the full power and flexibility of one of the most extensible programming languages.

The core of the quantum simulator API consists of the calls which carry out the 4 elementary transformations from the chosen universal set of gates. Following [3], this set includes three 1-qubit and one 2-qubit transformations, namely:

$$\mathrm{ph}(\delta)=\begin{pmatrix} e^{i\delta} & 0 \\ 0 & e^{i\delta} \end{pmatrix} \quad \mathrm{rz}(\alpha)=\begin{pmatrix} e^{i\frac{\alpha}{2}} & 0 \\ 0 & e^{-i\frac{\alpha}{2}} \end{pmatrix}$$

$$\mathrm{ry}(\theta)=\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & \sin\left(\frac{\theta}{2}\right) \\ -\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

$$\mathrm{cnot}=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (1)$$

with the corresponding API calls:

— **(ph x δ)** phase-shifts the amplitudes of qubit X through angle δ;

— **(rz x α)** — rotates the probability vector of qubit X around z axis through angle α/2;

— **(ry x θ)** — rotates the probability vector of qubit X around y through angle θ/2;

— **(c/c x)** controlled NOT with control qubit C and target qubit X.

The algorithms for applying these transformations are central for achieving the main property of the simulator — its efficiency and ability to handle operations with large registers. Their design is described in detail in [1].

As Barenco et al. show in [3], using these 4 elementary gates, it is possible to construct *any* unitary transformation. Using their approach, we have included several widely used gates in the core simulator API:

— **(h x)** - Hadamard transformation, which brings qubit X from a base state to a state of uniform superposition;

— **(q-not x)** - a trivial NOT transformation, which unconditionally negates qubit X;

— **(cc-not c1 c2 x)** - controlled-controlled-NOT, this is *the* universal gate, which in terms of classical computation corresponds to the ubiquitous NAND gate. It performs a NOT operation upon the target qubit X if both controlling qubits C1 and C2 include a non-zero probability for the state $|1\rangle$.

The matrices corresponding to these transformations are:

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

$$not = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

(2)

$$ccnot = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Figure 1 shows the implementation of the corresponding Lisp functions in terms of the elementary transformations.

```
(defun h (x)
  (ph x pi/2)
  (rz x -pi)
  (ry x pi/2))

(defun q-not (x)
  (ph x pi/2)
  (ry x pi)
  (rz x pi))

(defun srn (x)
  (ph x pi/4)
  (rz x -pi/2)
  (ry x pi/2)
  (rz x pi/2))

(defun srn^-1 (x)
  (ph x -pi/4)
  (rz x pi/2)
  (ry x pi/2)
  (rz x -pi/2))

(defun c-srn (c x)
  (rz c -pi/4)
  (ph c pi/8)
  (rz x -pi/2)
  (ry x pi/4)
  (c/ c x)
  (ry x -pi/4)
  (c/ c x)
  (rz x pi/2))

(defun c-srn^-1 (C X)
  (rz c pi/4)
  (ph c -pi/8)
  (rz x pi/2)
  (ry x pi/4)
  (c/ c x)
  (ry x -pi/4)
  (c/ c x)
  (rz x -pi/2))

(defun cc-not (c1 c2 x)
  (c-srn c2 x)
  (c/ c1 c2)
  (c-srn^-1 c2 x)
  (c/ c1 c2)
  (c-srn c1 x))
```

Figure 1. Implementation of h, q-not and cc-not in terms of elementary transformations. The implementation is created following the rules for arbitrary gate construction given in [3]

Apart from the transformations, the quantum simulator API also includes functions for creation, manipulation, measurement and optimization of the quantum register:

— ***qax*** — a global variable, quantum register, the defaulttarget for all transformations. Basically, this serves only as syntactic sugar — if no register is specified for a transformation, it will act upon the register in ***qax***;

— **(make-qreg size)** — creates a quantum register consisting of SIZE qubits. Note that due to the design of the simulator, SIZE can be arbitrary large — until entanglement occurs, the qubits behave like ordinary bits and memory requirements grow linearly with SIZE;

— **(measure qr)** — performs a measurement upon the quantum register QR. As a result it will return one of the base states that participate in QR's superposition, according to the probabilities encoded in the superposition's complex amplitudes;

— **(minimize qr)** — tries to optimize the register representation, dis-entangling qubits if possible. This function can be used to lower the complexity of the state, if possible, in order to minimize memory requirements and reduce the computational times of transformations to come.

— **(bstore/bload qr)** — pair of functions for storing and loading the current state of the quantum register QR in a file;

— **(qreg-traverse qr)** — prints out all the states that participate in the quantum register QR's superposition, along with the probability for reading them out when a measurement is performed.

This concludes the API exposed by the quantum simulator. The code spans about 600 lines of Lisp and allows the execution of an arbitrary quantum algorithm. Using the API, we implement Shor's factoring algorithm [2] in order to test the correctness and efficiency of the simulator.

3. Shor's Algorithm

Shor's algorithm is concerned with decomposing a given integer N into a product of primes. As Shor argues in [2], the fastest known classical algorithm for this task is of exponential complexity, while his quantum algorithm will only take $O((\log N)^2 (\log\log N) (\log\log\log N))$ steps on a quantum computer, along with some polynomial in $\log(N)$ classical post-processing.

Essentially, the algorithm finds the order of a randomly chosen element x in the multiplicative group mod (N) ; that is, the least integer r such that $x^r \equiv 1 \pmod N$. The input to the algorithm is the number N to be factored and $x < N$, a randomly chosen co-prime to N whose period r is to be found. The algorithm uses two registers and proceeds as follows:

1. Find m that satisfies:

(3) $$N^2 \le 2^m < 2N^2$$

The size of the first register is then set to m qubits and the size of the second is set to $l = m/2$ qubits.

2. Both registers are initialized to 0. The state $|\varphi\rangle$ of the quantum system is thus given by:

(4) $$|\varphi\rangle = |0\rangle|0\rangle.$$

3. Prepare the first register in uniform superposition of states representing all integers $a(\mathrm{mod}\, 2^m)$. The state of the system becomes:

$$|\varphi\rangle = \frac{1}{\sqrt{2^m}} \sum_{a=0}^{2^m-1} |a\rangle|0\rangle. \quad (5)$$

4. Compute $x^a(\mathrm{mod}\ N)$ in the second register:

$$|\varphi\rangle = \frac{1}{\sqrt{q}} \sum_{a=0}^{2^m-1} |a\rangle|x^a(\mathrm{mod}\, N)\rangle. \quad (6)$$

5. Perform Quantum Fourier Transform (QFT) on the first register:

$$|\varphi\rangle = \frac{1}{q} \sum_{a=0}^{q-1} \sum_{c=0}^{q-1} e^{i2\pi\frac{ac}{q}} |c\rangle|x^a(\mathrm{mod}\, N)\rangle. \quad (7)$$

6. Observe the quantum system, leaving it in a particular state $|c\rangle|x^k(\mathrm{mod}\, N)\rangle$. From this point forward the algorithm involves classical post-processing and possibly restarts from the beginning, so these are of no particular interest in terms of testing the simulator correctness and performance (see [2] for full treatment of Shor's algorithm and its reasoning).

All in all, the quantum simulator needs to: a) prepare the system in a particular suitable state; b) perform calculation of exponent modulo *N*; c) perform QFT and d) measure the outcome. The Appendix to this article includes the complete source code listing of the implemented quantum algorithm. The following sections provide details about the implementation.

4. Map of the Quantum Register

Before we proceed any further, a note on the size of the quantum register required to execute the algorithm and a map giving the purpose of its qubits must be made. The size of the register depends on the exact algorithm chosen for implementing the quantum computation in step 4, since it inevitably requires some scratch space. We have chosen an algorithm based on the treatment of elementary arithmetic operations presented by Vedral et al in [4] and in accordance with it we need $7l+3$ qubits, where l is the number of bits in the binary representation of *N*. For example, in order to execute Shor's algorithm for an 8-bit *N*, 59 qubits are needed. *Figure 2.* depicts the structure of the quantum register for a 4-bit *N*.

The simulator uses a notation in which the qubits in the register are numbered starting from 0 from left to right, most significant bits first (left). Having in mind this ordering, the purpose of the qubits is as follows:

– the leftmost $m = 2l$ qubits ($A_0 - A_{m-1}$) correspond to the first register in Shor's algorithm;

– the following $l+1$ qubits (C_R, $R_0 - R_{l-1}$) correspond to the second register in Shor's algorithm, along with an additional qubit (C_R) used throughout the execution of the adder subroutine to contain the most significant carry bit;

– the following $l+1$ qubits (C_Y, $Y_0 - Y_{l-1}$) are temporary scratch space used as a second operand in the **expt-mod** subroutine;

– the following l qubits ($C_0 - C_{l-1}$) are temporary scratch space used to store the carry bits throughout the execution of the **adder** subroutine;

– the following l qubits ($M_0 - M_{l-1}$) are temporary scratch space used as a second operand during in the **ctrl-mult-mod** subroutine;

– the following $l+1$ qubits (T, $N_0 - N_{l-1}$) are temporary scratch space used as a second operand in the **adder-mod** subroutine, along with a temporary qubit (T), used in the same subroutine.

The following sections explain in detail the subroutines involved in the exponent modulo N calculation. For full treatment of the algorithms involved see [4].

5. Main Routine

The main algorithm routine is found in the function **shor**, which receives *x* and *N* as arguments. Step 1 of Shor's algorithm is a trivial classical computation concerned with computing the number of bits *l* in the binary representation of *N* (carried out by **bitsize**) and doubling it in order to find *m*.

The simulator's API call **make-qreg** automatically puts the system in the trivial state given in (4), which takes care of step 2. As explained in Section 4 above, the size of the register is set to $7l+3$ qubits.

Step 3 is another trivial computation, this time quantum, and all it involves is a sequence of *m* Hadamard transformations h, each of them bringing the corresponding qubit from state $|0\rangle$ to state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. The collective result of performing these on the m qubits of the first register is exactly the state given in (5).

Step 4, the exponent modulo N, is computed in a separate sub-routine, **expt-mod**, which is described in detail in Section 6.

Step 5 is computed in a separate sub-routine, **qft**, which is described in detail in Section 7.

Instead of getting a single particular result $|c\rangle|x^k(\mathrm{mod}\, N)\rangle$ from the measurement, as in a real quantum computer, in Step 6 we are more interested in looking at all possible results for some x^k (mod *N*), along with their probabilities. This is what **shor-result** does, taking for definitiveness x^k (mod *N*) to be 1. Note how **bit-reverse** is used

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	C_R	R_0	R_1	R_2	R_3	C_Y	Y_0	Y_1	Y_2	Y_3	C_0	C_1	C_2	C_3	M_0	M_1	M_2	M_3	T	N_0	N_1	N_2	N_3
REG #1									REG #2					a^x%N				+				c-a^x%N					a+b%N			

Figure 2. Map of the quantum register for running the algorithm with a 4-bit *N*

upon the first register, since the QFT as implemented (and as prescribed by Shor in [2]) produces its result in reversed-bit binary notation.

6. Exponent Modulo N

Step 4 of the algorithm is implemented using the quantum algorithms for arithmetic operations given in [4]. These are fairly complex algorithms build bottom-up from simpler operations. Our implementation strictly follows the original prescriptions in the mentioned paper.

There is a fundamental complication in all quantum algorithms that need scratch space for temporary storage of intermediary data. Since quantum computation is reversible in nature, erasing (which is in essence an irreversible operation) is not allowed. This means that in order to be able to reuse the scratch space in further invocations, it has to be first brought back to its original state without erasing. As Bennett has shown in [5], this can be done by running the algorithms that changed it in first place, backwards. In terms of our simulation, this means we have to implement a „backward" version of the corresponding algorithms. A convention is adopted, in which the „backward" version of algorithm carried out by a function **x** is to be found in a function named **/x**.

This said, the most primitive operations involved are the **carry** and **sum** operations, along with their „backward" versions, **/carry** and **/sum**. Like their classical counterparts they are used to build a full adder block (the **adder** sub-routine) in very much a traditional way. The only complication arises with the need to reset the scratch space used to contain the intermediary carry calculations (the $C_0 - C_{l-1}$ block in the map, see *figure 2*) back to the initial state of 0.

Using **adder** and its counterpart **/adder**, the more complex **adder-mod** algorithm is implemented. One of the features in the adder-mod algorithm is that at certain stages it needs to perform a summation (subtraction) in which the value N participates. This can only be done by the adder algorithm if N is loaded in a quantum register. This requires the execution of certain transformations that bring a 0-initialized temporary sub-register (the $N_0 - N_{l-1}$ block in the map, see *figure 2*) to a state in which it contains N. A gate (at certain stages **q-not**, at other **c-not**) needs to be present at the corresponding places in the **adder-mod** algorithm for each 1 that appears in the binary representation of N. This is one example in which the structure of the quantum gate sequence depends on its input arguments and it is here that Lisp's ability to have the program dynamically construct its code during the execution proves very helpful. The **0->N** and **c-0->N** functions with the help of the **map-1s** macro emit the proper set of **q-not** gates (correspondingly **c-not** gates) which is then executed using Lisp's **apply** dynamic execution mechanism. **adder-mod** also uses one additional temporary qubit, T, as described in [4].

The next set of sub-routines, **ctrl-mult-mod** and **/ctrl-mult-mod**, are constructed in similar fashion. As with **adder-mod**, at certain stages they need to perform a quantum operation involving one of its arguments, this time however conditionally (either 0 or $y*2^i$ is used as an operand, controlled by certain qubits). The **cc->0->N** function is used to emit the set of gates that will bring the 0-initialized temporary sub-register denoted by $M_0 - M_{l-1}$ to the required state. The sole difference between the **0->N**, **c-0->N** and **cc-0->N** functions is that they emit different gates, but they do so for each 1 in the binary representation of its argument. That is why the same macro, **map-1s**, is used in all of them.

Finally, **expt-mod** is constructed using **ctrl-mult-mod** and **/ctrl-mult-mod,** as shown in [4]. Note how x, which is an argument to the function, becomes effectively built into the structure of the quantum algorithm through computing $x^{2^i} \pmod M$ and its inverse[1] at each step and passing it as argument to **ctrl-mult-mod**, which in turn uses the aforementioned **cc-0->N** to emit the needed quantum gates in run-time. The other classical argument — m (computed in Step 1) is also effectively built into the structure of the algorithm through the various loop constructs found throughout the sub-routines. This is entirely in correspondence with Shor's insight that[2] q, x and N need not even be stored in the quantum register; instead they are to be built into the structure of the network — their values determine the sequence of quantum operations to be performed.

7. Quantum Fourier Transform

The Quantum Fourier Transform (QFT) implementation is found in the **qft** sub-routine and follows the design given by Shor in Section 4 of [2]. It uses two types of quantum gates:

— a 1-qubit gate R_j which acts upon qubit j;

— and the 2-qubit gate $S_{k,j}$ which acts upon qubits k and j.

The transformation matrices of the two gates are as follows:

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

(8)

$$S_{k,j} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\frac{\pi}{2^{k-j}}} \end{pmatrix}.$$

The gate R_j is just the Hadamard gate, while $S_{k,j}$ is a controlled gate, which changes the phase of qubit j's |1> amplitude in case qubit k's superposition includes |1>. The amount of phase shift depends on the difference between the indices *k* and *j*. Shor then shows that the sequence

(9)

$$R_{l-1}S_{l-2,l-1}R_{l-2}S_{l-3,l-1}S_{l-3,l-2}R_{l-3}\cdots$$
$$\cdots R_1 S_{0,l-1}S_{0,l-2}\cdots S_{0,2}S_{0,1}R_0$$

[1] The multiplicative inverse of x^{2^i} modulo N is computed by the sub-routine **inv**, which implements the classical extended Euclid algorithm for finding the greatest common divisor of two integers.

[2] In Shor's paper, *q* is used instead of *m*, where $q = 2^m$

effectively implements the Fourier Transform taking the state $|a_{l-1}a_{l-2}...a_0\rangle$ to

$$\frac{1}{\sqrt{q}}\sum_{i=0}^{q-1} e^{i2\pi\frac{ac}{q}}|c\rangle, q=2^l. \quad (10)$$

The R and S gates are implemented by the sub-routines **qft-R** and **qft-S** correspondingly. The only difference with Shor's treatment comes from the qubit labeling – in Shor's article 0 denotes the least significant (rightmost) qubit, while in our simulator 0 labels the most significant (leftmost) qubit. This has two implications: a) the sequence (9) is changed accordingly:

$$R_0S_{1,0}R_1S_{2,0}S_{2,1}R_2... \quad (11)$$
$$...R_{l-2}S_{l-1,0}S_{l-1,1}...S_{l-1,l-3}S_{l-1,l-2}R_{l-1}$$

and b) since now $j>k$ inside S, a minus sign has to be incorporated in the phase shifter exponent in order to preserve the angle as it appears in (8). These two implications are visible in the implementation of **qft** and **qft-S**.

$S_{k,j}$ is implemented out of elementary quantum gates following the gate construction recipes found in [3], see the **c-phasor** subroutine.

8. Results

The described algorithm was run several times for differently sized *N*. The results prove the correctness of the simulator (and the algorithm implementation) and provide a measure for the simulator performance.

As an illustration regarding the correctness of the simulation (and the algorithm), *figure 4* shows the probabilities measured in Step 6 for a run with input parameters *x=7* and *N=55*. The size *m* of the first register is 12 qubits. This means that after QFT is performed in Step 5, its value *c* can be any number from 0 to 2^{12}-1. Certain values however have much larger probabilities than the rest and these values are located around multiples of ~ 205, which gives a correct answer for the wanted period $2^{12}/205 \approx 20$. See Shor [2] for detailed interpretation of the results and compare *figure 4* with his *figure 5.1*.

Regarding the performance of the simulator, *table 1* shows the execution times for running the algorithm with differently sized *N*.

9. Parallel Execution

A quantum mechanical system is by definition a linear system (see for example [6] for a formal description of Quantum Mechanics). The state of the system can be represented as a vector in a complex vector space of suitable dimensions and its evolution until an observation is made is given by a sequence of unitary transformations. The implications are that the superposition principle holds for quantum computations.

Table 1. Execution times for tasks with increasing number of qubits in the quantum register

x^a mod N	# of qubits	time, s
7^a mod 15	31	301.4688
5^a mod 21	38	1 705.0469
7^a mod 55	45	13 905.8910
13^a mod 119	52	62 684.9840

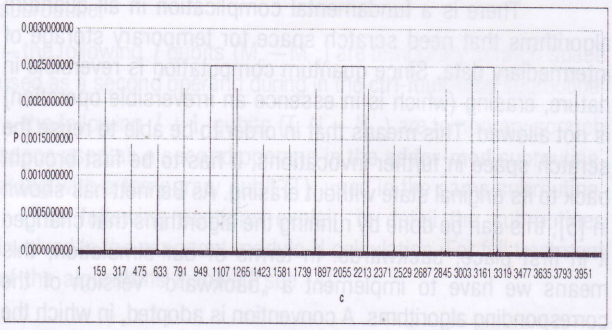


Figure 4. Probability **P** for observing the value *c* in the first register after Step 5 for x=7, N=55

The superposition principle states that the net result caused by two or more independent events is the sum of the results which would have been caused by each event individually.

We can directly use the superposition principle to parallelize the quantum computing simulation for any given task. Consider a computation in which the quantum register has to be prepared in a state of superposition given by:

$$|\Phi_{in}\rangle = \varphi_{00...0}|00...0\rangle + \varphi_{00...1}|00...1\rangle + ... \quad (11)$$
$$+\varphi_{11...1}|11...1\rangle.$$

An unitary transformation *U* is then to be applied, which would bring the register to the state:

$$|\Phi_{out}\rangle = U|\Phi_{in}\rangle. \quad (12)$$

The superposition principle states that:

$$|\Phi_{out}\rangle = U\varphi_{00...0}|00...0\rangle + \quad (13)$$
$$U\varphi_{00...1}|00...1\rangle +$$
$$+... +$$
$$U\varphi_{11...1}|11...1\rangle.$$

Each row in (13) is an application of *U* upon a certain base state and all these applications can be executed in parallel.

Moreover, the superposition principle allows us to group the inputs to the different jobs in whatever fashion we like

as long as they collectively form the initial state. For example, (13) can be rewritten as:

$$
\begin{aligned}
&|\Phi_{out}\rangle = U|\Phi_0\rangle + U|\Phi_1\rangle \\
&|\Phi_0\rangle = \varphi_{00\ldots0}|00\ldots0\rangle + \varphi_{00\ldots1}|00\ldots1\rangle + \ldots \\
&\qquad + \varphi_{01\ldots1}|01\ldots1\rangle \\
&|\Phi_1\rangle = \varphi_{10\ldots0}|10\ldots0\rangle + \varphi_{10\ldots1}|10\ldots1\rangle + \ldots \\
&\qquad + \varphi_{11\ldots1}|11\ldots1\rangle
\end{aligned} \tag{14}
$$

In (14), $|\Phi_0\rangle$ is a superposition of all states for which the most significant bit of the binary representation of their index is 0, and $|\Phi_1\rangle$ is a superposition of all states for which this same bit is 1. Collectively, these two states form the initial superposition $|\Phi_{in}\rangle$. The simulation is then run twice with $|\Phi_0\rangle$, respectively $|\Phi_1\rangle$ as an input, and the superposition of the outputs forms the outcome of the computation.

In effect, what happens in (14) is that instead of allowing the most significant qubit to go to the superposition $\varphi_0|0\rangle + \varphi_1|1\rangle$, it is being fixed to a predetermined state $-\varphi_0|0\rangle$ in the first run and $\varphi_1|1\rangle$ in the second. We call the process of splitting the input superposition „inhibition" and the corresponding rules „inhibitors". Thus, for the first run the inhibitor is „the most significant qubit must be fixed to 0" and for the second – „the most significant qubit must be fixed to 1".

Of course, this is not the only way of parallelizing the execution of (12). The most significant qubit is by no means special – any other qubit can be fixed instead in order to perform 2-way parallel execution. In a similar fashion a 4-way parallel execution can be achieved by fixing two qubits and in general, a 2^n-way parallel execution can be achieved by fixing n qubits – as is the case in (13).

Using the idea outlined above, we have prepared a version of Shor's algorithm suitable for parallel execution. The only change is in Step 3 of the algorithm, where instead of preparing the register in a state of uniform superposition (h on all qubits), we set the k least significant qubits to some predetermined fixed state in order to allow for 2^k-way parallel execution. The set of fixed values for the k least significant qubits are given as arguments to the main function. The corresponding qubit is then left alone if the its value is fixed to 0 and transformed via a **q-not** if the fixed value is 1. *Figure 5* depicts the main function modified accordingly.

All that is left to do is to run the simulation 2^k times, calling **shor** with all binary combinations of k bits as inhibitors. The results for the execution times are summarized in *table 2*. The results clearly indicate that the simulator exhibits more than ideal scaling – in all n-way parallel executions, the speedup factor is more than n. Moreover, there are cases in which the speedup factor enormously exceeds n – the 4-way and 8-way runs for 13^a mod 119, in which the speedup factor is around

```
(defun shor (a N &rest inhibitors)
  (let* ((l (bitsize N))
         (m (* 2 l))
         (*qax* (make-qreg (+ (* 7 l) 3)))
         (iinh 0))
    (dolist (x inhibitors)
      (if (= x 1) (q-not (- m 1 iinh)))
      (incf iinh))
    (dotimes (i (- m iinh)) (h i))
    (a^x%N a 0 N m m)
    (minimize)
    (qft 0 m)
    (minimize)))
```

Figure 5. Slight modification to the main function in order to allow parallel execution

600 instead of 4, respectively 8. This is to be attributed to following fact: the period of 13^a mod 119 is exactly 4. On the other hand, since the inhibitors are placed on the least significant bits, each 4-way (8-way) run contain in the first register only values whose difference is exactly 4 (8), thus they all produce the same result for 13^a mod 119 in the second register. This means that there is no superposition in the second register. The fact that this leads to great speed up proves that the simulation execution time depends mostly on the amount of superposition in the quantum register.

A slowdown factor of 5 – 8 is seen when the size of N is increased with 1 bit. Thus, it can be calculated that about 6 – 7 days would be needed for a non-parallel run for 8-bit N (for example 17^a mod 253). Parallelizing its execution between 8 workstations, the correct result was obtained in less than 12 hours. This proves that parallelizing the execution can not only decrease computation time for smaller tasks, but by doing so it enables computation to be carried out for tasks, whose execution would otherwise be impractical, if not impossible due to resource requirements.

Keeping track of the memory consumption was more difficult, since Lisp has a dynamic garbage collection facility in place and it is hard to calculate the exact amount of used memory, but even for the 59-qubit simulation the peak memory usage of each of the processes never exceeded 250MB. The parallel runs were conducted by the computing elements of the SEE GRID (http://www.see-grid.org), access to which was kindly provided by the Institute for Parallel Processing at the Bulgarian Academy of Science.

10. Conclusion

We have demonstrated an entirely quantum simulation of Shor's factoring algorithm by implementing all required quantum transformations. The largest number factored was the 8-bit number 253 and for this task a 59-qubit quantum register was used. We have thus shown the efficiency and correctness of our general purpose portable quantum computing simulator and

have also shown how easy it is to run simulations for even larger tasks by means of simple parallelization based on the linear superposition principle. By running the independent parallel jobs in Globus-derived GRID environment, we have also shown that such environments can be used for the purpose of quantum algorithm simulations. We believe that our simulator can be used as a valuable tool for quantum algorithm researchers and for the purpose of teaching quantum computing.

Table 2. Average execution times and speedup factor for 2-, 4- and 8-way parallel execution as compared to the non-parallel times

x^a mod N	# of qubits	Average time, s				Speedup factor, times		
		1x	2x	4x	8x	2x	4x	8x
7^a mod 15	31	301.46	20.60	18.31	17.34	14.63	16.46	17.39
5^a mod 21	38	1 705.04	703.03	322.97	210.24	2.43	5.28	8.11
7^a mod 55	45	13 905.89	6 552.51	3 273.74	1 575.83	2.12	4.25	8.82
13^a mod 119	52	62 684.98	24 803.42	105.85	103.66	2.53	592.21	604.72
17^a mod 253	59	-	-	-	46 982.43	-	-	-

Appendix: Source Code Listing

```
;;*****************************************************************************
;;  Shor's factoring algorithm
;;
;;  An implementation of Shor's factoring algorithm on the CL-QGP quantum
;;  computing simulator (slightly modified to save space).
;;
;;  Author: Valentin N. Pavlov
;;  Affiliation: Institute for Parallel Processing, Bulgarian Academy of Science
;;  Contacts: vpavlov@acad.bg, vpavlov@rila.bg, x.pavlov@gmail.com
;;
;;  This source code, along with the source code of the CL-QGP is available
;;  from the author upon request.
;;*****************************************************************************
(load "cl-qgp.lisp")
(use-package :cl-qgp)

;;-----------------------------------------------------------------------------
;;  MAIN
;;-----------------------------------------------------------------------------
(defun shor (x N)
   (let* ((l (bitsize N))
          (m (* 2 l))
          (*qax* (make-qreg (+ (* 7 l) 3))))
      (dotimes (i m) (h i))
      (expt-mod x 0 N m m)
      (minimize)
      (qft 0 m)
      (minimize)
      (shor-result m *qax*)))

(defun bitsize (N)
   (do ((i 0 (1+ i))
        (x 1 (* x 2)))
       ((> x N) i)))

;;-----------------------------------------------------------------------------
;;  MODULAR EXPONENT
;;-----------------------------------------------------------------------------
```

```
(defun carry (ci ai bi ci+1)
    (cc-not ai bi ci+1)
    (c-not ai bi)
    (cc-not ci bi ci+1))

(defun /carry (ci ai bi ci+1)
   (cc-not ci bi ci+1)
   (c-not ai bi)
   (cc-not ai bi ci+1))

(defun sum (c a b)
   (c-not a b)
   (c-not c b))

(defun /sum (c a b)
   (c-not c b)
   (c-not a b))

(defun adder (a0 b0 t0 1)
   (loop for i from (1- 1) downto 0 do
      (if (eql i 0)
            (carry t0 a0 (1+ b0) b0)
            (carry (+ t0 i) (+ a0 i) (+ b0 i 1) (+ t0 i -1))))
   (c-not a0 (1+ b0))
   (loop for i from 0 below 1 do
      (if (not (eql i 0))
            (/carry (+ t0 i) (+ a0 i) (+ b0 i 1) (+ t0 i -1)))
      (sum (+ t0 i) (+ a0 i) (+ b0 i 1))))

(defun /adder (a0 b0 t0 1)
   (loop for i from (1- 1) downto 0 do
      (/sum (+ t0 i) (+ a0 i) (+ b0 i 1))
      (if (not (eql i 0))
            (carry (+ t0 i) (+ a0 i) (+ b0 i 1) (+ t0 i -1))))
   (c-not a0 (1+ b0))
   (loop for i from 0 below 1 do
      (if (eql i 0)
            (/carry t0 a0 (1+ b0) b0)
            (/carry (+ t0 i) (+ a0 i) (+ b0 i 1) (+ t0 i -1)))))

(defun adder-mod (a0 b0 N t0 1)
   (let ((toN (0->N N (+ t0 1) 1))
      (c-toN (c-0->N (+ t0 (* 2 1)) N (+ t0 1) 1)))
       (mapcar #'(lambda (x) (apply (car x) (cdr x))) toN)
      (adder a0 b0 t0 1)
      (/adder (+ t0 1) b0 t0 1)
      (q-not b0)
      (c-not b0 (+ t0 (* 2 1)))
      (q-not b0)
       (mapcar #'(lambda (x) (apply (car x) (cdr x))) c-toN)
      (adder (+ t0 1) b0 t0 1)
       (mapcar #'(lambda (x) (apply (car x) (cdr x))) c-toN)
      (/adder a0 b0 t0 1)
      (c-not b0 (+ t0 (* 2 1)))
      (adder a0 b0 t0 1)
       (mapcar #'(lambda (x) (apply (car x) (cdr x))) toN)))

(defun /adder-mod (a0 b0 N t0 1)
   (let ((toN (0->N N (+ t0 1) 1))
      (c-toN (c-0->N (+ t0 (* 2 1)) N (+ t0 1) 1)))
      (mapcar #'(lambda (x) (apply (car x) (cdr x))) toN)
      (/adder a0 b0 t0 1)
      (c-not b0 (+ t0 (* 2 1)))
      (adder a0 b0 t0 1)
      (mapcar #'(lambda (x) (apply (car x) (cdr x))) c-toN)
      (/adder (+ t0 1) b0 t0 1)
      (mapcar #'(lambda (x) (apply (car x) (cdr x))) c-toN)
      (q-not b0)
      (c-not b0 (+ t0 (* 2 1)))
      (q-not b0)
```

```
      (adder (+ t0 1) b0 t0 1)
      (/adder a0 b0 t0 1)
      (mapcar #'(lambda (x) (apply (car x) (cdr x))) toN)))

(defun ctrl-mult-mod (c x a0 N y0 t0 l)
   (dotimes (i l)
      (let* ((x*2^i (mod (* x (expt 2 i)) N))
             (x*2^i-gates (cc-0->N c (+ a0 l -1 (- i)) x*2^i t0 l)))
        (mapcar #'(lambda (x) (apply (car x) (cdr x))) x*2^i-gates)
        (adder-mod t0 y0 N (+ t0 l) l)
        (mapcar #'(lambda (x) (apply (car x) (cdr x))) x*2^i-gates)))
   (q-not c)
   (dotimes (i l)
      (cc-not c (+ a0 i) (+ y0 i 1)))

   (q-not c))

(defun /ctrl-mult-mod (c x a0 N y0 t0 l)
   (q-not c)
   (loop for i from (1- l) downto 0 do
      (cc-not c (+ a0 i) (+ y0 i 1)))
   (q-not c)
   (loop for i from (1- l) downto 0 do
      (let* ((x*2^i (mod (* x (expt 2 i)) N))
             (x*2^i-gates (cc-0->N c (+ a0 l -1 (- i)) x*2^i t0 l)))
        (mapcar #'(lambda (x) (apply (car x) (cdr x))) x*2^i-gates)
        (/adder-mod t0 y0 N (+ t0 l) l)
        (mapcar #'(lambda (x) (apply (car x) (cdr x))) x*2^i-gates))))

(defun expt-mod (x a0 N t0 m)
   (let ((l (/ m 2)))
      (q-not (+ t0 l))
      (dotimes (i m)
         (let* ((x^2^i (mod (expt x (expt 2 i)) N))
                (x^-2^i (inv N x^2^i)))
      (ctrl-mult-mod (+ a0 m -1 (- i)) x^2^i (+ t0 1) N
                     (+ t0 l 1) (+ t0 m 2) l)
      (minimize)
      (/ctrl-mult-mod (+ a0 m -1 (- i)) x^-2^i (+ t0 l 2) N t0 (+ t0 m 2)l)
      (minimize)
      (dotimes (j l)
         (c-not (+ t0 l 2 j) (+ t0 1 j))
         (c-not (+ t0 1 j) (+ t0 l 2 j)))))))

(defun inv (a b)
   (let ((as (list 1 0 a))
      (bs (list 0 1 b)))
      (loop while (/= 0 (third bs)) do
         (let* ((q (floor (/ (third as) (third bs))))
                (ts (mapcar #'(lambda (a b) (- a (* b q))) as bs)))
           (setf as bs)
           (setf bs ts)))
      (if (< (second as) 0)
      (+ a (second as))
      (second as))))

(defmacro map-1s (op)
   `(let ((res nil))
      (dotimes (i l)
         (if (logbitp i N)
         (setf res (cons ,op res))))
      (nreverse res)))

(defun 0->N (N q0 l)          (map-1s `(q-not ,(+ q0 l -1 (- i)))))
(defun c-0->N (c N q0 l)      (map-1s `(c-not ,c ,(+ q0 l -1 (- i))))))
```

```
(defun  cc-0->N  (c1  c2  N  q0  1)  (map-1s  '(cc-not,c1,c2,(+  q01-1(-i)))))

;;-------------------------------------------------------------------------------
;;  QUANTUM  FOURIER  TRANSFORM
;;-------------------------------------
(defun  qft  (q0  l)
    (dotimes  (i  l)
        (dotimes  (j  i)
            (qft-S  (+  q0  i)  (+  q0  j)))
        (qft-R  (+  q0  i))))

(defun  qft-R  (j)
    (h  j))

(defun  qft-S  (j  k)
    (let  ((angle  (/  pi  (expt  2  (-  j  k)))))  ;;  !!!  because  of  reversed  notation
        (c-phasor  j  k  angle)))


(defun  c-phasor  (C  X  theta)
    (rz  C  (-  (/  theta  2)))
    (ph  C  (/  theta  4))
    (rz  X  (-  theta))
    (c-not  C  X)
    (rz  X  (/  theta  2))
    (c-not  C  X)
    (rz  X  (/  theta  2)))

;;-------------------------------------------------------------------------------
;;  DISPLAY  RESULTS
;;-------------------------------------------------------------------------------
(defun  shor-result  (size1  &rest  registers)
    (let*  ((res  (apply  #'qreg-combine  registers))
        (size  (slot-value  (first  registers)  'cl-qgp::size))
        (size2  (1+  (/  size1  2)))
        (mask2  (1-  (expt  2  size2))))
      (mapcar
        #'(lambda  (elem)
        (let  ((state  (car  elem))
                (prob  (cadr  elem)))
            (if  (=  (logand
                (truncate  state  (expt  2  (-  size  size1  size2)))  mask2)  1)
                (format  t  "~a~c~a~%"
                    (bit-reverse  size1
                        (truncate  state  (expt  2  (-  size  size1))))
                    #\Tab
                    prob))))
        res))
  nil)

(defun  bit-reverse  (n  int)
    (if  (=  int  0)  (return-from  bit-reverse  0))
    (let  ((res  0))
      (dotimes  (i  n)
        (if  (logbitp  i  int)
        (setf  res  (logior  res  (expt  2  (-  n  i  1))))))
      res))
```

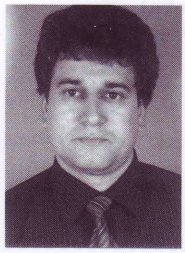
References

1. Pavlov, V. Efficient Quantum Computing Simulation.Proceedings of IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing. IEEE Computer Society Press, 2006, 235-239.
2. Shor, P. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. Proceedings of the 35th Annual Symposium on Foundations of Computer Science. IEEE Computer Society Press, 1994,124-134. (http://xxx.lanl.gov/pdf/quant-ph/9508027).
3. Barenco, A., C. Bennett, R. Cleve, et al. Elementary Gates for Quantum Computation. *Physical Review Letters* A, 52, 1995, 3457.
4. Vedral, V., A. Barenco and A. Ekert. QuantumNetworks for Elementary Arithmetic Operations. (http://xxx.lanl.gov/pdf/quant-ph/9511018).
5. Benett, C. Logical Reversibility of Computation-*IBM Journal of Research and Development,* 1973,17,525-532. (http://www.research.ibm.com/journal/rd/176_ibmrd1706G.pdf).
6. Dirac, P. A. M. The Principles of Quantum Mechanics. Oxford University Press, Oxford, 1947.

Additional Sources

7. Lenstra, H., H. Lenstra, Jr., M. Manasse, et al. The Number Field Sieve. Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, ACM, New York, 1990, 564-572.
8. Pavlov, V., H. Turlakov and K. Boyanov. Quantum Computation and Quantum Computers (Квантово изчисление и квантови компютри).-*Engineering Sciences*, XLII, 2005, 2, 5-28.
9. Deutsch, D. Quantum Computational Networks. Proceedings of the Royal Society, Ser. A, 425, 1989, 73-90.
10.Benioff, P. The Computer as a Physical System: Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines.-*Journal of Statistical Physics*, 22, 1980, 563-591.
11.Feynman, R. Simulating Physics with Computers International Journal of Theoretical Physics, 21, 1982,46-488.
12.Deutsch, D. and R. Jozsa. Rapid Solution of Problems by Quantum Computation.-Proceedings of the Royal Society, Ser. A, 439, 1992, pp. 553-558.
13. Landauer, R. Irreversibility and Heat Generation in the Computing Process. *IBM Journal*, 1961, 183-191. (http://www.research.ibm.com/journal/rd/053/ibmrd0503C.pdf).
14. Eisert, J. and M. M. Wolf. *Quantum Computing.* (http:xxx.lanl.gov/pdf/quant-ph/0401019).
15. DiVincenzo, D. *Two-bit Gates are Universal for Quantum Computation.* (http://xxx.lanl.gov/pdf/quant-ph/9407022).
16. Barenco, A. *A Universal Two-bit Gate for Quantum Computation.*(http://xxx.lanl.gov/pdf/quant-ph/9505016).
17. Deutsch, D., A. Barenco and A. Ekert. *Universality in Quantum Computation.* (http://xxx.lanl.gov/pdf/quant-ph/9505018).
18. Deutsch, D. and A. Ekert. *Machines, Logic and Quantum Physics* (http://xxx.lanl.gov/pdf/math.HO/9911150).

Manuscript received on 12.06.2007



Eng. Valentin Pavlov *(born 1974) has graduated from the Technical University of Sofia during 1997, „Computing Systems" speciality, with MSc thesis „Algorithm for discovering the generators, orbits and the order of the automorphism group of a graph". Since 1999 he is working in Rila Solutions EAD, currently occupying the Chief Technology Officer position. Since 2002 he is a PhD student at the Institute for Parallel Processing at the Bulgarian Academy of Sciences. In July 2007 he passed the preliminary defense of his PhD thesis „Quantum Computing Simulation". He is the laureate of the 2005 „John Atanasoff" Award of the President of Republic of Bulgaria for scientific and professional achievements in the IT field. His scientific interests include: quantum computing, GRID, distributed systems, computer and network architectures, protocols and standards.*

Contacts:
e-mail: vpavlov@rila.bg

continuation from 5

Kalinka M. Kaloyanova *received her M.Sc. in Computer Science from University of Sofia (1981), and her PhD degree in Computer Science from the Central Institute of Computer Technique and Technologies, Sofia(1989). She was Research Fellow in the Central Institute for Computing Technics and Technologies, Sofia (1981-1995) and BTC - Institute for Scientific Research of Telecommunications (1995-2003). Since 2003 she she has been working as Assoc. Professor at the University of Sofia, Faculty of Mathematics and Informatics.*

Contacts:
e-mail: kkaloyanova@fmi-uni-sofia.bg

Prof. DSc Kiril Boyanov *is a Academician at the Bulgarian Academy of Sciences. He is Director of Institute for Parallel Processing.He is the Bulgarian Representative in IFIP; Chairman of Bulgarian Computer Society of IEEE; Chairman of National IFIP Committee; National Coordinator of IST Programme; Member of IST Advisory Group (ISTAG) - European Commission. He has more than 100 publications in Bulgarian, more than 40 publications in foreign scientific magazines and international conferences proceedings. Author and co-author of 36 books and manuals (in Bulgarian mainly, several of them published in Russian). Editor of 4 books of conference proceedings published by North-Holland.Chief Editor of journal „Information Technologies and Control" - Bulgaria.*

Contacts:
e-mail: boyanov@acad.bg