Quantum Computing, Shor's Algorithm, and Parallelism

Arun Bhalla, Kenneth Eguro, Matthew Hayward

Original 2001, Revised 2015

Contents

1 Introduction

The size of components in classical computers is shrinking exponentially, if trends continue soon the behavior of the components will be dominated more by quantum physics than classical physics. Researchers have begun investigating the implications of these quantum behaviors on computation. Surprisingly it seems that a computer whose components are able to function in a quantum manner are more powerful than any classical computer can be.

The study of quantum physics has shown that the behaviors of physical systems that understood from classical physics break down in systems of sufficiently small scale. The scale which these classical assumptions break down is the scale we are rapidly approaching with components in modern computers. At the current rate of miniaturization memory components will reach the size of about one atom per bit around the year 2020. At this size the storage for a bit cannot be expected to behave in a classical manner, and the size of an elementary particle is an absolute lower bound for the classical bit size. This bound does not hail the eventual end to advances in computing hardware. The same quantum effects which will prevent the continual miniaturization of classical computers may allow the development of a quantum computer, which relies upon these quantum effects.

At the heart of quantum computing lies parallelism. This inherent parallelism, combined with the exponential slowdown in any classical simulation of a quantum system make simulation of a quantum system an ideal candidate for speedup through parallelism.

2 The Quantum Computer

2.1 The Classical Bit

In a classical computer a bit is typically stored in a silicone chip, a metal hard drive platter, or on a magnetic tape. About 10^5 atoms were used in the year 2000 to represent one bit of information. The smallest conceivable storage for a bit involves a single elementary particle of some sort. For example a spin-1/2 of particle such as a proton, neutron, or electron, which can be characterized by its spin value. The spin is measured to be either $+1/2$ or $-1/2$. We can encode 1 to be $+1/2$ and 0 to be $-1/2$, and if we assume we can measure and manipulate the spin of such a particle then we could theoretically use this particle to store one bit of information. If we were to try to use this spin-1/2 particle as a classical bit, one that is always in the 0 or 1 state, we would fail. We would be trying to apply classical physics on a scale where it simply is not applicable. This single spin-1/2 particle will instead act in a quantum manner. (Williams, Clearwater)

2.2 The Quantum Bit

This spin-1/2 particle which behaves in a quantum manner could be the fundamental building block of a Quantum computer. We could call it a qubit, to denote that it is analogous in some ways to a bit in a classical computer. Just as a memory register in a classical computer is an array of bits, a quantum memory register is composed of several spin-1/2 particles, or qubits. There is no particular need for the spin-1/2 particle, we could equally well use a Hydrogen atom, and designate its electron being measured in the ground state to be the 0 state, and it being in the first excited state to be the 1 state. There are a multitude of possible qubit representations that will work. For simplicity only the spin-1/2 particle will be discussed from here on.

2.3 State Vectors and Dirac Notation

We wish to know exactly how the behavior of the spin-1/2 particle, our qubit, differs from a that of a classical bit. A classical bit can store either a 1 or a 0, and when measured the value observed will always be the value stored. Quantum physics states that when we measure the spin-1/2 particles state we will determine that it is in the $+1/2$ state, or the spin $-1/2$ state. In this manner our qubit is not different from a classical bit, for it can be measured to be in the $+1/2$, or 1 state, or the $-1/2$, or 0 state. The differences between the qubit and the bit come from what sort of information a qubit can store when it is not being measured.

According to quantum physics we may describe that state of this spin-1/2 particle by a state vector in a Hilbert Space. A Hilbert Space is a complex linear vector space.

A complex vector space is one in which the lengths of the vectors within the space are described with complex numbers. A linear vector space is one that you may add and multiply vectors that lie in the space and the resulting vector will still lie within that space. (Williams, Clearwater)

The Hilbert Space for a single qubit will have two perpendicular axes, one corresponding to the spin-1/2 particle being in the $+1/2$ state, and the other to the particle being in the $-1/2$ state. These states which the vector can be measured to be are referred to as *eigenstates*. The vector which exists somewhere in this space which represents the state of the spin-1/2 particle is called the state vector. The projection of the state vector onto one of the axes shows the contribution of that axes' eigenstate to the whole state. This means that in general, the state of the spin-1/2 particle can be any combination of the base states. In this manner a qubit it totally unlike a bit, a bit can exist in only the 0 or 1 state, but the qubit can exist, in principle, in any combination of the 0 and 1 state, and is only constrained to be in the 0 or 1 state when we measured.

Now we introduce the standard notation for state vectors in Quantum physics. The state vector is written the following way is called a *ket vector* $|\psi>$. Where ψ is a list of numbers which contain information about the projection of the state vector onto its base states. The term ket and this notation come from the physicist Paul Dirac who wanted a concise shorthand way of writing formulas that occur in Quantum physics. These formulas frequently took the form of the product of a row vector with a column vector. Thus he referred to row vectors as *bra vectors* represented as $< y|$. The product of a bra and a ket vector would be written $< y|x >$, and would be referred to as a *bracket.* (Williams, Clearwater)

2.4 Superposition and Eigenstates

Earlier we said that the projection of the state vector onto one of the perpendicular axes of its Hilbert Space shows the contribution of that axes' eigenstate to the whole state. According to quantum physics a quantum system can exist in a mix of all of its allowed states simultaneously. This is the Principle of Superposition, and it is key to the power of the quantum computer. While the physics of superposition is not simple at all, mathematically it is not difficult to characterize this kind of behavior.

2.5 The Quantum qubit

Back to our qubit, the spin-1/2 particle. We know that while it can only be measured to have a spin of $+1/2$ or $-1/2$, it may in general be in a superposition of these states when we are not measuring it.

Let x_1 be the eigenstate corresponding to the spin $+1/2$ state, and let x_0 be the eigenstate corresponding to the spin $-1/2$ state. Let X be the total state of the state vector, and let w_1 and w_0 be the complex numbers that weight the contribution of the base states to the total state, then in general:

$$|X >= w_0 * |x_0 > +w_1 * |x_1 >== (w_0, w_1)$$

At this point it should be remembered that w_0 and w_1, the weighting factors of the base states are complex numbers, and that when the state of X is measured, we are guaranteed to find it to be in either the state:

$$0 * |x_0 > +w_1 * |x_1 >== (0, w_1)$$

or the state

$$w_0 * |x_0 > +0 * |x_1 >== (w_0, 0)$$

The state vector is a unit vector in a Hilbert space, which is similar to vector spaces you may be more familiar with, but it differs in that the lengths of the vectors are complex numbers. It is not necessary from a physics perspective for the state vector to be a unit vector (meaning it has a length of 1), but it makes for easier calculations further on, so we will assume from here on out that the state vector has length 1. This assumption does not invalidate any claims about the behavior of the state vector.

Thus we have fully defined the basic building block of a quantum computer, the qubit. It is fundamentally different from a classical bit in that it can exist in any superposition of the 0 and 1 states when it is not being measured. (Barenco, Ekert, Sanpera, Machiavello)

2.6 The Quantum Memory Register

Up till now we have considered a two state quantum system, specifically a spin-1/2 particle. However a quantum system is by no means constrained to be a two state system. Much of the above discussion for a 2 state quantum system is applicable to a general n state quantum system.

In an n state system the Hilbert Space has n perpendicular axes, or eigenstates, which represent the possible states that the system can be measured in. As with the two state system, when we measure the n state quantum system, we will always find it to be in exactly one of the n states, and not a superposition of the n states. The system is still allowed to exist in any superposition of the n states while it is not being measured.

Mathematically if two state quantum system with coordinate axes x_0, x_1 can be fully described by:

$$|X>= w_0 * |x_0> + w_1 * |x_1> == (w_0, w_1)$$

Then an n state quantum system with coordinate axes $x_0, x_1, \ldots, x_{n-1}$ can be fully described by:

$$|X>= \sum_{k=0}^{n-1} w_k * |x_k>$$

In general a quantum system with n base states can be represented by the n complex numbers w_0 to w_{n-1}. When this is done the state may be written as:

$$|X>= \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{pmatrix}$$

Where it is understood that w_k refers to the complex weighting factor for the k'th eigenstate.

Using this information we can construct a quantum memory register out of the qubits described in the previous section. Note that in general a quantum register composed of n qubits requires 2^n complex numbers to completely describe its state. A n qubit register can be measured to be in one of 2^n states, and each state requires one complex number to represent the projection of that total state onto that state. In contrast a classical register composed of n bits requires only n integers to fully describe its state.

This means that one can store an exponential amount of information in a quantum register relative to the number of qubits it contains. Here we see some of the first hints that a quantum computer can potentially be exponentially more efficient than a classical computer in some respects.

2.7 Probability Interpretation

Now that we know how to represent a state vector as a superposition of states, and yet can only measure the state vector to be in one of the base states. We must determine what happens when we measure the state vector.

The only way to observe the state of the state vector is to in some way cause the quantum mechanical system to interact with the environment. When the state vector is observed it makes a sudden discontinuous jump to one of the eigenstates. (Williams, Clearwater)

To perform any sort of useful calculation we must be able so say something about which base state a quantum mechanical system will collapse into. The probability that the state vector will collapse into the j'th eigenstate, is given by $|w_j|^2$ which is defined to be $a_j^2 + b_j^2$ if $w_j = a_j + i * b_j$, where w_j is the complex projection of the state vector onto the j'th eigenstate. In general the chance of choosing any given state is

$$Prob(j) = \frac{|w_j|^2}{\sum_{k=0}^{n-1} |w_k|^2}$$

but as mentioned earlier we will insist on having the state vector of length one, and in this case the probability expression simplifies to $Prob(j) = |w_j|^2$.

2.8 Quantum Parallelism

Quantum parallelism arises from the ability of a quantum memory register to exist in a superposition of base states. Each component of this superposition may be thought of as a single argument to a function. A function performed once on the register in a superposition of states is performed on each of the components of the superposition. Since the number of possible states is 2^n where n is the number of qubits in the quantum register, you can perform in one operation on a quantum computer what would take an exponential number of operations on a classical computer. This is fantastic, but the more superposed states that exist in you register, the smaller the probability that you will measure any particular one will become.

As an example suppose that you are using a quantum computer to calculate the function $\mathcal{F}(x) = 2 * x \bmod 7$, where x is the superposition of integers between 0 and 7 inclusive. You could prepare a quantum register that was in a equally weighted superposition of the states 0-7. Then you could perform the $2 * x \bmod 7$ operation once, and the register would contain the equally weighted superposition of 1,2,4,6,1,3,5,0 states, these being the outputs of the function $2 * x \bmod 7$ for inputs 0 - 7. When measuring the quantum register you would have a 2/8 chance of measuring 1, and a 1/8 chance of measuring any of the other outputs. It would seem that this sort of parallelism is not useful, as the more we benefit from parallelism the less likely we are to measure a value of a function for a particular input. Some clever algorithms have been devised, most notably by Peter Shor which succeed in using quantum parallelism on a function where there is interest in some property of all the inputs, not just a particular one.

This kind of parallelism is very appealing for simulation on a parallel computer. A n bit quantum register contains a superposition of each of its 2^n

possible base states, and we represent this by an array of 2^n complex numbers which are probabilities of measuring the quantum register to be the corresponding base state. To perform an operation on the quantum register, we simply modify each of the 2^n array locations. By splitting the calculations of how to change the probability values of the array locations into even ranges via process elements, we get nearly linear speedup.

3 Shor's Algorithm

3.1 Introduction to Shor's Algorithm

By 1993 it was know that a quantum computer could perform certain tasks asymptotically faster than any classical computer. Nonetheless research into quantum computing was largely driven by academic curiosity. In 1994 Peter Shor, a scientist working for Bell Labs, devised a polynomial time algorithm for finding prime factors of large numbers on a quantum computer. This discovery drew great attention to the field of quantum computing.

Shor's algorithm is viewed as important because the difficulty of finding prime factors of large numbers is relied upon for most cryptography systems. If an efficient method of factoring large numbers were to be discovered most of the current encryption schemes would be easily compromised. While it has not been proven that factoring large numbers can not be archived on a classical computer in polynomial time, the fastest published algorithm for factoring large number as of 2015 in $O(exp(\frac{64}{9}n^{1/3}(log\ n)^{2/3})$, operations where n is the number of bits used to represent the number: this runtime exceeds polynomial time. In contrast Shor's algorithm runs in $O((\log n)^2 * \log\log n)$ on a quantum computer, and then must perform $O(\log n)$ steps of post processing on a classical computer. Overall this time is polynomial. This discovery propelled the study of quantum computing forward, as such an algorithm is much sought after. (Shor)

3.2 Overview of Shor's Algorithm

Shor's algorithm hinges on a result from number theory. This result is: The function $\mathcal{F}(a) = x^a \bmod n$ is a periodic function, where x is an integer coprime to n. In the context of Shor's algorithm n will be the number we wish to factor. When two numbers are coprime it means that their greatest common divisor is 1.

Calculating this function for an exponential number of a's would take exponential time on a classical computer. Shor's algorithm utilizes quantum parallelism to perform the exponential number of operations in one step.

The reason why this function is of utility in factoring large numbers is this:

Since $\mathcal{F}(a)$ is a periodic function, it has some period r. We know that $x^0 \bmod n = 1$, so $x^r \bmod n = 1$, and $x^{2r} \bmod n = 1$ and so on since the function is periodic.

Given this information and through the following algebraic manipulation:

$$x^r \equiv 1 \bmod n$$

$$(x^{r/2})^2 = x^r \equiv 1 \bmod n$$

$$(x^{r/2})^2 - 1 \equiv 0 \bmod n$$

and if r is an even number

$$(x^{r/2} - 1)(x^{r/2} + 1) \equiv 0 \bmod n$$

We can see that the product $(x^{r/2} - 1)(x^{r/2} + 1)$ is an integer multiple of n, the number to be factored. So long as $x^{r/2}$ is not equal to $+-1$, then at least one of $(x^{r/2} - 1)$, $(x^{r/2} + 1)$ must have a nontrivial factor in common with n. So by computing $\gcd(x^{r/2} - 1, n)$, and $\gcd(x^{r/2} + 1, n)$, we will obtain a factor of n, where gcd is the greatest common denominator function.

3.3 Steps to Shor's Algorithm

Shor's algorithm for factoring a given integer n can be broken into some simple steps.

Step 1 Determine if the number n is a prime, a even number, or an integer power of a prime number. If it is we will not use Shor's algorithm. There are efficient classical methods for determining if a integer n belongs to one of the above groups, and providing factors for it if it is. This step would be performed on a classical computer.

Step 2 Pick a integer q that is the power of 2 such that $n^2 \leq q < 2n^2$. This step would be done on a classical computer.

Step 3 Pick a random integer x that is coprime to n. When two numbers are coprime it means that their greatest common divisor is 1. There are efficient classical methods for picking such an x. This step would be done on a classical computer.

Step 4 Create a quantum register and partition it into two parts, register 1 and register 2. Thus the state of our quantum computer can be given by: $|\text{reg1}, \text{reg2}\rangle$. Register 1 must have enough qubits to represent integers as large as $q - 1$. Register 2 must have enough qubits to represent integers as large as $n - 1$. The calculations for how many qubits are needed would be done on a classical computer.

Step 5 Load register 1 with an equally weighted superposition of all integers from 0 to $q - 1$. Load register 2 with all zeros. This operation would be performed by our quantum computer. The total state of the quantum memory register at this point is:

$$\frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |a, 0\rangle$$

Step 6 Now apply the transformation $x^a \bmod n$ to for each number stored in register 1 and store the result in register 2. Due to quantum parallelism this will take only one step, as the quantum computer will only calculate $x^{|a\rangle} \bmod n$, where $|a\rangle$ is the superposition of states created in step 5. This step is performed on the quantum computer. The state of the quantum memory register at this point is:

$$\frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |a, x^a \bmod n\rangle$$

Step 7 Measure the second register, and observe some value k. This has the side effect of collapsing register one into a equal superposition of each value a between 0 and $q-1$ such that

$$x^a \bmod n = k$$

This operation is performed by the quantum computer. The state of the quantum memory register after this step is:

$$\frac{1}{\sqrt{||A||}} \sum_{a'=a' \in A} |a', k\rangle$$

Where A is the set of a's such that $x^a \bmod n = k$, and $||A||$ is the number of elements in that set.

Step 8 Now compute the discrete Fourier transform on register one. The discrete Fourier transform when applied to a state $|a\rangle$ changes it in the following manner:

$$|a\rangle = \frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} |c\rangle * e^{2\pi iac/q}$$

This step is performed by the quantum computer in one step through quantum parallelism. After the discrete Fourier transform our register is in the state:

$$\frac{1}{\sqrt{||A||}} \sum_{a' \in A} \frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} |c, k\rangle * e^{2\pi ia'c/q}$$

Step 9 Measure the state of register one, call this value m, this integer m has a very high probability of being a multiple of q/r, where r is the desired period. This step is performed by the quantum computer.

Step 10 Take the value m, and on a classical computer do some post processing which calculates r based on knowledge of m and q. In particular:

- m has a high probability of being $= \lambda * (q/r)$ where λ is an integer
- If we perform floating point division of m/q, and then calculate the best rational approximation to m/q whose denominator is less than or equal to q

- We take this denominator to be a candidate for r.
- If our candidate r is odd we either double it if doing so leads to a value less than q

There are efficient ways to do this post processing on a classical computer.

Step 11 Once you have attained r, a factor of n can be determined by taking $\gcd(x^{r/2}+1, n)$ and $\gcd(x^{r/2}-1, n)$. If you have found a factor of n, then stop, if not go to step 4. This final step is done on a classical computer.

Step 11 contains a provision for what to do if Shor's algorithm failed to produce factors of n. There are a few reasons why Shor's algorithm can fail, for example the discrete Fourier transform could be measured to be 0 in step 9, making the post processing in step 10 impossible. The algorithm will sometimes find factors of 1 and n, which is not useful either. For these reasons step 11 must be able to jump back to step 4 to start over. (Williams, Clearwater)

3.4 Parallelizing Shor's Algorithm

The vast majority of the time spent in the processing of Shor's algorithm is in the discrete Fourier transform step. In the discrete Fourier transform we iterate from 0 to q, and for each possible value in that range we iterate over the entire register and perform some mathematical operations. It is trivial to divide this work among multiple process elements. One can simply iterate on each process element from 0 to q, and for each value in the range iterate over some prescribed subrange of the register.

In general Shor's algorithm simulation seems a good candidate for parallelization. The simulation can roughly be divided into three phases: prepossessing, simulation of the quantum register, and post processing. During the simulation of the quantum register, all the work is done in the form of applying the same operation to an entire array, where each array location represents one of the base states of the quantum register. This agrees with our conception of how a quantum register would function, as in a quantum computer, we are not free to perform an operation on only certain portions of the superposed state of the register, we must perform the operation on all portions.

4 A Simulation of Shor's Algorithm on a Classical Computer

As noted by Feynman and others it is impossible to simulate a quantum mechanical system on a classical computing device without incurring exponential slowdown, thus the simulation is only tolerable to observe for relatively small numbers. The simulation implements all of the steps in Shor's algorithm, and successfully factors numbers. All of the simulation code is written in C++ and can be found in appendix D.

4.1 Introduction to the Code for the Simulation

The code base for the simulation of Shor's algorithm consists of eight files.

complex.h/C : These files contain the simple complex number class that we wrote for storing state information about the quantum memory register in the simulation.

qureg.h/C : This file contains the quantum register class that simulates the behavior of the quantum memory register in Shor's algorithm. It is generic enough that it may be made to simulate any quantum memory register, although what happens in the event of a collapse not due to the direct measurement of the register is left to the programmer to perform. An example of such a collapse that is the collapse of the first partition of Shor's quantum memory register in step 7 of his algorithm.

util.h/C : This is a library of functions used by shor.C

timer.C : Function for measuring the running time of Shor's algorithm.

barrier.h/C : Contains code supporting barriers, where threads must synchronize.

range.h/C : Contains code for splitting up work between threads.

shor.C : Contains the simulation of the steps of Shor's algorithm.

4.2 The Simulation of Shor's Algorithm

The implementation of Shor's algorithm found in shor.C follows the steps outlined in the description of Shor's algorithm found above. There are some significant differences in the behavior of the simulator and the behavior of a actual quantum computer.

The simulator uses 2^{n+1} double precision floating point numbers to represent the state of the quantum register. It uses one Complex object to represent the probability amplitude of each of the 2^n eigenstates of a n bit quantum memory register, and each Complex object uses two double precision floating point numbers. A real quantum computer would use exactly n qubits as its memory register.

A second large difference is that during the modular exponentiation step of Shor's algorithm (Step 6 above) a quantum computer would perform one operation of $x^a \bmod n$, where a is the superposition of states in register 1. The simulation must calculate the superposition of values caused by calculating $x^a \bmod n$ for $a = 0$ through $q - 1$ iteratively. The simulation also stores the result of each modular exponentiation, and uses that information to collapse register 1 in step 7 in Shor's algorithm. A quantum computer would not be capable of performing this book keeping, as examining any particular result would collapse the existing superposition to be placed in register 2 at the end of step 6 of Shor's algorithm. A quantum computer would have no need whatsoever to do such book keeping, as when register 2 is measured in step 7, the collapse of register 1 is an automatic and unavoidable consequence of the measurement. A analogous argument follows for the use of the get probability function in step 8 of Shor's algorithm for calculating the discrete Fourier transform.

Aside from these differences, which are necessitated by the inability of a classical computer to accurately depict the behavior of a quantum mechanical system, the operations are performed by the shor.C program are identical to those called for in the description of Shor's algorithm.

As the algorithm runs the state of the quantum memory register changes in the manner laid out in the description of Shor's algorithm. After the final measurement of register 1 in step 9 we obtain some integer m, which has a high probability of being an integer multiple λ of q/r. r is the period of $x^a \bmod n$ that we are trying to find, so that we may calculate $x^{r/2}-1 \bmod n$ and $x^{r/2}+1 \bmod n$ in an effort to find numbers which share factors with n.

In the post processing step shor.C takes the integer m and divides it by q, thus yielding some number c which is approximately equal to λ/r, where λ is an integer, and r is the desired period. Then using a helper function it calculates the best rational approximation to c which has a denominator that is less than or equal to q (recall that q is the power of two such that $n^2 \leq q < 2n^2$, where n is the number to be factored by Shor's algorithm). The period must be less than or equal to q, because there are only q values total, so for them to be periodic, the period must be less than q.

We take this denominator to be the desired period. Shor's algorithm can only use even periods in determining factors of n, and so we check to see if r is even, if not we check to see if doubling the period would still yield a period less than q, if so we double the guessed period.

Taking the period, which is now guaranteed to be even, we proceed to calculate $x^{r/2}-1 \bmod n$ and $x^{r/2}+1 \bmod n$, calling these values a and b, we compute the greatest common denominator of a and n, and b and n, to see if we have attained a nontrivial factor of n.

4.3 Parallelizing Methodology

There is a strong correlation between the portions of Shor's algorithm which would be performed on a classical computer, portions which would be performed on a quantum computer, and portions of the simulation that do not parallelize, and portions that do.

The parts of Shor's algorithm which are pre and post processing which take place on a classical computer are not good candidates for parallelization. In contrast, the portions which the quantum computer would perform are easily parallelized.

The basis for parallelization in the simulation of Shor's algorithm is that at several points in the sequential code there are for loops that iterate over large arrays, and modify each element in some uniform manner.

Given this type of parallelism, each of the Charm++, pthreads, and MPI paradigms has some appeals and disadvantages. Charm++'s object paradigm coincides with the sequential codes object oriented approach. Charm++'s load balancing features are not of much utility, as the work load between even array portions is very even to start with, and it is not clear that the overhead imposed by Charm++ would be recovered by superior load balancing. MPI seems reasonable, as each process element iterates over its own exclusive region in the parallelized code, however, we then must communicate each portion to a manager processor, who would perform various operations.

Pthreads seemed the most natural version, since each thread iterates over its unique set of array locations, there is no need for locks, and no danger of deadlock or data corruption. If the array locations are suitably large, there is very little false sharing due to different array portions existing in the same cache lines of different threads. Once deciding on the pthreads paradigm, there are two hurdles that must be overcome. We must decide on a synchronization method, and we must decide how to split work.

Splitting the work is achieved in range.h, where given the values of n, q, and `num pthreads`, assigns values to each of `num pthreads` array locations, such that the i'th array location of `q range lower`, `q range upper`, `n range lower` and `n range upper` contains the array locations where the i'th process element should begin and end processing.

Setting barriers is achieved in barrier.h. We simply implement a barrier with pthread locks. These barriers occur before and after each of the parallel sections. They are necessary because frequently we perform an operation that involves some result of the entire array just before or after these parallelized sections.

In accordance with Amdahl's law our speedup is limited by the speedup of the parallelized sections, but the parallelized sections are such a large portion of the total running time, that the speedup is nearly linear.

5 Timing Results and speedup

Even given the small amount of code that was parallelized, we got remarkably good speedup results.

As can be seen, there is not a strong correlation between number of processors and run time for the trials factoring 15 (4 bits), although this can be accounted for by the fact that the run time is so small initially, and for all number of threads the run time is less than one fifth of a second. This will change rapidly, as the running time grows exponentially with n, the number to be factored.

In this graph for the factoring of 21 (5 bits), we begin to see possible speedup due to multiple threads, although it is still sporadic, again, this is not surprising due to the very small running times of the sequential version.

In this graph for running times and speedups while factoring 33 (6 bits), we see the beginning of our linear speedup with the number of pthreads. With the exception of the speedup for two threads, we follow a roughly liner speedup curve.

In the graph for the speedup with thread increase while factoring 77 (7 bits) we see even better behavior than we did for factoring 33. There is initially linear speedup, but once the run time gets sufficiently small, the speedup drops off, here we can see the results of the sequential time becoming the dominant portion of the running time.

In the graph for speedup and decrease run time while factoring 221 (8 bits) we see super linear speedup, which is as good as we can possibly hope for.

In the graph of speedup while factoring 391 (9 bits), we see that the good behavior while factoring 221 continues. At this point the exponential slowdown made it difficult to continue gathering speedup data for larger numbers.

6 Conclusion

Through the principle of superposition in quantum systems we can create useful memory components that are on the scale of an atom or smaller. These quantum memory registers may facilitate exponential computational speed increases by taking advantage of quantum parallelism.

Peter Shor has developed an algorithm which makes factoring large numbers tractable, and in doing so has drawn great attention to the field of quantum computing. Due to Shor's algorithm, we may someday have to turn to other means of encrypting data than currently employed.

7 Bibliography

Barenco, Ekert, Sanpera and Machiavello. "Un saut d'echelle pour les calculateurs," *La Recherche*, November 1996.

Benioff, p. "The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines," *Journal of Statistical Physics*, Vol. 22 (1980), pp. 563-591.

Berthiaume, Andre and Brassard, Gilles. "The quantum Challenge to Complexity Theory," *Proceedings of the 7th IEEE Conference on Structure in Complexity Theory* (1992), pp. 132-137.

Brassard, Gilles. "Searching a Quantum Phone Book," *Science*, 31 January 1997.

Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. "Introduction to Algorithms," St. Louis: McGraw-Hill, 1994.

Deutsch, David and Jozsa, Richard. "Rapid Solution of Problems by Quantum Computation," *Proceedings Royal Society London*, Vol. 439A (1992), pp. 553-558.

Feynman, Richard. "Simulating Physics with Computers," *Optics News* Vol. 11 (1982), pp. 467-488.

Grover, L. K. "A Fast Quantum Mechanical Algorithm for Database Search," *Proceedings of the 28'th Annual ACM Symposium on the Theory of Computing* (1996), pp. 212-219.

Shor, Peter. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124-134.

Steane, Andrew. "Quantum Computing," *Reports on Progress in Physics*, vol 61 (1998), pp 117-173.

Williams, Colin P. and Clearwater, Scott H. "Explorations in Quantum Computing," New York: Springer-Verlag, 1998.

8 Glossary

9 Glossary

This is a glossary of terms and variables used throughout this paper.

λ : In the context of Shor's algorithm in integer such that $m = \frac{\lambda q}{r}$.

a : A variable that when used in the context of Shor's algorithm is an argument to the function $\mathcal{F}(a) = x^a \bmod n$. It may be a single integer, or it may denote a superposition of states.

Classical computer : A computer whose internal workings behave in manner consistent with classical physics. Data registers in a classical computer can not exist in a superposition of states. By definition (e.g. the Church-Turing Thesis) a classical computer can only compute functions that a Turing machine can compute.

Classical physics : The model that was used to describe physical phenomenon before the advent of quantum physics. The predictions of classical physics with regard to the behavior of fundamental particles are incorrect.

Collapse : A collapse in the context of this paper is what happens to the state vector of a quantum mechanical system when that system is observed or measured. Since the system can only be measured to be in one of its base states, the state vector will collapse from some superposition of base states into the measured state only.

Complex vector space : A vector space in which the coordinates of a vector are complex numbers.

Complexity class : A grouping of algorithms based on how their memory usage and number of operations scale with the size of the input.

Coprime : Integers a and b are coprime if their greatest common denominator is one.

Discrete Fourier Transform : A transformation converts a finite list of equally spaced samples of a function into a list of coefficients of finite combinations of circles, ordered by their frequencies, that have the same values. In

Shor's algorithm it is used to calculate and multiple of the inverse period, where the period is the quantity which enables Shor's algorithm to find factors of a number n.

gcd : This is an abbreviation for the mathematical function which calculates the greatest common denominator of two integers. The greatest common denominator of two integers a and b is the largest integer c such that a/c and b/c are integers.

Hilbert Space : A complex linear vector space. The complete state of a n state quantum mechanical system can be represented by a vector in an n dimensional Hilbert Space.

Linear vector space : A vector space such that vectors within the space which are added or multiplied together result in vectors that also lie within the same space.

Memory register : A array of bits on a classical computer. A memory register of size n may store one of 2^n values.

n : In the context of Shor's algorithm, a number to be factored.

Periodic function : A function with a period r such that $\mathcal{F}(x) = \mathcal{F}(x+r) = \mathcal{F}(x+2r)$ and so on. Sine and Cosine are typical examples of periodic functions.

q : In the context of Shor's algorithm the power of 2 such that $n^2 \leq q < 2n^2$.

Quantum memory register : A array of n qubits which can exist in any superposition of its 2^n base states.

Quantum parallelism : The ability of a quantum computer to perform an operation on a quantum memory register which results in the simultaneous calculation of a function function of 2^n different values where n is the size of the quantum memory register.

Quantum physics : Currently the most complete model for describing the behavior of small physical systems.

Qubit : A two state quantum mechanical system, which can exist in any superposition of the 0 and 1 state. In this paper I have considered a spin-1/2 particle as a possible candidate for a qubit in a physical implementation of a qubit.

r : In the context of Shor's algorithm the period of the periodic function x^a mod n.

Shor's Algorithm : A algorithm designed by Peter Shor of Bell Labs which finds factors of a number n in polynomial time relative to the number of bits in n's binary representation on a quantum computer. The fastest published algorithm on a classical computer is slower than polynomial time, and the presumed intractability of this problem is the basis for many cryptographic systems.

Spin-1/2 particle : A particle which can be characterized as having a spin of +1/2 or -1/2. Examples include the proton, neutron, and electron.

State vector : In the context of this paper, the state vector in a Hilbert Space which completely describes a quantum mechanical state vector - such as the state of quantum memory register.

Superposition of states : A mixture of base states. The state vector for a quantum mechanical systems which can be measured in one of n base states can exist as any combination of components of the base states.

x : In the context of Shor's algorithm a integer which is coprime to n and used in the function $\mathcal{F}(a) = x^a \bmod n$.

A Mathematics Used in this Paper

A.1 Complex Numbers

A complex number is a number of the form $a + i * b$, where a and b are real numbers, and i is defined to be the square root of negative one. Addition of two complex numbers c_1 and c_2 is defined to be:

$$c_1 = a_1 + i * b_1$$

$$c_2 = a_2 + i * b_2$$

$$c_1 + c_2 = a_1 + a_2 + i * (b_1 + b_2)$$

The complex conjugate of a complex number c, denoted c^* is defined to be:

$$c = a + i * b$$

$$c^* = a - i * b$$

Multiplication of two complex numbers c_1 and c_2 is defined to be:

$$c_1 = a_1 + i * b_1$$

$$c_2 = a_2 + i * b_2$$

$$c_1 * c_2 = a_1 * a_2 - b_1 * b_2 + i * (a_1 * b_2 + a_2 * b_1)$$

Euler's Formula for complex numbers states that $e^{ix} = \cos x + i * \sin x$, this relationship is used in the discrete Fourier transform of Shor's algorithm.

A.2 Vector Mathematics

The only operations that are used in Shor's algorithm on vectors are addition, length determination, and scaling. The vector in question is that state vector of a quantum mechanical system, it is a complex vector in a Hilbert Space.

For example, a n state quantum system requires a n dimensional Hilbert Space to represent its state vector. The quantum system can be measured in any of the n states, and to represent this we imagine each of the n states as mutually perpendicular axes within a Hilbert space. Thus the state vector for a system in the j'th state is equal to:

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

For the n states, where the number at the top of the column is the length of the state vector projected onto the 1st state, and the 1 appears in the j'th row.

To add two vectors we simply add their components.

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}$$

Since this vector lies in a Hilbert Space the projections of the state vector onto the coordinate axes are allowed to be complex numbers, thus the definition of length is slightly different from what is expected.

The length of a vector in a Hilbert space with n components is defined to be: $\sqrt{\sum_{j=1}^{n} |w_j|^2}$ where w_j is the value of the j'th component of the vector, and $|w_j|^2$ is defined to be w_j times its complex conjugate, or when $w_j = a + i * b$, $|w_j|^2 = a^2 + b^2$. To scale a vector by any length l you simply multiply each component of the vector by the value l. In particular to scare a vector to length 1 you multiply each component by the inverse length of the vector.

B Code for the Simulation of Shor's Algorithm

B.1 complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex {
public:
```

```
  Complex();                  //Default constructor
  Complex( double, double );  // Argument constructor
  virtual ~Complex();          //Default destructor.
  void Set(double new_real, double new_imaginary); //Set data members.
  double Real() const;              //Return the real part.
  double Imaginary() const;         //Return the imaginary part.
  Complex & operator+(const Complex&); //Overloaded + operator
  Complex & operator*(const Complex&); //Overloaded * operator
  Complex & operator=(const Complex&); //Overloaded = operator
  bool operator==(const Complex&) const;    //Overloaded == operator
private:
  double real;
  double imaginary;
};

#endif
```

B.2 complex.C

```
#include <math.h>
#include "complex.h"

//Complex constructor, initializes to 0 + i0.
Complex::Complex(): real( 0 ), imaginary( 0 ) {}

//Argument constructor.
Complex::Complex( double r, double i ): real( r ), imaginary( i ) {}

//Complex destructor.
Complex::~Complex() {}

//Overloaded = operator.
Complex & Complex::operator=(const Complex &c) {
  if (&c != this) {
    real = c.Real();
    imaginary = c.Imaginary();
    return *this;
  }
}

//Overloaded + operator.
Complex & Complex::operator+(const Complex &c) {
  real += c.Real();
  imaginary += c.Imaginary();
  return *this;
}
```

```
//Overloaded * operator.
Complex & Complex::operator*(const Complex &c) {
  real = real * c.Real() - imaginary * c.Imaginary();
  imaginary = real * c.Imaginary() + imaginary * c.Real();
  return *this;
}

//Overloaded == operator.  Small error tolerances.
bool Complex::operator==(const Complex &c) const {
  //This is to take care of round off errors.
  if (fabs(c.Real() - real) > pow(10,-14)) {
    return false;
  }
  if (fabs(c.Imaginary()- imaginary) > pow(10,-14)) {
    return false;
  }
  return true;
}

//Sets private data members.
void Complex::Set(double new_real, double new_imaginary) {
  real = new_real;
  imaginary = new_imaginary;
  return;
}

//Returns the real part of the complex number.
double Complex::Real() const {
  return real;
}

//Returns the imaginary part of the complex number.
double Complex::Imaginary() const {
  return imaginary;
}
```

B.3 qureg.h

```
#include "complex.h"

#ifndef QUREG_H
#define QUREG_H

using namespace std;
```

```
class QuReg {
public:
  //Default constructor.  Size is the size in bits of our register.
  //In our implementation of Shor's algorithm we will need size bits
  //to represent our value for "q" which is a number we have chosen
  //with small prime factors which is between 2n^2 and 3n^2 inclusive
  //where n is the number we are trying to factor.  We envision our
  //the description of our register of size "S" as 2^S complex
  //numbers, representing the probability of finding the register on
  //one of or 2^S base states.  Thus we use an array of size 2^S, of
  //Complex numbers.  Thus if the size of our register is 3 bits
  //array[7] is the probability amplitude of the state |1,1,1>, and
  //array[7] * Complex Conjugate(array[7]) = probability of choosing
  //that state.  We use normalized state vectors thought the
  //simulation, thus the sum of all possible states times their
  //complex conjugates is = 1.
  QuReg(unsigned long long int size);

  QuReg(); //Default Constructor

  QuReg(const QuReg &); //Copy constructor

  virtual ~QuReg(); //Default destructor.

  //Measures our quantum register, and returns the decimal
  //interpretation of the bit string measured.
  unsigned long long int DecMeasure();

  //Dumps all the information about the quantum register.  This
  //function would not be possible for an actual quantum register, it
  //is only there for debugging.  When verbose != 0 we return every
  //value, when verbose = 0 we return only probability amplitudes
  //which differ from 0.
  void Dump(int verbose) const;

  //Sets state of the qubits using the arrays of complex amplitudes.
  void SetState(Complex *new_state);

  //Sets the state to an equal superposition of all possible states
  //between 0 and number inclusive.
  void SetAverage(unsigned long long int number);

  //Normalize the state amplitudes.
  void Norm();

  //Get the probability of a given state.  This is used in the
```

```
  //discrete Fourier transformation.  In a real quantum computer such
  //an operation would not be possible, on the flip side, it would
  //also not be necessary as you could simply build a DFT gate, and
  //run your superposition through it to get the right answer.
  Complex GetProb(unsigned long long int state) const;

  //Return the size of the register.
  int Size() const;

private:
  unsigned long long int reg_size;
  Complex *State;
};

#endif
```

B.4 qureg.C

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "complex.h"
#include "qureg.h"

using namespace std;

QuReg::QuReg(): reg_size( 0 ), State( NULL ) {}

//Constructor.
QuReg::QuReg(unsigned long long int size) {
  reg_size = size;
  State = new Complex[(unsigned long long int)pow(2, reg_size)];
  srand(time(NULL));
}

//Copy Constructor
QuReg::QuReg(const QuReg & old) {
  reg_size = old.reg_size;
  unsigned long long int reg_length = (unsigned long long int) pow(2, reg_size);
  State = new Complex[reg_length];
  for (unsigned int i = 0 ; i < reg_length ; i++) {
    State[i] = old.State[i];
  }
}
```

```
//Destructor.
QuReg::~QuReg() {
  if ( State ) {
    delete [] State;
  }
}

//Return the probability amplitude of the state'th state.
Complex QuReg::GetProb(unsigned long long int state) const {
  if (state >= pow(2, reg_size)) {
    cout << "Invalid state index " << state << " requested!"
 << endl << flush;
    throw -1;
  } else {
    return(State[state]);
  }
}

//Normalize the probability amplitude, this ensures that the sum of
//the sum of the squares of all the real and imaginary components is
//equal to one.
void QuReg::Norm() {
  double b = 0;
  double f, g;
  for (unsigned long long int i = 0; i < pow(2, reg_size) ; i++) {
    b += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
  }
  b = pow(b, -.5);
  for (unsigned long long int i = 0; i < pow(2, reg_size) ; i++) {
    f = State[i].Real() * b;
    g = State[i].Imaginary() * b;
    State[i].Set(f, g);
  }
}

//Returns the size of the register.
int QuReg::Size() const {
  return reg_size;
}

//Measure a state, and return the decimal value measured.  Collapse
//the state so that the probability of measuring the measured value in
//the future is 1, and the probability of measuring any other state is
//0.
unsigned long long int QuReg::DecMeasure() {
  int done = 0;
```

```
  int DecVal = -1; //-1 is an error, we did not measure anything.
  double rand1, a, b;
  rand1 = rand()/(double)RAND_MAX;
  a = b = 0;
  for (unsigned long long int i = 0 ; i < pow(2, reg_size)  ;i++) {
    if (!done ){
      b += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
      if (b > rand1 && rand1 > a) {
//We have just measured the i state.
for (unsigned long long int j = 0; j < pow(2, reg_size) ; j++) {
  State[j].Set(0,0);
}
State[i].Set(1,0);
DecVal = i;
done = 1;
      }
      a += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
    }
  }
  return DecVal;
}

//For debugging, output information about the register.
void QuReg::Dump(int verbose) const {
  for (unsigned long long int i = 0 ; i < pow(2, reg_size) ; i++) {
    if (verbose || fabs(State[i].Real()) > pow(10,-14)
|| fabs(State[i].Imaginary()) > pow(10,-14)) {
      cout << "State " << i << " has probability amplitude "
   << State[i].Real() << " + i" << State[i].Imaginary()
   << endl << flush;
    }
  }
}

//Set the states to those given in the new_state array.
void QuReg::SetState(Complex *new_state) {
  //Beware, this function will cause segfaults if new_state is too
  //small.
  for (unsigned long long int i = 0 ; i < pow(2, reg_size) ; i++) {
    State[i].Set(new_state[i].Real(), new_state[i].Imaginary());
  }
}

//Set the State to an equal superposition of the integers 0 -> number
//- 1
void QuReg::SetAverage(unsigned long long int number) {
```

```
  if (number >= pow(2, reg_size)) {
    cout << "Error, initializing past end of array in qureg::SetAverage.\n";
  } else {
    double prob = pow(number, -.5);
    for (unsigned long long int i = 0 ; i <= number ; i++) {
      State[i].Set(prob, 0);
    }
  }
}
```

B.5 util.h

```
#include "qureg.h"

#ifndef UTIL_H
#define UTIL_H

//This function takes an integer input and returns 1 if it is a prime
//number, and 0 otherwise.
unsigned long long int TestPrime(unsigned long long int n);

//This function takes an integer input and returns 1 if it is equal to
//a prime number raised to an integer power, and 0 otherwise.
unsigned long long int TestPrimePower(unsigned long long int n);

//This function computes the greatest common denominator of two integers.
//Since the modulus of a number mod 0 is not defined, we return a -1 as
//an error code if we ever would try to take the modulus of something and
//zero.
unsigned long long int GCD(unsigned long long int a, unsigned long long int b);

//This function takes and integer argument, and returns the size in bits
//needed to represent that integer.
unsigned long long int RegSize(unsigned long long int a);

//q is the power of two such that n^2 <= q < 2n^2.
unsigned long long int GetQ(unsigned long long int n);

//This function takes three integers, x, a, and n, and returns x^a mod
//n.  This algorithm is known as the "Russian peasant method," I
//believe, and avoids overflow by never calculating x^a directly.
unsigned long long int modexp(unsigned long long int x, unsigned long long int a, unsigned long long int n);

// This function finds the denominator q of the best rational
// denominator q for approximating p / q for c with q < qmax.
unsigned long long int denominator(double c, unsigned long long int qmax);
```

```
//This function takes two integer arguments and returns the greater of
//the two.
unsigned long long int max(unsigned long long int a, unsigned long long int b);

//This function computes the discrete Fourier transformation on a register's
//0 -> q - 1 entries.
void DFT(QuReg * reg, unsigned long long int q, unsigned int thread_id);

#endif
```

B.6 util.C

```
#include <iostream>
#include <math.h>
#include "qureg.h"
#include "range.h"
#include "barrier.h"

//This function takes an integer input and returns 1 if it is a prime
//number, and 0 otherwise.
unsigned long long int TestPrime(unsigned long long int n) {
  unsigned long long int i;
  for (i = 2 ; i <= floor(sqrt(n)) ; i++) {
    if (n % i == 0) {
      return(0);
    }
  }
  return(1);
}

//This function takes an integer input and returns 1 if it is equal to a
//prime number raised to an integer power, and 0 otherwise.
unsigned long long int TestPrimePower(unsigned long long int n) {
  unsigned long long int i,j;
  j = 0;
  i = 2;
  while ((i <= floor(pow(n, .5))) && (j == 0)) {
    if((n % i) == 0) {
      j = i;
    }
    i++;
  }
  for (unsigned long long int i = 2 ; i <= (floor(log(n) / log(j)) + 1) ; i++) {
    if(pow(j , i) == n) {
      return(1);
```

```
    }
  }
  return(0);
}

//This function computes the greatest common denominator of two integers.
//Since the modulus of a number mod 0 is not defined, we return a -1 as
//an error code if we ever would try to take the modulus of something and
//zero.
unsigned long long int GCD(unsigned long long int a, unsigned long long int b) {
  unsigned long long int d;
  if (b != 0) {
    while (a % b != 0) {
      d = a % b;
      a = b;
      b = d;
    }
  } else {
    return -1;
  }
  return(b);
}

//This function takes and integer argument, and returns the size in bits
//needed to represent that integer.
unsigned long long int RegSize(unsigned long long int a) {
  unsigned long long int size = 0;
  while(a != 0) {
    a = a>>1;
    size++;
  }
  return(size);
}


//q is the power of two such that n^2 <= q < 2n^2.
unsigned long long int GetQ(unsigned long long int n) {
  unsigned long long int power = 8; //265 is the smallest q ever is.
  while (pow(2,power) < pow(n,2)) {
    power = power + 1;
  }
  return((int)pow(2,power));
}

//This function takes three integers, x, a, and n, and returns x^a mod n.
//This algorithm is known as the "Russian peasant method," I believe.
```

```
unsigned long long int modexp(unsigned long long int x, unsigned long long int a, unsigned long long int n) {
  unsigned long long int value = 1;
  unsigned long long int tmp;
  tmp = x % n;
  while (a > 0) {
    if (a & 1) {
      value = (value * tmp) % n;
    }
    tmp = tmp * tmp % n;
    a = a>>1;
  }
  return value;
}


// This function finds the denominator q of the best rational
// denominator q for approximating p / q for c with q < qmax.
unsigned long long int denominator(double c, unsigned long long int qmax) {
  double y = c;
  double z;
  unsigned long long int q0 = 0;
  unsigned long long int q1 = 1;
  unsigned long long int q2 = 0;
  while (1) {
    z = y - floor(y);
    if (z < 0.5 / pow(qmax,2)) {
      return(q1);
    }
    if (z != 0) {
      //Can't divide by 0.
      y = 1 / z;
    } else {
      //Warning this is broken if q1 == 0, but that should never happen.
      return(q1);
    }
    q2 = (int)floor(y) * q1 + q0;
    if (q2 >= qmax) {
      return(q1);
    }
    q0 = q1;
    q1 = q2;
  }
}

//This function takes two integer arguments and returns the greater of
//the two.
```

```
unsigned long long int max(unsigned long long int a, unsigned long long int b) {
  if (a > b) {
    return(a);
  }
  return(b);
}

//BEGIN PARALLEL This is where we need parallelism the most.

Complex * init = NULL;
unsigned long long int count = 0;

void DFT(QuReg * reg, unsigned long long int q, unsigned int thread_id) {
  //The Fourier transform maps functions in the time domain to
  //functions in the frequency domain.  Frequency is 1/period, thus
  //this Fourier transform will take our periodic register, and peak it
  //at multiples of the inverse period.  Our Fourier transformation on
  //the state a takes it to the state: q^(-.5) * Sum[c = 0 -> c = q - 1,
  //c * e^(2*Pi*i*a*c / q)].  Remember, e^ix = cos x + i*sin x.

  if (thread_id == 0) {
    if (init) {
      delete [] init;
    }
    init = new Complex[q];
  }

  // Wait for other threads.
  barrier(&global_barrier_counter, num_threads, &global_barrier_mutex,
  &global_barrier_cond);

  Complex tmpcomp( 0, 0 );

  //Here we do things that a real quantum computer couldn't do, such
  //as look as individual values without collapsing state.  The good
  //news is that in a real quantum computer you could build a gate
  //which would what this out all in one step.

  for (unsigned long long int a = 0 ; a < q ; a++) {
    //This if statement helps prevent previous round off errors from
    //propagating further.
    if ((pow(reg->GetProb(a).Real(),2) +
 pow(reg->GetProb(a).Imaginary(),2)) > pow(10,-14)) {
      // Limit our attention to the potion of the overall range our
      // thread is responsible for.
      for (unsigned long long int c = q_range_lower[thread_id] ; c <= q_range_upper[thread_id] ; c++) {
```

```
tmpcomp.Set(pow(q,-.5) * cos(2*M_PI*a*c/(double)q),
    pow(q,-.5) * sin(2*M_PI*a*c/(double)q));
init[c] = init[c] + (reg->GetProb(a) * tmpcomp);
      }
    }

    // Log how thread_id 0 is doing.
    if (thread_id == 0) {
      count++;
      if (count == 1000) {
cout << "Making progress in Fourier transform, "
     << 100*((double)a / (double)(q - 1)) << "% done!"
     << endl << flush;
count = 0;
      }
    }
  }

  // Wait for the other threads to get here.
  barrier(&global_barrier_counter, num_threads, &global_barrier_mutex,
  &global_barrier_cond);

  if (thread_id == 0) {
    reg->SetState(init);
    reg->Norm();
  }
}

//END PARALLEL
```

B.7 timer.C

```
#include <iostream>
#include <sys/time.h>

using namespace std;

double  get_clock() {
   struct timeval tv; int ok;
   ok = gettimeofday(&tv, NULL);
   if (ok<0) { cout << "gettimeofday error" << endl;  }
   return (tv.tv_sec * 1.0 + tv.tv_usec * 1.0E-6);
}
```

B.8 barrier.h

```
#include <pthread.h>

#ifndef BARRIER_H
#define BARRIER_H

extern int num_threads;

void barrier(int* barrier_counter, int total_threads,
     pthread_mutex_t* barrier_mutex, pthread_cond_t* barrier_cond);

void final_barrier(int* barrier_counter, int total_threads,
   pthread_mutex_t* barrier_mutex, pthread_cond_t* barrier_cond);

extern pthread_mutex_t global_barrier_mutex;
extern pthread_cond_t global_barrier_cond;
extern int global_barrier_counter;

extern pthread_mutex_t global_barrier_mutex1;
extern pthread_cond_t global_barrier_cond1;
extern int global_barrier_counter1;

extern pthread_mutex_t global_barrier_mutex2;
extern pthread_cond_t global_barrier_cond2;
extern int global_barrier_counter2;

extern pthread_mutex_t global_barrier_mutex3;
extern pthread_cond_t global_barrier_cond3;
extern int global_barrier_counter3;

extern pthread_mutex_t global_barrier_mutex4;
extern pthread_cond_t global_barrier_cond4;
extern int global_barrier_counter4;

extern pthread_mutex_t global_barrier_mutex5;
extern pthread_cond_t global_barrier_cond5;
extern int global_barrier_counter5;

extern pthread_mutex_t global_barrier_mutex6;
extern pthread_cond_t global_barrier_cond6;
extern int global_barrier_counter6;

extern pthread_mutex_t global_barrier_mutex7;
extern pthread_cond_t global_barrier_cond7;
extern int global_barrier_counter7;
```

```
extern pthread_mutex_t global_barrier_mutex8;
extern pthread_cond_t global_barrier_cond8;
extern int global_barrier_counter8;

extern pthread_mutex_t final_barrier_mutex;
extern pthread_cond_t final_barrier_cond;
extern int final_barrier_counter;

#endif
```

B.9 barrier.C

```
#include <pthread.h>

#include "barrier.h"

int num_threads;

pthread_mutex_t global_barrier_mutex;
pthread_cond_t global_barrier_cond;
int global_barrier_counter = 0;

pthread_mutex_t global_barrier_mutex1;
pthread_cond_t global_barrier_cond1;
int global_barrier_counter1 = 0;

pthread_mutex_t global_barrier_mutex2;
pthread_cond_t global_barrier_cond2;
int global_barrier_counter2 = 0;

pthread_mutex_t global_barrier_mutex3;
pthread_cond_t global_barrier_cond3;
int global_barrier_counter3 = 0;

pthread_mutex_t global_barrier_mutex4;
pthread_cond_t global_barrier_cond4;
int global_barrier_counter4 = 0;

pthread_mutex_t global_barrier_mutex5;
pthread_cond_t global_barrier_cond5;
int global_barrier_counter5 = 0;

pthread_mutex_t global_barrier_mutex6;
pthread_cond_t global_barrier_cond6;
int global_barrier_counter6 = 0;
```

```
pthread_mutex_t global_barrier_mutex7;
pthread_cond_t global_barrier_cond7;
int global_barrier_counter7 = 0;

pthread_mutex_t global_barrier_mutex8;
pthread_cond_t global_barrier_cond8;
int global_barrier_counter8 = 0;

pthread_mutex_t final_barrier_mutex;
pthread_cond_t final_barrier_cond;
int final_barrier_counter = 0;

void barrier(int* barrier_counter, int total_threads,
     pthread_mutex_t* barrier_mutex, pthread_cond_t* barrier_cond) {
  int reuseCount;
  pthread_mutex_lock(barrier_mutex);
  (*barrier_counter)++;
  reuseCount = (*barrier_counter)/total_threads;

  if(*barrier_counter%total_threads == 0) {
    pthread_cond_broadcast(barrier_cond);
  } else {
    while (reuseCount == (*barrier_counter)/total_threads)
      pthread_cond_wait(barrier_cond, barrier_mutex);
  }
  pthread_mutex_unlock(barrier_mutex);
}

void final_barrier(int* barrier_counter, int total_threads,
     pthread_mutex_t* barrier_mutex, pthread_cond_t* barrier_cond) {
  int reuseCount;
  pthread_mutex_lock(barrier_mutex);
  (*barrier_counter)++;
  reuseCount = (*barrier_counter)/total_threads;

  if(*barrier_counter%total_threads == 0) {
    pthread_cond_broadcast(barrier_cond);
  } else {
    while (reuseCount == (*barrier_counter)/total_threads)
      pthread_cond_wait(barrier_cond, barrier_mutex);
  }
  pthread_mutex_unlock(barrier_mutex);
}
```

B.10 range.h

```
#ifndef RANGE_H
#define RANGE_H

//The i'th element of q_range_lower is the lower limit that the i'th
//process element should process, and q_range_upper's i'th element is
//the upper limit on what the i'th thread should process.
//The can be used in for loops like this:
//for(int i = q_range_lower[proc_rank] ; i <= q_range_upper[proc_rank] ; i++)
//Please note the <= in the boolean portion.

extern int * q_range_lower;
extern int * q_range_upper;

//Just like q_range for n, the number to be factored.
extern int * n_range_lower;
extern int * n_range_upper;

void init_ranges(int num_threads, int num_factor, int q);

#endif
```

B.11 range.C

```
#include "range.h"

int * q_range_lower;
int * q_range_upper;

//Just like q_range for n, the number to be factored.
int * n_range_lower;
int * n_range_upper;

void init_ranges(int num_threads, int num_factor, int q) {
  for (int i = 0 ; i < num_threads ; i++) {
    q_range_lower[i] = i*(q/num_threads);
    q_range_upper[i] = (i+1)*(q/num_threads) - 1;
    if (i == num_threads - 1) {
      q_range_upper[i] += q%num_threads;
    }
  }

  for (int i = 0 ; i < num_threads ; i++) {
    n_range_lower[i] = i*(num_factor/num_threads);
    n_range_upper[i] = (i+1)*(num_factor/num_threads) - 1;
```

```
    if (i == num_threads - 1) {
      n_range_upper[i] += num_factor%num_threads;
    }
  }
}
```

B.12 shor.C

```
#include <iostream>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#include "complex.h"
#include "util.h"
#include "timer.C"
#include "range.h"
#include "barrier.h"

using namespace std;

unsigned long long int num_fact;
pthread_t * thread_array;

void *Shor_Sim(void *my_thread_id);

int main(int argc, char * argv[]) {
  double start_time, end_time;

  if(argc <= 2) {
    cout << "Usage: " << argv[0] << " <num to be factored> <num threads>\n";
    exit(0);
  }

  num_fact = atoi(argv[1]);
  num_threads = atoi(argv[2]);

  //The i'th element of q_range_lower is the lower limit that the i'th
  //process element should process, and q_range_upper's i'th element is
  //the upper limit on what the i'th thread should process.
  //The can be used in for loops like this:
  //for(int i = q_range_lower[proc_rank] ; i <= q_range_upper[proc_rank] ; i++)
  //Please note the <= in the boolean portion.
```

```
  q_range_lower = new int[num_threads];
  q_range_upper = new int[num_threads];

  //Just like q_range for n, the number to be factored.
  n_range_lower = new int[num_threads];
  n_range_upper = new int[num_threads];

  pthread_mutex_init(&global_barrier_mutex, 0);
  pthread_cond_init(&global_barrier_cond, NULL);

  thread_array = (pthread_t *) malloc(sizeof(pthread_t) * num_threads);

  for (unsigned int i = 0 ; i < num_threads ; i++) {
    pthread_create(&thread_array[i], 0, Shor_Sim, (void*)(intptr_t)i);
  }

  final_barrier(&final_barrier_counter, num_threads+1, &final_barrier_mutex,
&final_barrier_cond);

  return 0;
}

//n is the number we are going to factor, get n.
unsigned long long int n;

//Now we must pick a random integer x, coprime to n.  Numbers are
//coprime when their greatest common denominator is one.  One is not
//a useful number for the algorithm.
unsigned long long int x = 0;

//Now we must figure out how big a quantum register we need for our
//input, n.  We must establish a quantum register big enough to hold
//an equal superposition of all integers 0 through q - 1 where q is
//the power of two such that n^2 <= q < 2n^2.
unsigned long long int q;

//This array will remember what values of q produced for x^q mod n.
//It is necessary to retain these values for use when we collapse
//register one after measuring register two.  In a real quantum
//computer these registers would be entangled, and thus this extra
//bookkeeping would not be needed at all.  The laws of quantum
//mechanics dictate that register one would collapse as well, and
//into a state consistent with the measured value in resister two.
unsigned long long int * modex;

//This array holds the probability amplitudes of the collapsed state
```

```
//of register one, after register two has been measured it is used
//to put register one in a state consistent with that measured in
//register two.
Complex *collapse;

//This is a temporary value.
Complex tmp;

//This is a new array of probability amplitudes for our second
//quantum register, that populated by the results of x^a mod n.
unsigned long long int array_size;

Complex *mdx;

// This is the second register.  It needs to be big enough to hold
// the superposition of numbers ranging from 0 -> n - 1.
QuReg *reg2;

//This is a temporary value.
unsigned long long int tmpval;

//This is a temporary value.
unsigned long long int value;

//c is some multiple lambda of q/r, where q is q in this program,
//and r is the period we are trying to find to factor n.  m is the
//value we measure from register one after the Fourier
//transformation.
double c,m;

//This is used to store the denominator of the fraction p / den where
//p / den is the best approximation to c with den <= q.
unsigned long long int den;

//This is used to store the numerator of the fraction p / den where
//p / den is the best approximation to c with den <= q.
unsigned long long int p;

//The integers e, a, and b are used in the end of the program when
//we attempts to calculate the factors of n given the period it
//measured.
//Factor is the factor that we find.
unsigned long long int e,a,b, factor;

//Shor's algorithm can sometimes fail, in which case you do it
//again.  The done variable is set to 0 when the algorithm has
```

```
//failed.  Only try a maximum number of tries.
unsigned int done = 0;
unsigned int tries = 0;

//START TIME HERE!
double startTime;

//Create the register.
QuReg * reg1;

void *Shor_Sim(void *my_thread_id) {
  unsigned int thread_id = (unsigned int)(intptr_t)my_thread_id;
  //  cout << "My thread is " << thread_id << endl;

  //  cout << "My number to be factored is " << num_fact << endl;

  if (0 == thread_id) {
    //Establish a random seed.
    srand(time(NULL));

    n = num_fact;

    //Test to see if n is factorable by Shor's algorithm.
    //Exit if the number is even.
    if (n%2 == 0) {
      cout << "Error, the number must be odd!" << endl << flush;
      exit(0);
    }
    //Exit if the number is prime.
    if (TestPrime(n)) {
      cout << "Error, the number must not be prime!" << endl << flush;
      exit(0);
    }
    //Prime powers are prime numbers raised to integral powers.
    //Exit if the number is a prime power.
    if (TestPrimePower(n)) {
      cout << "Error, the number must not be a prime power!" << endl << flush;
      exit(0);
    }

    //Now we must pick a random integer x, coprime to n.  Numbers are
    //coprime when their greatest common denominator is one.  One is not
    //a useful number for the algorithm.
    x = 1+ (unsigned long long int)((n-1)*(double)rand()/(double)RAND_MAX);
    while (GCD(n,x) != 1 || x == 1) {
      x = 1 + (unsigned long long int)((n-1)*(double)rand()/(double)RAND_MAX);
```

```
    }
    cout << "Found x to be " << x << "." << endl << flush;

    //Now we must figure out how big a quantum register we need for our
    //input, n.  We must establish a quantum register big enough to hold
    //an equal superposition of all integers 0 through q - 1 where q is
    //the power of two such that n^2 <= q < 2n^2.
    q = GetQ(n);
    cout << "Found q to be " << q << "." << endl << flush;

    init_ranges(num_threads, n, q);

    //Create the register.
    reg1 = new QuReg(RegSize(q) - 1);
    cout << "Made register 1 with register size = " << RegSize(q) << endl
<< flush;

    //This array will remember what values of q produced for x^q mod n.
    //It is necessary to retain these values for use when we collapse
    //register one after measuring register two.  In a real quantum
    //computer these registers would be entangled, and thus this extra
    //bookkeeping would not be needed at all.  The laws of quantum
    //mechanics dictate that register one would collapse as well, and
    //into a state consistent with the measured value in resister two.
    modex = new unsigned long long int [q];

    //This array holds the probability amplitudes of the collapsed state
    //of register one, after register two has been measured it is used
    //to put register one in a state consistent with that measured in
    //register two.
    collapse = new Complex[q];

    //This is a new array of probability amplitudes for our second
    //quantum register, that populated by the results of x^a mod n.
    array_size = (unsigned long long int) pow(2,RegSize(n));

    mdx = new Complex[array_size];

    // This is the second register.  It needs to be big enough to hold
    // the superposition of numbers ranging from 0 -> n - 1.
    reg2 = new QuReg(RegSize(n));
    cout << "Created register 2 of size " << RegSize(n) << endl << flush;

    //Shor's algorithm can sometimes fail, in which case you do it
    //again.  The done variable is set to 0 when the algorithm has
    //failed.  Only try a maximum number of tries.
```

```
    done = 0;
    tries = 0;

    //START TIME HERE!
    startTime = get_clock();

  } //Everything up to this point is pre processing done by thread 0.

  // Wait for everyone to finish.
  barrier(&global_barrier_counter1, num_threads, &global_barrier_mutex1,
  &global_barrier_cond1);

  while (!done) {

    if (0 == thread_id) {

      if (tries >= 5) {
cout << "There have been five failures, giving up." << endl << flush;
exit(0);
      }
      //Now populate register one in an even superposition of the
      //integers 0 -> q - 1.
      reg1->SetAverage(q - 1);

      //Now we preform a modular exponentiation on the superposed
      //elements of reg 1.  That is, perform x^a mod n, but exploiting
      //quantum parallelism a quantum computer could do this in one
      //step, whereas we must calculate it once for each possible
      //measurable value in register one.  We store the result in a new
      //register, reg2, which is entangled with the first register.
      //This means that when one is measured, and collapses into a base
      //state, the other register must collapse into a superposition of
      //states consistent with the measured value in the other..  The
      //size of the result modular exponentiation will be at most n, so
      //the number of bits we will need is therefore less than or equal
      //to log2 of n.  At this point we also maintain a array of what
      //each state produced when modularly exponised, this is because
      //these registers would actually be entangled in a real quantum
      //computer, this information is needed when collapsing the first
      //register later.

      //This counter variable is used to increase our probability amplitude.
      tmp.Set(1,0);
    }

    // Wait for all threads.
```

```
    barrier(&global_barrier_counter2, num_threads, &global_barrier_mutex2,
    &global_barrier_cond2);

    //This is the parallel version
    for (int i = q_range_lower[thread_id] ; i <= q_range_upper[thread_id] ; i++) {
      //We must use this version of modexp instead of c++ builtins as
      //they overflow when x^i > 2^31.
      tmpval = modexp(x,i,n);
      modex[i] = tmpval;
      mdx[tmpval] = mdx[tmpval] + tmp;
    }
    //END PARALLEL

    // Wait for other threads.
    barrier(&global_barrier_counter3, num_threads, &global_barrier_mutex3,
    &global_barrier_cond3);

    if (0 == thread_id) {
      //Set the state of register two to what we calculated it should be.
      reg2->SetState(mdx);

      //Normalize register two, so that the probability of measuring a
      //state is given by summing the squares of its probability
      //amplitude.
      reg2->Norm();

      //Now we measure reg1.
      value = reg2->DecMeasure();
    }

    // Wait for other threads.
    barrier(&global_barrier_counter4, num_threads, &global_barrier_mutex4,
    &global_barrier_cond4);

    //This is a parallelized version of this loop, which assumes the
    //code ad the top of this file has been appropriately
    //integrated.
    for (int i = q_range_lower[thread_id] ; i <= q_range_upper[thread_id] ; i++) {
      if (modex[i] == value) {
collapse[i].Set(1,0);
      } else {
collapse[i].Set(0,0);
      }
    }

    // Wait for other threads.
```

```
    barrier(&global_barrier_counter5, num_threads, &global_barrier_mutex5,
    &global_barrier_cond5);

    if (0 == thread_id) {
      //Now we set the state of register one to be consistent with what
      //we measured in state two, and normalize the probability
      //amplitudes.
      reg1->SetState(collapse);
      reg1->Norm();

      //Here we do our Fourier transformation.
      cout << "Begin Discrete Fourier Transformation!" << endl << flush;
    }

    // Wait for other threads.
    barrier(&global_barrier_counter6, num_threads, &global_barrier_mutex6,
    &global_barrier_cond6);

    DFT(reg1, q, thread_id);

    // Wait for other threads.
    barrier(&global_barrier_counter7, num_threads, &global_barrier_mutex7,
    &global_barrier_cond7);

    if (0 == thread_id) {
      //Next we measure register one, due to the Fourier transform the
      //number we measure, m will be some multiple of lambda/r, where
      //lambda is an integer and r is the desired period.
      m = reg1->DecMeasure();

      //If nothing goes wrong from here on out we are done.
      done = 1;

      //If we measured zero, we have gained no new information about the
      //period, we must try again.
      if (m == 0) {
cout << "Measured, 0 this trial a failure!" << endl << flush;
done = 0;
      }

      //The DecMeasure subroutine will return -1 as an error code, due
      //to rounding errors it will occasionally fail to measure a state.
      if (m == -1) {
cout << "We failed to measure anything, this trial a failure!"
     << "  Trying again." << endl << flush;
done = 0;
```

```
      }

      //If nothing has gone wrong, try to determine the period of our
      //function, and get factors of n.
      if (done) {
//Now c =~ lambda / r for some integer lambda.  Borrowed with
//modifications from Berhnard Ohpner.
c = (double)m  / (double)q;

//Calculate the denominator of the best rational approximation
//to c with den < q.  Since c is lambda / r for some integer
//lambda, this will provide us with our guess for r, our period.
den = denominator(c, q);

//Calculate the numerator from the denominator.
p = (int)floor(den * c + 0.5);

//Give user information.
cout << "measured " << m <<", approximation for " << c << " is "
     << p << " / " << den << endl << flush;

//The denominator is our period, and an odd period is not
//useful as a result of Shor's algorithm.  If the denominator
//times two is still less than q we can use that.
if (den % 2 == 1 && 2 * den < q ){
  cout << "Odd denominator, expanding by 2\n";
  p = 2 * p;
  den = 2 * den;
}

//Initialize helper variables.
e = a = b = factor = 0;

// Failed if odd denominator.
if (den % 2 == 1)  {
  cout <<  "Odd period found.  This trial failed."
       << "  Trying again." << endl << flush;
  done = 0;
} else {
  //Calculate candidates for possible common factors with n.
  cout <<  "possible period is " << den << endl << flush;
  e = modexp(x, den / 2, n);
  a = (e + 1) % n;
  b = (e + n - 1) % n;
  cout << x << "^" << den / 2 << " + 1 mod " << n << " = " << a
       << "," << endl
```

```
       << x << "^" << den / 2 << " - 1 mod " << n << " = " << b
       << endl << flush;
  factor = max(GCD(n,a),GCD(n,b));
}
      } //if done

      //GCD will return a -1 if it tried to calculate the GCD of two
      //numbers where at some point it tries to take the modulus of a
      //number and 0.
      if (factor == -1) {
cout << "Error, tried to calculate n mod 0 for some n.  Trying again."
     << endl << flush;
done = 0;
      }

      if ((factor == n || factor == 1) && done == 1) {
cout << "Found trivial factors 1 and " << n
     << ".  Trying again." << endl << flush;
done = 0;
      }

      //If nothing else has gone wrong, and we got a factor we are
      //finished.  Otherwise start over.
      if (factor != 0 && done == 1) {
cout << n << " = " << factor << " * " << n / factor << endl << flush;
      } else if (done == 1) {
cout << "Found factor to be 0, error.  Trying again." << endl
     << flush;
done = 0;
      }
      tries++;
    }//if 0 is thread id

    // Wait for other threads.
    barrier(&global_barrier_counter8, num_threads, &global_barrier_mutex8,
    &global_barrier_cond8);

  }//While not done

  if (0 == thread_id) {
    double endTime = get_clock();
    cout << "Runtime = " << endTime - startTime << " seconds" << endl;
  }

  final_barrier(&final_barrier_counter, num_threads+1, &final_barrier_mutex,
  &final_barrier_cond);
```

```
  pthread_exit(NULL);
}
```

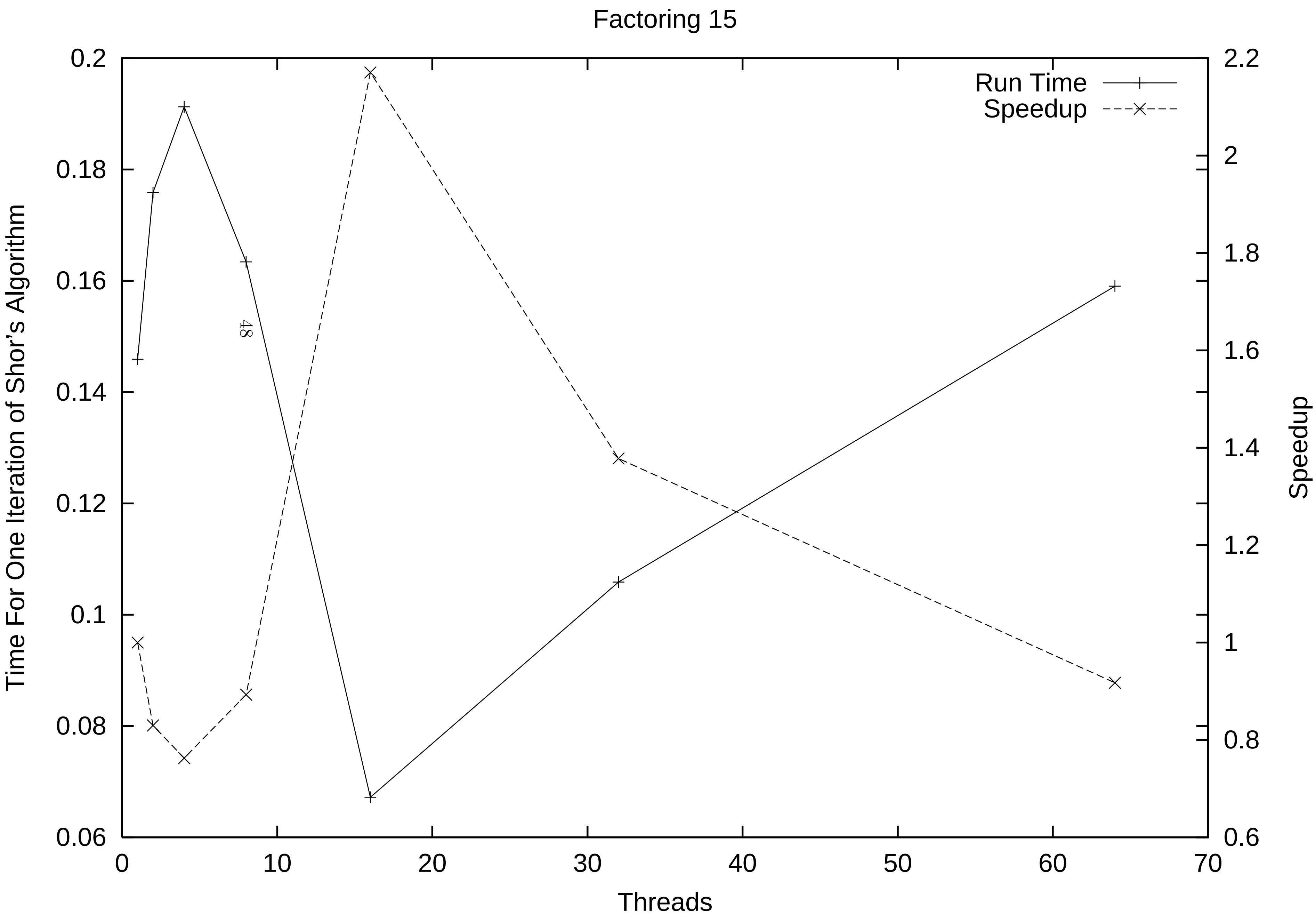

Figure 1: Run Times and Speedups for Simulation of Shor's Algorithm Factoring 15

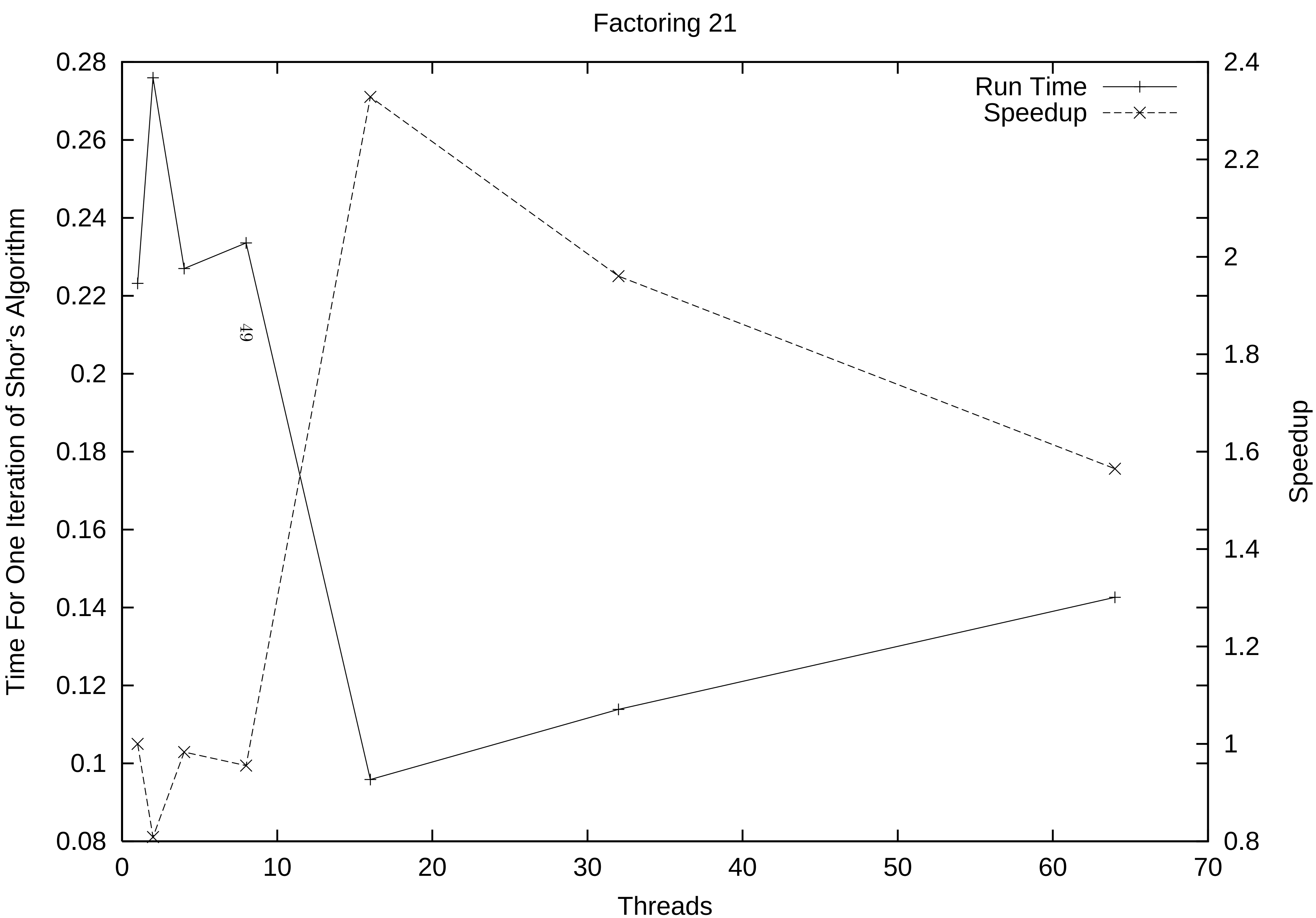

Figure 2: Run Times and Speedups for Simulation of Shor's Algorithm Factoring 21

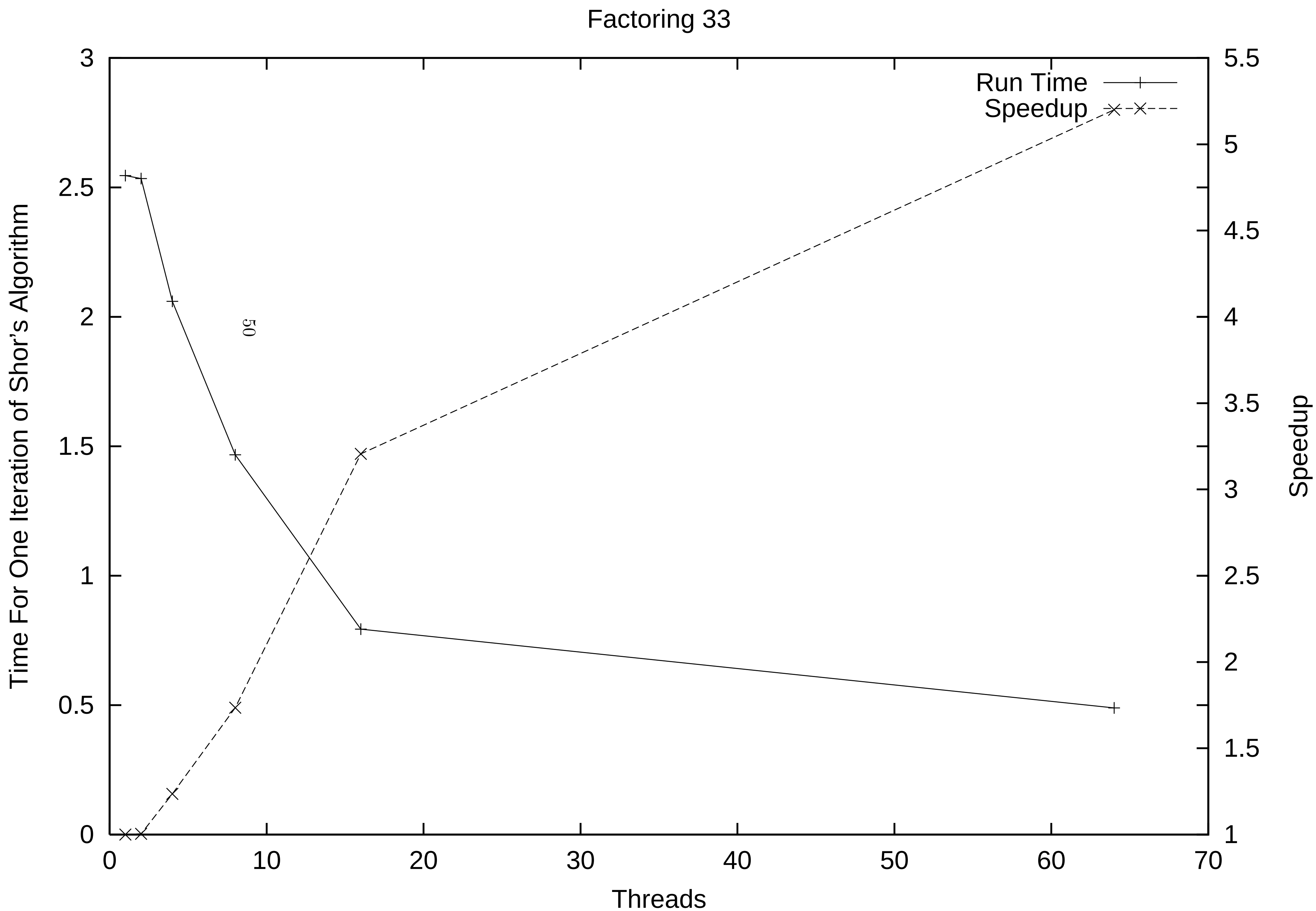

Figure 3: Run Times and Speedups for Simulation of Shor's Algorithm Factoring 33

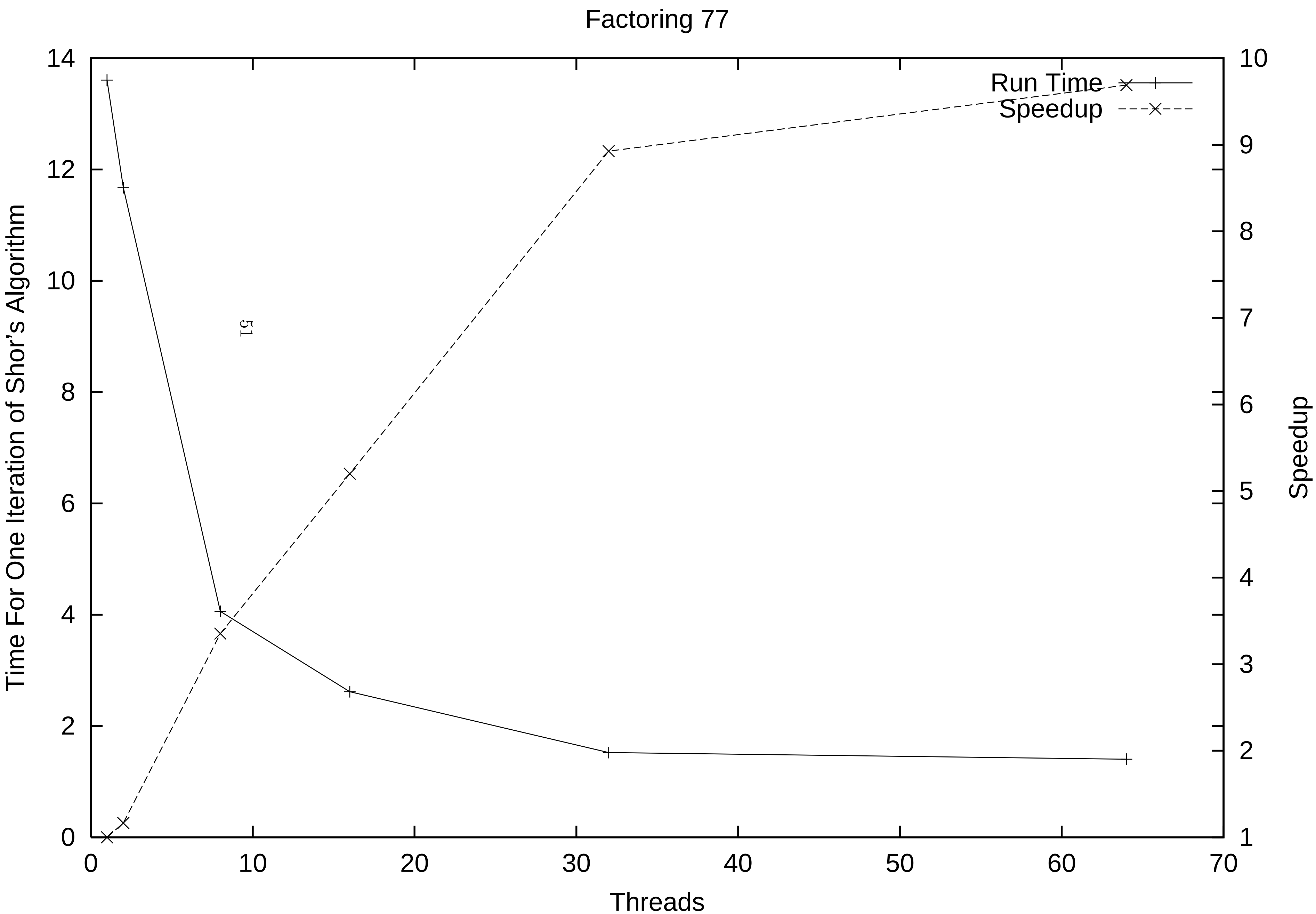

Figure 4: Run Times and Speedups for Simulation of Shor's Algorithm Factoring 77

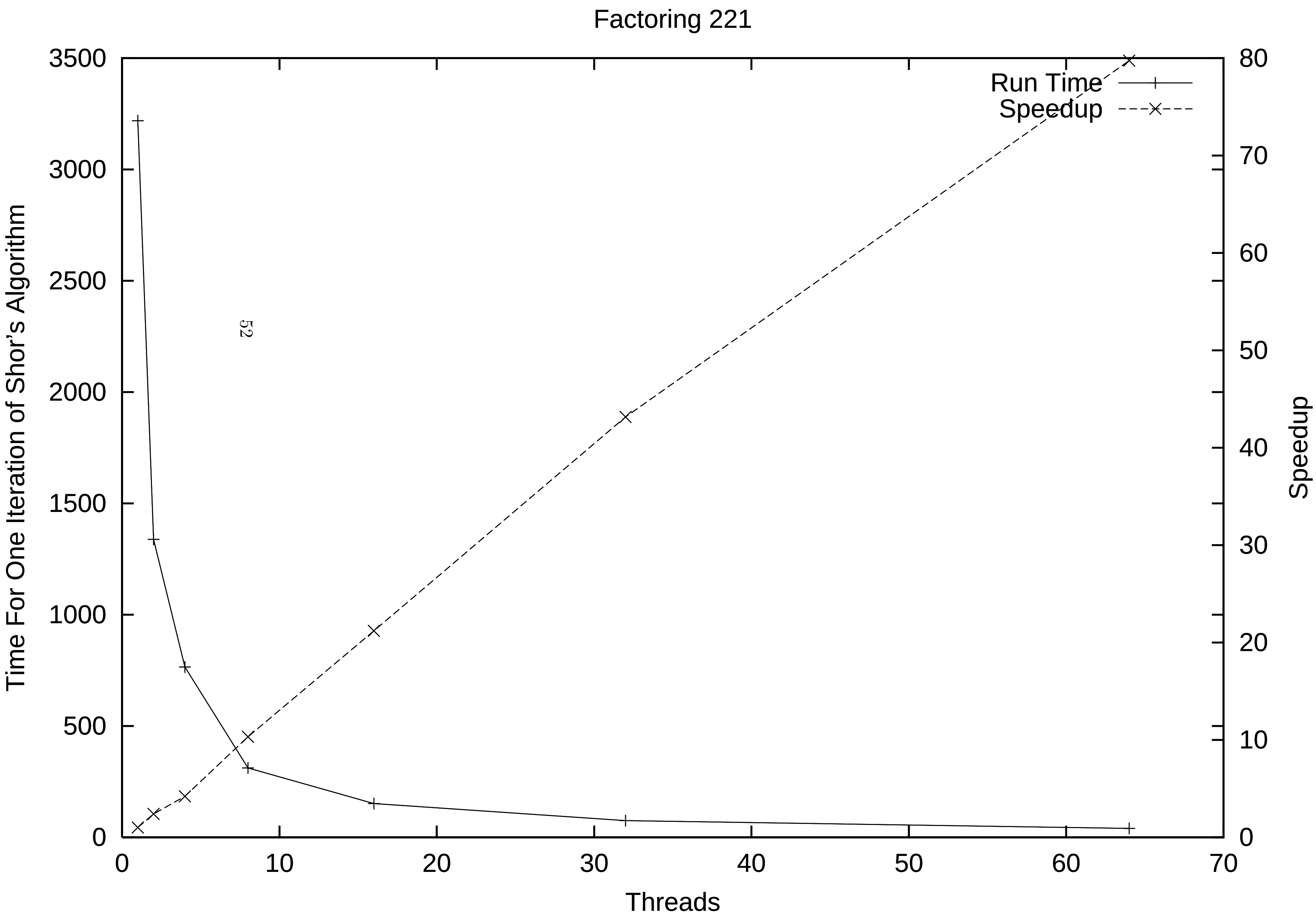

Figure 5: Run Times and Speedups for Simulation of Shor's Algorithm Factoring 221

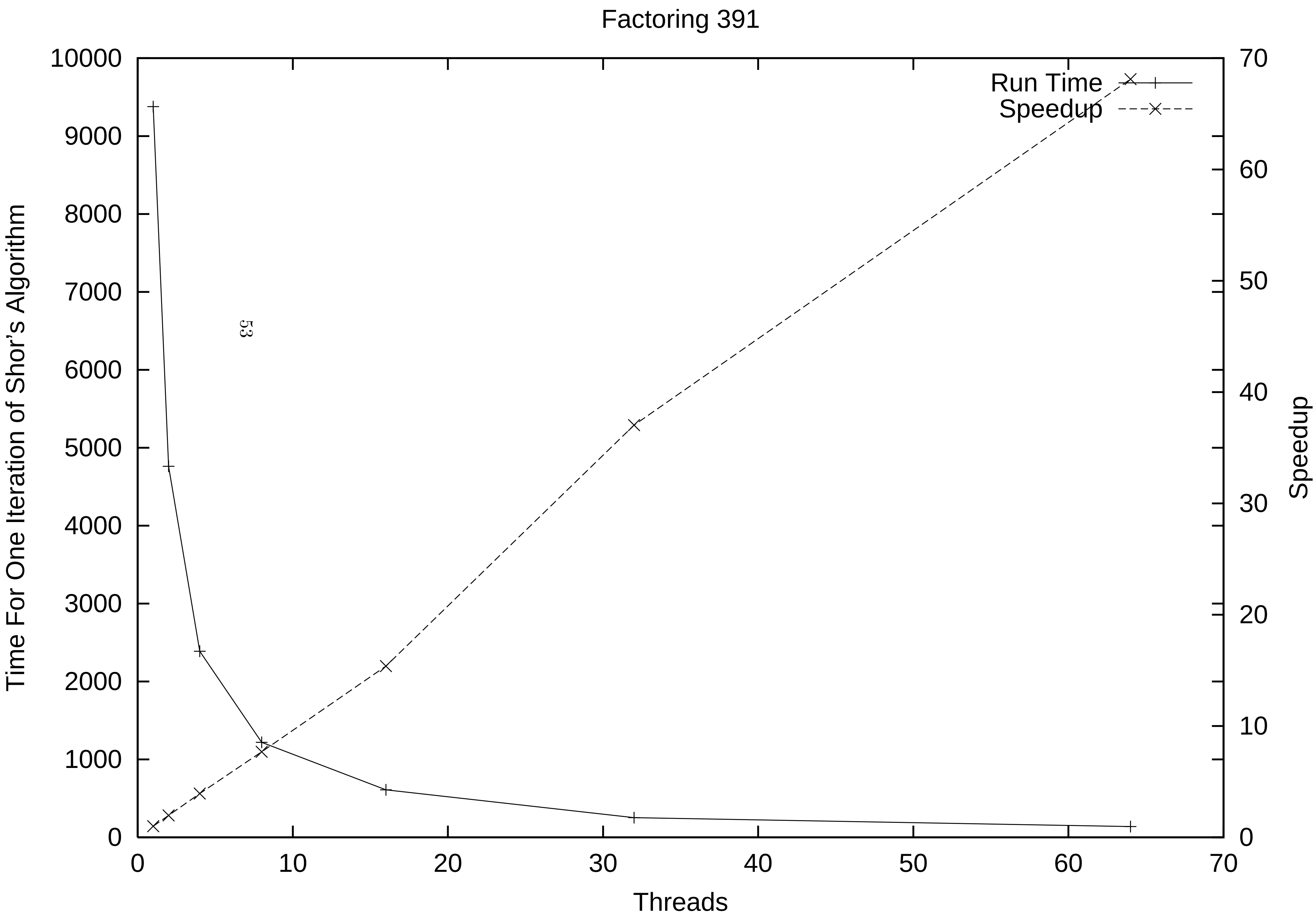

Figure 6: Run Times and Speedups for Simulation of Shor's Algorithm Factoring 391