

Quantum computing in Fortran

Arjen Markus¹
WL | Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

Quantum computing is the delicate art of using a quantum-mechanical system to perform some kind of computation. Such a system differs in a number of significant ways from ordinary computers:

- A quantum computer actually does a lot of computations *at the same time*, each solution is represented by a particular possible state. This is inherent to the physical properties of such a system.
- Unless you are sure that you have reached the final (unique) solution, you can not “look” at the “values” of the variables in the computer program that is being run: as all solutions are possible *at the same time*, observing the values will influence the system and put it into one particular, but otherwise arbitrary, state.
- All computations must be reversible, that is, a statement like “ $a = 5$ ” can only appear in the initialisation phase of a program, because the previous value will be lost. On the other hand, a statement like “ $a = a + 5$ ” is reversible, since you can retrieve the original value.

The properties of such a quantum computer make it possible to devise very efficient algorithms. One such algorithm, described by L. Grover (see the references), is searching in an unsorted array of items. With ordinary computers this can not be done more efficient than in $O(N)$ steps (an exhaustive search: each item is examined in the worst case), with this quantum algorithm, however, the number of steps needed is $O(N^{1/2})$.

Devising quantum algorithms requires a thorough understanding of quantum mechanics, there is no doubt about that. Another question is: do we need quantum computers and special programming languages to program (or perhaps emulate) these algorithms? This article tries to show that at least for the quantum search algorithm, a classical programming language like Fortran is quite fit (yes, the pun is fortunate).

The principles

To understand the quantum search algorithm we will need to consider some of the differences mentioned above in more details. The most peculiar difference between quantum computing and classical computing is that a variable in a quantum computer has all possible values at the same time, only each value has its own probability of being observed. Actually, each value is associated with the complex value of the quantum wave function for that variable. The probability of observing that value is the square of the modulus of that wave function.

If you do observe a quantum variable, then one of the possible values is selected – truly at random – and the wave function collapses: all values will get a wave function of 0, except the one that you observe, this will have a wave function of modulus 1.

As mentioned in the introduction, an important aspect is that you can not use simple assignments like “ $a = 5$ ” in a computation. This complicates in particular the use of random numbers. But there is a way out: you can store information about the previous value in the *phase* of the wave function. (This is known as the Walsh-Hadamard transformation). Using this clever trick you can retrieve the previous value by applying the same transformation again.

Note:

L. Grover does not comment on floating-point operations, but if a computation is to be reversible at all times, floating-point operations are going to be very different from what we are used to. After all, the reversibility of the statement “ $a = a + 5.0$ ” depends heavily on the size of the original value of the

¹ E-mail address: arjen.markus@wl.delft.nl

variable *a* in relation to the number 5.0. If *a* is very small, then the result will simply be 5.0 and even if *a* is large enough to make the result differ from 5.0, you probably loose accuracy.

Basically, the idea behind the quantum search algorithm is simple: use a random search like in the program sketched below and make sure that during each iteration the chance of finding the item we are looking for is increased. Here is a classical program for a random search:

```
integer, parameter      :: n = 100
integer                 :: i
integer                 :: idx
integer, dimension(1:n) :: item

do i = 1,n
    idx = random(n)    ! Return a random integer between 1 and n
    if ( item(idx) == 1 ) then
        write(*,*) 'Found the item: index = ',idx
        exit
    endif
enddo
```

On average you will need *n* steps to find the item and there is no guarantee that you will have found the index. But with the following quantum search algorithm the number of steps reduces to $0.5N^{1/2}$ to $0.8 N^{1/2}$, depending on the value of *N*:

```
integer      :: i
integer      :: r

integer      :: n      ! The number of items in the array
integer      :: eta   ! The number of iterations that gives us
                      ! the exact answer

r = q_random(n,0)

do i = 1,eta
    if ( item(r) == 1 ) then
        call invert_phase
    endif

    r = q_random(n,r)
    if ( r == 0 ) then
        call invert_phase
    endif

    r = q_random(n,r)
enddo

write(*,*) 'Index is : ', r
```

The precise number of iterations, *eta*, is important: only after precisely that number of iterations is it safe to observe the value of the variable *r*, because then the wave function has collapsed to the exact value that we are looking for: the index of the element in the array *item* that has the value 1.

Note that it is necessary to pass the previous value to the function that returns a random number. The routine *invert_phase* inverts the phase of the complete wave function.

As shown in the article that appeared in Dr Dobb's Journal, this algorithm gives the desired result and for *N*=4, the number of iterations (*eta*) is 1. (A complete analysis of the algorithm requires quite some quantum mechanics. It is described in the second reference.)

Implementation of the algorithm

How do we emulate this with Fortran? Well, we know that the variable *r* in the above program can only take on the integer values 1 to *N* and each is associated with a complex value – the value of the wave function. So, here is a derived type to simulate this:

```
type quantum_integer
```

```

integer :: i                      ! The value
complex :: ampl                  ! The wave function
end type quantum_integer

```

And the variable *r* becomes an array (in the program it is called *qi*) of such integers, to emulate the multiple values it can have:

```
type(quantum_integer), dimension(1:nostates) :: qi
```

The single check “if (*item(r) == 1*)”, which implicitly evaluates the condition for all possible values of *r* must be replaced by Fortran’s equivalent of such a check:

```

where ( iv == 1 )
    qi = q_invert_phase( qi )
endwhere

```

The body of the program mimicks the above quantum-mechanical code:

```

call q_init( qi )
iv = 0
iv(3) = 1
do i = 1,nosteps
    where ( iv == 1 )
        qi = q_invert_phase( qi )
    endwhere
    qi = q_random( nostates, qi )
    where ( qi == 1 )
        qi = q_invert_phase( qi )
    endwhere
    qi = q_random( nostates, qi )
enddo
write(*,*) q_observe( qi )

```

Of course, the full code is longer than this, as several operations on the derived type are involved. But the essence is that the quantum-mechanical version is emulated *step by step*. Nowhere do we use explicitly the values of the random variable or of the wave function. (Even the implementation of the function *q_observe()* is such that the wave function collapses to a single state).

Conclusion

Quantum computing is a fascinating area of research, which theoretically has a great potential for solving tough problems efficiently. However, the nature of quantum mechanics makes it difficult to design the corresponding algorithms. Maybe emulations of such systems as demonstrated here can help design them.

Of course, the Fortran version will not run in a time complexity of $O(N^{1/2})$: it still has to iterate over all possible values and that means the time it needs, $O(N^{3/2})$, is worse than an exhaustive search. Still, when you look at the number of accesses to the array as a whole, this is the same as for the quantum-mechanical version.

References

Lov K. Grover
 Searching with Quantum Computers
 Dr Dobb's Journal, april 2001, pp 34-43

Lov K. Grover
 From Schrödinger's Equation to the Quantum Search Algorithm
<http://www.bell-labs.com/user/lkgrover/papers.html>

The full program

Below you find the complete program (the write statements are merely debugging tools, they would be impossible in an actual quantum-mechanical program). The output is shown after that:

```

!
! Program ad hoc: implement quantum computing principles
! Based on an article by Lov K. Grover about the quantum
! search algorithm, DDJ, april 2001
!
module quantum_computing
  type quantum_integer
    integer :: i
    complex :: ampl
  end type quantum_integer

  interface operator(==)
    module procedure quantum_single_is_equal
    module procedure quantum_array_equals_value
  end interface

contains

subroutine q_init( qi )
  type(quantum_integer), dimension(:), intent(inout) :: qi
  integer :: i
  complex :: uniform_ampl

  uniform_ampl = cmplx( 1.0/sqrt(real(size(qi))), 0.0 )
  do i = 1,size(qi)
    qi(i)%i = i
    qi(i)%ampl = uniform_ampl
  enddo
end subroutine q_init

logical function quantum_single_is_equal( qi, iv )
  type(quantum_integer), intent(in) :: qi
  integer, intent(in) :: iv

  quantum_single_is_equal = qi%i == iv
end function quantum_single_is_equal

function quantum_array_is_equal( qi, iv ) &
  result(log_array)
  type(quantum_integer), dimension(:), intent(in) :: qi
  integer, dimension(:), intent(in) :: iv

  logical, dimension(1:size(qi)) :: log_array
  integer :: i

  do i = 1,size(qi)
    log_array(i) = qi(i)%i == iv(i)
  enddo
end function quantum_array_is_equal

function quantum_array_equals_value( qi, iv ) &
  result(log_array)
  type(quantum_integer), dimension(:), intent(in) :: qi
  integer, intent(in) :: iv

  logical, dimension(1:size(qi)) :: log_array
  integer :: i

  do i = 1,size(qi)
    log_array(i) = qi(i)%i == iv
  enddo
end function quantum_array_equals_value

function q_random( nstates, qi ) &
  result(qr)
  type(quantum_integer), dimension(:), intent(in) :: qi

```

```

type(quantum_integer), dimension(1:size(qi))      :: qr
integer, intent(in)                                :: nostates

integer :: i
integer :: j
integer :: k
integer :: and_bits
complex :: new_ampl
complex :: signed_ampl

new_ampl = cmplx( 1.0/sqrt(real(nostates)), 0.0 )

do i = 1,nostates
    qr(i)%i = i
    qr(i)%ampl = (0.0,0.0)
    do j = 1,nostates
        signed_ampl = new_ampl
        and_bits = iand( qi(j)%i-1, qr(i)%i-1 )
        do k = 0,bit_size(and_bits)-1
            if ( btest(and_bits,k) ) signed_ampl = -signed_ampl
        enddo
        qr(i)%ampl = qr(i)%ampl + signed_ampl * qi(j)%ampl
    enddo
enddo

end function q_random

function q_invert_phase( qi ) &
    result( qr )

type(quantum_integer), dimension(:, intent(in) :: qi
type(quantum_integer), dimension(1:size(qi))      :: qr

integer :: i

do i = 1,size(qi)
    qr(i)%i = i
    qr(i)%ampl = -qi(i)%ampl
enddo

end function q_invert_phase

integer function q_observe( qi )
type(quantum_integer), dimension(:) :: qi

integer :: i
real    :: rnd
real    :: probability

do i = 1,20
    call random_number( rnd )
enddo

probability = 0.0
do i = 1,size(qi)
    probability = probability + abs( qi(i)%ampl ) ** 2
    if ( probability > rnd ) then
        q_observe = qi(i)%i
        exit
    endif
enddo

do i = 1,size(qi)
    if ( qi(i)%i /= q_observe ) then
        qi(i)%ampl = (0.0,0.0)
    else
        qi(i)%ampl = (1.0,0.0)
    endif
enddo

```

```

end function q_observe

end module quantum_computing

program quantum_search
use quantum_computing
implicit none

integer, parameter :: nostates = 4
integer, parameter :: nsteps = 1
type(quantum_integer), dimension(1:nostates) :: qi
integer, dimension(1:nostates) :: iv
integer :: i
integer :: j

!
! Initialise the quantum "system"
!
call q_init( qi )
write(*,'(a,10f5.2)' ) 'Init', ( real(qi(j)%ampl), j = 1,nostates )

!
! The test function is represented as an array
!
iv = 0
iv(3) = 1

!
! Find the correct state
! Note:
! Counting starts at 1, not 0.
!
do i = 1,nsteps
  where ( iv == 1 )
    qi = q_invert_phase( qi )
  endwhere

  write(*,'(a,10f5.2)' ) 'First', ( real(qi(j)%ampl), j = 1,nostates )

  qi = q_random( nostates, qi )
  write(*,'(a,10f5.2)' ) 'Random', ( real(qi(j)%ampl), j = 1,nostates )

  where ( qi == 1 )
    qi = q_invert_phase( qi )
  endwhere
  write(*,'(a,10f5.2)' ) 'Second', ( real(qi(j)%ampl), j = 1,nostates )

  qi = q_random( nostates, qi )
  write(*,'(a,10f5.2)' ) 'SR', ( real(qi(j)%ampl), j = 1,nostates )
enddo
write(*,*) q_observe( qi )
stop
end program quantum_search

```

The output:

```

Init 0.50 0.50 0.50 0.50
First 0.50 0.50-0.50 0.50
Random 0.50-0.50 0.50 0.50
Second-0.50-0.50 0.50 0.50
SR 0.00 0.00-1.00 0.00
3

```

(The last line is the correct index).