International Journal of Quantum Information
Vol. 3, Supplement (2005) 133–141

World Scientific
www.worldscientific.com

TOWARD A PRACTICAL ENVIRONMENT FOR QUANTUM PROGRAMMING*

S. YAMASHITA, M. NAKANISHI, AND K. WATANABE

Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, JAPAN
E-mail: {ger, m-naka, watanabe}@is.naist.jp

This paper proposes a practical framework for quantum programming. In our framework, the parts of a program to be performed on a quantum computer are almost automatically determined, and the other parts are performed on a classical computer. We only consider Grover Search to be performed on a quantum computer in the framework because the other quantum algorithms known so far cannot be applied to general cases. By considering only Grover Search, we have several advantages that show our framework is really practical.

1. Introduction

Many efforts have focused on the physical realization of quantum computers and new efficient quantum algorithms for the realization of quantum computation. Such researches are obviously indispensable to perform quantum computation. However, even after the success of these researches, we cannot obtain a better computing environment than today's from a *practical* viewpoint because at least two of the following issues might exist and be obstacles to realize a *practical* quantum computing environment.

The first issue is that for some computational tasks, the cost of quantum computation (i.e., the time and money to perform computation) is much higher than classical computation. The number of necessary computational steps (in the Turing machine model) has been proven to be the same for some computational tasks between quantum and classical cases. For such computational tasks, quantum computers may not be a good choice since the number of necessary primitive operations to realize one computational step is much larger in quantum computation than in classical. It can also be expected that the cost of one primitive operation

*This work is supported by the Japanese Ministry of Internal Affairs and Communications under SCOPE project.

is higher than in classical cases. In other words, for some tasks, we should use a classical instead of a quantum computer.

The second issue is that it is almost impossible to manually design a desired sequence of primitive operations (i.e., a quantum circuit) for problems of practical sizes. Therefore, as in the classical case, we need both an efficient way of describing an algorithm in higher levels (i.e., programming language) and an automatic way of transforming it to primitive operations.

These two issues, which have not been discussed seriously in the research community, are the obstacles for practical quantum computation. Note that it is not trivial to deal with the above issues by simply using the techniques currently available in classical programming environments.

Therefore, in this paper, we propose a framework to initiate research that deals with the above issues. In the framework, a programmer can write a program by using (exactly the same) C++ language whereas most existing researches demand some extra knowledge to write a program for quantum computers[1,2,3]. Then the framework automatically extracts the parts of the program that can be performed faster by Grover Search[4] and generates corresponding quantum circuits and interface codes for a classical computer. We can also simulate the entire program in only a classical programming environment since we can easily prepare C++ source codes to simulate quantum circuits.

2. Overview of Proposed Strategy

We present an overview of the proposed framework in Section 2.3 after explaining the two main features of our strategy in Secs. 2.1 and 2.2.

2.1. *Classical and Quantum Co-design*

Let us start with an observation on quantum computations from the view point of *real* computation time, i.e., not asymptotic time complexity as usually discussed in this community. It is well-known that a quantum computer can perform all kinds of classical computation with only polynomial overhead. The overhead is mainly due to the simulation of classical computation by only using quantum elementary gates, i.e., reversible logic gates. It should also be assumed that one logical operation on a quantum computer is much slower than on a classical computer. Therefore, from a practical viewpoint, it may not be wise to use a quantum computer to only simulate classical computations: a quantum computer should be used only for parts that can actually receive the quantum speed up.

Therefore, we consider a quantum computer as a coprocessor that can boost the speed of some (not all) parts of a program on a classical computer. A similar

computation model called "QRAM" has been discussed in the literature[1].

We also consider that the separation of quantum and classical parts should be done automatically by our programming framework as much as possible. We call such separation **classical and quantum co-design** borrowing the terminology "hardware-software co-design[5]." If we consider the *practical* usage of quantum computers, the issue of classical and quantum co-design becomes important, and thus we propose our framework as a promising classical and quantum co-design methodology.

2.2. *Limitations to Grover Search*

As we mentioned, our classical and quantum co-design framework should determine a part in a program for quantum computation. Then the first question is: what types of parts are candidates for quantum computation? Many algorithms show quantum speed up, e.g., Shor's factoring algorithm[6], Grover Search[4], Quantum Walk, etc. Among them, only Grove Search has potential for general purposes. In other words, other quantum algorithms are useful for very specific situations, and thus we cannot consider using them in general programs. Thus we limit a quantum computer to be used as Grover Search in our framework. It should also be noted that we accrue many benefits (as we will describe below) from this limitation.

2.3. *Overview of Proposed Framework*

As mentioned above, only Grover Search may be used for general programs among the existing quantum algorithms. However, it is not trivial to use Grover Search in a general program. Thus, we propose an efficient framework to use Grover Search. Our framework is as shown in Fig. 1, and it has the following steps.

- **Step 1.** A programmer writes a program by using standard C++ programming language.
- **Step 2.** He/she inserts a special C++ comment to specify that the evaluation of the if-expression directly after the comment may be done by Grover Search. (This will be explained in detail in Sec. 3.)
- **Step 3.** Our framework automatically generates a corresponding quantum circuit to evaluate the if-expression specified at Step 2. (This step will be explained in detail in Sec. 4.1.)
- **Step 4.** By evaluating the number of primitive quantum gates in the quantum circuit generated at Step. 3, the framework determines whether the

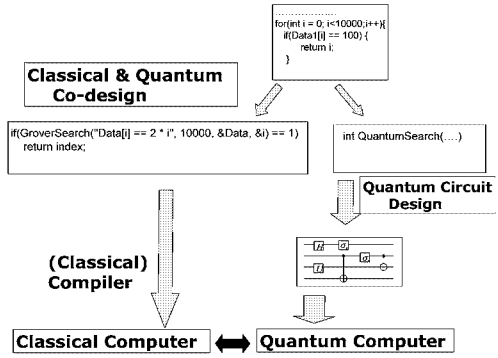

Figure 1. This fi gure shows an overview of proposed framework. Some parts of a source program are automatically extracted to be performed on a quantum computer. Corresponding quantum circuits for the parts are automatically generated.

processing time of the quantum circuit is faster than the processing time of the corresponding if-expression on a classical computer. If the framework determines that the quantum circuit is faster, it generates some interface source codes for a classical computer. (This will be explained in Sec. 4.2.)

- **Step 5.** The framework optionally generates a C++ source code that simulates the behavior of the quantum circuit generated at Step. 3. This code can be used for the entire simulation by using only a classical computer as we will explain in Sec. 4.2.

3. Program Part Extraction for Grover Search

Suppose we encounter the part as shown in Fig. 2 in our target program. This part essentially needs 10,000 evaluations of the if-expression on a classical computer. However, if we utilize Grover Search, we need only roughly $\sqrt{10,000}(= 100)$ times of Grover Iterations. However note that a single evaluation of the if-expression on a classical computer is much faster than a single Grover Iteration. Thus, as mentioned in the previous section, our strategy is the following: First, we construct a quantum circuit for the corresponding Grover Iteration by the method described in Section 4.1. Then we estimate the necessary steps, i.e., the number of elementary gates, on a quantum computer to perform Grover Search. Then we can determine whether this part should be performed on a quantum computer.

The above decision can be done automatically without any help from a programmer. However, the following issues cannot be solved without help from a programmer: Grover Search finds *one* solution *at random* among several solu-

tions, and therefore, the behavior of Grover Search is not exactly the same as an if-expression in a for-loop where the first solution in the for-loop always becomes the solution. In other words, Grover Search can be used for cases when a programmer wants to find only one solution, and he/she does not care about the order of the for-loop. This fact is obviously only known to the programmer. Therefore, in our framework a programmer is requested to put a special comment "//Quantum Search" to specify this property before the corresponding if-expression as shown in Fig. 2. This is a minimum load for programmers. Indeed they do not need to know the details of quantum computation, and the entire program can be written in pure C++ unlike other quantum language researches[1,2,3].

4. Compiling the Extracted Parts for Grover Search

For a given C++ program, we can extract a part that should be performed on a quantum computer as mentioned above. Then our framework generates a quantum circuit for a single Grover Iteration corresponding to the part as will be mentioned in Sec. 4.1. It also generates additional C++ source codes to utilize the quantum circuit from a classical computer and deals with Grover Search error, as will be mentioned in Sec. 4.2.

4.1. *Generating a Quantum Circuit for Grover Search*

Here we describe the construction of a quantum circuit fed to a quantum computer. Almost all the proposed quantum circuit design methods are based on decomposing a unitary matrix that corresponds to a target quantum algorithm into elementary gates[7]. Although such circuit design methods can be applied to any quantum algorithm, they cannot be applied to large problems since the size of a unitary matrix is 2^n for quantum algorithms with n qubits. Since program parts for quantum computation should include large size Grover Search (otherwise the parts should be performed on a classical Computer, as mentioned in Sec. 2.1), we cannot utilize unitary matrix decomposition in our framework. Thus we adopt the following design methodology.

- We adopt the structure of the Grover Iteration as shown in Fig. 3. In the figure, W means the Walsh-Hadamard transformation. C_0 is a transformation that inverts the phase of state $|0\cdots0\rangle$. Note that their structures are fixed for all problems, and thus we can easily construct them as shown in the figure.
- C_f is a transformation that inverts the phase of state $|x\rangle$ such that x is a binary representation of the solution for Grover Search. This is of-

```
for(int i = 0; i<10000;i++){
//Quantum Search
  if(Data[i] == 2 * i) {
      return i;
  }
}
```

Figure 2. Example of a for-loop that may be solved by Grover Search.

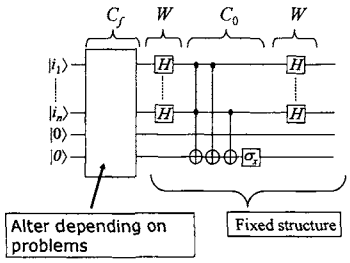

Figure 3. Quantum circuit for Grover Iteration.

ten called an "oracle" in the literature. This part changes depending on search problems, and thus we need to construct C_f from scratch.

In other words, we do not generate the entire quantum circuit from scratch, but we utilize the known structure of Grover Search as much as possible. Thus, our essential task is to construct a quantum circuit for C_f.

Below we explain how to construct C_f by using an example. Suppose we want to generate a quantum circuit for Grover Search that corresponds to the part, as shown in Fig. 2. Then C_f should work as follows.

- It takes an input quantum state $|i\rangle$ where i is a binary representation of integer i. (This needs $\log 10,000$ qubits.)
- It does not change the quantum states of $|i\rangle$, but transforms one bit ancilla qubit state from $|0\rangle$ to $|1\rangle$ if (Data[i] == 2 * i) is satisfied. Otherwise, it does not change the ancilla qubit.

In our framework, we assume that we have quantum registers (quantum memory) that take $|i\rangle$ as an input and outputs the value of the i-th element in the register into ancilla qubits. This is the most standard model of quantum memory. We can see an example of a quantum register in Fig. 4 where i and $Data[i]$ are encoded by three qubits.

Here we want to implement a function such that it becomes 1 only when (Data[i] == 2 * i). This is a Boolean function with respect to the bits for the binary expressions of i and $Data[i]$. For simplicity, we consider that i and $Data[i]$ are expressed by only three bits. Then our essential task is to construct a quantum circuit for the function with six input variables, and we can construct such a circuit by generalized Toffoli gates as shown in Fig. 4. In the figure, a black circle means the normal control bits whereas a white circle works in the opposite way, i.e., a generalized Toffoli gate works when the qubit state is $|0\rangle$. The function corresponding to (Data[i] == 2 * i) is calculated into the qubit on the bottom. The

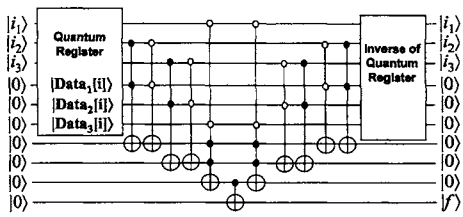

Figure 4. This figure shows an example of how to construct C_f. The left half of the circuit calculates the function into the qubit on the bottom. Then the remaining half of the circuit restores the initial states for the other qubits. i_1 and $Data_1[i]$ are the most significant bits for the encoding.

right half part of the circuit is for resetting ancilla qubits to the initial states. Note that this is necessary when used in Grover Search.

The expression in the example, (Data[i] == 2 * i), is relatively easy, but even for general cases, we can construct C_f as follows:

- **Step 1.** Generate a Boolean function f corresponding to the expression.
- **Step 2.** Generate an exclusive-OR sum of products expression (ESOP) form for f by a standard method in the (classical) logic design[8].
- **Step 3.** Construct a quantum circuit by replacing each product term in the ESOP form at Step. 2 with a generalized Toffoli gate.
- **Step 4.** Convert each generalized Toffoli gate into elementary gates by the method in Ref. 9.

Note that even though the above procedure is not sophisticated, it can be applied to large problems since we only treat the expression and representation of functions, not unitary matrices with exponential size. Therefore, we claim that our framework is practical. We may incorporate optimization into the above procedure.

4.2. *Generating Additional C++ Codes*

After generating a quantum circuit, our framework automatically modifies the part in the original C++ program. For example, the part as shown in Fig. 2 is converted into the following lines in the original C++ program:

```
if (GroverSearch("Data[i] == 2 * i", 10000, &Data, &i, "i") == 1)
    return i;
```

The function `GroverSearch` is also defined so that it has the following functionalities: (i) It asks a quantum computer to perform a single Grover Search and receives an answer from the quantum computer. (ii) It verifies the solution. If

the solution is incorrect, it continues to ask the quantum computer until a correct answer is obtained or until it has repeated it an appropriate number of times. (This will be mentioned below.) (iii) If a correct solution cannot be obtained within an appropriate number of repetitions, it performs the original C++ source code to find a correct solution. These additional operations are necessary because a single Grover Search has some constant error, i.e., we cannot always get a correct answer by Grover Search. The appropriate number of repetitions are determined as follows. Let the computational cost (on a classical computer) of all the evaluations in a for-loop in the original C++ source code be T_C. Also let the computational cost of a single Grover Search be T_Q. In the following discussion, for simplicity, let us assume the probability that a single Grover Search will succeed to be $3/4$. Then the probability that we cannot find a correct solution after k trials of Grover Search is $(\frac{1}{4})^k$. Thus, the expected total computational cost of the above procedure is $((\frac{1}{4})^k \cdot T_C) + \sum_{l=1}^{k-1}(\frac{3}{4} \cdot (\frac{1}{4})^{l-1} \cdot l \cdot T_Q)$. We can calculate an appropriate value for k to minimize the expected total computational cost, a value which our framework uses in `GroverSearch`.

Optionally, we can make a C++ source code to simulate the behavior of a quantum circuit, although the simulation on a classical computer may take a long time. If we replace the part that calls a quantum computer for Grover Search with such a code, the entire source code becomes pure C++ language so that we can perform it on a classical computer, i.e., we can simulate the behavior of classical and quantum computers by only using a classical computer.

5. Conclusion

Unlike existing research on quantum programming, our strategy only focuses on Grover Search. By doing so, we have the following advantages: (i) A programmer can write a program by only using (exactly the same) C++ language, and thus the entire program can be simulated by only using current programming environments. (ii) Programmers do not need to worry about which parts should be done by quantum computers. (iii) We can design quantum circuits with a strategy that can be applied to large size problems.

As far as we know, this is the first framework to build a *practical* programming environment for quantum computation. We are now constructing a prototype of our proposed framework and will evaluate it in the future. For other future work, we will consider the possibility of utilizing Amplitude Amplification[10], which is a generalization of Grover Search, in our framework.

References

1. S. Bettelli, T. Calarco, and L. Serafini, *Eur. Phys. J.* **25**, 181 (2003).
2. J. W. Sanders and P. Zuliani, *Math. of Program Construction*, 80 (2000).
3. B. Ömer, *PhD thesis*, Technical University of Vienna (2003).
4. L. K. Grover, *Symp. on Theory of Computing*, 212 (1996).
5. G. De Micheli and M. Sami, *Hardware-Software Co-design*, Kluwer Academic Publishers (1996).
6. Peter W. Shor, *Symp. on Foudations of Computer Science*, 124 (1994).
7. R. R. Tucci, quant-ph/9902062.
8. T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers (1999).
9. A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. *Physical Review A*, **52(5)**, 3457 (1995).
10. G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, quant-ph/0005055.