Hierarchical system mapping for large-scale fault-tolerant quantum computing

Yongssoo Hwang · Byung-Soo Choi

Received: 25 February 2020 / Accepted: 7 June 2021 / Published online: 24 June 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract
With respect to large-scale quantum computing, it is important to know precisely and quickly how much quantum computational resources are necessary to study components and the whole system. If such quantum resources are provided, how much performance a quantum computer provides is also a critical issue. Unfortunately, it is practically infeasible to deal with such problems with conventional methods based on a non-structured description about quantum algorithm. To overcome the problems, we propose a fast method by using a hierarchically structured description about quantum algorithm which is much more compact than the conventional method. During the process, the dedicated computing regions and their interconnection are dynamically mapped onto a structured quantum computing system architecture. In our study, the proposed method works very faster such as 1 h than 1500 days for Shor algorithm to factorize a 512-bit integer. Meanwhile, since the combination of structured code and architecture provides a high degree of locality, it requires less SWAP chains, and hence, it does not increase the quantum computation depth more than expected.

Keywords System mapping · Quantum assembly code · Large-scale quantum computing · Quantum computer architecture

1 Introduction and background
The era of quantum computing already started. Several big IT corporations have been devoted to develop quantum computing devices, and some of them are now providing quantum computing cloud service to the public [1–3]. But the scale of the devices are still small (dozens of qubits), and therefore, applications for them are also very
limited [4]. While we look forward to seeing a quantum supremacy with such a small-scale quantum device soon, most of the public have interest in a large-scale universal quantum computer that can run large-sized quantum algorithms to solve real-world problems.

Suppose that you have a quantum computing hardware and a quantum algorithm. What do you have to do to run the algorithm with the hardware? Quantum algorithm is a logical description of how to solve a given problem. In general, it is based on ideal quantum computing hardware with noiseless gate and arbitrarily long qubit interaction. However, in reality, a quantum computer as a physical entity has a certain physical and logical limitation. For example, qubits might be peculiarly arranged and are only interacting with neighbors (see Ref. [1,2]). Therefore, to execute a quantum algorithm on a quantum computer, we have to prepare an architecture-specific description of a quantum algorithm beforehand. This is why a quantum computing system mapping is necessary.

The principle of a quantum computing system mapping is straightforward. Given a target quantum computing system architecture and a quantum algorithm, then initialize a quantum computer architecture (initial qubit mapping), and recast a quantum algorithm for the architecture (gate scheduling). Obviously, the detailed mapping procedure completely depends on a quantum computer architecture, the representation of a quantum algorithm, and a mapping algorithm. In the early stage, only the qubit connectivity was considered for the quantum circuit mapping [5–9]. However, as the research of quantum computing keeps going on, more diverse factors such as the hardware calibration data [10,11] and the classical control scheme [12] of a quantum computing system are being considered.

An architecture-aware description of a quantum algorithm, the main product of the system mapping, is called a system code in this work. Given a system code, it is possible to run a quantum algorithm on the quantum computer in practice by executing the code. Another output of the system mapping is an expected performance of a quantum computing. Since the system mapping traces all quantum instructions of a quantum algorithm on the specified quantum computer architecture, it is possible to analyze some performance of a quantum computing, i.e., the quantity of quantum resource, a circuit depth, a T-gate depth, and so on.

To perform the system mapping, the quantum algorithm should be provided in a quantum assembly code format. A quantum assembly code (QASM) is an intermediate representation of a quantum algorithm between an abstract logical description and a physical machine instruction description [13–15]. It corresponds to the list of quantum instructions denoting a combination of a quantum gate and target qubit(s) and is generated through a compile by taking a programmed quantum algorithm written in a computer programming language. Due to the lack of the standard for QASM, the specific representations are slightly varied [2,13,14,16,17].

There are two kinds of QASMs in terms of structure: a non-modular code and a modular code. A non-modular QASM describes a quantum algorithm at the individual gate level. On the other hand, in a modular QASM, a quantum algorithm is described with respect to the complicated quantum functions called module. Figure 1 shows the examples of both QASMs. In general, it seems that a modular QASM looks like a modern computer programming language C/C++ [14] because the QASM is composed
Fig. 1 Two kinds of QASM for the algorithm Binary Welded Tree [14]: a non-modular QASM and b modular QASM. A non-modular QASM simply enumerates all the operations in the quantum gate level, but a modular QASM is composed of several modules each of which is working as a complicated quantum function.

Fig. 2 A typical form of the module in the modular QASM generated by ScaffCC compiler [14,18] of one main module and several sub-modules, and a module itself is defined by the executions of quantum gates and other modules. The typical form of a module is shown in Fig. 2.

To date, most quantum computing system mappings have been performed by taking the non-modular QASM [5–10,12]. System mapping usually deals with the problems that happened in the practical implementation and the execution of a quantum computing. Considering the research and development level on the quantum computing system, so far most of the system mapping research has focused on the small-scale
system and algorithm. For the small-scale quantum computing, we believe that it is enough to take a non-structured algorithm description, and a plain qubit array of the size as much as the number of algorithm qubits. Such an approach for small-scale quantum computing is a straightforward approach. Besides, since the methodology has been working well in that regime, the requests for other approaches have not been seriously raised. For reference, we show the flowchart of the non-modular QASM synthesis in Fig. 10.

When, however, we turn our attention to a large-sized quantum algorithm, several practical problems arise with the non-modular QASM. Please recall that a non-modular QASM simply enumerates quantum instructions in the quantum gate level. Therefore, as the size of a quantum algorithm increases, so does the size of the non-modular QASM. It follows the scale of the algorithm. For example, we obtained a 39 TB sized non-modular QASM for Shor algorithm [14,18,19] to factorize a 512-bit integer (Shor \( N = 512 \)). Please see Fig. 3. Due to the lack of classical storage and memory, we could not even attempt to generate the non-modular QASM for the more large-sized quantum algorithm.

Even if we generate a non-modular QASM for a large-sized quantum algorithm successfully, it does not seem practical to perform the system mapping by reading such huge size QASM. As will be argued later, we estimate it will take 1500 days for the system mapping by taking the 39 TB QASM as input (see Fig. 9). If any optimization is applied, it will take more time.

Obviously, the size of the interesting problem to the public goes beyond the above-mentioned problem size. Therefore, we definitely have to deal with a large-sized QASM, and in doing so, we believe practical problems caused by such enormous sized QASM will be one of the critical issues in the classical control part in quantum computing.

In this work, we present how to perform a system mapping for large-scale quantum computing. To this end, we turn our attention to a modular QASM instead. As mentioned above, a modular QASM is hierarchically structured as composed of mod-
ules, and we found that such a structure can avoid the problems scalability problem in the QASM. Suppose that there is a composite quantum operation $U$ composed of $k$ quantum gates, and $U$ is called as much as $n$ times in the algorithm. To represent such $n$ iterations, a non-modular QASM requires $k \times n$ quantum instructions, but a modular QASM only requires $k + n$ instructions by defining $U$ as a module. It takes $k$ instructions to define $U$ and $n$ instructions to call $U$ as much as $n$ times.

For example, in the modular QASM of Shor-4 (see Fig. 12), the module $QFT_2$ is composed of 28 sub-modules $cRz_IPx\ldots$ and 8 Hadamard gates. Each sub-module is made up of 3 DecompositionRotation modules and 2 CNOT gates. Then, to describe $QFT_2$ module in quantum gate level (non-modular QASM), a total of $84 \cdot n_{DR} + 64$ instructions are required where $n_{DR}$ is the length of the decomposed sequence of $R_Z(\theta)$ for an arbitrary $\theta$. Since the $QFT_2$ module is iterated 192 times overall, then to present all the parts of $QFT_2$ modules in the algorithm, $4.04 \times 10^6$ instructions are required. Note that a $R_Z(\theta)$ gate is usually decomposed into a total of around 250 $H$, $S$, and $T$ gates in the error precision $10^{-10}$ ($n_{DR} \approx 250$) [20]. On the other hand, in the modular description, a total of 483 instructions are enough for indicating all the parts of the module $QFT_2$ in the algorithm.

Even though the specific values of $k$ and $n$ are dependent on the algorithm and its program implementation, empirically they are large numbers in a non-trivial quantum algorithm. For example, for Shor algorithm, $k$ is $O(10^2) \sim O(10^4)$ and $n$ is $O(10^4) \sim O(10^6)$. In this regard, a modular QASM is much smaller than a non-modular QASM (see Fig. 3). Therefore, with the modular QASM approach, it will be possible to generate and manage a QASM practically for a large-sized quantum algorithm. Furthermore, it may be possible to perform the system mapping in a reasonable time-bound. Saying the result in advance, for the mapping of Shor $N=512$, it is enough around 1 hour instead of 1500 days (see Fig. 9).

Currently, only a few quantum compilers support a modular QASM. This work is currently compatible with open-source quantum compiler ScaffCC [14,18], but can be applied to openQASM [13] either. We believe the heart of the present work can be applied to most of the structured quantum assembly codes which will be coming in the future.

In the remainder of this paper, we describe how to exploit the hierarchical structure of the modular QASM on a pre-assumed quantum computer architecture and discuss the strengths and weaknesses of the proposed mapping.

### 2 Hierarchical system mapping

#### 2.1 Hierarchically structured qubit layouts

System mapping begins with the initialization of a quantum computer architecture, i.e., a qubit array. A qubit array is not specifically limited, but in this work we assume that it is hierarchically structured. The whole quantum computer is then made of modules (computing regions) and a communication bus interconnecting the modules. The number and the function of the modules are determined by the input modular QASM. At a module, qubits are manipulated by the quantum instructions described in...
a modular QASM, and transmitted between modules through the communication bus. All the modules and the bus are, respectively, composed of logical qubits encoded by quantum error-correcting codes. Figure 4 shows an example of a quantum computer architecture where modules are arranged on the 2D array.

The communication bus can make quantum computing more reliable and efficient. On the bus, a logical data qubit is permitted to interact with nearby ancilla qubits only, and therefore, a quantum error (even if occurred) is not propagated to other logical data qubits. Besides, the bus allows the parallel qubit movements between modules at the cost of qubits (large bandwidth). As will be discussed in Sect. 3, in quantum computing with the hierarchical system architecture, more qubit movements are required to perform the quantum operations at other modules, but most of them are executed in parallel.

In a modular QASM, qubits are classified into three types: local qubits, parameter qubits, and dummy qubits. Local qubits are initialized, manipulated, and measured within a module, whereas parameter qubits are passed between modules over a communication bus. For reference, in the non-modular QASM, there does not exist the parameter qubit explicitly. Besides, dummy qubits denoted by “NULL” are sometimes required to form the 2-dimensional rectangular shape of a module. The dummy qubits may be sometimes used as a communication route for the qubit movements within a module. Figure 5 shows an example of the module qubit array where three kinds of qubits are placed.
Hierarchical system mapping for large-scale fault-tolerant

2.2 Mapping algorithm

In general, we perform the system mapping to find a system code that is required to execute a quantum algorithm in practice. The heart of our mapping with the modular QASM is that a module always works with the same cost at a dedicated physical space (see Fig. 5) whenever it is called. In that case, it is enough to perform the mapping each module in the modular QASM only once as follows. When a new module is called, we perform the mapping of the module and records the result in a table. If an already mapped module is called, we then just need to refer to the result at the table. While the performance of a module is always the same, but the cost for the qubit movements to the dedicated physical space differs according to a calling (parent) module and arguments (parameter qubits). Therefore, whenever a module is called, we have to find the qubit movement path from the present module to the target module and calculate the cost such as the number of SWAP gates. The concepts of the proposed modular mapping and the non-modular mapping are compared in Fig. 6.

Our mapping begins with the mapping of the main module and proceeds the mapping of sub-modules in the logical order. For the mapping of a module, we first need to allocate a physical space for the module next to the previously allocated module and arrange the qubits of QASM on that space as much as required. The module space is made of cells as much as the number of qubits in the module. For the qubit arrangement, in this section, for the purpose of presentation, we simply arrange the algorithm qubits on the physical space (qubits) in the first-come-first-served manner. We then read each quantum instruction from the QASM and conduct an appropriate mapping task. After mapping a module, the allocated physical space for the module will be de-allocated.

As mentioned above, to perform the proposed mapping methodology, we need lookup tables to keep the mapping results of modules. To be more concrete, we need two types of lookup tables, one global table, and several local tables as much as the number of the modules on-working. These tables will be used to record the system

---

1 In Sect. 3, we will discuss an optimized qubit placement of a module.
Fig. 6 The concepts of the non-modular QASM mapping (a) and the proposed modular QASM mapping (b). In the non-modular QASM mapping, a complex quantum function composed of several quantum gates works with different cost depending on the status of the qubit array whenever it is called during the algorithm execution. On the other hand, the modular QASM mapping deals with the complex quantum function at the same cost by defining it as a module. Therefore, it is enough to perform the mapping of a module only once when it is called first and then refer to the mapping result when it is called later. In this regard, the solid line indicates performing the mapping and the dotted line means the referring to the lookup table to take the mapping result.

code, the performance, and resource of modules. At the beginning of the system mapping, a global lookup table is initialized as the performance (such as circuit depth) and the resource (such as the number of qubits) of all modules are zero (or empty). Then, at the beginning of the mapping of a module $M$, a local table $T_M$ is initialized as $cycle[q_i] = 0$ over $i = 1 \sim n$, where $n$ is the number of qubits of the module. Note that $cycle[]$ is a metric to measure the circuit depth of a module, and $cycle[q_i]$ denotes the last circuit index where a quantum operation is applied to the qubit $q_i$. More diverse metrics such as T-gate depth can be added to measure the performance of a module, and therefore the performance of the algorithm. As the mapping of a module proceeds, the metric is being updated as $cycle[q_i] = cycle[q_i] + wait_c + 1$. Note that $wait_c$ is the number of computing cycles required for multiple qubits to be in temporally and spatially ready status to perform a multi-qubit quantum operation. After the mapping, the mapping result of the module $M$ will be recorded on the global table. For example, the circuit depth of $M$ is determined by picking the maximum values from the local table $cycle_M = \max_{i=1}^{n}[cycle[q_i]]$.

In a modular QASM, there are three kinds of quantum instructions: 1- and 2-qubit gate and module. The mapping of an 1-qubit quantum gate is straightforward. It can be done independently from other qubits by arranging the gate to a target qubit $q_i$ at the circuit depth index $cycle[q_i]$. The execution of the gate finishes at circuit depth index $cycle[q_i] + 1$. On the other hand, the mapping of a 2-qubit gate requires that two qubits have to be in ready status spatially and temporally. In other words, both qubits have to be in neighbor, and one or both the qubits are not being manipulated by other operations. Otherwise, they should be placed adjacently by moving qubit, or one qubit has to wait until an operation acting on the other qubit is over. The gate
then acts at the circuit index \( \max\{\text{cycle}[q_i], \text{cycle}[q_j]\} \), where \( \text{cycle}[q_i] \) (\( \text{cycle}[q_j] \)) is the circuit index that a previous operation on the qubit \( q_i \) (\( q_j \)) is finished and ends at \( \max\{\text{cycle}[q_i], \text{cycle}[q_j]\} + 1 \).

The path for qubit movements can be found by applying the shortest path search algorithms such as A* algorithm [7] or Floyd–Warshall algorithm [8] by formatting the status of the qubit connectivity as a graph. These algorithms that search the shortest paths over multiple nodes on a graph at the cost of time and memory can be exploited to synthesize the most efficient quantum circuit. By the way, in this work, for searching the path efficiently, we apply Dijkstra’s shortest path search algorithm that is usually used to find the shortest path between two single nodes on a graph. The purpose of the present work is to present a modular mapping taking a hierarchically structured modular QASM and to discuss the pros and cons of the proposed method against the non-modular mapping. For that, we believe taking such a basic search algorithm is enough rather than struggling with finding the most optimal circuit with spending huge time and memory. Needless to say, for the fair comparison, the non-modular mapping (see Fig. 10) which we implemented also hires the same algorithm. For reference, given an appropriate initial mapping, the circuit mapping with Dijkstra’s shortest path search generates a quantum circuit of comparable performance with circuits by the above-mentioned algorithm with remarkably less time and memory.

The third type of quantum instruction, a module operation, seems like a complicated multi-qubit quantum function. Therefore, on the surface, the required mapping process for the module is similar to that for a 2-qubit gate. The execution of the module begins when all the qubits are temporally and spatially in ready status. The only difference from the case of the 2-qubit gate is, as mentioned above, that a physical space for the module should be allocated. Therefore, to perform the module mapping task, we need to consider qubit movements between the calling (parent) module and the called (child) module.

Suppose that a module \( M_i \) is being mapped now. Among the quantum instructions in \( M_i \), it is a turn to process the instruction \( "M_k(q_a, q_b, q_c)" \) which implies that \( M_i \) calls a module \( M_k \) with the arguments qubits \( q_a, q_b, \) and \( q_c \). For that, we pause the mapping of \( M_i \) and turn our attention to the mapping of \( M_k \). We allocate the physical space for \( M_k \) next to \( M_i \) and pass the argument qubits \( q_a \sim q_c \) to the parameter qubit area of \( M_k \) through the communication bus. We call this qubit movement a forward qubit passing. If all qubits are placed for their designated locations, we start to process the quantum instructions of \( M_k \). If another module is called during the mapping of \( M_k \), obviously we pause the mapping of \( M_k \) and perform the above-mentioned qubit allocation and forward qubit passing for the newly called module. When all the instruction in \( M_k \) is processed, the argument qubits have to be back to the original position of \( M_i \). This qubit movement is called a backward qubit passing. After the mapping of \( M_k \) including both qubit passings, we have to de-allocate the physical space for \( M_k \) and continue the mapping of \( M_i \). Figure 7 shows an example of the process for the module call, and we show the module allocation and de-allocation process throughout the algorithm in Section A.

Figure 11 describes the flowchart of the proposed modular mapping, and Algorithm 1 describes the pseudo-code. In the beginning, the global lookup table \( LT_G \) and the qubit array for main module should be initialized, and then, MAPPING- MODULE
Fig. 7 The mapping process for calling a module is composed of seven steps: 1. (forward) move qubits to the bus, 2. (forward) move to the target module, 3. (forward) move to the parameter qubit cells (dark gray cells), 4. module operations, 5. (backward) move qubits to the bus, 6. (backward) move to the original module, and 7. (backward) move to the original qubit positions.

function is called with the argument main module and LTG. After all the operation for main module, the qubit array should be de-allocated and the performance of the algorithm is determined by referring to LTG.

Algorithm 1 System Mapping for the modular QASM

```
function MAPPING-MODULE(M, LTG)  \Comment{Local Lookup Table}
    Initialize LT_M
    for inst in QASM_M do
        U, Q = inst[gate], inst[qubits]
        if U ∈ Modules then  \Comment{Module}
            Allocate Qubit Array for U
            Forward Qubit Passing: M → U
            if U is already mapped then
                Refer cycle_U to LT_G
            else
                Mapping-module(U, LT_G)
            end if
            Backward Qubit Passing: M ← U
            De-allocate Qubit Array for U
        else if U ∈ one-qubit gates then  \Comment{One-Qubit Gate}
            Apply U to Q
        else if U ∈ two-qubit gates then  \Comment{Two-Qubit Gate}
            Apply SWAP to Q for Q to be in neighborhood
            Apply U to Q
        end if
    end for
    cycle_M = max_Q \left( LT_M[Q](cycle) \right)  \Comment{Performance of the module M}
    LT_G[M](cycle = cycle_M)  \Comment{Update LT_G}
end function
```

Table 1 shows the system code of the module PARSENODEEVEN (see Fig. 5a) based on the initial qubit mapping of Fig. 5b. From the system code, we can obtain the circuit depth as follows,

```
PARSENODEEVEN_{depth} = 6 \cdot X_{depth} + 4 \cdot Toffoli\_Impl_{depth}
```
Table 1  The system mapping result of the module PARSENODEEVEN (see Fig. 5a)

| Index | Qubits (PARSENODEEVEN) |
|-------|------------------------|
|       | \( a[0] \)  | \( a[1] \)  | \( a[2] \)  | \( a[3] \)  | \( \text{even}[0] \)  | \( \text{scratch}[0] \)  | \( \text{scratch}[1] \)  | \( \text{scratch}[2] \)  |
| 1     | PrepZ               | PrepZ               | PrepZ               |
| 2     | X                   |                      |                      |
| 3     | \( FP \)            | \( FP \)            | \( FP \)            |
| 4     | ToffoliImpl         | ToffoliImpl         | ToffoliImpl         |
| 5     | BP                  | BP                  | BP                  |
| 6     | \( FP \)            | \( FP \)            | \( FP \)            |
| 7     | ToffoliImpl         | ToffoliImpl         | ToffoliImpl         |
| 8     | BP                  | BP                  | BP                  |
| 9     | X                   |                      |                      |
| 10    | \( \text{SWAP} \)   | CNOT                | CNOT                |
| 11    |                    | SWAP                |                      |
| 12    |                    | CNOT                | CNOT                |
| 13    | X                   |                      |                      |
| 14    | \( FP \)            | \( FP \)            | \( FP \)            |
| 15    | ToffoliImpl         | ToffoliImpl         | ToffoliImpl         |
| 16    | BP                  | BP                  | BP                  |
| 17    | X                   |                      |                      |
| 18    | \( \text{SWAP} \)   | CNOT                | CNOT                |
| 19    |                    | SWAP                |                      |
| 20    |                    | CNOT                | CNOT                |
| 21    | X                   |                      |                      |
| 22    | \( FP \)            | \( FP \)            | \( FP \)            |
| 23    | ToffoliImpl         | ToffoliImpl         | ToffoliImpl         |
| 24    | BP                  | BP                  | BP                  |

\( FP \) and \( BP \) indicate forward and backward parameter passings between the modules PARSENODEEVEN and ToffoliImpl, respectively. The instructions in lines 3, 5, 6, 8, 11, 14, 16, 18, 21, and 23 are added by the system mapping.

\[
\begin{align*}
+ ( FP_{\text{depth}} + BP_{\text{depth}} ) a[2], \text{scratch}[1], \text{scratch}[2] \\
+ ( FP_{\text{depth}} + BP_{\text{depth}} ) a[2], \text{even}[0], \text{scratch}[2] \\
+ ( FP_{\text{depth}} + BP_{\text{depth}} ) a[2], \text{scratch}[0], \text{scratch}[2] \\
+ ( FP_{\text{depth}} + BP_{\text{depth}} ) a[2], \text{scratch}[1], \text{scratch}[2] \\
+ 2 \cdot \text{SWAP}_{\text{depth}} + 3 \cdot \text{CNOT}_{\text{depth}} + \text{PrepZ}_{\text{depth}}.
\end{align*}
\]

Note that \( ( FP_{\text{depth}} + BP_{\text{depth}} ) a[2], \text{scratch}[1], \text{scratch}[2] \) is the sum of the forward qubit passing and the backward qubit passing between the argument qubits \(( a[2], \text{scratch}[1], \text{scratch}[2] )\) in the module PARSENODEEVEN and the designated space of the module ToffoliImpl. Other qubit passing cost terms are also similarly defined.
module DecomposeRotation_1_472600e00Q ( qbit q )
{
  S ( q );
  S ( q );
  S ( q );
  T ( q );
  H ( q );
  S ( q );
  T ( q );
  H ( q );
  S ( q );
  T ( q );
  H ( q );
  T ( q );
  S ( q );
  T ( q );
  H ( q );
  S ( q );
  T ( q );
  H ( q );
  T ( q );
  S ( q );
  T ( q );
  H ( q );
  T ( q );
  S ( q );
  T ( q );
  H ( q );
  S ( q );
  T ( q );
  H ( q );
  T ( q );
  S ( q );
  T ( q );
  H ( q );
  \}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig8.png}
\caption{An example of the one-parameter-qubit terminal module, the module that decomposes the rotational gate $R_z(\theta)$ where $\theta = 1.4726 \times 10^6$. This kind of module does not contain calling other modules and does not require more local qubits. Therefore, the module itself can be included in the parent module calling this module.}
\end{figure}

A modular QASM for a non-trivial quantum algorithm is usually composed of at least tens of thousands modules. All modules can be classified into terminal modules and non-terminal modules regarding whether they call other modules or not. By drawing a module hierarchy tree, it is possible to show the relation of the calling (parent) module and the called (child) modules. The terminal modules then place at the leaf of the hierarchy tree. Depending on quantum algorithms and their implementation, some terminal modules may take only one-parameter qubit. For example, some quantum algorithms exploit $R_z(\theta)$ gates frequently. By the way, since such rotational gates for an arbitrary rotational angle cannot be implemented in a fault-tolerant manner, they have to be decomposed into a sequence of fault-tolerantly implementable gates. This decomposition can be defined as a one-parameter-qubit module in the quantum algorithm.

Such module works as outputting the sequence of fault-tolerant gates from the input $R_z(\theta)$ gate (see Fig. 8). Therefore, it is possible to treat the module in an inlining way without treating it as a normal module requiring a separated physical space. By following the above concept, in the present work, we do not allocate separate physical space...
for the one-parameter-qubit terminal modules. Needless to say, allocating the physical space raises space (more qubits) and time overhead (qubit movements). Empirically, the number of one-parameter-qubit terminal modules and the frequency of calling such modules are not small. Please note that for a multi-parameter-qubit terminal module, we have to allocate separate physical space and perform the mapping of the module on that place to keep the heart of the proposed mapping.

3 Discussion

We discuss the strengths of the proposed system mapping of the modular QASM compared to the mapping of the non-modular QASM. First, it is possible to perform the mapping for a large-sized quantum algorithm. As mentioned above, for the same quantum algorithm, the size of the modular QASM is much smaller than that of the non-modular QASM. Therefore, it becomes possible to generate and manage an assembly code for a large-sized quantum algorithm (see Fig. 3). Besides, practically it requires less compile time to generate the modular QASM than the non-modular QASM [14].

Second, it takes much smaller time to perform the mapping. This is due to both the less-sized modular QASM and the one-time module mapping. In Sect. 1, we compared the size complexity between two types of QASMs per a specific composite quantum function. The time complexity of each mapping nearly follows the size complexity.

Figure 9 compares the mapping time between both QASMs. We have implemented both mappings with Python and tested them on a computer system of 3.5 GHz CPU and 128 GB Memory. The mapping time for the non-modular QASM mapping is estimated from the statistics obtained from both of the non-modular mappings of small algorithm cases (see Table 3) and the modular mapping, \( \sum \text{freq}_M \times \text{time}_{\text{net}}^M \) over all the modules \( M \) in the QASM. Note that \( \text{freq}_M \) is the frequency of calling the module \( M \), and \( \text{time}_{\text{net}}^M \) is the net time for mapping the module \( M \). The mapping for Shor N=512 with the non-modular QASM requires about 1500 days (1.32 \times 10^8 sec), but the proposed mapping can be completed in around 1 hour (3.96 \times 10^3 sec).

Besides, finding the shortest qubit path is another big issue related to such a difference in time between both mappings. In terms of the complexity for finding the shortest path for qubit move to perform an arbitrary CNOT gate, that of the modular mapping is much smaller than that of the non-modular mapping. For a CNOT gate, in the non-modular mapping, the whole qubit layout is the target space for searching the shortest path, but in the modular mapping, the current module is the target qubit layout, not the whole system. Please note that the size of the module is much smaller than the whole system (please see Avg. Module Size in Table 3). For example, in the non-modular mapping, for Shor-16, we need to form a graph of 182 nodes and search the shortest path on the graph. Besides, we have to iterate the job as much as the number of CNOT gates in the algorithm. For reference, for Shor-16, the number of CNOT gates is 2.6 \times 10^6 [14]. On the other hand, in the modular mapping, we need a graph of only 11 nodes on average for the same job. Please note that the number of the iterations is the same with the non-modular mapping. As the input size is increased, forming a graph and finding a shortest path in the graph become practically more challenging.
Fig. 9 The comparison in the execution time between the modular mapping and the non-modular mapping

The above-mentioned is the reason why we apply Dijkstra’s shortest path search algorithm in the present work. As mentioned before, it is possible to find more efficient quantum computing circuit by applying A* or Floyd–Warshall algorithm. But it requires much more time and memory than our report. It will be likely that the degree of the required resource will be larger for the non-modular mapping due to the above-mentioned situation. Then, the time for mapping Shor-512 with the non-modular QASM will be increased as much as several orders of magnitude from 1500 days. For reference, practically it is not easy to find an optimal quantum circuit of even the algorithm of a dozen of qubits [8].

Third, it is possible to analyze and optimize the implementation of a quantum algorithm. Through the proposed mapping, we can find out the critical (maybe the most frequently called) modules of the algorithm. The pre-analysis of the modular QASM, which may be required for optimizing the qubit mapping, can be done very efficiently than the non-modular QASM. This is because the modular QASM is functionally structured in addition to its small size. In this work, we optimize a qubit placement [6,21,22] by exploiting the data obtained from scanning the QASM in advance. In doing so, the number of SWAP operations required to perform a CNOT gate between the qubits placed at a distance can be reduced. In this work, we apply the linear programming approach [21] for Shor and BWT (Binary Welded Tree) [14] algorithms. The degree of the improvement depends on a quantum algorithm as shown in Table 2. We observed that the reduction of the circuit depth in the case of Shor algorithm is very negligible (less than 0.01%), but the circuit depth for BWT is reduced as much as 1% ~ 6%, i.e., $2.42 \times 10^5 \rightarrow 2.26 \times 10^5$ for BWT-10.

We now discuss the weakness of the proposed mapping in terms of quantum resources. Our mapping assumes a hierarchically structured quantum computer composed of multiple computing regions and a communication bus. Such a structured system architecture requires more physical space, i.e., qubits. As shown in Table 3, on average, 2.2 times more computing qubits are necessary for the structured system, and
### Table 2
The reduction of the circuit depth for Shor algorithm and BWT algorithm by applying linear programming optimization for the qubit placement

| Algorithm | Circuit depth | Non-optimized | Optimized |
|-----------|---------------|---------------|-----------|
| Shor \( N = 128 \) | \( 8.0042501 \times 10^{10} \) | \( 8.0042403 \times 10^{10} \) |
| Shor \( N = 256 \) | \( 1.0481693 \times 10^{12} \) | \( 1.0481684 \times 10^{12} \) |
| Shor \( N = 512 \) | \( 1.5098382 \times 10^{13} \) | \( 1.5098381 \times 10^{13} \) |
| BWT \( N = 10 \) | \( 2.42 \times 10^5 \) | \( 2.26 \times 10^5 \) |
| BWT \( N = 100 \) | \( 3.45 \times 10^6 \) | \( 3.30 \times 10^6 \) |
| BWT \( N = 1000 \) | \( 7.60 \times 10^7 \) | \( 7.46 \times 10^7 \) |

### Table 3
Summary of performance of three kinds of quantum computing in small scale

| Performance category | Shor-4 | Shor-8 | Shor-16 |
|----------------------|--------|--------|---------|
| Ideal QC (Non-mapping) | Qubit | 46 | 90 | 178 |
| Qubit (layout size) | Circuit depth | \( 3.61 \times 10^6 \) | \( 1.91 \times 10^7 \) | \( 7.19 \times 10^7 \) |
| Circuit depth | \( 4.92 \times 10^6 \) | \( 5.52 \times 10^7 \) | \( 2.40 \times 10^8 \) |
| Total SWAP | \( 8.73 \times 10^4 \) | \( 1.04 \times 10^6 \) | \( 2.32 \times 10^7 \) |
| Avg. Move distance | 4.42 | 6.14 | 8.81 |
| Mapping time (s) | \( 4.7 \times 10^2 \) | \( 8.6 \times 10^3 \) | \( 9.4 \times 10^4 \) |
| Memory (MB) | \( 4.1 \times 10^3 \) | \( 3.4 \times 10^4 \) | \( 1.5 \times 10^5 \) |
| Non-modular Modular Computing qubits | 105 | 204 | 401 |
| Bus qubit | 204 | 627 | 1690 |
| Circuit depth | \( 4.25 \times 10^7 \) | \( 2.58 \times 10^8 \) | \( 1.08 \times 10^9 \) |
| Total SWAP (module) | 1752 | 7248 | 59,072 |
| Avg. SWAP (module) | 0.02 | 0.01 | 0.01 |
| Total SWAP (bus) | \( 1.37 \times 10^6 \) | \( 2.02 \times 10^7 \) | \( 2.97 \times 10^8 \) |
| Avg. move distance | 0.043 | 0.021 | 0.022 |
| Module | 136 | 245 | 334 |
| Avg. module size | 4.04 | 5.37 | 10.77 |
| Module calls | \( 7.92 \times 10^4 \) | \( 6.39 \times 10^5 \) | \( 3.39 \times 10^6 \) |
| BUS size (length×width) | \( 17 \times 12 \) | \( 33 \times 19 \) | \( 65 \times 26 \) |
| Mapping time (s) | \( 6.0 \times 10^{-1} \) | \( 1.4 \times 10^0 \) | \( 3.4 \times 10^0 \) |
| Memory (MB) | \( 3.8 \times 10^0 \) | \( 8.2 \times 10^0 \) | \( 1.4 \times 10^1 \) |

The first is based on the ideal setting, and the second and the third are, respectively, by the non-modular mapping and the modular mapping.
| Performance items | Explanation |
|------------------|-------------|
| Ideal QC Qubit   | The number of qubits used in a quantum algorithm |
| Circuit depth    | The length of a quantum circuit where a two-qubit gate CNOT is allowed between an arbitrary pair of qubits |
| Non-modular Qubit (layout size) | The size of 2-dimensional rectangular physical space required to run the algorithm |
| Circuit depth    | The length of a quantum circuit including multiple qubit movement (SWAP) to keep the locality |
| Total SWAP       | The number of SWAP gates introduced during the non-modular mapping |
| Avg. Move distance | The average qubit move distance required to run a CNOT gate of the algorithm |
| Mapping time (secs) | The CPU wall time to perform the non-modular mapping |
| Memory (MB)      | The memory usage to perform the non-modular mapping |
| Modular Computing qubits | The sum of all the computing areas, each of which is formed in the 2-dimensional rectangle |
| Bus qubit        | The communication area that connecting all the computing areas |
| Circuit depth    | The length of the whole quantum circuit including the sub-circuits for both the computation conducted in the computing area and the communication between computing areas (module) |
| Total SWAP (module) | The sum of all the SWAP gates required during the computing in the computing areas (module) |
| Avg. SWAP (module) | The average number of SWAP gates required to run a module |
| Total SWAP (Bus) | The sum of all the SWAP gates required for qubit passing through the bus |
| Avg. Move distance | The average qubit move distance required to run a CNOT gate of the algorithm |
| Module           | The number of modules in a quantum algorithm |
| Avg. module size | The average number of qubits in a module |
| Module calls     | The total number of module calls during the algorithm execution |
| BUS size (length×width) | The size of the communication (qubit passing) area, this value coincides with the Bus Qubit. |
| Mapping time (s) | The CPU wall time to perform the proposed modular mapping |
| Memory (MB)      | The memory usage to perform the proposed modular mapping |
Table 5 Summary of performance of large-scale quantum computing (Shor-128, 256, 512)

| Performance category          | Shor-128 | Shor-256 | Shor-512 |
|-------------------------------|----------|----------|----------|
| Modular Computing qubits      | 3016     | 6017     | 11,886   |
| Bus qubit                     | 37,449   | 104,550  | 295,056  |
| Circuit depth                 | $8.00 \times 10^{10}$ | $1.05 \times 10^{12}$ | $1.51 \times 10^{13}$ |
| Total SWAP (module)           | $1.22 \times 10^{7}$   | $7.10 \times 10^{7}$   | $3.98 \times 10^{8}$   |
| Avg. SWAP (module)            | 0.009     | 0.006    | 0.005    |
| Total SWAP (Bus)              | $1.13 \times 10^{12}$ | $1.75 \times 10^{13}$ | $2.76 \times 10^{14}$ |
| Avg. Move distance            | 0.043     | 0.021    | 0.022    |
| Module                        | 136       | 245      | 334      |
| Avg. Module size              | 74.4      | 147.01   | 333.95   |
| Module calls                  | $1.35 \times 10^{9}$  | $1.09 \times 10^{10}$ | $8.78 \times 10^{10}$ |
| BUS size (length×width)       | 513×73    | 1025×102 | 2049×144 |
| Mapping time (s)              | $1.78 \times 10^{2}$ | $8.22 \times 10^{2}$  | $3.96 \times 10^{3}$  |
| Memory (MB)                   | $2.8 \times 10^{2}$   | $1.01 \times 10^{3}$  | $4.09 \times 10^{3}$  |

The algorithms in this input scale cannot be treated by the conventional non-modular mapping method in addition, bus qubits are additionally required. Please note that the number of bus qubits is determined by the specific system architecture and the algorithm size (see Table 3). Needless to say, for a larger system more qubit movements are required.

We just mentioned that more qubit movements are required for the proposed modular QASM mapping. However, we observed that surprisingly the length (circuit depth) of a quantum computing is not increased as much as the number of the increased SWAP gates. This is because most of the SWAP gates are exploited for the qubit passings between modules, which can be performed in parallel. Please note that for the qubit passings, all the parameter qubits are moved to the destination simultaneously. As shown in Table 3, the difference in the number of SWAP gates between mappings is decreased along with the input size. With this tendency, we guess that in the large-scale quantum computing, the difference in the number of qubit movements will be much smaller or negligible in the best case.

We need to say that for a CNOT gate over arbitrary two qubits, the modular mapping requires fewer qubit movements. Along with the algorithm input size, the size of the qubit layout is gradually increased regardless of the mapping type. As shown in Table 3, both Qubit (Layout Size) for the non-modular mapping and Avg. Module Size for the modular mapping are, respectively, increased. Then, the number of SWAP gates is increased in the non-modular mapping based on the monolithic qubit layout. On the other hand, for the modular mapping based on the distributed structured qubit layout, surprisingly it does not. This is because, in the modular mapping (and therefore structured architecture), the qubits for mutual interaction are usually localized within a module, but in the non-modular mapping they are located spread over the whole layout. As shown in Table 3, for Shor-4, 8, 16, the proposed mapping only requires on average 0.04, 0.02, and 0.02 SWAP gates for a CNOT gate, whereas the non-modular mapping requires 4.42, 6.14, and 8.81 SWAP gates. As the input size is increased, the
In this regard, we believe that, in the large-scale quantum computing regime, the proposed mapping makes a quantum computing can be done with less running time. Furthermore, the difference in the number of qubits between both mappings can be reduced by controlling the area of the communication bus, in particular the bandwidth of the bus. Reducing the bandwidth may raise the delay of SWAP gates, but in the very rare cases. Empirically, the qubit passing requiring the full bandwidth of the bus does not happen frequently.

4 Summary of performance results

In this section, we summarize the performance of the proposed modular mapping discussed in the previous section. Table 3 first shows the performance of quantum computing in the ideal setting, which can be directly obtained by analyzing quantum algorithm only. It then compares the performance of the quantum computing models based on non-modular mapping and modular mapping in the small input size regime. Please note that non-modular mapping is applicable to this regime. The meaning of
Fig. 11 A flowchart of the proposed modular QASM-based synthesis. Local Synthesizer synthesizes each module, and doubled line arrows outside the box indicate the global management of hierarchical synthesis. Dotted line indicates the intermediate data flow. Stack is used to keep the information about the synthesis-paused modules.

the performance metric used in the table is described in Table 4. Lastly, we show the performance of large-scale quantum computing (Shor-128, 256, and 512) in Table 5.

5 Conclusion

We have proposed an efficient circuit mapping methodology for a large-scale quantum computing. As mentioned several times, it is practically infeasible to deal with quantum computing in the large-scale regime with the conventional method based on a non-modular QASM. To overcome the problems, we turn our attention to structured description about quantum algorithm and perform the mapping the modular code onto
a hierarchically structured system architecture. Thanks to modularization, the system mapping in the large-scale regime can be carried out very efficiently. In the present work, we only have dealt with the modular QASM by the open-source compiler ScaffCC, but can be applied to any other structured QASM such as openQASM [13].

Our future work is to apply the most realistic quantum computer architecture [23] and to improve the practicality of the mapping algorithm. The present work assumes all qubits work concurrently. However, in reality, some qubits take longer time for error correction and/or non-transversal logical gates, and therefore, the system code from the present work may not work as expected.

A communication bus has to be controlled more delicately. Currently, we assume the bus works on demand and qubit passings are done on time. But, we believe that in practice network traffic may disturb such ideal communication, and the bus may be in congestion, in particular for highly parallel quantum application [24].

Lastly, the proposed mapping currently uses many qubits, but majority of them just wait from calling other modules during the most of the execution time. In future work, we will seriously consider how to improve the spatial efficiency of the mapping algorithm with keeping the heart of the proposed mapping (Figs. 10, 11).

Acknowledgements This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIP) (No. 2019-0-00003, Research and Development of Core technologies for Programming, Running, Implementing and Validating of Fault-Tolerant Quantum Computing System) and Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (18ZH1400, Research and Development of Quantum Computing Platform and its Cost-Effectiveness Improvement).

A illustration of module allocation

In this section, we conceptually illustrate the module allocation of the algorithm Shor-4. For that, we prepared an edited QASM (see Fig. 12). Besides, we also assumed that all the module has the same size, but in reality, it is not. Before the illustration, we need to remind readers that running a quantum algorithm written in the hierarchically structured QASM begins with the execution of the first instruction of the main module and ends with the last instruction of the same module.

1. At the beginning, the space for only main module is allocated (see line no. 98 in Figs. 12 and 13a).
2. When AQFT module is called (line no. 101), the space for the module is newly allocated and the communication bus is created (see Fig. 13b).
3. When CU_IP0_IPx_IPx_IPx_DPx_DPx_DPx_DPx module is called (line no. 90), the space for the module is also newly allocated and the existing communication bus is extended to cover the new space (Fig. 13c).
4. By calling QMAC_MOD2 module (line no. 72), the space is allocated (Fig. 13d).
5. In QMAC_MOD2, QFT_2 module is called (line no. 56, Fig. 13e).
6. By calling cRz_IPx_IPx_IPx_IPx_DP3_1416_DPx_DPx_DPx (line no. 12), the physical space is allocated for the module (Fig. 13f).
7. The above-mentioned module is composed of three DecomposeRotation modules and 2 CNOT gates (see lines no. 3-9). As mentioned before, since the Decom-
Hierarchical system mapping for large-scale fault-tolerant

Fig. 12 An edited version of QASM for Shor-04 for illustrating the module allocation process
Fig. 13 An example of the conceptual module allocation sequence during execution of Shor-4. The module with bold font indicates the working module at the present time. Please note that in each figure except (a) and (o) the center space denoted by “BUS” indicates the shared communication bus connecting all the computing regions.
poseRotation module is a one-parameter-qubit terminal module, we do not need to allocate the separated space for the module.

8. Even though we have not shown in Fig. 12, the module cRz_IPx_IPx_IPx_IPx_DP0_0491_DPx_DPx_DPx (line no. 14) is also composed with only one-parameter-qubit terminal module and quantum gates. Therefore, we do not allocate a physical space for the module (Fig. 13g).

9. The module allocation process of iQFT_2 module (lines no. 48–54) is similar to that of QFT_2 (Fig. 13h–k).

10. After the execution of iQFT_2 module (line no. 53), the execution control is back to the module QMAC_MOD2 (Fig. 13l).

11. The last module to execute in AQFT is CRinv_IP2_IPx_IPx_IPx_DPx_DPx_DPx_DPx (line no. 92, Fig. 13m).

12. After running the module CRinv_IP2_IPx_IPx_IPx_DPx_DPx_DPx_DPx and several quantum gates, the execution of the module AQFT is completed and then the control is back to the main module with the space de-allocation (Fig. 13n, o).

References

1. IBM: IBM Q Experience. https://quantumexperience.ng.bluemix.net/qx/experience
2. Rigetti: Rigetti Computing. https://www.rigetti.com/index.php/forest
3. IONQ: IONQ https://ionq.com
4. Preskill, J.: Quantum computing in the NISQ era and beyond. Quantum 2, 79 (2018)
5. Saeedi, M., Wille, R., Drechsler, R.: Synthesis of Quantum Circuits for Linear Nearest Neighbor Architectures pp. 1–14 (2012)
6. Pedram, M., Shafaei, A.: Layout optimization for quantum circuits with linear nearest neighbor architectures. IEEE Circ. Syst. Mag. 16(2), 62–74 (2016)
7. Zulehner, A., Paler, A., Wille, R.: An efficient methodology for mapping quantum circuits to the IBM QX architectures. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 38(7), 1226–1236 (2019)
8. Li, G., Ding, Y., Xie, Y.: Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In: The Twenty-Fourth International Conference, pp. 1001–1014. ACM Press, New York, USA (2019)
9. Guerreschi, G.G., Park, J.: Two-step approach to scheduling quantum circuits. Quant. Sci. Technol. 3(4), 045003 (2018)
10. Murali, P., Baker, J.M., Abhari, A.J., Chong, F.T., Martonosi, M.: Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In: The Twenty-Fourth International Conference, pp. 1015–1029. ACM Press, New York, USA (2019)
11. Tannu, S.S., Qureshi, M.K.: Not All Qubits Are Created Equal. In: The Twenty-Fourth International Conference, pp. 987–999. ACM Press, New York, USA (2019)
12. Lao, L., van Someren, H., Ashraf, I., Almudever, C.G.: Timing and resource-aware mapping of quantum circuits to superconducting processors (2019). https://arxiv.org/abs/1908.04226
13. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open Quantum Assembly Language. pp. 1–24 (2017). https://arxiv.org/abs/1707.03429
14. JavadiAbhari, A., Fatini, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: ScaffCC: scalable compilation and analysis of quantum programs. Parallel Comput. 45(C), 2–17 (2015)
15. Svore, K.M., Aho, A.V., Cross, A.W., Chuang, I.L., Markov, I.L.: A layered software architecture for quantum computing design tools. Computer 39(1), 74–83 (2006)
16. Khanna, N., Guerreschi, G.G., Ashraf, I., Hoboaboom, J.W., Almudever, C.G., Bertels, K.: cQASM v1.0: Towards a Common Quantum Assembly Language pp. 1–8 (2018)
17. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper. In: The 34th ACM SIGPLAN conference, pp. 333–342. ACM Press, New York, New York, USA (2013)
18. JavadiAbhari, A.: ScaffCC/ScaffCC. https://github.com/ScaffCC/ScaffCC
19. Pavlidis, Archimedes; Gizopoulos, Dimitris: Fast quantum modular exponentiation architecture for Shor’s factoring algorithm. Quant. Inf. Comput. 14(7 & 8), 649–682 (2013)
20. Ross, N.J., Selinger, P.: Optimal ancilla-free Clifford+T approximation of z-rotations. Quant. Inf. Comput. 16(11 & 12), 901–953 (2016)
21. Shafaei, A., Saeedi, M., Pedram, M.: Qubit placement to minimize communication overhead in 2D quantum architectures. In: 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 495–500. IEEE (2014)
22. Siraichi, M., Dos Santos, V.F., Collange, S., Pereira, F.M.Q.: Qubit Allocation. In: The 2018 International Symposium pp. 1–12–13 (2018)
23. Lin, C.C., Sur-Kolay, S., Jha, N.K.: PAQCS: physical design-aware fault-tolerant quantum circuit synthesis. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 23(7), 1221–1234 (2015)
24. Javadi-Abhari, A., Gokhale, P., Holmes, A., Franklin, D., Brown, K.R., Martonosi, M., Chong, F.T.: Optimized surface code communication in superconducting quantum computers. In: The 50th Annual IEEE/ACM International Symposium, pp. 692–705. ACM Press, New York, USA (2017)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.