


# Integrated Analysis of Performance and Resources in Large-Scale Quantum Computing

Yongsoo Hwang<sup>1</sup>, Taewan Kim, Chungheon Baek, and Byung-Soo Choi<sup>1\*</sup>  
*Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea*

 (Received 30 August 2018; revised manuscript received 20 March 2020; accepted 14 April 2020; published 14 May 2020)

To ascertain the feasibility of large-scale quantum computing, the performance and quantum resource of a practical quantum-computing situation needs to be analyzed. However, most of the analyses reported so far have focused on the statistical examination that simply calculates the performance and resource based on individual quantum-computing components. In this work, we propose an integrated analysis methodology that models a large-scale fault-tolerant quantum-computing system based on three components: *algorithm*, *error correction*, and *device*. Furthermore, to implement the proposed methodology, we develop a quantum-computing software platform composed of three functional layers: *compile*, *system*, and *building block*. By using our platform, we observe that it takes  $8.78 \times 10^5$  h, which is much longer than previous estimations, for factoring a 512-bit integer with Shor's factoring algorithm. We also discuss whether the proposed platform can play a significant role in finding an optimal concatenation level and/or code distance of quantum error-correcting code.

DOI: [10.1103/PhysRevApplied.13.054033](https://doi.org/10.1103/PhysRevApplied.13.054033)

## I. INTRODUCTION

In recent decades, diverse quantum-computing components from applications to physical devices have been actively explored and developed. Some of them already have theoretically optimal performance [1–6]. Gigantic IT corporations have announced that they succeeded in the development of dozens of qubit systems [7–9]. Even they expect to see a thousand qubit system within ten years. In addition, several startups began to develop quantum-computing hardware and software [10]. It seems that the era of quantum computing is gradually arriving.

So far, to assess the feasibility of large-scale quantum computing [11], some efforts have been devoted to investigating the amount of quantum resource needed [12–18] and the expected performance [16, 19–21]. Based on such an analysis, the security of a conventional cryptography against theoretically superfast quantum computing has also been studied [22, 23]. We believe that the analysis results reported so far are practically less accurate because the analysis methodologies they applied were theoretical statistical examinations based on the performance data of individual quantum-computing components, or some practical components are missed out. The statistical examination is nothing more than a simple arithmetic calculation with the data from individual components, and therefore some situations that may happen during quantum computing cannot be exactly taken and considered.

For practically accurate analysis, we need to take account of a practical quantum-computing situation. To this end, we first need to prepare quantum-computing components that can be applied in practice. For example, a quantum-computing algorithm has to be prepared as the list of individual and physically implementable quantum gates instead of a logical and abstract description itself. For that, we write a quantum-computing program code and then decompose (or compile) the code. By doing so, we can prepare a quantum algorithm to be executed in practice.

Second, provided that all quantum-computing components are prepared for practical quantum computing, we have to integrate all those components properly as if we really execute the algorithm. In the integration, we first associate algorithm qubits with physical qubits (*qubit mapping*) and then control the application order of algorithm gates in accordance with logical dependency (*gate scheduling*). By the system synthesis, we are able to implement a quantum algorithm for the given quantum-computing system with a specific physical and logical structure (or limitation).

The performance analysis based on the system synthesis can deal with some dynamic situations that cannot be treated with theoretical analysis methods applied in previous works. For example, qubit technologies such as superconductors and quantum dots allow interactions between nearest-neighbor qubits only. Therefore, to execute a two-qubit operation between distant qubits, a qubit must be moved to the neighboring position of the other qubit. Here,

\*bschoi3@etri.re.kr

the detailed sequence and the number of qubit movements can only be exactly determined with the system synthesis, not with the theoretical approach.

As we mention above, we think that it is possible to analyze the performance and the quantity of quantum resources for quantum computing more accurately by taking the practical situation into consideration. In this work, to perform such an analysis more efficiently, we propose and build up a quantum-computing platform composed of three functional layers with a well-defined role as follows.

(a) Compile layer: decomposition of a quantum algorithm into a sequence of quantum gates, called a *quantum-assembly code* (or QASM).

(b) System layer: integration of a quantum algorithm (quantum-assembly code) and building blocks under a target quantum-computing system architecture via a qubit mapping and a gate scheduling.

(c) Building-block layer: implementation of logical qubits and gates according to a chosen quantum error-correcting code and a fault-tolerant scheme.

The aim of this work is to estimate the most realistic and accurate quantum-computing performance and resource with the help of the platform we develop. For that, we have to create a quantum-computing program of the quantum algorithm we want to analyze, and at the same time, configure a quantum-computing system by choosing a specific fault-tolerant protocol, the architectures of system and quantum chip, and device performance. Then, the platform analyzes the performance and the quantum resource of the quantum-computing system running the quantum algorithm.

With the proposed platform, it is possible to estimate the performance of quantum-computing models in diverse

situations. First, we can see which quantum-computing component affects the performance of quantum computing most seriously. Second, we can see what happens in quantum computing if an individual component is improved. Third, it is also possible to see that a theoretically optimal component really leads to optimal quantum computing or rather works suboptimally in composition with other components. Finally, this work provides numerical data for theoretical conjectures in fault-tolerant quantum computing (FTQC), i.e., a trade-off between reliability and performance.

The remainder of this work is organized as follows. Section II reviews related works. Section III overviews the proposed quantum-computing framework and describes how to configure and analyze quantum computing with the framework. Section IV describes each layer of the proposed framework in detail. Section V describes the performance metric we evaluate in this work. The analysis results are shown in Sec. VI, and Sec. VII discusses, by exploiting the proposed framework, what happens in quantum computing if an individual quantum-computing component is improved. This work concludes with some discussions in Sec. VIII.

## II. RELATED WORKS

We review several quantum-computing frameworks discussing the performance and the quantum resource of quantum computing. Table I briefly summarizes the works.

Quipper [13,24,27] and Scaffold [17,25,26] are frameworks for quantum compile and resource estimation of a quantum algorithm. They basically compile a programmed quantum algorithm into a sequence of quantum instructions composed of a quantum gate and target qubit(s). From the compiled results, it is possible to statistically analyze the quantum resource such as the number of gates and

TABLE I. Summary of the related works. Note that “ $\Delta$ ” indicates that the component is partially applied. For example, the analyses of Refs. [19] and [20] are based on only the dominant part of a quantum algorithm ( $T$  gate and quantum adder), not covering all quantum gates. Besides, for the surface-code error correction, a two-dimensional qubit layout with nearest-neighbor qubit interaction is basically assumed as microarchitecture.

	Compile	FTQC	Microarchitecture (Qubit layout)	System Architecture and synthesis	Analysis Criterion
Quipper [13,24] Scaffold [17,25,26]	○	X	X	X	One time
Fowler <i>et al.</i> [19] Jones <i>et al.</i> [20] Van Meter <i>et al.</i> [21]	$\Delta$	Surface	$\Delta$	X	One time
Reiher <i>et al.</i> [14] QuRE [12,16]	○	Surface Steane, Bacon-Shor, Surface	$\Delta$ Layout of physical qubits	X X	One time One Time
Present work	○	Steane, Surface	Layouts of physical and logical qubits, Communication bus, computing regions	○	Fidelity 100%

qubits. All the analysis results come from both the quantum algorithm and the compile method.

Fowler *et al.* [19] approximated the quantum-computer size and the execution time of Shor's factoring algorithm ( $N = 2000$ ) with a surface-code quantum computing. Their analysis method was a completely theoretical approach based on the dominant part of the factoring algorithm, a logical  $T$  gate. The execution time of the factoring algorithm is only dependent on the  $T$  depth and the performance of a logical  $T$  gate, and by a time-optimal quantum-computing scheme [28] they applied, the logical  $T$  gate is completed within a physical measurement time ( $M_t$ ),  $T_{\text{depth}} \times M_t$ . Therefore, as they claimed, their estimation on the running time of the factoring algorithm is much shorter than previous reports.

Van Meter *et al.* [21] proposed a distributed quantum-computing architecture and estimated the resource and the performance of Shor's factoring algorithm on the architecture. Their analysis method focuses on a quantum adder only. They calculated the execution time of the algorithm by multiplying the running time of the quantum adder and the depth of the adder, which is the quantity of how much quantum adders are executed in series. The critical factor of their analysis is the running time of a Toffoli gate, which is the dominant gate of the quantum adder.

Jones *et al.* [20] proposed a layered quantum-computer architecture in terms of fault-tolerant logical-gate implementation, but analyzed the quantum-computing time by exactly following Ref. [21], but with an improved component. They improved the performance of a Toffoli gate within their quantum-dot-based quantum-computer architecture, and therefore reduced the quantum-computing time more than Ref. [21] for the same factoring algorithm.

Note that the above-mentioned Refs. [19–21] focused on the dominant parts of the factoring algorithm only, which can be found out from the theoretical foundation of the algorithm. Therefore, for the performance analysis, they did not need to decompose the algorithm fully into the individual quantum gates. We believe such approaches are applicable due to the well-known structure of the factoring algorithm and the system-oriented surface-code architecture. In other words, their approach may not be applied to analyze other less-studied and more complicated quantum algorithms, and/or FTQC based on concatenated quantum error-correcting codes. In this regard, we can say that Reiher *et al.* [14] may be general because they hired a quantum compiler and therefore took other quantum algorithms. Reiher *et al.* [14] estimated the quantum resource and the quantum-computing time for quantum simulations of several complex chemical systems. They analyzed the performance of the algorithm by considering all the quantum instructions decomposed by a compiler and the surface-code fault-tolerant quantum-computing protocol.

Unlike the above-mentioned works, the toolbox QuRE [12,16] covers Steane code and Bacon-Shor code besides

surface code for the first time. Besides, QuRE takes most of the quantum-computing components such as compile, physical qubit layout, and device performance. In this regard, we believe this toolbox performed the analysis based on more quantum-computing components than used previously. However, their analysis method did not consider the practical execution of quantum computing. They just applied a statistical examination based on the individual quantum-computing components. For example, their estimation to the algorithm execution time is defined as the number of quantum gates and the execution time of each gate,  $\sum 1/g_{\text{parallel}} \times \mathcal{N}g \times g_t$ . Note that  $g_{\text{parallel}}$  denotes the amount of gates  $g$  executed in maximally parallel and  $g_t$  is the execution time of the gate. Therefore, it is difficult to comment on whether their analysis results may coincide with a practical situation.

### III. CONFIGURATION OF QUANTUM-COMPUTING SYSTEM

We overview the structure and functionality of the proposed quantum-computing platform and describe how to perform the analysis by configuring a quantum-computing system. We also describe which schemes and protocols are currently supported by the platform.

#### A. Overview of platform

The proposed quantum-computing platform deals with programming, compile, computer architecture, fault-tolerant scheme, and device. To this end, it is composed of three functional layers: *compile*, *system*, and *building block*. Each layer has a well-defined role and provides several options to conduct their functions. All the layers are closely related to each other. Figure 1 shows the data flow between the layers.

The *compile layer* deals with programming and compiling a quantum algorithm. A quantum algorithm is written in a high-level programming language, and is compiled into the sequence of the quantum instructions, called a *quantum-assembly code*. Figure 2 shows the input and output of the compile layer. In the present work, we hire an open-source quantum-computing compiler *ScaffCC* [17, 25,26] that supports programming language *Scaffold* [29]. The details about quantum compile, quantum gates, and a quantum-assembly code is described in Sec. IV A. Why we exploit ScaffCC for large-scale quantum computing is also discussed in that section.

The *system layer* carries out a system synthesis, which integrates a quantum algorithm, a quantum-computer system architecture, and building blocks. Besides, this layer treats almost everything required to run a quantum algorithm on a target quantum-computer system. For example, by performing the system synthesis, the quantum algorithm is reconfigured for the target quantum-computing system architecture. Figure 3 describes the

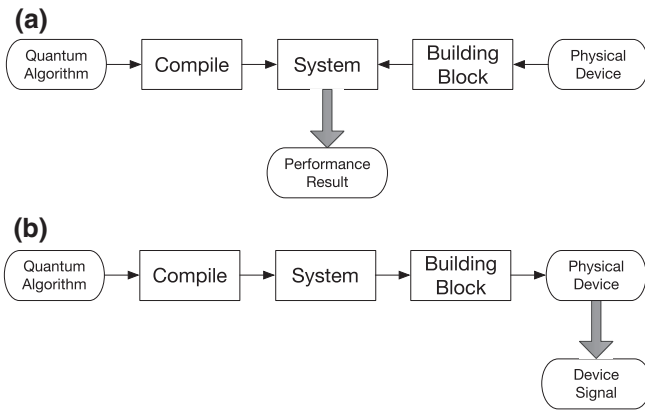


FIG. 1. Data flow over the layers. (a) For the performance analysis, the compile layer and the building-block layer respectively provide a quantum-assembly code and the performance of logical operations to the system layer. The system layer then performs the performance analysis via the system synthesis. (b) The platform can be used to perform practical quantum computing. For that, user input data flow sequentially from the most top layer to the bottom layer, in each layer, the data is properly translated suited for the next layer. Finally, a control signal to manipulate physical devices according to the user input has to be generated in the bottom layer.

input and output of the system layer. For the performance analysis, the inputs to the system layer are the quantum algorithm, fault-tolerant protocols, and quantum device in the proper format, and the output is the estimated performance and the quantity of the quantum resource.

The *building-block layer* is associated with the qubits and gates in the quantum algorithm. Since this work basically assumes fault-tolerant quantum computing, the qubits, and gates described in the algorithm correspond to logical qubits and gates encoded in a quantum error-correcting code. In this regard, the main functionality of

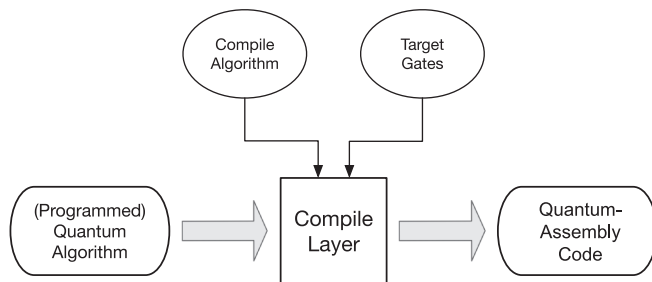


FIG. 2. Input and output in the compile layer. By a compile, a programmed quantum algorithm is decomposed into a quantum-assembly code. For the compile, a compile algorithm and target gates have to be determined beforehand. Target gates can be varied according to quantum-computing type such as fault-tolerant quantum computing or nonfault-tolerant (physical) computing. The selection of target gates can also be influenced by a qubit technology.

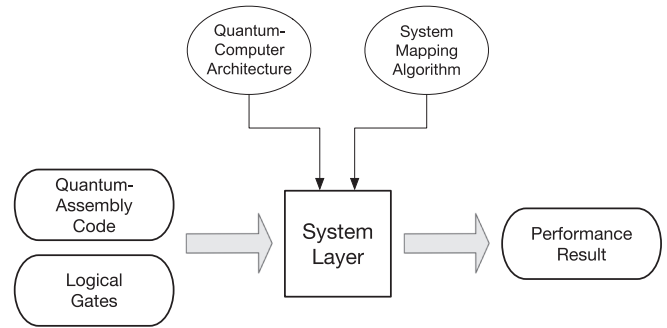


FIG. 3. Input and output of the system layer. Quantum-assembly code and the performance of logical building blocks are passed from the compile layer and the building-block layer, respectively. A quantum-computer architecture describes the layout of logical and physical qubits and a communication bus over qubits. A system synthesis algorithm describes how to integrate a quantum algorithm (quantum-assembly code) and a quantum-computer architecture via a qubit mapping and a gate scheduling.

the layer is to assemble physical qubits and gates to implement logical qubits and gates. Figure 4 shows the input and output of the building-block layer. In terms of the performance analysis, the output of the building-block layer is the performance of logical quantum operations, *time* and *fidelity*. Note that the proposed platform supports  $[[7, 1, 3]]$  Steane code and a double-defect-based surface code. In Sec. IV C, we describe the fault-tolerant logical-gate protocols in detail.

### B. Configuration of quantum computing

We describe how to configure quantum computing with the proposed platform. As mentioned above, it is possible to configure quantum computing by selectively choosing specific protocols or the properties of a physical device. For example, you can configure the Steane-code-based

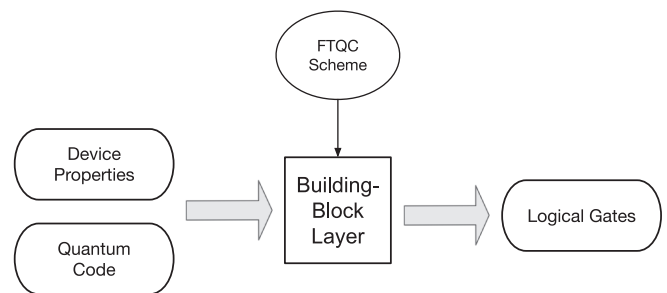


FIG. 4. Input and output in the building-block layer. To make logical qubits and gates, a quantum error-correcting code should be determined first. Besides, to derive the performance of logical operations, the properties of a physical device, time and fidelity, have to be considered. Then, based on the physical device property and the fault-tolerant quantum-computing scheme, logical operations with a specific performance are generated.



TABLE II. List of protocols and layouts currently supported by the framework. In the compile layer, we choose a compile type and a target gate set. The compile type is the format of a quantum-assembly code. The compile type is closely related to the mapping type and qubit layout. A specific qubit layout can be chosen within a selection. In the building-block layer, the scheme for a fault-tolerant quantum computing is determined. According to the scheme, the protocol for each fault-tolerant logical operation is fixed.

Layer	Options	Values
Compile	Compile type	Structured code, nonstructured code
	Target gate set	$\{X, Z, H, S(S^\dagger), T(T^\dagger), R_Z(\theta), \text{CNOT}\}$ , $\{X, Z, H, S(S^\dagger), T(T^\dagger), \text{CNOT}\}$
System	Mapping type	Structured, nonstructured
	Qubit layout	(Nonstructured) All-to-all, 1D, 2D, Arbitrary (Structured) All-to-all, (1D, 1D), (1D, 2D), (2D, 2D)
Building Block	FTQC scheme	Steane code, surface code
	Device	Time, fidelity

fault-tolerant quantum computing to run Shor's factoring algorithm. For that, we first write a quantum program code of the factoring algorithm and then compile to generate a quantum-assembly code. At the same time, we build up logical building blocks with the chosen quantum error-correcting code and quantum device. The following task is to determine a quantum-computer architecture of a specific physical and/or logical qubit layout. Suppose that all of a quantum algorithm, an error-correcting code, a device, and a system architecture are prepared. Then, it is time to perform the system synthesis. After the system synthesis, we see the analysis results. Note that it is also possible to choose physical nonfault-tolerant quantum computing by assuming a high-fidelity quantum-computing device (see Sec. VII B).

In Table II, we show the options our platform currently supports. In the table, the compile type and the mapping type (with qubit layout) completely depend on the type of quantum-assembly code, *structured* code, and *nonstructured* code. A structured quantum-assembly code is processed by a structured system mapping method for a structured quantum-computer architecture. Similarly, a nonstructured quantum-assembly code is taken by a nonstructured system mapping method for a nonstructured (monolithic) quantum-computer architecture. The details of the structured and nonstructured quantum-assembly codes is described in Sec. IV A.

With the proposed platform, a quantum-computing system model is configured by following the below steps. The first is to specify the type of quantum computing, a fault-tolerant quantum computing or a nonfault-tolerant physical quantum computing. In general, such a choice is closely depending on the size of a quantum algorithm and the performance of the physical device. Needless to say, a larger quantum algorithm requires the more reliable qubit and gate. If you decide a fault-tolerant quantum computing, you also need to choose a quantum error-correcting code. The platform currently supports [[7, 1, 3]] Steane code and double-defect surface code. According to the size of a quantum algorithm (equivalently KQ [41] of the

algorithm) and the reliability of a physical device, the concatenation level for the Steane code or the code distance for the surface code is determined.

The chosen quantum-computing type affects quantum compile. More precisely, a set of the target quantum gates for a compile completely is closely related to the quantum-computing type. For example, the  $R_Z(\theta)$  gate for an arbitrary rotational angle  $\theta$  is very exploited in a quantum algorithm. However, an error-corrected logical version of such a rotational gate cannot be generally implemented in a fault-tolerant manner. Only a few special angles such as  $\pi/2$  and  $\pi/4$  are allowed. Therefore, to realize the logical rotational gate, we have to decompose the  $R_Z(\theta)$  gate into a sequence of  $H$ ,  $S$ , and  $T$  gates that can be implemented in a fault-tolerant manner.

The second step is to make a quantum-computing program and compile it into a quantum-assembly code. As mentioned above, for the compile, target quantum gates determined in the previous step need to be selected. As of writing this paper, the proposed platform exploits open-source programming language *Scaffold* and compiler *ScaffCC*. You can see how to make a Scaffold program in Ref. [29] and how to use ScaffCC compiler in Refs. [17, 26]. In Sec. IV A, we show a simple example of a Scaffold program and the associated quantum-assembly code. Note that ScaffCC decomposes an arbitrary one-qubit gate into a sequence of  $H$ ,  $S$ , and  $T$  by exploiting the compile algorithm *gridsynth* [3] or *sqct* [30].

The third step is to choose a quantum-computing architecture, in particular, the qubit layout and the qubit connectivity. Our platform takes not only a simple regular one- or two-dimensional lattice but also a hierarchically structured qubit array. In the structured layout, a communication bus is considered to make an efficient interaction between distant qubits. Figure 11 shows an example of hierarchically structured quantum-computing architectures. In the case of the Steane-code quantum computing, the qubit connectivity seriously affects the performance of a quantum computing due to the limited local qubit interaction. We show such a limitation raises highly nontrivial temporal

overhead in Sec. VIC. On the other hand, the surface-code quantum-computing scheme is fundamentally established based on local qubit interaction on the two-dimensional lattice. Figure 14 shows a quantum-computing architecture for a surface-code quantum computing.

If the configuration is completed, we perform the integration of all the prepared components via the system synthesis. In the system synthesis, an algorithm qubit is mapped to physical qubit(s) and the execution sequence of algorithm gates based on the qubit mapping is determined. In doing so, the quantum algorithm is reformulated for the target quantum-computer architecture. By processing each instruction of the reformulated quantum algorithm, the platform evaluates the performance (circuit depth, execution time, fidelity, KQ) of a quantum algorithm and the required quantum resources (qubits).

### C. Analysis of quantum computing

We briefly mention what performance items are appraised by the platform, but the detailed analysis method is described in Sec. V. The platform first inspects the quantities of qubits and gates. Those figures are usually analyzed by a quantum-compile framework [13,17]. But, our framework examines such quantities with the consideration for fault-tolerant quantum computing and a quantum-computer system architecture. During the system synthesis, the quantity of ancilla qubits used to distill high-fidelity magic states is also taken into consideration. Therefore, it is possible to estimate the temporal and spatial overhead caused by factors that are veiled in a quantum algorithm, and therefore we believe our estimation nearly coincides with the resource to perform real quantum computing.

The platform examines the expected quantum-computing execution time (circuit depth, fidelity, KQ, and so on) of a quantum algorithm based on a quantum-computer architecture, fault-tolerant protocols, and a physical device. By applying the properties of a physical device and fault-tolerant protocols, we deduce the performance of logical gates and quantum error correction (QEC). Besides, from the system synthesis, we are able to obtain the execution time of a quantum algorithm, in particular, a single-round execution, by applying the performance of logical gates and error correction above. Our framework goes further for a more detailed analysis. The fidelity of quantum computing also can be calculated, and by reflecting the fidelity, it is possible to estimate the execution time of a quantum algorithm achieving a fidelity of 100%. In doing so, we can fairly compare fault-tolerant quantum computing and nonfault-tolerant quantum computing.

Besides the above mentioned, the framework can generate various performance data. Based on the obtained data, we can estimate the temporal and spatial overhead of quantum computing. For example, as mentioned above

limited local interaction between nearest-neighbor qubits sometimes requires additional qubit movements to perform a two-qubit CNOT gate locally. The quantity of such qubit movements corresponds to the temporal overhead. The platform evaluates such a temporal overhead as a ratio of SWAP gates to total quantum gates. As shown in Sec. VIC, quantum computing requires highly nontrivial temporal overhead.

## IV. PROPOSED QUANTUM-COMPUTING FRAMEWORK

### A. Compile layer

Quantum compile is a process that decomposes a quantum algorithm into a sequence of quantum gates. Here a quantum algorithm is in entirely programmed form by using a high-level abstract programming language. Recently, several research groups have developed programming environments for quantum computing by modifying conventional classical programming languages such as PYTHON and C/C++ [11,18,24,27,29,31,32].

It is well known that an arbitrary quantum algorithm can be decomposed into the combination of one-qubit rotational gates and two-qubit CNOT gates [33]. The target quantum gates for a compile can be varied according to a situation. For example, the set of  $H$ ,  $T$ , and CNOT is *de facto* standard for universal fault-tolerant quantum computing. But, to reduce the complexity in compile or to provide flexibility to a programmer, one usually adds more quantum gates to target gates. Furthermore, according to qubit technologies, physically implementable quantum gates slightly differ [33,34]. In this work, we utilize two sets of quantum gates  $\{X, Z, H, S(S^\dagger), T(T^\dagger), R_Z(\theta), \text{CNOT}\}$  and  $\{X, Z, H, S(S^\dagger), T(T^\dagger), \text{CNOT}\}$ . The difference between both is the  $R_Z(\theta)$  gate. As we mention before, the rotational gate for an arbitrary angle  $\theta$  can not be implemented in a fault-tolerant manner, and therefore in this work, the former is exploited for a physical quantum computing and the latter is used for fault-tolerant quantum computing.

The output of quantum compile, the sequence of quantum instructions composed of a quantum gate and target qubit(s), is called a *quantum-assembly code*. The quantum-assembly code is a sort of intermediate representation of a quantum algorithm between a mathematical description and a physical machine instruction description [17,35,36]. A standard format for a quantum-assembly code does not exist, and therefore a specific representation and a structure of such slightly differ according to the literature. For example, a quantum instruction to apply a Hadamard gate to a qubit  $q$  is represented as ‘‘Hadamard  $q$ ’’ or ‘‘ $H(q)$ .’’ Besides, a certain quantum-assembly code has its own structure. For example, Open QASM by IBM [36] provides a conditional statement ‘‘*if* . . . *then* . . . *else*’’ usually supported by classical conventional programming languages.

The proposed platform currently hires open-source quantum compiler *ScaffCC* [17,26]. It compiles a quantum-computing program written in quantum-computing programming language *Scaffold* [29]. A Scaffold program consists of one *main* module and multiple submodules (see Fig. 7). A module generally works as a function or a procedure of conventional programming languages such as C/C++, PYTHON and so on. It is composed of instructions calling quantum gates and/or other modules. The execution of a Scaffold program begins with the first instruction of the main module and terminates by conducting the last instruction of the module.

Previously, we mention that some quantum-assembly codes have unique features in their structure. So does the quantum-assembly code made by ScaffCC. The compiler generates a hierarchically structured quantum-assembly code, in which a quantum algorithm is composed of one main module and multiple submodules. A module consists of performing quantum gates and/or other modules. In the previous paragraph, we also mentioned that a Scaffold program is composed of modules. To avoid ambiguity between both, we need to say that a module in a Scaffold program is defined and written by an algorithm designer (programmer), and through a compile, it is converted to a module in a quantum-assembly code. Therefore, the functions of both are definitely identical. Figure 5 shows an example of a module in a quantum-assembly code.

```

// module name: module_name
// parameter qubits: a, b, ..
module module_name ( qbit a , qbit* b, ... )
{
    // local qubits
    qbit scratch[n];

    // classical bit
    cbit answer[m];

    // initializing a local qubit as |0>
    PrepZ ( scratch[0] , 0 );
    ...

    // one-qubit gate
    H ( scratch[0] );

    // two-qubit gate
    CNOT ( scratch[2] , scratch[1] );

    // sub-module
    moduleA ( scratch[1] , scratch[2] , a[2] );

    // measure
    answer[0] = MeasZ ( a[0] );
}

```

FIG. 5. Example of a module. Parameter qubits passed from external (parent or calling) modules are clearly specified at the beginning. A module is defined by the preparation of local qubits and classical bits, calling quantum gates and other submodules, and measurement of local qubits.

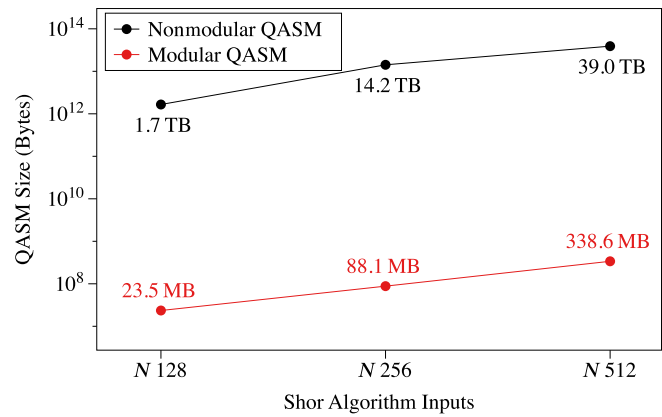


FIG. 6. Comparison in the file size between a nonmodular QASM and a modular QASM. Both codes are generated by the ScaffCC compiler [26]. As the input increases, the size of quantum-assembly code in the nonstructured format increases over dozens TB.

We now need to say why we exploit ScaffCC in this work. As we mention above, a quantum-assembly code is a list of quantum instructions. Therefore, as the size of a quantum algorithm increases, the size of the quantum-assembly code nontrivially increases. It completely follows the complexity of a quantum algorithm. Obviously, the size of a meaningful quantum algorithm, in reality, goes beyond the capability of a conventional supercomputing system. In other words, the size of a quantum-assembly code for our interested algorithm is very huge, and such scalability causes a practical problem in classical control of a quantum-computing system. For example, empirically the size of a quantum-assembly code of Shor's factoring algorithm to factorize 512-bit integer is around 39 TB (see Fig. 6). Therefore, we had trouble in generating and managing such a huge code with a classical computing system. We can not even attempt to generate a larger quantum algorithm than the algorithm above due to the lack of classical storage and memory.

On the other hand, we believe that the hierarchy provided by ScaffCC can suppress the scalability problem. For example, suppose that there exists a composite quantum operation that is composed of  $N$  gates and is called as much as  $M$  times. In a nonstructured quantum-assembly code, a total of  $MN$  quantum instructions ( $\mathcal{N}_{\text{gates}} \times \mathcal{N}_{\text{iteration}}$ ) are required to describe all the execution of such instructions. However, the hierarchical assembly code made by ScaffCC completes such executions with only  $M + N$  instructions ( $\mathcal{N}_{\text{gates}} + \mathcal{N}_{\text{iteration}}$ ) by defining the composite operation as a module.  $N$  instructions are required to define the operation and  $M$  instructions are used to call the module. By doing this, the size of the hierarchical quantum-assembly code is much smaller than that of the nonstructured quantum-assembly code as shown in Fig. 6.

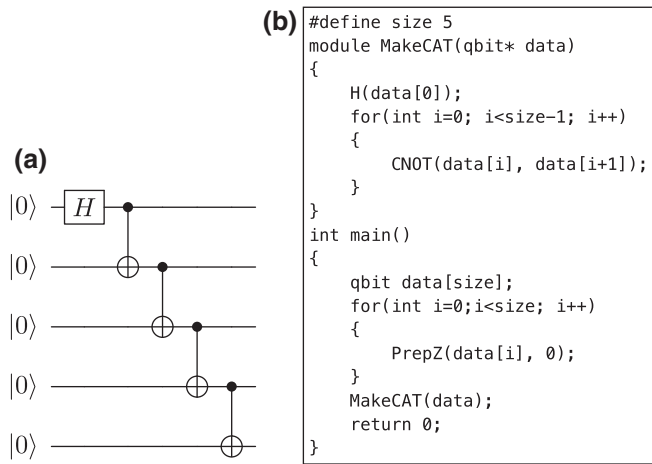


FIG. 7. Example of a Scaffold program to implement a five-qubit CAT state. (a) Quantum circuit and (b) its Scaffold program. The module *MakeCAT* makes the CAT state.

While we estimate that it takes at least 100 days to perform the system synthesis of the Shor  $N = 512$  written in the nonstructured assembly code, but only a few hours are enough for the structured code.

As of writing this paper, ScaffCC only supports such hierarchically structured quantum-assembly code. This is the critical reason why we exploit ScaffCC in the proposed platform in which we focus on large-scale quantum computing.

To conclude this section, we show a simple example of a Scaffold program to make a five-qubit CAT state  $\frac{1}{\sqrt{2}}(|0\rangle^{\otimes 5} + |1\rangle^{\otimes 5})$  and a corresponding quantum-assembly code. Readers can see how to make a Scaffold program in Ref. [29] and how to use ScaffCC compiler in Ref. [26]. Figure 7 shows a quantum circuit to implement a five-qubit CAT state and a corresponding Scaffold program. A structured and nonstructured quantum-assembly code generated via the ScaffCC compiler are respectively shown in Fig. 8.

### B. System layer

A quantum algorithm (quantum-assembly code) is a logic of how to solve a given problem. It is usually developed based on an ideal physical situation where noiseless physical gates and arbitrary long qubit interaction are allowed. In other words, a quantum algorithm is developed without considering any physical implementation.

On the other hand, a quantum computer where a quantum algorithm is executed has a certain logical and physical architecture such as a qubit connectivity. Therefore, to run a quantum algorithm on such a quantum computer, we need to reformulate the algorithm to be compatible with the given quantum-computer architecture. For example, real quantum-computing devices in IBM Quantum Experience [7] have very limited qubit connectivity and even worse

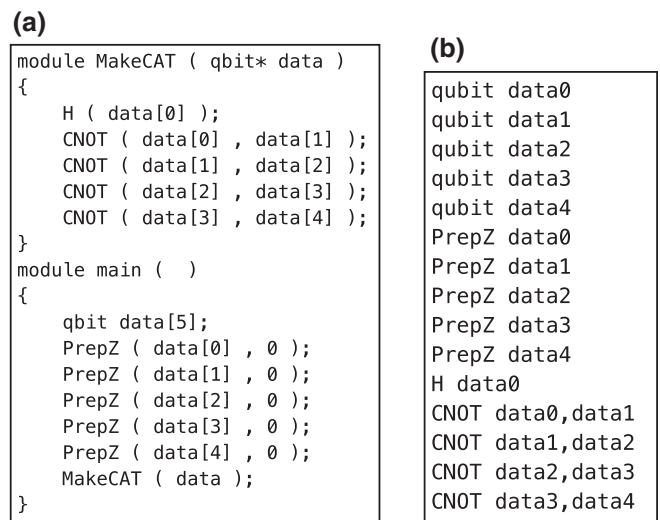


FIG. 8. Quantum-assembly codes to generate a five-qubit CAT state. (a) Structured code and (b) nonstructured code.

allow only one-directional CNOT. Therefore, the quantum-assembly codes shown in Fig. 8 can not be directly executed on the IBM QX4 device [see Fig. 9(a)] because the codes include unallowable CNOT gates. Therefore, for the execution, we have to recast a quantum-assembly code for IBM QX4. In Fig. 9(b), we show the recasted (nonstructured) quantum-assembly code for the device. This is the motivation of a quantum-computing system synthesis.

The principle of a quantum-computing system synthesis is very simple, (1) *set up* a quantum-computer architecture

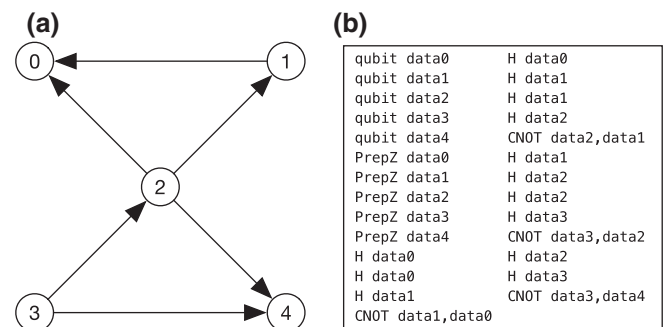


FIG. 9. (a) Qubit layout of IBM QX4 device [7]. A node indicates a qubit, and an edge with a direction implies that the application of a controlled-CNOT gate is possible, where the control qubit and the target qubit are the root and end of the arrow. Therefore, as can be seen bidirectional CNOT is allowed on the IBM QX4. (b) Recasted assembly code from Fig. 7(b). Since the instruction “CNOT *data0,data1*” is not allowed directly on the IBM QX4, Hadamard gates, “*H data1*” and “*H data2*,” are added before and after the instruction. Note that the node index  $k$  indicates the qubit data  $k$ . We so not cancel out repetitive Hadamard gates. By canceling out those gates, the circuit depth can be reduced from 12 to 9.



and (2) *recast* a quantum-assembly code for the architecture. The first *set up* is the mapping the algorithm qubits onto physical device qubits, and the following recast is the gate scheduling based on the arranged qubits. In what follows, we first describe a quantum-computer architecture and then show how to actualize a quantum algorithm on the target architecture.

### 1. Quantum-computer architecture

We discuss a hierarchically structured quantum-computer architecture for the proposed framework. In general, there is no restriction to the architecture. In other literature, a regular one- or two-dimensional lattice is usually exploited. But, in this work, we assume that it is hierarchically structured. A quantum computer is then composed of several computing regions and a communication bus connecting all the computing regions. With such a structured architecture, the system mapping with a hierarchically structured quantum-assembly code can be done efficiently.

A computing region is completely associated with a module in a quantum-assembly code. Therefore, a computing region has to support the functions of the associated module. The region has multiple qubit cells as much as the number of the qubits of the module. Some cells are allocated for parameter qubits passed from other modules, and others are allocated for local qubits temporarily used within the module. Additional space is sometimes required to form the two-dimensional rectangular shape of the module. Figure 10 shows the example of a module in a quantum-assembly code and its associated computing region.

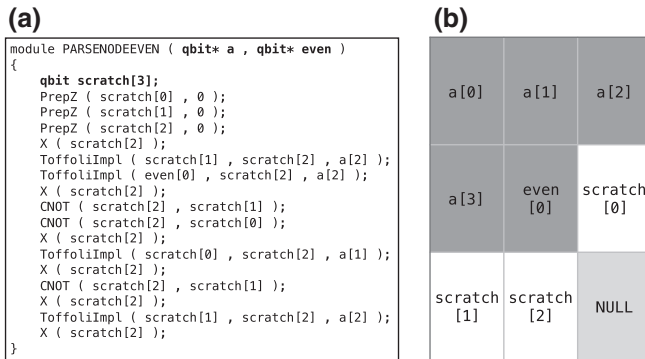


FIG. 10. (a) Example of a module in a quantum-assembly code and (b) the associated computing region on a quantum-computer architecture. In (a), the qubits  $a$  and  $even$  are parameter qubits passed from other modules, and the qubit  $scratch$  is a local qubit only used within the module. In (b), the dark gray cells are for the parameter qubits, the white cells are for the local qubits and the light gray cells are just empty space or null qubits working trivially. While the size of the parameter qubits  $a$  and  $even$  are not specified in the module definition, it can be determined by tracing back to all modules that calls the module.

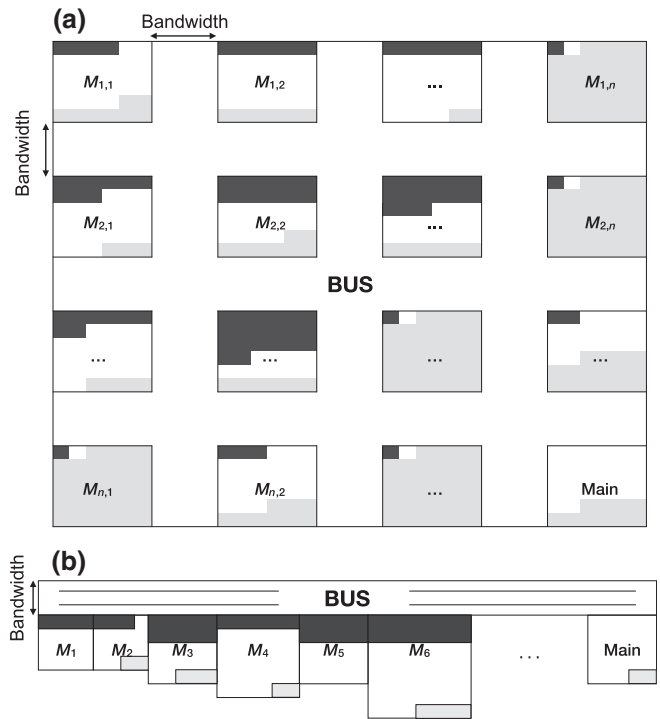


FIG. 11. Example of a proposed quantum-computer architecture. (a) 2D global layout and 2D local layout. (b) 1D global layout and 2D local layout. Logical qubits of a different color play a different role in a module; parameter qubits (dark gray), local qubits (white), and dummy qubits (light gray).

Figure 11 shows examples of a hierarchically structured quantum-computer architecture. The box labeled  $M_{i,j}$  ( $M_i$ ) indicates a module (computing region). We call the arrangement of modules a *global* layout on the whole quantum computer, and the arrangement of qubits within each module a *local* layout, respectively. All modules communicate with each other via a communication bus. In the figure, the bus is depicted as a white space outside of modules. We discuss the bandwidth of the communication bus in Sec. V.

Qubit that resides inside a module supports universal quantum operations. Such a qubit, a logical qubit encoded in a quantum error-correcting qubit, is composed of (low-level concatenated or physical) data qubits for holding data and ancilla qubits for error correction and logical operations. On the other hand, the qubits for a communication bus perform error correction and logical Clifford operations only. Therefore, the composition of logical qubits for a communication bus can be differed from that for modules according to a quantum error-correcting code and a fault-tolerant quantum-computing scheme.

### 2. System synthesis

Figure 1 shows that for the performance analysis all the data are collected in the system layer and there the

performance of quantum computing is evaluated. In this section, we describe the system synthesis in terms of the performance evaluation.

We first describe the qubit mapping that associates algorithm qubits with physical qubits. In this work, we map the algorithm qubits onto the physical qubits in a first-come-first-served manner without considering any optimized mapping technology. Figure 10 shows an example of the method as the qubits in the assembly code are allocated to the computing region sequentially as they appear in the code. Please note that several improved quantum-circuit mapping methods have been proposed to reduce the amount of qubit movements [37–39], and by taking such methods, it is possible to make the performance of quantum computing better than the above naive mapping at the cost of the efficiency of the system synthesis.

Provided that all the algorithm qubits are mapped onto the qubit layout. Then, we have to determine the schedule of all the algorithm instructions for the target qubit layout. A specific process definitely depends on the type of quantum instructions from the quantum-assembly code. Quantum instructions in the hierarchically structured quantum-assembly code are classified into three types: one-, two-qubit gate, and module.

The set of one-qubit gates includes  $X$ ,  $Z$ ,  $H$ ,  $S$  ( $S^\dagger$ ),  $T$  ( $T^\dagger$ ),  $R_Z(\theta)$  and a preparation and a measurement in the  $Z$  basis. The synthesis process for such a one-qubit gate is straightforward and can be done independently from other qubits. Suppose that it is scheduled that a Hadamard gate is applied to a qubit  $q$ . If the qubit is in idle status, i.e., no gate is acting on the qubit now, the Hadamard gate can be executed immediately. Otherwise, the Hadamard gate is performed just after the previously scheduled operations are finished. If the scheduled operations are over at time  $t(q)$ , then Hadamard operation starts at  $t(q)$  and finishes at  $t(q) + H_t$ , where  $H_t$  is the execution time of the gate. This is everything for the synthesis of a one-qubit gate. Note that the execution time and fidelity of a quantum gate are provided from the building-block layer.

For a two-qubit gate, the present work deals with a CNOT gate only. When a SWAP gate is required, we implement a SWAP gate via three CNOT gates. We assume that a CNOT gate can be executed on the two qubits located in the nearest neighbor. Suppose that it is scheduled that a CNOT gate is applied to qubits  $q_a$  and  $q_b$ . For that, both qubits have to be in temporarily and spatially ready status. First, if they are apart, we make both qubits be located in neighbor via qubit movements such as SWAP operations. Second, if one qubit (or both qubits) is being manipulated by previously scheduled operations, we delay the CNOT gate operation until both qubits are in idle status. In that case, the CNOT operation definitely begins at  $\max\{t(q_a), t(q_b)\}$ , and finishes on time  $\max\{t(q_a), t(q_b)\} + \text{CNOT}_t$ . Note that  $\max\{t(q_a), t(q_b)\}$  is the time both qubits are in idle status, and  $\text{CNOT}_t$  is the execution time of a CNOT gate.

The third type of quantum instruction, a module, works like a multiqubit quantum operation. Therefore, on the surface, it seems that the synthesis of a module is very similar to the synthesis process of a two-qubit CNOT gate. For the synthesis of a module, several argument qubits for the module should be temporally and spatially ready. The critical difference from the case of a CNOT gate is that a distinguished physical space (the physical space for a module is the computing region we describe before) should be allocated for a module. Therefore, to perform the synthesis of a module, we take into account the qubit movements from the present module to a target module, and vice versa. The detailed process for the qubit movements between modules is described in the following paragraphs.

Suppose that module  $A$  is being synthesized now. The quantum instruction to process in the next turn is “ $M(q_a, q_b, q_c)$ ,” which implies that module  $M$  is called with argument qubits  $q_a$ ,  $q_b$ , and  $q_c$ . For that, we first transmit the argument qubits to the designated area of the module  $M$ . The qubit transmission is achieved by a sequence of SWAP operations through a communication bus. We call the transmission from the calling module  $A$  to the called module  $M$  a *forward qubit passing*. After the forward qubit passing, the qubits are placed at the parameter qubit section of the module  $M$ . See Fig. 10(b) for the parameter qubit section of a module.

All the quantum instructions of the module  $M$  are then executed (synthesized) over the qubits, where some qubits are just passed from the module  $A$  and the others are local qubits of the module  $M$ . If it is faced with a quantum instruction calling another module  $M'$ , then some qubits in the module  $M$  are passed to the designated space of the newly called module  $M'$  and manipulated thereby following the quantum instructions of the module. After executing all quantum instructions of the module  $M$ , the passed qubits have to be back to the original module  $A$ . We call this returning transmission a *backward qubit passing*. Figure 12 shows the module operation including the forward and backward qubit passings.

We now describe how to evaluate the performance of a quantum algorithm. For the evaluation, we need to keep two lookup tables, a *local* lookup table and a *global* lookup table. The evaluation proceeds by performing all modules sequentially as they appear in a quantum-assembly code. For each module, we initialize a local lookup table for all qubits in the module and update the operation time of each qubit as we process each quantum instruction in the module. Algorithms 1 and 2 summarize the system-synthesis method in terms of the performance analysis. As the mapping proceeds instruction by instruction in the quantum-assembly code, the performance criterion cycle and time are updated per a qubit. For a two-qubit gate, we use the notations *cycle\** and *time\**, which are the latest cycle and time between

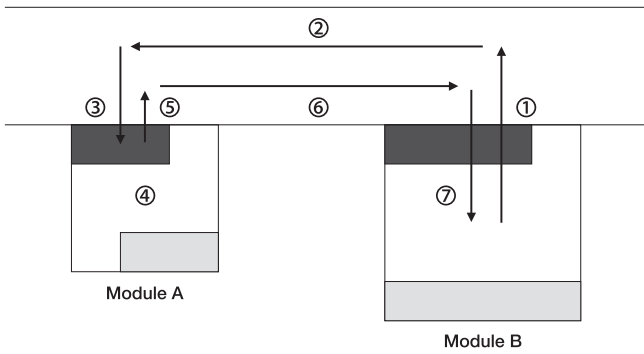


FIG. 12. Example of the module operation that consists of seven steps: (1) (forward) move qubits to the bus, (2) (forward) move to the target module, (3) (forward) move to the parameter qubit cells (dark gray cells), (4) module operations, (5) (backward) move qubits to the bus, (6) (backward) move to the original module, and (7) (backward) move to the original qubit positions.

both qubits,  $\text{cycle}^* = \max\{T[i]_{\text{cycle}}, T[j]_{\text{cycle}}\}$  and  $\text{time}^* = \max\{T[i]_{\text{time}}, T[j]_{\text{time}}\}$ .

After processing all the instructions of a module, we determine the circuit depth and the execution time of the module by picking up the maximum execution time among all qubits,  $\max_Q\{T[Q]_{\text{cycle}}\}$  and  $\max_Q\{T[Q]_{\text{time}}\}$ . The performance of a module is then recorded in the global lookup table. In the middle of the synthesis of modules, if a module already synthesized is called, then we can refer to the performance of the module from the global lookup table instead of performing the synthesis again. Note that our analysis method assumes that a module is executed with the same cost whenever it is called in the algorithm, and therefore by performing the synthesis of a module only once is enough in the hierarchical system synthesis.

Provided that the synthesis of all modules are completed, we can determine the execution time of a quantum algorithm as the maximum operation time among the qubits in the *main* module. This is a single-round execution time of a quantum algorithm. For example, the quantum-assembly code in Fig. 8(a) can be mapped as shown in Table III, and its execution time is determined as  $\text{PrepZ}_t + \text{FP}_{\text{main} \rightarrow \text{MakeCAT}} + \text{MakeCAT}_t + \text{BP}_{\text{main} \leftarrow \text{MakeCAT}}$ . The operation length and the execution time of the module *MakeCAT* are respectively 5 and  $H_t + 4\text{CNOT}_t$ , where  $H_t$ ,  $\text{CNOT}_t$  and  $\text{PrepZ}_t$  are the execution times of *H*, *CNOT* and *PrepZ* (preparation operation with *Z* basis). As shown in Fig. 9, the execution of a two-qubit *CNOT* gate sometimes requires qubit movements if control and target qubits are not positioned in the neighborhood. In Table III, for the purpose of the presentation, we do not consider such a situation in detail. Note that  $\text{FP}_{\text{main} \rightarrow \text{MakeCAT}}$  ( $\text{BP}_{\text{main} \leftarrow \text{MakeCAT}}$ ) is the time for the forward (backward) qubit passing, which is determined based on the distance between both the modules, the position of the argument qubits to be passed and the qubit passing

TABLE III. Mapping result of the quantum-assembly code in Fig. 8(a).

Operation index	data[0]	data[1]	data[2]	data[3]	data[4]
1	PrepZ	PrepZ	PrepZ	PrepZ	PrepZ
2	Forward passing (main $\rightarrow$ MakeCAT)				
3	H				
4	CNOT				
5	CNOT				
6	CNOT				
7	CNOT				
8	Backward passing (main $\leftarrow$ MakeCAT)				

technology [40]. As discussed later, the execution time of quantum gates are determined based on the quantum error correction and fault-tolerant gate protocols. In the above example, the quantity of the qubits to run the algorithm is a sum of five algorithm qubits and  $Q_{\text{bus}}$  bus qubits. According to the type of quantum computing, an algorithm qubit corresponds to a physical qubit or an error-corrected logical qubit.

So far, we describe a system synthesis algorithm for a hierarchically structured quantum-assembly code. The presented algorithm can be applied to a nonstructured quantum assembly code needless to say. In such a code, there are two types of quantum instructions: one- and two-qubit gate. Regardless of the type of a quantum-assembly code, as mentioned before the heart of the system mapping is (1) *set up* a quantum-computer architecture and (2) *recast* quantum algorithm for the architecture by performing the system mapping. To be compatible with the quantum-assembly code, a simple qubit array such as a regular two-dimensional lattice may be enough. The proposed framework supports a system mapping on an arbitrary qubit layout as shown in Fig. 9(a).

### C. Building-block layer

It is well known that very robust and accurate qubits and quantum gates are required to achieve reliable quantum computing. For example, by following a *KQ* formalism [41], to accurately run a quantum algorithm of 50 qubits and 1000 circuit depth the physical error rate of a quantum gate should be less than 0.00002. Otherwise, a quantum error may happen during the computation, which causes the quantum computation to be broken. Furthermore, the interesting quantum algorithms such as integer factoring [42] and unstructured database search [43] need much more qubits and much longer circuit depth. The required error rate for those algorithms should be below as much as  $O(10^{-15})$ – $O(10^{-20})$ , but it looks almost impossible to achieve it practically. Fortunately, it is well known that we can make an effective error rate of quantum gates lower to the arbitrary degree by employing a quantum error correction [44–48] and a fault-tolerant protocol [41,49–52].

```

1: function MAPPING-GLOBAL(QASM)
2:    $M$  = the set of modules in QASM
3:    $N$  = the number of modules in QASM
4:   Let  $LT$  be a list of size  $N$   $\triangleright$  global look-up table
5:   for  $m = 1$  to  $N$  do
6:      $m$  = the index of a module
7:      $M_m$  = QASM of the module  $m$ 
8:      $L_m$  = the list of qubits in the module  $m$ 
9:      $n_m$  = the number of qubits in the module  $m$ ,  $|L_m|$ 
10:    Let  $A_m$  be 2D square qubit array of size  $\lceil\sqrt{n_m}\rceil \times \lceil\sqrt{n_m}\rceil$ 
11:    Initialize  $A_m \leftarrow L_m$ 
12:     $\triangleright$  arrange algorithm qubits onto physical qubits
13:    Initialize  $LT[m](cycle = 0, time = 0)$ 
14:    Mapping-module ( $m, M_m, A_m, n_m, LT$ )
15:     $\triangleright$  mapping function for each module
16:  end for
17:  QC time =  $\max_m \{LT[m](time)\}$ 
18: end function

```

Algorithm 1. System mapping for hierarchical QASM

In the present work, we apply the fault-tolerant quantum-computing protocols based on  $[[7, 1, 3]]$  Steane code [47] and surface code [19,53]. Both codes have well-studied logical-gate protocols. The concatenation level of Steane code and the code distance of a surface code are completely determined by the size of a given quantum algorithm and physical error rate [16,20]. In this work, we set both figures by using KQ formalism [41]. Note that the

KQ value, the number of qubits and the circuit depth, completely depends on the size of a quantum algorithm, and therefore it should be provided from the compile layer.

### 1. Steane code

$[[7, 1, 3]]$  Steane code encodes logical quantum information in a qubit into seven physical qubits and protects it from an arbitrary one-qubit quantum error. Since the transversal implementations for a logical Hadamard and a logical CNOT gate are supported well, many studies on the fault-tolerant quantum computing based on the Steane code are conducted. In Ref. [5], an optimal design of a logical qubit for Steane code under the two-dimensional nearest-neighbor qubit interaction was proposed. They achieved the threshold  $O(10^{-5})$  with 48 lower-level qubits and modified quantum error correction.

In this work, we redesign a logical qubit with 30 lower-level qubits. Seven qubits among them are used for holding data, and the others are temporarily used for logical operations and error correction. We apply the Shor quantum error correction [49], exploiting a four-qubit Shor state for the syndrome measurement. For that, we prepare and verify the Shor state [54]. We implement the preparation of a logical state  $|0\rangle_L$  by following Ref. [15]. Most of the logical gates are implemented as transversal gates, and the non-Clifford  $T$  gate is implemented by exploiting a magic state. We generate magic states by employing a

```

function MAPPING-MODULE( $m, M_m, A_m, n_m, LT$ )
   $N_m$  = the number of instructions in  $M_m$ 
  Let  $T$  be an array of  $\lceil\sqrt{n_m}\rceil \times \lceil\sqrt{n_m}\rceil$   $\triangleright$  local look-up table
  Initialize  $T[Q](cycle = 0, time = 0)$  over  $Q \in \{1, 2, \dots, n_m\}$ 
  for  $i = 1$  to  $N_m$  do
     $U, Q$  = Gate and Qubit(s) in  $M_m[i]$ 
    if  $U \in \{X, Z, Y, H, S, S^\dagger, T, T^\dagger Prep, Meas\}$  then  $\triangleright$  one-qubit gate
      Apply  $U$  to  $Q$ 
      Update  $T[Q](cycle = cycle + 1, time = time + U_{time})$ 
    else if  $U$  is a CNOT gate then  $\triangleright$  CNOT gate
      while Qubits are not in neighborhood in  $A_m$  do
        Apply SWAP to Qubits
        Update  $A_m$ 
        Update  $T[Q](cycle = cycle^* + 1, time = time^* + SWAP_{time})$   $\triangleright$  updating the position of the qubits
      end while
      Apply CNOT to  $Q$ 
      Update  $T[Q](cycle = cycle^* + 1, time = time^* + CNOT_{time})$ 
    else if  $U \in$  modules then  $\triangleright$  module
      1. Apply forward parameter passing to  $Q$ 
        Update  $T[Q](cycle = cycle^* + 1, time = time^* + SWAP_{time} \times distance)$ 
      2. Apply module operation to  $Q$ 
        Update  $T[Q](cycle = cycle^* + U_{cycle}, time = time^* + U_{time})$   $\triangleright U_{time, cycle} \leftarrow LT[U](time, cycle)$ 
      3. Apply backward parameter passing to  $Q$ 
        Update  $T[Q](cycle = cycle^* + 1, time = time^* + SWAP_{time} \times distance)$ 
    end if
  end for
   $LT[m](cycle = \max_Q(T[Q](cycle)), time = \max_Q(T[Q](time)))$   $\triangleright$  Update  $LT$ 
end function

```

Algorithm 2. System mapping for each module



seven-qubit Shor state without resource-consuming magic state distillation [55].

Accuracy threshold theorem [56,57] says that if we have a quantum device of physical error rate below a code threshold, it is possible to achieve arbitrarily reliable quantum computing. But, for a very large-sized quantum algorithm, encoding only once may not be enough. Fortunately, by encoding a qubit recursively [51], we can lower the effective error rate to the degree where reliable quantum computing is possible.

Given a quantum algorithm, we can calculate KQ and determine the maximum tolerable error rate  $P_{\max}$  as  $1/\text{KQ}$ . We then determine the concatenation level  $l$  by the following inequalities satisfies

$$P_{\max} \geq \frac{(c_{\text{op}} P^2)^{2^l}}{c_{\text{op}}}, \quad (1)$$

where  $\text{op}$  is quantum error correction and logical operations, and  $c_{\text{op}}$  is the constant factor of a specific logical operation  $\text{op}$ . We obtain the constant values of each logical operation from KQ of a quantum circuit for the operation. For example,  $c_{\text{QEC}}$  corresponds to KQ of the QEC quantum circuit. In this work, we do not optimize the arrangement of qubits (see Table IV), and therefore the quantum error correction and a logical operation work sub-optimally, and therefore the threshold may be lower than the optimal value  $O(10^{-5})$ . (Please note that the objective of our work is not to increase a code threshold, but to configure quantum computing and analyze its performance accurately.)

Suppose that a concatenation level for quantum computing is determined as  $l$ . The implementation of a logical  $T$  gate in the level  $l$  consists of only Clifford operations at a lower level  $k < l$ . Then, in the level  $k$ , the implementation of a logical  $T$  gate is not necessary and therefore the qubits to implement a magic state, the seven-qubit Shor state, are not strongly required. Therefore, only 23 qubits

TABLE IV. Arrangement of qubits to implement a logical qubit in the concatenation level  $l$ . The component qubits are in the concatenation level  $l - 1$ . The qubit denoted by  $D[i]$  indicates a  $i$ th data qubit. The qubits  $4\text{Sh}[i]$  and  $7\text{Sh}[j]$  are for four- and seven-qubit Shor states for syndrome measurement and a logical  $T$  gate, and  $V_{4\text{Sh}}$  and  $V_{7\text{Sh}}$  are used to verify the four- and seven-qubit Shor states, respectively. The qubit  $M[i]$  is also used to implement a logical  $T$  gate.

$V_{4\text{Sh}}[1]$	$V_{4\text{Sh}}[2]$	$D[1]$	$D[2]$	$D[3]$
4Sh[1]	4Sh[2]	$D[4]$	$D[5]$	$D[6]$
4Sh[4]	4Sh[3]	$D[7]$	$V_{7\text{Sh}}[1]$	$V_{7\text{Sh}}[2]$
$M[1]$	$M[2]$	$M[3]$	$M[4]$	$M[5]$
$M[6]$	$M[7]$	$V_{7\text{Sh}}[3]$	7Sh[1]	7Sh[2]
7Sh[3]	7Sh[4]	7Sh[5]	7Sh[6]	7Sh[7]

are required to implement a logical qubit in the lower-level  $k$ . But, to form a two-dimensional rectangular shape of a logical qubit, we require 25 qubits ( $5 \times 5$  layout) for a lower-level qubit in the level  $k$ . In Table IV, the qubit denoted by  $7\text{Sh}[i]$  is not required in the lower-level qubit  $k$ . On the other hand, the qubits  $V_{7\text{Sh}}[i]$  (the main role of which is to verify the seven-qubit Shor state) are used for the other purpose, logical measurement.

## 2. Surface code

Two-dimensional surface-code-based fault-tolerant quantum computing is recognized as the most promising fault-tolerant quantum-computing scheme due to physically less challenging requirements. The code has a high threshold around  $O(10^{-2})$  [53,58,59], and its structure is well suited to nearest-neighbor interacting qubits arranged on a two-dimensional lattice. In this work, we implement double-defect-based logical qubits and logical gates described in Refs. [16,19,53]. The detailed protocols are beyond the scope of the present work, and we describe performance parameters only.

We use the KQ formalism to determine a code distance  $d$  [16,20]. The objective error rate of quantum computing is determined by  $P_{\text{fail}} \approx \text{KQ}\epsilon_L$ , where  $\epsilon_L$  is a logical error rate. The code distance  $d$  is then determined as

$$d \approx \frac{2(\log \epsilon_L - \log C_1)}{\log C_2 + \log \frac{\epsilon_p}{\epsilon_{\text{th}}}} - 1, \quad (2)$$

where  $\epsilon_p$  and  $\epsilon_{\text{th}}$  are the physical error rate and the threshold of the surface, respectively.  $C_1$  and  $C_2$  are code parameters, and we use the specific figures,  $C_1 \approx 0.13$ ,  $C_2 \approx 0.61$ , from Ref. [20]. We apply the code threshold  $\epsilon_{\text{th}} = 0.009$  from the same reference.

Now it is possible to determine the execution time of surface-code logical gates. Above all, we stress that the surface-code error correction has to iterate  $d$  rounds of a syndrome measurement to avoid an effect by noise during the measurement [19]. We assume that logical Pauli operators are performed in classical control software by updating the logical Pauli frame [19]. A logical CNOT gate between the same type ( $X$ -cut or  $Z$ -cut) logical qubits consists of three CNOT gates between different type of logical qubits. For that, we prepare a pair of different types of logical ancilla qubits [19]. A logical Hadamard gate protocol consists of cutting and reconnecting a target logical qubit from/to a whole qubit array and performing transversal physical Hadamard and SWAP gates [19,60]. The Hadamard gate makes the role of syndrome qubits interchanged, and the syndrome qubit reverts to the original position (role) via the SWAP operation.

We now turn our attention to the nontransversal gates  $S$  and  $T$ . A logical  $S$  gate is deterministically implemented by using a high-fidelity magic state  $|Y_L\rangle = \frac{1}{\sqrt{2}}(|0_L\rangle + i|1_L\rangle)$

[19,20]. Since the gate protocol does not measure  $|Y_L\rangle$ , it is possible to reuse it after we generate plenty of the states at the beginning. How many  $|Y_L\rangle$  states should be prepared is discussed later. A logical  $T$  gate is implemented by consuming a high-fidelity magic state  $|A_L\rangle = \frac{1}{\sqrt{2}}(|0_L\rangle + \exp^{i\pi/4}|1_L\rangle)$  [19]. The magic state has to be prepared for every  $T$  gate. We assume that a magic state is prepared and supplied in offline. In other words, the preparation of the high fidelity  $|A_L\rangle$  is not included in the quantum-computing time. On the other hand, the logical  $T$ -gate operation is probabilistically achieved up to the logical  $S$ -gate correction. Therefore, to implement a logical  $T$  gate, a  $|Y_L\rangle$  state is probabilistically required.

We now discuss the quantity of the required  $|Y_L\rangle$  states. It depends on the quantity of the states maximally required at one time. Since a logical  $T$  gate probably requires a  $|Y_L\rangle$ , we have to prepare  $|Y_L\rangle$  as much as  $\max\{\text{parallel } T, \text{parallel } S\}$ , where parallel  $T$  (parallel  $S$ ) is the number of logical  $T$  ( $S$ ) gates executed in parallel. Note that the quantities of parallel  $T$  and parallel  $S$  can be found from the system-synthesis process.

The preparation of a high-fidelity magic state takes two steps, *state injection* and *state distillation* [19,61]. The state injection in the surface-code quantum computing injects an arbitrary logical state into the distance 1 logical qubit called a *short* qubit and makes the logical qubit larger [19]. Enlarging a double-defect logical qubit consists of multicell qubit movements and measurement on data qubits. The state-distillation protocol takes  $m$  noisy states and generates  $k$  less noisy states, where  $m > k$ . By performing multiple rounds of the distillation, the magic spread over many states can be concentrated onto only a few states and therefore we can obtain high-fidelity magic states. In this work, we deal with the magic-state-distillation protocols described in Refs. [19,53]. The required iteration of the protocol is completely determined by the objective fidelity and a physical error rate [16]. We set the objective error rate of the magic states as  $10^{-12}$  to achieve high fidelity for the configured quantum computing ( $1/\mathcal{N}T$  gates), and empirically the two-round distillation achieved the objective error rate in the physical error rate  $10^{-3}$ – $10^{-5}$ .

We determine the capacity of a magic state factory that prepares and supplies  $|A_L\rangle$  states. The capacity depends on a quantum algorithm and the running times of a state distillation and a logical  $T$  gate. Imagine that a logical  $T$  gate is applied consecutively to a certain qubit. If a magic state factory generates only one magic state at a time, there a latency for the supply of the magic states may happen if the magic state distillation takes more time than the execution time of a logical  $T$  gate. Therefore, the magic state factory has the capacity to prepare at least  $\max\{\text{parallel } T\} \times \text{time}(\text{MSD})/\text{time}(T)$  states at a time where  $\text{time}(\text{MSD})$  and  $\text{time}(T)$  are the running times

of the magic state distillation and the logical  $T$ -gate protocol. Empirically, the  $\text{time}(\text{MSD})/\text{time}(T)$  is approximately 20 in our estimation.

To conclude, the required physical qubits for  $|A_L\rangle$  and  $|Y_L\rangle$  are respectively

$$\begin{aligned} \max\{\text{parallel } T\} \times \frac{\text{time}(\text{MSD})}{\text{time}(T)} \times (15 \times Q_L)^{r-1} \\ \times (16 \times Q_L), \end{aligned} \quad (3)$$

and

$$\max\{\text{parallel } T, \text{parallel } S\} \times (7 \times Q_L)^{r-1} \times (8 \times Q_L), \quad (4)$$

where  $Q_L$  is the number of physical qubits to implement a logical qubit and  $r$  is the required distillation rounds. The last distillation round requires one more logical qubit from the Bell state [19]. Above this, the ancilla qubits to perform CNOT gates during the distillation protocol also should be included. Recall that to perform a CNOT operation between the same type of logical qubits ( $Z$  cut or  $X$  cut), the opposite type of logical qubit is required as an ancilla.

## V. PERFORMANCE METRIC

We describe how to evaluate the quantum-computing metrics, execution time, fidelity, and the number of qubits.

### A. Execution time

We examine the quantum-computing time in two steps. In the system synthesis, we obtain the single-round execution time  $T_{\text{one}}$  of a quantum algorithm. However, the single-round execution does not guarantee reliable quantum computing. Noisy components may make quantum computing broken. To overcome such a problem, we calculate the average execution time  $T_{\text{avg}}$  by reflecting the number of the required iterations to achieve the fidelity of 100% as

$$T_{\text{avg}} = T_{\text{one}}/F_{\text{alg}}. \quad (5)$$

We believe this averaged time shows the time required for getting a reliable answer from quantum computing. (This does not indicate that the output from quantum computing is an exact solution. We do not consider the probabilistic nature of a quantum algorithm.) Note that how to calculate the fidelity of quantum computing is described in the following section.

### B. Fidelity

The fidelity of fault-tolerant quantum computing can be calculated based on the fidelity of logical quantum gates as

follows [16]:

$$F_{\text{alg}} = \prod_g F_g^{\mathcal{N}_g}, \quad (6)$$

where  $g$  is a quantum gate utilized in the algorithm.  $F_g$  is the fidelity of the gate  $g$ , and  $\mathcal{N}_g$  is the total count of the gate in the algorithm. The value  $\mathcal{N}_g$  can be found from the system synthesis and  $F_g$  is determined in the building-block layer. It is worthwhile to note, this fidelity calculation is only applicable to Steane-code-based quantum computing. As shown in Sec. IV C 2, the final fidelity of surface-code-based quantum computing is given by  $F_{\text{alg}} = 1 - \text{KQ} \times \epsilon_L$ .

### C. Number of physical qubits

We examine the quantity of physical qubits required to run a quantum algorithm. Since the quantity of the required qubits differs according to a fault-tolerant quantum-computing scheme, we first identify the common factor, the qubits in a quantum algorithm, and then go into specific cases later.

The proposed hierarchical quantum-computer structure consists of multiple modules and a communication bus connecting all modules. In the quantum-assembly code, we can find the quantity of logical (or physical) qubits for a module.

$$Q_{\text{comp}} = \sum_{m \in M} (Q_{\text{local}}^m + Q_{\text{param}}^m), \quad (7)$$

where  $Q_{\text{local}}^m$  ( $Q_{\text{param}}^m$ ) is the number of local (parameter) qubits of a (computing) module  $m$ . Note that  $M$  is the set of all modules in the quantum algorithm.

#### 1. Steane-code quantum computing

We consider Steane-code quantum computing. The structure of a communication bus depends on the chosen global layout over all modules. On the 1D global layout, the number of qubits can be simply calculated as  $Q_{\text{comm}} = \text{bandwidth} \times \text{length}$ , where length is obtained as

$$\text{length} = \sum_m m_{\text{width}}, \quad (8)$$

where  $m_{\text{width}}$  is the width of a module, which is 1 for 1D local layout in common and  $\lfloor \sqrt{Q^m} \rfloor$  for 2D local layout. Note that  $Q^m = Q_{\text{local}}^m + Q_{\text{param}}^m$ .

On the other hand, on the 2D global layout, the number of qubits can be calculated as follows. Let us suppose that the number of modules is  $|M|$ . Then,  $\lfloor \sqrt{|M|} \rfloor \times \lfloor \sqrt{|M|} \rfloor$ -sized 2D global layout is necessary. To keep the shape of a module on the 2D layout, all modules have the same

size cells  $n \times n$ , where  $n = \lfloor \sqrt{\max_{m \in M} \{Q^m\}} \rfloor$ . Then, the required logical qubits for the communication bus is

$$Q_{\text{comm}} = 2 \times \text{bandwidth} \times n \times A \times B + (n \times A)^2, \quad (9)$$

where  $A = \lfloor \sqrt{|M|} \rfloor - 1$  and  $B = \lfloor \sqrt{|M|} \rfloor$ . In this work, we determine the bandwidth of a bus as the maximum number of parameter qubits,  $\text{bandwidth} = \max_{m \in M} \{Q_{\text{param}}^m\}$ .

We now turn our attention to the quantity of lower-level qubits forming a logical qubit of the concatenation level  $l$ . As mentioned in Sec. IV C 1, a logical qubit in the concatenation level  $k = 1 \sim l - 1$  is composed of 25 lower-level qubits, and the qubit in the level  $l$  is composed of 30 qubits in the  $l - 1$  level. According to the physical error rate and the size of the quantum algorithm, the concatenation level is determined as mentioned before. The quantity of total physical qubits in the Steane-code quantum computing is then

$$Q_{\text{Steane}} = 25^{r-1} \times 30 \times Q_{\text{comp}} + 25^r \times Q_{\text{comm}}, \quad (10)$$

where  $r$  is the concatenation level.

#### 2. Surface-code quantum computing

We implement double-defect-based logical qubits. For a double-defect logical qubit with code distance  $d$ , each defect has to be apart from a boundary as much as  $d$  data qubits and double defects also should be separated as much as  $d$  data qubits. On the other hand, to perform a braiding operation in a fault-tolerant manner, the space between double defects has to be at least  $2d + \lfloor d/4 \rfloor$  rather than only  $d$ . Therefore, to implement a double-defect logical qubit of code distance  $d$ ,  $(2A + 1)(2B + 1)$  physical qubits are required, where  $A = (2d - 2 + \lfloor d/4 \rfloor)$  and  $B = (4d - 4 + 3\lfloor d/4 \rfloor)$ . Figure 13 shows a double-defect logical qubit of code distance 3. A total of 253 physical qubits, 126 data qubits and 125 syndrome qubits, are required.

Two neighboring logical qubits are also separated as much as  $\lfloor d/4 \rfloor$  data qubits to keep the code distance between both qubits during the fault-tolerant braid transformation. In this regard, if  $N$  logical qubits are arranged on the two-dimensional layout of  $n_h \times n_w$ , we need

$$\left\{ 2 \left[ n_w A + (n_w - 1) \lfloor d/4 \rfloor \right] + 1 \right\} \times \left\{ 2 \left[ n_h B + (n_h - 1) \lfloor d/4 \rfloor \right] + 1 \right\} \quad (11)$$

qubits, where  $A$  and  $B$  are as we mention above.

We take account of the ancilla qubits required for a CNOT gate. As mentioned before, the CNOT gate between the same type ( $X$ -cut or  $Z$ -cut) logical qubits consists of three CNOT gates between different types of logical qubits. For that, two ancilla qubits,  $X$ -cut qubit  $|g_L\rangle$  and  $Z$ -cut qubit  $|+_{\pm L}\rangle$  are required. We allocate a pair of both ancilla qubits

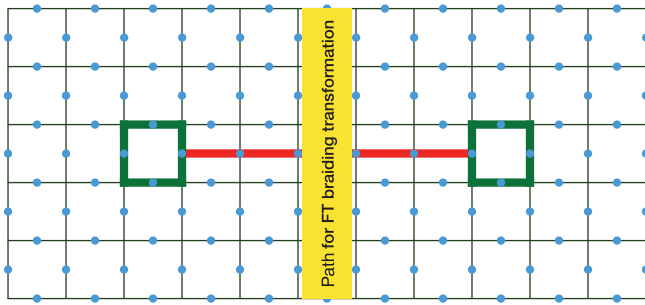


FIG. 13. Double Z-cut qubit of a code distance 3. The blue dots indicate data qubits. One of the green chains indicates a logical Z operator, and the red chain indicates a logical X operator. Through the yellow line, it is possible to perform a fault-tolerant braiding operation from other X-cut qubit to this Z-cut qubit. Each defect has to be away from boundary as much as 3 data qubits, and both defects have to be separated 6 data qubits. 126 data qubits and 125 syndrome qubits are required to implement a distance-3 logical qubit.

to every module where a CNOT gate is performed. In that case, the number of logical qubits for a module is the sum over parameter qubits, local qubits, and two ancilla qubits instead of Eq. (7).

In the case of surface-code quantum computing, the communication bus is not strongly required for the CNOT operation between distant qubits. Instead of the sequence of SWAP operations as much as the passing distance, the qubit transmission in the double-defect surface-code quantum computing is much more efficient. A defect can be moved to an arbitrary apart place via performing a multi-cell movement once [16,19]. However, the qubit passing may be useful to make the interaction between distant qubits belong to different modules more efficient [62].

We assume that the surface-code quantum computing performs the qubit passing sequentially on the bus with the narrow bandwidth. We set the bandwidth of the bus as  $\lfloor d/4 \rfloor$ , and additionally, the movement path should be away from a boundary as much as  $d$  data qubits. Figure 14 shows the quantum-computer architecture based on a surface code and a structured quantum-assembly code, where all the modules are arranged on the one-dimensional layout by keeping the space as much as  $\lfloor d/4 \rfloor$  data qubits between both modules.

## VI. ANALYSIS OF PERFORMANCE AND RESOURCE

We show the performance analysis results of the quantum-computing models we configure. For that, we, first of all, set the objective fidelity of quantum computing as 70%. This means that we set the error correction strength (concatenation level or code distance) to let the fidelity of a single-round quantum computing be at least 70%.

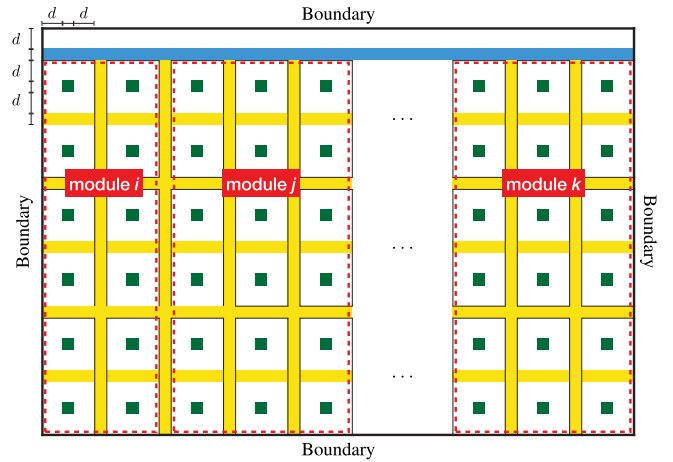


FIG. 14. Quantum-computer structure based on the surface-code quantum computing. Dark green dots indicate defects and yellow cells are used as a path for the braiding transformations. Blue cells can be used for the forward-backward qubit passings over distance modules. An enclosed section by dotted red line is a computing region, a module. A defect has to be away from the boundary of a logical qubit or a braiding path as much as  $d$  data qubits, and two logical qubits are mutually separated as much as  $\lfloor d/4 \rfloor$  data qubits.

We basically assume that the error rates of physical operations in the Steane-code quantum computing and surface-code quantum computing as  $10^{-9}$  and  $10^{-3}$ , respectively. However, we show the changes in the performance and the required resource of quantum computing as varying the physical error rates in Sec. VII. Besides, we assume that the execution time of a physical operation is  $1 \mu\text{s}$  conservatively. This assumption may be pessimistic because other literature usually takes physical gates with tens–hundreds of nanoseconds.

In the following subsections, we configure and analyze quantum computing by applying realistic factors one by one. In the beginning, we show the effect by a compile in Sec. VIA by comparing the performance between a  $R_z(\theta)$  decomposed code and a nondecomposed code. We then show that how much a fault-tolerant protocol changes the performance and the resource of quantum computing Sec. VIB. There we assume that nonlocal multiqubit interaction, i.e., a CNOT between qubits located apart from each other, can be used. On the other hand, in Sec. VIC, we analyze fault-tolerant quantum computing with the assumption that a local interaction for multiqubit operation is only allowed.

We finally recall that the benchmarks for all the analyses in this paper are Shor’s factoring algorithm [16,26,63] except the case of Fig. 18, and the plotted quantum-computing time is based on the unit of  $T_{\text{avg}}$ , which indicates the total running time of repetitive quantum-computing executions to achieve the algorithm execution fidelity of 100%.



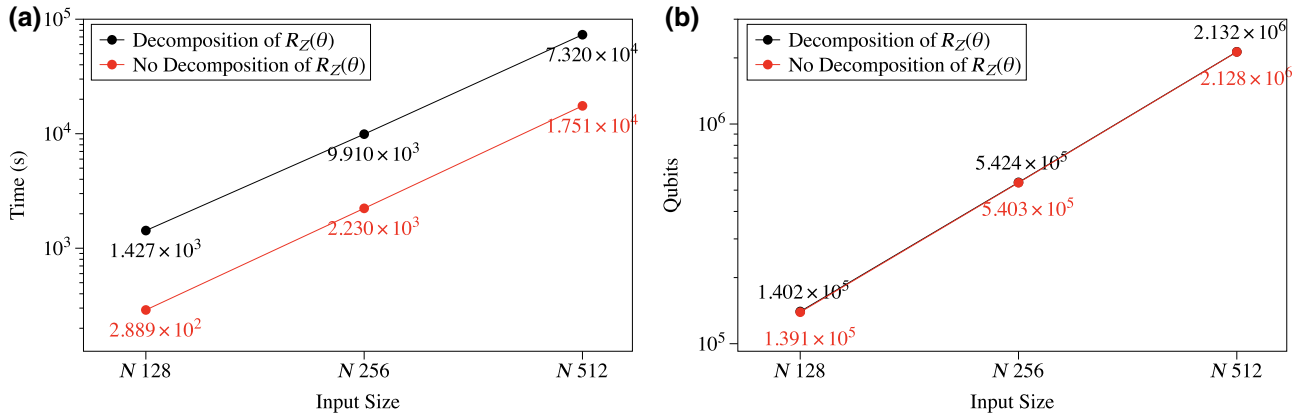


FIG. 15. We show the quantum-computing performance change by only the compile effect. (a) Quantum-computing time and (b) number of physical qubits.

### A. Case of applying compile

We show the change of the performance of quantum computing by applying quantum compile, i.e., the decomposition of a  $R_Z(\theta)$  gate into a sequence of  $H$ ,  $S$ , and  $T$  gates. Even though generally such a decomposition is strongly required to implement a fault-tolerant quantum computing, in this section we perform physical quantum computing without quantum error correction to see the effect of the compile only. For that, we assume that the components of the other layers are ideal.

We set the precision of the decomposition as  $10^{-10}$ , which means that the decomposed gate sequence of the  $R_Z(\theta)$  gate achieves the exact  $R_Z(\theta)$  operation with an error probability  $10^{-10}$ . Consequentially, both  $R_Z(\theta_1)$  and  $R_Z(\theta_2)$  can be decomposed into the same sequence of  $H$ ,  $S$ , and  $T$  if  $|\theta_1 - \theta_2| \leq 10^{-10}$ . Under such precision, a  $R_Z(\theta)$  gate is usually decomposed into a sequence of 250  $H$ ,  $S$ , and  $T$  gates [64]. Note that the decomposition algorithm works probabilistically [3,30].

Figure 15 compares the performance. By decomposing the  $R_Z(\theta)$  gate, the quantum-computing time increases as much as 4–5 times, but the number of physical qubits stays equivalently. In general, the  $R_Z(\theta)$  gate takes more than half of all quantum gates in our benchmark algorithm (see Table V). Here, on considering that the  $R_Z(\theta)$  gate is decomposed into a sequence of hundreds of  $H$ ,  $S$ , and  $T$  gates as we mention above, readers may guess that the performance difference between both cases should be larger than that shown in the figure.

TABLE V. Proportion of the  $R_Z(\theta)$  gate in Shor  $N = 128$ .

Input size	$R_Z(\theta)$	Total gates	Proportion
128	$2.036 \times 10^9$	$3.399 \times 10^9$	59.90%
256	$1.630 \times 10^{10}$	$2.719 \times 10^{10}$	59.94%
512	$1.304 \times 10^{11}$	$2.175 \times 10^{11}$	59.95%

Recall that we set the precision of the decomposition as  $10^{-10}$  above. Most  $\theta$  in Shor's factoring algorithm are very small ( $\theta = \pi/2^{n-1}$  with  $n = 1 \sim N$  for  $N$ -bit integer factoring), and therefore the decomposition of such rotation operation works as the identity operation. We show the top dominant  $\theta$  used in Shor's  $N = 128$  algorithm in Table VI. All the angles are less than  $10^{-10}$ . While we do not describe all  $\theta$  in the algorithm in the table, empirically 75% of the angles applied in the algorithm are less than  $10^{-10}$ . In this regard, the degree of the performance change by decomposing  $R_Z(\theta)$  gates is not so remarkable regardless of the quantity of  $R_Z(\theta)$  gates in the algorithm.

### B. Case of applying compile and error correction

We show the performance of quantum computing by applying a quantum error correction. For that, as mentioned above, we need to compile a quantum algorithm by decomposing the  $R_Z(\theta)$  gate into the  $H$ ,  $S$ , and  $T$  gate. Here, we concentrate only on the effect by applying

TABLE VI. List of top ten dominant angles in Shor's factoring algorithm,  $N = 128$ . The  $\theta$  listed in this table is less than 0.01 and therefore  $R_Z(\theta)$  works as an identity operator. The rotational angle  $\theta$  of the gate is from  $\pi/2^{n-1}$  in quantum Fourier transform, and the exact representation of the angle is limited by a classical computer precision.

$\theta$	Count	Proportion
$0.000\,000 \times 10^0$	$6.88 \times 10^8$	0.3381
$-0.000\,000 \times 10^0$	$3.44 \times 10^8$	0.1691
$-5.000\,000 \times 10^{-5}$	$3.13 \times 10^7$	0.0154
$-1.000\,000 \times 10^{-4}$	$3.11 \times 10^7$	0.0153
$5.000\,000 \times 10^{-5}$	$3.10 \times 10^7$	0.0152
$-2.000\,000 \times 10^{-4}$	$3.09 \times 10^7$	0.0152
$1.000\,000 \times 10^{-4}$	$3.08 \times 10^7$	0.0151
$-4.000\,000 \times 10^{-4}$	$3.08 \times 10^7$	0.0151
$2.000\,000 \times 10^{-4}$	$3.07 \times 10^7$	0.0151
$-7.500\,000 \times 10^{-4}$	$3.06 \times 10^6$	0.0150

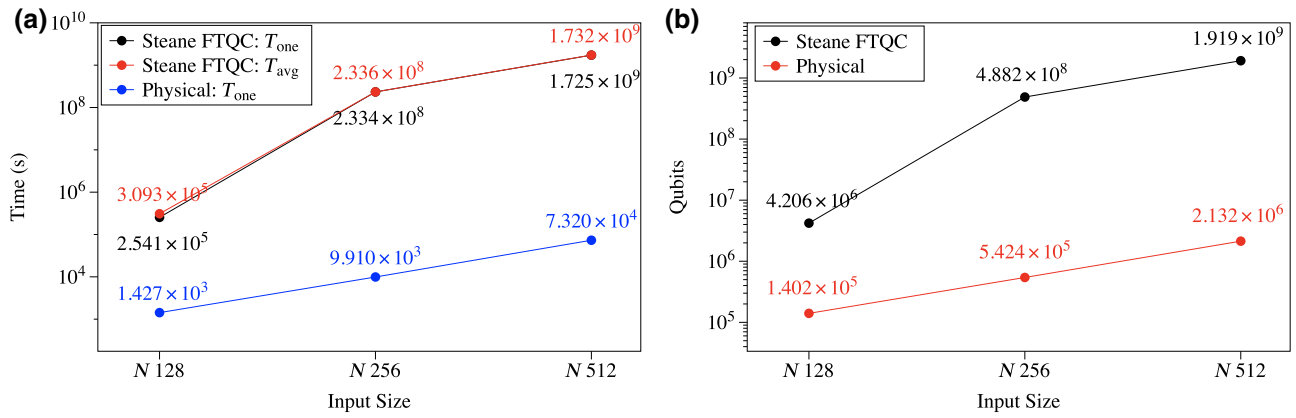


FIG. 16. Quantum-computing performance and resource by applying quantum error correction. We assume that an arbitrary long qubit interaction is allowed here. For the fault-tolerant operation, we compile a quantum algorithm by decomposing  $R_Z(\theta)$  gates into a sequence of  $H$ ,  $S$ , and  $T$  gates. In this evaluation, the quantum-computer architecture and local qubit interaction are not completely considered. (a) Quantum-computing time and (b) number of physical qubits. The concatenation level for the input size 128 is 1, and 2 for the other cases.

quantum error correction. To this end, we assume that a nonlocal interaction between distant qubits can be directly applied. By doing so, we can disregard the effect of a quantum-computer system architecture. Therefore, a communication bus and a qubit movement are not required in this case. We deal with a Steane-code quantum computing only because the surface-code quantum computing inherently takes account of the two-dimensional qubit array allowing nearest-neighbor qubit interaction.

Figure 16 shows the quantum-computing performance. The execution time and the number of qubits increase remarkably when the input size increases from 128 to 256. This is because the required concatenation level increases from 1 to 2 there to satisfy the objective fidelity of 70%. But, as the concatenation level stays as 2 when the input increases from 256 to 512, the increases of a

quantum-computing time and the number of qubits are rather modest.

As we mention above, the changes in the performance and the resource shown in the figure are only caused by the fault-tolerant quantum-computing protocol. For example, in Fig. 16(b), the numbers of qubits in the Steane-code quantum computing are bigger than physical computing as much as 30, 900, and 900 times, respectively. Recall that we design a logical qubit by assembling 30 low-level qubits.

### C. Case of applying compile, error correction, and system architecture

We analyze the quantum-computing performance by considering all the realistic factors we describe previously.

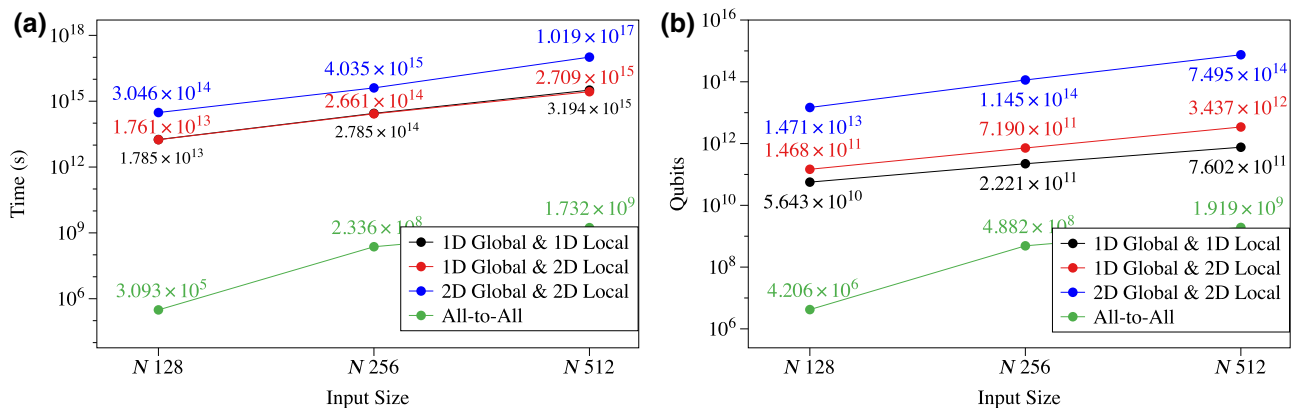


FIG. 17. Quantum-computing performance of the Steane-code-based local fault-tolerant quantum computing. To see the influence by the architectural limitation, we also add the performance when the arbitrary long qubit interaction is allowed. (a) Quantum-computing time and (b) number of physical qubits. All concatenation levels for the local qubit interaction cases (black, red, and blue lines) are 3 in common. But, in the case of the nonlocal qubit interaction (green line), the concatenation level is 1 for the input size 128 and 2 for the others. See Fig. 11 about the quantum-computer architectures.

TABLE VII. Proportion of SWAP gate in Shor's factoring algorithm. The layout indicates a combination of global layout and local layout.

Input size	Layout	SWAP	Total gates	Proportion
128	(1D, 1D)	$7.371 \times 10^{11}$	$7.405 \times 10^{11}$	99.54%
	(1D, 2D)	$1.262 \times 10^{12}$	$1.266 \times 10^{12}$	99.73%
	(2D, 2D)	$1.527 \times 10^{13}$	$1.527 \times 10^{13}$	99.97%
256	(1D, 1D)	$1.116 \times 10^{13}$	$1.118 \times 10^{13}$	99.76%
	(1D, 2D)	$1.856 \times 10^{13}$	$1.859 \times 10^{13}$	99.85%
	(2D, 2D)	$2.719 \times 10^{14}$	$2.720 \times 10^{14}$	99.99%
512	(1D, 1D)	$1.068 \times 10^{14}$	$1.070 \times 10^{14}$	99.80%
	(1D, 2D)	$1.811 \times 10^{14}$	$1.813 \times 10^{14}$	99.88%
	(2D, 2D)	$5.789 \times 10^{15}$	$5.789 \times 10^{15}$	99.99%

We apply fault-tolerant quantum computing based on certain quantum-computer architectures where only nearest neighbored qubits can interact. In this section, we deal with both the Steane code and the surface code as a fault-tolerant quantum-computing scheme.

We first show the performance and the resource of the Steane-code quantum computing by varying the quantum-computing architectures as (1D global, 1D local), (1D global, 2D local) and (2D global, 2D local). See Fig. 11 for the quantum-computer architectures. Figure 17 shows the performance and the resource of the Steane-code quantum computing. To see the influence caused by a local qubit interaction, we also compare the performance of the quantum computing based on nonlocal qubit interaction ("all-to-all") shown in the previous section.

As shown in the figure, the performance degradation by the local qubit interaction on a quantum computer architecture is highly nontrivial. This is because many modules are spread over the quantum computer, and the communication (qubit passing) are performed frequently. Table VII shows the proportion of SWAP gates in the implementation of the factoring algorithm. Surprisingly, on the proposed

quantum-computer architecture with the nearest-neighbor qubit interaction, most of the quantum operations in the Steane-code quantum computing are qubit movements rather than qubit manipulations.

We think the quantity of the qubit movements is a temporal overhead to implement a quantum algorithm on a quantum computer. Such a large overhead caused by the qubit movements can be reduced by improving a quantum-computer structure, a fault-tolerant quantum-computing scheme, or a system-synthesis algorithm. For reference, many efforts are being dedicated to how to reduce such an overhead by qubit movements [37–39,65–69].

The figure shows that a quantum-computer architecture of the 1D global layout provides better performance than a quantum computer of the 2D global layout. However, it may not always be the case. It completely depends on the number of modules in a quantum-computing program (see Fig. 7), and the arrangement of the modules on the architecture. In general, the 2D global layout is a better architecture in terms of the qubit movements when the number of modules is very large. On average, the arrangement of the modules on the 2D global layout can reduce the distance between modules more than the 1D global layout. Therefore, the communication cost of the qubit passing is less than the 1D global layout. As an example, Fig. 18 shows that a ground-state estimation (GSE) algorithm [16,17,26] works better on a quantum-computer architecture with the 2D global layout because the GSE algorithm is composed of much more modules than the factoring algorithm benchmark.

In what follows, we show the performance and the resource of the surface-code computing in Fig. 19. The quantum-computer architecture for the case is shown in Fig. 14. In the error rate  $10^{-3}$ , as the input size increases, the required code distance is raising 25, 27, and 30 to satisfy the objective fidelity. In the figure, we also show the number of physical qubits to run a magic state factory

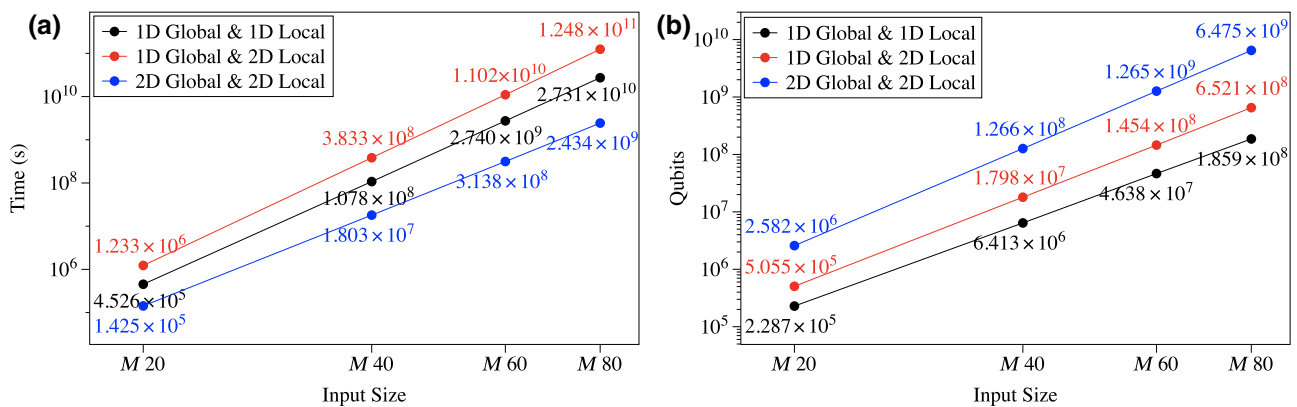


FIG. 18. Quantum-computing performance of ground-state estimation algorithms over input size  $M = 20, 40, 60, 80$ . Quantum gates are noiseless, and only nearest-neighbor qubits mutually interact on the quantum-computer architectures. (a) Quantum-computing time and (b) number of physical qubits.

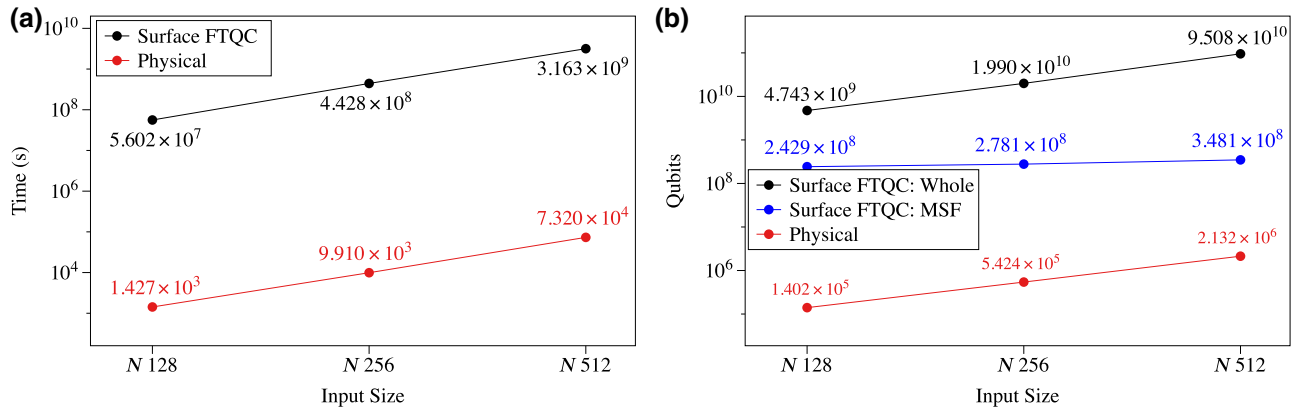


FIG. 19. Quantum-computing performance based on the surface-code quantum computing. (a) Quantum-computing time and (b) number of physical qubits. The code distances are respectively 25, 27, and 30. In (b), we also show the required physical qubits for a magic state factory.

that supplies  $|A_L\rangle$  states during the quantum computing. As shown in the figure, the capacity of a magic state factory stays almost the same regardless of the input size of the factoring algorithm. It increases as much as the code distance.

Reference [19] estimated the surface-code quantum-computing execution time of the factoring algorithm to factorize a 2000-bit integer. By following their method, the quantum-computing time to factorize a 512-bit integer is only 0.45 h ( $40 \times 512^3 \times 3 \times 100$  ns) regardless of physical error rates. But, our estimation says that  $8.78 \times$

$10^5$  h are required for the same task in the physical error rate of  $10^{-3}$ .

We believe this shocking discrepancy is caused by the quantum-computing scheme and the variant of Shor’s factoring algorithm they applied rather than our platform itself. First, Ref. [19] applies the *time-optimal quantum-computing* scheme [28] as a computing model. The scheme claims that a Clifford gate can be completed effectively zero time and a logical T gate can be completed within a single physical measurement time. Therefore, the execution time of a fault-tolerant quantum computing is independent of the error-correction strength such as a code distance. On the other hand, as we mention before, our platform counts the execution times of all quantum operations including the error correction by  $d$ -round syndrome measurements. In particular, in our platform, the execution time of a logical T gate is basically defined by  $\text{CNOT}_i + \text{Meas } Z_i + S_i/2$ , where  $S_i/2$  is added because a logical T gate requires an S-gate correction with probability of 50%. For this reason, the execution time by the time-optimal quantum computation is much less than both the previously expected and ours.

Second, the variant of the factoring algorithm examined in Ref. [19] has a small-sized quantum-assembly code than our benchmark in the 512-bit input regime. Their benchmark is composed of a Toffoli-gate-based quantum adder, but our benchmark [63] is based on a QFT adder. In terms of the circuit complexity, our benchmark is superior to their algorithm as  $2000n^2$  vs  $O(n^3)$  for the modular exponentiation. However, the QFT circuit contains many  $R_Z(\theta)$  gates, which should be decomposed into  $H$ ,  $S$ , and  $T$  gates beforehand. Such a gate decomposition from  $R_Z(\theta)$  gates rather raises a larger circuit in the Shor  $N = 512$  regime than the counterpart algorithm. From our analysis, the T-gate depth in our benchmark is 20-fold longer. See Fig. 20. In this respect, for Shor  $N = 512$ , our benchmark shows poorer performance than its counterpart.

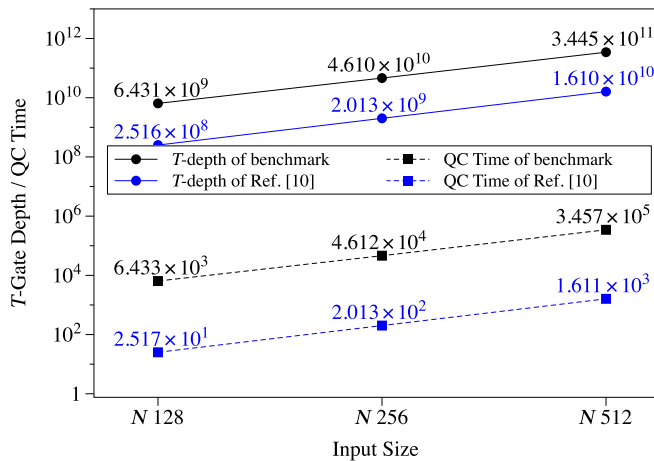


FIG. 20. Comparison in quantum-computing time and T-gate depth between our benchmark and Ref. [19]. Quantum-computing time of our benchmark is also obtained by applying a time-optimal quantum-computing scheme in our platform. The T-gate depth of our benchmark is obtained from the system layer since any rigorous analysis on the T-gate depth is not conducted in the original literature [63]. The difference in quantum-computing time between both is as much as the quantity of T-depth times 10. The constant factor 10 is caused by the difference in the execution time of a physical measurement gate.



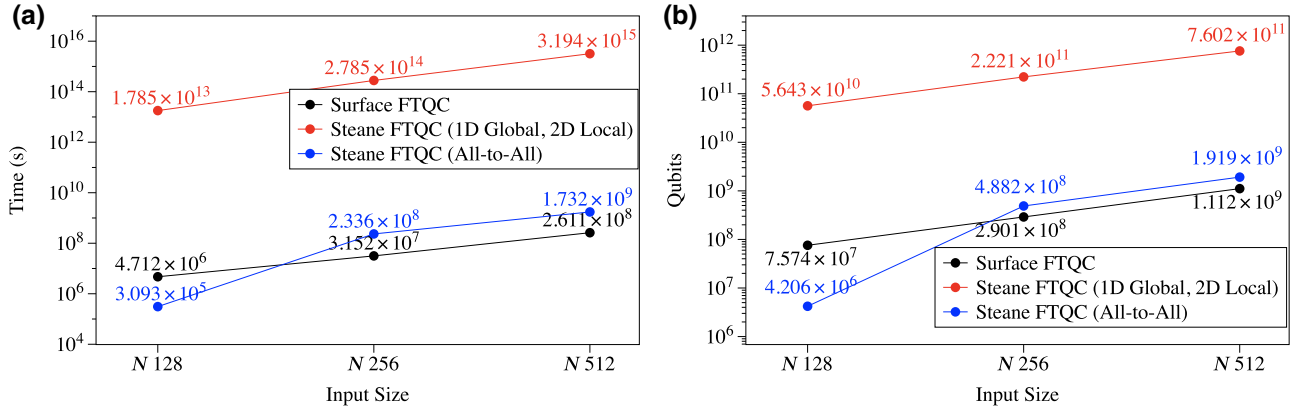


FIG. 21. We simply compare the quantum resource in the Steane-code quantum computing and the surface-code quantum computing. The physical error rate is  $10^{-9}$ . The quantum-computer architecture for the Steane-code quantum computing is the 1D global layout and the 2D local layout because as shown in Fig. 17 the architecture shows the best performance in this work. We also compare the Steane-code quantum computing with nonlocal qubit interaction because the performance of the Steane-code quantum computing significantly depends on the quantum-computer architecture.

To validate our platform, we apply the time-optimal quantum-computing scheme to our platform, and analyze the Shor algorithm. Figure 20 shows the analysis result and compares it to Ref. [19]. Our data is strikingly reduced from  $3.163 \times 10^9$  s to  $3.5 \times 10^5$  s at  $N = 512$ . Our estimation is still larger than Ref. [19] as much as 200 times. As mentioned above, there exists a discrepancy in  $T$  depth between ours and Ref. [19]. The other factor 10 is caused by the difference in the physical measurement gate running time. All of our performance analysis in this work is based on a physical gate with the execution time  $1 \mu\text{s}$ , but Ref. [19] assumes 100 ns running physical measurement gate. Therefore, considering all of the above-mentioned differences, we believe our platform provides a reasonable analysis result.

References [20,21] also estimated the running time of Shor's factoring algorithm based on surface-code FTQC. As we mention in Sec. II, they focused on the dominant part of the algorithm, quantum adder. Reference [21] determines the execution time of the algorithm by multiplying the running time of a quantum adder and the adder depth, which is the quantity of quantum adders executed in series in the factoring algorithm. By applying their method, we find that 503 h are required to execute Shor  $N = 512$ . Note that the running time of a quantum adder is  $4 \times \log_2 N \times \text{Toff}_i$  and the adder depth is  $4 \times N^2$  in their Shor benchmark algorithm. The running time of a Toffoli gate  $\text{Toff}_i$  is 48 ms there. Reference [20] estimates the execution time of the factoring algorithm by exactly following Ref. [21] except the performance of a Toffoli gate. By taking an improved Toffoli gate ( $\text{Toff}_i = 930 \mu\text{s}$ ), 9.75 h are required to run Shor  $N = 512$ .

One of the reasons why a surface code has attracted so much attention is it requires relatively less quantum resources. In what follows, we simply compare the

Steane-code FTQC and surface-code FTQC in light of quantum resources without considering their theoretical foundation. For the fair comparison, we assume the error rate of the physical device is  $10^{-9}$  for both cases. Figure 21 shows the quantum-computing time and qubits to run the factoring algorithm. As we mention before, the performance of the Steane-code quantum computing completely depends on a quantum-computer architecture. Therefore, to focus on the difference in the quantum resource only by a fault-tolerant quantum computing, we also compare the situation where a nonlocal qubit interaction is allowed. As shown in the figure, in the small input size, the Steane-code quantum computing requires less time and qubits than the surface-code quantum computing when nonlocality of a multiqubit operation can be directly applied. But, as the input size increases, the surface-code quantum computing shows better performance than the Steane-code quantum computing even a nonlocal qubit interaction is allowed.

To summarize this section, in Fig. 22, we show the quantum-computing times are increased by applying realistic quantum-computing components one by one.

## VII. USABILITY OF THE PROPOSED FRAMEWORK

The aim of the proposed methodology and platform is to help to design and analyze a quantum computing system. In this regard, in this section, we show how to exploit it for analyzing high-performance quantum-computing components. The first is an efficient compile (Sec. VII A), and the second is an improved physical gate (Sec. VII B) and the last is the strategy for the fault tolerance (Sec. VII C).

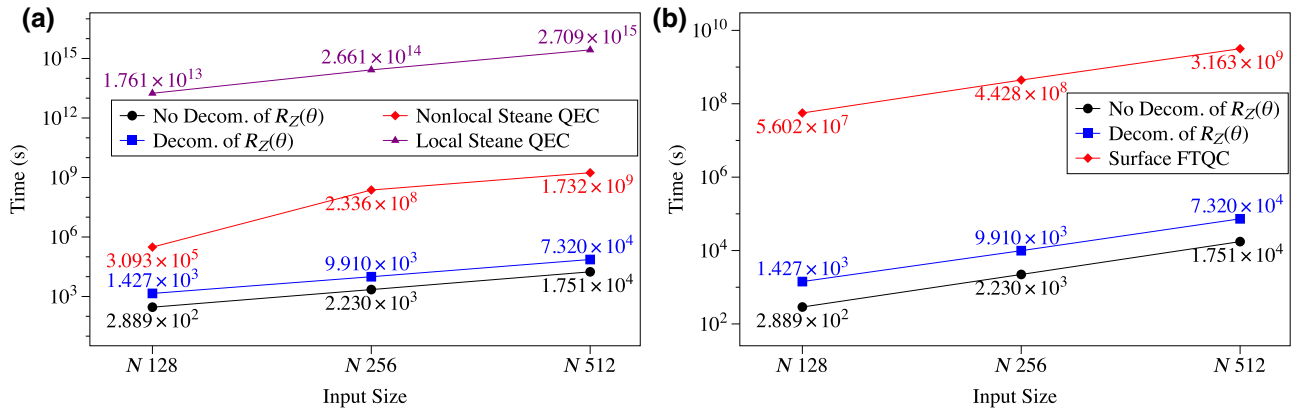


FIG. 22. Performance changes of (a) Steane-code FTQC and (b) surface-code FTQC. Both figures show the quantum-computing times are increased by applying compile, quantum error correction, local gate. Note that surface-code FTQC and Steane-code FTQC work under the physical error rate  $10^{-3}$  and  $10^{-9}$ , respectively.

### A. Efficient decomposition of controlled $R_n$

The authors proposed an efficient decomposition algorithm for a controlled- $R_n$  gate [70]. By hiring an ancilla qubit, they reduced the total number of quantum gates  $\{H, S, T\}$  from 35 (Ref. [30]) to 21. We show how the proposed compile algorithm affects the execution time of our benchmark. Even though the proposed algorithm itself requires more qubits, by reducing the size of the quantum circuit (therefore the algorithm execution time) and increasing the fidelity of quantum computing simultaneously, in total fewer qubits are used.

Figure 23 shows the performance improvement by the efficient compile in the Steane-code quantum computing. At the input size  $N = 128$ , the improved decomposition lowers the quantum-computing time as much as over 400 times and the qubits as much as 30 times. The degree of performance improvement depends on the input size. As shown in the figure, at the input size where the required concatenation level lowers by applying the

proposed decomposition, the performance improvement is remarkable.

Figure 24 shows the performance improvement by the efficient compile in the surface-code quantum computing. Unlike the Steane-code quantum computing, the performance improvement in the quantum-computing time increases gradually as the input size increases. This is because there always exists a difference in the code distance. By the improved decomposition, the required code distance lowers from 25, 27, 30 to 22, 24, 27, respectively.

### B. Accurate quantum gates

Previously, we basically assumed that the physical error rate is respectively  $10^{-9}$  for Steane-code quantum computing and  $10^{-3}$  for surface-code quantum computing. In this section, we show what happens in quantum computing if a quantum device becomes more reliable. For that, we show the performance evaluations based on the physical error

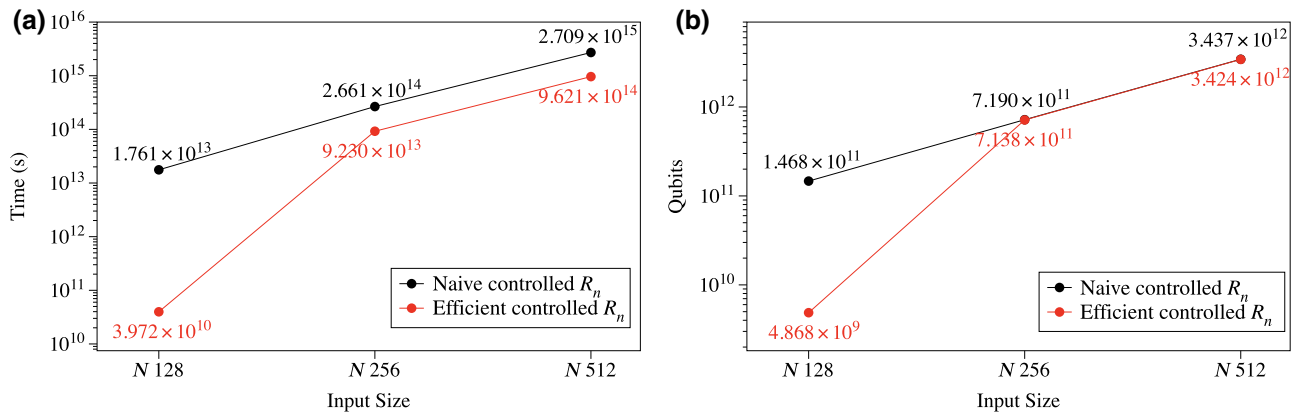


FIG. 23. Performance comparison between a naive compile and the proposed efficient compile [70] for controlled- $R_n$  gate under Steane-code-based quantum computing. (a) Quantum-computing time and (b) number of physical qubits.

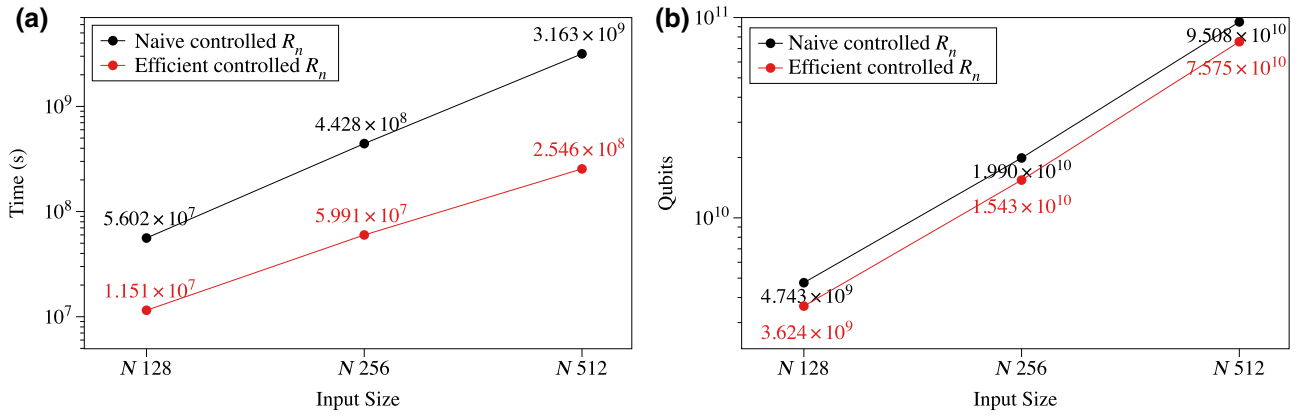


FIG. 24. Performance comparison between a naive compile and the proposed efficient compile [70] for controlled- $R_n$  gate under surface-code-based quantum computing. (a) Quantum-computing time and (b) number of physical qubits.

rates  $10^{-9}$ – $10^{-15}$  for Steane-code quantum computing and  $10^{-3}$ – $10^{-9}$  for surface-code quantum computing.

Figure 25 shows the Steane-code quantum-computing performance over physical error rates  $10^{-9}$ – $10^{-15}$ . We also compare a nonfault-tolerant physical quantum computing to those fault-tolerant quantum computing at the physical error rate  $10^{-15}$ .

The performance improvement by lowering the error rate from  $10^{-9}$  to  $10^{-12}$  is highly nontrivial because the required concatenation level is reduced from 2 and 3 to 1 in both cases, respectively. But, lowering the error rate additionally does not lead to better fault-tolerant quantum-computing performance. In other words, the fault-tolerant quantum computing in the physical error rate  $10^{-15}$  does not show any advantage against the quantum computing in the physical error rate of  $10^{-12}$ . This is because as the physical error rate lowers the fault-tolerant quantum

computing with the same concatenation level achieves very high fidelity ( $>90\%$ ). If both quantum computings are performed with the same concatenation level, both have the same single-round quantum-computing time. In that case, if there is no big difference between fidelities, the average quantum computing  $T_{\text{avg}}$  is very similar.

For the same reason, in the physical error rate  $10^{-15}$ , a nonfault-tolerant physical quantum computing shows better performance than fault-tolerant quantum computing because the physical quantum computing already achieves high fidelity. From our analysis,  $T_{\text{one}}$  of the physical quantum computing in the error rate  $10^{-15}$  is  $6.89 \times 10^8$  with the fidelity 64.33%. On the other hand,  $T_{\text{one}}$  of the fault-tolerant quantum computing is  $7.78 \times 10^{10}$  with the fidelity 99.99%. Obviously, physical quantum computing requires less quantum-computing time in terms of  $T_{\text{avg}}$ ,  $(6.89 \times 10^8)/0.6433 < (7.78 \times 10^{10})/0.9999$ .

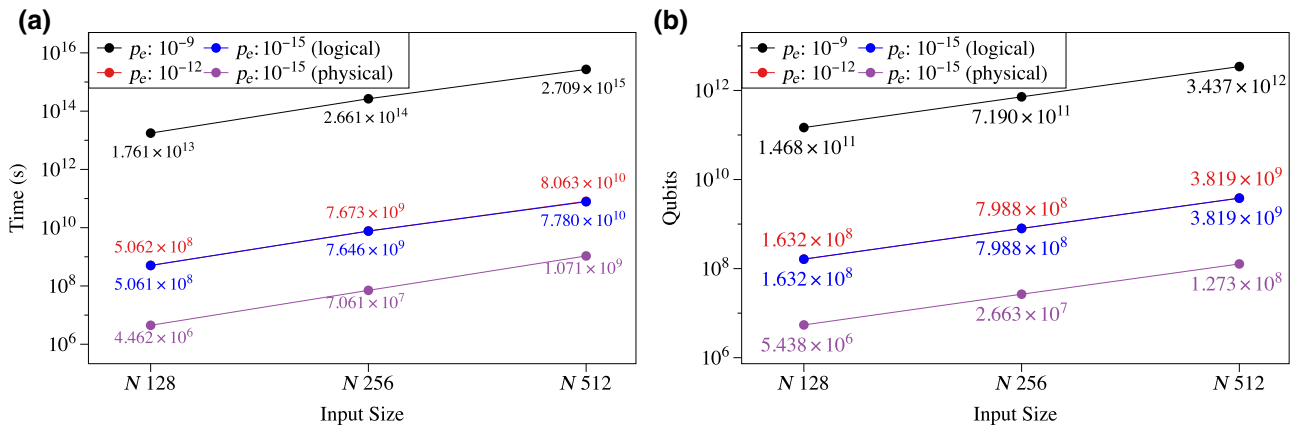


FIG. 25. Performance comparison over physical error rates  $10^{-9}$ – $10^{-15}$  under Steane-code-based quantum computing. (a) Quantum computing time and (b) number of physical qubits. At the error rate  $10^{-15}$ , as shown in this figure, fault-tolerant quantum computing is not required.

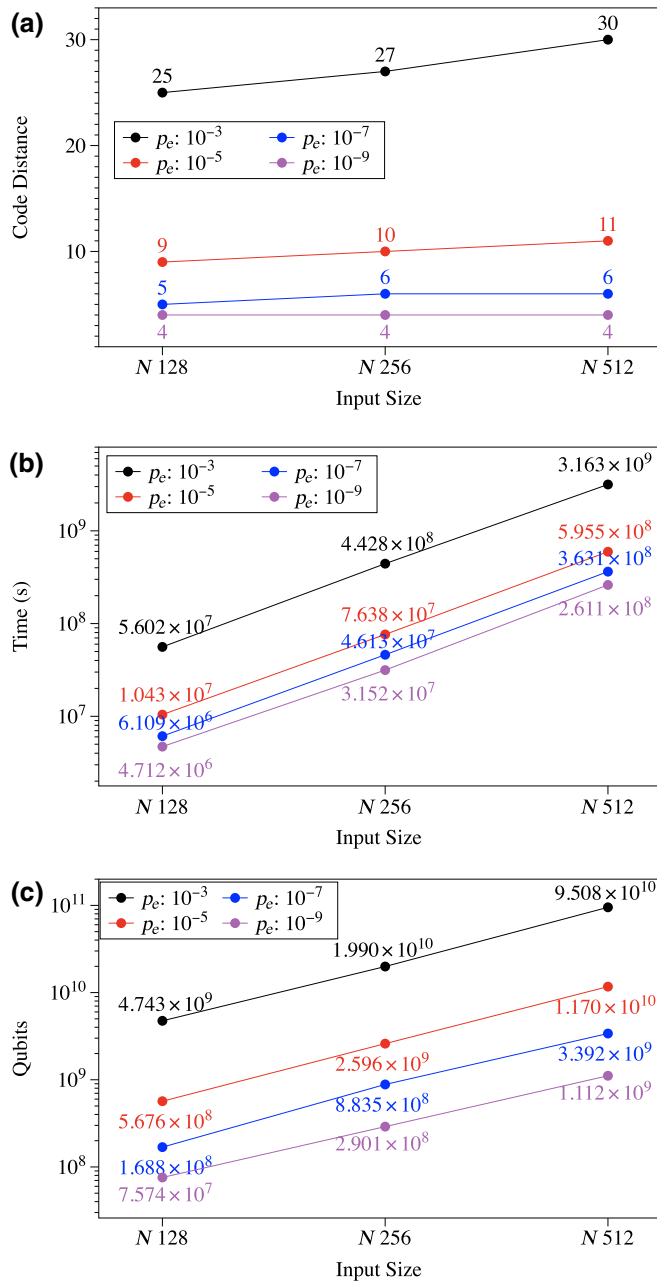


FIG. 26. Performance comparison over physical error rates  $10^{-3}$ – $10^{-9}$  under surface-code-based quantum computing. (a) Code distance, (b) quantum-computing time, and (c) number of physical qubits.

Figure 26 shows the performance improvement in the surface-code quantum computing over physical error rates  $10^{-3}$ – $10^{-9}$ . In the figure, we also compare the required code distance. As shown in the figure, as the physical error rate lowers, the required code distance decreases and therefore the performance increases. But, since the code distance is already too low, 4 or 5, there is not enough room for the performance improvement as the gate is improved more.

### C. Degree of fault tolerance

Accuracy threshold theorem [56,57] says that if we have a quantum device of physical error rate below a threshold, it is possible to achieve an arbitrary long quantum computation. By applying a recursive concatenated coding [51], we can lower the effective error rate to where a reliable quantum computing is possible. As we increase the concatenation level, the fidelity of quantum computing is definitely improved. But, the running time of quantum computing is also increased by following the increased concatenation level. Therefore, the higher concatenation level does not always make the more efficient quantum computing possible. Figure 27 shows that there exists a trade-off for the concatenation level in the Steane-code quantum computing, in particular for a quantum-computing time. Needless to say, the number of required qubits becomes larger as the concatenation level increases.

In the case of surface-code quantum computing, the performance completely depends on the code distance. The code distance is determined to satisfy the objective fidelity of quantum computing, but in most cases, the accuracy of the quantum computing by the chosen code distance exceeds the target fidelity. In this regard, considering the averaged quantum-computing time  $T_{\text{avg}}$ , the chosen code distance may not bring the best quantum-computing performance as shown in the Steane-code case.

Figure 28 shows that the surface-code quantum computing has the best performance with a code distance of 31, but the code distance determined by the equation is 30. Even though the code distance determined from the target fidelity 70% is 30, the goal of quantum computing is to find

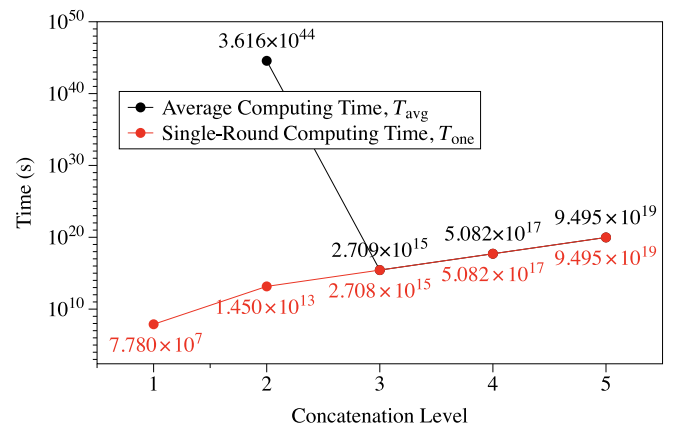


FIG. 27. Quantum computing time of Shor  $N = 512$ . We evaluate the quantum-computing performance  $T_{\text{one}}$  and  $T_{\text{avg}}$  according to the concatenation levels 1–5. After the concatenation level 3, the fidelity of quantum computing is almost 100% and therefore the average computing time closely approaches the single-round computing time. When the concatenation level is 1, the fidelity of quantum computing is almost vanishing and therefore the average computing goes to almost infinity.



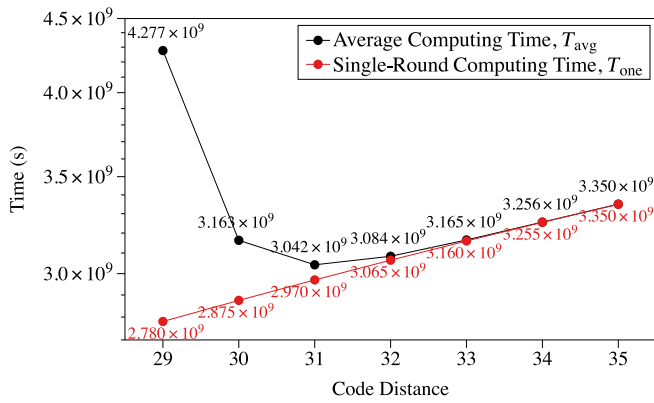


FIG. 28. Quantum-computing time of Shor  $N = 512$ . We evaluate the quantum-computing performance  $T_{\text{one}}$  and  $T_{\text{avg}}$  by varying the code distance from 29 to 35. The calculated code distance for the objective fidelity is 30. As shown in the figure, the code distance 31 introduces the best quantum-computing performance. By taking the code distance 31, we can reduce the quantum-computing time as much as 1400 days than the case of the distance 30 at the cost of qubits.

an exact answer, not a probable answer. By applying the code distance 31, we can reduce the quantum-computing time as much as 1400 days than the expected time by the code distance 30 at the cost of qubits.

## VIII. DISCUSSION

We propose an integrated method for analyzing the performance and the resource of a large-scale quantum computing. By considering practically running a quantum algorithm on a quantum computer hardware of specific system architecture, we obtain the most realistic performance and resource where the effects by all of the fully decomposed algorithm, fault-tolerant gate protocols, system architecture, and quantum device are involved. To perform such an analysis efficiently, we propose and develop a quantum-computing platform composed of three functional layers where each layer plays a definite role in quantum computing.

By exploiting the platform, we can configure a quantum-computing model by selecting specific protocols and/or properties. By doing so, we can analyze not only the performance and resource of a quantum computing but also the impact of specific components on the entire quantum computing. For example, we discuss an optimal concatenation level or code distance of fault-tolerant quantum computing. We believe such a discussion is possible due to the proposed framework.

In this work, we report that the quantity of the required qubits and the quantum-computing time are too enormous. However, based on the analysis results, we do not insist that the future of quantum computing is so pessimistic. As we mention several times, those figures completely

depend on the protocols and the architectures we employ. This means that by applying advanced technologies the analysis data is improved. For example, in Sec. VII, we show an improved compile algorithm and device can make the amount of the required resource less and the performance better. Besides, we also argue that by taking an advanced system architecture and synthesis algorithm, we can decrease the temporal overhead that happened during practical quantum computing. As quantum computing components are being improved more and more, the performance of a quantum-computing system will be better and the required resource will be less than this report.

The objective of the present work is to provide the most realistic performance and resource of quantum computing. On the other hand, we believe the proposed software platform can play a significant role in practically running quantum computing with a real quantum-computing hardware later if some components are added (see Fig. 1). For example, a classical controller to manage and control a real quantum device is required in the building-block layer. The system layer also requires functions that execute a quantum algorithm and a quantum error correction efficiently.

## ACKNOWLEDGMENTS

This work is supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (Grant No. 18ZH1400, Research and Development of Quantum Computing Platform and its Cost-Effectiveness Improvement).

- 
- [1] Cody Jones, Multilevel distillation of magic states for quantum computing, *Phys. Rev. A* **87**, 042305 (2013).
  - [2] Sreraman Muralidharan, Linshu Li, Jungsang Kim, Norbert Lütkenhaus, Mikhail D. Lukin, and Liang Jiang, Optimal architectures for long distance quantum communication, *Sci. Rep.* **6**, 20463 (2016).
  - [3] Neil J. Ross and Peter Selinger, Optimal ancilla-free Clifford+T approximation of z-rotations, *Quantum Inf. Comput.* **16**, 901 (2016).
  - [4] Héctor Bombín, Single-Shot Fault-Tolerant Quantum Error Correction, *Phys. Rev. X* **5**, 031043 (2015).
  - [5] Krysta M. Svore, David P. DiVincenzo, and Barbara M. Terhal, Noise threshold for a fault-tolerant two-dimensional lattice architecture, *Quantum Inf. Comput.* **7**, 297 (2007).
  - [6] Héctor Bombín and M. A. Martin-Delgado, Optimal resources for topological two-dimensional stabilizer codes: Comparative study, *Phys. Rev. A* **76**, 012305 (2007).
  - [7] IBM, IBM Quantum Experience, <https://quantum-computing.ibm.com>.
  - [8] Julian Kelly, Google AI Blog, <https://ai.googleblog.com>.
  - [9] Intel, Quantum Computing — Intel Newsroom, <https://newsroom.intel.com/press-kits/quantum-computing>.
  - [10] Quantum Computing Report, <https://quantumcomputing-report.com>.

- [11] Frederic T. Chong, Diana Franklin, and Margaret Martonosi, Programming languages and compiler design for realistic quantum hardware, *Nature* **549**, 180 (2017).
- [12] Martin Suchara, John D. Kubiatowicz, Arvin Faruque, Frederic T. Chong, Ching-Yi Lai, and Gerardo Paz, in *2013 IEEE International Conference on Computer Design (ICCD)* (IEEE, USA, 2013), p. 419.
- [13] Jonathan M. Smith, Neil J. Ross, Peter Selinger, and Benoit Valiron, Quipper: Concrete Resource Estimation in Quantum Algorithms, <https://arxiv.org/abs/1707.03429>, 2014.
- [14] Markus Reiher, Nathan Wiebe, Krysta M. Svore, Dave Wecker, and Matthias Troyer, Elucidating reaction mechanisms on quantum computers, *Proc. Natl. Acad. Sci.* **114**, 201619152 (2017).
- [15] Hayato Goto, Minimizing resource overheads for fault-tolerant preparation of encoded states of the Steane code, *Sci. Rep.* **6**, 19578 (2016).
- [16] Martin Suchara, Arvin Faruque, Ching-Yi Lai, Gerardo Paz, Frederic T. Chong, and John D. Kubiatowicz, Estimating the Resources for Quantum Computation with the QuRE Toolbox, Technical Report UCB/ECS-2013-119, University of California, Berkeley, 2013.
- [17] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi, ScaffCC: Scalable compilation and analysis of quantum programs, *Parallel Comput.* **45**, 2 (2015).
- [18] Damian S. Steiger, Thomas Haner, and Matthias Troyer, ProjectQ: An open source software framework for quantum computing, *Quantum* **2**, 49 (2018).
- [19] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland, Surface codes: Towards practical large-scale quantum computation, *Phys. Rev. A* **86**, 032324 (2012).
- [20] N. Cody Jones, Rodney Van Meter, Austin G. Fowler, Peter L. McMahon, Jungsang Kim, Thaddeus D. Ladd, and Yoshihisa Yamamoto, Layered Architecture for Quantum Computing, *Phys. Rev. X* **2**, 031007 (2012).
- [21] Rodney Van Meter, Thaddeus D. Ladd, Austin G. Fowler, and Yoshihisa Yamamoto, Distributed quantum computation architecture using semiconductor nanophotonics, *Int. J. Quantum Inf.* **8**, 295 (2009).
- [22] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt, in *The 7th International Conference on Post-Quantum Cryptography, PQCrypto 2016* (Springer, Japan, 2016), p. 29.
- [23] Vasileios Mavroeidis, Kamer Vishi, D. Mateusz, and Audun Jøsang, The impact of quantum computing on present cryptography, *Int. J. Adv. Comput. Sci. Appl.* **9**, 1 (2018).
- [24] Alexander S. Green, Peter Le Fanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoit Valiron, in *The 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, USA, 2013), p. 333.
- [25] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi, in *The 11th ACM Conference on Computing Frontiers* (ACM, Italy, 2014).
- [26] Ali JavadiAbhari, ScaffCC, <https://github.com/ScaffCC/ScaffCC>.
- [27] Alexander S. Green, Peter Le Fanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoit Valiron, in *International Conference on Reversible Computation 2013* (2013), p. 110.
- [28] Austin G. Fowler, Time-optimal quantum computation, <https://arxiv.org/abs/1210.4626>, 2012.
- [29] Ali JavadiAbhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Frederic T. Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd A. Brun, Scaffold: Quantum Programming Language, Technical Report TR-934-12, 2012.
- [30] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca, Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates, *Quantum Inf. Comput.* **13**, 607 (2013).
- [31] Microsoft, Quantum Development Kit, <https://www.microsoft.com/en-us/quantum/development-kit>.
- [32] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu, in *The 44th ACM SIGPLAN Symposium* (ACM Press, Paris, France, 2017), p. 818.
- [33] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, UK, 2000).
- [34] Chia-Chun Lin, Amlan Chakrabarti, and Niraj K. Jha, FTQLS: Fault-tolerant quantum logic synthesis, *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **22**, 1350 (2014).
- [35] K. M. Svore, A. V. Aho, A. W. Cross, Isaac L. Chuang, and I. L. Markov, A layered software architecture for quantum computing design tools, *Computer* **39**, 74 (2006).
- [36] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta, Open Quantum Assembly Language, <https://arxiv.org/abs/1707.03429>, 2017.
- [37] Gian Giacomo Guerreschi and Jongsoo Park, Two-step approach to scheduling quantum circuits, *Quantum Sci. Technol.* **3**, 045003 (2018).
- [38] Alwin Zulehner, Alexandru Paler, and Robert Wille, An efficient methodology for mapping quantum circuits to the IBM QX architectures, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **38**, 1226 (2019).
- [39] Gushu Li, Yufei Ding, and Yuan Xie, in *Architectural Support for Programming Languages and Operating Systems, ASPLOS'19* (ACM, USA, 2019), p. 1001.
- [40] Depending on qubit-passing technologies, the qubit-passing cost may be trivial or nontrivial. Concatenated-code-based quantum computing performs the qubit movement by sequentially swapping data qubits (SWAP-based movement) or ancilla qubits (teleportation-based movement) between source and target. Swapping data qubits directly requires quantum error correction, but error detection may be enough for the ancilla qubit swapping. On the other hand, a double-defect surface code can move the qubits very efficiently by performing only measurement gates once.
- [41] Andrew M. Steane, Overhead and noise threshold of fault-tolerant quantum error correction, *Phys. Rev. A* **68**, 042322 (2003).
- [42] Peter W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Comput.* **25**, 1484 (1997).

- [43] Lov K. Grover, in *ACM symposium on Theory of Computing 1996* (ACM, USA, 1996), p. 212.
- [44] Peter W. Shor, Scheme for reducing decoherence in quantum computer memory, *Phys. Rev. A* **52**, R2493 (1995).
- [45] Emanuel Knill and Raymond Laflamme, Theory of quantum error-correcting codes, *Phys. Rev. A* **55**, 900 (1997).
- [46] A. R. Calderbank and Peter W. Shor, Good quantum error-correcting codes exist, *Phys. Rev. A* **54**, 1098 (1997).
- [47] Andrew Steane, Multiple-particle interference and quantum error correction, *Proc. R. Soc. A: Math., Phys. Eng. Sci.* **452**, 2551 (1996).
- [48] Daniel Gottesman, PhD thesis, California Institute of Technology, 1997.
- [49] Peter W. Shor, in *FOCS Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (IEEE, USA, 1996), p. 56.
- [50] Daniel Gottesman, Theory of fault-tolerant quantum computation, *Phys. Rev. A* **57**, 127 (1998).
- [51] Emanuel Knill and Raymond Laflamme, Concatenated Quantum Codes, <https://arxiv.org/abs/quant-ph/9610011>, 1996.
- [52] Emanuel Knill, Raymond Laflamme, and Wojciech H. Zurek, Resilient quantum computation, *Science* **279**, 342 (1998).
- [53] Austin G. Fowler, Ashley M. Stephens, and Peter Groszkowski, High-threshold universal quantum computation on the surface code, *Phys. Rev. A* **80**, 052312 (2009).
- [54] Yaakov S. Weinstein and Sidney D. Buchbinder, Use of Shor states for the  $[[7,1,3]]$  quantum error-correcting code, *Phys. Rev. A* **86**, 052336 (2012).
- [55] Y. S. Weinstein, in *SPIE Sensing Technology + Applications* (SPIE, Baltimore Maryland, USA, 2015), p 95000Q.
- [56] E. Knill, Raymond Laflamme, and W. Zurek, Threshold Accuracy for Quantum Computation, <https://arxiv.org/abs/quant-ph/9610011>, 1996.
- [57] Dorit Aharonov and Michael Ben-Or, Fault-tolerant quantum computation with constant error rate, *SIAM J. Comput.* **38**, 1 (2008).
- [58] D. S. Wang, Austin G. Fowler, A. M. Stephens, and Lloyd C. L. Hollenberg, Threshold error rates for the toric and planar codes, *Quantum Inf. Comput.* **10**, 456 (2010).
- [59] Robert Raussendorf and Jim Harrington, Fault-Tolerant Quantum Computation with High Threshold in Two Dimensions, *Phys. Rev. Lett.* **98**, 190504 (2007).
- [60] Austin G. Fowler, Low-overhead surface code logical hadamard, *Quantum Inf. Comput.* **12**, 970 (2012).
- [61] Austin G. Fowler, Simon J. Devitt, and Cody Jones, Surface code implementation of block code state distillation, *Sci. Rep.* **3**, 1939 (2013).
- [62] It is possible to perform a fault-tolerant braiding between very distant logical qubits in different modules. On considering that, the qubit passings may not be required. Therefore, a braiding between distant logical qubits requires so many physical measurements, which may decrease the reliability of quantum computing. In this regards, we believe that a braiding operation between nearby qubits after moving qubits closer to the target qubits can be performed more reliably. Therefore, we also perform the qubit passings in the surface-code quantum computing.
- [63] Archimedes Pavlidis and Dimitris Gizopoulos, Fast quantum modular exponentiation architecture for shor's factoring algorithm, *Quantum Inf. Comput.* **14**, 649 (2013).
- [64] If we set the precision degree with a smaller number, we get a longer sequence of  $H$ ,  $S$ , and  $T$  gates. Such a sequence can achieve the target  $R_z(\theta)$  gate more exactly. However, the quantum-computing time is larger than the time shown in this work. Besides, practically the duration to conduct the performance analysis also increases nontrivially.
- [65] Ali Javadi-Abhari, Pranav Gokhale, Adam Holmes, Diana Franklin, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong, in *The 50th Annual IEEE/ACM International Symposium* (ACM, Cambridge Massachusetts, 2017), p. 692.
- [66] Dmitri Maslov, Sean M. Falconer, and Michele Mosca, Quantum circuit placement, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **27**, 752 (2008).
- [67] Yuichi Hirata, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima, in *2009 Third International Conference on Quantum, Nano and Micro Technologies (ICQNM)* (IEEE, Mexico, 2009), p. 26.
- [68] Marcos Siraichi, Vinicius Fernandes Dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira, in *2018 International Symposium on Code Generation and Optimization* (ACM, Austria, 2018), p. 113.
- [69] L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. G. Almudever, Mapping of lattice surgery-based quantum circuits on surface code architectures, *Quantum Sci. Technol.* **4**, 015005 (2019).
- [70] Taewan Kim and Byung-Soo Choi, Efficient decomposition methods for controlled- $R_n$  using a single ancillary qubit, *Sci. Rep.* **8**, 5445 (2018).