Resource-Aware Deadlock Freedom for Distributed Quantum Programs

Ryo Wakizaka[1] * Shin Nishio[2] †

[1] *Graduate of Informatics, Kyoto University, Japan*
[2] *Department of Informatics, School of Multidisciplinary Science, SOKENDAI (The Graduate University for Advanced Studies), Tokyo, Japan*

Abstract. A distributed quantum program consists of concurrent processes with remote operations that consume entangled resource states. The number of communication qubits for storing the shared resource state is limited, so some processes may need to wait for its release. This may cause deadlock, where some of the processes wait for each other and then get stuck. To avoid wasting the runtime of the distributed systems, it is necessary to verify deadlock freedom for distributed quantum programs.

We propose a type-based approach to resource-aware deadlock verification for distributed quantum programs. With the proposed approach, distributed quantum programs that have passed type checking can be guaranteed to be deadlock-free. We also discuss relative topics such as scheduling and routing for entanglement generation.

Keywords: Distributed Quantum Computing, Quantum Programming Languages, Type Systems, Verification

1 Introduction

Various quantum hardware developments are underway to exploit the advantages of the computational complexity of quantum computation [1, 2, 3, 4].

While the number of qubits in a processor and the accuracy of gate operations are improving, experimental results are showing physical constraints on the size of the processor [5, 6, 7]. Consequently, research has focused on interconnecting multiple quantum processors to form large-scale distributed quantum computer clusters.

In general, it is difficult for users to write distributed programs without introducing bugs, such as deadlocks, caused by concurrent execution. Writing programs manually for hardware with many constraints is also difficult since memory management is not an easy problem for humans. To alleviate this burden on users, research and development of software for distributed quantum computation have been active in recent years. For example, distributed quantum compilers are designed to convert monolithic programs into distributed programs automatically. Frameworks such as QMPI [8, 9] have also been developed to support the easy writing of distributed quantum programs by humans.

Verifying that a given program can be executed without runtime errors is also a fundamental task to accelerate the development and utilization of distributed quantum computers. This is because large-scale distributed quantum programs require several hours or days [4, 10, 3] to run, and therefore, a runtime error is a harmful problem for developments.

One of the key tasks in distributed program verification is deadlock detection. A deadlock is a situation in which a distributed program gets stuck when multiple concurrent processes are waiting for each other. This usually happens when there is a circular dependency at the instruction level between processes running in parallel. Additionally, in distributed quantum computing, multiple processes may compete for entangled resources, and some processes may need to wait to be supplied, which causes deadlocks (see Section 2).

Despite the importance of the verification of distributed quantum programs, to the best of our knowledge, there are currently no tools available for performing resource-aware deadlock verification for distributed quantum programs.

We propose a type-based approach to verifying statically that distributed quantum programs do not terminate abnormally due to deadlocks. Type systems such as behavioral types and session types [11, 12, 13] have been widely used to verify the deadlock freedom of classical distributed programs. We aim to extend these methods to distributed quantum programs and guarantee resource-aware deadlock freedom.

In addition, we will discuss cooperation among the instruction set of distributed quantum programs, static verification tools, and routing algorithms for entanglement generation. Specifically, we will examine how the cost of static verification and software to control distributed systems varies depending on the degree to which the user has self-regulation over routing for quantum communication. While these discussions are crucial for implementing distributed quantum computing, to the best of our knowledge, they have not been fully explored to date.

2 Background

2.1 Execution Model of Distributed Quantum Computing

A distributed quantum computer consists of multiple interconnected quantum computer nodes. Each node has data qubits for local operations and communication qubits to store shared entangled states for remote quantum operations between nodes. Each node has a limited number of data and communication qubits, and the program must ensure that the number of qubits used does not exceed this limit. For simplicity of discussion, we

*wakizaka@fos.kuis.kyoto-u.ac.jp
†parton@nii.ac.jp

assume that there are a sufficient number of data qubits but a very small number of communication qubits.

Interconnects distribute entangled states, which can be consumed to perform remote operations on qubits in different nodes. For example, Figure 1 shows a quantum circuit that uses a Bell pair to realize the remote CX gate between processors 1 and 2.

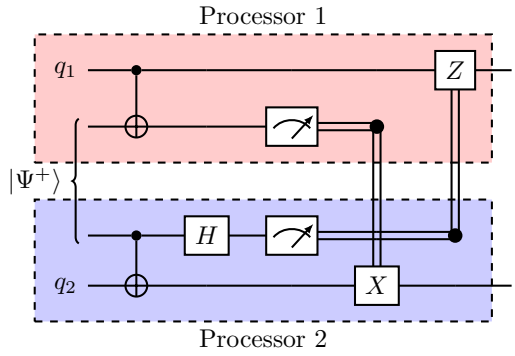

Figure 1: A remote CX gate.

The connection between nodes is not necessarily a fully connected graph, but a procedure called *entanglement swapping* [14] can create entanglement states between nodes that are not directly connected. Entanglement swapping generates the desired entanglement by selecting a path between nodes and consuming the entanglement along that path.

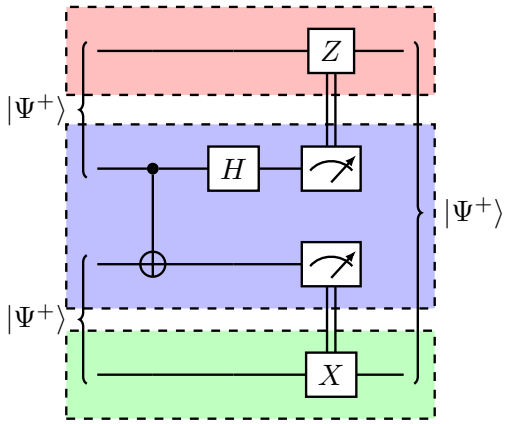

Figure 2: Entanglement swapping.

2.2 Motivating Example

As explained in the previous section, entangled states play an important role in distributed quantum computation, but the number of communication qubits available to each node is extremely limited. Consequently, the progress of the program may get stuck as concurrent processes compete for entangled resources. This section outlines this problem with an example.

Consider a distributed quantum computer as represented in Figure 3. This distributed quantum computer has four computation nodes connected linearly, and each link has the capacity to share one Bell pair in two processors. We would like to execute the distributed quantum program shown in this computer in Figure 4. In this program, the **reqlink** instruction requests the system to generate entanglement using links on the specified path from the current node. Additionally, the **release** instruction drops the ownership of qubit x, allowing the system to reuse that qubit.

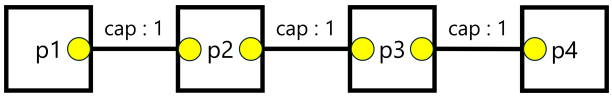

Figure 3: An Example of distributed quantum computer

```
// Process A on node p1
let x = reqlink[p1,p2,p3,p4]();
let v = /* compute with x */
release x;
send[p2](v); // sends v to p2 (non-blocking)
```

```
// Process B on node p2
let y = reqlink[p2,p3]();
let v = recv[p1](); // receive v from p1 (
    blocking)
/* compute with y and v */
release y;
```

Figure 4: Distributed quantum programs on nodes p1 and p2.

In this example, the requests in the first line of Processes A and B access the link between node 2 and node 3, but they are not directly dependent on each other. Therefore, it is non-deterministic which of them is executed first. If Process A's request is executed first, the program will obviously be completed safely. On the other hand, if Process B's request is executed first, these processes terminate in a deadlock state as follows:

1. (Process B) The request is processed, and the entanglement y between p_1 and p_2 is generated.
2. (Process B) Waits for (classical) data v to be sent from process A.
3. (Process A) Before computing and sending v, it is necessary to wait for y in process B to be released.
4. Deadlock occurs between Processes A and B.

As can be seen from this example, even if the dependencies of communication instructions are not circular, a deadlock condition can still occur due to the contention for entangled states, which are limited in number.

3 Our Approach

In this section, we propose a method to statically verify deadlock freedom, as described in the previous section, by

applying behavioral types [13, 15]. For readers unfamiliar with the type system, we omit detailed rules as much as possible and explain by gradually adding types to the example program used in Figure 4.

The basic idea of deadlock verification by type systems is to keep track of the type level of time constraints on the instructions at which quantum and classical communications occur. For example, Figure 4 contains three communication instructions **reqlink**, **send**, and **recv**, and we can assign time variables to each communication as shown in Figure 5.

```
// Process A on node p1
let x = reqlink[p1,p2,p3,p4](); // (t1, t2)
let v = ...;
release x;    // t2
send[p2](v); // t3
```

```
// Process B on node p2
let y = reqlink[p2,p3](); // (t4, t5)
let v = recv[p1](); // t6
...;
release y; // t5
```

Figure 5: Adding times variables to communications.

First, **reqlink** is assigned two time variables (t_a, t_b). This means that the process will obtain an entanglement at time t_a, and its ownership is discarded at time t_b. Other classical communications, such as **recv** and **send** instructions, are assigned a single variable, which simply means that the instruction is issued at that time.

We use these variables to analyze (1) the order relation R_1 on the time variables, which is always observed when the program is executed, and (2) the time constraint R_2 (which is also an order relation) to prevent deadlocks from occurring. The ordering relation R_1 represents the topological order in which each instruction of the program is executed. That is, if $t_1 < t_2 \in R_1$, then the instruction corresponding to t_1 is always executed before the instruction corresponding to t_2. Conversely, if $t_1 < t_2 \notin R_1$, it means that it is not known which instruction will be executed first until execution. The ordering relation R_2 signifies that a deadlock will occur if the program executes instructions in an order that violates R_2.

From the definitions of R_1 and R_2, we can guarantee that deadlocks do not occur when the following proposition holds:

$$t_1 < t_2 \in R_2 \Rightarrow t_1 < t_2 \in R_1$$

In the following sections, we describe how to obtain the ordinal relations R_1 and R_2.

3.1 R_1 : Dependency Analysis

Obtaining the relation R_1 is straightforward. Essentially, we can analyze the program sequentially and order it so that the time variable of the subsequent instruction is larger. For example, in Figure 5, we can obtain the order relations $t_1 < t_2, t_2 < t_3, t_4 < t_6$, and $t_6 < t_5$.

In addition to this, because of the existence of communication instructions in distributed programs, it is necessary to analyze dependencies across processes. We perform this analysis by attaching the type U defined below to the classical communication channel c_{ij} between nodes p_i and p_j.

$$U \coloneqq \mathbf{end} \mid I(t); U \mid O(t); U$$

In this definition, $I(t)$ and $O(t)$ denote that the receive or send instruction occurs at time t, respectively. If communication occurs more than once through the same channel, we express such communication as in $I(t_1); O(t_2); \ldots$, which means that the process receives a value first, followed by a sending operation. In Figure 5, $\mathbf{send}[p_2](v)$ gives the type $c_{12} : O(t_3)$ in process A. Similarly, the type of the channel c_{12} in process B is $c_{12} : I(t_6)$.

The typing of classical communication channels is first done process by process, and finally, the dependency is obtained by integrating the type information of the entire distributed process. In this example, the channel c_{12} is typed $O(t_3)$ in process A and $I(t_6)$ in process B. Since the sending instruction always precedes the receiving instruction, we obtain the ordering relation $t_3 < t_6$.

Following the discussion up to this point, we finally obtain the order relation $R_1 = \{t_1 < t_2, t_2 < t_3, t_4 < t_6, t_6 < t_5, t_3 < t_6\}$. Finally, a (partially) typed program is shown in Figure 6.

```
// Process A on node p1
let x = reqlink[p1,p2,p3,p4](); // (t1, t2)
let v = ...;
release x;    // t2
send[p2](v); // c12: O(t3)
```

```
// Process B on node p2
let y = reqlink[p2,p3](); // (t4, t5)
let v = recv[p1](); // c12: I(t6)
...;
release y; // t5
```

Figure 6: Adding channel types for classical communication.

3.2 R_2 : Resource-Aware Analysis of Quantum Communication

In the following description, we will write e_{ij} for the quantum communication channel between nodes p_i and p_j, similar to the classical channel.

Whenever a deadlock occurs due to waiting for entanglement generation, the communication qubits of a process are exhausted. For example, in Figure 6, the quantum communication channel e_{23} is used by process B and continues until the last **release** y. We define the region where the number of unused communication qubits for e_{ij} is zero as the *critical section about* e_{ij}.

If there is a receive instruction in a critical section of e_{ij} and it requires a link of e_{ij} for the calculation of the

value to be sent in another process, the calculation must be completed before the last **reqlink** instruction using e_{ij} is processed. This is because the receiving instruction is a blocking communication and continues to prevent other processes requiring e_{ij} from running.

To express this time constraint, we add to the communication instructions in the critical section, at the type level, the time information of the quantum communication on which it depends. First, when **reqlink**$[\vec{p}]$ is present in the program, we keep the first mentioned time variable pair (t_a, t_b) at the type level for all e_{ij} specified by the path. We will write this as $e_{ij} : (t_a, t_b)$. Next, add $e_{ij} : t_a$ to c_{kl} as type information for the receiving instruction using c_{kl} that appears in the critical section of e_{ij}[1]. The updated type of c_{kl} is written as $I(t)[e_{ij} : t_a]$.

Also, if there is a transmission instruction using c_{ij} and so far we have used quantum communication channels associated with node p_j (i.e., channels denoted as e_{kj} for some k), then from their type information $(e_{kj} : (t_{a,kj}, t_{b,kj}))$, $t_{a,kj}$ is taken out and added to the type of c_{ij}. The updated type based on the discussion up to this point is Figure 7.

```
// Process A on node p1
let x = reqlink[p1,p2,p3,p4](); // e12: (t1, t2
    ), e23: (t1, t2), e34: (t1, t2)
let v = ...;
release x;    // t2
send[p2](v); // c12: O(t3)[e23: t1]
```

```
// Process B on node p2
let y = reqlink[p2,p3](); // e23: (t4, t5)
let v = recv[p1](); // c12: I(t6)[e23: t4]
...;
release y; // t5
```

Figure 7: Adding the type of quantum communication channels.

Finally, when integrating the type information for the entire distributed program, if $c_{ij} : I(t_1)[e_{kl} : t_2]$ corresponds to $c_{ij} : O(t_3)[e_{kl} : t_4]$, we add $t_2 < t_4$ to R_2. In the above example, we get $R_2 = \{t_1 < t_4\}$.

3.3 Checking $R_2 \subseteq R_1$

Finally, we verify that the propositions stated at the beginning of this section hold. The R_1 and R_2 obtained so far are written again as follows.

$$R_1 = \{t_1 < t_2, t_2 < t_3, t_4 < t_6, t_6 < t_5, t_3 < t_6\}$$
$$R_2 = \{t_1 < t_4\}$$

In this case, we can determine that this program may deadlock since $t_1 < t_4 \in R_2$ but not $t_1 < t_4 \in R_1$. To resolve the deadlock, we can explicitly create a dependency by inserting an instruction to wait for other processes, for example:

[1]We need not add t_b in this time.

```
// Process A on node p1
let x = reqlink[p1,p2,p3,p4]();
let v = ...;
release x;
send[p2](); // Notify to process B.
send[p2](v);
```

```
// Process B on node p2
let _ = recv[p1](); // Wait for process A
let y = reqlink[p2,p3]();
let v = recv[p1]();
...;
release y;
```

Figure 8: Avoiding the deadlock by inserting the waiting instruction.

4 Discussion

Practically, whether a deadlock due to lack of communication qubits occurs depends on how the system schedules the entanglement generation. In this section, we discuss the relationship between the routing algorithm for entanglement generation and the behavior of distributed quantum programs and how this relationship affects the verification of deadlocks. Note that the routing protocol also plays an important role in quantum networking [16, 17].

Routing strategies for entanglement generation can be broadly divided into two categories: *static routing* and *dynamic routing*, as follows.

Static routing: The timing and path for generating entanglements are specified in the program. For example, the program in Figure 4 assumes static routing because the **reqlink** instruction specifies a path. Several existing distributed quantum compilers [18, 19] perform compile-time optimizations, including the routing of entanglement paths.

Dynamic routing: The program specifies to which node the entanglement is to be generated but does not specify the timing or path. The request to generate an entanglement is added to the queue of the system that manages the distributed computers. The system can select the timing and path to generate the entanglement at runtime based on the information of the node pairs in the queue.

In terms of deadlock verification, it is easier to perform verification assuming static routing. With dynamic routing, it is not known which path will be selected until runtime, so all possible paths must be verified, which is extremely costly. Thus, it may not scale for the large quantum applications that we aim to run with distributed quantum computation.

On the other hand, dynamic routing has the advantage of flexible routing using information that is not known until execution. For example, in a program that includes instructions for which the time to complete the computation is not known until runtime, such as high-fidelity

entanglement generation, it may make sense to choose the path to be used depending on the situation. However, in the case of dynamic routing, the routing algorithm must manage communication qubits so that the entire system does not halt regardless of future requests made by the program. This is generally difficult to achieve.

To achieve a balance between the advantages of dynamic and static routing, it is necessary to establish some assumptions in advance between the compiler, including the verifier, and the software that manages a distributed quantum system. One example is to allow the system to select paths only from among the shortest paths, which are determined in advance at runtime. This can somewhat reduce the cost of program verification while retaining some flexibility in the system. Determining the best assumptions for the efficient operation of a distributed quantum computer remains an open problem, and we believe it is necessary to discuss this issue in the future.

5 Conclusion

In this paper, we propose an approach to deadlock verification of distributed quantum programs running on distributed quantum computers where the number of communication qubits is limited. Our approach is primarily based on behavioral types used for deadlock verification of classical distributed programs. We also discuss the relationship between deadlock verification techniques, such as our proposed approach, and how the system schedules entanglement generation.

Much future work remains to be done in this research. First, the mathematical correctness of the proposed method (that is, type safety) has not yet been proved, and we plan to do this first. Another future task is to implement the proposed method into a distributed quantum compiler and demonstrate its usefulness. The implementation will be incorporated into InQuIR [20], which is an intermediate representation for distributed quantum computers.

References

[1] R. Babbush, C. Gidney, D. W. Berry, N. Wiebe, J. McClean, A. Paler, A. Fowler, and H. Neven, "Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity," *Physical Review X*, vol. 8, p. 041015, Oct. 2018.

[2] J. Lee, D. W. Berry, C. Gidney, W. J. Huggins, J. R. McClean, N. Wiebe, and R. Babbush, "Even More Efficient Quantum Computations of Chemistry Through Tensor Hypercontraction," *PRX Quantum*, vol. 2, p. 030305, July 2021.

[3] M. E. Beverland, P. Murali, M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo, "Assessing requirements to scale to practical quantum advantage," Nov. 2022.

[4] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits," *Quantum*, vol. 5, p. 433, Apr. 2021.

[5] S. Krinner, S. Storz, P. Kurpiers, P. Magnard, J. Heinsoo, R. Keller, J. Lütolf, C. Eichler, and A. Wallraff, "Engineering cryogenic setups for 100-qubit scale superconducting circuit systems," *EPJ Quantum Technology*, vol. 6, pp. 1–29, Dec. 2019.

[6] A. Gold, J. P. Paquette, A. Stockklauser, M. J. Reagor, M. S. Alam, A. Bestwick, N. Didier, A. Nersisyan, F. Oruc, A. Razavi, B. Scharmann, E. A. Sete, B. Sur, D. Venturelli, C. J. Winkleblack, F. Wudarski, M. Harburn, and C. Rigetti, "Entanglement across separate silicon dies in a modular superconducting qubit device," *npj Quantum Information*, vol. 7, pp. 1–10, Sept. 2021.

[7] S. Tamate, Y. Tabuchi, and Y. Nakamura, "Toward realization of scalable packaging and wiring for large-scale superconducting quantum computers," *IEICE Transactions on Electronics*, vol. advpub, p. 2021SEP0007, 2021.

[8] T. Häner, D. S. Steiger, T. Hoefler, and M. Troyer, "Distributed Quantum Computing with QMPI," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, Nov. 2021.

[9] Y. Shi, T. Nguyen, S. Stein, T. Stavenger, M. Warner, M. Roetteler, T. Hoefler, and A. Li, "A Reference Implementation for a Quantum Message Passing Interface," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, (New York, NY, USA), pp. 1420–1425, Association for Computing Machinery, Nov. 2023.

[10] N. Yoshioka, T. Okubo, Y. Suzuki, Y. Koizumi, and W. Mizukami, "Hunting for quantum-classical crossover in condensed matter problems," Oct. 2022.

[11] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *Programming Languages and Systems* (C. Hankin, ed.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 122–138, Springer, 1998.

[12] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *ACM SIGPLAN Notices*, vol. 43, pp. 273–284, Jan. 2008.

[13] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro, "Foundations of Session Types and Behavioural Contracts," *ACM Computing Surveys*, vol. 49, pp. 3:1–3:36, Apr. 2016.

[14] M. Żukowski, A. Zeilinger, M. A. Horne, and A. K. Ekert, ""Event-ready-detectors" Bell experiment via entanglement swapping," *Physical Review Letters*, vol. 71, pp. 4287–4290, Dec. 1993.

[15] N. Kobayashi, "Type-based information flow analysis for the π-calculus," *Acta Informatica*, vol. 42, pp. 291–347, Dec. 2005.

[16] R. Van Meter, T. Satoh, T. D. Ladd, W. J. Munro, and K. Nemoto, "Path selection for quantum repeater networks," *Networking Science*, vol. 3, pp. 82–95, 2013.

[17] R. Van Meter, R. Satoh, N. Benchasattabuse, K. Teramoto, T. Matsuo, M. Hajdušek, T. Satoh, S. Nagayama, and S. Suzuki, "A quantum internet architecture," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 341–352, IEEE, 2022.

[18] D. Cuomo, M. Caleffi, K. Krsulich, F. Tramonto, G. Agliardi, E. Prati, and A. S. Cacciapuoti, "Optimized compiler for Distributed Quantum Computing," *arXiv:2112.14139 [quant-ph]*, Dec. 2021.

[19] D. Ferrari, A. S. Cacciapuoti, M. Amoretti, and M. Caleffi, "Compiler Design for Distributed Quantum Computing," *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–20, 2021.

[20] S. Nishio and R. Wakizaka, "InQuIR: Intermediate Representation for Interconnected Quantum Computers," Feb. 2023.