Parallel Simulation of Quantum Networks with Distributed Quantum State Management

XIAOLIANG WU, Illinois Institute of Technology, United States
ALEXANDER KOLAR, University of Chicago, United States
JOAQUIN CHUNG, Argonne National Laboratory, United States
DONG JIN, Illinois Institute of Technology, and University of Arkansas, United States
RAJKUMAR KETTIMUTHU, Argonne National Laboratory, United States
MARTIN SUCHARA, Argonne National Laboratory, United States

Quantum network simulators offer the opportunity to cost-efficiently investigate potential avenues to building networks that scale with the number of users, communication distance, and application demands by simulating alternative hardware designs and control protocols. Several quantum network simulators have been recently developed with these goals in mind. However, as the size of the simulated networks increases, sequential execution becomes time consuming. Parallel execution presents a suitable method for scalable simulations of large-scale quantum networks, but the unique attributes of quantum information create some unexpected challenges. In this work we identify requirements for parallel simulation of quantum networks and develop the first parallel discrete event quantum network simulator by modifying the existing serial SeQUeNCe simulator. Our contributions include the design and development of a quantum state manager (QSM) that maintains shared quantum information distributed across multiple processes. We also optimize our parallel code by minimizing the overhead of the QSM and decreasing the amount of synchronization among processes. Using these techniques, we observe a speedup of 2 to 25 times when simulating a 1,024-node linear network with 2 to 128 processes. We also observe efficiency greater than 0.5 for up to 32 processes in a linear network topology of the same size and with the same workload. We repeat this evaluation with a randomized workload on a caveman network. Finally, we also introduce several methods for partitioning networks by mapping them to different parallel simulation processes. We released the parallel SeQUeNCe simulator as an open-source tool alongside the existing sequential version.

CCS Concepts: • Computing methodologies → Massively parallel algorithms; Discrete-event simulation; Massively parallel and high-performance simulations; • Networks → Network simulations.

Additional Key Words and Phrases: parallel discrete-event simulation, quantum network

1 INTRODUCTION

Over the last decade, the development of quantum networking testbeds has generated a lot of interest in the scientific community [9, 12, 14, 20, 28, 40]. However, the scale of these demonstrators is limited to a handful of nodes and communication distances that typically do not exceed metropolitan scale. To advance the state of the art, simulations of quantum networks can help researchers develop and test new control protocols, plan and validate experiments, and compare alternative network architecture designs. They provide a cost-effective way to complement experimental research that develops real-world testbeds and prototypes.

A quantum network simulator must offer realism (accurate timing and representation of quantum states), flexibility (support of new types of hardware and protocols), and scalability (large number of nodes and events) for experimenters to evaluate their design and applications. Existing quantum network simulators such as SeQUeNCe [49], Netsquid [8], and QuISP [29] achieve realism and flexibility by modeling quantum networks...
using sequential discrete-event simulation (DES). However, the sequential execution limits scalability since the simulation speed is limited by the performance of a single core. Simulating large-scale quantum networks through sequential execution takes a long time because quantum networks are more complex than their classical counterparts. The complexity of quantum network simulations comes from the need to model the behavior of several new types of hardware, and take into consideration processing and maintenance of quantum states, which typically generates a massive number of events. For example, the SeQUenCe quantum network simulator tracks millions of events per second of simulation time, and one example of a simple metropolitan network simulation generated approximately 2 billion events [49]. Nevertheless, high performance of a quantum network simulator is necessary to allow researchers to design quantum networks and uncover potential risks before deployment in hardware testbeds.

Parallel discrete-event simulation (PDES) [17] improves the scalability of DES and significantly reduces the execution time by utilizing multiple processors to simultaneously execute events that ultimately advance the simulation of experiments faster than sequential simulation does. A quantum network (that is being simulated) could be partitioned into multiple subnetworks, each handled by a separate processor. The PDES synchronizes the states of subnetworks across different processes and advances the global simulation timeline. This technique has been extensively applied in PDES for classical networks [31, 32].

The key to successfully parallelizing simulations of quantum networks is the efficient synchronization of events and states among processes. In this work we parallelize SeQUenCe using multiprocessing with an effective synchronization algorithm by considering specific quantum network characteristics. First, the Message Passing Interface (MPI) [43] is used to synchronize cross-process quantum channels and classical channels. Although multithreading can help speed up the execution on a single node, too many threads can lead to diminishing returns (e.g., when the number of threads exceeds the number of cores). MPI-based multiprocessing supports the execution of the simulation on a cluster of MPI nodes in both shared- and distributed-memory systems for scalability enhancement. Second, we design a hierarchical quantum state manager (QSM) to store and synchronize the quantum state of simulated qubits. The QSM consists of one local QSM per process and a global QSM for handling cross-process entangled qubits. Processes use TCP-based socket communication to access the quantum states on the global QSM. Third, we develop a conservative synchronization algorithm to orchestrate MPI processes and QSM for parallel simulation. The algorithm enables MPI processes to exchange cross-process events at a dynamically computed synchronization point.

The design of our parallel quantum network simulator carefully considers requirements for scalability, accuracy, and usability. The scalability of the parallel simulator is determined by the performance of state synchronization among processes. We improve the simulation scalability in two ways. First, we save the communication time between the local QSMs and the global QSM by reducing the number of packets. The global QSM offloads portion of the computational work to the local QSMs and batches the request processing. Second, we reduce the overhead caused by an unbalanced workload by reducing the number of synchronization events. In particular, we use high lookahead values (within the constraints of conservative synchronization) based on the quantum network characteristics.

The accuracy of a parallel simulator requires that the parallel execution of events does not affect the simulation results, in other words, that the parallel execution is identical to the sequential execution. We design the synchronization algorithm by considering the communication between the local QSMs and the global QSM. The optimized lookahead value guarantees that the parallel event execution does not affect the causality of events. Additionally, we modify the functions to access the random number generator used in the simulation. Without this modification, events in PDES may trigger different actions because of different random numbers. These modified functions ensure that the sequence of random numbers used in PDES is identical to that in the sequential simulation. As a result, the parallelized simulator can correctly reproduce sequential results.
To improve usability, the kernel of the parallel simulator enables systematic model parallelization and hides functions such as QSM and random number generation from the simulation design. Notably, users can develop and use the same simulation setup for both parallel and sequential simulation.

We perform extensive evaluation of the parallelized SeQUeNCe with three network topologies: linear topology, caveman graph [44], and Internet autonomous system [15]. We simulate quantum networks consisting of 1,024 quantum routers with 1 to 128 processes using up to 4 nodes on the Bebop supercomputer [26], a high-performance computing cluster located at Argonne National Laboratory. We use the speedup and execution time to evaluate the scalability of our simulator. The results show that PDES can speed up the networks with linear topology up to 25 times. We also analyze the sources of parallelization overhead. The overhead from the unbalanced workloads increases with the growing number of processes. Moreover, we study the performance improvement by reducing the QSM communication time (i.e., up to 11.5 times speedup) and enlarging the lookahead value (i.e., more than 50% synchronization time reduction).

We summarize the contributions of this work as follows.

1. We analyzed the requirements for parallel simulation and designed a PDES-based simulation framework of quantum networks based on SeQUeNCe that allow users to keep using the models and setup they already defined for the sequential version.
2. We developed a quantum state manager that stores and synchronizes the state of simulated qubits across multiple processes.
3. We conducted extensive performance evaluation of our parallel quantum network simulation with three network topologies on an HPC cluster, showing that our approach provides up to $25\times$ speed-up.

The remainder of the paper is organized as follows. In Section 2 we introduce the background on quantum information and quantum networks as well as the SeQUeNCe simulator and parallel simulation of classical networks. In Section 3 we analyze the requirements for parallel simulation of quantum networks and present the design architecture of our simulator. In Section 4 we describe the QSM design that maintains quantum states in parallel simulations. In Section 5 we discuss lookahead optimization for reducing the simulation overhead. In Section 6 we evaluate the performance of the parallel simulator. In Section 7 we describe related work. In Section 8 we conclude and briefly outline future work.

2 BACKGROUND

In this section we introduce a few elementary concepts in quantum information science. We also introduce the SeQUeNCe quantum network simulator and discuss the role of parallelization in classical network simulations.

2.1 Quantum Information and Quantum Network Protocols

Superposition allows a quantum system to be simultaneously in two different states. These states can be mathematically represented as a probability distribution of the possible measurement outcomes on the quantum system. For example, the measurement of a particle in the superposition state $|\psi\rangle := \alpha|0\rangle + \beta|1\rangle$ yields the measurement outcome $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$, where $\alpha$ and $\beta$ are two complex numbers. Upon measurement (on the z-axis), the quantum state $|\psi\rangle$ collapses to either $|0\rangle$ or $|1\rangle$. Optical quantum networks use photons to carry quantum states.

Multi-particle quantum systems can exhibit entanglement, where the quantum state of each particle in the group cannot be described independently of the others. The simplest example is the EPR pair named after Einstein, Podolsky and Rosen [13]: $|\phi^+\rangle := \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Here the measurement outcomes of the two particles are perfectly correlated with a 50% chance of being $|0\rangle$, $|0\rangle$ and 50% chance being $|1\rangle$, $|1\rangle$. The measurement of the first particle immediately collapses the state of the other particle to the same state, even if the particles are
spatially separated. We note that with increasing number of particles, the space required to represent quantum states grows exponentially, presenting a significant challenge for computer simulations.

![Quantum Circuit](image)

**Fig. 1.** Quantum circuit for quantum teleportation.

One can use entanglement to *teleport* an arbitrary quantum state over a long distance without worrying about state loss due to channel attenuation [4]. The teleportation protocol is depicted in Figure 1. Solid lines represent qubits, double lines classical bits, circles quantum states, and boxes quantum operations (quantum gates). Initially the sender and the receiver share an entangled qubit pair (green circles) that is distributed between them using the quantum network. The arbitrary quantum state \( |\psi\rangle \) we want to transmit is prepared in another qubit (blue circle). The sender performs the indicated elementary operations on its two qubits (CNOT and Hadamard gates) and measures them. One of four possible measurement outcomes can be realized: 00, 01, 10, or 11. The measurement outcome is then sent to the receiver using a classical channel, and the receiver uses it to perform controlled quantum operations on its qubit in a process called Pauli frame correction. Upon the correction, it can be shown that the receiver’s qubit (red circle) will be in state \( |\psi\rangle \). Teleportation is one example of a quantum network protocol and it allows quantum state transmission by only distributing EPR pairs and classical information.

![Long-Distance Entanglement](image)

**Fig. 2.** Two short-distance entangled pairs are used to create a long-distance entanglement with the swapping protocol.

Reliable, long-distance entanglement distribution is an important service provided by quantum networks and many other protocols depend on it, including teleportation. The success probability of entanglement distribution decreases exponentially with distance. Fortunately, the *entanglement swapping* protocol [51] can be used to "stitch together" multiple shorter-distance entanglements. The swapping protocol is illustrated in Figure 2. In the first step, entanglement is generated between the adjacent nodes A-B and B-C. Entangled photons may be stored...
in quantum memories to ensure that they are available precisely when needed [23]. To create entanglement between nodes A-C, node B performs Bell state measurement (BSM) on its two qubits. The measurement outcome is sent from node B to either node A or node C to perform Pauli frame correction. After the correction the qubits on nodes A and C become entangled in the EPR state. This article simulates entanglement distribution and entanglement swapping in networks with up to 1,024 nodes, a computationally challenging problem.

2.2 SeQUeNCe: A Customizable Quantum Network Simulator

SeQUeNCe is an open-source quantum network simulator [49] that allows simulations with photon-level details. The SeQUeNCe project website [36] provides the complete source code, documentation and tutorials that allow easy usability and extendability. Among the many use cases of the simulator are understanding of the tradeoffs of alternative quantum network architectures, optimizing quantum hardware, and developing a robust control plane. SeQUeNCe uses a modularized design, as shown in Figure 3. This enables researchers to flexibly customize the simulated network and reuse existing models. The Application module represents quantum network applications and their service requests. The Resource Management and Network Management modules use classical control messages to manage allocation of network resources. The Entanglement Management and Hardware modules encompass protocols and hardware operations that act on qubits that are located in quantum memories or in photons at telecommunication wavelengths in optical network links. The SeQUeNCe simulator stores these states in the Quantum State Manager (QSM). For sequential simulation, a local QSM is sufficient to trace the quantum states. However, parallel simulation requires the simulator to synchronize multiple QSMs, a requirement that is addressed in this work. This work also significantly redesigned the Simulation Kernel to allow parallel simulations. Further details about the design and use cases of the sequential version of the SeQUeNCe simulator can be found in our earlier work [47–49].

![Fig. 3. Modularized design of SeQUeNCe.](image)

2.3 Parallel Simulation of Classical Networks

Researchers have built time-parallel and space-parallel network simulators to improve scalability in terms of both execution speed and network size. Time-parallel simulation methods [1, 24, 46] parallelize a fixed-sized problem by partitioning the simulation time axis into intervals and assigning a processor to simulate the system over its assigned time interval. Parallel (and distributed) discrete-event methods use a space-parallel approach that divides a single simulation topology into distinct pieces and executes them in parallel on multiple interconnected processors. These efforts started about 15 years ago with Parallel/Distributed ns [37] and the Georgia Tech Network Simulator [38]. Distributed simulation support was added to ns-3 [33] in 2011 [35].
Ensuring that events are processed in time-stamped order is a fundamental part of distributed simulation—the so-called causality constraint; see Figure 4. Almost all parallel network simulators use a conservative synchronization algorithm that takes precautions to avoid violating the causality constraint. The synchronization overhead significantly limits the speedup that can be achieved with parallel and distributed simulations. In fact, synchronization overheads in conservative approaches can cause parallel execution to be slower than sequential execution [25]. In contrast, optimistic synchronization algorithms allow violations to occur but are able to detect and recover from them. Optimistic synchronization has been applied successfully in other contexts such as high-performance computing (HPC), storage, and HPC interconnects [6, 7] but has not been used to simulate wide area networks and TCP/IP-based networks (to the best of our knowledge).

3 PARALLEL QUANTUM NETWORK SIMULATOR DESIGN

The parallel simulation splits the simulated quantum network into multiple non-overlapping subnetworks. Each subnetwork consists of a subset of the network nodes, including components of the node and associated qubits,
that are executed by one processor. The design of the parallel quantum network simulator should satisfy three requirements.

(1) **Accuracy**: The results of the parallel simulation should faithfully reproduce results from the sequential simulation. Parallelization should not sacrifice realism.

(2) **Scalability**: The parallelized simulator should have good scalability. The design must carefully address the parallelization overhead, otherwise the parallel simulator performance could be suboptimal and in the extreme case even worse than that of its sequential counterpart.

(3) **Usability**: The parallel simulator should be compatible with models used for sequential simulation. This requirement aims to reduce errors as well as allow switching between parallel and sequential simulation, thereby increasing the usability and flexibility of the simulator.

In this work we extend SeQUeNCe [49] to a parallel simulator and demonstrate how the design achieves all three requirements.

### 3.1 Requirement Analysis

The **accuracy** of the simulation depends on the correct order of event execution and the accurate modeling of network state. During a parallel simulation, a large network is partitioned into multiple non-overlapping subnetworks. Events in a subnetwork are executed by one assigned processor. Event execution may affect the order of events and the state of the network on the other processors. The accuracy of the simulation requires the simulator to synchronize these effects among processors. We summarize three methods that can affect the state of nodes on the other subnetworks.

1. One node can send qubits to another node over a quantum channel. This action generates an event on the receiver node to receive qubits. This is typically used for entanglement generation between adjacent quantum routers.

2. One node can send classical messages to another node on a classical channel. This action creates an event to receive messages. Control protocols on nodes process these classical messages.

3. Entangled qubits that belong to different processors can change the shared entanglement state. For example, in quantum teleportation, the measurement on the sender node changes the quantum state of the qubit on the receiver node without any explicit quantum or classical message transmission.

The first two methods affect states on other processors by generating events in the future. Processors can synchronize their states by exchanging cross-process events, and the synchronization algorithm can successfully avoid out-of-order event execution. The third method affects states on other processors by changing a shared state. To ensure consistency of shared states, we make entangled qubits share a single entanglement state instance instead of storing multiple copies for every qubit. A read/write lock is applied to avoid conflict when accessing those shared states.

Random number generation is another factor affecting reproducibility. Given the randomness in measuring qubits, the parallel simulator should generate random numbers identical to those of the sequential simulation. Therefore, we assign a random number generator (RNG) to every network node both in the sequential and parallel simulation. Events executed on a node or its components use the RNG on the node to generate a random number. Since components connected to a single network node are never split between processes, identical results will be produced as long as events on the node are executed in the correct sequential order.

The **scalability** of the simulator depends on the reduction of parallelization overhead. In a parallel quantum network simulator, the overhead results from synchronizing processors, exchanging data, and accessing the shared quantum states. Our design aims to reduce the execution time and complexity of these functions in order to improve the simulation scalability.
The usability of the parallel simulator depends on its ease of use. Models developed for parallel simulation ought to be compatible with the sequential simulation (one can configure the parallel simulator to run experiments with one processor). Therefore, our design should allow efficient and error-free simulation design; in other words, the parallelization should be systematically handled by the simulation kernel, with the process completely transparent to users.

3.2 Parallel Simulator Architecture

Figure 5 depicts the architecture of our parallel simulator. We parallelize SeQUeNCe by multiprocessing, where a separate processes handles each subnetwork. We use MPI [43] for interprocess communication including cross-process events, produced by cross-process quantum and classical channels, for both shared- and distributed-memory systems. These two techniques allow users to run a parallel simulation on an MPI cluster. Although multiprocessing may incur a higher communication cost than multithreading does, high-performance computing cluster optimizations have reduced the MPI communication overhead [10]. Our experiments in Section 6 also demonstrate that MPI communication has a limited impact on the parallelization overhead.

To track simulated quantum states, we design a quantum state manager, which provides an interface for manipulating and reading quantum states. For shared entanglement, we also design a global QSM as a separate server. During the simulation, all cross-process entanglement states are stored in the global QSM, while non-entangled qubits or those entangled with only local qubits are stored in a local QSM. Entangled qubits hold the key for accessing their shared state only within a local or global QSM. An individual process may then communicate with the global QSM to retrieve or update the state of an entangled qubit, either directly or through a quantum circuit. Quantum circuit operations, represented as unitary matrices, are not generally commutative. These operations therefore cannot be reordered, implying a requirement for proper ordering with respect to the simulation time for global QSM requests. Since quantum circuits may operate only on qubits stored in local hardware, however, circuits on separate processes will not overlap. Thus, the execution order of requests does not affect final results. The global QSM may be implemented as a single-process program, a multithreaded program, or a cluster according to the scale of the parallel simulation. Thus, the global QSM can handle requests from different processes more efficiently and reduce potential resource competition. While interprocess events are exchanged using MPI, the simpler point-to-point communication between the local QSMs and the global QSM utilizes TCP/IP communication. This is to avoid the additional overhead from using MPI while maintaining QSM communication. Further QSM implementation details are described in Section 4.
To ensure proper execution of events, we adjust the conservative synchronization algorithm [30] not only to exchange cross-process events but also to synchronize states on the global QSM. The existing conservative algorithm has been demonstrated to work for networks without entangled states. Figure 6 depicts the operation of the conservative algorithm. Events on separate processes execute in parallel during the \textit{Computing} phase. If an event execution schedules a future event to be run on a separate process, the future event is stored in the output queue \textit{OutQ}. When a process executes all local events up to \textit{sync\_time}, it enters the \textit{Waiting} phase until all processes are finished. The processes then exchange events in the \textit{OutQ} during the \textit{Communicating} phase. Additionally, a future \textit{sync\_time} is calculated by using a lookahead value.

We describe our implementation of the conservative algorithm with the addition of a global QSM in Algorithm 1. In this algorithm, we introduce the following data structures:

- \textit{Queue}: a queue storing events to be executed locally
- \textit{OutQ}: a queue of events that are scheduled locally but should be executed on a different process
- \textit{InQ}: a queue of events that are scheduled by a different process but should be executed locally

At each iteration of the algorithm loop, we check to see whether \textit{local\_time} (the simulation time of event execution on the process) is less than \textit{end\_time} (the simulation time at which event execution should halt). If the condition is met, the algorithm moves into the \textit{Communicating} phase; otherwise, the loop exits because the simulation has finished.

In the \textit{Communicating} phase, MPI processes first calculate the minimum timestamp of events stored locally (\textit{Local\_min\_time}) by comparing the minimum timestamp of the events in the queue (\textit{Queue}) and output queue (\textit{OutQ}). Next, MPI processes move events in the local output queue to the corresponding process and broadcast their \textit{local\_min\_time}. The function ”MPI\_EXCHANGE\_DATA” returns events from other processes to the input queue (\textit{InQ}) and a list of \textit{local\_min\_times} from other processes in the \textit{remote\_min\_times} list. By calculating the minimum value of the \textit{remote\_min\_times} list and \textit{local\_min\_time}, the algorithm determines \textit{global\_min\_time}, which is the minimum timestamp of all events in the simulation. If all processes have an empty event queue,

![Diagram](image)

(a) Parallel simulation from the perspective of simulation time. Processes independently execute events with a timestamp lower than the \textit{sync\_time}. If the process generates events belonging to another process, these events are stored in the \textit{OutQ} of the process until all processes reach the \textit{sync\_time}. The synchronization algorithm ensures that all events in each \textit{OutQ} have a timestamp that is larger than the \textit{sync\_time}.

(b) Parallel simulation from the perspective of wall-clock time. During the \textit{Computing} phase, processes independently execute events until the barrier required for the collective communication using MPI. During the \textit{Communicating} phase, processes exchange events and decide the next \textit{sync\_time} using MPI communication. During the \textit{Waiting} phase, processes wait for the slowest process to reach the synchronization barrier.

Fig. 6. Conservative synchronization algorithm.
Algorithm 1: Conservative Synchronization

\begin{verbatim}
while local_time < end_time do
  // Communicating phase
  local_min_time ← min({Queue.min_time(), OutQ.min_time()})
  InQ, remote_min_times ← MPI_EXCHANGE_DATA(OutQ, Local_min_time)
  global_min_time ← min({Local_min_time} ∪ remote_min_times)
  sync_time ← min({global_min_time + lookahead, end_time})
  push_events_to_queue(Queue, InQ)
  if global_min_time is ∞ then
    break
  end
  // Computing phase
  while Queue.size() > 0 and Queue.top().time < sync_time do
    event ← Queue.pop()
    local_time ← event.time
    execute(event)
  end
  local_time ← sync_time
  notify_server_about_synchronization()
end
\end{verbatim}

\textit{global_min_time} will be infinity, and execution is halted. Otherwise, we can use \textit{global_min_time} and the \textit{lookahead} value to get the time window [30] bounded by a future \textit{sync_time} (in simulation time). If the time window extends beyond \textit{end_time}, we instead set \textit{sync_time} to be \textit{end_time}.

As shown in Figure 6a, the \textit{lookahead} value is set so that all cross-process events may be scheduled for simulations times only after \textit{sync_time}. We subsequently move into the \textit{Computing} phase, where events in the local Queue are executed and \textit{local_time} is updated accordingly. After all events before \textit{sync_time} are executed, we update \textit{local_time} to be \textit{sync_time}. Additionally, the process sends a message to the global QSM to serve all pending requests. The process is unblocked upon receiving a response from the global QSM to ensure that all the early requests on the global QSM are completely handled.

The parallel simulation kernel also supports sequential models for compatibility when needed. As shown in Figure 5, models can schedule events and manipulate quantum states using interfaces offered by the kernel. When the simulation is executed sequentially, the event scheduler does not need to communicate with other processes, and the local QSM does not need to connect with the global QSM. Therefore, the same model can be executed using sequential or parallel simulation with only a simple setup configuration change.

3.3 Parallelization Overhead Sources

The proposed architecture introduces parallelization overhead. This overhead may be measured as an increase in execution time beyond that required for execution of simulated events. Figure 6b shows the state of processes using the conservative synchronization algorithm. Processes have three states: \textit{Computing}, \textit{Communicating}, and \textit{Waiting}. The \textit{Computing} state denotes that the process is executing events whose timestamps are within the time window. The \textit{Communicating} state denotes the MPI communication that creates a synchronization barrier among processes. The \textit{Waiting} state denotes that a process has completed the local event execution and is now blocked by the communication barrier. We identify four sources of overhead. First, the barrier caused by the MPI
communication makes a process (e.g., light workload) wait for other processes (e.g., heavy workloads) in the Waiting state. The second source is due to the data exchange among processes. The number of exchanged events affects the communication overhead. The third source is the cost for barrier synchronization (e.g., the calculation of the local minimum timestamp for the next computing phase and the time window until the next synchronization). Apart from the overhead for the synchronization algorithm, the global QSM introduces the fourth overhead source. In the parallel simulation, the global QSM handles operations on a cross-process entanglement state. As a result, the communication spent on the global QSM increases the time of manipulating entanglement states. A parallel simulation with a larger amount of cross-process entanglement may thus introduce larger parallelization overhead due to the growing global QSM communication. In Section 4 we introduce multiple techniques for reducing the overhead from these sources.

4 QUANTUM STATE MANAGER

This section describes the quantum state manager for our PDES-based simulator of quantum networks. Section 4.1 describes the essential functions provided by the QSM in the simulation kernel and the memoization technique used for accelerating matrix multiplication. Section 4.2 describes the global QSM, in which we explain the mechanism used for distinguishing cross-process entanglement and single-process entanglement. Additionally, we illustrate the local computing, traffic compression, and multithreading techniques that are used to reduce the overhead of communication between the local and global QSMs.

4.1 Local Quantum State Manager

The QSM within the simulation kernel is designed to provide methods for manipulating quantum states. Models of hardware and protocols may manipulate the quantum state of qubits with a qubit key that is produced by a universal unique identifier [27]. The QSM maintains the relation between keys and quantum states as a map data structure. For entangled states, multiple keys may point to the same quantum state object. Here we introduce three important functions provided by the QSM:

- set(keys, state): The QSM sets the quantum state to a given state for qubit(s) with a given key(s).
- get(key): The QSM returns the state of the qubit with a given key.
- run(circuit, keys, prob_sample): The QSM runs a quantum circuit on the qubit(s) with a given key(s). The circuit measures qubits, prob_sample, a random number in the range from 0 to 1 for deciding the collapsed quantum state. The function returns the outcome of the measurement.

Quantum states (including entanglement) are stored as vectors (Bra-ket notation [39]) or matrices (density matrix [16]) together with a list that indicates the order of entangled qubits as a list of keys. The get and set functions allow models to read and write the quantum state of qubits. Compared with the set function, hardware and protocol models mostly use the run function to change the quantum state of qubits. For example, entanglement purification [11] and swapping [51] protocols usually describe their protocols as a quantum circuit. A quantum circuit consists of quantum logic gates and/or measurement. The gates in the circuit may be represented by a unitary matrix. Unlike quantum logic gates, measurement is a stochastic nonreversible operation. The simulator thus needs to use a random number to determine the measured state. To reproduce identical outcomes, the run function takes the random number prob_sample from the caller function if the circuit contains a measurement operation. As described in Section 3.1, this random prob_sample is produced by the random number generator on the calling node to ensure consistency of simulations.

In SeQUeNCe, we use the QuTiP [22] Python library to generate the unitary matrix of a given circuit. The output state is produced by matrix multiplication of the unitary matrix of the circuit gates with the input quantum state of the qubit(s), which is a CPU-intensive task. Furthermore, the simulation of a quantum network involves repeated computation of this matrix multiplication. Therefore, we utilize the memoization technique [41] to
(a) All qubits belong to one MPI process. The local QSM of the process updates the quantum state of qubits locally.

(b) Qubits belong to different MPI processes; entities in the left (right) part belong to process 1 (process 2); the photon $q_a$ moves from process 1 to process 2. When process 1 detects that the destination of $q_a$ is on process 2, the local state is forwarded to the global QSM in 1. The optical hardware models on process 2 use the local QSM to generate entanglement in 2. The local QSM detects that a required qubit is stored on a separate process and forwards to the global QSM in 3. The global QSM then produces the required entanglement state and returns results to the local QSM on process 2.

Fig. 7. Entanglement generation with the meet-in-the-middle protocol under two scenarios; quantum memories $q_A$ and $q_B$ emit photons $q_a$ and $q_b$, respectively, to generate entanglement. $S_A$ ($S_B$) denotes the quantum state of qubits $q_A$ ($q_B$). The optical fiber transmits photons to the BSM node located in the middle. Once detectors are triggered by photons, the BSM node sends messages to nodes containing memories. Protocols may determine the current quantum state of memories with these messages.

speed up redundant functions, such as the run function. We implement least recently used caches with fixed sizes to cache the result of these multiplications.

4.2 Global Quantum State Manager

While unentangled states or entangled states confined to one process may be managed by a single local QSM, cross-process entangled states may need to be accessed by multiple local QSMs. We thus designed a global QSM as a server to ensure the consistency and accuracy of quantum states within the parallel simulation. While the quantum state of qubits is stored in the centralized global QSM, the local QSMs on separate MPI processes can manipulate a shared state without conflict by using identical functions.

Since the global QSM is executed on an independent process, it must receive requests and send responses over network communication in contrast to a local QSM. Multiple MPI processes may communicate with the global QSM simultaneously, which can result in resource competition. Such competition may increase the processing time on the global QSM when performing large-scale parallel simulations. The communication time and resource competition constitute the parallelization overhead from the global QSM as described in Section 3.3. The global QSM may thus become a bottleneck for parallel simulation. To address this, we implement multiple techniques to efficiently reduce the global QSM overhead.

4.2.1 Mitigating Communication Overhead. The total communication time depends on the number of transmissions and the latency of each transmission. In this work we reduced the number of transmissions between the local and global QSMs. These transmissions include three types of communication categorized by their goals:
Parallel Simulation of Quantum Networks with Distributed Quantum State Management

- **Forward function call.** The local QSM forwards function calls to the global QSM and waits for a response.
- **Synchronize quantum state.** Operations on the local QSM may cause a state inconsistency, which requires the local QSM to update the state in the global QSM.
- **Acknowledge state of global QSM.** This is for a local QSM to confirm that the global QSM has completed executing all the early forwarded functions.

To reduce the number of transmissions, we first offload most of the computational work to the local QSM. We use the entanglement generation meet-in-the-middle protocol [23] as an example to show interactions between the local and global QSM in Figure 7. Qubits $q_A$ and $q_B$ stored in each quantum memory are entangled with photons $q_a$ and $q_b$ with the shared quantum state $S_A(S_B)$. Photons are sent to the intermediate BSM node to generate entanglement between $q_A$ and $q_B$. In the parallel simulation, qubits either belong to one process, shown in Figure 7a, or belong to two processes, shown in Figure 7b. If the MPI process contains all required qubits, the local QSM can produce the entangled state between $q_A$ and $q_B$ locally without forwarding the function to the global QSM. Otherwise, the local QSM must forward the function to the global QSM since the “local QSM 2” executing events on the BSM node does not hold the state $S_A$. As a result, a simulation with fewer cross-process entanglement states has fewer communication requests with the global QSM and thus reduces the communication overhead.

The challenge of offloading work to the local QSMs is to ensure the consistency of quantum states between the local and global QSMs. For example, in Figure 7b, the quantum state $S_A$ in the global QSM should be identical to the state $S_A$ stored in “local QSM 1” before the global QSM executes the request from “local QSM 2.” The timing of synchronizing local states with the global QSM thus becomes an issue, since the parallel simulator must keep this consistency while avoiding redundant synchronization.

Investigation of existing entanglement protocols shows that the synchronization communication is triggered by two conditions: (1) transmitted qubits sharing an entangled state and (2) forwarded function calls requiring a local quantum state. The first condition may separate two entangled qubits that belong to one process, so we synchronize the entanglement state to the global QSM. The management of this state is moved from the local QSM to the global QSM. To the best of our knowledge, all entanglement generation protocols rely on quantum channels to entangle qubits in two locations. The synchronization triggered by the first condition can avoid inconsistent states between the local and global QSMs. The second condition requires the global QSM to execute operations. The local QSM should push the local quantum state to the global QSM before forwarding the function call. The correctness of entanglement protocols, such as entanglement swapping, can be protected by these triggering conditions.

The synchronization requirement demands the management of qubits at the global QSM. The local QSM forwards the function calls related to either unknown qubits (i.e., qubits produced by other processes) or qubits managed by the global QSM. Once the qubits lose their entangled state, the management of qubits will be transferred back to the local QSM. In the parallel SeQUeNCe simulator, either the measurement on qubits or the set function can return the qubits back to the local QSM by destroying an entangled state.

In addition to offloading work from the global QSM, we reduce the communication time by grouping multiple requests into one packet. We observe that some requests to the global QSM do not expect responses, such as the run function without measurement and the set function. Therefore, the simulator caches requests with no responses. These requests are transmitted to the global QSM only if a new request needs results from the global QSM or the simulator forcefully transmits all requests in the cache. This mechanism leads to efficient state synchronization since the synchronization does not require a response from the global QSM.

We implement the communication between the local and global QSM using the TCP/IP protocol. Functional calls are serialized in JSON format. We note that the communication latency highly depends on the performance
of the MPI cluster. Thus, we execute the global QSM process on the same cluster to reduce the communication latency.

4.2.2 Mitigating Resource Competition. The resource competition of MPI processes reduces the performance of the global QSM. Since the global QSM needs to handle requests from all MPI processes, the performance of the global QSM will degrade with a growing number of processes and communication, and thus the global QSM may become the parallelization bottleneck.

To address this problem, we first design the global QSM as a multithreaded program. The global QSM uses independent threads to concurrently serve MPI processes. Compared with a single-process program, the multithreading technique reduces the waiting time for executing the early arrival requests. Compared with a multiprocessing program, the multithreading technique provides a lighter way to share resources among threads.

Using the multithreading technique requires thread safety for the shared quantum states in the global QSM. In the parallel simulation of a quantum network, multiple MPI processes may simultaneously operate on the same quantum state. For example, step 2 of Figure 2 shows that nodes A and C need to run a circuit to correct the quantum state of entanglement between A and C. If nodes A and C belong to two MPI processes, the two threads serving these MPI processes may simultaneously process requests that modify the entanglement state shared between A and C, which may cause a race condition. We use a set of locks to guarantee thread safety. The global QSM assigns each qubit a lock. When the global QSM receives a function call, the global QSM collects all qubit keys related to the function. These qubits include not only qubits listed in the argument of the function but also qubits entangled with listed qubits, since all these qubits share a single entangled state. Then, the thread acquires locks for these qubits in order to avoid deadlocks. After the execution of the function, the thread releases all locks.

In contrast to the rest of the SeQUeNCe source code, we implement the global QSM in C++. Since simulation models interact only with the local QSM, the global QSM is hidden from users and their models. Implementing the global QSM in C++ rather than Python results in higher performance while still maintaining the ease of use for the simulator. The C++ library Quantum++ [18] is used to manipulate quantum states on the global QSM. As an additional benefit, Quantum++ claims better performance than QuTiP [22]. The memoization mentioned in Section 4.1 is also implemented to reduce redundant computational work.

5 LOOKAHEAD OPTIMIZATION

The lookahead value determines the frequency of synchronization among processes. Parallel simulations with larger lookahead values synchronize the processes less often and thus exhibit better performance. Additionally, the overhead from the unbalanced workload can be mitigated by a larger lookahead value. In order to support PDES of a quantum network, the minimum latency of cross-process channels can be selected as a baseline lookahead value. Although both quantum and classical channels introduce a propagation delay, the latency on the classical channels also includes processing delay, transmission delay, and queuing delay, which make the latency on a classical channel \(T_{cc}\) generally larger than the latency on a quantum channel \(T_{qc}\). Therefore, the lookahead value usually equals the latency of the shortest quantum channel.

However, we propose that the lookahead value \(\Delta t\) could be increased to \(\frac{T_{cc}}{2}\), where \(T_{cc}\) is the classical delay between routers and intermediate BSM nodes if the simulated quantum network uses the meet-in-the-middle protocol to generate entanglement. Users could choose \(\frac{T_{cc}}{2}\) as the lookahead value if \(\frac{T_{cc}}{2} > T_{qc}\). In a general entanglement distribution network, only the entanglement generation protocol relies on the quantum channel; the remaining protocols manipulate entanglement using quantum circuit operations on a single node. When the longer latency is chosen, the simulator can guarantee the accuracy of simulating a network only with the meet-in-the-middle configuration. Figure 7 depicts the process of the meet-in-the-middle entanglement generation protocol. One BSM node is placed in the middle of two quantum routers. Routers first send qubit-encoded photons to the BSM node and then receive messages from the BSM node. In a parallel simulation, if two quantum routers
Quantum routers synchronize their time by Algorithm 1, and the BSM node exchanges cross-process events only at the time window. We label the order of execution on the figure as follows:

1. Quantum routers execute events from $t_0$ to $t_0 + \Delta t$ and exchange events. Cross-process events are moved to the BSM node.
2. The BSM node executes events from $t_0$ to $t_0 + \Delta t$, while routers execute events from $t_0 + \Delta t$ to $t_0 + 2*\Delta t$. The BSM node sends produced cross-process events to the routers and receives cross-process events from $t_0 + \Delta t$ to $t_0 + 2*\Delta t$.
3. The BSM node executes events from $t_0 + \Delta t$ to $t_0 + 2*\Delta t$, and the routers execute events from $t_0 + 2*\Delta t$ to $t_0 + 3*\Delta t$. Then, the nodes exchange data as described in 2.

The BSM node receives events at the barrier $t_0 + \Delta t$ and executes events from simulation time $t_0$ to $t_0 + \Delta t$. Since the time of the BSM node is slower than the time on the routers by at most $2\Delta t$, cross-process events produced by the BSM node must have a timestamp larger than $t_0 + 2*\Delta t$. The second synchronization barrier at $t_0 + 2*\Delta t$ can move cross-process events produced by the BSM node to the routers. In this way, all cross-process events are exchanged and executed in the correct order, and PDES can use the $\Delta T = \frac{T_{cc}}{2}$ lookahead time to reduce the parallelization overhead if $\frac{\Delta t}{2} > T_{qc}$.

To apply this mechanism, the MPI process picks out BSM nodes that connect with routers on other processes before running the simulation. The MPI process maintains two simulation times and event queues for the selected BSM nodes and other nodes, respectively. The simulation times for the selected BSM nodes are allowed to be slower than for the others. MPI processes then execute events as shown in Algorithm 1 with the mechanism described previously to run the parallel simulation. We evaluate the improvement from this optimization in Section 6.
6 PERFORMANCE EVALUATION

We conduct extensive evaluation experiments for the parallelized SeQUeNCe on three quantum network topologies. We report four components of the wall-clock execution time of the simulations: MPI communication, socket communication, event execution, and barrier synchronization. The MPI communication includes the overhead from the data exchange and the unbalanced workload. The socket communication denotes the aggregated pending time of a process caused by the communication between the local and global QSMs. The execution time speedup (strong scaling) and efficiency (weak scaling) \( T_s/T_p \), are used as the metrics to quantify the scalability of the simulator, where \( T_s \) and \( T_p \) denote the execution time used for sequential simulation and parallel simulation consisting of \( p \) processes, respectively. The barrier synchronization algorithm overhead is very low (less than 1 second), and thus we do not plot it in the following figures.

The simulations of the three quantum networks utilize identical models of software and hardware. Simulations execute up to 30 million events by using 1-128 processors. Considering the scale of the simulated network, we limit the maximum number of processors to 128, but our parallel simulator supports parallel simulations using more processors. In Table 1 we list values of parameters shared among simulations; the detailed model design can be found in our prior work [49]. For classical communication, the nodes are connected with the channels in a fully connected topology.

The three simulated quantum networks have different network topologies and flows. In Section 6.1 the first quantum network uses a linear topology to distribute entanglement between the end nodes. Such a chain of quantum repeaters is useful for long-distance distribution of entanglement. The network uses only one traffic flow to distribute entanglement between the two routers on the endpoints.

In Section 6.2 the second quantum network uses the topology of a connected caveman graph [44] that consists of 128 connected cliques, each having 8 quantum routers. This topology is more complicated than the linear topology. We simulate multiple random flows in the network. The random flows are produced by the network nodes, and each node selects a random destination.

The third quantum network uses the topology of an Internet autonomous system network [15] generated by Networkx [19]. Similar to the second network, every network node produces a flow with a random destination. The challenge of parallel simulation for this kind of network results from its irregular topology. We introduce three heuristic methods to optimize the network partitioning in Section 6.3.

All the simulations were executed on the Broadwell partition of Bebop [26], the high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. Each of the Broadwell nodes has 36 cores of Intel Xeon E5-2695v4 and 128 GB of DDR4 running CentOS 7. All programs are interpreted using Python 3.8.5 or compiled using g++ 4.8.5.

Bebop nodes use an Intel Omni-Path interconnect for communication, with a local MPI implementation. The other libraries used for simulation were NumPy (version 1.18.5), pandas (version 0.24.2), QuTiP (version 4.6.0.dev) [22], tqdm (version 4.47.0), NetworkX (version 2.4), mpi4py (version 3.0.3), and Quantum++ (version 2.6) [18]. Python packages were managed using Anaconda version 4.8.3, and the C++ QSM server was built using CMake version 3.18.4.

6.1 Chain of Quantum Routers

The first kind of network has a quantum router chain that serves only one flow that distributes entanglement between the two end routers. The quantum network uses a chain of routers to mitigate the exponentially increased loss rate of photons caused by the increased distance [5]. Figure 9a illustrates the topology of a quantum network with five routers. In this type of network, adjacent nodes first use the meet-in-the-middle generation scheme [23] to distribute entanglement over a short distance. Then, the entanglement swapping [51] and purification protocols [11] are used to distribute entanglement over a longer distance with high fidelity.
Table 1. Parameters of hardware models and their default values.

| Parameter                                           | Value       |
|-----------------------------------------------------|-------------|
| Memory efficiency                                   | \( \eta_m = 0.75 \) |
| Memory frequency                                    | \( f_m = 20 \text{ kHz} \) |
| Memory coherence time                               | \( t_c = 1.3 \text{ s} \) |
| Fidelity of entanglement from generation protocol   | \( F_0 = 0.99 \) |
| Detector efficiency                                 | \( \eta_d = 0.9 \) |
| Detector count rate                                 | \( r = 25 \text{ MHz} \) |
| Detector dark count                                 | \( d \approx 0 \text{ s}^{-1} \) |
| Detector resolution                                 | \( s = 150 \text{ ps} \) |
| Attenuation                                         | \( \alpha_0 = 0.2 \text{ dB/km} \) |
| Speed of light                                      | \( c^* = 2 \times 10^8 \text{ m/s} \) |
| Channel TDM time frame                              | \( t_f = 12.5 \text{ ns} \) |
| Gate fidelity                                        | \( F_{\text{gate}} = 1 \) |
| Swap success probability                            | \( p_{\text{swap}} = 1 \) |
| Quantum channel length                              | \( L_{\text{qc}} = 1 \text{ km} \) |
| Classical channel latency                           | \( T_{\text{cc}} = 0.3 \text{ ms} \) |
| Simulation time                                     | 100 ms |

For the parallel simulation experiments, we evenly divide the number of routers by the number of processes \((n = 1024/p)\) and assign them sequentially to each process (i.e., routers 1 to \(n\) will be assigned process 1, routers \(n+1\) to \(2n\) to process 2, etc.). For simulations with the increased lookahead value introduced in Section 5, the BSM nodes that connect two different processes are placed in a separate process where the simulation time is allowed to lag.

All our linear topology simulations use a 1 km distance between all adjacent routers. The BSM node is in the middle of each link; in other words, the length of optical fiber is 0.5 km. All routers contain 100 quantum memories.

We first evaluate the strong scaling of the simulator in a quantum network with 1,024 routers and a total of 102,400 quantum memories. Strong scaling requires simulations with a fixed workload and an increasing number of processes. We simulate the network with 1 to 128 processes. For this and future simulations using 1 process, the sequential version of SeQUeNCe was used (rather than the parallel version running with 1 MPI process). Since one computing node contains only 36 cores, parallel simulations with 64 and 128 processes are executed on 2 and 4 computing nodes, respectively. Figure 9b illustrates the average execution time of each simulation process for 10 separate simulations and the speedup ratio achieved by the parallel simulation. Since the standard deviation in measured execution time ranges from 7.4 to 189.6 seconds for strong scaling (with 32 and 1 processes, respectively) and from 0.8 to 13.48 seconds for weak scaling (with 16 and 128 processes, respectively), we do not include error bars in the plot. We observe that the execution time decreases with the increasing number of processes. The parallel simulation can speed up the simulation by a factor up to 25. When we double the number of processors, the execution time is halved (indicated by the blue part). However, the synchronization time occupies a larger portion of the total time with the growing number of processes. We also notice that the time for the socket communication between the local and global QSMs is trivial in this case. The time of socket communication is increased from 2.0 seconds to 18.6 seconds as the number of processes increases from 2 to 128.

We also evaluate the performance of the simulator with weak scaling using a scaled workload. We fix the number of routers on one process to 8 and increase the size of the network with the number of processes.
Figure 9c illustrates the execution time and efficiency of the simulations. We observe that the execution time increases with the number of processes. The majority of overhead results from the synchronization algorithm; little overhead is generated from the socket communication. Local QSMs process more than 90% of requests without socket communication. Note that we scale the workload by increasing the size of the network; therefore, the workload does not exactly increase linearly and thus causes the increased time of event execution. In addition, the parallelization efficiency decreases with the number of processes resulting from the increased overhead. The efficiency of the 2-process simulation is 0.96, whereas the efficiency of the 128-process simulation is 0.19.

Both strong scaling and weak scaling evaluations show an increasing overhead. We observe that the performance in both cases drops significantly when the number of processes increases from 32 to 64. There are two reasons for this drop. First, the performance of the multithreaded global QSM drops when the number of processes is
larger than 32. Since we execute the global QSM on one computer that serves all MPI processes, the capability of the global QSM is limited by the number of processors and affected by the MPI processes on the same machine. When the number of processes is less than or equal to 36, the number of processors on one Broadwell node, the global QSM threads do not compete with each other. The block of socket communication on the MPI processes avoids competition between threads and MPI processes. However, when the number of processes is greater than 36, threads and MPI processes compete for computation resources and thus show worse performance. We observe that the global QSM achieves a similar processing speed for parallel simulation with up to 32 processes. The processing time of the global QSM is doubled for simulations with 64 processes and tripled for 128 processes. The degraded performance is also demonstrated with the growing socket communication time as the number of processes increases from 32 to 128 (see Figure 9c).

The second overhead is due to the unbalanced workload. The red bars in Figure 9 indicate the synchronization overhead including the unbalanced workload and the transmission of data. Therefore, we design further experiments to distinguish and quantify which one actually results in the increased synchronization time. The experiments duplicate the cross-process data before exchanging events. When the processes receive data from other processes, the duplicated data is discarded. These experiments increase the amount of cross-process transmission during synchronization without affecting the distribution of the workload. Figure 9d shows the execution time of simulations that duplicate data from 0 (i.e., no duplication) to 8 times. The four simulations show a similar time in terms of not only event execution and socket communication but also synchronization. Therefore, we conclude that the unbalanced workload during the time window causes a large overhead even if the aggregated workload is balanced. Since the workload balance depends on the simulation setup, addressing this kind of overhead is out of the scope of the design of a parallel simulator.

We also evaluate the lookahead value improvement technique with both strong scaling and weak scaling. Figures 9e and 9f illustrate the performance of the simulator with and without this technique, respectively. For parallel simulation, the left bar represents the larger lookahead value \( \frac{T_{cc}}{2} \), and the right bar represents the smaller lookahead value \( T_{qc} \). As a result, the lookahead value is increased from 2.5 µs to 0.15 ms. This proposed technique reduces the simulation synchronization overhead in all cases and thus improves the scalability of the simulator. We also verify that the number of executed simulation events is identical in all cases, which suggests that the increased lookahead time does not affect the simulation accuracy.

### 6.2 Quantum Network with Caveman Graph Topology

The second network uses the caveman graph topology [44] to form the quantum network. The caveman graph \( \text{Caveman}(n, k) \) is formed from a set of \( n \) isolated \( k \)-node caves by removing one edge from each cave and using it to connect a neighboring cave along a cycle. The community cave structure of the caveman graph allows us to easily group nodes into sets of nodes for parallelization. Nodes in the same cave are assigned to the same process, and neighboring caves are placed together. Compared with networks in Section 6.1, this network serves multiple flows with a smaller number of hops. Figure 10a illustrates the network using the caveman graph \( \text{Caveman}(4, 4) \) as the topology, which denotes that both the size of one cave and the number of caves are 4.

All simulations use the \( \text{Caveman}(128, 8) \) network topology that serves 1,024 random flows. Every node in the caveman graph represents one quantum router, and every edge in the graph represents optical fibers and the BSM node used for the meet-in-the-middle protocol. The distance between adjacent routers is 10 km. A sufficient number of quantum memories is allocated on each node to support all flows that pass through them. In our setup 81,500 quantum memories are used in total. At the beginning of the simulation, every router produces a flow that occupies 25 memories in the source and destination routers of the flow and 50 memories in all intermediate routers. The destination routers are selected randomly based on the number of hops in the flow. The number
of hops follows an exponential distribution with $\lambda = 1$. We assume that the number of short-distance flows is greater than the number of long-distance flows in a realistic quantum network.

Figure 10b illustrates the performance of simulations executed on 1 to 128 processes, with execution time shown as an average of all processes for 5 simulations. Again, we note that the standard deviation in execution time ranges from only 6.0 to 164.0 seconds and thus error bars are not plotted. We evaluate the scalability of the parallel simulation by inspecting the execution time and the speedup with strong scaling. We observe that the execution time decreases as the number of processes grows. The parallel simulation reduces the execution time by a factor up to 20.5. Although the speedup rate of simulating this network is similar to the speedup rate shown in Section 6.1 in scenarios with fewer than 64 processors, the speedup rate decreases around 23% for parallel simulations with 64 and 128 processors. The scalability degradation results from an unbalanced workload. Here we use the number of quantum memories utilized on each process to quantify load imbalance on MPI processes. We calculate the coefficient of variation ($CV$), which divides the standard deviation by the mean. We observe that the value of $CV$ increases from 0.01 (2 processes) to 0.17 (128 processes). The unbalanced distribution of quantum memories causes the unbalanced workload, which increases the overhead and therefore reduces the scalability of parallelization. However, placing routers in the same cluster within one process significantly reduces the amount of cross-process entanglement. Less than 10% of work is forwarded to the global QSM. Even if the 128-process simulation has the most cross-process entanglement states, the simulation only incurs 30-second wait time due to responses from the socket communication. The partition scheme used for caveman networks balances the number of nodes in processes and reduces the number of cross-process entanglement states, but the unbalanced distribution of quantum memories still reduces the scalability. Therefore, we use the next simulation to investigate how partition schemes affect the scalability of parallel simulation on a more complicated topology.

6.3 Quantum Network with Autonomous System Network Topology

We choose the graph resembling the Internet autonomous system (AS) network as the topology of the third simulated network. As with previous simulations, nodes of the graph represent quantum routers, and edges represent optical fibers with BSM nodes used for the meet-in-the-middle protocol. These routers follow the same method introduced in Section 6.2 to produce random flows. We simulate the quantum network with 1,024 routers in the AS network topology. There are 81,300 quantum memories used (again, based on flow usage) to serve 1,024
fig 11. Simulations of an autonomous system network graph

random flows. We investigate the scalability of the parallel simulator under this scenario. We use Networkx [19] to generate the graph $\text{AS}(1024,0)$ that denotes a random 1024-node graph using seed 0.

Figure 11a illustrates a smaller network $\text{AS}(32,1)$. Unlike the previous two network topologies, this network topology includes many edge routers that are connected by the core routers. These core routers have a higher degree than the edge routers, which results in an unbalanced distribution of workload among the routers. The unbalanced workload on the routers is a challenging scenario for the network partition algorithm for parallel simulation.

We propose three heuristic solutions for network partitioning: $\text{Partition1}$, $\text{Partition2}$, and $\text{Partition3}$. The proposed solutions use simulated annealing [42] to optimize their energy functions. We define three required components for the simulated annealing: state, neighboring state, and energy function. State defines the mapping relation between nodes and processes. The initial state evenly assigns routers to processes. Neighboring state is the state that could be generated by swapping two routers in different processes. The initial state and neighboring state make the partition scheme always distribute nodes to processes evenly. The simulated annealing method starts from the initial state and randomly jumps between neighboring states. After a few iterations, the state with the lowest energy is the final partition scheme. Given different energy functions, simulated annealing may produce different partition schemes. We design three energy functions using different assumptions.
(1) \textit{Partition1} assumes that users know all flows in the simulation. The energy function is to calculate the number of cross-process flows.

(2) \textit{Partition2} assumes that users know only the topology of the network. The energy function is to calculate the number of cross-process quantum channels.

(3) \textit{Partition3} assumes that users know how many memories are used in each router. The energy function is designed to calculate the imbalance of memory distribution.

The energy functions of \textit{Partition1} and \textit{Partition2} are designed to reduce the overhead of the socket communication, while the energy function of \textit{Partition3} is designed to reduce the overhead of unbalanced workloads. We conduct parallel simulations with these three partition solutions on 2 to 128 processes. These simulations help us explore how various network partition affects the parallelization performance.

Figure 11b illustrates the execution time of the simulations. We note again that the maximum standard deviation in simulation time is 220.8 seconds, and so error bars are not plotted. For parallel simulations, the left, middle, and right bars represent partitioning using \textit{Partition1}, \textit{Partition2}, and \textit{Partition3}, respectively. We observe that parallel simulations using \textit{Partition3} consume more time for socket communication than the other two partition methods because of the different objective functions. Figure 11c demonstrates that the increased socket communication time results from the additional requests processed by the global QSM. Compared with \textit{Partition1}, \textit{Partition3} requires the local QSM to forward around 30% requests to the global QSM. Among the three methods, \textit{Partition1} performs the best when the number of processes is 2, 4, 64, and 128; \textit{Partition2} performs the worst in most cases; \textit{Partition3} performs the best when the number of processes is 8, 16, and 32. When the parallel simulation uses fewer processes, \textit{Partition1} finds a partition with low overhead from the socket communication, and the imbalance of workload is suppressed by the limited number of processes as well as the equal number of nodes in the processes. As the number of processes increases, the overhead from the unbalanced workload dominates the execution time, and the socket communication time is mitigated by the multithreaded global QSM. Thus, \textit{Partition3} shows the best parallel performance. However, when the number of processes is too large, equally distributing nodes to processes becomes a bottleneck in \textit{Partition3}. For example, with a parallel simulation of 128 processes, we expect every process to control \( \frac{81300}{128} \approx 89 \) memories on average, but the heaviest router controls 2,675 memories. Seven routers need to be placed in the same process with the heaviest router, which deteriorates the workload balance. In addition to the decreased overhead on socket communication, three partition solutions show similar performance when the number of processes is too large.

\textit{Partition3} demonstrates the largest socket communication time. Therefore, those scenarios are good candidates for evaluating the proposed techniques to reduce communication time, as described in Section 4.2. We compare the performance of the parallel simulation with work offloading and requests grouping, respectively. Figure 11d illustrates the normalized execution time of the simulation. We normalize the execution time of parallel simulations by dividing the execution time of the parallel simulation that ignores optimizations on the socket communication. We observe that both optimization techniques always speed up simulations. Offloading work from the global QSM to the local QSMs achieves better performance for simulations with fewer processes. Since the simulation using fewer processes leads to less cross-process entanglement, more manipulations of quantum states can be handled locally instead of using the global QSM. Compared with work offloading, grouping multiple requests to one message achieves a more significant performance improvement, especially for simulations with more processes. By grouping requests, parallel simulations can save more than 75% of the communication time. Moreover, the benefit from this optimization takes less impact from the increased number of processes. After aggregating both optimization methods, we speed up the simulation using 128 processes by 4.5 times and the simulation using 2 processes by 11.5 times, which significantly improves the scalability of our simulator.
7 RELATED WORK

Recently, several discrete-event quantum network simulators have been developed, including QuISP [29], NetSquid [8], SQUANCH [3], and SeQUeNCe [47–49]. These simulators offer a flexible and reproducible testing environment to study the performance of quantum networks, but differ in the supported features, hardware and protocol models, as well as simulation use cases. With the exception of SQUANCH, the performance of these simulators has been limited due to their sequential execution of simulation events. We believe that the techniques that allowed us to parallelize quantum state management in the SeQUeNCe simulator can be transferred to the other simulators.

Thus far, large-scale quantum network simulations used sequential processing and achieved improved scalability by relying on simplified qubit representations. For example, QuISP [29] aims to simulate a full Quantum Internet consisting of up to 100 networks of up to 100 nodes each, and scalability is achieved by replacing full quantum-state tracking by maintaining an error model of qubits in large networks. NetSquid allows the user to choose between several quantum state representations that vary in their scalability, namely ket vector, density matrix, stabiliser, and graph state representation. The NetSquid simulator was successfully used in a recent simulation of a chain of up to 1,000 nodes [8]. SQUANCH [3] is notable for allowing parallelized simulations that use multiprocessing where each process manages one simulated network node. However, the simulator does not attempt to model interactions of protocols or details at the physical layer, and it does not guarantee the order of event executions.

Several classical network simulators support parallelization, such as ns-3 [33] and OMNeT++ [34]. However, parallelization of classical network simulators is simpler due to the absence of exponentially large quantum state representations and non-local effects of quantum state measurements. Kazer et al. [25] used the built-in MPI-based PDES framework of OMNeT++ to simulate a highly interconnected data center network and found that synchronization can actually cause parallel DES to perform worse than a single-threaded implementation. Nikolaev et al. [32] evaluated the performance of the distributed ns-3 network simulator and concluded that it scales only to about 100–200 cores/processes. Beyond that limit, performance suffers because of increased interprocess synchronization and communication costs. Barnes et al. [2, 31, 32] simulated networks with 1 billion nodes (routers and hosts) using ns-3 on a supercomputer. Their scalability metric and their implementation of a NULL message synchronization algorithm [17] may be useful in future work.

Our recent work [50] demonstrated that parallelization can greatly speed up the simulation of quantum key distribution networks [21]. In that work, a network model was effectively divided into multiple components, each of which was executed on a different logic process with a synchronization mechanism to ensure the correct execution order of cross-process events. A QKD network composed of QKD terminals and channels for transmitting photons as well as packets was simulated. Similarly as in this work, we applied the conservative synchronization algorithm [30]. However, this earlier work was limited to tracing quantum superposition states and did not consider distribution of entanglement which is necessary in general-purpose quantum communication networks. In this paper we address this shortcoming by designing and implementing simulation techniques that use distributed quantum state management that allows efficient parallel simulation of entanglement distribution.

8 CONCLUSION

In this paper we present a parallelized version of the SeQUeNCe simulator for quantum communication networks. The simulator uses a conservative synchronization algorithm for maintaining event execution order between parallel processes. A quantum state manager (QSM) was developed to track shared quantum states among processes, with a centralized global QSM servicing requests from local QSMs on individual processes. We additionally investigate multiple methods for optimizing the performance of a quantum network simulator.
Furthermore, by working within the modularized architecture of the SeQUeNCe simulator, we provide a parallel simulator that requires minimal adaptations to sequentially run simulations. The kernel module, which processes events and stores quantum states, was fully parallelized while leaving the higher-level hardware and protocol layers with few changes. Our design ensures identical results for sequential and parallel simulations in conjunction with the well-ordered event execution guaranteed by the conservative synchronization algorithm.

We are releasing the parallelized SeQUeNCe simulator as open-source software with the goal of providing an easy avenue for constructing parallel simulations whose performance will enable development of future large-scale quantum networks. Our plans for future development of the SeQUeNCe simulator include introduction of new optical hardware models, including new types of quantum memories, and the release of an interactive GUI for the intuitive construction of new simulations.

ACKNOWLEDGMENTS

We thank Siu Man Chan for contributions to the SeQUeNCe software package and Gail Pieper for help with editing. The work of M. S., R. K. and A. K. is supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers. The work of X. W. and D. J. is partly sponsored by the Air Force Office of Scientific Research (AFOSR) under Grant YIP FA9550-17-1-0240 and the National Science Foundation (NSF) under Grant CNS-1730488 and DMR-1747426. We gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] Sigrún Andradóttir and Teunis J. Ott. 1995. Time-Segmentation Parallel Simulation of Networks of Queues with Loss or Communication Blocking. ACM Trans. Model. Comput. Simul. 5, 4 (Oct. 1995), 269–305. https://doi.org/10.1145/226275.226276
[2] PD Barnes. 2011. A benchmark model for parallel ns3. Technical Report. Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States).
[3] Ben Bartlett. 2018. A distributed simulation framework for quantum networks and channels. arXiv preprint arXiv:1808.07047 (2018).
[4] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. Phys. Rev. Lett. 70 (Mar 1993), 1895–1899. Issue 13. https://doi.org/10.1103/PhysRevLett.70.1895
[5] H-J Briegel, Wolfgang Dür, Juan I Cirac, and Peter Zoller. 1998. Quantum repeaters: the role of imperfect local operations in quantum communication. Physical Review Letters 81, 26 (1998), 5932.
[6] Christopher D Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. J. Parallel and Distrib. Comput. 62, 11 (2002), 1648–1669.
[7] CODES-org. 2021. CODES Discrete-event Simulation Framework. https://github.com/codes-org/codes.
[8] Tim Coopmans, Robert Kneegjens, Axel Dahlgberg, David Maier, Loek Nijsten, Julio de Oliveira Filho, Martijn Papendrecht, Julian Rabbie, Filip Rozpędek, Matthew Skrzypeczek, Leon Wubben, Walter de Jong, Damian Podareanu, Ariana Torres-Knoop, David Elkouss, and Stephanie Wehner. 2021. NetSquid, a NETwork Simulator for QUantum Information using Discrete events. Communications Physics 4, 1 (2021), 164. https://doi.org/10.1038/s42005-021-00647-8
[9] Rachel Courtland. 2016. China’s 2,000-km quantum link is almost complete [News]. IEEE Spectrum 53, 11 (2016), 11–12. https://doi.org/10.1109/MSPEC.2016.7607012
[10] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An In-Depth Analysis of the Slingshot Interconnect. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. 1–14. https://doi.org/10.1109/SC41405.2020.00039
[11] Wolfgang Dür and Hans J Briegel. 2007. Entanglement purification and quantum error correction. Reports on Progress in Physics 70, 8 (2007), 1381.
[12] JF Dynes, Adrian Wonfor, WW-S Tam, AW Sharpe, R Takahashi, M Lucamarini, A Plews, ZL Yuan, AR Dixon, J Cho, et al. 2019. Cambridge quantum network. npj Quantum Information 5, 1 (2019), 1–8.
[13] A. Einstein, B. Podolsky, and N. Rosen. 1935. Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? Physical Review 47, 10 (May 1935), 777–780. https://doi.org/10.1103/PhysRev.47.777
Parallel Simulation of Quantum Networks with Distributed Quantum State Management

[14] Chip Elliott, Alexander Colvin, David Pearson, Oleksiy Pikalo, John Schlafer, and Henry Yeh. 2005. Current status of the DARPA quantum network. In Quantum Information and computation III, Vol. 5815. International Society for Optics and Photonics, 138–149.

[15] Ahmed Elmokashfi, Amund Kvalbein, and Constantine Dovrolis. 2010. On the scalability of BGP: The role of topology growth. IEEE Journal on Selected Areas in Communications 28, 8 (2010), 1250–1261.

[16] Ugo Fano. 1957. Description of states in quantum mechanics by density matrix and operator techniques. Reviews of Modern Physics 29, 1 (1957), 74.

[17] Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation. Commun. ACM 33, 10 (Oct. 1990), 30–53. https://doi.org/10.1145/84537.84545

[18] Vlad Gheorghiu. 2018. Quantum++: A modern C++ quantum computing library. PloS one 13, 12 (2018).

[19] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. Exploring network structure, dynamics, and function using NetworkX. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[20] INQNET. 2021. Fermilab Quantum Network (FQNET). http://inqnet.caltech.edu/fqnet/.

[21] J Robert Johansson, Paul D Nation, and Franco Nori. 2012. QuTiP: An open-source Python framework for the dynamics of open quantum systems. Computer Physics Communications 183, 8 (2012), 1760–1772.

[22] Cody Jones, Danny Kim, Matthew T Rakher, Paul G Kwiat, and Thaddeus D Ladd. 2016. Design and analysis of communication protocols for quantum repeater networks. New Journal of Physics 18, 8 (2016), 083015.

[23] K. C. Jones and S. R. Das. 2001. Time-parallel algorithms for simulation of multiple access protocols. In MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. 49–58. https://doi.org/10.1109/MASCOT.2001.948853

[24] Charles W Kazer, João Sedoc, Kelvin KW Ng, Vincent Liu, and Lyle H Ungar. 2018. Fast Network Simulation Through Approximation or: How Blind Men Can Describe Elephants. In Proceedings of the 17th ACM Workshop on Hot Topics in Networks. 141–147.

[25] INQNET. 2021. Fermilab Quantum Network (FQNET). http://inqnet.caltech.edu/fqnet/.

[26] Paul Leach, Michael Mealling, and Rich Salz. 2005. A universally unique identifier (uuid) urn namespace.

[27] Sergei Nikolaev, Eddy Banks, Peter D. Barnes, David R. Jefferson, and Steven Smith. 2015. Pushing the Envelope in Distributed ns-3. In Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques (SimuTools ’13). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 17–23.

[28] Louise Lerner. 2018. Quantum network to test unhackable communications. https://www.anl.gov/article/quantum-network-to-test-unhackable-communications.

[29] Takaaki Matsuo, Clément Durand, and Rodney Van Meter. 2019. Quantum link bootstrapping using a RuleSet-based communication protocol. Physical Review A 100, 5 (2019), 052320.

[30] David M Nicol. 1996. Principles of conservative parallel simulation. In Proceedings of the 28th conference on Winter simulation. 128–135.

[31] Sergei Nikolaev, Eddy Banks, Peter D. Barnes, David R. Jefferson, and Steven Smith. 2015. Pushing the Envelope in Distributed ns-3 Simulations: One Billion Nodes (WNS3 ’15). Association for Computing Machinery, New York, NY, USA, 67–74. https://doi.org/10.1145/2756509.2756525

[32] Sergei Nikolaev, Peter D. Barnes, James M. Brase, Thomas W. Canales, David R. Jefferson, Steve Smith, Ron A. Soltz, and Peter J. Scheibel. 2013. Performance of Distributed ns-3 Network Simulator. In Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques (Cannes, France) (SimuTools ’13). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 17–23.

[33] n3s 2021. ns-3: network simulator. https://www.nsnam.org/.

[34] OMNET 2021. OMNeT++: Discrete Event Simulator. https://omnetpp.org/.

[35] GitHub repository. 2020. SeQuence: Simulator of Quantum Network Communication. https://github.com/sequence-toolbox/SeQuence

[36] George Riley. 1999. PDNS - Parallel/Distributed NS. https://www.cc.gatech.edu/computing/pads/pdns.html.

[37] George F. Riley. 2003. The Georgia Tech Network Simulator. In Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (Karlsruhe, Germany) (MoMeTools ’03). Association for Computing Machinery, New York, NY, USA, 5–12. https://doi.org/10.1145/944773.944775

[38] Ramamurti Shankar. 2012. Principles of quantum mechanics. Springer Science & Business Media. 5 pages.

[39] Christoph Simon. 2017. Towards a global quantum network. Nature Photonics 11, 11 (2017), 678–680. https://doi.org/10.1038/s41566-017-0032-0

[40] Mirko Stoffers, Daniel Schiemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2018. On automated memoization in the field of simulation parameter studies. ACM Transactions on Modeling and Computer Simulation (TOMACS) 28, 4 (2018), 1–25.
[42] Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 7–15.

[43] David W Walker and Jack J Dongarra. 1996. MPI: a standard message passing interface. *Supercomputer* 12 (1996), 56–68.

[44] Duncan J Watts. 1999. Networks, dynamics, and the small-world phenomenon. *American Journal of sociology* 105, 2 (1999), 493–527.

[45] HPC Wiki. 2020. Scaling tutorial. https://hpc-wiki.info/hpc/index.php?title=Scaling_tutorial&oldid=3609.

[46] H. Wu, R. M. Fujimoto, and M. Ammar. 2003. Time-parallel trace-driven simulation of CSMA/CD. In *Seventeenth Workshop on Parallel and Distributed Simulation, 2003. (PADS 2003). Proceedings*. 105–112. https://doi.org/10.1109/PADS.2003.1207426

[47] Xiaoliang Wu, Joaquin Chung, Alexander Kolar, Eugene Wang, Tian Zhong, Rajkumar Kettimuthu, and Martin Suchara. 2019. Photon-Level Simulation of Quantum Key Distribution with Picosecond Accuracy. In *Single Photon Workshop*.

[48] Xiaoliang Wu, Joaquin Chung, Alexander Kolar, Eugene Wang, Tian Zhong, Rajkumar Kettimuthu, and Martin Suchara. 2019. Simulations of Photonic Quantum Networks for Performance Analysis and Experiment Design. In *2019 IEEE/ACM Workshop on Photonics-Optics Technology Oriented Networking, Information and Computing Systems (PHOTONICS)*. IEEE, 28–35.

[49] Xiaoliang Wu, Alexander Kolar, Joaquin Chung, Dong Jin, Tian Zhong, Rajkumar Kettimuthu, and Martin Suchara. 2021. SeQUeNCe: a customizable discrete-event simulator of quantum networks. *Quantum Science and Technology* 6, 4 (2021), 045027. https://doi.org/10.1088/2058-9565/ac22f6

[50] Xiaoliang Wu, Bo Zhang, and Dong Jin. 2020. Parallel Simulation of Quantum Key Distribution Networks. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 187–196.

[51] Marek Żukowski, Anton Zeilinger, Michael A Horne, and Artur K Ekert. 1993. Event-ready-detectors Bell experiment via entanglement swapping. *Physical Review Letters* 71, 26 (1993), 4287.