Abstract—Simulation of quantum systems is challenging due to the exponential size of the state space. Tensor networks provide a systematically improvable approximation for quantum states. 2D tensor networks such as Projected Entangled Pair States (PEPS) are well-suited for key classes of physical systems and quantum circuits. However, direct contraction of PEPS networks has exponential cost, while approximate algorithms require computations with large tensors. We propose new scalable algorithms and software abstractions for PEPS-based methods, accelerating the bottleneck operation of contraction and refactorization of a tensor subnetwork. We employ randomized SVD with an implicit matrix to reduce cost and memory footprint asymptotically. Further, we develop a distributed-memory PEPS library and study accuracy and efficiency of alternative algorithms for PEPS contraction and evolution on the Stampede2 supercomputer. We also simulate a popular near-term quantum algorithm, the Variational Quantum Eigensolver (VQE), and benchmark Imaginary Time Evolution (ITE), which compute ground states of Hamiltonians.

Index Terms—Tensor network, Projected Entangled Pair States, Quantum system

I. INTRODUCTION

The degrees of freedom required to directly represent a quantum state grow exponentially with respect to the size of the system. However, for slightly entangled quantum states, polynomial-size representations are possible by using tensor networks [1], [2]. The 1D tensor networks, also known as matrix product states (MPS), have proven successful in modeling 1D strongly correlated quantum systems [3], and various types of tensor networks have been invented to generalize this method to higher dimensions [4]–[7]. Among these variations, the 2D tensor lattice networks, known as projected entangled pair states (PEPS), enable effective representation of many physical systems of interest due to their 2D structure and are therefore widely used in condensed matter physics [4], [8], [9]. For the same reason, PEPS are also suitable to model near-term quantum computers with 2D architectures [10], [11]. In fact, this method has been recently applied to a large-scale exact simulation of 2D random quantum circuits [12].

Computations with PEPS involve high-order tensors and are computationally demanding. For example, exact contraction of PEPS to a scalar to calculate an expectation value is a #P-complete problem [13]. Approximations must be taken to avoid exponential cost, but the order of complexity is still high. Given a PEPS network of bond dimension $r$, i.e., composed of tensors that are of dimension $r \times r \times r \times r$, and maintaining approximate intermediates that have bond dimensions no larger than some $m \propto r^2$, state-of-the-art PEPS contraction algorithms have a cost that scales as $O(m^3 r^2)$ via an iterative optimization procedure.

In search for an efficient parallel implementation for these PEPS algorithms, we propose an abstraction called einsumsvd for operations that contract a set of tensors into one and then refactorize it into two new tensors as described in Section II. This abstraction encapsulates the most costly operations in many tensor network applications, and it can be implemented with a variety of algorithms. In Section IV, we discuss an approach for performing einsumsvd that leverages randomized singular value decomposition (SVD) with implicit applications of the tensor operator. When applying this idea to the contraction of PEPS, we obtain a new algorithm that minimizes the cost relative to state-of-the-art approaches, while requiring only a single pass over the tensors. In the same section, we also present an intermediate caching strategy that further lowers the cost of calculating expectation values by trading space for time.

We implement these algorithms as part of a new Python library called Koala for PEPS-based simulations of quantum systems (Section V). This library supports various PEPS evolution and contraction algorithms, including our improvements, in the sequential/threaded setting using NumPy [14], the GPU setting using CuPy [15], and the distributed-memory setting using Cyclops [16], [17]. To improve the performance with Cyclops, we propose a parallel approach for tensor orthogonalization that avoids matricization of a high-order tensor, reducing the need to rearrange data. This approach forms small Gram matrices in local memory so that matricization can be done locally while large-tensor contractions are kept distributed.

We evaluate the proposed algorithms and software in Section VI via a performance study of the PEPS evolution and contraction methods, using single-node and many-node execution on the Stampede2 supercomputer. Our experimental results demonstrate that the Gram matrix method improves the parallel performance of PEPS evolution up to 3.7X, and the randomized SVD approach improves the performance of PEPS contraction up to 19X in the sequential/threaded setting and up to 7X in the distributed-memory setting (combined with the Gram matrix method). The performance improvement due to the proposed caching optimization is also quantified.
in this section. For parallel scaling, we perform strong and weak scaling tests with up to 64 nodes with 64 cores per node and show good weak scaling performance with respect to the tensor size (i.e. bond dimension).

As driver applications, we consider a common method for ground state calculation using tensor networks, imaginary time evolution (ITE), as well as a simulation of a prominent candidate algorithm for near-term quantum computers, the variational quantum eigensolver (VQE). These two applications, which are representative of many algorithms in the domain, are both implemented by the PEPS evolution and contraction primitives provided by our library. In Section VI-D, we study the numerical accuracy achieved by PEPS simulations with various bond dimensions for ITE and VQE. Our results demonstrate that higher accuracy is achieved by the use of larger tensors (i.e. larger bond dimension) with PEPS.

Overall, this paper makes the following contributions:

- a new approach for PEPS contraction that minimizes cost and does not require successive rounds of optimization,
- an automatic mechanism to cache intermediate contraction results during PEPS contraction,
- novel distributed-memory implementations of multiple PEPS contraction algorithms,
- a software library, Koala, providing sequential and parallel primitives for PEPS-based approximate simulations of quantum systems as well as encapsulating above algorithms via new software abstractions,
- to the best of our knowledge, the first study of accuracy and scalability of massively-parallel approximate PEPS algorithms with distributed-memory tensors.

A comparison between this work and related work is presented in Section VII.

II. BACKGROUND

In this section, we provide a brief review on tensor networks and their applications with a focus on the concepts used in this paper. Broader surveys on these topics are available [1]. We list notational conventions employed in the paper in Table I.

### Table I: Common notation used in this paper.

| Notation | Meaning |
|-----------|---------|
| \( \mathcal{A} \) (bold, calligraphic capital letters) | Tensor network |
| \( \mathbf{A} \) (bold capital letters) | Matrix or high-order tensor |
| \( \alpha \) (bold lowercase letters) | Vector |
| \( a_{ijk} \) | Element of the tensor \( \mathbf{A} \) |
| \( a^* \), \( \mathbf{A}^* \) | Complex conjugate of a scalar \( a \) or conjugate transpose of an operator \( \mathbf{A} \) |
| \( \mathbf{A} \in \mathbb{C}^{d_1 \times \cdots \times d_n} \) | Complex tensor \( \mathbf{A} \) of dimension \( d_1 \times \cdots \times d_n \) |
| \( \mathbf{A} : \mathbb{C}^{p_1 \times \cdots \times p_s} \rightarrow \mathbb{C}^{q_1 \times \cdots \times q_t} \) | Complex tensor \( \mathbf{A} \) treated as a linear operator |
| \( |\psi\rangle \) | Quantum state labeled by \( \psi \) |
| \( \langle \psi | \) | Conjugate transpose of \( |\psi\rangle \) |

A. Quantum States as Tensors

The state \( |\psi\rangle \) of a quantum computer with \( n \) qubits can be described by a unit vector in \( \mathbb{C}^{2^n} \). By choosing \( 2^n \) orthonormal basis vectors/states to be denoted as \( |i\rangle \) with \( i = i_1 \cdots i_n \in \{0, 1\}^n \), \( |\psi\rangle \) can be written as

\[
|\psi\rangle = \sum_{i \in \{0,1\}^n} t_i^{(\psi)} |i\rangle.
\]

Here, the amplitudes \( t_i^{(\psi)} \) are elements of an order \( n \) tensor \( T^{(\psi)} \in \mathbb{C}^{2^n \times 2^n} \). The inner product of two quantum states \( |\psi\rangle \) and \( |\phi\rangle \) is given by

\[
\langle \phi | \psi \rangle = \sum_{i \in \{0,1\}^n} (t_i^{(\phi)})^* t_i^{(\psi)},
\]

and an operator \( \mathbf{H} \) acting on a quantum state is a linear transformation given by

\[
|\phi\rangle = \mathbf{H} |\psi\rangle \quad \Rightarrow \quad t_i^{(\phi)} = \sum_{j \in \{0,1\}^n} h_{ij} t_j^{(\psi)}.
\]

The expectation value of the Hermitian operator \( \mathbf{H} \) for a quantum state \( \psi \) is defined by \( \langle \psi | \mathbf{H} |\psi\rangle \), following the definitions of operators and inner products.

A quantum gate is an operator that can act on a small subset of qubits. For example, a single qubit gate \( G^{(k)} \) acting on the \( k \)th qubit gives

\[
|\phi\rangle = G^{(k)} |\psi\rangle \quad \Rightarrow \quad t_i^{(\phi)} = \frac{1}{2^n} \sum_{j=0}^{2^n} \delta_{i,j} t_j^{(\psi)}, \quad (1)
\]

while a 2-qubit gate \( G^{(k,l)} \) acting on qubits \( k, l \) with \( k < l \) gives

\[
|\phi\rangle = G^{(k,l)} |\psi\rangle \quad \Rightarrow \quad t_i^{(\phi)} = \frac{1}{2^n} \sum_{j,k=0}^{2^n} \delta_{i,j} \delta_{j,l} t_j^{(\psi)}, \quad (2)
\]

These operator applications are particular examples of tensor contractions, which allow for different types of products among tensors.

B. Tensor Networks for Quantum States

The contraction of a set of tensors defines a tensor network, which could serve to provide a tensor decomposition \([18], [19]\) of a quantum state.

The matrix product state (MPS) \([20], [21]\) is a 1D tensor network that describes a quantum state \( \psi \) with \( n \) sites by \( n \) tensors \( M^{(1)}, \ldots, M^{(n)} \), such that

\[
t_i^{(\psi)} = \sum_{k_1 \cdots k_{n-1}} \frac{1}{n!} m^{(1)}_{i_1 k_1} m^{(2)}_{i_2 k_1 k_2} \cdots m^{(n-1)}_{i_{n-1} k_{n-2} k_{n-1}} m^{(n)}_{i_n k_{n-1}}.
\]

By convention, \( i_1, \ldots, i_n \) are called physical indices and \( k_1, \ldots, k_{n-1} \) are called bond indices, while their dimensions are called physical dimensions and bond dimensions respectively.
An operator $H$ may also be described as a 1D tensor network, which is known as the matrix product operator (MPO) [1], [22], [23]:

$$h_{i_{1}...i_{n},j_{1}...j_{n}} = \sum_{k_{1}...k_{n-1}} m_{i_{1}j_{1}k_{1}}^{(1)} m_{i_{2}j_{2}k_{1}k_{2}}^{(2)} ... m_{i_{n}j_{n}k_{n-1}}^{(n)}.$$

The projected entangled pair state (PEPS) [21] is a 2D tensor network that represents a quantum state with $n^2$ sites on a square lattice by $n^2$ tensors in a way analogous to MPS. These $n^2$ tensors are arranged on an $n \times n$ grid, so one can denote the tensor at the $p$'th row and $q$'th column as $M^{(p,q)}$. Then, each amplitude of a quantum state $|\psi\rangle$ is defined as

$$k_{i_{1}...i_{n_{2}}}=\sum_{k} m_{i_{p}=i_{q}=k(p,q)}^{(p,q)}$$

where $k$ denotes all the indices shared by tensors in the PEPS and $k^{(p,q)}$ represents all the indices of the site tensor $M^{(p,q)}$ that are shared with other site tensors. Similarly to MPS, indices $i_{1}...i_{n_{2}}$ are called physical indices and $k^{(p,q)}$ are called bond indices.

A tensor network can be visualized by a graph (tensor diagram), in which each vertex denotes a tensor and each edge denotes an index. Tensor diagrams describing MPS, MPO, and PEPS are displayed in Figure 1.

![Figure 1: Tensor diagrams for (a) an MPS, (b) an MPO and (c) a 3 × 3 PEPS.](image)

**C. Basic Tensor Network Computations**

We now describe common computational primitives to transform tensor networks, focusing on PEPS.

1) One-Site and Two-Site Operators: To apply a one-site operator $G$ on the site $M^{(p,q)}$ of a PEPS, it suffices to compute the updated site $M^{(p,q)}$ by performing the contraction,

$$m_{i_{k_{1}}k_{2}k_{3}k_{4}}^{(p,q)} = \sum_{j} g_{ij}^{(1)} m_{j_{k_{1}}k_{2}k_{3}k_{4}}^{(p,q)}.$$  

To apply a two-site operator $G$ on two neighboring sites $M^{(p,q)}$ and $M^{(p,q+1)}$, we consider a variant of the simple update algorithm [24]. Two site tensors are contracted with the operator and then decomposed into two new site tensors $M^{(p,q)}$ and $M^{(p,q+1)}$,

$$\sum_{k_{5}} m_{i_{1}k_{1}k_{2}k_{3}k_{4}}^{(p,q)} \approx \sum_{j_{1},j_{2}} g_{j_{1}j_{2}i}^{(2)} m_{j_{1}k_{1}k_{2}k_{3}k_{4}}^{(p,q)} m_{j_{2}k_{5}k_{6}k_{7}}^{(p,q+1)}.$$  

This contraction and refactorization procedure is described by the first and last tensor diagrams in Figure 4. Applying a two-site operator on non-neighboring sites is possible by applying a chain of two-site operators (i.e. SWAP gates) on neighboring sites.

The contraction and refactorization necessary to apply two-site operators may be executed by contracting the tensors (utilizing einsum in Python’s NumPy), followed by reshaping the result into a matrix, computing a low-rank matrix factorization such as the truncated SVD, and reshaping the resulting factors into the appropriate tensors. We refer to the combined refactorization operation as einsumsvd. This primitive takes as input a tensor network and produces a two-site tensor network with a single virtual leg connecting the two sites. With the notion of einsumsvd, we can rewrite Equation (4) as

$$\tilde{M}^{(p,q)} = \tilde{M}^{(p,q+1)} \leftarrow \text{einsumsvd}(G, M^{(p,q)}, M^{(p,q+1)}).$$

Figure 2 depicts other examples of einsumsvd.

![Figure 2: einsumsvd examples: (a) a 5-site tensor network is contracted and then refactorized into two sites. (b) PEPS contraction step in which two tensors that share a neighbor are merged and the combined dimension is truncated.](image)

2) Amplitudes and Expectation Values: A given amplitude of a state $|\psi\rangle$, such as $t_{1}^{(p)}$ may be obtained by computing $\langle \phi | T_{1}^{(p)} | \psi \rangle$. When represented by PEPS, the bond dimension of $|\phi\rangle$ is 1 and each component of $|\phi\rangle$ can be applied to the respective site of $|\psi\rangle$, resulting in a contraction of a PEPS without physical dimensions, i.e., a one-layer contraction.

Consider a Hermitian operator $H$ composed of local terms $H_{i}$, where $H_{i}$ are operators acting on local sites of the PEPS. One way to calculate its expectation value is to calculate it as a sum of the expectation values of $H_{i}$,

$$\langle \psi | H | \psi \rangle = \sum_{i=1}^{N} \langle \psi | H_{i} | \psi \rangle = \sum_{i=1}^{N} \langle \psi | \phi_{i} \rangle.$$  

$|\phi_{i}\rangle$ can be obtained by the algorithms of applying local operators on $|\psi\rangle$. The calculation of $\langle \psi | \phi_{i} \rangle$ involves a contraction of two PEPS, i.e. a two-layer contraction, as depicted in Figure 3.

Both one-layer and two-layer contractions require exponential cost in the number of sites if done in an exact way. A variety of approximate schemes with polynomial cost exists, which we survey in Section III.

**D. Tensor Network Applications**

Most tensor network computations are based on the operator application and contraction primitives introduced in the...
previous subsection. We provide a technical introduction to two representative methods below.

1) Imaginary Time Evolution: Consider a Hamiltonian $H = \sum_{j=1}^{N} H_j$, where $H_j$ are local operators. Imaginary time evolution (ITE) evolves a quantum state toward its ground state by repetitively applying the ITE operator, $e^{-\tau H}$, in effect computing power iteration on the matrix $e^{-\tau H}$, which generally converges to the eigenvector with the smallest eigenvalue of $H$. ITE performs this iteration via the time-evolution block decimation (TEBD) algorithm, which enables the approximate evolution of a tensor network state by decomposing $e^{-\tau H}$ into a sequence of local operators using the Trotter-Suzuki evolution of a tensor network state by decomposing.

2) Quantum Circuit Simulation: Following the prescription for gate application described in Section II-A, any quantum circuit can be expressed in terms of one-site and two-site operators acting on an initial state (e.g. a basis state). Simulation of quantum circuits is useful for the development and verification of quantum algorithms and software. Approximate simulation additionally sheds insight on the propagation of error within quantum algorithms, which is useful for understanding the effect of noise on computations with near-term quantum devices [25]. Among the quantum algorithms for near-term quantum devices, the variational quantum eigen solver (VQE) [26] is one of the most promising.

VQE is a quantum-classical hybrid algorithm that computes the ground state of a Hamiltonian $H$ by optimizing a quantum state parameterized by a quantum circuit. The quantum part of the algorithm evaluates the objective $\langle \psi(\theta) | H | \psi(\theta) \rangle$ for given parameters $\theta$, and the classical part of the algorithm takes the measurement results from the quantum device and runs classical optimization algorithms to tune the circuit parameters. A careful design of the circuit structure, depending on the Hamiltonian of interest, is critical in improving the optimization efficiency and achieving desirable accuracy. A number of efforts have been made to design more efficient circuits for different Hamiltonians [27] and to apply the algorithm to quantum chemistry simulations [28]-[30].

III. PREVIOUS WORK ON PEPS ALGORITHMS

Our contributions build directly on existing algorithms for evolution and contraction of PEPS.

A. PEPS Evolution Algorithms

Direct evaluation of Equation 4 reveals a time complexity of $O(d^3r^3)$ and space complexity of $O(d^2r^6)$, where $d$ is the size of the physical dimension and $r$ is the size of the bond dimension. Assuming $d \ll r$, the time and space complexity can be reduced to $O(d^2r^6)$ and $O(dhr^4)$ respectively by a QR factorization of the two site tensors prior to the application of the operator [25], as shown in Algorithm 1 and Figure 4.

Algorithm 1 Operator application by QR-SVD

Input: two site tensors $M^{(p,q)}$ and $M^{(p,q+1)}$, a two-site operator $G$

Output: updated site tensors $\tilde{M}^{(p,q)}$ and $\tilde{M}^{(p,q+1)}$

1. $Q^{(p,q)}, R^{(p,q)} \leftarrow QR(M^{(p,q)})$ \{step (1) $\rightarrow$ (2)\}
2. $Q^{(p,q+1)}, R^{(p,q+1)} \leftarrow QR(M^{(p,q+1)})$ \{step (1) $\rightarrow$ (2)\}
3. $\tilde{R}^{(p,q)}, \tilde{R}^{(p,q+1)} \leftarrow \text{einsvmd}(G,R^{(p,q)},R^{(p,q+1)})$ \{step (2) $\rightarrow$ (4)\}
4. $\tilde{M}^{(p,q)} \leftarrow Q^{(p,q)} \tilde{R}^{(p,q)}$ \{step (4) $\rightarrow$ (5)\}
5. $\tilde{M}^{(p,q+1)} \leftarrow Q^{(p,q+1)} \tilde{R}^{(p,q+1)}$ \{step (4) $\rightarrow$ (5)\}
6. return $\tilde{M}^{(p,q)}, \tilde{M}^{(p,q+1)}$

B. PEPS Contraction Algorithms

Many approximate PEPS contraction algorithms have been proposed in pursuit of reducing memory footprint while maintaining high accuracy. These include the original Boundary MPS (BMPs) [4] algorithm and its variants [31]-[33], Tensor Renormalization Group (TRG) [34] and many TRG-inspired renormalization group methods [35]-[42], Corner Transfer Matrix (CTM) [43], [44], as well as contraction algorithms for new 2D tensor network structures based on PEPS [45], [46]. We focus on a variant of BMPS for one-layer and two-layer contractions as introduced below.

1) One-Layer BMPS: Let $P$ be an $n$-by-$n$ PEPS of bond dimension $r$ without physical indices and $P^{(1)}, \ldots, P^{(n)}$ be the rows of $P$. Treating $P^{(1)}$ as an MPS and $P^{(2)}, \ldots, P^{(n)}$ as MPOs, contracting $P$ by BMPS can be considered as applying MPOs $P^{(2)}, \ldots, P^{(n)}$ on the MPS $P^{(1)}$ approximately.
The approximation is necessary for avoiding exponential cost and it is done by truncating the bond dimension of result MPS to a predefined $m$. This approach is described in Algorithm 2.

**Algorithm 2 Boundary MPS**

**Input:** $n \times n$ PEPS $\mathcal{P}$ consisting of rows $\{\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(n)}\}$,

algorithm ApproxApply, truncation bond dimension $m$

**Output:** scalar $v$

1: $\mathcal{S} \leftarrow \mathcal{P}^{(1)}$
2: for $i = 2, 3, \ldots, n$ do
3: $\mathcal{S} \leftarrow$ ApproxApply$(\mathcal{S}, \mathcal{P}^{(i)}, m)$
4: end for
5: $\mathcal{S}$ is an MPS without physical indices at this point
6: $v \leftarrow$ contracting $\mathcal{S}$ to a scalar
7: return $v$

The approximate application of an MPO on an MPS can be performed in various ways. The original BMPS algorithm [4] involves solving a linear equation $\mathcal{E} \mathcal{T} = \mathcal{P}$ for one site $\mathcal{T}$ while fixing the other sites (the environment $\mathcal{E}$) at a time with respect to the target MPS $\mathcal{P}$. This method, while having a low cost of $O(dnm^2r^3) + O(mnr^2)$ if the MPO has a two-layer structure as discussed below [33], is typically done with multiple rounds of alternating optimization of the sites of the MPS. Our approach is most closely related to the zip-up algorithm proposed in [47], where a series of einsumsvds are conducted to truncate the bond dimension of $\mathcal{P}$, as shown in Figure 5 and described in Algorithm 3.

![Figure 5: Apply MPO on MPS approximately (Algorithm 3).](image)

**Algorithm 3 Apply MPO on MPS approximately**

**Input:** MPS $\mathcal{S}$ consisting of tensors $\{S^{(1)}, \ldots, S^{(n)}\}$,

MPO $\mathcal{O}$ consisting of tensors $\{O^{(1)}, \ldots, O^{(n)}\}$,

algorithm einsumsvd, truncation bond dimension $m$

**Output:** MPS $\mathcal{S}'$

1: $V^{(1)} \leftarrow$ Contract $S^{(1)}$ and $O^{(1)}$
2: for $i = 2, 3, \ldots, n$ do
3: $V^{(i)} \leftarrow$ einsumsvd$(V^{(i-1)}, S^{(i)}, O^{(i)}, m)$
4: end for
5: $\mathcal{S}' \leftarrow \{V^{(1)}, \ldots, V^{(n)}\}$
6: return $\mathcal{S}'$

2) Two-Layer BMPS: Given two $n$-by-$n$ PEPS $\mathcal{A}$ and $\mathcal{B}$ of bond dimension $r_1$ and $r_2$ respectively, their inner product can be computed in two ways. The naive way is to contract over all the physical indices between $\mathcal{A}$ and $\mathcal{B}$ to create a new $n$-by-$n$ PEPS $\mathcal{P}$ with bond dimensions $r = r_1r_2$ and no physical dimensions, and then perform one-layer BMPS. However, forming the combined $\mathcal{P}$ requires $O(r_1^3r_2^3)$ memory, while storing $\mathcal{A}$ and $\mathcal{B}$ separately only needs $O(r_1^2 + r_2^2)$. An improvement is to maintain the two-layer structure and only contract the corresponding sites from two layers to one layer when needed, which is numerically equivalent to the naive way but saves memory. This approach is often used in practice and referred to interchangeably with the naive approach in the literature. In Section IV, we show that when einsumsvd is equipped with implicit randomized SVD, the implicit structure of the two-layer BMPS approach yields reduced computational complexity.

**IV. NEW ALGORITHMS FOR PEPS SIMULATION**

In this section, we describe our algorithmic improvements to basic PEPS computations, specifically einsumsvd and the expectation value calculation introduced in Section II-C.

**A. Implicit Tensor Network Refactorization**

Randomized SVD is an algorithm that approximates truncated SVD of rank $r$ of an $m$-by-$n$ matrix with time complexity $O(mnr)$ [48] by the use of orthogonal iteration. This algorithm has been directly applied to replace the full SVD algorithm to apply operators on MPS [49], which reduces the cost by one order of the physical dimension. Here we apply this idea more generally to einsumsvd, resulting in asymptotic reductions in cost for one-layer and two-layer BMPS contraction.

**Algorithm 4 Randomized SVD**

**Input:** an operator $A : \mathbb{C}^{n_1 \times \cdots \times n_t} \mapsto \mathbb{C}^{m_1 \times \cdots \times m_s}$, rank $r$, number of iteration $k$

**Output:** an approximated truncated SVD of the input operator tensor operator $U \in \mathbb{C}^{m_1 \times \cdots \times m_s \times r}$, $C^r \mapsto \mathbb{C}^{m_1 \times \cdots \times m_s}$, diagonal matrix $\Sigma \in \mathbb{C}^{r \times r}$, tensor operator $V \in \mathbb{C}^{r \times n_1 \times \cdots \times n_t}$: $\{Q : C^r \mapsto \mathbb{C}^{n_1 \times \cdots \times n_t}\}$

1: $Q \leftarrow$ draw a random tensor from $[-1, 1]^{n_1 \times \cdots \times n_t}$
2: $P \leftarrow$ orthogonalize$(AQ)$ $\{P : C^r \mapsto \mathbb{C}^{m_1 \times \cdots \times m_s}\}$
3: for $i = 1, 2, \ldots, k$ do
4: $Q \leftarrow$ orthogonalize$(A^*P)$ $\{Q : C^r \mapsto \mathbb{C}^{n_1 \times \cdots \times n_t}\}$
5: $P \leftarrow$ orthogonalize$(AQ)$ $\{P : C^r \mapsto \mathbb{C}^{m_1 \times \cdots \times m_s}\}$
6: end for
7: $U, \Sigma, V \leftarrow$ SVD$(P^*A)$ $\{P^*A : \mathbb{C}^{n_1 \times \cdots \times n_t} \mapsto C^r\}$
8: $U \leftarrow PU$
9: return $U, \Sigma, V$

As shown in Algorithm 4, randomized SVD does not require an explicit form of the operator $A$ but only needs to know how to apply $A$ and $A^*$. Thus, the operator $A$ could be given implicitly as an uncontracted tensor network. Depending on the structure of the tensor network $A$, it is possible to perform $AQ$ and $A^*P$ more efficiently and with less memory than by explicitly forming $A$ and applying it.
A similar implicit method has been specifically applied to the TRG algorithm [50]. Here, we apply this method to the \textit{einsumsvd} step of the BMPS contraction algorithm (Algorithm 3). With the best choice of contraction order, the use of implicit methods in BMPS yields an asymptotic reduction in cost and, in some cases, memory footprint, compared with a naive implementation of \textit{einsumsvd}. We refer to this approach as \textit{implicit} BMPS (IBMPS) contraction. Two-layer BMPS can achieve greater benefit with the use of implicit randomized SVD, as it naturally maintains tensors in implicit forms. We compare the asymptotic costs for BMPS, IBMPS, and two-layer IBMPS in Table II.

### B. Intermediate Caching for PEPS Expectation Values

To compute expectation values of PEPS efficiently, we develop a caching strategy for consecutive executions of BMPS (Algorithm 2) when evaluating Equation (5). We observe that these contractions share common intermediates due to the locality of $H_j$, and our approach reuses partial contractions of rows (or columns) of PEPS to compute $\langle \psi | H_j | \psi \rangle$.

For example, consider a $4 \times 4$ PEPS and two local operators $H_1$ and $H_2$ both acting on the third row of the PEPS, as shown in Figure 6. We cache the two boundary MPS that represent the partial contraction results of the first two rows and last row to accelerate the calculation of $\langle \psi | H_1 | \psi \rangle$ and $\langle \psi | H_2 | \psi \rangle$.

Figure 6: Shared intermediates for expectation values of two local operators on a $4 \times 4$ PEPS.

When applied to an $n \times n$ PEPS, this caching strategy requires two full two-layer PEPS contractions to generate all the cached intermediates, and then each local-operator expectation value is calculated with a $3 \times n$ PEPS contraction. This strategy can be easily extended to compute a collection of expectation values, e.g., to compute gradients of parameters of local operators (as used in the Adapt-VQE method [29]). In Section VI, we demonstrate the speed-up achieved by this strategy (see Figure 9).

Besides evaluating Equation (5) with intermediate caching, $\langle \psi | H | \psi \rangle$ can be calculated in an alternative approach. Using the Trotter-Suzuki decomposition, $e^{\tau H} = \prod_{j=1}^{N} e^{e^{\tau H_j}H_j}$, as in ITE, and the Taylor expansion, $e^{\tau H} = I + \tau H + O(\tau^2)$, one can approximate

$\langle \psi | H | \psi \rangle = \frac{1}{\tau} \left( \langle \psi | \prod_{j=1}^{N} e^{e^{\tau H_j}H_j} | \psi \rangle - \langle \psi | \psi \rangle \right) + O(\tau).$  \hspace{1cm} (6)

In comparison with the previous method, Equation (6) requires one two-layer PEPS contraction as opposed to two, but application of an additional ITE step would grow the bond dimension of $|\psi\rangle = \prod_{j=1}^{N} e^{H_j} |\psi\rangle$ or require approximation.

### V. Koala: High-Performance PEPS Simulation

We implement the algorithms that are discussed above within an open-source Python library called “Koala”, which is available at https://github.com/cyclops-community/koala.

#### A. Interface and Features

Koala provides explicit primitives for constructing PEPS networks, applying operators, and computing expectation values. Tensor library backends, as well as different algorithms for applying operators and contracting PEPS, can be specified as arguments to these primitive routines. An example code for constructing a PEPS, applying operators, and computing an expectation value is below.

```python
from koala import peps, Observable
from koala.peps import QRUpdate, BMPS
from tensorbackends.interface import ImplicitRandomizedSVD

# Create a 2-by-3 PEPS in distributed memory
qstate = peps.computational_zeros(
    nrow=2, ncol=3, backend="ctf"
)

# Apply one-site and two-site operators with QR-SVD
Y = qstate.backend.astensor([...]).reshape(2, 2)
CX = qstate.backend.astensor([...]).reshape(2, 2, 2, 2)
qstate.apply_operator(Y, [1])
qstate.apply_operator(CX, [1, 4], QRUpdate(rank=2))

# Calculate the expectation value with IBMPS
H = Observable.ZZ(3, 4) + 0.2 * Observable.X(1)
result = qstate.expectation(
    H, use_cache=True,
    contract_option=BMPS(ImplicitRandomizedSVD(rank=4))
)
```

### B. Distributed Tensor Parallelization

Koala supports both sequential and parallel execution. This portability is achieved by an abstraction layer of tensor data types and operations that enables different tensor libraries to be used with little change in the source code. Currently, the tensor...
computation. We address this bottleneck within implicit einsumsvd (Algorithm 4) and QR-SVD (Algorithm 1), by performing orthogonalization via eigendecomposition of a Gram matrix formed by tensor contraction. For the QR factorizations done in QR-SVD, forming the Gram matrix requires a reshape of a small tensor. For orthogonalization in einsumsvd, no reshape of the Gram matrix is necessary. In both cases, this matrix is small, so the latency overhead of using distributed memory is alleviated by performing computations on the Gram matrix sequentially. Algorithm 5 describes this general approach. When Algorithm 5 is applied to QR-SVD, we can also perform einsumsvd for operator application sequentially by leveraging the fact that the operator \( G \) and the \( R \) factors are all small enough to fit into local memory.

VI. RESULTS

We benchmark our implementation of the PEPS algorithm variants that are discussed above. The overall accuracy of PEPS-based simulation for two example applications is also studied to demonstrate how accuracy is affected by the bond dimension. Parallel experiments are all done on the Stampede2 supercomputer using 64 cores per node for Cyclops and 32 MKL [52] threads on one node for NumPy. The number of processes per node (PPN) is always chosen as 64 unless otherwise specified.

A. PEPS Evolution Benchmark

Figure 7 studies the performance of applying one layer of TEBD operators on PEPS of two different sizes: \( 8 \times 8 \) and \( 15 \times 15 \). For the smaller system size, which fits into memory on 1 node, we compare the performance of NumPy and Cyclops, as shown in Figure 7a. We observe that while NumPy outperforms Cyclops for small bond dimensions, their performance becomes similar as bond dimension grows. We also compare three different algorithms for a \( 15 \times 15 \) PEPS on 16 nodes using Cyclops in Figure 7b, which shows that reshape-avoiding orthogonalization (Algorithm 5) can accelerate the parallel execution of QR-SVD (Algorithm 1) by factors of up to 3.7X.

B. PEPS Contraction Benchmark

Figure 8a shows the performance of contracting an \( 8 \times 8 \) PEPS on a single KNL node of Stampede2 using BMPS, IBMPS, two-layer IBMPS, and an exact algorithm [12] with NumPy and Cyclops as backends. Since contracting the inner product between two identical PEPS limits the selection of bond dimensions, we directly generate a PEPS without physical indices to obtain more data points. However, two-layer IBMPS is only applicable for the inner product, so we include fewer data points.

The contraction bond dimension is set equal to the initial bond dimension, which varies from 2 to 64. The trade-off between backends is similar to the PEPS evolution benchmark in that Cyclops is more scalable despite NumPy being faster for small bond dimensions. Further, we show that IBMPS and two-layer IBMPS not only have an advantage on asymptotic computational complexity, but are also more memory-efficient compared to BMPS. In practice, we observe that only IBMPS and two-layer IBMPS are able to contract an \( 8 \times 8 \) PEPS with initial and contraction bond dimension both equal to 64 on a single node. Moreover, we test the highest achievable
Figure 7: Running time versus bond dimension for one-layer TEBD on PEPS. Local-gram-qr denotes the variant of QR-SVD where orthogonalization is done locally; local-gram-qr-svd denotes the variant where both orthogonalization and einsumsvd are done locally.

Figure 8: Running time for fully contracting a PEPS as the bond dimension grows.

Figure 9: Running time of the expectation value calculation with and without caching for a square PEPS with bond dimension 4 on 1 node.
been proven to satisfy both average-case hardness and anti-concentration property [25], [55]–[57]. It is especially challenging to simulate RQCs using tensor networks, since the states created by RQCs are strongly entangled and very sensitive to approximations. By applying our algorithms to this problem, we show that approximate contraction is effective even for one of the hardest simulations.

Following the construction procedure proposed in [54], we apply iSWAP gates for all pairs of neighboring sites every four layers, increasing the bond dimension by a factor of 4. For the RQC benchmarking, we use $4 \times 4$, $5 \times 5$, $6 \times 6$, and $7 \times 7$ PEPS generated by 8 layers of RQC with exact evolution, which has an initial bond dimension of 16. Then, BMPS and IBMPS with varying contraction bond dimension are used to compute one amplitude of the circuit. The result is compared to the exact contraction algorithm to determine the relative error, as shown in Figure 10. We show that the use of implicit randomized SVD in IBMPS does not incur additional error as compared to the naive SVD. We also observe that after increasing the contraction bond dimension above a certain threshold, the relative error quickly drops to near machine epsilon. This threshold is positively correlated to the PEPS size and the initial bond dimension. For $6 \times 6$ and $7 \times 7$ PEPS, we achieved one digit of accuracy with a contraction bond dimension of 256 on only 64 nodes with 7808 gigabytes of memory in total.

C. Parallel Scaling

We study the parallel scaling of applying one layer of TEBD operators (PEPS evolution) and IBMPS (PEPS contraction) in Figure 11 and Figure 12. In the strong scaling analysis shown in Figure 11, we test a smaller problem that occupies most of the memory on 16 nodes for both PEPS evolution and contraction. For better performance with Cyclops, we choose to use PPN=16 for PEPS evolution on 128/256 nodes ($2^{13}/2^{14}$ cores) and PEPS contraction on 256 nodes ($2^{14}$ cores). When executing on a single node with multiple cores (less than $2^6$ cores), the running time roughly halves when number of cores doubles, and good strong scaling efficiency is achieved up until all cores of the node are utilized.

When the benchmarks are executed on the smaller PEPS with multiple nodes, we obtain a speed-up of 4.3X on 16 nodes ($2^{10}$ cores) for PEPS evolution and a speed-up of 13.9X on 64 nodes for PEPS contraction with respect to a single node ($2^6$ cores), after which the performance deteriorates. We observe that the fraction time in local matrix multiplication (GEMM), in the smaller PEPS contraction experiments, is 62% on 1 node, and 16% on 16 nodes. For the larger problem starting from 16 nodes, we observe that initially the performance scales well but for PEPS evolution, the performance deteriorates after 128 nodes, and for PEPS contraction, the performance roughly decreases by one-half on 256 nodes. PEPS evolution has a cost that is only slightly superlinear in the size of the tensor sites, so this kernel tends to be communication-bound.

In the weak scaling analysis shown in Figure 12, we focus on increasing the bond dimension of PEPS while keeping the memory usage per node constant. This case better reflects the practical usage of large PEPS simulations, where the major bottleneck is memory and not execution time. As shown in the figure, we observe sustained weak scaling for the full PEPS evolution and contraction benchmark up to 64 nodes ($2^{12}$ cores). Similarly, the time spent in GEMM is always around 60-70% for PEPS contraction. Overall, our approach shows good weak scalability for dealing with large bond dimensions that are prohibitive in cost or infeasible in memory footprint for typical single-node architectures.

D. Applications

We apply the PEPS algorithms to imaginary time evolution and variational quantum eigensolver simulation to solve for the ground states of quantum systems.
Figure 12: Weak scaling for PEPS evolution (applying one layer of TEBD operators) and PEPS contraction (IBMPS for PEPS with no physical indices) with evolution bond dim. \( r \) and contraction bond dim. \( m \):
\( r = 70, 83, 98, 117, 140, 166, 197, 235, 280, \)
\( m = 80, 95, 113, 134, 160, 190, 226, 269, 320. \)
(missing data points will be added soon)

1) Imaginary Time Evolution: We simulate the spin-1/2 J1-J2 Heisenberg Model [58] with the Hamiltonian defined as the operator,

\[
H = \sum_{\langle i,j \rangle} (J^x_i X_i X_j + J^y_i Y_i Y_j + J^z_i Z_i Z_j) + \sum_{\langle i,j \rangle} (J^x_j X_j X_i + J^y_j Y_j Y_i + J^z_j Z_j Z_i) + \sum_i (h^x_i X_i + h^y_i Y_i + h^z_i Z_i),
\]

where \( X, Y, \) and \( Z \) are Pauli operators and the indices/ subscripts appearing in each Pauli operator indicate the site the operators act on. The notation \( \sum_{\langle i,j \rangle} \) denotes a sum over pairs of sites \( (i \) and \( j) \) that are adjacent on a 2D lattice, while \( \sum_{\{i,j\}} \) is the same for pairs of sites that are diagonally adjacent (e.g., sites at lattice points \( i = (k,l) \) and \( j = (k+1,l-1) \)). Further, \( J_n \) defines the coupling constants between neighbors, and \( h \) represents the strength of a transverse magnetic field along a particular axis.

We perform ITE steps as described in Section II-D1 on PEPS with evolution bond dimension \( r \) and contract the resulting PEPS using IBMPS with contraction bond dimension \( m \). Figure 13 shows the iterations of PEPS ITE for small bond dimensions and the result of PEPS ITE after 150 steps as the bond dimension grows for a 4 \( \times \) 4 J1-J2 model. The energy of PEPS calculated by IBMPS converges to the ground state calculated by the state vector simulation after 1000 ITE steps when we increase the bond dimension. Also, we compare the choice of contraction bond dimension \( m = r^2 \) and \( m = r \) and observe that for this model, their accuracy is similar while the latter requires much less computation.

2) Variational Quantum Eigensolver Simulation: We benchmark VQE with the transverse-field Ising (TFI) Model with the Hamiltonian defined as

\[
H = \sum_{\langle i,j \rangle} J^z_i Z_i Z_j + \sum_i h^x_i X_i, \tag{8}
\]

which is a special case of Equation (7) when \( J^x_i = J^y_i = J^z_i = J^x_j = J^y_j = h^x_i = h^y_i = h^z_i = 0 \). We set \( J^x_i = -1 \) and \( h^x_i = -3.5 \) in Equation (8) to simulate the Ferromagnetic TFI Model [59].

We utilize the sequential least squares programming (SLSQP) optimization algorithm [60], provided via the scipy.optimize.minimize function in Python. Our variational ansatz, which is passed to the optimizer, applies a parameterized quantum circuit composed of repeated layers to construct the state prior to calculating the expectation value of the energy. Each circuit layer consists of single rotation gates \( R_x(\theta) \) applied to each qubit. The rotation gates are followed by a set of CNOT gates, which are applied to each nearest neighbor pair.

Each VQE iteration involves the contraction of PEPS in order to calculate the expectation value, which creates a substantial bottleneck in terms of running time. Since each optimizer iteration computes multiple objective function evaluations, VQE itself involves running hundreds or even thousands of function evaluations prior to convergence. Consequently, we focus our accuracy experiments on a small number of qubits.

As can be seen in Figure 14, for a 3 \( \times \) 3 qubit system increasing the bond dimension used in the PEPS simulation improves the lowest energy per site that each system reaches. The energy obtained is \(-3.50000, -2.35467, -3.54174, \) and \(-3.54640 \) for a maximum bond dimension of 1, 2, 3, and 4, respectively. By comparison, the state vector system reaches a value of \(-3.57049 \), while the exact ground state energy per site is \(-3.60024 \).

VII. Related Work

While algorithms and parallel computations with PEPS are a relatively new area, there exist several works that have taken similar approaches to ours. The PEPS++ method has been proposed by He et al. as a computational method that enables parallel computation for quantum systems simulation [61]. To overcome the high order of complexity, i.e. \( O(r^{10}) \), they combine quantum Monte Carlo methods with PEPS algorithms to avoid the two-layer PEPS contraction. Instead, they perform one-layer PEPS contraction along with a sampling procedure, which reduces the asymptotic cost to \( O(r^3) \) for each sampling step. In the implementation, they use processes-level parallelism for sampling steps and thread-level parallelism for tensor computations, which differs from the distributed-memory tensor computation considered in this work.

Like our work, Guo et al. have carried out large-scale parallel quantum circuit simulations utilizing PEPS [12]. They simulate random quantum circuits exactly by applying gate operators on PEPS without truncation, and they calculate amplitudes by projecting the resulting PEPS onto the corresponding basis states. The cost of this method is associated with the
entanglement generated by the quantum circuit instead of the circuit size, but could still be exponential because it requires exact PEPS contraction. They also use Cyclops for distributed-memory tensor computations, and their results include an exact PEPS contraction. They also use Cyclops for distributed-memory parallelism, but could still be exponential because it requires entanglement generated by the quantum circuit instead of the circuit size.

A related algorithm, the higher-order tensor renormalization group (HOTRG) has also been studied in a distributed parallel setting [62], where an optimal reordering procedure for tensor contractions used in HOTRG is proposed. In comparison, our work provides a more general approach via the use of Cyclops, which automates the performance optimization within tensor contractions. We also consider a broad set of PEPS primitives as opposed to a single component of a PEPS contraction algorithm.

There are also a number of libraries available for quantum system simulation using tensor networks, such as ITensor [63], NCON [64], quimb [65], TeNPy [66], Uni10 [67], and qTorch [68]. These libraries implement useful algorithms in the field with good sequential, threaded, or GPU performance. Our library, Koala, contributes to this community with the use of distributed-memory parallelism.

VIII. CONCLUSION

As demonstrated by our application studies, tensor network methods can systematically improve the accuracy for approximate simulation of quantum systems and near-term quantum devices by using larger bond dimensions (i.e. larger tensors). The new algorithms and Koala library we introduced provide an efficient approach for PEPS evolution and contraction and enable the use of distributed-memory tensors. Performance studies on the Stampede2 supercomputer indicate that our approach provides scalable PEPS primitives and systematically improvable accuracy.
ACKNOWLEDGMENTS

We would like to thank Phillip Helms, Ryan Levy, Garnet Chan, and Bryan Clark for helpful conversations and insights on tensor network methods, Yuchen Pang, Annika Dugad, and Edgar Solomonik were supported via the US NSF OAC RAISE/TAQS program, award number 1839204. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. We used XSEDE to employ Stampede2 at the Texas Advanced Computing Center (TACC) through allocation TG-CCR180006.

REFERENCES

[1] R. Orús, “A practical introduction to tensor networks: Matrix product states and projected entangled pair states,” Annals of Physics, vol. 349, pp. 117 – 158, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0003491614001596

[2] G. Vidal, “Efficient classical simulation of slightly entangled quantum computations,” Phys. Rev. Lett., vol. 91, p. 147902, Oct 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.91.147902

[3] U. Schollwöck, “The density-matrix renormalization group in the age of matrix product states,” Annals of Physics, vol. 326, no. 1, pp. 96–192, Jan 2011. [Online]. Available: http://dx.doi.org/10.1016/j.aop.2010.09.012

[4] F. Verstraete and J. I. Cirac, “Renormalization algorithms for quantum-many body systems in two and higher dimensions,” arXiv preprint cond-mat/0605696, 2006.

[5] F. Verstraete, J. J. García-Ripoll, and J. I. Cirac, “Matrix product density operators: Simulation of finite-temperature and dissipative systems,” Physical Review Letters, vol. 93, no. 20, Nov 2004. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.93.207204

[6] G. Vidal, “Class of quantum many-body states that can be efficiently simulated,” Phys. Rev. Lett., vol. 101, p. 110501, Sep 2008. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.101.110501

[7] Y.-Y. Shi, L.-M. Duan, and G. Vidal, “Classical simulation of quantum many-body systems with a tree tensor network,” Physical review a, vol. 74, no. 2, p. 022320, 2006.

[8] V. Murg, F. Verstraete, and J. I. Cirac, “Variational study of hard-core bosons in a two-dimensional optical lattice using projected entangled pair states,” Phys. Rev. A, vol. 75, p. 033605, Mar 2007. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.75.033605

[9] J. Jordan, R. Orús, G. Vidal, F. Verstraete, and J. I. Cirac, “Classical simulation of infinite-size quantum lattice systems in two spatial dimensions,” Phys. Rev. Lett., vol. 101, p. 250602, Dec 2008. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.101.250602

[10] J. Preskill, “Quantum computing in the NISQ era and beyond,” Phys. Rev. X, vol. 9, no. 2, p. 021002, 2019. [Online]. Available: http://dx.doi.org/10.1103/PhysRevX.9.021002

[11] J. Preskill, “Quantum computing in the NISQ era and beyond,” Phys. Rev. X, vol. 9, no. 2, p. 021002, 2019. [Online]. Available: http://dx.doi.org/10.1103/PhysRevX.9.021002

[12] N. Schuch, M. M. Wolf, F. Verstraete, and J. I. Cirac, “Computational complexity of projected entangled pair states,” Phys. Rev. Lett., vol. 98, p. 140506, 2007. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.98.140506

[13] T. E. Oliphant, A guide to NumPy. Trelgol Publishing USA, 2006, vol. 1.

[14] R. Nishino and S. H. C. Loomis, “CuPy: A NumPy-compatible library for NVIDIA GPU calculations,” 31st conference on neural information processing systems, p. 151, 2017.

[15] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, “A massively parallel tensor contraction framework for coupled-cluster computations,” Journal of Parallel and Distributed Computing, vol. 74, no. 12. pp. 3176–3190, 2014.
