

# QuOp\_MPI: a framework for parallel simulation of quantum variational algorithms.

Edric Matwiejew<sup>a,\*</sup>, Jingbo B. Wang<sup>a</sup>

<sup>a</sup>*Department of Physics, The University of Western Australia, Perth, Australia*

## Abstract

QuOp\_MPI is a Python package designed for parallel simulation of quantum variational algorithms. It presents an object-orientated approach to quantum variational algorithm design and utilises MPI-parallelised sparse-matrix exponentiation, the fast Fourier transform and parallel gradient evaluation to achieve the highly efficient simulation of the fundamental unitary dynamics on massively parallel systems. In this article, we introduce QuOp\_MPI and explore its application to the simulation of quantum algorithms designed to solve combinatorial optimisation algorithms, including the Quantum Approximation Optimisation Algorithm, the Quantum Alternating Operator Ansatz, and the Quantum Walk-assisted Optimisation Algorithm.

**Keywords:** quantum algorithms, quantum walks, combinatorial optimisation, parallel simulation, software packages

## 1. Introduction

With the first generation of quantum computers currently in operation, the start of a new computing paradigm appears just around the corner [1, 2]. Contributing to this optimism has been the development of algorithms that exploit a combination of classical and quantum hardware to solve optimisation problems [3–7]. Compared to many exclusively quantum algorithms, these quantum variational algorithms (QVAs) require minimal quantum operations and are inherently resilient to system noise [2, 8]. For these reasons, QVAs are strong contenders for early practical applications of quantum computing in the Noisy Intermediate Scale Quantum (NISQ) era [9]. Examples of such QVAs include the Quantum Approximate Optimisation Algorithm (QAOA) [3, 7], the Quantum Alternating Operator Ansatz (QAOAz) [4], and the Quantum Walk-assisted Optimisation Algorithm (QWOA) [5, 6], which have been designed to find optimal, or near-optimal, solutions to combinatorial optimisation problems.

Combinatorial optimisation problems (COPs) —the task of finding the best combination of items from a set—are nearly ubiquitous [10]. They are present in fields such as logistics [11], drug design [12], software compilation [13] and finance [14, 15]. These problems are difficult to solve classically due to a lack of identifiable structure and exponential growth of the solution space. Quantum variational algorithms can provide a polynomial speedup compared to a classical random search [5, 6] which is an attractive prospect in problems with great humanitarian or financial consequence.

To solve COPs, QVAs exploit quantum superposition to act on the entire problem-specific solution space in quantum parallel. They apply a sequence of alternating unitaries; the first encodes the solution ‘qualities’ into the phase of superposed

quantum states, and the second ‘mixes’ probability amplitude between the states. The phase-shift and mixing unitaries are parameterised by scalar variables adjusted iteratively by a classical optimiser that minimises the average measured solution quality. By encoding optimal solutions as minima in the solution space, lowering the average solution quality corresponds to amplifying probability density at quantum states associated with optimal or near-optimal solutions.

Classical numerical simulation plays a vital role in the development of QVAs. Through simulation of the idealised quantum dynamics, researchers can study QVAs independently of implementation-specific hardware constraints and at scales that still exceed the functional limitations of current quantum hardware [16]. To assist with these efforts, we have developed QuOp\_MPI (**Q**uantum **O**ptimisation with **M**PI) [17], which provides a flexible framework for the design and classical simulation of QVAs.

There is currently significant interest in developing tools for the simulations of QVAs in a high-performance computing setting. Recent examples include TensorFlow Quantum, a software framework for quantum machine learning [18], and the Jülich universal quantum computer simulator [16]. Both utilise GPU acceleration, with the latter being targeted at distributed GPU clusters. Also of note is the XACC framework and qFlex, which utilise a tensor network approach to quantum simulation [19, 20]. These packages take a quantum-gate based approach to algorithm simulation and can simulate QVAs with a large number of qubits (e.g. more than 50) given a quantum circuit structure that is parsimonious to their underlying simulation methods. QuOp\_MPI presents a distinct option for QVA simulation in that it does not take a gate-based or approximative approach; instead, it focuses on the simulation of the fundamental unitary dynamics across the complete quantum state-space. It also provides the first tool for ready-made simulation of the Quantum Walk-assisted Optimisation Algorithm.

The structure of the paper is as follows. In Section 2 we

\*Corresponding author.

E-mail address: Edric.Matwiejew@research.uwa.edu.au

define the generalised QVA, introduce the QAOA, QAOAz and QWOA, and specify the problem of combinatorial optimisation. In Sections 3 and 4 we discuss the data structures, algorithms and parallelisation schemes leveraged by QuOp\_MPI. This is followed by an overview of the package structure, functionality and workflow types in Section 5. In Section 6 we provide usage examples drawn from literature followed by a discussion of the package's performance in Section 7. Finally, concluding statements are presented in Section 8.

## 2. Theoretical Background

### 2.1. Quantum Variational Algorithms

For a quantum system of size  $N = 2^n$ , where integer  $n$  is a number of qubits with basis states  $\left\{ |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$ , QuOp\_MPI defines a generalised QVA as

$$|\theta\rangle = \left( \prod_{i=1}^D \hat{U}(\theta_i) \right) |\psi_0\rangle, \quad (1)$$

where  $|\psi_0\rangle \in \mathbb{C}^N$  is an initial quantum state with basis states  $\{|i\rangle\}_{i=0, \dots, N-1}$ ,  $\hat{U} \in \mathbb{C}^{N \times N}$  is the ansatz<sup>1</sup> unitary, integer  $D \geq 0$  specifies the number of applications of  $\hat{U}$  to  $|\psi_0\rangle$  (the 'depth') and  $\theta = \{\theta_i \in \mathbb{R}\}$  is an ordered set of classically tunable values that parameterise  $\hat{U}$ . The ansatz unitary  $\hat{U}$  and initial quantum state  $|\psi_0\rangle$  together define a specific QVA.

A Quantum Variational Algorithm is executed by repeatedly preparing  $|\theta\rangle$  and measuring the expectation value

$$f(\theta) = \langle \theta | \hat{Q} | \theta \rangle, \quad (2)$$

where  $\hat{Q} \in \mathbb{R}^{N \times N}$  is a diagonal matrix operator with entries  $\text{diag}(\hat{Q}) = q_i$  that specify the 'quality' associated with quantum state  $|i\rangle$ . The variational parameters  $\theta$  are updated using a classical optimiser with the objective being minimisation of  $f$ .

The ansatz operator  $\hat{U}$  specifies a sequence of alternating unitaries. This can include phase-shifts

$$\hat{U}_{\text{phase}}(\gamma) = \exp(-i\gamma\hat{O}), \quad (3)$$

where  $\hat{O} = \sum_{i=0}^{N-1} o_i |i\rangle\langle i|$  is a diagonal phase-shift matrix operator,  $\gamma \in \theta$  and  $\hat{U}_{\text{phase}}$  applies a phase-shift proportional to  $o_i$ , as well as mixing-unitaries

$$\hat{U}_{\text{mix}}(t) = \exp(-it\hat{W}), \quad (4)$$

where  $t \in \theta$  is non-negative and  $\hat{W} = \sum_{i,j=0}^{n-1} w_{ij} |j\rangle\langle i|$  is a mixing matrix operator in which non-diagonal entries specify coupling between states  $|i\rangle$  and  $|j\rangle$ . Mixing-unitaries  $\hat{U}_{\text{mix}}$  drive the transfer of probability amplitude between quantum states, during which encoded phase differences contribute to constructive and destructive interference.

<sup>1</sup>Originating from a German word that refers to the starting thought of a process. In mathematics, an ansatz is an educated guess or assumption made to help solve a problem.

Phase-shift operators  $\hat{O}$  and mixing operators  $\hat{W}$  may also be parameterised by  $\theta$ . As these operators are time-independent Hamiltonians of the time-evolution operator, changes to the corresponding  $\theta_i$  alter the element-wise magnitudes or structure of the matrix exponent before preparation of  $|\theta\rangle$ .

Typically, the ansatz unitary  $\hat{U}$  is applied  $D$  times to  $|\psi_0\rangle$  with each repetition parameterised by subset  $\theta \subseteq \theta$ . Doing so increases the potential for constructive and destructive inference to concentrate probability amplitude at high-quality solutions; at the expense of classical optimisation over a larger parameter space and a deeper quantum circuit. In practice, a QVA must balance the improved convergence afforded by increases to  $D$  against the ability of the quantum hardware to maintain coherence over a longer sequence of quantum operations.

Sections 2.3 and 2.4 introduce four distinct QVAs for solving constrained and unconstrained COPs. We summarise here the following notational conventions for a given QVA:

- $n$ : the number of qubits.
- $|\psi_0\rangle$ : the initial quantum state vector.
- $\hat{U}$ : a sequence of phase-shift and mixing operators.
- $|\psi\rangle$ :  $|\psi_0\rangle$  after  $D \geq 0$  applications of  $\hat{U}$ .
- $|\theta\rangle$ :  $|\psi_0\rangle$  after  $D \geq 1$  applications of  $\hat{U}$ .
- $\theta$ : classically tunable variables parameterising  $\hat{U}$  with starting values  $\theta_0$  and optimised values  $\theta_f$ .
- $f$ : the ansatz objective function.

### 2.2. Combinatorial Optimisation with QVAs

Combinatorial optimisation problems seek optimal solutions  $\bar{s}$  of the form,

$$\bar{s} = \left\{ s \mid C(s) \in \min \{C(s) \mid s \in \mathcal{S}'\} \right\}, \quad (5)$$

where the problem cost-function  $C(s)$  maps a solution  $s$  from an ordered set of problem solutions  $\mathcal{S} = \{s_i\}$  to  $\mathbb{R}$ ,  $s$  is a  $k$ -permutation of discrete elements from a finite set  $\zeta$  and

$$\mathcal{S}' = \{s \mid s \in \mathcal{X}\} \quad (6)$$

is the problem-specific valid solution space where

$$\mathcal{X} = \bigcup_i \left\{ s \mid \chi_i(s) = a_i \right\} \quad (7)$$

denotes any constraints on  $\bar{s}$  and  $\mathbf{a} = \{a_i\}$  defines the constraints.

Problems of this type are often difficult to solve as  $\mathcal{S}$  grows factorially with  $|\zeta|$  and, in general, lacks identifiable structure. For this reason, heuristic and metaheuristic algorithms are often used to find solutions that satisfy the relaxed condition of  $C(\bar{s})$  being a 'sufficiently low' local minimum.

To apply a quantum variational algorithm to a given combinatorial optimisation problem, an injective map is defined between  $\mathcal{S}$  and  $\mathcal{H}$  with the cost-function values forming the diagonal of the quality operator  $\text{diag}(\hat{Q}) = C(s_i)$ . For example,

a problem with four solutions,  $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$ , maps to a two-qubit system as

$$\begin{aligned} |00\rangle &= |0\rangle \rightarrow |s_0\rangle \\ |01\rangle &= |1\rangle \rightarrow |s_1\rangle \\ |10\rangle &= |2\rangle \rightarrow |s_2\rangle \\ |11\rangle &= |3\rangle \rightarrow |s_3\rangle, \end{aligned} \quad (8)$$

where  $\text{diag}(\hat{Q}) = (C(s_0), C(s_1), C(s_2), C(s_3))$ .

For a combinatorial optimisation problem to be efficiently solvable by a QVA, it must satisfy three conditions:

1. The number of solutions  $|\mathcal{S}|$  must be efficiently computable in order to establish a bound on the size of the required Hilbert space  $\mathcal{H}$ .
2. For all solutions  $s$ ,  $C(s)$  must be computable in polynomial time.
3. For all solutions  $s$ ,  $C(s)$  must be polynomially bounded with respect to  $|\mathcal{S}|$ .

Conditions one and two ensure that the objective function (Equation (2)) is efficiently computable as classical computation of  $C(s)$  is required to compute  $f$  and boundedness in  $C(s)$  ensures that the number of measurements required to accurately compute  $f$  does not grow exponentially with  $|\mathcal{S}'|$  [21]. These conditions constrain the application of QVAs to polynomially bounded (PB) COPs in the non-deterministic polynomial-time optimisation problem (NPO) complexity class (together denoted as NPO-PB) [21].

### 2.3. Unconstrained Optimisation

For the case of unconstrained optimisation, the valid solution space  $\mathcal{S}'$  is equivalent to  $\mathcal{S}$ . For these COPs a quantum encoding of  $C(s)$  is equivalent to the bijective map  $\mathcal{S} \rightarrow \mathcal{H}$ .

#### 2.3.1. QAOA

The Quantum Approximate Optimisation Algorithm is comprised of two alternating unitaries. Firstly the phase-shift-unitary

$$\hat{U}_Q(\gamma_i) = \exp(-i\gamma_i \hat{Q}) \quad (9)$$

and, secondly, the mixing operator

$$\hat{U}_X(t_i) = \exp(-it_i \hat{W}_X), \quad (10)$$

where  $\hat{W}_X = X^{\otimes N}$  and  $X$  is the Pauli- $X$  (or quantum NOT) gate. The mixing operator  $\hat{W}_X$  induces a coupling topology that is equivalent to an  $n$ -dimension hypercube graph, as shown in Figure 1.

The initial state  $|\psi_0\rangle_{\text{QAOA}}$  is prepared as an equal superposition over  $\mathcal{H}$ ,

$$|+\rangle = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle. \quad (11)$$

The final quantum state is then

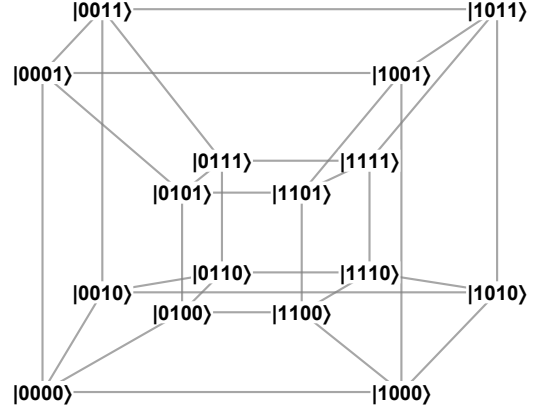


Figure 1: Coupling topology of  $W_X$  in the QAOA for  $|\mathcal{S}| = 16$  ( $n = 4$ ).

$$|\theta\rangle_{\text{QAOA}} = \left( \prod_{i=1}^D \hat{U}_X(t_i) \hat{U}_Q(\gamma_i) \right) |+\rangle, \quad (12)$$

where  $\theta = \{\gamma_i, t_i\}$  and  $|\theta| = 2D$  [3].

#### 2.3.2. Extended-QAOA

A variation of the QAOA, ‘extended-QAOA’ (ex-QAOA), utilises a sequence of phase-shift unitaries,

$$\hat{U}_{\text{Qext}}(\gamma_i) = \prod_{j=1}^{|\Sigma|} \exp(-i(\gamma_i)_j \Sigma_j), \quad (13)$$

where  $\Sigma_j$  are non-identity terms in a Pauli-gate decomposition of  $\hat{Q}$  and  $|\Sigma|$  is the number of non-identity terms [7]. This increases the number of variational parameters to  $|\theta| = (1 + |\Sigma|)D$  with the intent of achieving a higher degree of convergence to optimal solutions at a lower circuit depth.

The final state of ex-QAOA is

$$|\theta\rangle_{\text{ex-QAOA}} = \left( \prod_{i=1}^D \hat{U}_X(t_i) \hat{U}_{\text{Qext}}(\gamma_{i,:}) \right) |+\rangle, \quad (14)$$

where  $|+\rangle$  and  $\hat{U}_X$  are defined as in Equations (10) and (11) and  $\theta = \{\gamma_{ij}, t_i\}$ .

### 2.4. Constrained Optimisation

Constrained optimisation problems seek valid solutions  $s'$  from a subset of  $\mathcal{S}$  as defined by constraints  $\chi$ . Encoding of the solution constraints  $\chi$  is achieved by restricting the action of the mixing-unitaries  $\hat{U}_{\text{mix}}$  and initialising  $|\psi_0\rangle$  over a subspace of  $\mathcal{H}$ .

#### 2.4.1. QAOAz

The Quantum Alternating Operator Ansatz was developed to solve problems for which  $\chi$  creates a correspondence between  $\mathcal{S}'$  and quantum states of equal parity – states with the same

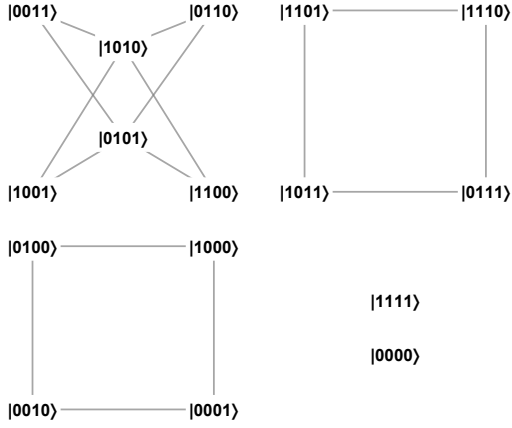


Figure 2: Coupling topology of  $\hat{W}$  for the QAOAz ( $n = 4$ ). Note that  $\mathcal{H}$  is partitioned into subgraphs of equal state parity.

number of  $|1\rangle$  states. This algorithm consists of the phase-shift-unitary defined in Equation (9), followed by a sequence of three  $\hat{U}_{\text{mix}}$  with mixing operators

$$\begin{aligned}\hat{B}_{\text{odd}} &= \sum_{a \text{ odd}}^{N-1} X_a X_{a+1} + Y_a Y_{a+1} \\ \hat{B}_{\text{even}} &= \sum_{a \text{ even}}^N X_a X_{a+1} + Y_a Y_{a+1} \\ \hat{B}_{\text{last}} &= \begin{cases} X_N X_1 + Y_N Y_1, & \text{odd} \\ I, & \text{Neven,} \end{cases}\end{aligned}\quad (15)$$

which together form the parity-conserving mixing operator

$$\hat{U}_{\text{parity}}(t) = e^{-it\hat{B}_{\text{last}}} e^{-it\hat{B}_{\text{even}}} e^{-it\hat{B}_{\text{odd}}} \quad (16)$$

that mixes probability amplitude between subgraphs of equal parity as illustrated in Figure 2.

By initialising  $|\psi_0\rangle_{\text{QAOAz}}$  in a quantum state that satisfies the parity constraint, probability amplitude is constrained to  $\mathcal{S}'$ .

The state evolution of the QAOAz is

$$|\theta\rangle_{\text{QAOAz}} = \left( \prod_{i=1}^D \hat{U}_{\text{parity}}(t_i) \hat{U}_Q(\gamma_i) \right) |\psi_0\rangle_{\text{QAOAz}}, \quad (17)$$

where  $|\psi_0\rangle_{\text{QAOAz}}$  is an initial state satisfying the parity constraint [4].

#### 2.4.2. QWOA

The Quantum Walk-assisted Optimisation Algorithm implements  $\chi$  given the existence of an efficient indexing algorithm for all  $s \in \mathcal{S}'$ . Under this condition, the QWOA implements an indexing unitary

$$U_{\#}^{\dagger} |i\rangle = \begin{cases} |\text{id}_{\chi}(i)\rangle, & |i\rangle \in \mathcal{S}' \\ |i\rangle, & \text{otherwise,} \end{cases} \quad (18)$$

where  $U_{\#}^{\dagger}$  maps states corresponding to valid solutions  $|s'\rangle$  to

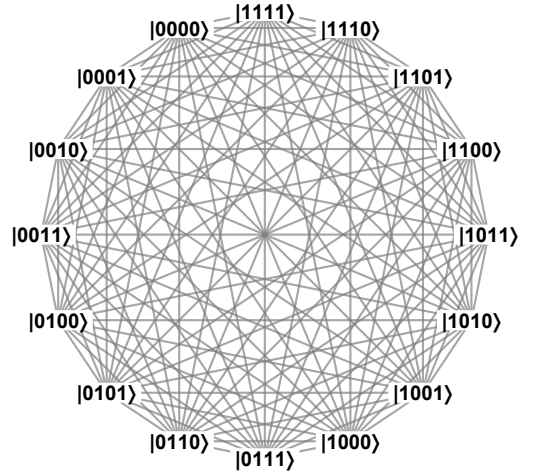


Figure 3: Coupling topology of  $\hat{W}$  for the QWOA QWOA ( $n = 4$ ).

indexed states  $|\text{id}_{\chi}(i)\rangle$ . By preparing  $|\psi_0\rangle_{\text{QWOA}}$  as an equal superposition over  $|\text{id}_{\chi}(i)\rangle$

$$|\psi_0\rangle_{\text{QWOA}} = \frac{1}{\sqrt{|\mathcal{S}'|}} \sum_{i \in \mathcal{S}'} |i\rangle, \quad (19)$$

probability amplitude is restricted to the subspace of indexed states.

The indexing unitary  $U_{\#}^{\dagger}$  and its conjugate unindexing unitary  $\hat{U}_{\#}$  occur either side of a mixing-unitary that acts on  $|\text{id}_{\chi}(i)\rangle$ :

$$\hat{U}_{\text{index}}(t) = \hat{U}_{\#} \exp(-it\hat{W}_{\text{QWOA}}) \hat{U}_{\#}^{\dagger} \quad (20)$$

Where efficiency in the implementation of  $U_{\#}^{\dagger}$  dictates that  $\hat{W}_{\text{QWOA}}$  is circulant. Most commonly,  $\hat{W}_{\text{QWOA}}$  is chosen to be the adjacency matrix of the complete graph as it produces a maximal and unbiased coupling over  $|\mathcal{S}'\rangle$  (see Figure 3).

The state evolution of the QWOA is

$$|\theta\rangle_{\text{QWOA}} = \prod_{i=1}^D \hat{U}_{\text{index}}(t_i) \hat{U}_Q(\gamma_i) |\psi_0\rangle_{\text{QWOA}}, \quad (21)$$

where  $\theta = \{\gamma_i, t_i\}$  and there are  $|\theta| = 2D$  variational parameters [5, 6].

### 3. Numerical Methods

By default, QuOp\_MPI presents three approaches by which to compute the action of a phase-shift  $\hat{U}_{\text{phase}}$  or mixing-unitary  $\hat{U}_{\text{mix}}$ .

As phase-shift unitaries  $\hat{U}_{\text{phase}}$  have a diagonal exponent matrix  $\hat{O}$ , the action of a  $\hat{U}_{\text{phase}}(\gamma)$  is efficiently computed by noting that

$$\hat{U}_{\text{phase}}(\gamma) |\psi\rangle = \sum_i^{N-1} e^{-i\gamma o_i |i\rangle\langle i|} c_i |i\rangle, \quad (22)$$

where  $|\psi\rangle$  is an arbitrary quantum state with complex coefficients  $c_i$ .

For the mixing unitaries  $\hat{U}_{\text{mix}}$ , non-diagonal entries in  $\hat{W}$ , necessitate accurate computation of the action of the matrix exponential. Given a circulant  $\hat{W}$ , QuOp\_MPI takes advantage of the relationship between the eigensystem of circulant matrices and the discrete Fourier transform. The analytical solution for the eigenvalues of a circulant matrix are given by

$$\lambda_j = w_0 + w_{M-1}\omega^j + w_{M-2}\omega^{2j} + \dots + w_1\omega^{(M-1)j}, \quad (23)$$

where  $M$  is the size of the matrix,  $w_{i=0,\dots,M-1}$  defines the first row of the circulant matrix,  $\omega = \exp(\frac{2\pi i}{M})$  is a primitive  $M^{\text{th}}$  root of unity and  $j = 0, \dots, M-1$ . The corresponding eigenvectors,

$$v_j = \frac{1}{\sqrt{M}}(\omega^j, \omega^{2j}, \dots, \omega^{(M-1)j}), \quad (24)$$

then form the matrix of the discrete Fourier transform. As such, the action of a  $\hat{U}_{\text{mix}}$  with a circulant  $\hat{W}$  may be implemented as

$$\hat{U}_{\text{mix}}(t)|\psi\rangle = F^{-1}e^{it\Lambda}F|\psi\rangle, \quad (25)$$

which is carried out in QuOp\_MPI using algorithms provided by the Fastest Fourier Transform in the West (FFTW) library [22, 23]. For the case of sparse mixing operators, QuOp\_MPI utilises a variant of the scaling and squaring algorithm, adapted from an implementation previously developed by the authors [24].

The above numerical methods support a simulation workflow distinct from gate-based quantum algorithm simulation packages. For instance, efficient gate-based simulation of the complete wavefunction can be achieved by combining one and two-qubit gates to reduce the total number of required matrix multiplications [18]. Alternatively, tensor-network based approximations reduce the computational cost by disregarding long-range interactions or qubit couplings, in addition to forming an efficient decomposition of the quantum circuit as a sequence of tensor products [19].

Gate based simulation efficiency is highly dependent on the structure of the  $\hat{U}_{\text{mix}}$  and  $\hat{U}_{\text{phase}}$  matrix exponents. In general,  $\hat{U}_{\text{mix}}$  and  $\hat{U}_{\text{phase}}$  are approximated as per the quantum Hamiltonian Simulation Algorithm [25], which is based on a Trotter-Suzuki decomposition of the matrix exponential [26]. Such representations are computationally efficient given  $\hat{U}_{\text{mix}}$  with sparse matrix exponents expressed in the Pauli basis [26]. However, accurate simulation of arbitrary mixing operators is generally not possible as the computational cost of simulation is proportional to the length of the quantum circuit [26]. Efficient gate-base representations can also be hard to realise for highly entangling quantum algorithms - which offer some of the best examples of quantum advantage. For instance, one such algorithm, the Quantum Fourier Transform, offers an exponential advantage over its classical counterpart [27].

For these reasons, the numerical methods provided with QuOp\_MPI focus on providing QVA simulations to double-precision accuracy in a manner that is agnostic to any specific

implementation. Together, QuOp\_MPI, and other gate-based simulation packages provide for different avenues of investigation. The former enables research into the limiting characteristics of QVAs, and the latter supports the investigation of gate-based realisations of QVAs.

The choice of initial values for the variational parameters  $\theta$  and the accompanying classical optimisation algorithm is an active area of research [28]. By default, QuOp\_MPI uses the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [29] provided by SciPy via its `minimize` function [30] as the authors have found it to behave reliably across a wide variety of QVAs (see Section 7). Users are able to adjust the parameters and optimisation algorithms used by the `minimize` function or opt to use algorithms provided by the NLOpt package [31] through an included ‘SciPy-like’ interface [32].

#### 4. Parallelisation Schemes

Parallelisation in QuOp\_MPI is implemented using the Message Passing Interface (MPI) standard. In general terms, MPI supports a distributed-memory model of parallel computing in which concurrent instances of the same program operate within isolated memory and namespaces, communicating with each other using ‘message-passing’ directives. As opposed to shared-memory parallel frameworks, this allows for the use of large scale distributed computers (i.e. supercomputers).

A group of program copies (MPI *processes*) that are capable of MPI communication form an MPI *communicator*. Within an MPI communicator, each process is identified by a sequential *rank* ID that ranges from 0 to size - 1, where size is the total number of MPI processes in the communicator. Communicator subsets (*sub-communicators*) can be created and assigned to sub-tasks. Note that, while an MPI process is also commonly referred to as a *node*, in this work *node* refers only to a compute-node in a computational cluster. Depending on user-controlled settings, QuOp\_MPI operates on the default global MPI communicator or a variable configuration of MPI sub-communicators.

The primary QuOp\_MPI parallelisation scheme COMM- $|\theta\rangle$  is illustrated in Figure 4. The initial state  $|\psi_0\rangle$ , evolved state  $|\theta\rangle$  and observable values  $\text{diag}(\hat{Q})$  are distributed over an MPI communicator with each rank containing a partition of sequential elements. The global position of a vector partition is thus specified by two variables; the number of vector elements in the local partition

$$\text{local}_i \quad (26)$$

and the vector element index offset

$$\text{local\_i\_offset} = \sum_{m=0}^{\text{rank}-1} \text{local}_m. \quad (27)$$

Unitary evolution is carried out using MPI-parallelised subroutines that act on the local vector partitions.

At run-time, QuOp\_MPI attempts to partition the global vectors equally over each rank in COMM- $|\theta\rangle$  while satisfying any

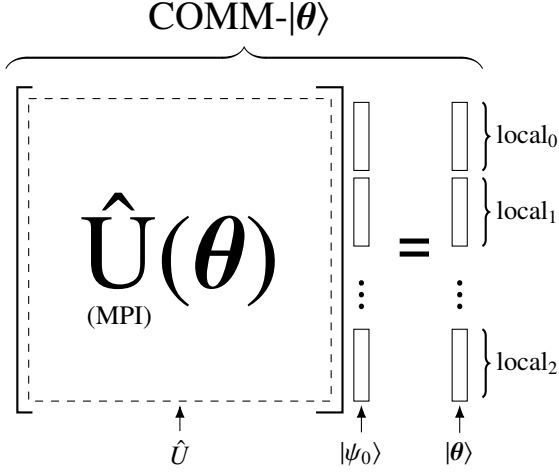


Figure 4: The partitioning of  $|\psi_0\rangle$  and  $|\theta\rangle$  over an MPI communicator of size  $= l$  where  $local_i$  denotes the number of vector elements stored at rank  $i$ . The ansatz unitary  $\hat{U}$  is implemented as a sequence of MPI parallelised subroutines that receive the distributed state vector and  $\theta_i$  as inputs and return  $|\theta\rangle$  as an identically distributed state vector.

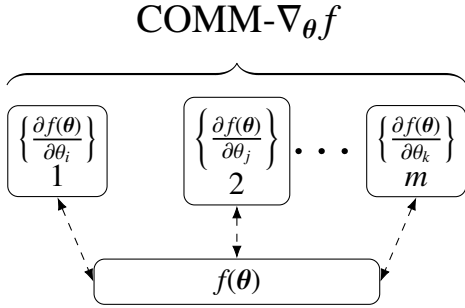


Figure 5: Parallel computation of the objective function gradient  $\nabla_{\theta} f$  where each box represents an MPI sub-communicator implementing an instance of  $COMM-|\theta\rangle$ . The objective function  $f$  is calculated by the  $COMM-|\theta\rangle$  containing the MPI process with rank = 0 in the global MPI communicator and the partial derivatives by sub-communicators 1 through to  $m$  where  $m \leq |\theta|$ .

partitioning constraints associated with external libraries such as FFTW. If a process receives zero vector elements, they are excluded from  $COMM-|\theta\rangle$  and the vector partitioning is recalculated. State evolution and calculation of  $f$  is then carried out over  $COMM-|\theta\rangle$  in parallel.

With each evaluation of  $|\theta\rangle$ ,  $f$  is sent to rank 0 of  $COMM-|\theta\rangle$  where it is received by the optimisation algorithm. The adjusted  $\theta$  are then broadcast to all nodes in  $COMM-|\theta\rangle$ , and the cycle repeats until the optimisation algorithm terminates. The distributed  $|\theta_f\rangle$  may then be written to disk using parallel HDF5 or gathered at the root rank.

In the case of optimisation algorithms that make use of gradient information, the user may choose to calculate the objective function gradient  $\nabla_{\theta} f$  in parallel. As shown in Figure 5, the global communicator is split into  $m+1$  MPI sub-communicators of which  $m$  are assigned the task of approximating the partial derivative of  $f$  (via forward or central differences) for a subset of  $\theta$ . The number of created sub-communicators depends on  $|\theta|$ , the number of compute nodes in the global communicator and

a user-definable ‘parallel’ attribute (see Table 3):

1. If  $|\theta| < \text{nodes} + 1$  and `parallel = ‘jacobian’`: create sub-communicators consisting of multiple nodes.
2. If  $|\theta| > \text{nodes} + 1$  and `parallel = ‘jacobian’`: create multiple sub-communicators within each compute node.
3. If `parallel = ‘jacobian_local’`: create one sub-communicator per compute node.

The default behaviour of QuOp\_MPI is parallelisation of  $COMM-|\theta\rangle$  exclusively (`parallel = ‘global’`). Together,  $COMM-|\theta\rangle$  and  $COMM-\nabla_{\theta} f$  allow the user to specify an MPI process configuration that is optimal for their hardware and simulation scale.

## 5. Package Overview

QuOp\_MPI is a Python module that provides an object-orientated approach to QVA design and parallel simulation. Foremost, it presents an approachable workflow in which users can write efficient and scalable quantum simulations without requiring prerequisite knowledge of compiled programming languages or parallel programming techniques. This is underpinned by flexible class structures and parallelisation schemes that have been designed to streamline the integration of additional parallel simulation methods.

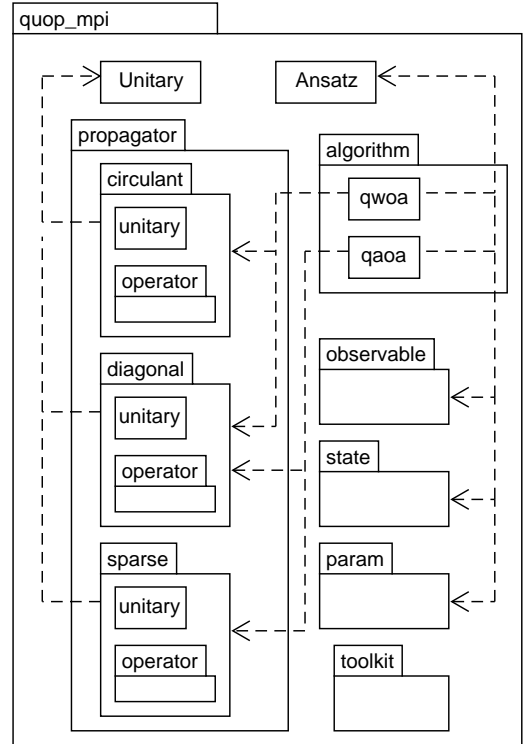


Figure 6: The user-level package structure of the QuOp\_MPI Python module.

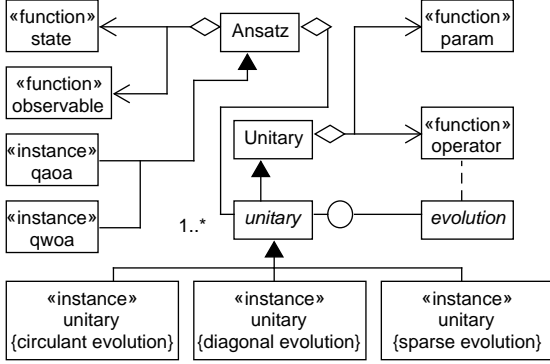


Figure 7: QuOp.MPI class structure. The Unitary class provides an interface to a state *evolution* method that is compatible with the COMM- $|\theta\rangle$  parallelisation scheme (see Section 4) that, when aggregated with an operator and *param* function, implements the action of a  $\hat{U}_{\text{phase}}$  or  $\hat{U}_{\text{mix}}$  on  $|\psi\rangle$ . The Ansatz class coordinates evaluation and minimisation of  $f$  when aggregated with a list of unitary instances, a state function and an observable function (see Table 2). QuOp.MPI includes unitary classes for circulant, diagonal and sparse matrix exponents (see Section 3), and two Ansatz instances, qaoa and qwoa, which implement simulation of the QAOA and QWOA.

#### plan(self, int N, Intracomm MPI\_COMM):

Determine the COMM- $|\theta\rangle$  parallel partitioning scheme of  $|\theta\rangle$  over Intracomm MPI\_COMM for a system of size int N and allocate memory required by external libraries (e.g. C, C++, or Fortran subroutines) called via the propagate method.

#### copy\_plan(self, unitary u):

Copy a parallel partitioning scheme from the int local\_i, int local\_i\_offset and array(int) partition\_table attributes of a planned unitary. Carry out planning tasks as described above.

#### propagate(self, float/array(float) thetas):

Compute the action of a  $\hat{U}_{\text{phase}}$  or  $\hat{U}_{\text{mix}}$  in MPI parallel over COMM- $|\theta\rangle$  given input  $\theta_i$  and attributes Intracomm MPI\_COMM, array(complex) initial\_state (the partitioned  $|\psi\rangle$ ), array(complex) final\_state (the partitioned  $|\theta\rangle$ ) and matrix exponents(s) obj operator.

#### destroy(self):

End background processes and free memory allocated in plan or copy\_plan that are not managed by the Python garbage collector.

Table 1: Unimplemented methods in the Unitary class. A backend for parallel evaluation of a  $\hat{U}_{\text{phase}}$  or  $\hat{U}_{\text{mix}}$  unitary is incorporated into QuOp.MPI through implementation of these methods in an unitary subclass. Attribute types array and Intracomm are defined as in Table 2. Type complex refers to the NumPy numerical type for double-precision complex numbers numpy.complex128.

#### Function: operator

**Associated class:** Unitary

**Binds to attributes:** int system\_size, int local\_i, array(int) partition\_table, array(float) variational\_parameters, int seed, Intracomm MPI\_COMM.

**Description:** Implements parallel generation of a mixing or phase-shift matrix exponent for a given unitary state propagation method. The matrix obj operator may be constant or parameterised by an arbitrary number of  $\theta_i$  passed via the variational\_parameters attribute. In the latter case, operator is called with each update of  $\theta_i$ .

#### Function: param

**Associated class:** Unitary

**Binds to attributes:** int system\_size, obj operator, int n\_params, int seed, Intracomm MPI\_COMM.

**Description:** Generates initial  $\theta_i$  for a unitary instance. Required positional argument n\_params specifies the size of the associated  $|\theta_i|$ . The obj operator attribute references the matrix exponent returned by the bound operator function.

#### Function: observable

**Associated class:** Ansatz

**Binds to Attributes:** int system\_size, int local\_i, int local\_i\_offset, array(int) partition\_table, Intracomm MPI\_COMM

**Description:** Implements parallel generation of  $\text{diag}(\hat{Q})$ , returning a local vector partition as an array(float) of size local\_i with a global offset of local\_i\_offset.

#### Function: state

**Associated class:** Ansatz

**Binds to attributes:** int system\_size, int local\_i, int local\_i\_offset, array(int) partition\_table, Intracomm MPI\_COMM

**Description:** Implements parallel generation of  $|\psi_0\rangle$ , returning a local vector partition as described above.

Table 2: QuOp.MPI function types. Passed to the Ansatz class by the corresponding 'set' method (see Table 3), positional arguments are mapped to attributes of the either the Unitary and Ansatz classes. An arbitrary number of keyword arguments are permitted. Attributes local\_i, local\_i\_offset and partition\_table are defined in accordance with Equations (26) and (27) with partition\_table containing the global start and end partition indexes of the distributed state vector. Integer seed is provided for reproducible pseudorandom number generation. Type array refers to a 1-dimensional NumPy ndarray and type Intracomm refers to an MPI4Py MPI intra-communicator.

<b>set_unitaries</b> [required]: Define $\hat{U}$ via a list of unitary instances.	<b>gen_initial_params</b> : Generate and return $\theta_0$ .
<b>set_observables</b> [required]: Define $\text{diag}(\hat{Q})$ via a quality function.	<b>evolve_state</b> : Compute $ \theta_f\rangle$ for input $\theta_0$ .
<b>set_initial_state</b> [default $ \psi_0\rangle =  +\rangle$ ]: Define $ \psi_0\rangle$ via a state function.	<b>execute</b> : Minimise $f(\theta)$
<b>set_depth</b> [default $D = 1$ ]: Define $D$ .	<b>benchmark</b> : Minimise $f(\theta)$ over $D = (d_{\min}, \dots, d_{\max})$ .
<b>set_optimiser</b> [default Scipy BFGS, tol = $1^{-5}$ ]: Specify the classical optimiser.	<b>get_final_state</b> : Return $ \theta_f\rangle$ at rank = 0 of the global MPI communicator.
<b>set_seed</b> [default seed = 0]: Specify a random seed for QuOp_MPI functions.	<b>get_probabilities</b> : Return $\langle s_i   \theta_f \rangle$ at rank = 0 of the global MPI communicator.
<b>set_parallel</b> [default COMM- $ \theta\rangle$ only]: Specify the parallelisation scheme.	<b>get_expectation_value</b> : Return $f(\theta_f)$ at rank = 0 of the global MPI communicator.
<b>set_log</b> [optional]: Specify simulation data-logging.	<b>save</b> : Write $ \theta\rangle$ , $\text{diag}(\hat{Q})$ , $\theta$ and the optimisation result to disk.
<b>(un)set_observable_map</b> [optional]: Define $g$ in $q = g(\text{diag}(\hat{Q}))$ , where $g : \mathbb{R}^N \rightarrow \mathbb{R}^N$ .	<b>print_optimiser_result</b> : Print the result summary of the classical optimiser.
<b>(un)set_objective_map</b> [optional]: Define $h$ in $f(\theta) = h(\langle \theta   \hat{Q}   \theta \rangle)$ , where $h : \mathbb{R} \rightarrow \mathbb{R}$ .	

Table 3: An overview of the public Ansatz class methods. Methods tagged as ‘required’ must be called before initialisation of QVA state propagation via the evolve\_state, execute or benchmark methods.

The user-level structure of QuOp\_MPI is shown in Figure 6. The package is centred around the Ansatz and Unitary ‘template’ classes. The Ansatz class manages the parallelisation scheme, definition, execution of the QVA and the recording of simulation results for a specific QVA. The Unitary class provides a scaffolding with which parallel algorithms for the computation of the action of  $\hat{U}_{\text{phase}}$  or  $\hat{U}_{\text{mix}}$  on  $|\psi\rangle$  are integrated with QuOp\_MPI. Overviews of the key methods of the Ansatz class and the user-implemented methods required to integrate a state evolution method into QuOp\_MPI via the Unitary class are given in Tables 1 and 3 respectively.

As shown in Figure 7 and Table 1, a QuOp\_MPI compatible method for simulating the action of a  $\hat{U}_{\text{phase}}$  or  $\hat{U}_{\text{mix}}$  on  $|\psi\rangle$  is implemented through the creation of an Unitary subclass (unitary) which defines methods responsible for determination of the COMM- $|\theta\rangle$  parallel partitioning scheme (see Figure 4), computation of the action of the unitary and management of the ancilla requirements of any external subroutines.

The propagator submodule contains predefined unitary classes together with an operator submodule containing functions for the generation of compatible matrix exponents  $\hat{O}$  or  $\hat{W}$ . Three unitary classes are included as part of the diagonal, sparse and circulant submodules, which simulate  $\hat{U}_{\text{phase}}$ ,  $\hat{U}_{\text{mix}}$  with sparse matrix exponents and  $\hat{U}_{\text{mix}}$  circulant matrix exponents (see Section 3).

Submodules state, param and observable provide functions that, when passed to the Ansatz class, define  $|\psi_0\rangle$ ,  $\theta_0$  and  $\text{diag}(\hat{Q})$  for a particular QVA.

Two predefined QVAs are included in the algorithm submodule, qwoa and qaoa. These Ansatz subclasses implement the QWOA and QAOA respectively. The toolkit submodule provides convenience functions to assist in constructing matrix operators and quantum states involving the tensor product of Pauli matrices and bit-string qubit states.

Finally, QuOp\_MPI is highly extensible through its support for user-defined functions for the generation of the matrix exponents,  $\theta_0$ ,  $\text{diag}(\hat{Q})$  and  $|\psi_0\rangle$ , as described in Table 2.

Within this structure, QuOp\_MPI thus presents several levels of usage:

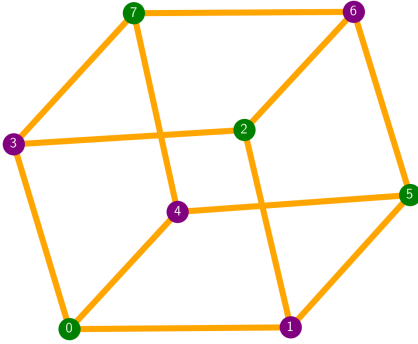
1. Simulation of the QWOA or QAOA using the qwoa or qaoa classes with user defined  $\text{diag}(\hat{Q})$ .
2. Simulation of the QWOA or QAOA using user-defined parallel generation of  $\text{diag}(\hat{Q})$ .
3. Design and simulation of a QVA with the Ansatz class with included or user defined functions specifying matrix exponents,  $\theta_0$ ,  $\text{diag}(\hat{Q})$  and  $|\psi_0\rangle$ .
4. Integration of additional state evolution methods for  $\hat{U}_{\text{mix}}$  or  $\hat{U}_{\text{phase}}$  through the creation of Unitary subclasses.



```

1 from quop_mpi.algorithm import qaoa
2 from quop_mpi import observable
3 from quop_mpi.toolkit import I, Z
4 import networkx as nx
5
6 graph = nx.circular_ladder_graph(4)
7
8 n_qubits = graph.number_of_nodes()
9 system_size = 2 ** n_qubits
10
11 def maxcut_qualities(graph, n_qubits):
12     C = 0
13     for edge in graph.edges():
14         C += 0.5*(1(n_qubits) + \
15
16         (Z(edge[0], n_qubits) @ Z(edge[1], n_qubits)))
17     return C.diagonal()
18
19 alg = qaoa(system_size)
20
21 alg.set_qualities(observable.serial,
22                  {"function": maxcut_qualities,
23                   "args": [graph, n_qubits]})
24
25 alg.set_depth(2)
26 alg.execute()
27 alg.print_optimiser_result()
28 alg.save("maxcut", "depth_2", "w")
    
```

Figure 8: Example 1: Simulation of the QAOA applied to the max-cut problem.


 Figure 9: Graph for which the max-cut problem is solved in Examples 1 and 2. The most probable solution for  $|\theta\rangle_{\text{QAOA}}$  and  $|\theta\rangle_{\text{ex-QAOA}}$  ( $\bar{s} = (0, 1, 0, 1, 1, 0, 1, 0)$ ) is shown by vertex colouring with purple (darker) indicating a 0 and green (lighter) indicating a 1. This partitioning corresponds to the optimal solution for which  $C(\bar{s}) = q_{90} = 0$ .

## 6. Usage Examples

The following introduces typical QuOp\_MPI usage by simulating the QAOA, ex-QAOA, QAOAz and QWOA as applied to the max-cut and portfolio-re-balancing optimisation problems.

### 6.1. The max-cut problem.

The max-cut problem seeks to partition the vertices of a graph such that a maximum number of neighbouring nodes are assigned to two disjoint sets [3]. A quantum encoding of the max-cut problem is a bijective mapping of the vertices of a graph  $G$  to  $n$  qubits, with the set membership indicated by the corresponding qubit state. For example, a two vertex graph with vertices  $\{0, 1\}$  has a solution space that is completely represented by an equal superposition over a two-qubit system:  $\{\{0, 1\} \rightarrow |00\rangle, \{0, 1\} \rightarrow |01\rangle, \{0, 1\} \rightarrow |10\rangle$  and  $\{0, 1\} \rightarrow |11\rangle$ .

The cost function is then implemented as

$$C(s) = \sum_{E(i,j) \in G} \frac{1}{2} (\mathbb{I} + Z_i Z_j), \quad (28)$$

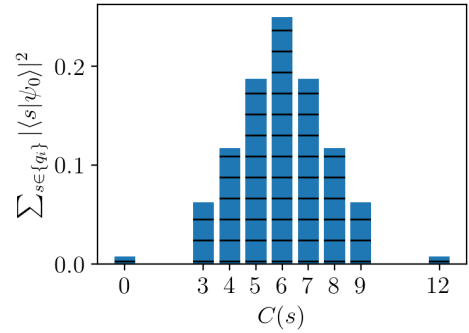


Figure 10: The initial solution probability distribution for the max-cut problem simulated in Examples 1 and 2.

where  $Z_i$  is a Pauli Z gate acting on the  $i^{\text{th}}$  qubit,  $E(i, j)$  is an edge in  $G$  connecting vertex  $i$  to vertex  $j$ , and  $Z_i Z_j$  has eigenvalue 1 if qubits  $i$  and  $j$  are in the same state or  $-1$  otherwise.

#### 6.1.1. QAOA

In Example 1 the QAOA is applied to the max-cut problem for the graph shown in Figure 9. The predefined Ansatz subclass `qaoa` forms the basis of the simulation.

To generate the graph we use the external module `networkx` (Line 6). On Lines 11 to 16, the cost function is defined. By using the `I` and `Z` functions from the `toolkit` submodule, we are able to directly implement Equation (28). The matrices computed on Lines 14 and 15 are in a SciPy sparse matrix format.

Lines 18 to 27 demonstrate standard use of the `qaoa` class. An instance of the class is instantiated for `system_size = N`. Next, the `diag(Q)` is defined via the `set_qualities` method. For this, we pass the `serial` observable function along with a dictionary of its keyword arguments. The `serial` function assists with memory-efficient simulation given a serial observable function by calling the function at the root MPI process and distributing its output over `COMM-0`. The ansatz depth ( $D = 2$ ) is then defined via the `set_depth` method.

Now that the `qaoa` instance is fully specified, simulation of the algorithm (as defined in Equation (12)) proceeds via the

QuOp\_MPI/examples/max-cut\_extended/maxcut\_extended.py

```

1 from quop_mpi import Ansatz
2 from quop_mpi.propagator import diagonal, sparse
3 from quop_mpi.observable import serial
4 from quop_mpi.param.rand import uniform
5 from quop_mpi.toolkit import I, Z
6 import numpy as np
7 import networkx as nx
8
9 graph = nx.circular_ladder_graph(4)
10
11 n_qubits = graph.number_of_nodes()
12 n_edges = 2 * graph.number_of_edges()
13
14 system_size = 2 ** n_qubits
15
16 def maxcut_terms(graph, n_qubits):
17     terms = []
18     for edge in graph.edges:
19         term = 0.5*(I(n_qubits) + Z(edge[0], n_qubits) @ \
20             Z(edge[1], n_qubits))
21         terms.append(term.diagonal())
22     return terms
23
24 def maxcut_qualities(terms):
25     return np.sum(terms, axis=0)
26
27 computed_terms = maxcut_terms(graph, n_qubits)
28
29 UQ = diagonal.unitary(
30     diagonal.operator.serial,
31     operator_kwargs={"function": maxcut_terms,
32                     "args": [graph, n_qubits]},
33     unitary_n_params=n_edges,
34     parameter_function=uniform)
35
36 UW = sparse.unitary(sparse.operator.hypercube,
37                    parameter_function=uniform)
38
39 alg = Ansatz(system_size)
40
41 alg.set_unitaries([UQ, UW])
42
43 alg.set_observables(serial,
44                    {"function": maxcut_qualities,
45                     "args": [computed_terms]})
46
47 alg.execute()
48 alg.print_optimiser_result()
49 alg.save("maxcut_extended", "depth_1", "w")

```

Example 2: Simulation of the ex-QAOA applied to the max-cut problem.

QuOp\_MPI/examples/maxcut\_plots.py

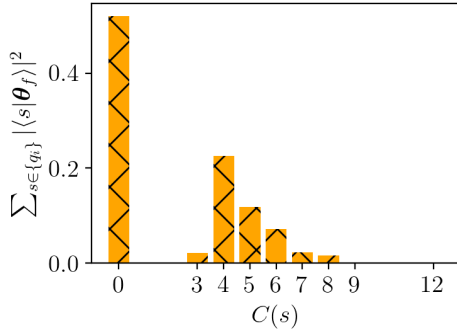


Figure 11: Solution quality probability distribution of  $|\theta\rangle_{\text{QAOA}}$  as simulated in Example 1.

execute method. By calling execute without specifying  $\theta$  we choose to use default param functions which generate  $\theta_i$  from a uniform distribution over  $(0\pi, 2\pi]$ .

Finally, the optimiser result is displayed using the print\_optimiser\_result method and the simulation results are saved to the HDF5 file 'maxcut.h5' under the 'depth 2' group. As compared to a starting expectation value of 7.43, the final value of the objective function (fun) is approximately 2.30 with variational parameters (x) 3.59, 6.88, 3.95 and 5.92.

Figures 10 and 11, illustrates the initial and final probability distributions with respect to unique values of  $q_i$ . After application of the QAOA to the initial superposition, probability density is concentrated at high-quality solutions with the optimal solution ( $q_{90} = 0$ ) having the highest probability of measurement.

QuOp\_MPI/examples/maxcut/maxcut\_parallel\_qualities.py

```

1 def maxcut(local_i, local_i_offset, graph=None):
2
3     n_qubits = graph.number_of_nodes()
4     n_edges = graph.number_of_edges()
5
6     q = np.full(local_i, n_edges, dtype = np.float64)
7
8     start = local_i
9     end = local_i + local_i_offset
10
11     for i in range(start, end):
12         bs = np.binary_repr(i, width = n_qubits)
13         for edge in graph.edges:
14             if bs[edge[0]] != bs[edge[1]]:
15                 q[i - local_i_offset] -= 1
16
17     return q

```

Example 3: User-defined quality function for the max-cut problem.

### 6.1.2. Extended-QAOA

Having demonstrated the effectiveness of the QAOA in finding high-quality max-cut solutions, we will now explore the application of the ex-QAOA to the same task. Example 2 demonstrates the implementation of ex-QAOA using the Ansatz class. As in Example 1, the graph and its adjacency matrix are generated using networkx.

The functions needed to implement Equation (13) and Equation (10) are defined from Lines 16 to 22 and Lines 24 to 25 respectively. The first of these, maxcut\_terms, returns an array of the summation terms in Equation (28) with which the maxcut\_qualities function returns  $\text{diag}(\hat{Q})$ .

A two-step calculation of the solution qualities is chosen as the ex-QAOA phase-shift operator associates a  $\theta_i$  with each

```

1 from quop_mpi.algorithm import qwoa
2 from quop_mpi import observable
3 import pandas as pd
4
5 qualities_df = pd.read_csv('qwoa_qualities.csv')
6 qualities = qualities_df.values[:,1]
7
8 system_size = len(qualities)
9
10 alg = qwoa(system_size)
11
12 alg.set_qualities(
13     observable.array,
14     {'array': qualities})
15
16 alg.set_log(
17     'qwoa_portfolio_log',
18     'qwoa',
19     action = 'w')
20
21 alg.benchmark(
22     range(1,6),
23     3,
24     param_persist = True,
25     filename = 'qwoa_portfolio',
26     save_action = 'w')

```

Example 4: Simulation of the QWOA applied to the portfolio re-balancing problem.

term of a Pauli-matrix decomposition of  $\text{diag}(\hat{Q})$ . The phase-shift-unitary is implemented using the propagator submodule diagonal. An instance of the diagonal submodule unitary class (UQ) is defined on Lines 29 to 34. The first argument specifies the operator function responsible for generating the Pauli-matrix terms  $\Sigma_j$ . The second specifies a dictionary of user-defined keyword arguments for the operator function. The third argument specifies the number of  $\theta_i$  associated with UQ and, finally, the fourth argument specifies the param function used to initialise the unitary's variational parameters. The param function generates  $\theta_i$  as described in Section 6.1.1.

The operator function `diagonal.serial` executes the serial `maxcut_terms` function at the root MPI process and distributes the array of Pauli-matrix terms over  $\text{COMM-}|\theta\rangle$ . The `unitary_n_params` keyword argument describes the number of operator terms returned by `diagonal.serial`, which are mapped to a sequence of  $\hat{U}_{\text{phase}}$  unitaries each parameterised by a unique  $\theta_i$ .

Definition of the mixing-unitary UW occurs on Lines 36 to 37. As with UQ, the first argument specifies the operator function and the `parameter_function` argument specifies the param function. The operator function `sparse.operator.hypercube` generates a parallel-partitioned instance of the hypercube mixing operator (see Equation (20)).

On Line 41 the defined unitaries UQ and UW are passed to an instance of the Ansatz class via the `set_unitaries` method. The objective function is then defined by passing the `maxcut_qualities` function to the `set_observables` method.

The ex-QAOA simulation is then executed on Line 47. As  $D$  has not been specified via the `set_depth` method, the algorithm is simulated with the default ansatz depth of  $D = 1$  (see Figure 12).

### 6.1.3. Parallel Computation of the Cost Function

As a corollary of condition two in Section 2.2 it will generally be the case that computation of any particular  $C(s)$  is independent of the rest of the cost function values. In such instances, the generation of  $\text{diag}(\hat{Q})$  is an embarrassingly parallel

problem, and, as such, users are encouraged to implement parallel quality functions. A parallel observable function for the max-cut problem is shown in Example 3 as per the requirements described in Table 2. At run-time, this function is called by each rank in  $\text{COMM-}|\theta\rangle$  to generate the  $q_i$  specific to the local vector partitions.

QuOp-MPI/examples/maxcut\_extended/maxcut\_extended-plots.py

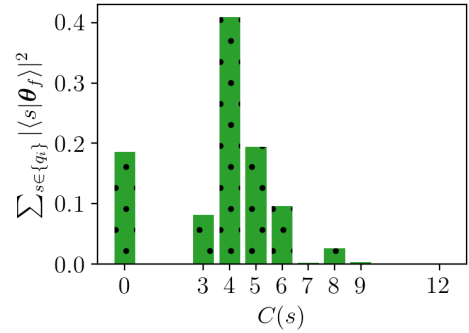


Figure 12: Final probability distribution of the max-cut solutions following the execution of the ex-QAOA defined in Example (2).

## 6.2. Portfolio Re-balancing

To explore the case of constrained optimisation using the QWOA and the QAOAz we will consider the problem of portfolio re-balancing. For each asset in a portfolio of size  $M$ , an investor must choose one of the following positions:

1. Short position: buying and selling an asset with the expectation that it will drop in value.
2. Long position: buying and holding the asset with the expectation that it will rise in value.
3. No position: taking neither the long or short position.

A quantum encoding of the possible solutions uses two qubits per asset.

1.  $|01\rangle \rightarrow$  short position

```

1 from quop_mpi import Ansatz, observable, state, param
2 from quop_mpi.propagator import diagonal, sparse
3 from quop_mpi.toolkit import kron, kron_power
4 from quop_mpi.toolkit import string, X, Y
5 from qaoaz_qualities import qaoaz_portfolio
6 from numpy import sqrt
7
8 def parity_ring(i, j, n_qubits):
9     parity = X(i, n_qubits) @ X(j, n_qubits) \
10         + Y(i, n_qubits) @ Y(j, n_qubits)
11     return parity
12
13 def parity_mixer(qubits, n_qubits):
14
15     odd = 0
16     even = 0
17
18     n_subset = len(qubits)
19
20     for i in range(n_subset - 1):
21
22         if (i % 2 != 0):
23             odd += parity_ring(qubits[i],
24                               qubits[(i + 1) % n_subset],
25                               n_qubits)
26
27         elif i % 2 == 0:
28             even += parity_ring(qubits[i],
29                                qubits[(i + 1) % n_subset],
30                                n_qubits)
31
32     mixer = [odd, even]
33
34     if len(qubits) % 2 != 0:
35         last = parity_ring(qubits[-1],
36                            qubits[1],
37                            n_qubits)
38
39     mixer.append(last)
40
41     return mixer
42
43 def mixer(n_qubits):
44     short_qubits = [i for i in range(0, n_qubits, 2)]
45     long_qubits = [i for i in range(1, n_qubits, 2)]
46     short_mixer = parity_mixer(short_qubits, n_qubits)
47     long_mixer = parity_mixer(long_qubits, n_qubits)
48     return short_mixer + long_mixer
49
50 def parity_state(n_qubits, D):
51     M = n_qubits // 2
52     term_1 = kron_power(string('01'), D)
53     term_2 = kron_power(
54         1/sqrt(2) * (string('11') + string('00')),
55         M-D)
56     state = kron([term_1, term_2])
57     return state
58
59 n_qubits = 8
60 system_size = 2*n_qubits
61
62 UQ = diagonal.unitary(
63     qaoaz_portfolio,
64     parameter_function = param.rand.uniform)
65
66 UW = sparse.unitary(
67     sparse.operator.serial,
68     operator_kwargs = {
69         'function': mixer,
70         'args': [n_qubits]},
71     parameter_function = param.rand.uniform)
72
73 alg = Ansatz(system_size)
74
75 alg.set_unitaries([UQ, UW])
76
77 alg.set_initial_state(
78     state.serial,
79     {'function': parity_state,
80      'args': [n_qubits, 2]})
81
82 alg.set_observables(0)
83
84 alg.set_log(
85     'qaoaz_portfolio_log',
86     'qaoaz',
87     action = 'w')
88
89 alg.benchmark(
90     range(1, 6),
91     3,
92     param_persist = True,
93     filename = 'qaoaz_portfolio',
94     save_action = 'w')

```

Example 5: Simulation of the QAOAz applied to the portfolio re-balancing problem.

2.  $|10\rangle \rightarrow$  long position
3.  $|00\rangle$  or  $|11\rangle \rightarrow$  no position

The discrete mean-variance Markowitz model provides a means of evaluating the quality associated with a given combination of positions. It can be expressed through minimisation of the cost function,

$$C(s) = \omega \sum_{i,j=1}^M \sigma_{ij} Z_i Z_j - (1 - \omega) \sum_{i=1}^M r_i Z_i, \quad (29)$$

subject to the constraint,

$$\chi_{\text{asset}}(s) = \sum_{i=1}^M z_i. \quad (30)$$

In this formulation, the Pauli-Z gates  $Z_i$  encode a particular portfolio where, for each asset, eigenvalue  $z_i \in \{1, -1, 0\}$  rep-

resents a choice of long, short or no position. Associated with each asset is the expected return  $r_i$  and covariance  $\sigma_{ij}$  between assets  $i$  and  $j$ ; which are calculated using historical data. The risk parameter,  $\omega$ , weights consideration of  $r_i$  and  $\sigma_{ij}$  such that as  $\omega \rightarrow 0$  the optimal portfolio is one providing maximum returns. In contrast, as  $\omega \rightarrow 1$ , the optimal portfolio is the one that minimises risk. The constraint  $\chi_{\text{asset}}(s)$  works to maintain the relative net position with respect to a pre-existing portfolio [33].

In the following examples, we demonstrate the application of the QWOA and QAOAz to a small ‘portfolio’ consisting of four assets taken from the ASX 100, under the constraint  $\chi_{\text{asset}}(s) = 2$ .

#### 6.2.1. Comparison of the QWOA and QAOAz.

As outlined in Section 2.4.2, QWOA uses an indexing unitary  $U_{\#}^{\dagger}$  to encode constraints on the solution space. As QuOp\_MPI is interested only in the unitary dynamics of the quantum state

evolution, implementation of the indexing unitary simply requires that the user supply a quality function that returns the restricted solution space in a consistent order. For the QWOA example, the set of valid solutions was calculated using the module ‘qwoa\_qualities.py’, which was originally written for [33]. It makes use of the pandas-webreader package [34] to source the daily adjusted close price of a given list of stocks from the Yahoo Finance website [35]. When run as the main program, ‘qwoa\_qualities.py’ returns the  $\text{diag}(\hat{Q})$  corresponding to solutions in  $\mathcal{S}'$  using historical data between user-specified dates and outputs  $\text{diag}(\hat{Q})$  to a CSV file. For this example, the stocks AMP.AX, ANZ.AX and AMC.AX were considered between 1/1/2017 and 12/31/2018.

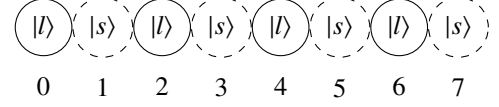
The QWOA algorithm is included with QuOp\_MPI as a pre-defined module. As such, its simulation, shown in Example 4, is similar in structure to the QAOA program outlined in Section 6.1.1. However, as in this instance, because  $\text{diag}(\hat{Q})$  is stored in a CSV file, we use the external package Pandas to read the quality values and the diagonal submodule operator function `serial_array` to pass these values to the `set_qualities` method on Lines 12 to 14.

Note that the size of the simulation is determined by the number of valid solutions  $|\mathcal{S}'|$ . This is distinct from a quantum implementation of the QWOA algorithm as, while its  $\hat{U}_{\text{mix}}$  occurs over  $\mathcal{S}'$ , its  $\hat{U}_{\text{phase}}$  still acts on  $\mathcal{S}$ . However, because  $|\psi_0\rangle_{\text{QWOA}}$  is initialised as an equal superposition over  $\mathcal{S}'$ , quantum states associated with invalid solutions do not influence the idealised quantum dynamics. Hence, we can gain a significant performance advantage at no cost to simulation accuracy by restricting the classical simulation to  $\mathcal{S}'$ .

In Section 6.1.1, the final state  $|\theta\rangle$  and  $\text{diag}(\hat{Q})$  were saved to an HDF5 file and analysis of algorithm performance was determined via computation on these arrays. Studying the complete quantum state is essential to understanding the dynamics associated with a particular QVA application. Still, often a researcher is concerned more immediately with  $f(\theta_f)$  with respect to changes in  $D$  or  $|\mathcal{S}'|$ . For this reason QuOp\_MPI supports the recording of important simulation metrics in a log file. Created via the `set_log` method on Lines 16 to 19, the first argument specifies the name of the CSV output log file, the second argument specifies the simulation label, and the third argument specifies the write action, which follows the convention of a to append or w to (over)write. With a log file set the system size  $N$ , ansatz depth  $D$ , optimised objective function value  $f(\theta_f)$ , state norm  $\langle\theta_f\rangle$ , in-program simulation time, MPI communicator size, number of  $|\theta\rangle$  evaluations and the success status of the optimiser are recorded for each simulation instance.

To study how  $f_{\text{QWOA}}(\theta_f)$  changes as  $D$  increases we call the `benchmark` method on Lines 21 to 26. The first argument is an iterable object that provides a sequence of  $D$  values, the second is the number of repeat simulations at each  $D$ . The keyword arguments `filename` and `save_action` specify that  $|\theta_f\rangle_{\text{QWOA}}$  and  $\text{diag}(\hat{Q})$  be saved to the new HDF5 file ‘qwoa\_portfolio.h5’. The `param_persist` argument specifies a schema for  $\theta_0$ . If True, for  $D > 1$  the best-performing  $\theta$  at depth  $D$  are used as the  $\theta_0$  for the first  $D$  ansatz iterations at depth  $D + 1$ .

A QAOAz approach to the portfolio optimisation problem uses two parity mixers that act on the short and long qubits, respectively, such that the  $\mathcal{S}$  is partitioned into subgraphs of the same  $\chi_{\text{asset}}(s)$  value. For this example, we are considering four assets so the two parity mixers act on separate subspaces of  $\mathcal{H}$  as shown below:



Where  $|l\rangle$  denotes a ‘long’ qubit,  $|s\rangle$  denotes a ‘short’ qubit, and the numbering indicates the global index of each qubit.

To constrain probability amplitude to  $\mathcal{S}'$ ,  $|\psi_0\rangle_{\text{QAOAz}}$  is prepared as

$$|\psi_0\rangle_{\text{QAOAz}} = |01\rangle^{\otimes A} \left( \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)^{2N-A} \right), \quad (31)$$

where  $A$  is the desired value of  $\chi_{\text{asset}}(s)$ . This creates a (non-equal) superposition of states across all qubit subgraphs with a net parity of  $A$ .

To implement this algorithm in QuOp\_MPI we use the `Ansatz` class, the `sparse_propagator` submodule, the `diagonal_propagator` submodule, the `observable` submodule, `state` submodule and `param` submodule. The toolkit functions `string`, `X` and `Y` are also used to define the parity mixers and  $|\psi_0\rangle_{\text{QAOAz}}$ .

As with Example 4, the quality function is located in the external module ‘qaoaz\_portfolio.py’, which is included in QuOp\_MPI/examples/portfolio. It follows the same method as ‘qwoz\_portfolio.py’, differing in that it has been written as a parallel quality function (see Section 6.1.3) that returns local partitions of the complete  $\mathcal{S}$ .

The dual parity mixing operators are defined over three functions. The first of these (Lines 8 to 11) defines a generalisation of the Pauli-matrix terms used for the  $B_{\text{odd}}$ ,  $B_{\text{even}}$  and  $B_{\text{last}}$  mixing operations in Equation (10). The second function (Lines 13 to 41) takes a list of qubit indexes specifying a subspace of  $\mathcal{H}$  and the total number of qubits as its arguments and returns  $B_{\text{odd}}$ ,  $B_{\text{even}}$  and  $B_{\text{last}}$  acting on the subspace. The third function, `portfolio_mixer` (Lines 43 to 48), takes a number of qubits as its argument. It partitions the input number of qubits into subgroups, as depicted in Figure 2, and returns a list containing the six mixing operators in the SciPy CSR sparse matrix format.

A function to generate  $|\psi_0\rangle_{\text{portfolio}}$  is defined on lines 50 to 57 where, on Lines 52 and 53, the `kron_power` function takes NumPy array  $a$  and integer  $N$  as its arguments and returns  $a^{\otimes N}$  and, on Line 54, the `string` function generates a qubit state from its bit-string representation. The function `kron`, on Line 56, takes a list of arrays and returns their tensor product.

We then proceed to the definition of  $\hat{U}_{\text{QAOAz}}$  using the `Ansatz` class. An initial state other than an equal superposition is specified using the `set_initial_state` method on Lines 77 to 80. It follows the same input convention as the previously described ‘set’ methods. The wrapper function `state.serial` is used to parse and distribute the output of the `serial_parity_state` function.

As, in this instance, the diagonal of the phase-shift matrix exponent is equal to  $\text{diag}(\hat{Q})$ , the objective function is defined by calling `set_observables` on Line 82 with an integer argument that specifies the position of the mixing operator in the input list of unitaries (Line 75).

Finally, an output log is specified, and the benchmark method is called to trial the QVA over the same range of  $D$  and number of repeats as the QWOA simulations. The benchmark method generates a reproducible sequence of integers used as random seeds for all param functions in the param submodule. In this way, we ensure that the QWOA and QAOAz simulations are carried out over the same set of  $\theta_0$  at the starting ansatz depth of  $D = 1$ .

A comparison of the two algorithms is shown in Figure 13 where the  $f(\theta_f)$  was taken from the log file. For this brief comparison the QWOA  $f(\theta_f)$  outperforms the QAOAz for all  $D > 1$ .

QuOp\_MPI/examples/portfolio\_rebalancing/portfolio\_plots.py

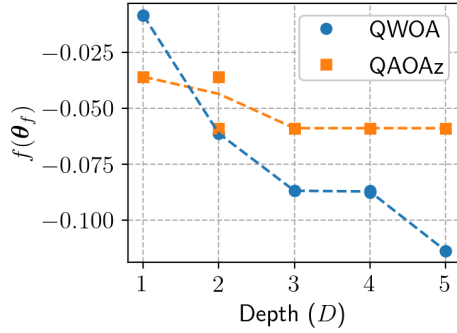


Figure 13: Optimised objective function value  $f(\theta_f)$  for the portfolio rebalancing problem using the QWOA and QAOAz.

## 7. Performance

The performance of QuOp\_MPI was assessed on the ‘Magnus’ system at the Pawsey Supercomputing Centre - a Cray XC40 Series Supercomputer with an Aries interconnect rated at 72 gigabits-per-second per node. Each node consisted of two Intel Xeon E5-2690V3 ‘Haswell’ CPUs with 12 cores clocked at 2.6 GHz and 64 GB of RAM.

The strong scaling behaviour of the QAOA and QWOA evolution methods on a single compute node is shown in Figures 14a and 14b. As defined in Equations (12) and (21), each of the unitaries contains a phase-shift-unitary followed by a mixing-unitary. For  $|\theta\rangle_{\text{QAOA}}$  these are implemented using the diagonal and sparse unitary classes and, for  $|\theta\rangle_{\text{QWOA}}$ , the diagonal and circulant unitary classes. A parallel advantage in QAOA state evolution is observed for systems of at least 12 qubits, with a system of 20 qubits scaling with an efficiency greater than 0.5 up to eight CPU cores. For QWOA, parallel advantage beyond two CPU cores starts at 12 qubits, with a system of 16 qubits achieving a speedup of 15.1 with an efficiency of 0.63 at 24 CPU cores.

Strong scaling behaviour for QAOA and QWOA evolution across multiple nodes is shown in Figures 14c and 14d. Efficient scaling of state evolution to two nodes occurs at 17 qubits for QAOA and 15 qubits for QWOA. For QAOA at 24 qubits, a speedup of 5.23 times at an efficiency of 0.44 was achieved at 12 nodes (288 cores) with respect to the wall-time of a fully occupied single node (24 cores). For QWOA, the equivalent comparison shows a speedup of 9.25 with an efficiency of 0.77 at 12 nodes.

The QWOA and QAOA state evolution methods were profiled using Arm Map 19.0.1 to quantify the degree of communication overhead in a distributed computing environment as shown by Figure 16. For QAOA state evolution at 22 qubits, the overhead ranged from a total of 17.2 % at one node and 49.3 % at six nodes. The increase in communication overhead is responsible for the decrease in strong scaling efficiency with QAOA state evolution at 22 qubits having an efficiency that falls below 0.5 for nodes greater than six (see Figure 14c). Almost all of the QAOA state evolution MPI call time is spent in collective calls, of which the majority are ‘Alltoallv’ calls responsible for the sending and receiving of state vector elements during matrix multiplication. State evolution for the QWOA is dominated by a one-dimensional Fourier transform computed in MPI parallel using the FFTW3 package. For QWOA state evolution at 19 qubits, the time spent in MPI calls ranges from 4.8 % for one node and 19.6 % at six nodes, a modest increase that is in line with the efficient scaling depicted in Figure 14d.

Scalability of the state evolution methods for qaoa and qwoa at a constant MPI process load (weak scaling) is shown in Figure 15 with respect to cores on a single node and a cluster of multiple nodes. In each instance, imperfect weak scaling is observed as, for both QAOA and QWOA, increases in  $n$  are accompanied by an increased degree of inter-qubit coupling. For high process loads, the qaoa state evolution method scales more efficiently than qwoa state evolution, which is consistent with the structure of the corresponding mixing operators. For the QAOA the hypercube matrix operator has a sparse banded-diagonal structure that, for  $\text{local}_i \bmod 2 = 0$ , induces a communication overhead of  $O(\log_2(\text{size}))$ . In contrast, the complete-graph mixing operator of the QWOA requires communication between all of the MPI processes resulting a communication overhead of  $O(\text{size})$ .

As a measure of solution quality, deviation from the norm was calculated for the state evolution results shown in Figures 14 and 15. Figure 17 shows the total deviation divided by the system size to indicate the per-state accuracy. For both QAOA and QWOA the deviation is on the order of  $10^{-13}$  or below, which is consistent with double precision accuracy.

The effectiveness of the various optimisation algorithms included with the SciPy and NLOpt packages was considered with respect to simulation of the QAOA and QWOA. This comparison adopted methodology outlined in the NLOpt documentation [31]. A system of 16 qubits ( $N = 2^{16}$ ) was considered with a randomly generated  $\text{diag}(\hat{Q})$  consisting of values from a uniform distribution over (0, 1]. Five sets of  $\theta_0$  were generated for  $D = 5$  ( $|\theta| = 10$ ) and the algorithms simulated using the optimisers listed in the caption of Figure 18. For each of the five



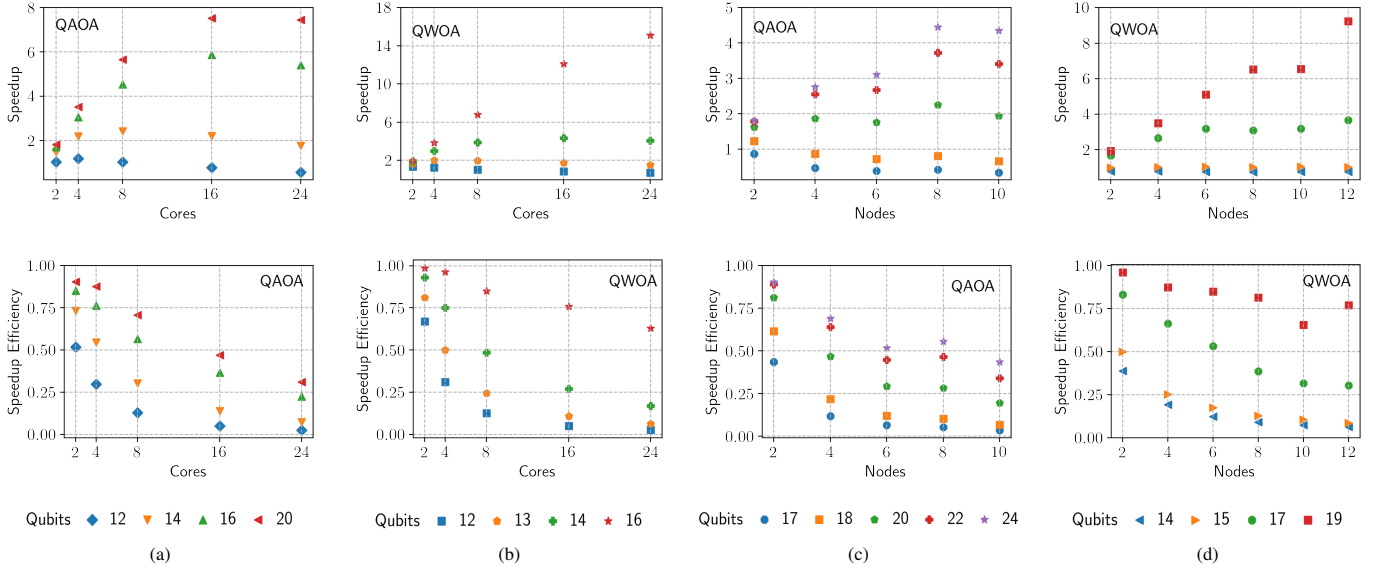


Figure 14: Strong scaling speedup and efficiency for the `qaoa` and `qwoa` state evolution methods running on a single and multiple nodes. For each trial, a `qaoa` or `qwoa` instance was instantiated with a  $\text{diag}(\hat{Q})$  consisting of uniformly distributed floats in  $(0, 1]$ . The ansatz depth was set to  $D = 15$  such that calling the `Ansatz evolve_state` method resulted in 15 repeats of the state evolution subroutines implementing the phase-shift and mixing-unitaries. The  $\theta_0$  were prepared identically for all trials at the same number of qubits from the uniform distribution  $(0, 2\pi]$ . For (a) and (b) speedup is reported proportional to the time taken using a single CPU core (1 MPI process). The `qaoa` the single-node wall-times were 3.27 s, 5.35 s, 17.3 s and 265 s for 12, 13, 14 and 16 qubits respectively and, for `qwoa`, 2.39 s, 2.50 s, 5.05 s, 24.0 s and 727 s for 10, 12, 14, 16 and 20 qubits respectively. Efficiency is defined as the speedup divided by the number of CPU cores. For (c) and (d) all trails run on fully occupied nodes and the reported speedup is proportional to the time taken on one fully occupied node (24 MPI processes). For `qaoa` the single-node wall-times were 7.64 s, 17.8 s, 97.5 s, 496 s and 2365 s for 12, 13, 14 and 16 qubits respectively and, for `qwoa`, 4.26 s, 7.03 s, 60.0 s and 967 s for 10, 12, 14, 16 and 20 qubits respectively. Efficiency in (c) and (d) is defined as the speedup divided by the number of nodes.

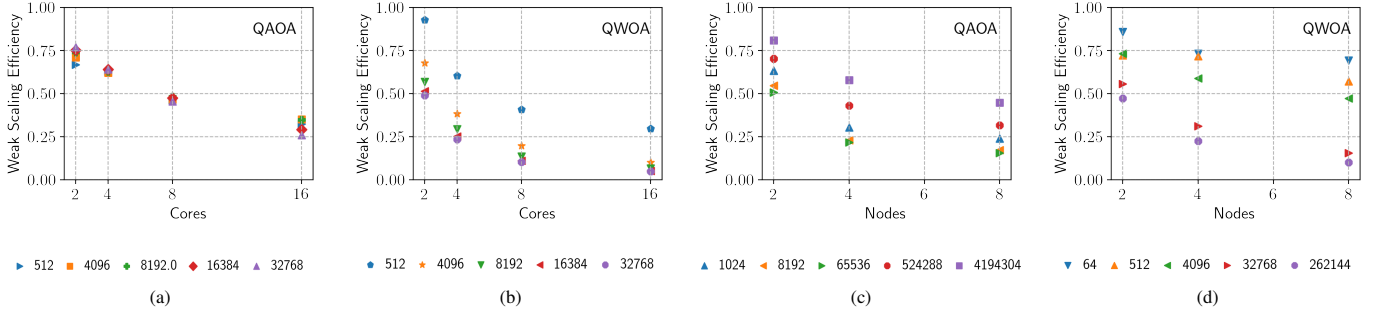


Figure 15: The weak scaling efficiency for the `qaoa` and `qwoa` state evolution methods with state vector partitions of size `local_i` as indicated by the corresponding plot legends. For (a) and (b), efficiency is defined as  $T(1)/T(\text{Cores})$  where  $T(1)$  is the wall-time for one MPI process with a system size of `local_i`. For (c) and (d), efficiency is defined as  $T(1)/T(\text{Nodes})$  where  $T(1)$  is the wall-time for one Node of 24 MPI processes with `local_i` state vector elements and all nodes were fully occupied at 24 MPI process per-node.

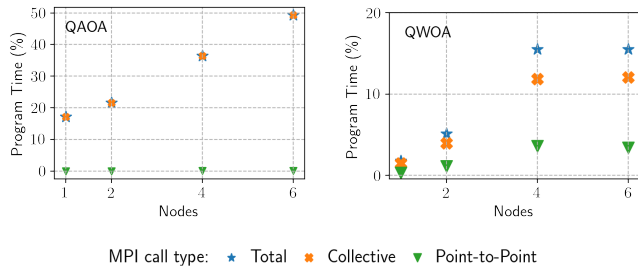


Figure 16: The percentage of program wall-time spent in MPI calls for the `qaoa` and `qwoa` state evolution methods at 22 and 19 qubits, respectively, as reported by Arm Map version 19.0.1. All nodes were fully occupied at 24 MPI processes per node.

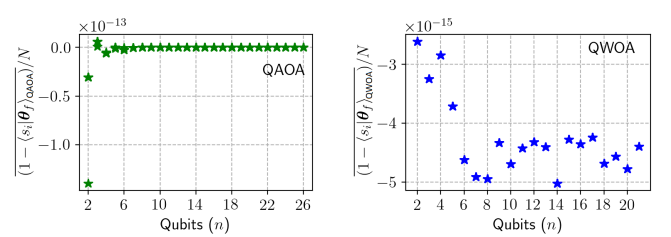


Figure 17: Average deviation from the norm for the  $|\theta_f\rangle_{QAOA}$  (left) and  $|\theta_f\rangle_{QWOA}$  (right) depicted in Figures 14 and 15

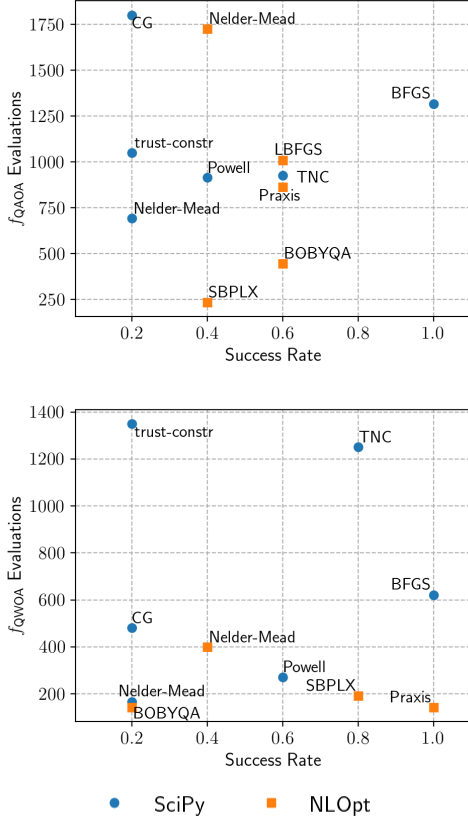


Figure 18: A comparison of the optimisation algorithms included with the SciPy and NLOpt packages. The plots depict algorithms that satisfied the convergence criteria in at least one out of the five trials. The complete list of considered algorithms is SciPy: BFGS, CG, Nelder-Mead, trust-constr, Powell, TNC and NLOpt: LD\_LBFGS, LN\_BOBYQA, LN\_PRAXIS, LN\_NELDERMEAD, LN\_SBPLX, LD\_MMA, LD\_CCSAQ. The comparison was carried out on 17 nodes over 48 hours.

sets of  $\theta_0$  the lowest  $\theta_f$  was used to define five instances of the modified objective function

$$f'(\theta) = |\min(f_{\theta_i}) - f|, \quad (32)$$

where  $\min(f_{\theta_i})$  is the minimum  $f(\theta_f)$  found by any of the considered optimisation algorithms with initial variational parameters  $\theta_i$ . Each optimisation algorithm was then trialled with starting parameters  $\theta_i$  and the objective function defined as in Equation (32). A particular algorithm was considered to have ‘succeeded’ if it converged to a point satisfying  $f'(\theta) < b$ , where  $b = 0.8$  was chosen as it produced an informative measure across a large subset of the considered optimisers.

For the QAOA trials the minimum final objective function values  $f_{\text{QAOA}}(\theta_f)$  were 0.162 (Powell), 0.156 (BFGS), 0.120 (BOBYQA), 0.228 (LD\_LBFGS) and 0.154 (LD\_LBFGS). For the QWOA the  $f_{\text{QWOA}}(\theta_f)$  were 0.074 (BFGS), 0.086 (LN\_SBPLX), 0.078 (BFGS), 0.06 (BFGS) and 0.073 (Powell). As shown in Figure 18, BFGS was the only algorithm which consistently satisfied the convergence test for both the QAOA and the QWOA. This result, in combination with a relatively low number of associated  $|\theta\rangle$  evaluations, supports the use of BFGS as the default QuOp\_MPI optimisation algorithm.

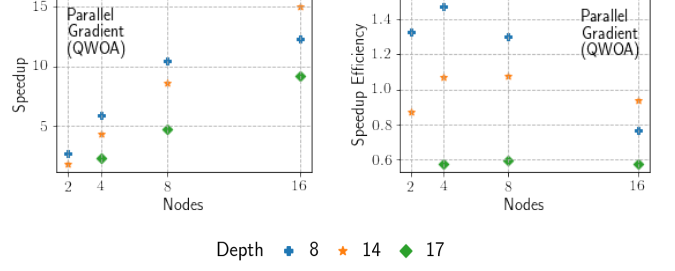


Figure 19: Speedup achieved with parallel computation of  $\nabla_{\theta} f$  for the QWOA at 16 qubits with the  $\text{diag}(\hat{Q})$  and  $\theta_0$  defined as described in Figure 14. Each node introduced an additional COMM- $|\theta\rangle$  sub-communicator with 24 MPI processes.

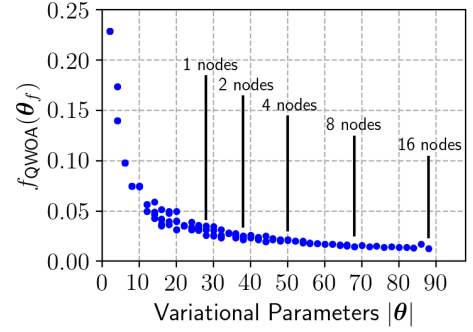


Figure 20: The optimised objective function  $f_{\text{QWOA}}(\theta_f)$  for QWOA simulations as described in Figure 19 using 1, 2, 4, 8 and 16 compute nodes. Markers indicate the maximum number of  $\theta$  simulated for the given number of nodes at a cumulative program wall-time of one hour.

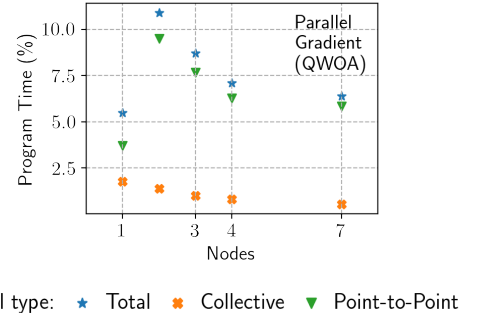


Figure 21: The percentage of program wall-time spent in MPI calls for execution of the QWOA at  $D = 14$  (see Figure 19) as reported by Arm Map version 19.0.1.

The scaling behaviour for parallel computation of the gradient  $\nabla_{\theta} f$  is shown in Figure 19 for simulation of the QWOA algorithm with 18 qubits at  $D = 8, 14, 17$ . This mode of parallelism scales very efficiently; at 16 nodes, there was a maximum speedup of 15.0 with an efficiency of 0.94 ( $D = 14$ ) and a minimum speedup of 9.17 with an efficiency of 0.57 ( $D = 8$ ). As shown by Figure 21, additional nodes resulted in a negligible increase in MPI overhead as communication between sub-communicators consisted of only the updated  $\theta$  and the partial derivatives of  $f_{\text{QAOA}}$ .



The convergence and simulation wall-time of QuOp\_MPI was compared to TensorFlow Quantum (TFQ); a Python package released in 2020 to support research in quantum-classical machine learning. This package was chosen for comparison as it targets a similar userbase through its approachable Python interface, focus on classically parameterised quantum algorithms and performant simulation of the complete wavefunction [18]. TensorFlow Quantum differs from QuOp\_MPI in two key areas. Firstly, it implements a gate-based approach to quantum simulation and, secondly, the package utilises a GPU accelerated library that computes  $|\theta_f\rangle_{\text{QAOA}}$  to single-precision accuracy [25].

The two packages were applied to simulation of the QAOA applied to the max-cut problem for a regular random graph of degree three (see Section 6). Simulations were carried out at  $D = 2$  over two non-identical sets of five  $\theta_0$  for all even  $n$  in [14, 26]. Optimisation was carried out for a maximum of 1000  $f_{\text{QAOA}}$  evaluations under the convergence criteria  $\Delta f_{\text{QAOA}} \leq 10^{-4}$ .

Implementation of QAOA in TFQ built on an example included in the TFQ white-paper [18]. A quantum circuit exactly implementing Equations (12) and (28) was generated using the `tfq.util.exponential` function. This was used to define a Keras model with a single hidden layer for which  $|\psi_0\rangle_{\text{QAOA}}$  was passed to the input layer, and  $f_{\text{QAOA}}(\theta_f)$  was returned by the output layer. The model was trained up to a maximum of 1000 epochs using the ‘Adam’ optimiser, an absolute mean error loss function, and the training data-set ( $|\psi_0\rangle_{\text{QAOA}}, 0$ ) (where 0 is the minimum of Equation (28)). The convergence criteria were implemented using an early-stopping callback function, stopping when the criteria were met over ten successive epochs.

The QAOA was implemented in QuOp\_MPI as shown in Example 1 with the  $\text{diag}(\hat{Q})$  computed in parallel (see Example 3). The L-BFGS-B algorithm provided by SciPy was selected over the BFGS algorithm as it supported specification of the convergence criteria and maximum  $f_{\text{QAOA}}$  evaluations via its `ftol` and `maxfun` options.

Comparison simulations were carried out on a workstation (QuOp\_MPI and TFQ) and the ‘Magnus’ cluster (QuOp\_MPI only). The workstation was equipped with an AMD Ryzen Threadripper 3970X 32-Core Processor at 3.7 GHz, 64 GB of RAM and an Nvidia RTX 3070 GPU. For all trials on the workstation, TFQ offloaded compute to the GPU and QuOp\_MPI ran with one MPI process per CPU core. Trials on the cluster ran on a variable number of nodes that were selected with reference to Figure 14.

Overall, the two packages performed similarly for minimisation of  $f_{\text{QAOA}}$ , with the lowest minima being 5.21 at 14 qubits (TFQ), 6.00 at 16 qubits (QuOp\_MPI), 6.80 at 18 qubits (QuOp\_MPI), 7.82 at 20 qubits (QuOp\_MPI) and 8.21 at 22 qubits (TFQ). QuOp\_MPI had an average  $f_{\text{QAOA}}(\theta_f)$  of 5.56 at 14 qubits, 6.01 at 16 qubits, 7.00 at 18 qubits, 8.09 at 20 qubits and 8.59 at 22 qubits. The TFQ average  $f_{\text{QAOA}}(\theta_f)$  were higher with 6.05 at 14 qubits, 6.70 at 16 qubits, 7.82 at 18 qubits, 8.63 at 20 qubits and 8.93 at 22 qubits. To investigate the source of this discrepancy, two sets of equivalent QuOp\_MPI max-cut simulations were carried out at 12, 14 and 16 qubits over sets of 50  $\theta_0$  with the  $f_{\text{QAOA}}$  returned to single-precision for the first

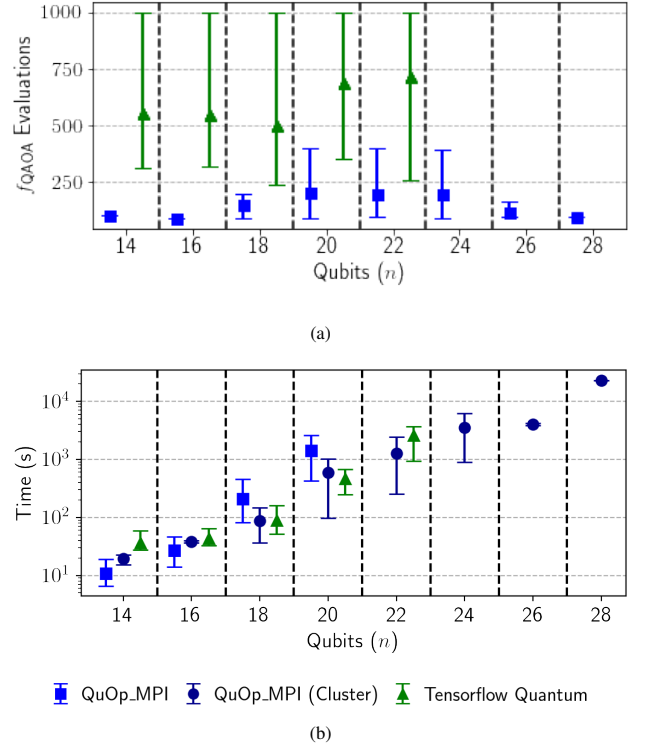


Figure 22: (a) Mean number of  $f_{\text{QAOA}}$  evaluations and (b) mean program wall-time for simulation of the QAOA as applied to the max-cut problem for regular random graphs of degree three using QuOp\_MPI on the workstation, QuOp\_MPI on the ‘Magnus’ cluster and TensorFlow Quantum on the workstation. For (a) and (b) from 14 to 26 qubits the markers depict the mean value over sets of five  $\theta_0$ , the lower bar indicates the set minimum and the upper bar indicates the set maximum. The data shown for 28 qubits is for a single QuOp\_MPI simulation. On the workstation, simulations beyond 20 qubits for QuOp\_MPI and 22 qubits for TFQ were not possible due to memory constraints. The compute node configurations for QuOp\_MPI simulations on the cluster were 12 processes on one node for 14 qubits, 24 processes on one node for 16 qubits, 96 processes on two nodes for 18 qubits, 144 processes on six nodes for 20 qubits, 288 processes on 12 nodes for 22 qubits, 384 processes on 16 nodes for 24 qubits, 432 processes on 18 nodes for 26 qubits and 3360 processes on 140 nodes for 28 qubits.

set and double-precision for the latter. Returning the objective function to double-precision accuracy (the QuOp\_MPI default) resulted in  $f_{\text{QAOA}}(\theta_f)$  that were consistently lower than the  $f_{\text{QAOA}}(\theta_f)$  obtained with a single-precision  $f_{\text{QAOA}}$  (0.47 lower on average). This result indicates that the difference in simulation precision likely contributes to the observed difference in the mean  $f_{\text{QAOA}}(\theta_f)$  between QuOp\_MPI and TFQ.

Figure 22 depicts the mean number of  $f_{\text{QAOA}}$  evaluations and the mean simulation wall-time for QuOp\_MPI and TFQ. Over the range of comparable simulations, QuOp\_MPI requires a smaller number of  $f_{\text{QAOA}}$  evaluations. As such, QuOp\_MPI on the workstation has a simulation wall-time that is close to TFQ at 14 and 16 qubits. The simulation wall-time for TFQ at 18 and 20 qubits is significantly lower than QuOp\_MPI - which is consistent with TFQ’s use of GPU acceleration and lower target precision. At 22 qubits, TFQ had an average wall-time of 2595 s, with the equivalent QuOp\_MPI simulation taking an average of 1267 s on 12 nodes (288 cores). This was

the largest system simulated with TFQ, as simulations beyond this point were not possible due to GPU memory limitations. The distributed-memory parallelism of QuOp\_MPI allowed for simulations beyond 22 qubits with an average wall-time for 24 qubits of 3516 s on 16 compute nodes (384 cores) and, for 26 qubits, 4013 s on 18 nodes (432 cores). A single simulation at 28 qubits had a wall time of 23028 s on 140 compute nodes (3360 cores). Altogether these results demonstrate the utility of the high-precision simulation and scalable distributed memory parallelism of QuOp\_MPI.

## 8. Conclusion

QuOp\_MPI provides a highly scalable and flexible platform for parallel simulation and design of QVAs. As shown by example, researchers can quickly write programs to simulate several previously studied quantum optimisation algorithms, including the QAOA, ex-QAOA, QWOA and QAOAz, which are capable of running efficiently on massively parallel systems.

While this introduction to the package has focused on combinatorial optimisation following a pattern of alternating phase-shift and mixing-unitaries, the flexibility afforded of QuOp\_MPI allows for exploration of QVAs beyond this paradigm. Also not explored has been the application of QuOp\_MPI to the simulation of quantum variational eigensolver algorithms, which, while falling within the simulation framework of QuOp\_MPI, lie outside the immediate research interests of the authors.

Currently, QuOp\_MPI supports the efficient simulation of sparse and circulant mixing operators. While this covers the majority of mixing operators considered in the literature of QVAs, the scope of the package would be improved by the inclusion of a propagation method supporting dense mixing operators and a tensor network backend for the approximation of larger quantum systems. These features are slated for a future update.

## Acknowledgements

This work was supported by resources provided by the Pawsey Supercomputing Centre with funding from the Australian Government and the Government of Western Australia. EM acknowledges the support of the Australian Government Research Training Program Scholarship. The authors would like to thank Sam Marsh, Nicholas Slate, Tavis Bennett, Mark Walker, Burbukje Shakjiri, Andrew Freedland, Zecheng Li, Yuhui Wang and Jianing Sun for their valuable feedback and code testing during the development of QuOp\_MPI.

## References

- [1] D. Matthews, How to get started in quantum computing, *Nature* 591 (2021) 166.
- [2] M. Cerezo, A. Arrasmith, R. Babbush, et al., Variational quantum algorithms, *Nature Review Physics* 3 (2021) 625.
- [3] E. Farhi, J. Goldstone, S. Gutmann, A Quantum Approximate Optimization Algorithm, arXiv:1411.4028 [quant-ph] (2014).
- [4] S. Hadfield, Z. Wang, B. O’Gorman, E. G. Rieffel, D. Venturelli, R. Biswas, From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz, *Algorithms* 12 (2019) 34.
- [5] S. Marsh, J. B. Wang, A quantum walk-assisted approximate algorithm for bounded NP optimisation problems, *Quantum Information Processing* 18 (2019) 61.
- [6] S. Marsh, J. B. Wang, Combinatorial optimization via highly efficient quantum walks, *Physical Review Research* 2 (2020) 023302.
- [7] G. G. Guerreschi, M. Smelyanskiy, Practical optimization for hybrid quantum-classical algorithms, arXiv:1701.01450 [quant-ph] (2017).
- [8] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, J. L. O’Brien, A variational eigenvalue solver on a photonic quantum processor, *Nature Communications* 5 (2014).
- [9] J. Preskill, Quantum Computing in the NISQ era and beyond, *Quantum* 2 (2018) 79.
- [10] D. B. Kell, Scientific discovery as a combinatorial optimisation problem: How best to navigate the landscape of possible experiments?, *Bioessays* 34 (2012) 236.
- [11] F. F. S. Sánchez, C. A. L. Lazo, F. Y. S. Quiñónez, Comparative Study of Algorithms Metaheuristics Based Applied to the Solution of the Capacitated Vehicle Routing Problem, *IntechOpen*, 2020.
- [12] R. Liu, X. Li, K. S. Lam, Combinatorial Chemistry in Drug Discovery, Current opinion in chemical biology 38 (2017) 117–126.
- [13] R. C. Lozano, M. Carlsson, G. H. Blindell, C. Schulte, Combinatorial Register Allocation and Instruction Scheduling, *ACM Transactions on Programming Languages and Systems* 41 (2019) 17:1–17:53.
- [14] H. Markowitz, Portfolio Selection, *J. Finance* 7 (1) (1952) 77–91.
- [15] A. Palczewski, LP Algorithms for Portfolio Optimization: The PortfolioOptim Package, *R J.* 10 (2018) 308–327.
- [16] D. Willsch, M. Willsch, F. Jin, K. Michielsen, H. De Raedt, GPU-accelerated simulations of quantum annealing and the quantum approximate optimization algorithm, arXiv:2104.03293 [physics, physics:quant-ph] (2021).
- [17] E. Matwiejew, QuOp\_MPI (v1.0.0). URL [https://github.com/Edric-Matwiejew/QuOp\\_MPI/releases/tag/v1.0.0](https://github.com/Edric-Matwiejew/QuOp_MPI/releases/tag/v1.0.0)
- [18] M. Broughton, G. Verdon, T. McCourt, A. J. Martinez, J. H. Yoo, S. V. Isakov, P. Massey, M. Y. Niu, R. Halavati, E. Peters, M. Leib, A. Skolik, M. Streif, D. Von Dollen, J. R. McClean, S. Boixo, D. Bacon, A. K. Ho, H. Neven, M. Mohseni, TensorFlow Quantum: A Software Framework for Quantum Machine Learning, arXiv:2003.02989 [cond-mat, physics:quant-ph] (2020).
- [19] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu, S. S. Powers, T. S. Humble, XACC: a system-level software infrastructure for heterogeneous quantum-classical computing, *Quantum Science and Technology* 5 (2020) 024002.
- [20] B. Villalonga, S. Boixo, B. Nelson, C. Henze, E. Rieffel, R. Biswas, S. Mandrà, A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware, *npj Quantum Information* 5 (2019) 1–16.
- [21] P. Crescenzi, V. Kann, R. Silvestri, I. L. Trevisan, Structure in Approximation Classes, *SIAM Journal on Computing* 28 (1999) 24.
- [22] M. Frigo, S. G. Johnson, The Fastest Fourier Transform in the West: (1997).
- [23] M. Frigo, S. Johnson, The Design and Implementation of FFTW3, *Proceedings of the IEEE* 93 (2005) 216–231.
- [24] E. Matwiejew, J. B. Wang, QSW\_mpi: A framework for parallel simulation of quantum stochastic walks, *Computer Physics Communications* 260 (2021) 107724.
- [25] quantum (2021). URL <https://github.com/tensorflow/quantum>
- [26] M. A. Nielsen, I. L. Chuang, Quantum computation and quantum information, 10th Edition, Cambridge University Press, Cambridge ; New York, 2010.
- [27] L. Hales, S. Hallgren, An improved quantum fourier transform algorithm and applications (2000) 515–525doi:10.1109/SFCS.2000.892139.
- [28] L. Zhou, S.-T. Wang, S. Choi, H. Pichler, M. D. Lukin, Quantum Approximate Optimization Algorithm: Performance, Mechanism, and Implementation on Near-Term Devices, *Physical Review X* 10 (2020) 021067.
- [29] J. Nocedal, S. J. Wright, Numerical optimization, 2nd Edition, Springer series in operations research, Springer, New York, 2006.

- [30] E. Jones, T. Oliphant, P. Peterson, SciPy: Open source scientific tools for Python (2001).  
URL <http://www.scipy.org/>
- [31] S. G. Johnson, The NLOpt nonlinear-optimization package.  
URL <http://github.com/stevengj/nlopt>
- [32] D. Steinberg, revrand.  
URL <https://travis-ci.org/github/NICTA/revrand>
- [33] N. Slate, E. Matwiejew, S. Marsh, J. B. Wang, Quantum walk-based portfolio optimisation, Quantum 5 (2021) 513.
- [34] pandas-datareader, version Number: 0.10.0.  
URL <https://github.com/pydata/pandas-datareader/>
- [35] Yahoo Finance – stock market live, quotes, business & finance news.  
URL <https://au.finance.yahoo.com/>



**Edric Matwiejew** is a PhD candidate at The University of Western Australia with the Quantum Information, Algorithms and Simulation (QUISA) Research Centre led by Prof. Jingbo Wang. He develops software for the high-performance simulation of quantum systems, which he applies to the design of quantum algorithms with near-term applications. In

his downtime, he enjoys re-imagining scientific concepts in modular synthesizer design.



**Professor Jingbo Wang** is the Director of the QUISA Research Centre (<https://quisa.tech/>) hosted at The University of Western Australia, leading an active group in the area of quantum information, simulation, and algorithm development. Prof. Wang and her team pioneered quantum walk-based algorithms

to solve problems of practical importance otherwise intractable, which include complex network analysis, graph theoretical studies, machine learning, and combinatorial optimisation. She is currently also the Head of Physics Department, Deputy Head of School of Physics, Mathematics and Computing, and Chair of QST (Quantum Science and Technology) Topical Group under the Australian Institute of Physics.