A Multilevel Framework for Partitioning Quantum Circuits

Felix Burt[1,2], Kuan-Cheng Chen[1,2], and Kin K. Leung[1]

[1]Electrical and Electronic Engineering, Imperial College London, South Kensington Campus, London SW7 2AZ, UK
[2]Imperial Quantum Engineering, Science and Technology Centre (QuEST)

arXiv:2503.19082v4 [quant-ph] 20 Jan 2026

Executing quantum algorithms over distributed quantum systems requires quantum circuits to be divided into sub-circuits which communicate via entanglement-based teleportation. Naively mapping circuits to qubits over multiple quantum processing units (QPUs) results in large communication overhead, increasing both execution time and noise. This can be minimised by optimising the assignment of qubits to QPUs and the methods used for covering non-local operations. Formulations that are general enough to capture the spectrum of teleportation possibilities lead to complex problem instances which can be difficult to solve effectively. This highlights a need to exploit the wide range of heuristic techniques used in the graph partitioning literature. This paper formalises and extends existing constructions for graphical quantum circuit partitioning and designs a new objective function that captures further possibilities for non-local operations via *nested state teleportation*. We adapt the well-known Fiduccia-Mattheyses heuristic to the constraints and problem objective and explore multilevel techniques that coarsen hypergraphs and partition at multiple levels of granularity. We find that this reduces runtime and improves solution quality of standard partitioning. We place these techniques within a larger framework, through which we can extract full distributed quantum circuits including teleportation instructions. We compare the entanglement requirements and runtimes with state-of-the-art methods, finding that we achieve the lowest entanglement costs in most cases. Averaging over a wide range of circuits, we reduce the entanglement requirements by 35% compared with the next best-performing method. We also find that our techniques can scale to much larger circuit sizes than competing methods, provided the number of partitions is not too large.

Felix Burt: f.burt23@imperial.ac.uk
Kuan-Cheng Chen: kuan-cheng.chen17@imperial.ac.uk
Kin K. Leung: kin.leung@imperial.ac.uk

1 Introduction

Distributed quantum computing (DQC) is becoming an increasingly popular paradigm for building scalable quantum computers [1]. The ability to share entanglement among nodes in a network of quantum processing units (QPUs) grants access to the full quantum computational power across the network, at the expense of the additional time and noise overheads incurred by entanglement distribution [2]. Key milestones for facilitating DQC have recently been demonstrated [3–8], while early-stage experimental demonstrations have given credibility to the approach [9–11].

Despite these important achievements, DQC systems face a number of challenges that are not present in traditional, monolithic systems, which must be overcome to achieve the desired scaling. Most notably, the overhead introduced by entanglement distribution is significant. This can lead to deep, slow circuits that require frequent inter-QPU entanglement sharing. Estimates suggest operations requiring shared entanglement are at least an order of magnitude slower than those executed on a single QPU [12]. Coupling high demands for entanglement with high noise and latency can quickly render circuits infeasible to execute.

A growing body of literature is concerned with minimising this additional overhead, using optimised routines for assigning logical qubits to

QPUs and covering resulting non-local operations [13–35]. The resulting optimisation problems promise reductions in entanglement requirements but are typically NP-hard [36], with no polynomial-time solutions for obtaining global optima. However, similar problems have been faced and extensively explored in networking and VLSI circuit design, in which very mature, effective and efficient algorithms for partitioning graphs and circuits have been developed [37–47].

This article aims to combine a number of methods and techniques employed for large-scale hypergraph partitioning and apply them to the unique case of quantum circuit partitioning. The result is a multilevel partitioning framework, inspired by state-of-the-art graph partitioners such as METIS [48], hMETIS [44] and KaHyPar [47], that exploits the particular structure of quantum circuits. The problem is broken down into a number of stages, each of which can be adapted to the particular instance. The stages are *transpilation*, *graph conversion*, *gate grouping*, *coarsening*, *partitioning*, *refinement* and *circuit extraction*.

We use existing transpilation techniques to convert circuits into an appropriate gate set, then convert the resulting circuit to a *temporally-extended* graph representing interaction of qubits over time. We then generalise this graph to a hypergraph by grouping gates together based on their "compatibility" for multi-gate teleportation, i.e., whether or not they can be teleported using the same entanglement resource. We use this hypergraph structure to define a cost function which captures the entanglement cost of the resulting partitioning, considering the possibility of state teleportation, gate teleportation, multi-gate teleportation and a new protocol referred to as *nested state teleportation*, in which we partially collapse gate teleportation procedures into state teleportation. We then design a variant of the Fiduccia-Mattheyses (FM) algorithm [38] that is compatible with the unique constraints of the problem and tweaked to be resistant to local minima. Following this, we introduce a multilevel framework that applies partitioning at varying levels of temporal granularity. We investigate three different strategies for coarsening hypergraphs along the time axis, using the FM algorithm to refine the partitions at each level of granularity. We show that this both improves the time efficiency of the algorithm and improves the partitioning quality in terms of entanglement costs. The final stage produces a distributed quantum circuit, including all necessary instructions for teleportation of qubits and gates between QPUs.

We demonstrate the effectiveness of the framework by comparing the entanglement costs achieved by the best of our multilevel techniques with those obtained by state-of-the-art methods in the literature. Our techniques are shown to be the most effective at reducing entanglement requirements across a wide range of circuit types, including those with high connectivity and depth, while existing methods tend to be effective in a restricted number of cases. On average, we achieve a 35% improvement in the average ratio of entangled pairs of qubits, or *e-bits*, to two-qubit gates compared with the best benchmark method. In most cases, these results are achieved with a lower amount of computation time. Furthermore, the multilevel framework allows the complexity of the problem to be tailored to the available computational resources and can thus be applied to large-scale circuits with hundreds of qubits and gates with proper use of coarsening.

We note that, while the methods investigated here achieve state-of-the-art performance, they operate under an assumption of all-to-all connectivity between QPUs, which may not be feasible in large-scale systems. Additionally, we restrict our investigations to networks with fewer than 20 QPUs, since the partitioning problem scales with the size of the network. However, in follow-up work we tackle both of these issues, adapting the framework to arbitrary network topologies and larger numbers of QPUs [49]. Both the current and follow-up work form the open-source repository `disqco`, which is available on GitHub [50], and can be used to reproduce the results presented here.

2 Background

2.1 Distributed Quantum Computing

Distributed quantum computing (DQC) is concerned with the execution of quantum algorithms across multiple linked quantum processing units (QPUs). While this can feasibly be achieved using classical communication via circuit cutting and circuit knitting [51–58], these represent a near-term form of DQC that is limited by the exponential scaling of the sampling overhead re-

quired to recover the correct results. The alternative is entanglement-based DQC, which requires links capable of sharing purely quantum information among QPUs. This is typically implemented by optical fibres transmitting photonic qubits [59]. Qubit transmission over fibres is used to distribute pairs of entangled qubits between geographically separated locations. Shared entangled pairs are known as *e-bits* or *EPR pairs* and represent a consumable resource required for distributed quantum computation.

Two QPUs holding a shared e-bit can use local operations and classical communication (LOCC) to teleport the state of a qubit that is actively involved in a computation from one QPU to the other [60]. This is referred to as *state teleportation*. Alternatively, the e-bit can directly moderate a non-local controlled-unitary operation [61, 62], analogously referred to as *gate teleportation*. Gate teleportation has been shown to be a flexible procedure, that can cover multiple non-local operations using the same shared e-bit [22, 62, 63], creating a large design space for optimising the entanglement requirements of distributed quantum circuits.

2.2 Non-local gate coverage

Non-local gate coverage is a key target for optimisation in DQC. Any method of dividing or partitioning a quantum circuit results in some number of non-local operations. We can formalise this by defining a set of QPUs as Q, and a set of logical qubits q. Partitioning a quantum circuit requires making a *partition assignment* $\phi : q \to Q$ which maps logical qubits to QPUs. For any two-qubit gate, defined partly by the qubits on which it operates as $g(q_i, q_j)$, the gate is non-local under ϕ if $\phi(q_i) \neq \phi(q_j)$. The cost of a partitioning is thus closely related to the non-local operations present in the circuit. However, since there are many ways to cover non-local operations, the cost of a partitioning does not directly correspond to the number of non-local operations. Rather, it is a quantity that depends on the method chosen for covering non-local operations.

In general, a given non-local operation can be covered in three different ways. First, state teleportation can be used to teleport the state of a qubit from one QPU to another, modifying the assignment function ϕ, allowing the gate to be executed locally. Second, if the gate is a controlled-unitary operation, it can be executed remotely using gate teleportation, a procedure that entangles the control qubit with a communication qubit in the target QPU, then uses the linked communication qubit to control the application of the unitary. Both methods require a single e-bit, though state teleportation may modify the locality of future gates in the circuit. Third, when certain compatibility conditions are met, separate gate teleportation procedures can be merged into a single gate teleportation, which consumes just one e-bit [22]. This has been referred to as the EJPP protocol [22], burst communication [23], extended gate teleportation [64], or simply teleportation [65]. For simplicity, we refer to the full process of teleporting multiple gates within the same process as *multi-gate teleportation*.

2.2.1 Multi-gate teleportation

Gate teleportation uses a shared e-bit to link a control qubit with a remote communication qubit. The process creates a partially accessible clone of the control qubit at multiple locations, allowing compatible two-qubit gates to be performed using the communication qubit. *Multi-gate* teleportation refers to the process of using a single e-bit to teleport multiple gates. This is based on the fact that certain operations kill the link between the control and the communication qubit, while certain others do not. A full multi-gate teleportation procedure can be broken up into three stages.

First, a qubit is entangled with a communication qubit at a distance via a shared e-bit. Following this, a series of remote operations is performed using the communication qubit as control. When all compatible gates are complete, the qubit is then disentangled from the communication qubit. The first and final steps have been referred to as the *cat-entangler* and *cat-disentangler* primitives [65], since the state of the communication qubit mirrors the state of the original qubit, a situation reminiscent of Schrödinger's cat. For consistency with previous work, we adopt the terminology and notation introduced by Wu et al. [22], calling the multi-gate teleportation primitives the *entanglement-assisted starting and ending processes*. These processes are illustrated in Fig. 1.

Mathematically, the starting and ending processes are described as linear maps on general

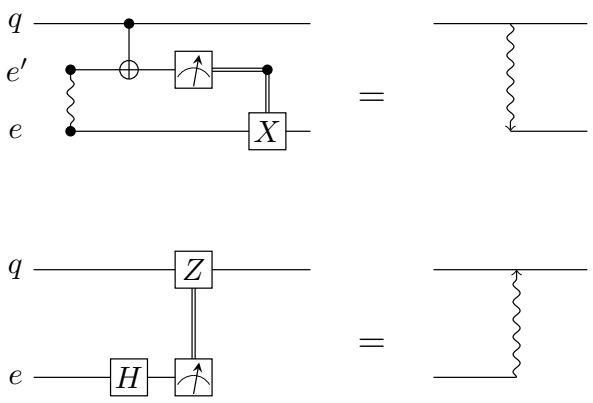


Figure 1: The starting and ending processes. The starting process $S_{q,e}$ is a linear map which maps the state of the root qubit q onto a joint state between q and an auxiliary qubit e in another QPU. The ending process $E_{q,e}$ applies the inverse map, disentangling the auxiliary qubit from the root qubit, returning the original state to q. Diagrammatic notation from Ref. [22]

quantum states that compose to form the identity. The starting process $S_{q,e}$ maps an input state $|\psi\rangle_q$ for some root qubit q on Hilbert space $\mathcal{H}_q$ onto a joint state between q and an auxiliary qubit e in another QPU:

$$\begin{aligned} S_{q,e}(|\psi\rangle) &= \langle 0|_q |\psi\rangle_q |0\rangle_q |0\rangle_e + \langle 1|_q |\psi\rangle_q |1\rangle_q |1\rangle_e \\ &= CX_{q,e} |\psi\rangle_q |0\rangle_e , \end{aligned} \tag{1}$$

which has the effective action of a CX gate between q and e, causing a 'fan-out' of the state of qubit q onto e, facilitated by the shared e-bit. The symmetry of the resulting state in Eq. (1) means that both the root qubit q and the communication qubit e behave the same when used as a control in a controlled-unitary operation. This symmetry is preserved until a non-diagonal single-qubit gate is applied to either qubit, that mixes the $|0\rangle$ and $|1\rangle$ basis components of the state. To ensure that we regain the correct state on q, the gate teleportation ending process must be applied before any gate that is not diagonal or anti-diagonal in the computational basis of the root qubit is applied [22]. The ending process, $E_{q,e}$, applied to the state $|\psi'\rangle_{q,e} = S_{q,e}(|\psi\rangle_q)$, applies the inverse map of Eq. (1):

$$\begin{aligned} &E_{q,e}(|\psi'\rangle_{q,e}) \\ &= Tr_e(CX_{q,e} |\psi'\rangle_{q,e} \langle\psi'|_{q,e} CX_{q,e}), \end{aligned} \tag{2}$$

where $Tr(|i\rangle \langle j|) = \delta_{ij}$ and Tr_k is the partial trace on qubit k, which removes it from the state description. As a result, $E_{q,e} \circ S_{q,e}(|\psi\rangle_q) = |\psi\rangle_q$ [22]. This has the effect of a 'fan-in' operation, disentangling the auxiliary qubit e from the root qubit q, returning the original state to q.

If we want to perform a controlled unitary $CU_{q,q'}$, where

$$\begin{aligned} CU_{q,q'} &= |0\rangle_q \langle 0|_q \otimes I_{q'} + |1\rangle_q \langle 1|_q \otimes U_{q'} \\ &= \begin{pmatrix} I_{q'} & 0 \\ 0 & U_{q'} \end{pmatrix}, \end{aligned} \tag{3}$$

this is equivalent to performing

$$E_{q,e} \circ CU_{e,q'} \circ S_{q,e}(|\psi\rangle_q) = CU_{q,q'} |\psi\rangle_q . \tag{4}$$

If two contiguous gate teleportation processes share the same root and auxiliary qubits, q and e, then they can be merged into a single, multi-gate teleportation procedure that uses the same starting and ending processes. As described in Ref. [66], this can be straightforwardly generalised to link multiple QPUs.

A k-fold starting process $S_{q,\mathbb{E}}$ rooted on q acting on a set of auxiliary qubits $\mathbb{E}$ spanning k QPUs, can be described as:

$$\begin{aligned} S_{q,\mathbb{E}}(|\psi\rangle) &= \langle 0|_q |\psi\rangle_q |0\rangle_q \bigotimes_{e\in\mathbb{E}} |0\rangle_e \\ &\quad + \langle 1|_q |\psi\rangle_q |1\rangle_q \bigotimes_{e\in\mathbb{E}} |1\rangle_e \\ &= \prod_{e\in\mathbb{E}} CX_{q,e} |\psi\rangle_q \bigotimes_{e\in\mathbb{E}} |0\rangle_e , \end{aligned} \tag{5}$$

which has the effect of fanning out the state of q onto all auxiliary qubits in $\mathbb{E}$.

Without any network constraints, n e-bits allows n QPUs to be linked to the root qubit q. Once the link state is active, it can be used to control any number of contiguous controlled-unitary operations in each linked QPU via merging the teleportation procedures:

$$\begin{aligned} &\prod_{q'\in\mathbb{Q}} U_{q,q'} |\Psi\rangle = E_{q,\mathbb{E}} \circ (\\ &\prod_{q',e\in\mathbb{Q}\times\mathbb{E}} (U_q \otimes I_e \otimes U_{q'})(I_q \otimes CU_{e,q'})) \circ S_{q,\mathbb{E}}(|\Psi\rangle), \end{aligned} \tag{6}$$

where $|\Psi\rangle$ is the state across the full Hilbert space of the root qubit q and all $q' \in \mathbb{Q}$, which are the *receivers* in the gate teleportation. The starting and ending processes, however, still act only on the subspace of the root qubit and the auxiliary

qubits. A *compatible* two-qubit unitary is given by

$$\prod_{q' \in \mathbb{Q}} U_{q,q'} = \prod_{q',e \in \mathbb{Q} \times \mathbb{E}} (U_q \otimes U_{q'}) C U_{q,q'}, \quad (7)$$

for $\theta = 2n\pi$ (diagonal) or $\theta = (2n+1)\pi$ (anti-diagonal) in the general single-qubit unitary $U(\theta, \phi, \lambda)$ Eq. (15) on q. In the anti-diagonal case, $U_e = X_e$ is applied to account for the flipping of the basis components caused by U_q. Otherwise, $U_e = I_e$. Eq. (7) omits single-qubit and two-qubit gates acting on qubits in Q which are independent of the root qubit q, since there are no restrictions on such gates for multi-gate teleportation. Fig. 2 contrasts the coverage of a non-local operation using the starting and ending processes with a state teleportation procedure.

An important point to note is that the most effective method for covering non-local operations depends greatly on circuit structure, and the choice of method can have a significant impact on the resulting entanglement cost, quantified by the number of e-bits required. For example, methods that depend entirely on state teleportation, such as the fine-grained partitioning approach of Baker et al. [17], are very effective for quantum volume and arithmetic circuits, while they tend to be less effective for circuits with less regular connectivity and high potential for multi-gate teleportation [17, 64]. Alternatively, multi-gate teleportation methods are effective for circuits densely packed with two-qubit gates, since many opportunities arise for merging teleportation procedures together [66].

2.2.2 Nested state teleportation

Since both state teleportation and gate teleportation are effective in different cases, it is desirable to exploit both methods. Beyond just treating them as separate options for covering non-local operations, we may also gain benefit from combining them into the same procedure.

First note an interesting feature of the above formulation of gate teleportation – the starting process and the ending process are identical to those used in state teleportation, except the indices for the ending process are flipped. Accordingly, we may define a state teleportation procedure from q to e as:

$$E_{e,q} \circ S_{q,e}(|\psi\rangle_q) = |\psi\rangle_e, \quad (8)$$

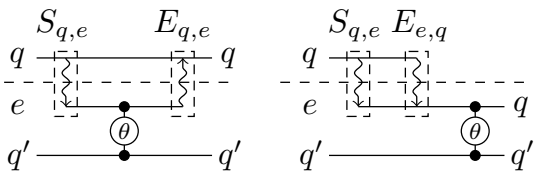


Figure 2: Methods for covering a non-local operation. On the left is the entanglement-assisted starting and ending processes $S_{q,e}$ and $E_{q,e}$, used for a gate teleportation. On the right, we flip the direction of the ending process to $E_{e,q}$, which allows us to teleport the state of q onto e. The two circuits are equivalent up to relabelling of qubits after the ending process.

where we have exchanged the indices q and e, meaning that q is measured, and the classically controlled operation is applied, such that the state of q is teleported onto e. In Fig. 2, we use the same diagrammatic notation to indicate a state teleportation built out of the starting and ending processes. What is interesting about this is that it indicates we may always collapse a gate teleportation procedure into a state teleportation by redirecting the ending process away from the QPU of the root qubit. If we perform a k-fold starting process $S_{q,\mathbb{E}}$ on q and a set of auxiliary qubits $\mathbb{E}$, we create the state in Eq. (5). To collapse the state onto $\tilde{e}$ we perform the following

$$E_{\tilde{e},\tilde{\mathbb{E}}} \circ S_{q,\mathbb{E}}(|\psi\rangle_q), \quad (9)$$

where $\tilde{\mathbb{E}} := (\mathbb{E} \setminus \tilde{e}) \cup \{q\}$, such that we replace q with $\tilde{e}$ in the ending processes for all other auxiliary qubits, and interchange the indices q and $\tilde{e}$ for the ending process corresponding to $S_{q,\tilde{e}}$. In Fig. 3, we illustrate a situation where this is of use. We refer to this as *nested state teleportation*, since we are effectively nesting a state teleportation inside a k-fold multi-gate teleportation process. If a qubit has been linked to many QPUs, ending processes can be successively composed to induce multiple nested state teleportations, until the final state is collapsed onto a single auxiliary qubit.

This tells us that choosing between state and gate teleportation reduces to choosing the direction of the ending processes of a multi-gate teleportation procedure. Covering non-local operations always requires a starting process, but the character of the teleportation is determined by the ending process.

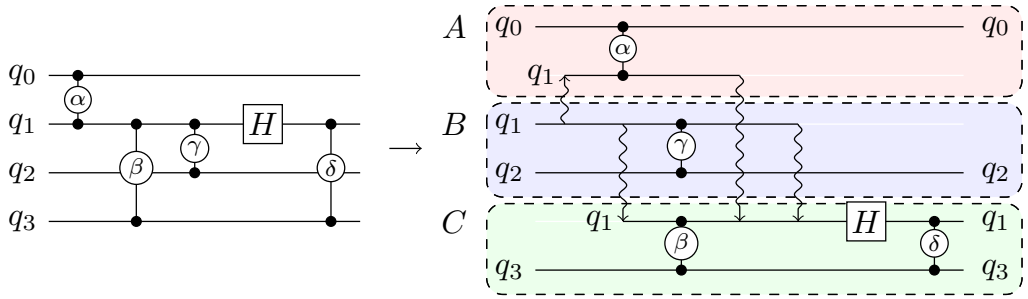


Figure 3: The nested state teleportation procedure. A 2-fold starting process is applied to entangle q_1, located in QPU B, with auxiliary qubits in A and C. This fans the state out, allowing $CP(\theta)$ gates with q_0 and q_3 to be executed, as well as the local gate with q_2. The H gate on q_1 requires the gate teleportation to be ended. Since there is another gate between q_1 and q_3, all ending processes should be re-routed towards C, collapsing the final state onto the auxiliary qubit. In fact, for each ending process, we may route the ending process to any QPU that still has an active link, provided the final ending process ends up at our chosen destination. The circuit on the right is equivalent to the one on the left up to a relabelling of qubits, as shown at the end of the wires.

2.3 Generalised Circuit Partitioning

Generalised circuit partitioning (GCP) was introduced in Burt et al. [64], with the aim of creating an optimisation framework that considers both multi-gate and state teleportation methods when partitioning quantum circuits. Here we recap the basics and provide some extensions and clarifications on the notation.

GCP uses a graphical picture of quantum circuits, with a spatial and a temporal component, which is extended to a hypergraph construction.

Consider a circuit $\mathcal{C}$, consisting of d layers of gates $\mathcal{L} = \bigcup_t^d \mathcal{L}^{(t)}$, where each t corresponds to an independent time step of the circuit. Each $\mathcal{L}^{(t)}$ contains the single and two-qubit gates for time step t of the circuit, where each gate is defined by the qubits on which it acts, which are elements of the set of logical qubits Q_L, and its parameters. The total number of logical qubits is $n_q = |Q_L|$.

The graph, $H(V, E)$, consists of $n_q d$ nodes, where each node is associated with a qubit and a time step. We can denote this using a time map $\tau : V \to \{1, ..., d\}$, and a qubit map $\kappa : V \to Q_L$, where d is the total depth of the circuit, i.e., the number of independent time steps, including initialisation and measurement. We use the shorthand $v_q^{(t)}$ to denote a node v for which $\tau(v) = t$ and $\kappa(v) = q$. For convenience, we refer to the *temporal* graph (or hypergraph), as $H(V, E; \tau, \kappa)$. The total set of nodes V can be considered as the union of d subsets $V^{(t)}$ of nodes

$$V = \bigcup_{t=1}^{d} V^{(t)}, \tag{10}$$

where

$$V^{(t)} = \{v \mid \tau(v) = t, \forall i \in Q_L\}. \tag{11}$$

The set of edges can be considered to be the union of two separate sets, the state edges E_s and the gate edges E_g.

$$E = E_s \cup E_g, \tag{12}$$

$$E_s = \{(v, u) \mid \kappa(v) = \kappa(u), \tau(u) = \tau(v) + 1 \quad \forall i \in Q_L, \forall t \in \{1, ..., d-1\}\} \tag{13}$$

$$E_g = \{(v, u) \mid (i, j) \in \mathcal{L}^{(t)}, \quad \forall t \in \{1, ..., d\}\} \tag{14}$$

For simplicity, the graph is built using a general gate set consisting of the following two gates:

$$U(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i(\phi+\lambda)}\sin(\theta/2) & e^{i\phi}\cos(\theta/2) \end{pmatrix}, \tag{15}$$

$$CP(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}. \tag{16}$$

The gate set has a number of advantages. It allows us to compress many single-qubit gates together into one single U – giving us smaller problem instances – while also allowing us to identify gates only by their respective parameters (three for each single-qubit gate and one for each two-qubit gate). Single-qubit gate parameters can be

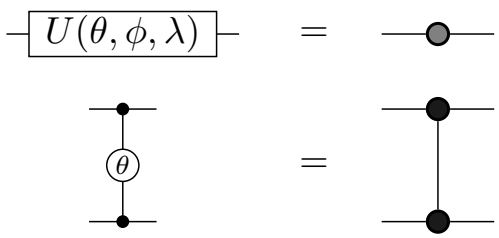


Figure 4: Correspondence between single and two-qubit gates and temporal hypergraphs. We represent any single-qubit gate as a grey node, storing the parameters θ, ϕ, λ as node attributes. Two-qubit $CP(\theta)$ gates are represented as an edge between two black nodes, storing the phase parameter θ as an edge attribute.

stored as node attributes while the two-qubit gate parameters can be stored as edge attributes, making the process of circuit extraction straightforward. Furthermore, the $CP(\theta)$ gate is symmetric, which means both qubits can play the role of the root in a gate teleportation procedure [15]. In contrast, for a CX gate, only the control qubit can be the root. This allows for many more gate grouping possibilities.

Furthermore, common two-qubit gates such as CZ and CX can be constructed from a single $CP(\theta)$ gate and single-qubit unitaries, whereas an arbitrary $CP(\theta)$ gate requires two CX gates and single-qubit unitaries.

While it is possible to construct the graph using other gate sets, this often results in more two-qubit gates and smaller gate groups – both likely contributors to higher e-bit cost. At present, the construction is not directly compatible with multi-qubit gates, such as the Toffoli, though we believe the extension would be straightforward.

A circuit can be converted to and from our desired gate set using standard transpilation techniques [67, 68]. The correspondence between nodes and edges and gates is illustrated in Fig. 4.

An example of the base temporal graph is shown in Fig. 5a. At this stage, partitioning the nodes corresponds to choosing a set of state and gate teleportation operations to cover the non-local gates in the circuit, where each *cut* edge indicates the use of an e-bit. A cut state-edge corresponds to state teleportation, and a cut gate-edge to gate teleportation (see Fig. 5b). However, we would like to consider opportunities for multi-gate teleportation. Typically, circuit partitioning methods choose to allocate qubits before merging teleportation procedures, separating the problem into distinct components [22, 23, 69]. An exception to this is the original hypergraph partitioning strategy of Andres-Martinez and Heunen [15], that is also used in Ref. [66], which also chooses gate groups in the process of partitioning. Since we would like to partition qubits over time, but also consider multi-gate teleportation, we group gates in advance of partitioning, by choosing a subset of all possible gate groups that are merged into hyper-edges before partitioning. Grouped gates may end up as local or non-local after the partitioning. Thus, we partition the graph with prior knowledge of how gates will be grouped together if they are non-local, allowing us to efficiently determine the resulting e-bit cost. The limitation of this is that we must choose a strategy for grouping gates beforehand, which may turn out to be sub-optimal for the resulting partitions. We conjecture that this is still more effective than the reverse procedure, in which we choose the partition first, which may turn out to be sub-optimal for the best resulting set of gate teleportations. We note that it is only because we are using symmetric two-qubit gates that multiple grouping options exist – the choice arises from the freedom in choosing which qubit is the root of the gate teleportation. We elaborate on this in the following Section 2.3.1.

To ensure that the partitioning objective directly corresponds to the e-bit cost, we need to design a unique cost function for the problem. This is also described in Section 2.3.1, ultimately resulting in Eq. (19).

2.3.1 Gate grouping

Gate grouping requires conditions for teleportation compatibility, as described in Section 2.2, as well as a rule for choosing which gates should be added to which groups where there are multiple possibilities. Multiple gates can be considered a teleportation-compatible group if:

1. Two-qubit gates share a common control qubit.
2. Gates are contiguous on the common control qubit.
3. Single-qubit gates on the common control are *diagonal* or *anti-diagonal*.

The common control qubit is assigned as the "root" qubit of the group, while the other qubits

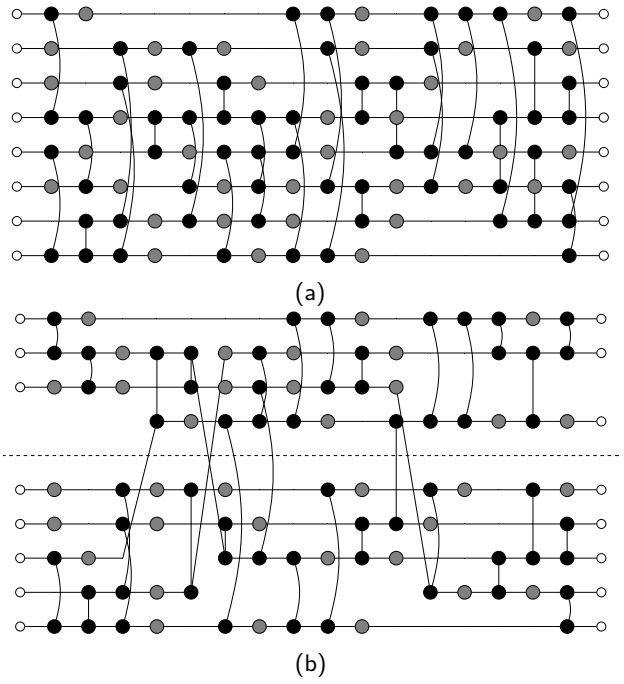


Figure 5: Base graph for a random, 8-qubit circuit. Grey nodes correspond to single-qubit gates, while black nodes correspond to qubits involved in $CP(\theta)$ gates, as illustrated in Fig. 4. State edges connect nodes representing the same qubit at different time steps. In Fig. 5b, an optimised partitioning of the nodes is illustrated by dragging nodes vertically into different QPU regions. Cut state edges correspond to state teleportation, while cut gate edges correspond to gate teleportation.

are assigned as "receiver" qubits. Recall that, since $CP(\theta)$ gates are symmetric, both qubits can be considered to be a control, making gate teleportation possible in both directions. This means that either qubit can serve as the root. However, once we have assigned an edge to a group as a receiver, we do not consider other possible groups rooted on that edge. This means that adding edges to groups is order-dependent, and we must define a strategy for choosing which group to add an edge to when multiple options exist. Note that this choice only arises due to the symmetric two-qubit gates. If instead we used an asymmetric two-qubit gate such as CX, the starting process can only be rooted on the control qubit. This means that maximising the size of the gate groups is trivial and can be achieved by merging all contiguous control chains together until there are no more possibilities. So while using symmetric gates increases the number of possible groupings, it also adds a new degree of freedom into the problem that requires a strategy to resolve.

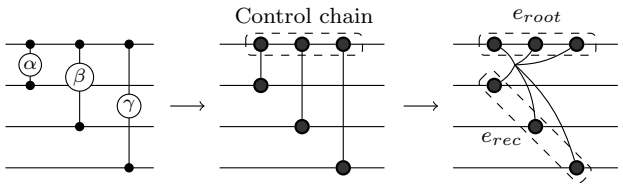


Figure 6: Identification of a compatible gate group. The $CP(\theta)$ (left) gates share a common control qubit, are contiguous on the control qubit, and there are no incompatible single-qubit gates on the control qubit between the two-qubit gates. The circuit is first converted to a temporal graph (middle), then the compatible-edges are grouped into a hyper-edge (right).

To define the groups, we use the routine described in Algorithm 1. In essence, the algorithm works by scanning the graph from time step $t = 0$ to d, identifying potential groupings, and adding identified gates to the largest existing group with which it is compatible. If instead we used the asymmetric CX gate, there would only be one compatible group (rooted on the control qubit), so this choice would not arise. When encountering an incompatible single-qubit gate (non-diagonal) on the root of a group, the corresponding group must be closed. For each identified group, we replace all gate edges with a hyper-edge object identified by the *first root* node (i.e., the node $v_i^{(t)}$, where t is the time step of the first gate in the group).

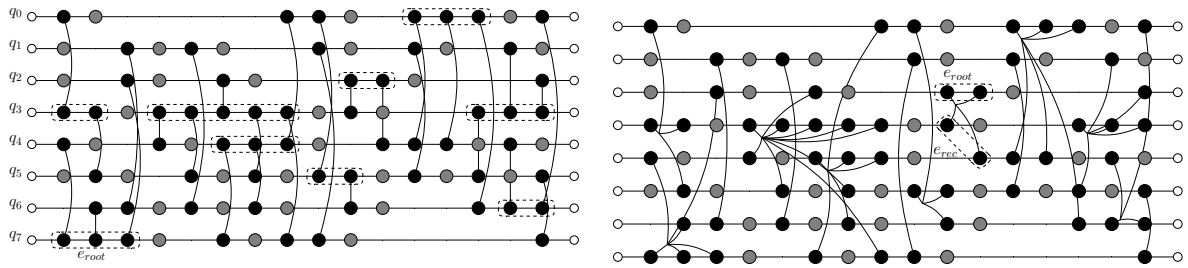
The hyper-edge, in this case, is defined by two sets of nodes referred to as the *root set* and the *receiver set*.

Formally, for each edge

$$e = e_{root} \cup e_{rec}, \tag{17}$$

where $e_{root}, e_{rec} \subset V$. This distinction makes the hyper-edges akin to *directed* hyper-edges, where the root set plays the role of the "tail", and the receiver set plays the role of the "head" [70].

The root set contains each node associated with the root qubit from the time step of the first gate in the group to the time step of the last gate in the group, while the receiver set consists of all gate-like edges stemming from the root control node. Defined this way $|e_{root}| \geq |e_{rec}|$ for all groups, since the partition assignment of the root at each time step may affect the cost of the group, while for the receivers we are only concerned with the assignment at the time step of the gate. We show an example of the hyper-edge construction in Fig. 6.



(a) (b)

Figure 7: Illustration of grouping gate edges into hyper-edges. Fig. 7a shows the root sets identified using Algorithm 1. The root set is the set of nodes corresponding to the root qubit of the group, i.e. the qubit which will be linked to receiving QPUs via the starting process. Fig. 7b shows the resulting hyper-edges after grouping. Each hyper-edge contains a root set and a receiver set.

The definition of the group in terms of the root and receiver set provides a simple mechanism for calculating the cost of a partitioned hyper-edge, despite the nodes representing different points in time. To see this we must first define an assignment function $\Phi : V \to Q$ mapping nodes to the set of QPUs, thus partitioning the nodes into disjoint subsets which, for the i^{th} QPU, we may call P_i. The cost of each hyper-edge must correspond to the entanglement cost associated with covering all gates in the group given the assignment of the qubits. If the root qubit remains in its starting QPU for the duration of the group, then this corresponds to the standard hyper-cut, given by one less than the number of partitions spanned by the hyper-edge. However, if a state edge between two nodes in the root set is cut, this indicates a nested state teleportation. For each partition k which appears in the set $\{\Phi(v) \mid v \in e_{rec}\}$, a starting process must be performed. However, for each partition in $\{\Phi(u) \mid u \in e_{root}\}$, we have a corresponding ending process routed to $\Phi(u)$ which collapses the starting process to $\Phi(u)$. If we have only one partition in the root set, then we simply collapse the state back to the root qubit at the end of the group, completing a standard multi-gate teleportation. For each additional partition in the root set, we collapse the state by re-routing the ending process to this partition. The e-bit cost is thus accounted for by the cut state-edge, rather than the hyper-edge.

When the group is complete, the state of the root qubit will be located at $\Phi(u')$, where u' corresponds to the final root node in the group. This allows us to define the hyper-edge cost as the length of the set of partitions in the receiver set that are not present in the root set, i.e. the number of partitions that are not accounted for by nested state teleportation. We write this as

$$c_e(\Phi) = |\{\Phi(v) \mid v \in e_{rec}\} \setminus \{\Phi(u) \mid u \in e_{root}\}|. \tag{18}$$

Since the standard state and gate edges are special cases of hyper-edges where the distinction between root and receiver sets is arbitrary, we define all edges this way, ensuring that state and gate edges contain one node in each of the root and receiver sets. For state edges, we use the convention that the root set contains the node at time t and the receiver set contains the node at time $t + 1$. For symmetric gate edges, we assign the qubit with the lower index as the root. This allows us to use Eq. (18) to calculate the cost of all edges in the graph.

This allows us to define the partitioning objective

$$\min_{\Phi} \sum_{e \in E} c_e(\Phi), \tag{19}$$

that we refer to as the *entanglement cost*, since it is designed to correspond to the number of e-bits required to partition the circuit.

Eq. (19) can be written in a number of different forms – the form presented is appropriate under the assumption of all-to-all connectivity. In the case of limited connectivity, the cost function must be decomposed such that each contribution is scaled by a path dependent function which describes the auxiliary entanglement requirements for entanglement swapping and purification. We restrict to all-to-all connectivity as a base for demonstrating the capability of the framework, though we handle the more complex case in follow-up work, where we instead optimise over trees connecting root to receiver QPUs in the network [49].

Algorithm 1: GreedyGrouping

Input: $H = (V, E; \tau, \kappa)$: Temporal hypergraph, with time map $\tau : V \to \{1, \ldots, d\}$ and qubit map $\kappa : V \to Q_L$.

Output: $H' = (V, E')$: A new hypergraph with merged hyper-edges.

$H' \leftarrow H$, $E' \leftarrow E$;

`// Initialize prospective groups.`
foreach $\ell \in Q_L$ **do**
  $R_\ell \leftarrow \emptyset$, $S_\ell \leftarrow \emptyset$;
  $\mathcal{G}[\ell] \leftarrow (R_\ell, S_\ell)$;

for $t' \leftarrow 1$ **to** d **do**
  $V_{t'} \leftarrow \{v \in V \mid t(v) = t'\}$;
  $E_{g,t'} \leftarrow \{e \in E_g \mid t(u) = t(v) = t'\}$;
  `// 2-qubit gates at time` t'
  foreach $e = \{u, v\} \in E_{g,t'}$ **do**
    $\ell_u \leftarrow q(u)$, $\ell_v \leftarrow q(v)$;
    $E' \leftarrow E' \setminus e$;
    $s_u \leftarrow |R_{\ell_u} \cup S_{\ell_u}|$;
    $s_v \leftarrow |R_{\ell_v} \cup S_{\ell_v}|$;
    if $s_u \geq s_v$ **then**
      $S_{\ell_u} \leftarrow S_{\ell_u} \cup \{v\}$;
    else
      $S_{\ell_v} \leftarrow S_{\ell_v} \cup \{u\}$;
    `// Clear incompatible groups`
    foreach $\ell \in Q_L$ **do**
      if $\neg$Compatible(ℓ, ℓ_u, ℓ_v) **then**
        $R_\ell \leftarrow \emptyset$;
        $S_\ell \leftarrow \emptyset$;
  `// Single-qubit or idle nodes`
  $V_{t'}^{(1)} \leftarrow \{\, v \in V_{t'} \mid v$ not in any $e \in E_{t'}^{(2)} \,\}$;
  foreach $v \in V_{t'}^{(1)}$ **do**
    $(\theta_v, \phi_v, \lambda_v) \leftarrow$ parameters of v;
    $\ell \leftarrow q(v)$;
    if $\theta_v \notin \{0, \pi, 2\pi\}$ **then**
      `// Non-diagonal: terminate group`
      if $|S_\ell| > 0$ **then**
        $e' \leftarrow (R_\ell, S_\ell)$;
        $E' \leftarrow E' \cup e'$ to E';
      $R_\ell \leftarrow \emptyset$;
      $S_\ell \leftarrow \emptyset$;

return (V, E');

The constraints on Φ are determined by the qubit capacity of the QPUs. Since we have a set of nodes for each time step, we call the subset of nodes in partition i at time t $P_i^{(t)}$. Each $P_i^{(t)}$ should be a subset of $V^{(t)}$, giving us a sequence of partitions covering each t. This allows us to define the balance constraints according to the data qubit capacity of each QPU, which cannot be exceeded at any time step of the circuit.

$$|P_i^{(t)}| \leq |Q_i|, \forall i \in \{1, ..., K\}, \forall t \in \{1, ..., d\}, \tag{20}$$

where $K = |Q|$ is the number of QPUs, corresponding to the number of partitions. In other words, there is a set of balance constraints to satisfy. For each time step t, there are K constraints, one for each QPU.

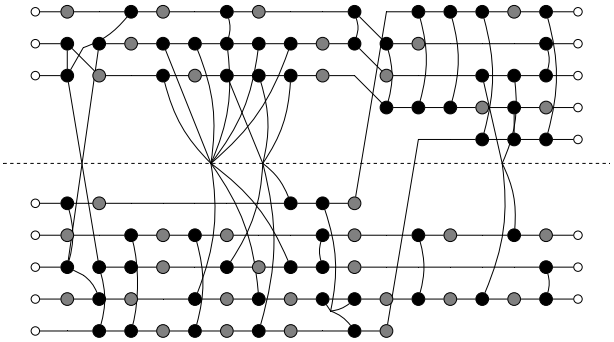


Figure 8: Graph from Fig. 7b after partitioning. For visual clarity, we return all local contributions to hyper-edges to their original gate edges, such that only non-trivial hyper-edges are shown. Note that some hyper-edge contributions which still appear to be local indicate that there has been a nested state teleportation. For such an edge, a gate teleportation procedure is started and then converted to a state teleportation after the final gate in the group.

2.4 Fiduccia-Mattheyses

The Fiduccia-Mattheyses algorithm is an efficient, widely studied graph partitioning heuristic [38], which has a time complexity that is linear in the number of nodes and can effectively handle more complex structures such as hypergraphs. The algorithm leverages efficient data structures for storing and updating the net cost of moving nodes between partitions, avoiding redundant calculations. Balance constraints can be straightforwardly enforced by avoiding moves that violate the conditions or using a tolerance. Given an initial bipartitioning of the nodes, a single pass of the algorithm is executed as follows:

1. Calculate the cost of moving each node to the other partition.
2. Sort all moves into *gain buckets*, where each bucket contains all moves resulting in a particular gain or loss in total cost.
3. Choose a node from the highest (lowest) gain bucket and move it to the other partition.
4. Remove the chosen node from the bucket and update the gains of all neighbouring nodes, moving them to their new buckets.
5. Repeat steps (3-4) until no nodes are left in the bucket.
6. Roll back to the partitioning at the iteration of maximum (minimum) net gain.

The choice of maximising or minimising gain is a matter of convention, depending on whether gain is defined as a reduction or increase in cost. For the rest of this work, we use the negative gain definition, defining gain as the cost after the move minus the cost before the move.

The pass is repeatedly executed up until no more reduction is achieved or a pass limit is reached. Storing the gains of all node exchanges and only updating the neighbours results in a time complexity $\mathcal{O}(n)$ which is linear in the number of nodes n, provided the gain updates are performed in constant time. This relies on the number of neighbours of each node being bounded by a constant. When dealing with problems requiring k partitions, the algorithm can be modified by calculating gains for moving each node to each external partition, such that the number of node exchanges queried increases from n to $n(k-1)$, resulting in a time complexity $\mathcal{O}(kn)$.

3 Quantum Circuit Partitioning with Fiduccia-Mattheyses

The fundamental differences between the quantum circuit case and standard balanced min-cut graph partitioning lie in the balance constraints and the objective. Since FM extends naturally to hypergraphs and handles balance constraints easily, it is a strong candidate to be adapted for the temporal hypergraph construction. In particular, temporal graphs give balance constraints that are defined by the data qubit capacity of each QPU, which must be satisfied for each time step of the circuit, resulting in d separate balance constraints per graph.

The objective also differs: the cost function for quantum circuits is designed to minimise e-bit use, so the cost of moving nodes between partitions must be defined in terms of its effect on Eq. (19). As a result, although the overall structure of the FM algorithm remains largely unchanged, new conditions are required for enforcing balance, and the computation of move costs must be handled differently. Below we describe how move costs can be computed efficiently for the quantum circuit partitioning problem.

First we define $A(v) \subset E$ as the adjacency set of node v, containing all edges and hyper-edges in which v is present. Contrast this with the set of neighbouring nodes, $N(v) \subset V$, which is the set of all nodes present in the edges in $A(v)$. Each edge in $A(v)$ has a contribution to the cost under assignment $\Phi(v)$. When a node is moved, $\Phi(v)$ updates to $\Phi(v)'$. The change in cost for each e in $A(v)$ can be calculated simply as

$$\delta_e(\Phi, \Phi') = c_e(\Phi') - c_e(\Phi). \tag{21}$$

We refer to this as the *gain contribution* from e, adhering to the convention that a negative gain decreases the cost. The total *gain* of moving node v is thus the sum of the gain contributions from all adjacent edges:

$$g_v(\Phi, \Phi') = \sum_{e \in A(v)} \delta_e(\Phi, \Phi'). \tag{22}$$

Equally important is the change in gain after a particular node has been moved, since this must be computed for all neighbours $N(v)$ of the moved node v after the action is taken. This is referred to as the *delta gain* $\Delta g_{u,v}$, for node u from moving node v. To retain the time complexity of FM, the delta gain must be computed in constant time and should avoid recomputing any auxiliary values where possible. To do this, we pre-compute the cost of all possible cost configurations of a hyper-edge. Following the approach of Huang and Khang [42], we denote the *configuration* of a set of nodes A assigned to partitions by Φ as a binary string of length $K = |Q|$:

$$cfg_i^{(A)}(\Phi) = \begin{cases} 1 & \text{if } \exists v \in A : \Phi(v) = i \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

Algorithm 2: Fiduccia–Mattheyses for temporal hypergraphs

Input:
$H = (V, E; \tau, \kappa)$: Temporal hypergraph extracted from a quantum circuit.
K: Number of partitions (QPUs)
$\Phi : V \to \{1, \ldots, K\}$: Initial partition assignment
Cap$[k]$: max qubit capacity in each partition k
Costs: precomputed cost table for hyper-edge configurations
Output:
Updated partition assignment $\widetilde{\Phi}$ with reduced entanglement cost $\tilde{c}$.

FM $(H, K, \Phi, \text{Cap}, \text{Costs})$**:**

$\tilde{c} \leftarrow \texttt{Cost}(H, \Phi, \text{Costs})$
$\mathcal{X} \leftarrow \emptyset$ `// Lock set of already-moved nodes`
$\Gamma \leftarrow \texttt{ComputeGains}(H, \Phi, K, \text{Costs})$ `// Γ(v,p): gain of moving v to partition p`
$\mathcal{B} \leftarrow \texttt{BuildGainBuckets}(\Gamma)$ `// Gain bucket structure built from Γ`
$g_{cum} \leftarrow 0$ `// Cumulative gain`
$\mathbf{g} \leftarrow (g_{cum})$ `// Sequence of cumulative gains`
$\mathbf{\Phi} \leftarrow (\Phi)$ `// Sequence of assignments`
while $|\mathcal{X}| < |V|$ **do**
  $(v^\star, p^\star) \leftarrow \texttt{BestMove}(\mathcal{B}, \mathcal{X}, \text{Capacity}, \Phi)$ `// Best admissible move`
  if $(v^\star, p^\star) = \perp$ **then**
    break `// No valid moves remain in this pass`
  $g \leftarrow \Gamma(v^\star, p^\star)$
  $g_{cum} \leftarrow g_{cum} + g$
  $\Phi \leftarrow \texttt{MoveNode}(\Phi, v^\star, p^\star)$ `// Update assignment by moving` $v^\star$ `to partition` $p^\star$
  $\mathbf{g} \leftarrow \mathbf{g} \,\|\, (g_{cum})$ `// Append current cumulative gain and assignment to the sequences`
  $\mathbf{\Phi} \leftarrow \mathbf{\Phi} \,\|\, (\Phi)$
  $\mathcal{X} \leftarrow \mathcal{X} \cup \{v^\star\}$
  $\texttt{UpdateGains}(H, \Gamma, \mathcal{B}, v^\star, p^\star, \Phi, \text{Costs})$ `// Update gain structure`
`// Roll back to the best iteration in this pass (min cumulative gain)`
bestIter $\leftarrow \arg\min_i \mathbf{g}_i$
$\Phi \leftarrow \mathbf{\Phi}_{bestIter}$
$g_{min} \leftarrow \mathbf{g}_{\text{bestIter}}$
if $g_{min} < 0$ **then**
  $\widetilde{\Phi} \leftarrow \Phi$
  $\tilde{c} \leftarrow \tilde{c} + g_{min}$
return $\widetilde{\Phi}$

where a 1 at element i indicates the presence of at least one node in P_i. Many different assignments may map to the same configuration, though the cost is completely determined by the configuration. Since we are interested in the e-bit count for each hyper-edge, we use the configurations of the root and receiver set to define the overall configuration of the edge. Entry i of the configuration will be 1 if there is at least one receiver node in i and no root node in i. We can define this using the bitwise expression:

$$cfg_i^{(e)}(\Phi) = cfg_i^{(e_{rec})}(\Phi) \wedge \neg cfg_i^{(e_{root})}(\Phi). \quad (24)$$

The associated cost is the bit count, i.e., the number of ones, in the final configuration. There are $2^K - 1$ possible configurations, and each configuration string identifies with an integer between 0 and $2^K - 1$. The cost of each configuration $c_e(\Phi)$ can be pre-computed and stored at the associated index of a vector C, such that it can be called in $\mathcal{O}(1)$ time. This is feasible for moderately sized K (≤ 25) architectures, though for future large scale-systems this may not be possible. In such cases, the cost can be computed on-the-fly from the configuration string in $\mathcal{O}(K)$ time and cached for future use. In practice, many configurations will not appear in the partitioning process and each cost will only need to be computed once, making caching a more practical alternative in general. However, this is required for ensuring efficent delta gain calculations, which is only guaranteed if all configurations can be queried in constant time.

To efficiently calculate the delta gain, we need to store the configuration of the edge and determine the change in configuration after an action is made. Since any node move has a unique source and destination, finding the new configuration requires changing a maximum of two elements of the configuration, so can be done in constant time. The nature of the gate grouping routine results in each node being part of at most *four* edges, since each node is part of at most two state edges and two gate edges. This is because a root node in a particular group may be part of the receiver set of another group. In this case, each node can be involved in a maximum of four edges and a minimum of one edge (note that, when the multilevel framework is introduced in Section 4, this will not be the case).

With this restriction, calculating $\Delta g_{u,v}(\Phi, \Phi', \tilde{\Phi}, \tilde{\Phi}')$ requires calculating the contribution from each edge adjacent to u, each of which requires *at most* four configurations [42] for each edge, giving a maximum of 16 configurations. To see this, consider node v being moved from P_i to P_j, updating Φ to Φ'. Before the move, the action of moving node u from P_k to P_l is $g_u(\Phi, \tilde{\Phi})$. Calling the new gain after both moves have been made $g_u(\Phi', \tilde{\Phi}')$, we note that the delta gain must obey the relation

$$g_u(\Phi', \tilde{\Phi}') = g_u(\Phi, \tilde{\Phi}) + \Delta g_{u,v}(\Phi, \Phi', \tilde{\Phi}, \tilde{\Phi}'), \quad (25)$$

where $\tilde{\Phi}'$ corresponds to the assignment after both moves have been made. Eq. (25) implies

$$\begin{aligned}\Delta g_{u,v}(\Phi, \Phi', \tilde{\Phi}, \tilde{\Phi}') &= g_u(\Phi', \tilde{\Phi}') - g_u(\Phi, \tilde{\Phi}) \\ &= \sum_{e \in A(u) \cap A(v)} [\delta_e(\Phi', \tilde{\Phi}') - \delta_e(\Phi, \tilde{\Phi})] \\ &= \sum_{e \in A(u) \cap A(v)} [c_e(\Phi') - c_e(\tilde{\Phi}') - c_e(\Phi) + c_e(\tilde{\Phi})].\end{aligned} \quad (26)$$

Each assignment edge pair corresponds to a particular configuration, thus if a node saturates the maximum of 4 edges, 16 configurations must be queried. Additionally, since there exist edges in $A(u)$ that may not have been affected by moving node v, we need only consider contributions from edges that are adjacent to both u and v. Thus, for each edge e in $A(v)$, we update each node comprising e using only the contributions from e. If, for any node u, there is more than one common edge between u and v i.e., $|A(u) \cap A(v)| > 1$, then each contribution will be added separately to the delta gain. This must be repeated for each neighbouring node of the moved node v, for each destination.

To perform each gain update in $\mathcal{O}(1)$ time, we must store the configuration of each edge and the cost of each configuration. In addition, we need to store two auxiliary objects for each edge, which we call the *root counts* and the *receiver counts*. Each of these is simply a vector of length $|P|$, where each element corresponds to the number of nodes from the root/receiver set in a particular partition. Each move will increment either the root or the receiver counts by removing one from the source element and adding one to the destination element. The full configuration can be updated directly from the root and receiver counts, checking whether each of the counts has changed from zero to non-zero, or vice-versa. The

cost can then be read from the pre-computed cost table.

Storage and retrieval of the counts and configurations of each edge adds an overhead, and for small graphs it may be quicker to directly compute the edge costs to find the delta gain. However, it is necessary for maintaining the time-complexity of the full algorithm, thus the benefits are clear for larger circuit sizes. The full algorithm logic is given in Algorithm 2. Since much of the heavy lifting is done in the auxiliary routines for calculating gains and delta gains, the main logic remains similar to standard FM. We include pseudocode for initialisation routines, auxiliary routines and delta gain calculation in Section A.1, namely Algorithms 5 to 7.

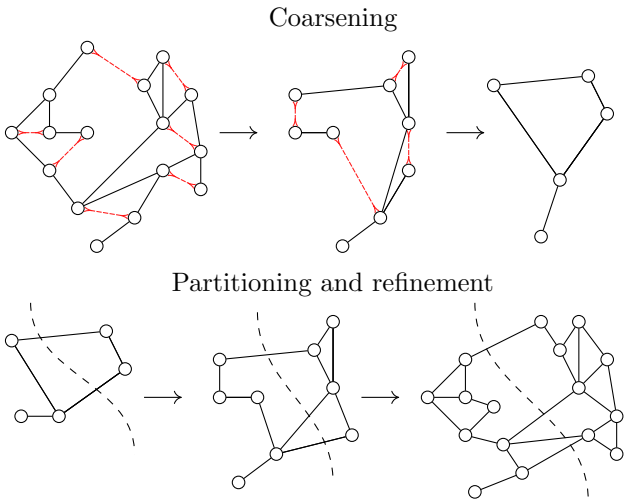


Figure 9: Illustration of multilevel partitioning for graphs. The graph is first coarsened by iteratively merging nodes together and contracting edges (red) between them. The coarsest graph is partitioned, and the solution is then uncoarsened level-by-level, applying heuristic refinement at each level to improve the solution.

4 Multilevel quantum circuit partitioning

In this section, we introduce multilevel partitioning, a powerful technique for improving partitioning quality on large-scale problems. We show how this can be applied to quantum circuit partitioning using temporal coarsening strategies that exploit the fixed temporal structure of the graphs.

4.1 Multilevel partitioning

The multilevel paradigm is a crucial part of the success of state-of-the-art methods for large-scale graph partitioning, as demonstrated by partitioners such as METIS [44, 48] and KaHyPar [47]. While heuristic refinement methods, such as FM, have proved effective for small to medium problem sizes, the solution quality typically drops as problem sizes increase. This is expected as the solution space expands exponentially, making local search heuristics increasingly less effective at exploring the solution space. Multilevel methods employ *coarsening* routines for transforming large-scale, complex problems into smaller problems, for which good results can be obtained. This is typically followed by iterative *uncoarsening* and *refinement* of the results into a solution for the larger problem.

A graph can be coarsened by merging nodes together and contracting the edges between them. A good multilevel partitioner relies on effective coarsening, as this performs part of the heavy lifting of exploring the solution space. During uncoarsening, the solution from the previous level is used as a starting point for refinement at the new level. Applying a local search heuristic between each level allows the solution to be gradually improved as more degrees of freedom are reintroduced into the problem. An illustration of a basic multilevel procedure is shown in Fig. 9.

We describe our coarsening strategy in Section 4.2, followed by the partitioning and refinement procedure in Section 4.3. An illustration of the full multilevel procedure applied to temporal hypergraphs is found in Fig. 9.

4.2 Temporal coarsening

Many multilevel partitioners will employ coarsening routines which are agnostic to the graph structure, such as random or heavy-edge matching [44]. Alternatively, structural properties can be extracted and exploited using community-aware coarsening [47]. Since our hypergraphs have a fixed, temporal, structure, we propose coarsening strategies that exploit this.

Since each qubit is expanded into d nodes, each associated with a particular time step, we propose *temporal coarsening*. For each node v such that $\tau(v) = t$ and $\kappa(v) = q$, the graph will always contain a corresponding node u such that $\tau(u) = t + 1$ and $\kappa(u) = q$, unless $t = d$, i.e., the final time step. This means that we can always merge

nodes associated with the same qubit at adjacent time steps. Moreover, since each time step has exactly one node per qubit, we can merge full time steps, or intervals of time steps, while still ensuring that each merged node corresponds to a unique qubit. This makes the process of enforcing balance constraints straightforward, since there is no need to check which qubits are represented by merged nodes.

The coarsening phase gives a sequence of graphs with decreasing temporal resolution, where the coarsest graph may have only a single time step. In this limit, we recover a *static* partitioning problem, and the hyper-edge costs reduce to standard hyper-cuts, leaving no possibilities for state teleportation. The uncoarsening phase then gradually reintroduces temporal resolution, allowing state teleportation to be be considered at increasingly many time steps.

Algorithm 3: Single-layer time contraction of a temporal hypergraph

Input:
$H = (V, E; \tau, \kappa)$: Temporal hypergraph, with time map $\tau : V \to \{1, \dots, d\}$ and qubit map $\kappa : V \to Q_L$.
s: The source time-layer to be merged.
t: The target time-layer into which the nodes at s will be merged.
Output: A coarsened hypergraph H' in which all nodes at time-layer s have been merged into t.

ContractTime (H, s, t)**:**
- $H' \leftarrow H$;
- $V^{(s)} \leftarrow \{ v \in V \mid \tau(v) = s \}$;
- $V^{(t)} \leftarrow \{ v \in V \mid \tau(v) = t \}$;
- **foreach** $v \in V^{(s)}$ **do**
  - $u \leftarrow w \in V : \kappa(w) = \kappa(v)$;
  - `// Contract self-loop`
  - $E \leftarrow E \setminus (v, u)$;
  - **foreach** $e \in E$ ***such that*** $v \in e$ **do**
    - **if** $u \in e$ **then**
      - $e \leftarrow e \setminus v$;
    - **else if then**
      - $e \leftarrow e \setminus v$;
      - $e \leftarrow e \cup u$;
- `// Remove original s-layer nodes`
- $V \leftarrow V \setminus V^{(s)}$ **return** H';

After applying some initial partitioning, the graph is then uncoarsened by a single-level, using the solution from the previous level as a starting solution. When uncoarsening, we can transform the partitioning solution from the previous level into a starting solution for the new level by assigning each newly revealed node to the same partition as the merged node from the previous level. We can do this by storing a contraction history, which tells us which nodes have been merged between each level. Between phases of uncoarsening, the solution may be refined using the FM algorithm, or some other search-based heuristic, since each level introduces new degrees of freedom into the problem. This is repeated until the full temporal resolution is restored.

Even restricting ourselves to temporal coarsening, there are still many different possible coarsening strategies, which differ in how time steps are merged together and how many merges are performed between each level. The only constraint that we place on the coarsening is full time steps of nodes must be merged in a single operation, which ensures that we always have the same number of nodes per time step as qubits.

We explore three different temporal basic coarsening strategies, referred to as *window*, *block* and *recursive* coarsening. While each strategy has its advantages and disadvantages, we find the recursive coarsening to perform best overall, so we include the pseudocode Algorithm 4 in the main text. The other two strategies are included in Section B.1 for completeness (Algorithms 8 and 9).

4.2.1 Window coarsening

The most basic uncoarsening procedure can be implemented by contracting time-intervals one-by-one from d down to 0. When uncoarsening, this is analogous to revealing the full temporal resolution one step at a time. At any given level, everything below a certain time step t is fully resolved, while everything above t is fully contracted. We still have information about the full graph, but this remains coarse for future time steps. We can generalise this procedure by revealing an interval of time steps at each level, as if sliding a window along the time-axis. Calling the size of each interval w, we can coarsen the graph by successively merging nodes in each interval into a single node, such that the number of

time steps is reduced by w at each level. This is repeated until only one time step remains, where the final interval may be smaller than w. After each interval has been contracted, a copy of the graph is stored for the refinement. This is referred to as *window* coarsening. If we set the interval size to $w = 1$, then we reveal the graph one time step at a time. If instead we input the number of levels, we can calculate the required interval sizes as $w = d/n_{levels}$, and adjust for any remainder. The full window coarsening algorithm is given in Algorithm 8 in Section B.1 and illustrated in Fig. 10.

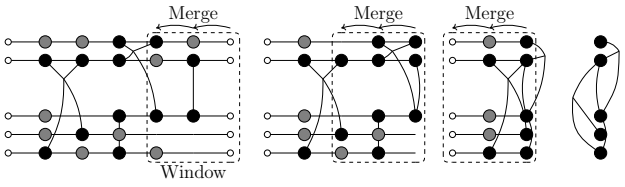


Figure 10: Illustration of the window-based coarsening procedure. A window size of 3 is used, such that three time steps are contracted at each level.

4.2.2 Block coarsening

We may also coarsen the graph into blocks of nodes by performing window coarsening simultaneously on different regions of the graph. We call this block coarsening, since each region is coarsened down into a single time block. In this case, the size of each block b corresponds to the number of levels, since we uncoarsen one time step per block at each level. We may also choose to coarsen all blocks down to a single time step. We can either set the number of blocks and calculate the block size from $b = d/n_{blocks}$ or choose the block size directly. The pseudocode is given in Algorithm 9 in Section B.1. An illustration is shown in Fig. 11, where the blocks are coarsened simultaneously, and the remaining time steps are merged into a single time step at the final level.

4.2.3 Recursive coarsening

The final method considered is referred to as *recursive* coarsening. We divide the graph into adjacent pairs of time steps then merge the nodes between them at each level. The number of nodes thus decreases exponentially with each level, making the number of levels automatically $\log_2(d)$, unless we choose to cap it earlier. Pseudocode for recursive coarsening is given in Algorithm 4.

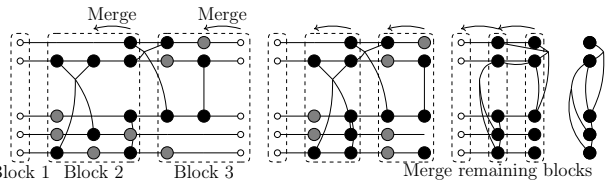


Figure 11: Illustration of the block-based coarsening procedure. A maximimum block size of 3 is used. At each level, one time step is contracted in each block. Optionally, the final level may contract all remaining time steps into a single time step.

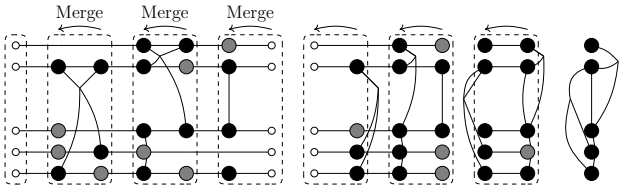


Figure 12: Illustration of the recursive coarsening procedure.

4.3 Partitioning and refinement

The full multilevel partitioning procedure is illustrated in Fig. 13, using recursive coarsening. After coarsening the graph into a series of levels, we perform an initial partitioning on the coarsest graph. Though it is common in multilevel schemes to use a different partitioning algorithm we found that it was sufficient to start with a greedy initialisation followed by FM refinement. The greedy initialisation simply fills each partition up to capacity, iterating through the numerical indices of the qubits. This is followed by a number of FM passes. Between each uncoarsening phase, the solution must be transformed into a starting solution for the next level. This is straightforward, since merged nodes do not mix qubits, allowing us to simply expand the number of time steps by assigning each newly revealed node to the same partition as the node with which it was previously merged.

5 Circuit extraction

The final step in the workflow is to extract the circuit from the partitioned graph and the assignment function. Since the partitioned circuit will require entanglement distribution, we need to

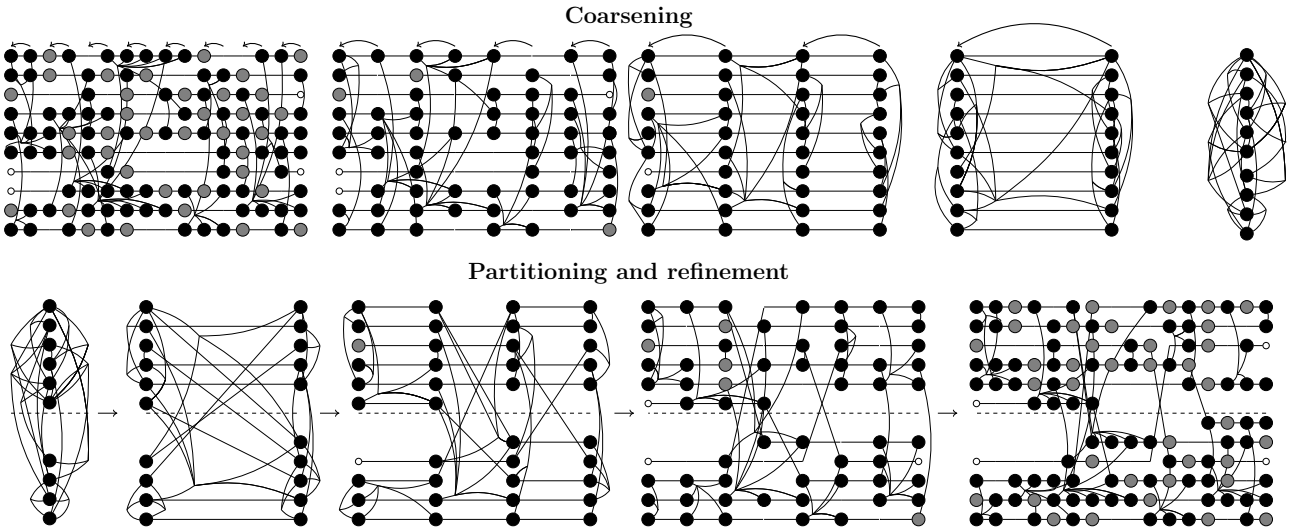


Figure 13: Full multilevel partitioning procedure using recursive coarsening. In the top sequence, the sequence of coarsened graphs is shown, starting from the full temporal resolution graph on the left, down to the coarsest graph on the right. Arrows indicate time steps being merged between levels. On the bottom, the partitioning and refinement procedure is illustrated. An initial partitioning is performed on the coarsest graph, which gives a single partition assignment for each qubit for the time of the full circuit. In the second level, the graph now splits each qubit into two time steps, corresponding to the first and second half of the circuit. Each time step is split into two at each new level, introducing new degrees of freedom for the partitioning. Between each level, FM refinement is able to improve the cost of the partitioning.

have a number of ancillary communication qubits in each QPU. If we have not used a gate grouping pass, communication qubit requirements are modest, since they are released immediately after each state and gate teleportation. However, if we have used gate grouping, there is likely to be a requirement for multiple communication qubits to be active at the same time, demanding larger communication qubit capacities. We choose to start with a predefined number of communication qubits, though we allow the possibility of dynamically adding communication qubits as needed. While this may not always be supported by physical architectures, spare data qubits may also be considered as pseudo communication qubits for links that are kept active over many time steps. Pseudo communication qubits may not be able to generate e-bits, but they can store states linked from starting processes to free up additional communication qubits. If this is not possible, then we may have to collapse and end the link with an early ending process. The previous teleportation procedure may then be restarted, at the cost of an additional e-bit. Naturally, this method will increase the depth of the resulting circuit and may introduce additional noise. We explore this trade-off further in Section 7.2. Assuming we have sufficient communication qubit capacity, we proceed as follows.

We start by creating a d-dimensional, empty list L, representing the time steps of the original circuit. We then pass through the gate edges and hyper-edges in the graph in order of time, appending operations to the corresponding entry in the list. For each cut state-edge, we schedule a state teleportation. For each gate edge, including hyper-edges, we check which partitions are spanned by both the root and receiver sets. If it is fully local, we schedule all gates as normal. If we have non-local components, we schedule a starting process for all partitions apparent in both the root and the receiver sets, and schedule all gates to their local copy of the root qubit. Since we scheduled starting processes across the root set as well as the receiver set, this means that the starting process for any nested state teleportations are already included. This means that we need to track when any linked communication qubits are no longer needed, and then schedule the corresponding ending process. If we encounter a cut-state edge while the root qubit is still active, we delay the ending process until all non-local gates in the partition have been executed, as shown in Fig. 14. If we encounter an anti-diagonal single-qubit gate that occurs on the root qubit of an active group, we apply an X

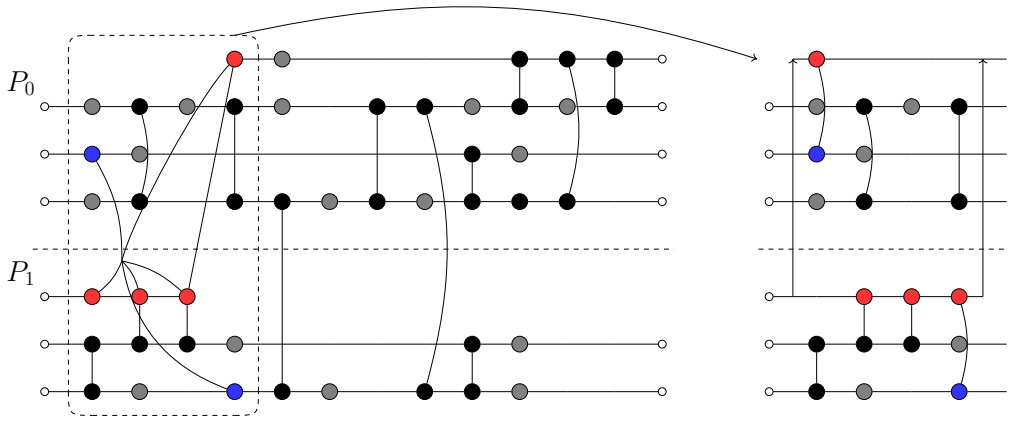


Figure 14: Partitioned graph with nested state teleportation. In the left graph, the highlighted hyper-edge has four root nodes (red) and two receiver nodes (blue). Both the receiver nodes and the root nodes span across P_0 and P_1, indicating the presence of a nested state teleportation. According to Eq. (18), the hyper-edge cost is 0. The subgraph on the right shows how the edge configuration is converted to teleportation primitives. A starting process is performed from P_1 to P_0, but since the final root node is assigned to P_1, the ending process is used to teleport the state to P_1. The 'cost' is accounted for by the cut state-edge. We show the full extracted circuit in Fig. 15.

Algorithm 4: Recursive coarsening

Input:
$H_0 = (V_0, E_0, \tau, \kappa)$: temporal hypergraph with time-layers $1, \ldots, d$.
Output:
Coarsening hierarchy $\mathcal{H} = (H_0, H_1, \ldots, H_K)$.

CoarsenRecursive (H_0)**:**
- $H \leftarrow H_0$;
- $\mathcal{H} \leftarrow (H_0)$;
- **while** $|\{\tau(v) : v \in V(H)\}| > 1$ **do**
  - `// Sorted list of distinct time-layers present in` H
  - $L \leftarrow (\ell_1, \ell_2, \ldots, \ell_m)$ with $\ell_1 < \ell_2 < \cdots < \ell_m$;
  - **for** $i \leftarrow 1$ **to** $m - 1$ **step** 2 **do**
    - $H \leftarrow$ `ContractTime`(H, ℓ_{i+1}, ℓ_i);
  - $\mathcal{H} \leftarrow \mathcal{H} \,\|\, H$;
- **return** $\mathcal{H}$;

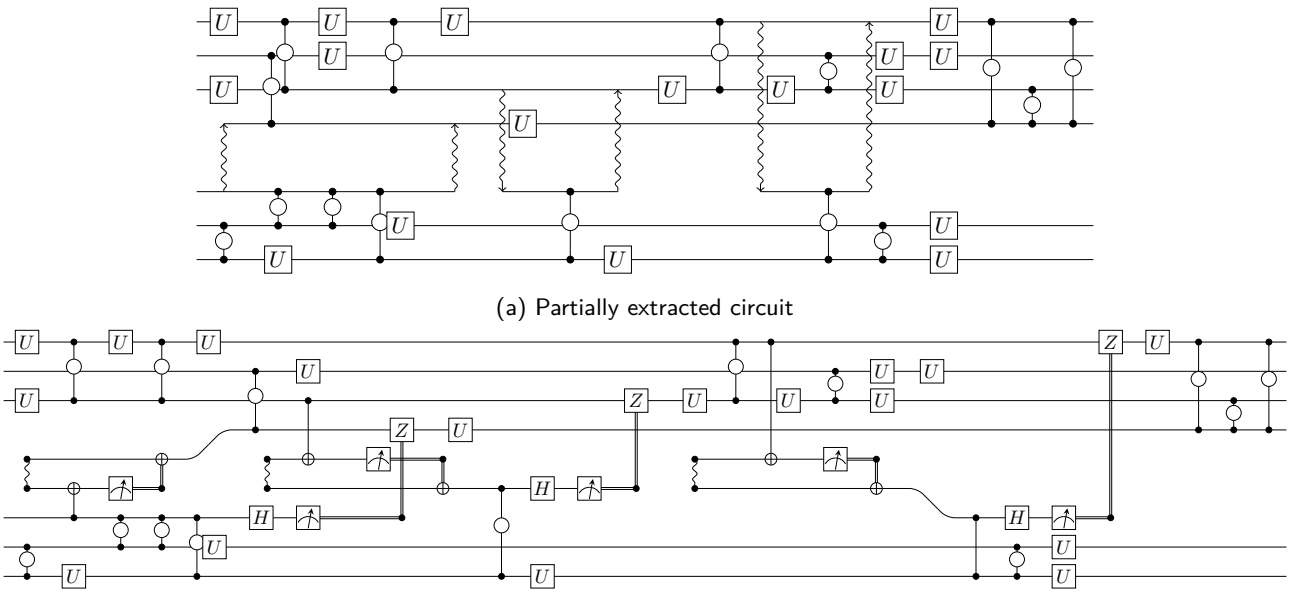
gate to each linked communication qubit, as well as the root, as described in Section 2.2.1. This laves us with a partially extracted circuit, where all hyper-edges have been converted to starting and ending processes, but not yet decomposed into standard gates. An example of a partially extracted circuit is shown in Fig. 15a.

The starting processes and ending processes are then decomposed into the sub-circuits in Fig. 1, adding in the necessary e-bit generation and classically controlled operations. This gives the fully extracted circuit, as shown in Fig. 15b.

The above procedure is sufficient for extracting a circuit with at the desired e-bit cost, though may still require further compilation within each sub-circuit to match the desired gate set or connectivity requirements. In future work, we aim to identify further avenues for optimisation, such as gate commutation within groups and detached gates, in which both controls of a symmetric two-qubit gate are linked to an external partition.

An implementation of the circuit extraction is available in the `disqco` repository [50].

To ensure that the circuit extraction has succeeded, we can simulate the distributed circuit and the original circuit and compare the measurement probabilities. Within `disqco`, the extracted circuit is a Qiskit circuit object, containing custom blocks for entanglement generation and a classical register for communication. This can be simulated within Qiskit, or converted to a QASM string for use in other simulators. We use the Qiskit sampler primitive [67] to perform the simulation. In Fig. 16 we compare the output distribution of a random 8-qubit circuit with the output distribution of the circuit extracted from the partitioned graph. This also allows us to compare the depth of the original circuit with the extracted circuit, as well as the demand for communication qubits. Typically, if gate grouping is used, the resulting depth is reduced, since



(a) Partially extracted circuit

(b) Fully extracted circuit

Figure 15: Circuit extracted from the partitioned graph in Fig. 14. In (a), the first stage of extraction is shown, where the hyper-edges are converted to the corresponding starting and ending processes. In (b) the starting and ending processes are decomposed down to standard gates, completing the circuit extraction. The result can then be passed into further hardware-specific compilation, provided the entanglement generation operations are left untouched. For simplicity, we omit the specific phase angle labels from the CP gates.

the entanglement requirements are lower, while the demand for communication qubits is higher. The extracted circuit can then be subject to further compilation passes for intra-QPU routing, gate decomposition or *fault-tolerant compilation*, which we discuss briefly in Section 10.

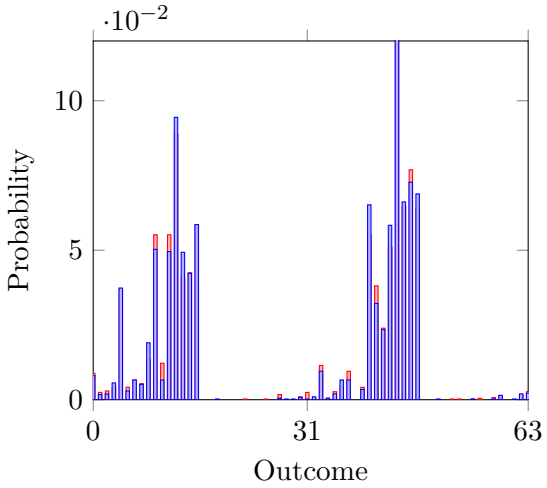


Figure 16: Comparison of the output distribution of the original circuit with that of the partitioned circuit, generated using the Qiskit Sampler. The partitioned graph is shown in Fig. 14 and the output circuit in Fig. 15. The output distribution of the partitioned circuit matches the original when looking at the reduced subspace of only data qubits.

6 Complexity analysis and optimisations

6.1 Complexity of FM for quantum circuits

The time complexity of the original FM algorithm, for bipartitioning, is linear in n where $n = |V|$ is the number of nodes in the graph. This relies on the assumption that the gain updates can be performed in constant time and the degree (number of neighbours) of each node is bounded. Each pass requires the gain of each node to be queried once when initialising the buckets, for time $\mathcal{O}(n)$, after which each node is moved once, updating the gains of all neighbours. This means the gains of the neighbours must be updated n times per pass. If there is no bound on the number of neighbours, this results in a time complexity $\mathcal{O}(n^2)$, since each node has a maximum of $n-1$ neighbours. However, in many realistic scenarios, the maximum number of neighbours is bounded by a constant that is much smaller than the total number of nodes in the graph. If we have such a bound on the number of neighbours, and delta gain calculations are $\mathcal{O}(1)$ for each of these neighbours, leaving a time complexity $\mathcal{O}(n)$ for both the initialisation and the main loop, giving $\mathcal{O}(n)$ overall. When extending to multi-partitioning,

the complexity increases linearly with the number of partitions, since the cost of moving each node must be calculated for each external partition. This makes the pass complexity $\mathcal{O}(kn)$, where k is the number of partitions.

While the algorithm structure is similar for quantum circuit graphs, the number of nodes is in fact equal to the number of qubits n_q times the depth of the circuit d, such that bipartitioning has complexity $\mathcal{O}(n_q d)$, making the pass complexity for k-partitioning $\mathcal{O}(kn_q d)$. Section 2.3 shows how to perform the gain updates in constant time for the temporal hypergraph, provided the costs of all hyper-edge configurations are pre-computed. We note that the complexity of the pre-computation is exponential in the number of partitions, since there are exponentially many possible edge configurations that must be pre-computed. However, since k is typically much smaller than n_q, this has a negligible effect up to 20 partitions. Scaling beyond this requires costs to be calculated on the fly and cached for future lookup. This is a more practical strategy even below 20 partitions, though for large k this cost calculation starts to become significant. We briefly highlight strategies for handling this in Section A.2 and point the reader to the follow-up work for more details [49].

Without gate grouping, the number of neighbours is bounded by 3, so the time complexity is maintained. However, after gate grouping, the upper bound of the number of neighbours scales with d, since this is the maximum number of gates that could exist in a group. For most circuit types, the groups will be far from this threshold. However, circuit structures that lead to very large groups do exist. An example is the QFT, which contains control chains which span the depth of the circuit. Such circuits are more computationally intensive to optimise, though they typically result in a low proportion of e-bits to gates, as can be seen in the QFT result in Fig. 31. As a result, they tend to converge to a solution earlier, requiring fewer passes to achieve a good result. The pass complexity, in this worst case, is $\mathcal{O}(kn_q d^2)$. If a constant number of passes is used, the total complexity is $\mathcal{O}(kn_q d^2)$.

6.2 Improving efficiency of FM

When dealing with large circuits, it may be necessary to look for adaptations to increase the efficiency. Firstly, we need not move every node in each pass for large graphs, since typically the iteration of best gain is achieved relatively early (after $10-20\%$ of all nodes have been moved) in the pass. We can, therefore, limit the number of nodes that are moved in a pass. The limitation is that we become less likely to escape a local minimum. There are various possibilities for remedying this – we choose to alternate between exploratory and exploitative passes. Exploitative passes are the normal passes defined in Algorithm 2. Exploratory passes, on the other hand, still seek the best gain with each mvoe but *do not* roll-back to the iteration of best cumulative gain at the end of the pass. As a result, the cost often increases after an exploratory pass, but tends to decrease on average. We find this to be effective provided we are limiting the number of moves to a small ($\simeq 0.1 n_q d$) subset of nodes, preventing exploratory passes from increasing the cost too much. We compare the effect of using exploration vs only exploiting in Fig. 23a. Clearly, while the cost changes much more rapidly in exploratory runs, the minimum value is often significantly lower, despite the fact that fewer nodes are actually moved.

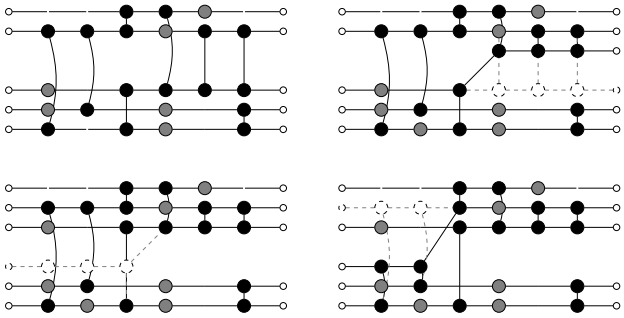


Figure 17: Illustration of a local minimum in the circuit optimization process. In the top left graph, there is no local action changing the assignment which improves the cut. A sequence of 4 neighbouring nodes moved results in the top right graph. Another 4 nodes are moved to achieve the bottom left. Finally, now that space has been made, 3 more nodes can be moved to achieve the final partitioning at the bottom right, with just two cut edges.

6.3 Avoiding local minima

Part of the reason the exploratory passes are effective is that they help to avoid local min-

ima. In all cases, we start with a *static* partitioning, meaning that $\Phi(v_i^{(t)}) = \Phi(v_i^{(1)})$ for all $t \in \{2, 3, ..., d\}$. In other words, all qubits remain in the same partition for the full duration of the circuit. The static assignment is a good starting point, since it ensures all $n_q(d-1)$ state-edges are local. The downside is that this will always be a local minimum, since each node has two local state edges (except for nodes $v_i^{(1)}$ and $v_i^{(d)}$, which have one state edge but no gate edges) and a maximum of one gate edge, since no nodes have been merged. As a result, the best local action can only increase the total cost by 1. If we consider the case after gate grouping, the same is true, since each node can be involved in a maximum of 4 edges, two state edges and two gate edges, so in the best two state edges become non-local and two gate edges become local, resulting in no net change.

As is illustrated in Fig. 17, escaping such local minima often requires a long series of local moves to open up spaces for improvement. In standard FM, each series of 'bad' moves can only move each node once due to the roll back at the end of the pass. This means that no move sequences that move multiple nodes more than once can be explored. With exploratory FM, however, this is possible. At the end of an exploratory pass, we continue the move sequence from the current configuration, unlocking all the nodes again for movement. In this way, we can explore long sequences of moves that may ultimately lead to a better configuration. By alternating between exploratory and exploitative passes, we tend to converge to lower costs, as shown in Fig. 23a.

6.4 Complexity of multilevel FM

While we still perform partitioning at each level of the coarsening hierarchy, we gain a speed up from the fact that, at coarser levels, the number of nodes is reduced. In general, each pass has a complexity $\mathcal{O}(k|V| \max_{v \in V} |N(v)|)$, where $|V|$ is the number of nodes in the graph, and $\max_{v \in V} |N_H(v)|$ is the maximum degree of any node in the graph. When we coarsen down to a single time step, we decrease both the number of nodes in the graph from $n_q d$ to n_q, and the maximum degree from $\mathcal{O}(d)$ to $\mathcal{O}(n_q)$. However, the total complexity is not reduced if we still optimise at the finest level, as this level will dominate the scaling. This is demonstrated in Fig. 25b, where the slope of each of the multilevel methods steepens with each level until it reaches the finest.

There are approaches we can take to reduce the complexity of the finer levels. If the finest graph used is the original graph, and the number of levels is predefined, then there is no change to the complexity. When dealing with deep circuits, we may restrict the finest phase used to reduce the required computation. For example, if we temporally coarsen the graph down to n_q nodes, we reduce the number of nodes as well as the maximum number of neighbours, which goes from $\mathcal{O}(d)$ to $\mathcal{O}(n_q)$. However, the number of edges each node is involved in, $A(v)$, has now gone from being bounded by 3 (or 4 after gate grouping) to being bounded by d. In this worst case scenario, where a node v has an independent gate edge from each time step, then each of these must connect to a single other neighbour, i.e., they are all 2-edges rather than hyper-edges. This means that for each of these, we only have one neighbour to update, for a maximum of n_q. Alternatively, if the node is involved in hyper-edges that extend over d time steps, the same restriction applies, since each time step over which the hyper-edge spans constitutes a time step in which the node cannot be involved in another hyper-edge or gate-edge. Thus, even though we have coarsened the maximum number of neighbours to n_q, the maximum number of contributions to the delta gain is still d, so the pass complexity at the coarsest level is $\mathcal{O}(kn_q d)$.

As we uncoarsen, we regain the factor of d due to the increased number of time steps. However, due to the effectiveness of the coarser levels, we can put a firmer cap on the number of nodes moved at each pass. If we limit the number of nodes moved to n_q, we retain the pass complexity $\mathcal{O}(kn_q d)$, with an additional factor for the number of levels. In practice, the length of passes will still change between levels due to the initial gain computation, which scales with the number of nodes at the given level. We enforce the same number of levels for each coarsening strategy. In the recursive coarsening strategy, the resulting number of levels is logarithmic in the depth of the circuit. Thus, in all tests, we use $\log_2(d)$ coarsening levels. The resulting complexity of each of the multilevel methods is $\mathcal{O}(kn_q d \log_2(d))$. In Section 7.4, we experimentally compare the speed and performance of each method.

7 Results

We run several tests to benchmark the performance of the different workflows. Recalling the sequence of stages in the framework, we have *transpilation*, *gate grouping*, *graph conversion*, *coarsening*, *partitioning*, *uncoarsening/refinement* and *circuit extraction*. Not all of these phases are varied in the benchmarking. Notably, transpilation, graph conversion and circuit extraction remain the same in all cases. We use the basic Qiskit transpiler into the gate set described in Eq. (15). For gate grouping, we investigate the *greedy gate grouping* method, described by Algorithm 1, and the trivial case of no gate grouping. For coarsening, we investigate the basic FM with no coarsening, the window-based multilevel method, MLFM-W, the block-based method, MLFM-B, and the recursive method, MLFM-R. For the partitioning algorithm, we briefly compare the FM algorithm with its exploratory variant. For each of the k QPUs used, we allow a qubit capacity $\lfloor n_q/k \rfloor + 1$, allowing an additional space to facilitate state teleportation. We make no assumptions on the number of communication qubits.

All tests are performed on a MacBook Pro M3 Pro with 18GB RAM and an 11-core CPU containing 6 performance cores and 5 efficiency cores, with performances cores clocked at 4.05 GHz and efficiency cores at 2.8 GHz. All code used to generate the results is available in the `disqco` repository [50], with scripts to reproduce the results in the `benchmarking/` directory.

7.1 Benchmark circuits

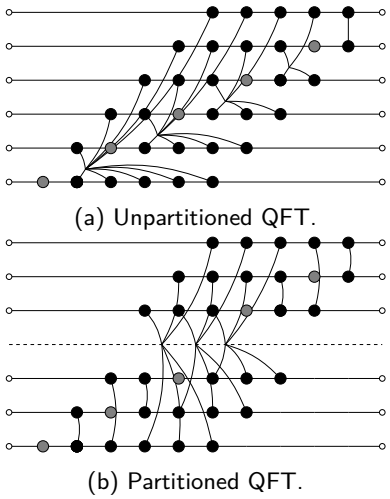
The set of circuits to be used in benchmarking is chosen to cover a wide spectrum of circuit structures. This allows us to identify cases where existing methods perform well, and cases where they do not, while demonstrating the consistency of the proposed framework in all cases. This also allows us to draw some conclusions about which circuits are best suited to which methods of non-local gate coverage.

7.1.1 CP-fraction

CP-fraction circuits are a generalisation of the CZ-fraction circuits introduced by Sundaram et al. [20]. They provide a useful tool for benchmarking the performance of circuit partitioning and distribution algorithms, since the proportion of two-qubit gates can be varied for a fixed depth and number of qubits. The CP-fraction variant was introduced in Ref. [64], as a means of generalising to a universal gate set, since transpiled CZ-fraction circuits often lose depth due to commutation and cancellation between Hadamard gates and CZ gates. A CP-fraction circuit can be constructed as follows. For a given probability p, qubit count n_q and depth d, single-qubit gates are applied to each qubit with probability $1-p$, while the remaining qubits are paired and acted on with two-qubit gates. This is repeated up to depth d. We use $U(\theta, \phi, \lambda)$ and $CP(\theta)$ gates with randomised parameters.

7.1.2 Quantum Fourier transform

The quantum Fourier transform (QFT) is a well-known subroutine that is used in various quantum algorithms. The prevalence of the QFT makes it an essential benchmark. The QFT can be built using only controlled-phase ($CP(\theta)$) gates and Hadamard H gates. Long chains of contiguous $CP(\theta)$ gates lead to large hyper-edges, which favours methods permitting gate grouping. Furthermore, it is observed that in our framework, the lowest entanglement cost is achieved using the trivial, initial placement of qubits, indicating that the initial placement is a global optimal for this structure. We note that similar results have been found elsewhere [71].



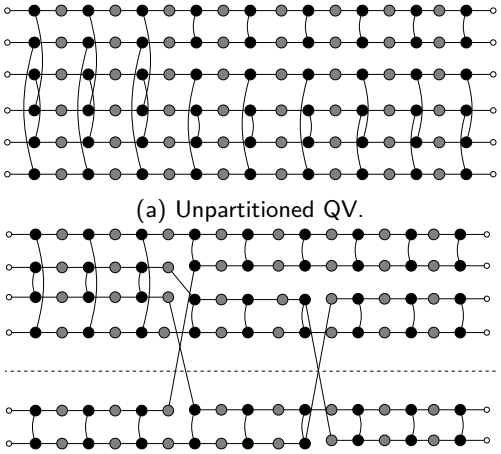
(a) Unpartitioned QFT.

(b) Partitioned QFT.

Figure 18: 6-qubit QFT graph after gate grouping. After partitioning, we remove all local contributions to hyper-edges. Note that, with gate grouping, the optimised result is the same as the initial placement.

7.1.3 Quantum volume

Quantum volume (QV) circuits, introduced by Cross et al. [72], are designed to benchmark the performance of quantum computers. QV circuits consist of layers of Haar random unitaries selected from the special unitary group $SU(4)$ applied to randomly chosen pairs of qubits. QV circuits provide a nice balance between structure and randomness, since the decomposed layers require the same qubits to interact for a number of time steps before the next layer is applied. Notably, since the unitaries are chosen randomly, decomposed two qubit-gates are unlikely to contain potential for gate grouping, since the resulting blocks do not contain contiguous $CP(\theta)$ gates, and any single-qubit gates are unlikely to be diagonal (or anti-diagonal). As a result, methods that are reliant on gate teleportation typically perform poorly, while methods employing state teleportation perform well, as observed in Ref. [64]. For all tests, we set the number of layers to be equal to the number of qubits.



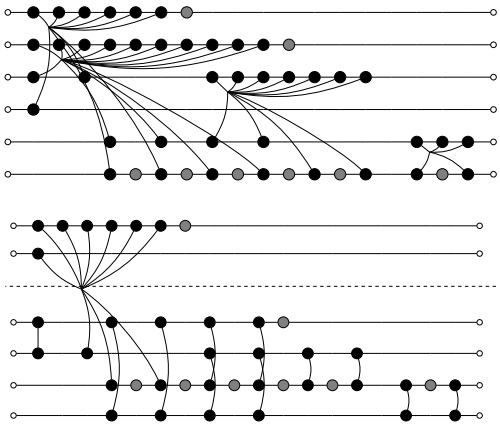
(a) Unpartitioned QV.

(b) Partitioned QV.

Figure 19: Example graph from 6-qubit from QV circuit. Note that the grouping algorithm has no effect on QV circuits, since there are no contiguous $CP(\theta)$ gates and random unitaries are unlikely to decompose into diagonal or anti-diagonal single-qubit gates.

7.1.4 Quantum approximate optimisation algorithm

The quantum approximate optimisation algorithm (QAOA) was introduced as a method for efficiently solving hard combinatorial optimisation problems, such as MaxCut and MaxSat [73]. QAOA circuits comprise parameterised single- and two-qubit gates, for which the parameters are classically optimised to find an approximately optimal solution. We consider randomly initialised QAOA circuits for MaxCut problems over random graphs with an edge probability of 50%. QAOA circuits, like QFT, permit large groups of gates for multi-gate teleportation.



(a) Partitioned QAOA.

Figure 20: Example graph from 6-qubit QAOA circuit. Circuit is a randomly initialised instance of QAOA for MaxCut over a random graph with edge probability 50%.

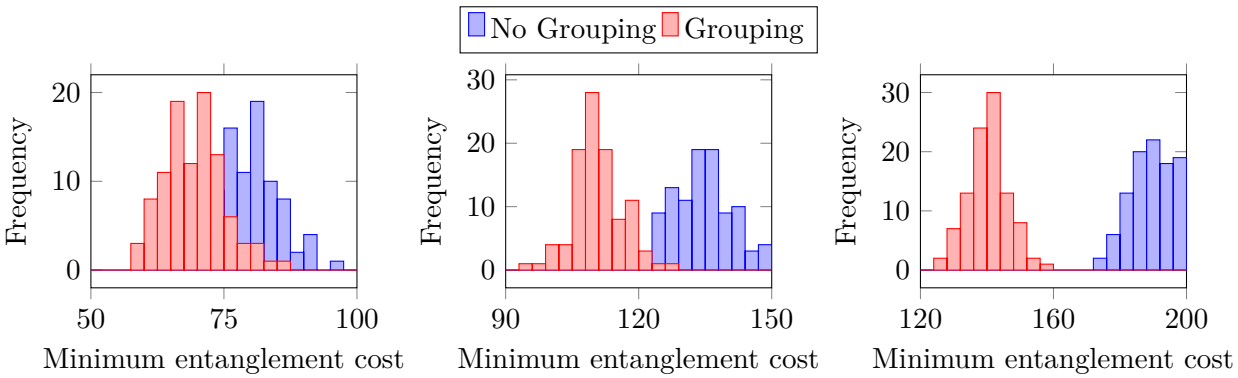
7.1.5 QASM benchmark suite

In addition to the circuits described above, we test on a number of circuits provided by the QASM benchmark suite [74]. We test on all circuits in the large category up to 100 qubits and 1000 time steps after transpilation, excluding QFT and QV since they are tested separately.

7.2 Benefits and limitations of gate grouping

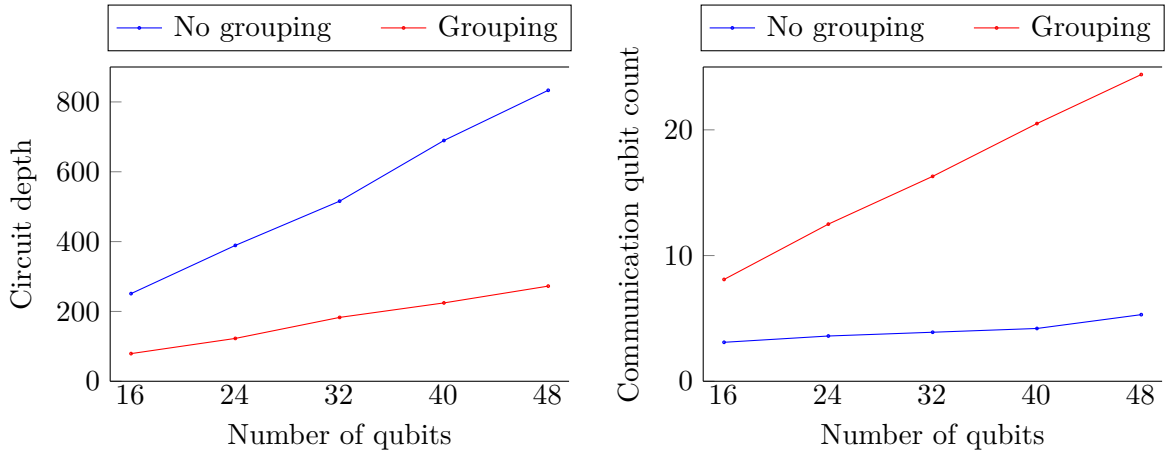
7.2.1 Reduced entanglement requirements

Here we provide a brief demonstration of the benefits and limitations of gate grouping. Using the CP fraction benchmark circuits, we vary the fraction of two-qubit gates, for a constant depth and qubit count of 32, partitioning the resulting circuits across 4 QPUs. In Fig. 21, we plot the distributions of the entanglement costs achieved for 100 randomly generated CP fraction circuits. While the effects are less noticeable for low fractions, the benefits of gate grouping become apparent as the fraction of two-qubit gates increases, where the distribution of the results with gate grouping is shifted towards lower cost. However,



(a) 30% CP gates (b) 50% CP gates (c) 70% CP gates

Figure 21: E-bit reduction from gate grouping. As the fraction of two-qubit gates increases, gates are grouped into larger hyper-edges, allowing for reduced entanglement costs via multi-gate teleportation. Plots produced from 100 runs of FM, on a 32-qubit CP-fraction circuit with depth 32, partitioned across 4 QPUs.



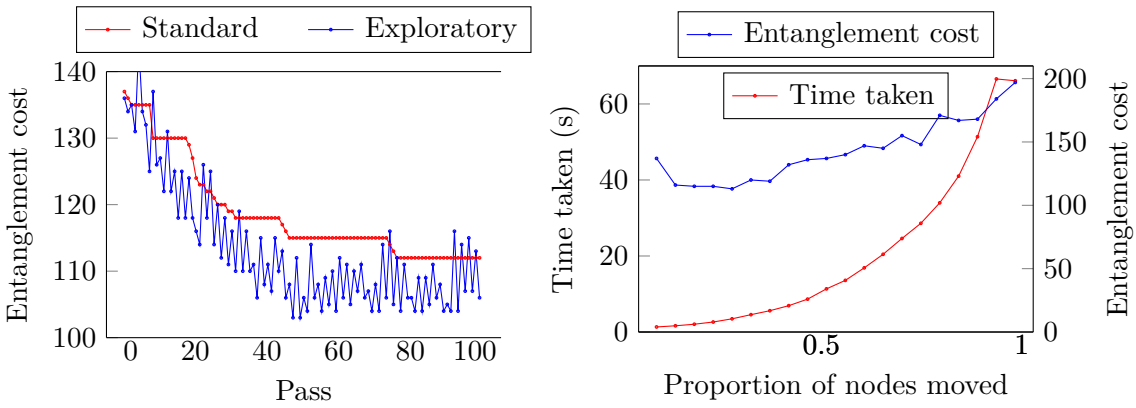
(a) Distributed circuit depth (b) Communication qubit requirements

Figure 22: Comparison of circuit depth and communication qubit requirements with and without gate grouping. The mean results are shown from 10 runs of FM on QFT circuits, each partitioned over 4 QPUs. The circuit depth is significantly reduced when using gate grouping, though this is compensated by an increase in the number of communication qubits required.

this can come at an extra cost in time as the fraction approaches 1, since the gain updates must be performed over very large hyper-edges. Additionally, as discussed in Section 5, the demand for communication qubits increases as the fraction of two-qubit gates increases. Without multi-gate teleportation, communication qubits are released immediately after use, while multi-gate teleportation requires communication qubits to be kept alive for the duration of the group. Conversely, a large communication qubit capacity reduces the additional communication depth, since entanglement can be generated in parallel.

7.2.2 Increased communication qubit requirements

A notable limitation of the current work is that no constraints are placed on the number of communication qubits permitted. This is of significance when considering instances of hypergraph with large hyper-edges, since a large cut hyperedge corresponds to a communication qubit being used over a large number of time steps. Facilitating multiple linked root qubits concurrently may require a large number of communication qubits. The benefit of not using cat-entanglements is that we only require a fewer communication qubits, since each gate teleportation and each state tele-



(a) Cost vs. pass number

(b) Final cost vs. proportion of nodes moved.

Figure 23: Illustration of the exploratory process to avoid local minima in FM. Fig. 23a shows the entanglement cost of the circuit after each pass. The cost in the exploratory method fluctuates significantly between passes but ultimately reaches lower cost regions. Fig. 23b indicates that the exploratory method is more effective when the number of moves per pass is limited. The optimal region is between 0.1 and 0.2 of the total number of nodes. Consequently, the exploratory method achieves lower costs and is faster than the standard FM algorithm.

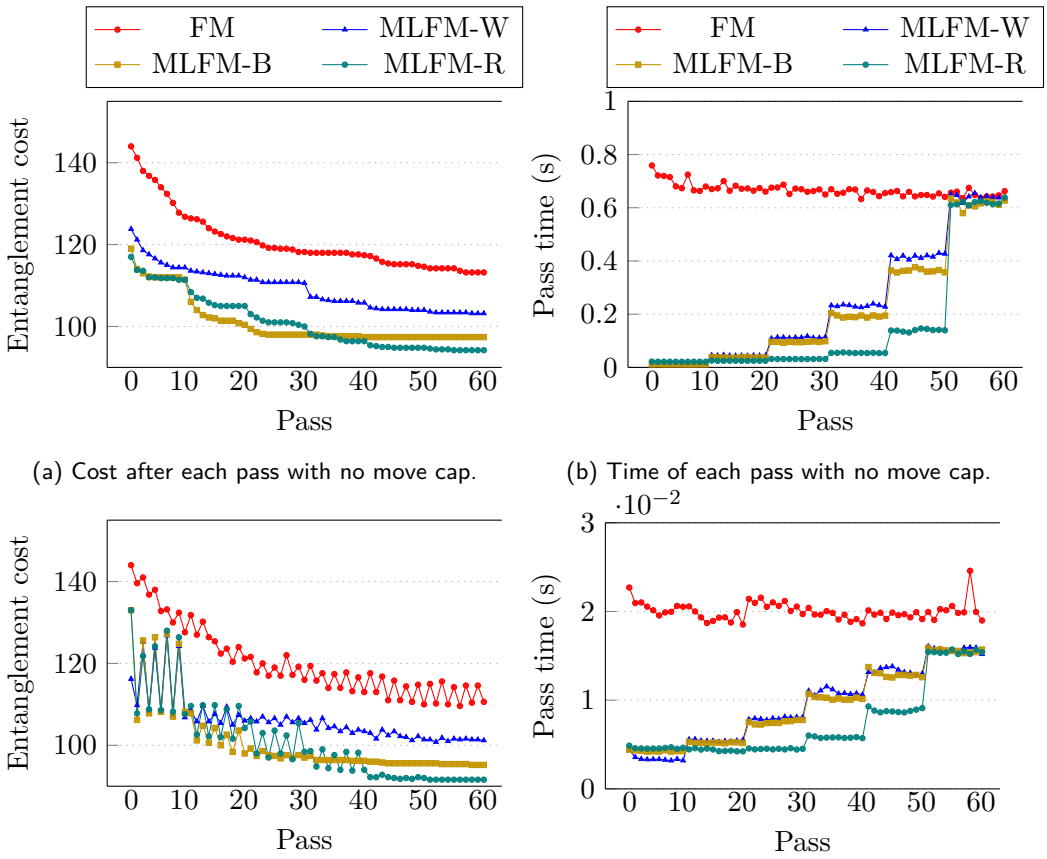
portation results in communication qubits being released immediately after use, unlike for multi-gate teleportation. The only time we may require additional communication qubits is if we need to handle multiple state teleportations at the same time step, though any additional communication qubits are still freed up directly after use. The trade-off is that we will likely end up with a deeper circuit, requiring more entanglement distribution. We briefly numerically analyse this trade-off with the following experiment on QFT circuits. For each circuit, we run the standard FM algorithm, with and without using the gate-grouping pass. When extracting the circuit, we plot both the resulting depth and the communication qubit requirements. It can be seen that the number of communication qubits required is constant when not using gate grouping, while the circuits are significantly deeper. On the other hand, the gate-grouping pass gives us a much shallower circuit but with an increasing communication qubit requirement. The choice of whether or not to use gate grouping should be made on the basis of additional qubit capacity. Additionally, we hope the lower e-bit costs from gate grouping serves to guide future DQC architectures to not under-allocate communication qubits, since they permit significantly reduced circuit depth.

7.3 Exploratory vs exploitative FM

Here we demonstrate the effectiveness of exploratroy FM described in Section 6.2 for avoiding local optima. To recall, the method involves alternating between exploratory and exploitative passes, where the former does not roll-back to the iteration of best gain at the end of the pass. This allows the algorithm to explore larger sequences of moves which may allow the same node to be moved multiple times before a roll-back. We find that this is most effective when the number of nodes moved in each pass is limited, since the algorithm is less likely to escape a local minimum. This has an added benefit of providing a reduction of the pass time. However, we typically use a constant fraction of the number of nodes, so this reduction is only constant. The results are shown in Fig. 23.

7.4 Multilevel partitioning

We compare the cost and time performance of the coarsening routines described in Section 4. Partitioning is performed using no coarsening (FM), window coarsening (MLFM-W) Section 4.2.1, block coarsening (MLFM-B) Section 4.2.2 and recursive coarsening (MLFM-R) Section 4.2.3. In Fig. 24, we show the entanglement cost and time taken as function of the number of passes for each method. For each of the multilevel methods, we run 10 passes at each level, and for FM we run $10\lceil \log_2(d) \rceil$ passes, such that the total number of passes is the same in all cases. Fig. 24a shows the entanglement cost after each pass, demonstrating the improved performance of the multilevel methods. After each 10 passes, the entan-



(a) Cost after each pass with no move cap.

(b) Time of each pass with no move cap.

(c) Cost after each pass with exploration and move cap.

(d) Time of each pass with exploration and move cap.

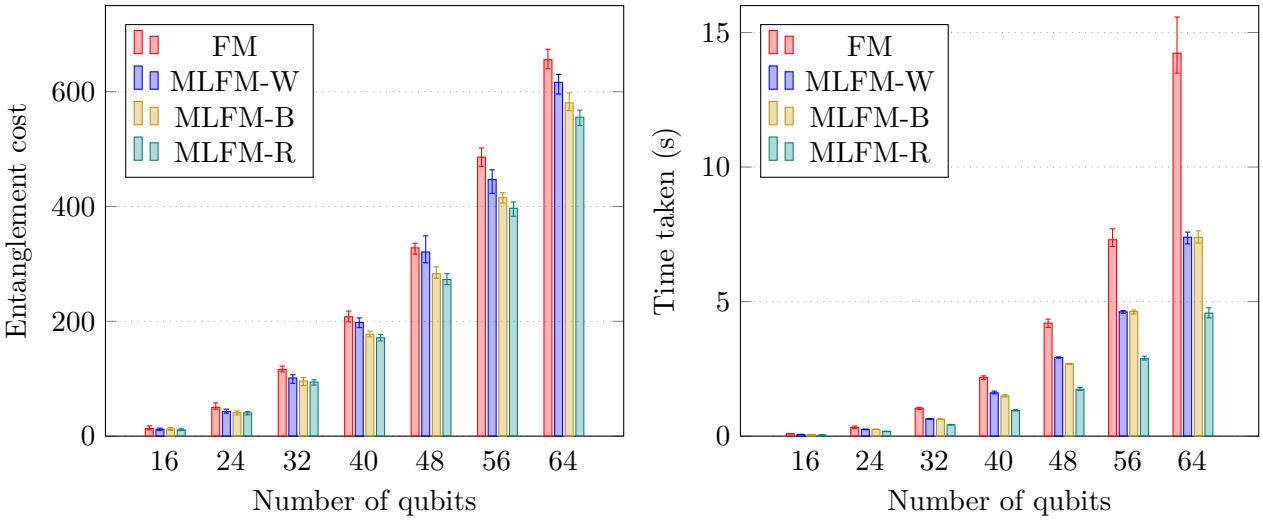
Figure 24: Performance results of the coarsening methods. All results are produced from 10 runs on a 32-qubit CP-fraction circuit with depth 32 and 50% two-qubit gates, partitioned across 4 QPUs. In Figs. 24a and 24b, we implement no limit on the number of nodes moved per pass, such that the time of the pass scales with the number of nodes in the graph. In Figs. 24c and 24d, we place a cap of n_q on the number of nodes moved per pass for the ML methods, as discussed in Section 6.4.

glement cost plateaus, indicating that the methods are reaching a local optimum for the given level, which is then quickly improved upon at the next level. Fig. 24b shows the time taken for each pass, demonstrating the improvement in run time of coarser passes. Notably, the last 10 passes are performed at the finest level of granularity, such that there is no improvement on the pass time. Overall, time taken is significantly reduced from the coarser levels. Figs. 24c and 24d employ the exploratory method with a cap on the number of nodes moved per pass. The cap is set to $0.125n_qd$ (the optimal proportion of nodes to move from Fig. 23b) for the fine-grained FM method and n_q for the multilevel methods, since the performance is retained for fewer node moves. We see similar results for the entanglement costs, while the time taken is much lower for all coarsened methods. In particular, the recursive method is both the quickest and achieves the lowest entanglement costs.

The results in Fig. 25 are generated using the same parameters as Figs. 24c and 24d. The recursive method maintains the lowest entanglement costs and is significantly faster than the other methods.

7.5 Benchmark algorithms

To compare the performance of the multilevel framework, we compare the results with the best performing methods from the literature. We use a number of workflows from Pytket-DQC [75], which are the subject of References [15, 22, 66], and the fine-grained partitioning method from Ref. [17]. We use the best performing pipeline from our methods, which uses gate grouping, recursive coarsening and exploratory FM, referred



(a) Entanglement cost

(b) Time taken

Figure 25: Comparison of the coarsening methods in terms of entanglement cost and time. Tests use a CP-fraction circuit with 50% two-qubit gates, using the same total number of passes for each method.

to as *MLFM-R*. The number of moves per pass is capped at n_q, as discussed in Section 6.4, and 10 passes are performed at each level. Below we describe the methods used for comparison.

7.5.1 Pytket-DQC

Pytket-DQC is a freely available python library within Tket, Quanntinuum's open-source quantum computing toolkit [68], containing various methods for quantum circuit optimisation. Pytket-DQC is an extension that contains a number of tools for distributing quantum circuits over multiple QPUs. The optimisation techniques available are the subject of References [15, 22, 66]. The most basic workflow available, referred to as *Partition* (P), is a static hypergraph partitioning framework [15] that employs a third-party software, KaHyPar [47], to obtain low cost assignments of qubits to QPUs and leverages *detached gates* to reduce entanglement costs [66]. The methods of Ref. [22] are integrated into Pytket-DQC, using minimum vertex cover and embedding techniques to reduce the entanglement costs of the circuit. These methods are combined in Ref. [66], with additional workflows for refining the results. The best performing workflow, based on the results in Ref. [66], is the *CoverEmbedSteinerDetached* (ESD) method, which uses various methods to refine an initial partition, though the combination of techniques means it is also the slowest. The fastest method is Partition (P), since KaHyPar is a highly optimised library written in C++. Despite being the quickest, the performance of Partition is often comparable with the more advanced techniques. A method for refining the result of Partition to incorporate embedding is also available, referred to as PartitionEmbed (PE). This method is slower than Partition, but faster than EmbedSteinerDetached, and tends to achieve better results than Partition. None of the methods available use state teleportation, resulting in certain cases where all techniques underperform, notably the Quantum Volume benchmarks shown in Fig. 33. This was highlighted by the authors in Ref. [66], though not attributed to the absence of state teleportation capabilities. We compare results with Partition, PartitionEmbed and EmbedSteinerDetached where applicable. We note that Pytket-DQC requires its own transpilation pass to be used before distribution. This gate-set also contains controlled-phase gates, so two-qubit gate counts should be similar though we cannot guarantee identical circuits. We pass the circuits as transpiled by Qiskit directly to Pytket-DQC, which performs its own transpilation internally.

7.5.2 Fine-grained partitioning

A pioneering technique in the quantum circuit partitioning literature was introduced by Baker et al. [17]. The paper provides a method for covering all non-local gates using state tele-

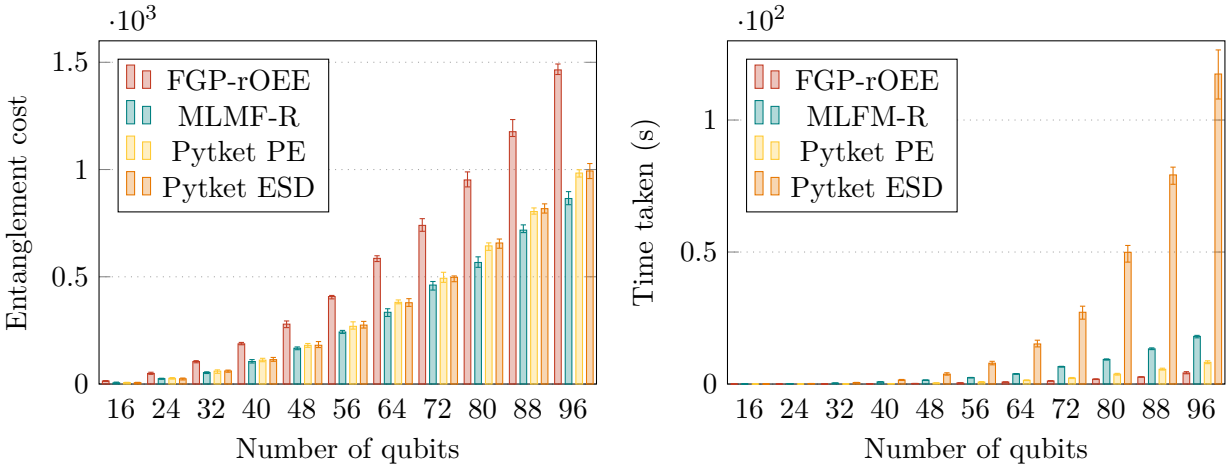


Figure 26: Circuit with 30% two-qubit gates. Starting with two QPUs for a 16 qubit circuit, for each increase in number of qubits we add a new QPU to the system. Each QPU has 9 qubit slots, which leaves a potential free data slot for teleportation on each QPU.

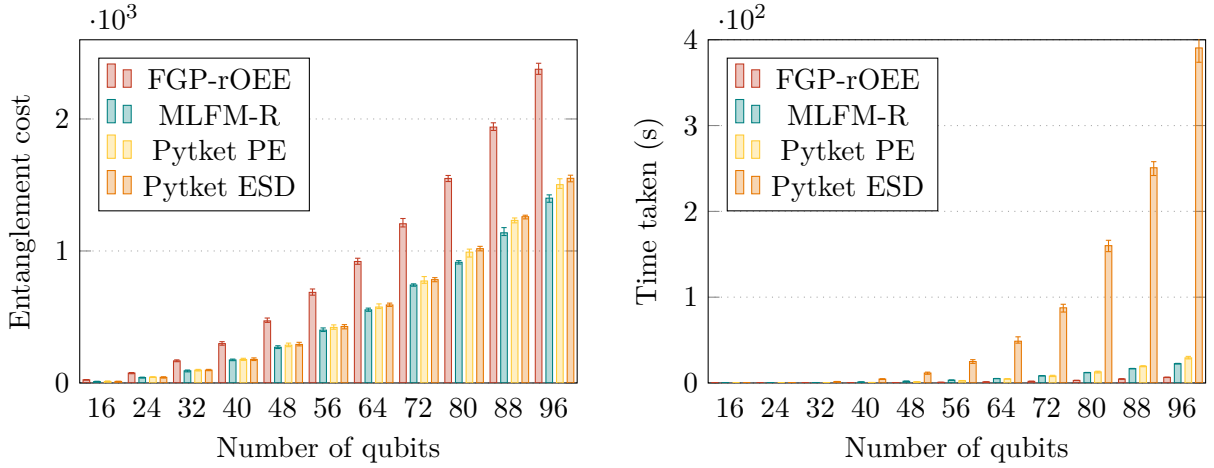


Figure 27: Circuit with 50% two-qubit gates.

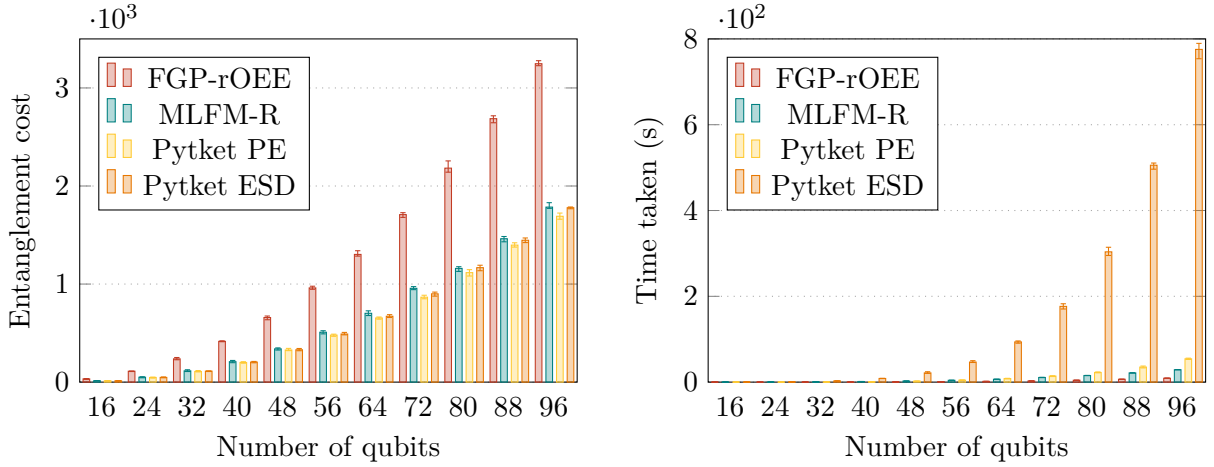


Figure 28: Circuit with 70% two-qubit gates.

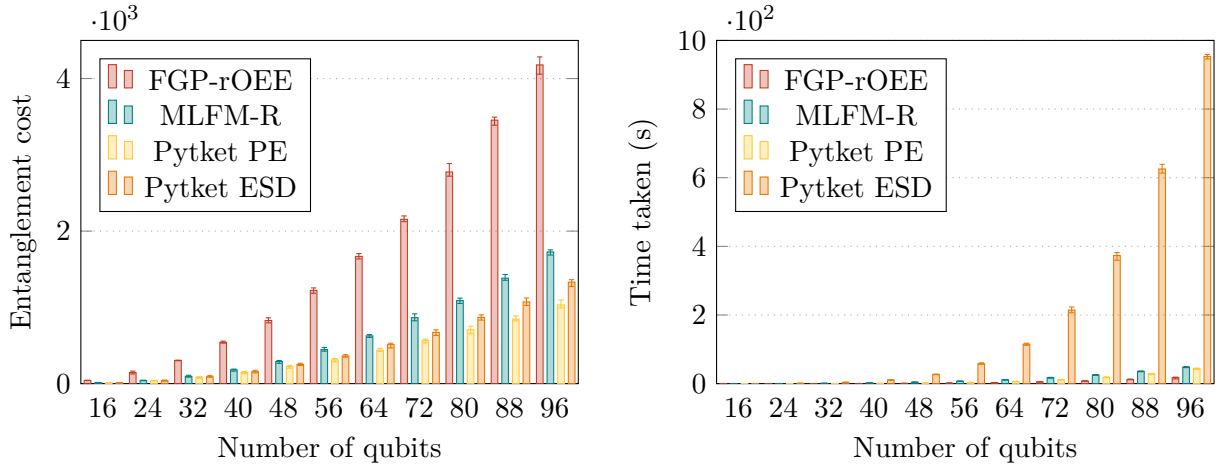


Figure 29: Circuit with 90% two-qubit gates.

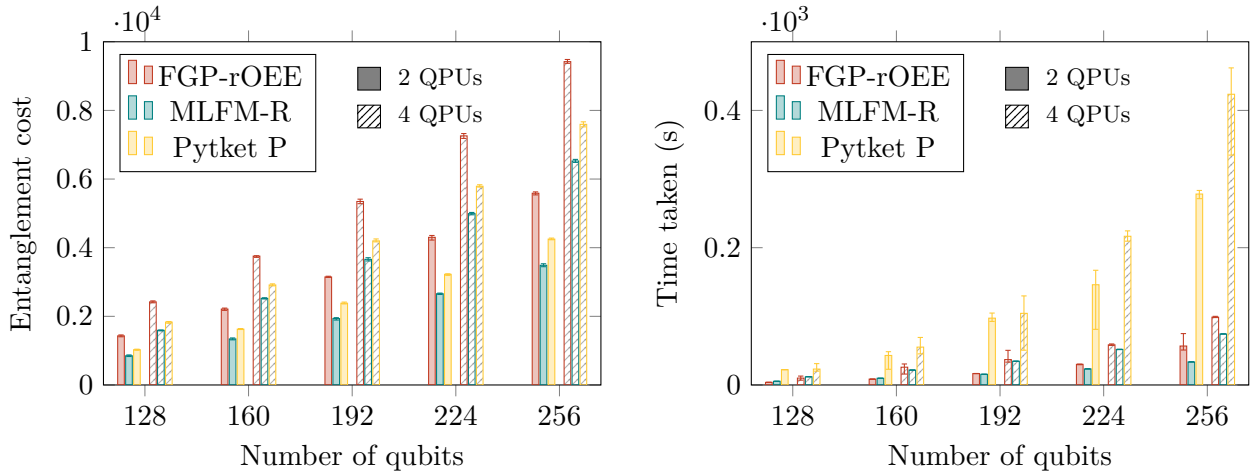


Figure 30: Large CP-fraction circuits with 50% two-qubit gates, partitioned over 2 and 4 QPUs.

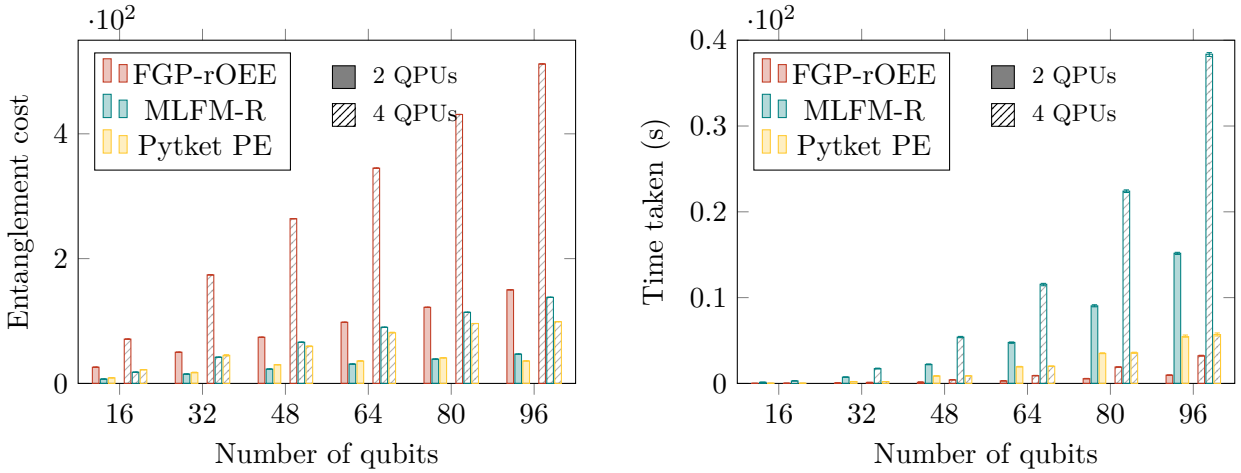


Figure 31: QFT circuits partitioned over 2 and 4 QPUs.

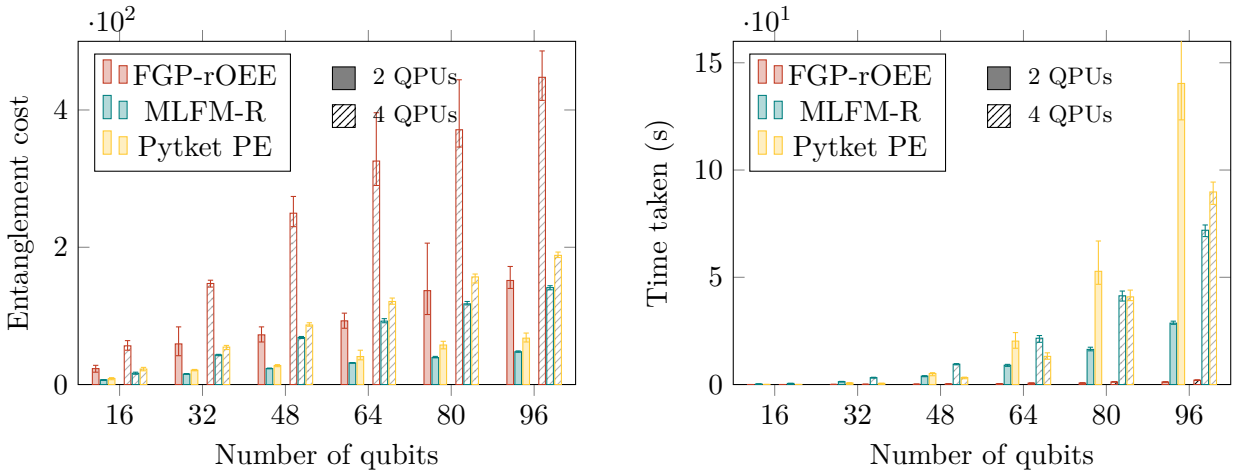


Figure 32: QAOA circuits partitioned over 2 and 4 QPUs.

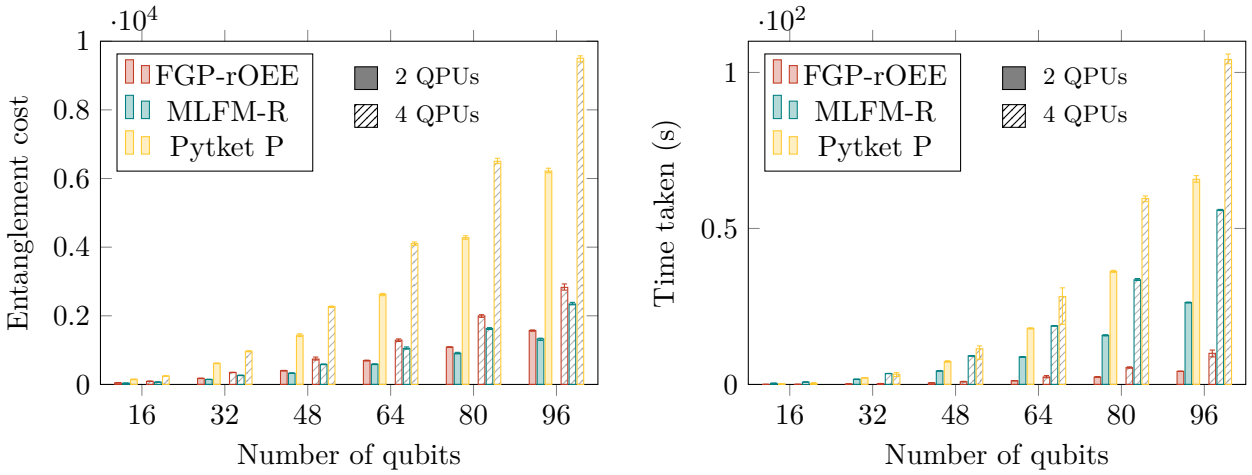


Figure 33: QV circuits, partitioned over 2 and 4 QPUs.

portation, referred to as fine-grained partitioning. The essence of the method is to split a quantum circuit into a sequence of static graphs indicating the qubit interactions at each time step, with time decaying contributions for future time steps. A refinement heuristic, called overall extreme exchange, is employed to achieve a fully local partitioning for each time step, using the assignment for each time step as the starting point for the next. In similar fashion to our approach, the end result is a sequence of assignments of qubits to QPUs, one for each time step of the circuit. The difference is that the assignment guarantees that all two-qubit gates occur locally, using state teleportations to transition between each contiguous assignment. The algorithm used is referred to as FGP-rOEE. FGP-rOEE does not allow for gate teleportation or multi-gate teleportation, so it typically underperforms in circuits that permit large groups of teleportation compatible groups.

8 Discussion

The results indicate that the proposed framework can achieve lower entanglement costs than existing, state-of-the-art techniques, provided that an appropriate algorithm is used to partition the resulting hypergraph. The results in Fig. 21 show the effectiveness of employing a gate grouping routine, allowing entanglement costs to be reduced via multi-gate teleportation. However, Fig. 22b highlights that this is traded off with the communication qubit capacity required to achieve these entanglement costs. While this means some systems may be limited in the extent to which they can benefit from multi-gate teleportation, we believe that this should serve as an indication that large numbers of communication qubits are

desirable for DQC architectures. Fig. 23 shows the effectiveness of using an exploratory variant of the FM algorithm which is more effective at escaping local minima. This is particularly important given the structure of the hypergraph, where using a static initial assignment of qubits to partitions initialises the algorithm in a local minimum, as seen in Fig. 17. The exploratory variant allows larger sequences of moves to be probed, which may move the same node more than once, ultimately breaking into lower cost solution regions.

Investigation of the multilevel framework shows that all coarsening strategies result in both lower entanglement costs and shorter runtime than the basic FM approach, shown in Fig. 24. The recursive coarsening strategy proves to be both the most effective and the most time efficient, so is used as the main method for comparison with other techniques. Figs. 26 to 29 compare the recursive multilevel FM (MLFM-R) with two methods from Pytket-DQC (Partition and PartitionEmbed) and the FGP-rOEE algorithm from Ref. [17]. MLFM-R achieves the lowest entanglement costs for two-qubit gate proportions of 0.3 and 0.5, roughly matches the best performing methods for 0.7, though underperforms for 0.9. This is likely due to the fact that the Pytket-DQC methods optimise for detached gates, those which are executed in a partition which is not local to either the root or the receiver qubit. This is an effective technique for extremely high proportions of two-qubit gates. Since we do not consider this possibility, we limit the scope of multi-gate teleportation. We aim to extend the generality of the framework in this way in future. We also note that Pytket-DQC requires its own transpiler pass before distribution, which may merge together adjacent two-qubit gates when the proportion of two-qubit gates is very high.

In the time results, MLFM-R is significantly faster than Pytket ESD, moderately faster than Pytket PE, but slower than FGP-rOEE. Notably, the core FM algorithm also scales with the number of partitions. In these tests, the number of QPUs is increased by one each time the number of qubits is increased, such that the time is scaling with both the circuit size and the number of partitions. This is a limitation that we aim to mitigate in the future. However, in Fig. 30 we show that, for a constant number of partitions (2 and 4), the MLFM-R outperforms even the fastest Pytket-DQC method, Partition, in both entanglement costs and time, as well as the FGP-rOEE algorithm. We were unable to complete these tests for Pytket PE or ESD, since the runtime was too long. This demonstrates that MLFM-R is able to scale to circuits with much larger numbers of qubits when there is an appropriate limit on the number of partitions.

Furthermore, it is possible to reduce the time even further by terminating before the finest levels of refinement, though we do not investigate this here. In Fig. 31 we investigate the performance of the algorithm on QFT circuits. Interestingly, as mentioned in Section 7.1.2, we note that no improvement is achieved for any QFT circuits using any FM methods, though we achieve a very low entanglement cost. Previously we speculated that the initial placement may be a global optimum after the greedy gate grouping. In practice, it may not be necessary to perform any optimisation, and the initial placement can be used alongside the greedy gate grouping routine. However, for larger QFT circuits, we noted that Pytket PE was able to achieve lower entanglement costs than our initial placement. However, the QFT cost after 80 qubits seems to plateau, indicating that some approximation in the QFT circuit may be leading to lower costs. Pytket-DQC requires its own transpiler pass before distribution, which may include some truncation of small angle rotations, though we have not verified this. While large hyper-edges in QFT circuits leads to slower runtimes for MLFM-R, in practice we would not need to use the FM algorithm for QFT circuits, since the greedy gate grouping pass is sufficient to achieve low entanglement costs.

Fig. 32 displays the results generated from QAOA circuits, in which MLFM-R achieves the lowest entanglement costs. The time taken here is much longer than for FGP-rOEE, but shorter than Pytket PE, which displays erratic performance on these tests.

Fig. 33 shows the results for QV circuits. Previously, QV circuits were identified as difficult circuits to partition using only gate teleportation [64, 66], since they contain no potential for gate grouping. Consequently, methods such as FGP-rOEE, which use only state teleportation, are very effective at distributing QV circuits. Despite this, MLFM-R is still able to outperform

Table 1: Overall comparison of MLFM-R and Pytket PE across all tests completed. We calculate a size independent metric for each circuit, where we divide the mean best e-bit cost achieved by the total number of two-qubit gates in each circuit, averaging over all circuits for each benchmark. A * indicates where we were unable to use Pytket PE due to runtime, so instead used P.

			E-bit Fraction	
Circuit	Partitions	Ref.	MLFM-R	PE
CP	2-12	[26,27,28,29]	0.421	0.414
CP-large*	2-4	[30,30]	0.304	0.358
QAOA	2-4	[32,32]	0.044	0.059
QASM	2-4	[2]	0.038	0.098
QASM-large	2-4	[3]	0.023	0.054
QFT	2-4	[31,31]	0.044	0.046
QV*	2-4	[33,33]	0.133	0.553
Mean	N/A	N/A	0.144	0.222

FGP-rOEE in entanglement costs, highlighting the versatility of the temporal graph construction. From Pytket-DQC, we were only able to use Partition, since PartitionEmbed and EmbedSteinerDetached were too slow for QV circuits and showed no benefit over Partition at small scales. Partition is the slowest method in this case.

Finally, Tab. 2 and Tab. 3 show results from the QASM benchmark suite circuits. For each circuit, we optimise the partitioning over $2, 3$ and 4 QPUs. In the majority of cases, MLFM-R achieves the lowest entanglement cost, though is marginally slower than its competitors in many of these cases. We summarise the comparison between MLFM-R and the next best performing method in Pytket-DQC in Tab. 1. Pytket EmbedSteinerDetached is too slow to run on many of the benchmarks, so we primarily compare with PartitionEmbed. However, in some cases, the embedding refinement was too slow and we had to use just Partition, which usually gives similar results and is significantly faster. These cases are indicated on the table. We use a size-independent metric, in which the optimised entanglement cost in terms of e-bit count is divided by the total number of qubits, giving the *e-bit fraction*. For the e-bit fraction, over all tests, we outperform the next best method by over 35%, which is significant. MLFM-R is often faster than PartitionEmbed and is also faster than Partition for the large benchmarks (though Partition is faster in most other cases). FGP-rOEE is almost always the fastest method, but the limitation of only allowing state teleportation means that the entanglement costs are usually significantly higher, so we do not include the results in Tab. 1.

9 Conclusions

In this work, we introduced and investigated a multilevel framework for partitioning quantum circuits over distributed quantum processing units. We formulated quantum circuit partitioning as a problem of partitioning temporally-extended hypergraphs with an objective that counts entanglement costs from quantum state teleportation, multi-gate teleportation and a new procedure termed *nested state teleportation*. The partitioning is based on the Fiduccia-Mattheyses algorithm – a well-known, efficient heuristic for hypergraph partitioning. We adapted the algorithm to fit the constraints of the problem and designed a unique objective corresponding to entanglement requirements. We illustrated how to incorporate this objective into the algorithm efficiently. We extended the framework to include temporal coarsening of problem hypergraphs, allowing us to partition at multiple resolutions of time. We demonstrated that multilevel partitioning can both reduce runtime and improve solution quality. We were able to significantly outperform state-of-the-art methods in terms of entanglement requirements, for similar or lower runtime in most cases. The results indicate that multilevel partitioning, coupled with a sufficiently general problem formulation, forms a highly effective framework for entanglement minimisation that should be integrated into future distributed

quantum computing compiler stacks.

10 Future work

While the multilevel partitioning framework has proved to be very effective, a number of improvements still need to be made. A limitation of the framework is that communication qubits are only accounted for in the final circuit extraction phase. This means that the algorithm may produce a partitioning that is unfeasible due to a lack of communication qubits. We plan to introduce problem variants where we explicitly account for communication qubits available in the system, while also providing post-processing methods for mitigating communication qubit limitations. Additionally, techniques such as detached gates and embedding of non-local unitaries, identified and used in Refs. [20, 22, 66], could be incorporated into the framework to further reduce entanglement costs.

We note also that the present work has not explicitly considered general quantum networks, with limited connectivity and noisy hardware. This is avoided here, since the focus has been on clearly defining the core partitioning problem. However, we have already addressed many concerns in follow-up work, providing techniques for extending the multilevel framework to large-scale constrained networks [49]. In addition to the current work, these extensions have been incorporated into the `disqco` library, with the hope of providing a complete framework for quantum circuit partitioning and distribution.

A note on fault-tolerance

Finally, we make a brief note on the relevance of this work to the context of fault-tolerant quantum computing. It is likely that small scale implementations of NISQ-era DQC will coincide with small-scale monolithic implementations of fault-tolerant quantum algorithms. However, the future of large-scale quantum computing lies in distributed, fault-tolerant architectures [76]. While we do not explicitly address fault-tolerant DQC in this work, we believe many of the techniques and ideas are directly transferrable. Since we make no assumptions about intra-QPU connectivity and optimise purely for the bottleneck of the system – inter-QPU entanglement distribution – the techniques are directly applicable to fault-tolerant architectures where QPUs host multiple code patches.

In such cases, we can either replace the state and gate teleportation primitives with their transversal, logical, equivalents [77, 78], or use non-local variations of lattice surgery [79]. It has already been demonstrated that lattice surgery, mediated by e-bits, can be used to perform both state and gate teleportation at the logical level [76, 80] and early-stage experimental demonstrations have been performed [77, 81]. There is also active research in optimising non-local lattice surgery to reduce resource requirements and error propagation [82–84].

Other proposals for fault-tolerant DQC consider higher rate codes spanning full modules using specialised communication links [85]. In such cases, logic can be performed locally and non-locally using generalisations to lattice surgery based on the extractors framework [86]. A bolder proposition is to consider individual codes distributed over multiple modules [87, 88], permitting full exploitation of the scaling properties of high-rate codes.

While the mapping is less direct in the latter two cases, inter-module logical operations are likely to be a bottleneck, regardless of whether they occur by interacting multiple code patches using surgery, or interaction of logicals within a single code.

In future work, we aim to explicitly consider fault-tolerant DQC architectures, incorporating the partitioning techniques we have developed into a complete compiler stack for fault-tolerant distributed quantum computing.

Acknowledgements

The authors acknowledge funding from the Engineering and Physical Sciences Research Council (EPSRC) funded Distributed Quantum Computing project, grant number EP/W032643/1.

References

[1] Marcello Caleffi, Michele Amoretti, Davide Ferrari, Jessica Illiano, Antonio Manzalini, and Angela Sara Cacciapuoti. Distributed quantum computing: A survey. *Computer Networks*, 254:110672, December 2024. ISSN 1389-1286. DOI: 10.1016/j.comnet.2024.110672.

[2] Nemanja Isailovic, Yatish Patel, Mark Whitney, and John Kubiatowicz. Interconnection Networks for Scalable Quantum Computers. *ACM SIGARCH Computer Architecture News*, 34(2):366–377, May 2006. ISSN 0163-5964. DOI: 10.1145/1150019.1136505.

[3] M. Akhtar, F. Bonus, F. R. Lebrun-Gallagher, N. I. Johnson, M. Siegele-Brown, S. Hong, S. J. Hile, S. A. Kulmiya, S. Weidt, and W. K. Hensinger. A high-fidelity quantum matter-link between ion-trap microchip modules. *Nature Communications*, 14(1): 531, February 2023. ISSN 2041-1723. DOI: 10.1038/s41467-022-35285-3.

[4] V. Krutyanskiy, M. Galli, V. Krcmarsky, S. Baier, D. A. Fioretto, Y. Pu, A. Mazloom, P. Sekatski, M. Canteri, M. Teller, J. Schupp, J. Bate, M. Meraner, N. Sangouard, B. P. Lanyon, and T. E. Northup. Entanglement of Trapped-Ion Qubits Separated by 230 Meters. *Physical Review Letters*, 130(5):050803, February 2023. DOI: 10.1103/PhysRevLett.130.050803.

[5] Jameson O'Reilly, George Toh, Isabella Goetting, Sagnik Saha, Mikhail Shalaev, Allison L. Carter, Andrew Risinger, Ashish Kalakuntla, Tingguang Li, Ashrit Verma, and Christopher Monroe. Fast Photon-Mediated Entanglement of Continuously Cooled Trapped Ions for Quantum Networking. *Physical Review Letters*, 133(9): 090802, August 2024. DOI: 10.1103/PhysRevLett.133.090802.

[6] Photonic Inc, Francis Afzal, Mohsen Akhlaghi, Stefanie J. Beale, Olinka Bedroya, Kristin Bell, Laurent Bergeron, Kent Bonsma-Fisher, Polina Bychkova, Zachary M. E. Chaisson, Camille Chartrand, Chloe Clear, Adam Darcie, Adam DeAbreu, Colby DeLisle, Lesley A. Duncan, Chad Dundas Smith, John Dunn, Amir Ebrahimi, Nathan Evetts, Daker Fernandes Pinheiro, Patricio Fuentes, Tristen Georgiou, Biswarup Guha, Rafael Haenel, Daniel Higginbottom, Daniel M. Jackson, Navid Jahed, Amin Khorshidahmad, Prasoon K. Shandilya, Alexander T. K. Kurkjian, Nikolai Lauk, Nicholas R. Lee-Hone, Eric Lin, Rostyslav Litynskyy, Duncan Lock, Lisa Ma, Iain MacGilp, Evan R. MacQuarrie, Aaron Mar, Alireza Marefat Khah, Alex Matiash, Evan Meyer-Scott, Cathryn P. Michaels, Juliana Motira, Narwan Kabir Noori, Egor Ospadov, Ekta Patel, Alexander Patscheider, Danny Paulson, Ariel Petruk, Adarsh L. Ravindranath, Bogdan Reznychenko, Myles Ruether, Jeremy Ruscica, Kunal Saxena, Zachary Schaller, Alex Seidlitz, John Senger, Youn Seok Lee, Orbel Sevoyan, Stephanie Simmons, Oney Soykal, Leea Stott, Quyen Tran, Spyros Tserkis, Ata Ulhaq, Wyatt Vine, Russ Weeks, Gary Wolfowicz, and Isao Yoneda. Distributed Quantum Computing in Silicon. *arXiv preprint*, (arXiv:2406.01704), June 2024. DOI: 10.48550/arXiv.2406.01704.

[7] Aziza Almanakly, Beatriz Yankelevich, Max Hays, Bharath Kannan, Réouven Assouly, Alex Greene, Michael Gingras, Bethany M. Niedzielski, Hannah Stickler, Mollie E. Schwartz, Kyle Serniak, Joel Î-j Wang, Terry P. Orlando, Simon Gustavsson, Jeffrey A. Grover, and William D. Oliver. Deterministic remote entanglement using a chiral quantum interconnect. *Nature Physics*, pages 1–6, March 2025. ISSN 1745-2481. DOI: 10.1038/s41567-025-02811-1.

[8] Sagnik Saha, Mikhail Shalaev, Jameson O'Reilly, Isabella Goetting, George Toh, Ashish Kalakuntla, Yichao Yu, and Christopher Monroe. High-fidelity remote entanglement of trapped atoms mediated by time-bin photons. *Nature Communications*, 16(1): 2533, March 2025. ISSN 2041-1723. DOI: 10.1038/s41467-025-57557-4.

[9] Kevin S. Chou, Jacob Z. Blumoff, Christopher S. Wang, Philip C. Reinhold, Christopher J. Axline, Yvonne Y. Gao, L. Frunzio, M. H. Devoret, Liang Jiang, and R. J. Schoelkopf. Deterministic teleportation of a quantum gate between two logical qubits. *Nature*, 561(7723):368–373, September 2018. ISSN 1476-4687. DOI: 10.1038/s41586-018-0470-y.

[10] Yong Wan, Daniel Kienzler, Stephen D. Erickson, Karl H. Mayer, Ting Rei Tan, Jenny J. Wu, Hilma M. Vasconcelos, Scott Glancy, Emanuel Knill, David J. Wineland, Andrew C. Wilson, and Dietrich Leibfried. Quantum gate teleportation between separated qubits in a trapped-ion processor. *Science*, 364(6443):875–878, May 2019. DOI: 10.1126/science.aaw9415.

[11] D. Main, P. Drmota, D. P. Nadlinger, E. M. Ainley, A. Agrawal, B. C. Nichol, R. Srinivas, G. Araneda, and D. M. Lucas. Distributed quantum computing across an optical network link. *Nature*, 638(8050):383–388, February 2025. ISSN 1476-4687. DOI: 10.1038/s41586-024-08404-x.

[12] James Ang, Gabriella Carini, Yanzhu Chen, Isaac Chuang, Michael Demarco, Sophia Economou, Alec Eickbusch, Andrei Faraon, Kai-Mei Fu, Steven Girvin, Michael Hatridge, Andrew Houck, Paul Hilaire, Kevin Krsulich, Ang Li, Chenxu Liu, Yuan Liu, Margaret Martonosi, David McKay, Jim Misewich, Mark Ritter, Robert Schoelkopf, Samuel Stein, Sara Sussman, Hong Tang, Wei Tang, Teague Tomesh, Norm Tubman, Chen Wang, Nathan Wiebe, Yongxin Yao, Dillon Yost, and Yiyu Zhou. ARQUIN: Architectures for Multinode Superconducting Quantum Computers. *ACM Transactions on Quantum Computing*, 5(3):19:1–19:59, September 2024. DOI: 10.1145/3674151.

[13] David Barral, F. Javier Cardama, Guillermo Díaz-Camacho, Daniel Faílde, Iago F. Llovo, Mariamo Mussa-Juane, Jorge Vázquez-Pérez, Juan Villasuso, César Piñeiro, Natalia Costas, Juan C. Pichel, Tomás F. Pena, and Andrés Gómez. Review of Distributed Quantum Computing: From single QPU to High Performance Quantum Computing. *Computer Science Review*, 57: 100747, August 2025. ISSN 1574-0137. DOI: 10.1016/j.cosrev.2025.100747.

[14] Mariam Zomorodi-Moghadam, Mahboobeh Houshmand, and Monireh Houshmand. Optimizing Teleportation Cost in Distributed Quantum Circuits. *International Journal of Theoretical Physics*, 57(3):848–861, March 2018. ISSN 1572-9575. DOI: 10.1007/s10773-017-3618-x.

[15] Pablo Andrés-Martínez and Chris Heunen. Automated distribution of quantum circuits via hypergraph partitioning. *Physical Review A*, 100(3):032308, September 2019. DOI: 10.1103/PhysRevA.100.032308.

[16] Omid Daei, Keivan Navi, and Mariam Zomorodi-Moghadam. Optimized Quantum Circuit Partitioning. *International Journal of Theoretical Physics*, 59(12):3804–3820, December 2020. ISSN 0020-7748, 1572-9575. DOI: 10.1007/s10773-020-04633-8.

[17] Jonathan M. Baker, Casey Duckering, Alexander Hoover, and Frederic T. Chong. Time-sliced quantum circuit partitioning for modular architectures. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, pages 98–107, New York, NY, USA, May 2020. Association for Computing Machinery. ISBN 978-1-4503-7956-4. DOI: 10.1145/3387902.3392617.

[18] Davood Dadkhah, Mariam Zomorodi, and Seyed Ebrahim Hosseini. A New Approach for Optimization of Distributed Quantum Circuits. *International Journal of Theoretical Physics*, 60(9):3271–3285, September 2021. ISSN 1572-9575. DOI: 10.1007/s10773-021-04904-y.

[19] Davide Ferrari, Angela Sara Cacciapuoti, Michele Amoretti, and Marcello Caleffi. Compiler Design for Distributed Quantum Computing. *IEEE Transactions on Quantum Engineering*, 2:1–20, 2021. ISSN 2689-1808. DOI: 10.1109/TQE.2021.3053921.

[20] Ranjani G Sundaram, Himanshu Gupta, and C. R. Ramakrishnan. Efficient Distribution of Quantum Circuits. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41:1–41:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-210-5. DOI: 10.4230/LIPIcs.DISC.2021.41.

[21] Eesa Nikahd, Naser Mohammadzadeh, Mehdi Sedighi, and Morteza Saheb Zamani. Automated window-based partitioning of quantum circuits. *Physica Scripta*, 96(3): 035102, January 2021. ISSN 1402-4896. DOI: 10.1088/1402-4896/abd57c.

[22] Jun-Yi Wu, Kosuke Matsui, Tim Forrer, Akihito Soeda, Pablo Andrés-Martínez, Daniel

Mills, Luciana Henaut, and Mio Murao. Entanglement-efficient bipartite-distributed quantum computing. *Quantum*, 7:1196, December 2023. DOI: 10.22331/q-2023-12-05-1196.

[23] Anbang Wu, Hezi Zhang, Gushu Li, Alireza Shabani, Yuan Xie, and Yufei Ding. AutoComm: A Framework for Enabling Efficient Communication in Distributed Quantum Programs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1027–1041, Chicago, IL, USA, October 2022. IEEE. ISBN 978-1-6654-6272-3. DOI: 10.1109/MICRO56248.2022.00074.

[24] Anbang Wu, Yufei Ding, and Ang Li. QuComm: Optimizing Collective Communication for Distributed Quantum Computing. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 479–493, New York, NY, USA, December 2023. Association for Computing Machinery. ISBN 979-8-4007-0329-4. DOI: 10.1145/3613424.3614253.

[25] Daniele Cuomo, Marcello Caleffi, Kevin Krsulich, Filippo Tramonto, Gabriele Agliardi, Enrico Prati, and Angela Sara Cacciapuoti. Optimized Compiler for Distributed Quantum Computing. *ACM Transactions on Quantum Computing*, 4(2):1–29, June 2023. ISSN 2643-6809, 2643-6817. DOI: 10.1145/3579367.

[26] Davide Ferrari, Stefano Carretta, and Michele Amoretti. A Modular Quantum Compilation Framework for Distributed Quantum Computing. *IEEE Transactions on Quantum Engineering*, 4:1–13, 2023. ISSN 2689-1808. DOI: 10.1109/TQE.2023.3303935.

[27] Pau Escofet, Anabel Ovide, Carmen G. Almudever, Eduard Alarcón, and Sergi Abadal. Hungarian Qubit Assignment for Optimized Mapping of Quantum Circuits on Multi-Core Architectures. *IEEE Computer Architecture Letters*, 22(2):161–164, July 2023. ISSN 1556-6064. DOI: 10.1109/LCA.2023.3318857.

[28] Leo Sünkel, Manik Dawar, and Thomas Gabor. Applying an Evolutionary Algorithm to Minimize Teleportation Costs in Distributed Quantum Computing. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, pages 167–172, September 2024. DOI: 10.1109/QCE60285.2024.10272.

[29] Xinyu Chen, Zilu Chen, Xueyun Cheng, and Zhijin Guan. Circuit Partitioning and Transmission Cost Optimization in Distributed Quantum Computing. *arXiv preprint*, (arXiv:2407.05953), September 2024. DOI: 10.48550/arXiv.2407.05953.

[30] Oliver Crampton, Panagiotis Promponas, Richard Chen, Paul Polakos, Leandros Tassiulas, and Louis Samuel. A Genetic Approach to Minimising Gate and Qubit Teleportations for Multi-Processor Quantum Circuit Distribution. *Journal of Quantum Computing*, 7:1–15, 2025. ISSN 2579-0137, 2579-0145. DOI: 10.32604/jqc.2025.061275.

[31] Waldemir Cambiucci and Regina Melo Silveira. Hypergraphic Partitioning Framework for Static and Adaptive Quantum Circuits. In *2025 23rd International Symposium on Network Computing and Applications (NCA)*, pages 174–181, November 2025. DOI: 10.1109/NCA67271.2025.00037.

[32] Panagiotis Promponas, Akrit Mudvari, Luca Della Chiesa, Paul Polakos, Louis Samuel, and Leandros Tassiulas. Compiler for Distributed Quantum Computing: A Reinforcement Learning Approach. In *ICC 2025 - IEEE International Conference on Communications*, pages 4615–4621, June 2025. DOI: 10.1109/ICC52391.2025.11161115.

[33] Eneet Kaur, Hassan Shapourian, Jiapeng Zhao, Michael Kilzer, Ramana Kompella, and Reza Nejabati. Optimized quantum circuit partitioning across multiple quantum processors. In *Quantum Computing, Communication, and Simulation V*, volume 13391, pages 152–156. SPIE, March 2025. DOI: 10.1117/12.3042502.

[34] Pau Escofet, Anabel Ovide, Medina Bandic, Luise Prielinger, Hans van Someren, Sebastian Feld, Eduard Alarcon, Sergi Abadal, and Carmen Almudever. Revisiting the Mapping of Quantum Circuits: Entering the Multi-core Era. *ACM Transactions on Quantum Computing*, 6(1):4:1–4:26, January 2025. DOI: 10.1145/3655029.

[35] Enrico Russo, Elio Vinciguerra, Maurizio Palesi, Davide Patti, Giuseppe Ascia, and Vincenzo Catania. TeleSABRE: Layout Synthesis in Multi-Core Quantum Systems with Teleport Interconnect. *arXiv preprint*, (arXiv:2505.08928), May 2025. DOI: 10.48550/arXiv.2505.08928.

[36] Yingling Mao, Yu Liu, and Yuanyuan Yang. Qubit Allocation for Distributed Quantum Computing. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, May 2023. DOI: 10.1109/INFOCOM53939.2023.10228915.

[37] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2): 291–307, February 1970. ISSN 0005-8580. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[38] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181, June 1982. DOI: 10.1109/DAC.1982.1585498.

[39] L. A. Sanchis. Multiple-Way Network Partitioning. *IEEE Trans. Comput.*, 38(1):62–81, January 1989. ISSN 0018-9340. DOI: 10.1109/12.8730.

[40] Jason Cong and M'Lissa Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In *Proceedings of the 30th International on Design Automation Conference - DAC '93*, pages 755–760, Dallas, Texas, United States, 1993. ACM Press. ISBN 978-0-89791-577-9. DOI: 10.1145/157485.165119.

[41] Frank M. Johannes. Partitioning of VLSI circuits and systems. In *Proceedings of the 33rd Annual Conference on Design Automation Conference - DAC '96*, pages 83–87, Las Vegas, Nevada, United States, 1996. ACM Press. ISBN 978-0-89791-779-7. DOI: 10.1145/240518.240535.

[42] Dennis J.-H. Huang and Andrew B. Kahng. Partitioning-based standard-cell global placement with an exact objective. In *Proceedings of the 1997 International Symposium on Physical Design*, ISPD '97, pages 18–25, New York, NY, USA, April 1997. Association for Computing Machinery. ISBN 978-0-89791-927-2. DOI: 10.1145/267665.267674.

[43] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 526–529, New York, NY, USA, June 1997. Association for Computing Machinery. ISBN 978-0-89791-920-3. DOI: 10.1145/266021.266273.

[44] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1998. ISSN 1064-8275, 1095-7197. DOI: 10.1137/S1064827595287997.

[45] George Karypis and Vipin Kumar. Multilevel k -way hypergraph partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, pages 343–348, New Orleans Louisiana USA, June 1999. ACM. ISBN 978-1-58113-109-3. DOI: 10.1145/309847.309954.

[46] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning Complex Networks via Size-constrained Clustering. *arXiv preprint*, (arXiv:1402.3281), March 2014. DOI: 10.48550/arXiv.1402.3281.

[47] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-Quality Hypergraph Partitioning. *ACM J. Exp. Algorithmics*, 27:1.9:1–1.9:39, February 2023. ISSN 1084-6654. DOI: 10.1145/3529090.

[48] George Karypis and Vipin Kumar. Metis—a software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing ordering of sparse matrices. 01 1997.

[49] Felix Burt, Kuan-Cheng Chen, and Kin K. Leung. Entanglement-efficient distribution of quantum circuits over large-scale quantum networks. In *2025 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1111–1122, 2025. DOI: 10.1109/QCE65121.2025.00125.

[50] Felix Burt. Felix-burt/DISQCO. https://github.com/felix-burt/DISQCO, 2025.

[51] Tianyi Peng, Aram Harrow, Maris Ozols, and Xiaodi Wu. Simulating Large Quantum

Circuits on a Small Quantum Computer. *Physical Review Letters*, 125(15):150504, October 2020. ISSN 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.125.150504.

[52] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. CutQC: Using small Quantum computers for large Quantum circuit evaluations. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 473–486, New York, NY, USA, April 2021. Association for Computing Machinery. ISBN 978-1-4503-8317-2. DOI: 10.1145/3445814.3446758.

[53] Wei Tang and Margaret Martonosi. Cutting Quantum Circuits to Run on Quantum and Classical Platforms. *arXiv preprint*, (arXiv:2205.05836), May 2022. DOI: 10.48550/arXiv.2205.05836.

[54] Saikat Basu, Arnav Das, Amit Saha, Amlan Chakrabarti, and Susmita Sur-Kolay. *FragQC*: An efficient quantum error reduction technique using quantum circuit fragmentation. *Journal of Systems and Software*, 214:112085, August 2024. ISSN 0164-1212. DOI: 10.1016/j.jss.2024.112085.

[55] Teague Tomesh, Zain H. Saleem, Michael A. Perlin, Pranav Gokhale, Martin Suchara, and Margaret Martonosi. Divide and Conquer for Combinatorial Optimization and Distributed Quantum Computation. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1–12, September 2023. DOI: 10.1109/QCE57702.2023.00009.

[56] Marvin Bechtold, Johanna Barzen, Frank Leymann, Alexander Mandl, Julian Obst, Felix Truger, and Benjamin Weder. Investigating the effect of circuit cutting in QAOA for the MaxCut problem on NISQ devices. *Quantum Science and Technology*, 8(4):045022, October 2023. ISSN 2058-9565. DOI: 10.1088/2058-9565/acf59c.

[57] Turbasu Chatterjee, Arnav Das, Shah Ishmam Mohtashim, Amit Saha, and Amlan Chakrabarti. Qurzon: A Prototype for a Divide and Conquer-Based Quantum Compiler for Distributed Quantum Systems. *SN Computer Science*, 3(4):323, June 2022. ISSN 2661-8907. DOI: 10.1007/s42979-022-01207-9.

[58] Sebastian Brandhofer, Ilia Polian, and Kevin Krsulich. Optimal Partitioning of Quantum Circuits Using Gate Cuts and Wire Cuts. *IEEE Transactions on Quantum Engineering*, 5:1–10, 2024. ISSN 2689-1808. DOI: 10.1109/TQE.2023.3347106.

[59] David P. DiVincenzo and IBM. The Physical Implementation of Quantum Computation. *Fortschritte der Physik*, 48 (9-11):771–783, September 2000. ISSN 00158208, 15213978. DOI: 10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E.

[60] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, 70(13):1895–1899, March 1993. DOI: 10.1103/PhysRevLett.70.1895.

[61] Daniel Gottesman and Isaac L. Chuang. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature*, 402 (6760):390–393, November 1999. ISSN 1476-4687. DOI: 10.1038/46503.

[62] J. Eisert, K. Jacobs, P. Papadopoulos, and M. B. Plenio. Optimal local implementation of nonlocal quantum gates. *Physical Review A*, 62(5):052317, October 2000. DOI: 10.1103/PhysRevA.62.052317.

[63] Anocha Yimsiriwattana and Samuel J. Lomonaco Jr. Generalized GHZ States and Distributed Quantum Computing. *arXiv preprint*, (arXiv:quant-ph/0402148), March 2004.

[64] Felix Burt, Kuan-Cheng Chen, and Kin K. Leung. Generalised Circuit Partitioning for Distributed Quantum Computing. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, pages 173–178, September 2024. DOI: 10.1109/QCE60285.2024.10273.

[65] Anocha Yimsiriwattana and Samuel J. Lomonaco Jr. Distributed quantum computing: A distributed Shor algorithm. In Eric Donkor, Andrew R. Pirich, and Howard E. Brandt, editors, *Defense and Security*, page

360, Orlando, FL, August 2004. DOI: 10.1117/12.546504.

[66] Pablo Andres-Martinez, Tim Forrer, Daniel Mills, Jun-Yi Wu, Luciana Henaut, Kentaro Yamamoto, Mio Murao, and Ross Duncan. Distributing circuits over heterogeneous, modular quantum computing network architectures. *Quantum Science and Technology*, 9(4):045021, August 2024. ISSN 2058-9565. DOI: 10.1088/2058-9565/ad6734.

[67] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit. *arXiv preprint*, (arXiv:2405.08810), June 2024. DOI: 10.48550/arXiv.2405.08810.

[68] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. T|ket⟩: A retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, November 2020. ISSN 2058-9565. DOI: 10.1088/2058-9565/ab8e92.

[69] Ranjani G. Sundaram, Himanshu Gupta, and C. R. Ramakrishnan. Distribution of Quantum Circuits Over General Quantum Networks. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 415–425, September 2022. DOI: 10.1109/QCE53715.2022.00063.

[70] Giorgio Ausiello and Luigi Laura. Directed hypergraphs: Introduction and fundamental algorithms—A survey. *Theoretical Computer Science*, 658:293–306, January 2017. ISSN 0304-3975. DOI: 10.1016/j.tcs.2016.03.016.

[71] Hyunho Cha and Jungwoo Lee. Module-conditioned distribution of quantum circuits. *arXiv preprint*, (arXiv:2501.11816), January 2025. DOI: 10.48550/arXiv.2501.11816.

[72] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, September 2019. ISSN 2469-9926, 2469-9934. DOI: 10.1103/PhysRevA.100.032328.

[73] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A Quantum Approximate Optimization Algorithm. *arXiv preprint*, (arXiv:1411.4028), November 2014. DOI: 10.48550/arXiv.1411.4028.

[74] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2), February 2023. DOI: 10.1145/3550488. URL https://doi.org/10.1145/3550488.

[75] Pablo Andres-Martinez, Daniel Mills, Tim Forrer, and Luciana Henaut. CQCL/pytket-dqc. Quantinuum Ltd, June 2024.

[76] Joshua Ramette, Josiah Sinclair, Nikolas P. Breuckmann, and Vladan Vuletić. Fault-tolerant connection of error-corrected qubits with noisy links. *npj Quantum Information*, 10(1):1–6, June 2024. ISSN 2056-6387. DOI: 10.1038/s41534-024-00855-4.

[77] C. Ryan-Anderson, N. C. Brown, C. H. Baldwin, J. M. Dreiling, C. Foltz, J. P. Gaebler, T. M. Gatterman, N. Hewitt, C. Holliman, C. V. Horst, J. Johansen, D. Lucchetti, T. Mengle, M. Matheny, Y. Matsuoka, K. Mayer, M. Mills, S. A. Moses, B. Neyenhuis, J. Pino, P. Siegfried, R. P. Stutz, J. Walker, and D. Hayes. High-fidelity teleportation of a logical qubit using transversal gates and lattice surgery. *Science*, 385(6715):1327–1331, September 2024. DOI: 10.1126/science.adp6016.

[78] John Stack, Ming Wang, and Frank Mueller. Assessing Teleportation of Logical Qubits in a Distributed Quantum Architecture under Error Correction. *arXiv pre-print*, (arXiv:2504.05611), April 2025. DOI: 10.48550/arXiv.2504.05611.

[79] Dominic Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14(12): 123011, December 2012. ISSN 1367-2630. DOI: 10.1088/1367-2630/14/12/123011.

[80] Áron Márton, Luis Colmenarez, Lukas Bödeker, and Markus Müller. Lattice surgery-based logical state teleportation via noisy links. *Phys. Rev. Res.*, 7: 033238, Sep 2025. DOI: 10.1103/ppng-vbqj. URL https://link.aps.org/doi/10.1103/ppng-vbqj.

[81] Alexander Erhard, Hendrik Poulsen Nautrup, Michael Meth, Lukas Postler,

Roman Stricker, Martin Ringbauer, Philipp Schindler, Hans J. Briegel, Rainer Blatt, Nicolai Friis, and Thomas Monz. Entangling logical qubits with lattice surgery. *Nature*, 589(7841):220–224, January 2021. ISSN 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03079-6.

[82] Trond Hjerpekjøn Haug, Timo Hillmann, Anton Frisk Kockum, and Raphaël Van Laer. Lattice surgery with Bell measurements: Modular fault-tolerant quantum computation at low entanglement cost. *arXiv pre-print*, (arXiv:2510.13541), October 2025. DOI: 10.48550/arXiv.2510.13541.

[83] Hugo Jacinto, Élie Gouzien, and Nicolas Sangouard. Network Requirements for Distributed Quantum Computation. *arXiv pre-print*, (arXiv:2504.08891), April 2025. DOI: 10.48550/arXiv.2504.08891.

[84] Mohamed A. Shalby, Renyu Wang, Denis Sedov, and Leonid P. Pryadko. Optimized noise-resilient surface code teleportation interfaces. *Physical Review A*, 112(2):L020403, August 2025. DOI: 10.1103/xqrn-wdw1.

[85] Theodore J. Yoder, Eddie Schoute, Patrick Rall, Emily Pritchett, Jay M. Gambetta, Andrew W. Cross, Malcolm Carroll, and Michael E. Beverland. Tour de gross: A modular quantum computer based on bivariate bicycle codes. *arXiv pre-print*, (arXiv:2506.03094), June 2025. DOI: 10.48550/arXiv.2506.03094.

[86] Zhiyang He, Alexander Cowtan, Dominic J. Williamson, and Theodore J. Yoder. Extractors: QLDPC Architectures for Efficient Pauli-Based Computation. *arXiv pre-print*, (arXiv:2503.10390), March 2025. DOI: 10.48550/arXiv.2503.10390.

[87] Evan Sutcliffe, Bhargavi Jonnadula, Claire Le Gall, Alexandra E. Moylett, and Coral M. Westoby. Distributed quantum error correction based on hyperbolic floquet codes. In *2025 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 649–657, 2025. DOI: 10.1109/QCE65121.2025.00076.

[88] Oscar Higgott and Nikolas P. Breuckmann. Constructions and Performance of Hyperbolic and Semi-Hyperbolic Floquet Codes. *PRX Quantum*, 5(4):040327, November 2024. ISSN 2691-3399. DOI: 10.1103/PRXQuantum.5.040327.

A FM implementation

A.1 Auxiliary functions

Here we include some of the auxiliary functions used in the FM algorithm. Alg. 5 contains functions used for initialising the main algorithm. This includes calculating the gains for all moves and converting them into a structure which allows for easy retrieval and updates. We omit the basic helper functions for gain calculation and cost retrieval, since these are defined in the main text.

Algorithm 5: Initial routines for FM

Input:
$H(V, E; \tau, \kappa)$: Temporal hypergraph extracted from a quantum circuit.
$\mathcal{C}$: Precomputed cost table for hyper-edge configurations.
$\Phi : V \to \{1, \dots, K\}$: Current partition assignment.
K: Number of partitions (QPUs).

ComputeAllGains (H, Φ, K, $\mathcal{C}$)**:**
- Construct the gain function $\Gamma : V \times \{1, \dots, K\} \to \mathbb{R} \cup \{\infty\}$.
- **foreach** $v \in V$ **do**
  - **for** $p \leftarrow 1$ **to** K **do**
    - **if** $p = \Phi(v)$ **then**
      - $\Gamma(v, p) \leftarrow \infty$ `// Moving to current partition is disallowed`
    - **else**
      - $\Gamma(v, p) \leftarrow$ `Gain`$(v, p, \Phi, \mathcal{C})$
- **return** Γ

BuildGainBuckets (Γ)**:**
- Build bucket structure $\mathcal{B}$ grouping moves (v, p) by their gain g.
- `//` $\mathcal{B}(g)$ `denotes the set of moves with gain` g
- $\mathcal{B} \leftarrow \emptyset$
- **foreach** (v, p) with $\Gamma(v, p) \neq \infty$ **do**
  - $g \leftarrow \Gamma(v, p)$
  - **if** $g \notin \mathrm{dom}(\mathcal{B})$ **then**
    - $\mathcal{B}(g) \leftarrow \emptyset$
  - $\mathcal{B}(g) \leftarrow \mathcal{B}(g) \cup \{(v, p)\}$
- **return** $\mathcal{B}$

Alg. 6 contains the sub-routines for choosing the best moves at each iteration, as well as the `UpdateGains` function, which is the bulk of the computation in the main FM algorithm.

Algorithm 6: Auxiliary routines for FM

Input:
$H(V, E; \tau, \kappa)$: Temporal hypergraph extracted from a quantum circuit.
$\Gamma : V \times \{1, \dots, K\} \to \mathbb{R} \cup \{\infty\}$: Gain function.
$\mathcal{B}$: Gain bucket structure built from Γ.
$\mathcal{C}$: Precomputed cost table for hyper-edge configurations.
$\Phi : V \to \{1, \dots, K\}$: Current partition assignment.
$\mathcal{X} \subseteq V$: Set of locked nodes.
$\mathrm{Cap} : \{1, \dots, K\} \to \mathbb{N}$: Capacity limits for each partition.

BestMove ($H, \mathcal{B}$, $\mathcal{X}$, Cap, Φ)**:**
- Select the best admissible move according.;
- **foreach** g *in gains of* $\mathcal{B}$ *ascending* **do**
  - **while** $\mathcal{B}(g) \neq \emptyset$ **do**
    - $(v^\star, p^\star) \leftarrow$ `ChooseRandom`$(\mathcal{B}(g))$;
    - $\mathcal{B}(g) \leftarrow \mathcal{B}(g) \setminus \{(v^\star, p^\star)\}$;
    - **if** $v^\star \notin \mathcal{X}$ and $|\{u \in V^{(\tau(v^\star))} | \Phi(u) = p^\star\}| < \mathrm{Cap}[p]$ **then**
      - **return** $(v^\star, p^\star)$;
- **return** $\perp$; `// No valid moves`

UpdateGains (H, Γ, $\mathcal{B}$, $v^\star$, $p^\star$, Φ, $\mathcal{C}$)**:**
- Update the gain function after moving $v^\star$ to $p^\star$.;
- **foreach** $e \in E$ *with* $v^\star \in e$ **do**
  - **foreach** $u \in e$ *with* $u \neq v^\star$ **do**
    - **for** $p \leftarrow 1$ **to** K **do**
      - $\delta \leftarrow$ `DeltaGainContrib`$(u, e, p, \Phi, \mathcal{C})$;
      - $g_{\text{old}} \leftarrow \Gamma(u, p)$;
      - **if** $g_{old} \neq \infty$ **then**
        - $\mathcal{B}(g_{\text{old}}) \leftarrow \mathcal{B}(g_{\text{old}}) \setminus \{(u, p)\}$;
      - $\Gamma(u, p) \leftarrow \Gamma(u, p) + \delta$;
      - $g_{\text{new}} \leftarrow \Gamma(u, p)$;
      - **if** $g_{new} \neq \infty$ **then**
        - $\mathcal{B}(g_{\text{new}}) \leftarrow \mathcal{B}(g_{\text{new}}) \cup \{(u, p)\}$;

There is some freedom involved in choosing the best move given that it is common to have multiple moves with the same gain. Here, we choose randomly among the best moves, which helps to add some stochasticity to the algorithm, potentially avoiding poor local minima. The gain update function is where the structure of our algorithm differs most from standard FM implementations. Notably, since gains are calculated from differences in edge costs, which depend on hyper-edge configurations, we update the gains on a per-edge basis. This means, after a given move, we first identify which edges are affected by the move. We then iterate over the nodes in each edge, calculating the delta gain contribution from that edge to each node's gain for each possible move. This is slightly different to normal FM implementations, which often find the neighbours of the moved node first, then calculate the delta gains for each neighbour. Since we iterate over hyper-edges first, we avoid redundant calculations from edges which are not affected by the move.

At first glance, it may seem that we are double counting some contribution, since each neighbour may be involved in multiple edges with the moved node. However, since we are calculating the delta gain contribution from each edge separately, we recover the correct total delta gain for each neighbour after summing over all edges, as seen in Eq. 26. After this, we proceed to resort the gain buckets according to these changes. Since the calculation of the delta gain contributions is somewhat involved, we dedicate Alg. 7 to describing this process.

The bulk of these routines consists in retrieving and updating the hyper-edge configurations for each possible move. Asymptotically, this is efficient, since we can update the configurations and retrieve the costs in constant time, but the implementation involves retrieving and updating a number of different objects. As alluded to in the main text, in addition to storing and updating the root and receiver configurations, we also store the root and receiver *counts* objects, which track the number of nodes in each partition for the root and receiver sets, respectively. For each move, we increment the counts at the source and the destination of the move, then check whether the counts have changed from zero to non-zero or vice versa, indicating that the configuration

Algorithm 7: Delta-gain calculation for a single edge e

DeltaGainContrib (v, e, p, Φ, $\mathcal{C}$)**:**
- Compute $\delta_e(\Phi, \Phi') = c_e(\Phi') - c_e(\Phi)$ for the move $v : \Phi(v) \to p$. `// 1. Retrieve root/receiver multiplicities and total edge configuration`
- $(\mathbf{r}_e, \mathbf{s}_e) \leftarrow$ `RetrieveCounts`(e);
- $\text{cfg}_{\text{old}} \leftarrow$ `RetrieveCfg`(e);
- $c_{\text{old}} \leftarrow \mathcal{C}(\text{cfg}_{\text{old}})$;
- $\mathbf{r}_e[\Phi(v)] \leftarrow \mathbf{r}_e[\Phi(v)] - 1$;
- $\mathbf{r}_e[p] \leftarrow \mathbf{r}_e[p] + 1$;
- `// (Receiver counts` $\mathbf{s}_e$ `updated analogously if` v `is a receiver node)`
- $\text{cfg}_{\text{new}} \leftarrow$ `UpdateCfg`$(\mathbf{r}_e, \mathbf{s}_e, \text{cfg}_{\text{old}})$;
- $c_{\text{new}} \leftarrow \mathcal{C}(\text{cfg}_{\text{new}})$;
- **return** $c_{\text{new}} - c_{\text{old}}$;

UpdateConfig ($\mathbf{r}'_e$, $\mathbf{s}'_e$, src, p)**:**
- `// Update` $cfg^{(e)}(\Phi)$ `after moving node from src to p`
- **if** $\mathbf{s}'_e[\text{src}] = 0$ **then**
  - Cfg[src] $\leftarrow 0$;
- **else**
  - **if** $\mathbf{r}'_e[\text{src}] = 0$ **then**
    - Cfg[src] $\leftarrow 1$;
  - **else**
    - Cfg[src] $\leftarrow 0$;
- **if** $\mathbf{s}'_e[\text{p}] = 0$ **then**
  - Cfg[p] $\leftarrow 0$;
- **else**
  - **if** $\mathbf{r}'_e[\text{p}] = 0$ **then**
    - Cfg[p] $\leftarrow 1$;
  - **else**
    - Cfg[p] $\leftarrow 0$;
- **return** Cfg;

must change too. Importantly, for each move, since nodes are either in the root or receiver sets, we need only update the root or receiver counts. Furthermore, since only the entries for the source and destination partition change, we only update these entries. For the full process, we need the edge costs for Φ, Φ', $\tilde{\Phi}$ and $\tilde{\Phi}'$. In each case, $'$ corresponds to the node which has actually been moved, whereas $\tilde{}$ corresponds to the prospective move for which we are updating the delta gain. In this way, $\tilde{\Phi}$ corresponds to the prospective move being made *before* the actual node was moved. This means that, for each prospective move, we need to update the configuration for two assignment functions, corresponding to $\tilde{\Phi}$ and $\tilde{\Phi}'$. We already have the costs for Φ and Φ', since Φ was already stored, and Φ' is updated and stored after the completed move was chosen.

A.2 Pre-computation costs

Here we briefly address the issue of pre-computation overhead. The efficiency guarantee of FM for our custom cost function is reliant on the fact that we can pre-compute the cost of all edge configurations in pre-processing. However, the number of configurations is exponential in the number of partitions, and so this is only feasible for less than 20 partitions. This will not be an issue for most early stage quantum networks, though in the future we will want algorithms that can handle this.

There are a number of things that could be done to mitigate this. We may choose to omit the pre-computation, and rather store the cost of each configuration as it appears in computation, such that we need only perform the calculation once per configuration. In fact, it is often better to do this in practice, since many configurations will never appear in the computation, and so we can save on both time and memory by only storing the configurations that are actually used. Alternatively, we may choose to perform the partitioning *recursively*. In contrast to the recursive coarsening spoken about in the main text, recursive partitioning refers to multi-partitioning techniques where we partition a graph for a small number of partitions, then continue to partition the resulting subgraphs until we reach the desired number of partitions. This is also a common technique in many large-scale graph partitioning algorithms. Such techniques become necessary not only when the number of partitions is large, but also when the network has constrained connectivity, such that not all partitions are directly connected to each other. While we do not address this case here, it is clear that this case requires additional calculation to accurately model costs, since edges which span multiple partitions may need to be routed through intermediate partitions. We have addressed these issues in follow-up work, showing that recursive methods permit scaling to constrained networks of over 50 QPUs [49].

B Coarsening

B.1 Temporal coarsening strategies

Here we detail the algorithms employed for the coarsening process in the FM algorithm. Since we perform only temporal contraction, we simply need to track which time steps have been contracted in order to transform a coarse solution to a finer solution. After partitioning nodes in a coarsened graph, we read from the contraction path which nodes will be uncoarsened at the next level, and assign them the same partition as the parent node from which they have been uncoarsened. Alg. 8 details the window-based coarsening strategy, which coarsens nodes in a window of size w at each time step. The window is moved along the time axis, coarsening nodes in the window at each step. We set the input to the algorithm as the number of levels, from which we calculate the window size. This allows us to restrict the number of levels to a fixed number. As mentioned briefly in Sec. 4, after nodes have been uncoarsened and refined once, we lock them in place to reduce the number of nodes which are moved at each level. This number is then proportional to $n_q w$, such that the uncoarsening acts like a moving window scanning and partitioning the nodes over time. Alg. 9 details the block-based coarsening strategy, which instead identifies blocks of contiguous nodes to coarsen into a single node. Similarly to the window method, we would like to be able to set the number of levels. This determines the number of blocks. We then coarsen the blocks one time step per-level, such that the resulting number of levels is the number of time steps in each block. We also choose to add an extra level, which performs a full coarsening of the final blocks to a single time step. Alg. 4 details

the recursive coarsening strategy, which coarsens pairs of neighbouring nodes at each time step, such that the total number of nodes is halved at each time step. This turns out to be the most effective strategy and is also leads to the fastest results when we don't cap the number of nodes moved per pass. We conjecture that this is because, at each level, all nodes are roughly as coarse as each other, i.e. each node "contains" roughly the same number of nodes. This means that the gains will not be skewed towards large super-nodes, thus controlling the partitioning at an appropriate level of granularity.

Algorithm 8: Window coarsening

Input:
$H_0 = (V_0, E_0, \tau, \kappa)$: temporal hypergraph with time-layers $1, \ldots, d$.
L: number of coarsening levels.
Output:
Coarsening hierarchy $\mathcal{H} = (H_0, H_1, \ldots, H_M)$ for some $M \leq L$.

CoarsenWindow (H_0, d, L)**:**

```
H ← (H_0);
H ← H_0;
w ← ⌊d/L⌋;                    // Window size
ℓ ← d;
while ℓ > 1 do
    ℓ_start ← max(1, ℓ − w);
    for ℓ' ← ℓ down to ℓ_start + 1 do
        H' ← ContractTime(H, ℓ', ℓ' − 1);
        H ← H';
    H ← H ‖ H;
    ℓ ← ℓ − w;
return H;
```

Algorithm 9: Block coarsening

Input:
$H_0 = (V_0, E_0, \tau, \kappa)$: temporal hypergraph with time-layers $1, \ldots, d$.
L: number of coarsening levels (blocks).
Output:
Coarsening hierarchy $\mathcal{H} = (H_0, H_1, \ldots, H_M)$ for some $M \leq L$.

CoarsenBlocks (H_0, d, L)**:**

```
H ← (H_0)
H ← H_0
b ← ⌊d/L⌋                     // Block size
ℓ ← d
for ℓ − 1 > d − b do
    t ← ℓ
    while t > 1 do
        H' ← ContractTime(H, t, t − 1)
        H ← H'
        t ← t − b
    H ← H ‖ H
    ℓ ← ℓ − 1 // Slide block window by one layer
// Optionally merge all blocks into one layer
return H
```

B.2 Spatial coarsening

While the paper explores only "temporal" coarsening, i.e., coarsening along the time axis, we briefly discuss the idea of spatial coarsening and how it could be incorporated in future work. Spatial coarsening corresponds to coarsening gate-like edges, merging the nodes involved together. Consider first a basic graph, for which no gate grouping has occurred. Contracting a gate-like edge merges nodes from two qubits at the same time step together, such that partitioning of this node determines exactly where this gate will be performed, locally. In this way, if all gate edges are contracted, any partitioning of the nodes results in all gates are covered by state teleportation, since there are no gate-edges to cut. For circuits where state teleportation methods are most effective, such as quantum volume, this would be a useful strategy. However, in many other cases, this would be a limitation, as we are removing the possibility of gate grouping. When we consider hypergraphs, there is more flexibility involved in spatial coarsening. Since hyper-edges can extend over multiple time steps, we could choose to contract nodes which are not at the same time step. However, this makes the process of enforcing capacity constraints more complicated, since these constraints are defined at each time step.

Alternatively, we may want to consider spatial coarsening after a full coarsening along the time axis. For example, if we have a very large circuit, with over 1000 qubits, then we still have a 1000

nodes after coarsening along the time axis. We can reduce the problem size by performing further coarsening along the spatial axis, merging qubit nodes together. This could be a useful strategy for dealing with very large circuits, though we do not investigate this here. If we are including gate grouping as well, there are certain considerations to make, since hyper-edges will extend over a number of time steps.

We leave a full investigation of integrating spatial and temporal coarsening to future work, though we note that many compiler problems in quantum computing, such as qubit routing, may benefit from both spatial and temporal coarsening strategies.

C QASM results tables

In Tab. 2 and Tab. 3, we show the results of the QASM benchmark suite circuits [74], partitioned over 2, 3 and 4 QPUs. The results are generated using the exploratory MLFM-R algorithm with a cap of n_q on the number of nodes moved per pass, for 10 passes, as for other results. Note that we transpile circuits into the $U(\theta, \phi, \lambda), CP(\theta)$ gate set before partitioning, using the *Qiskit* transpiler [67], though Pytket-DQC will run additional transpilation for compatibility with its distributors. We test circuits from the *larger* section of the benchmark suite, though omit circuits of depth greater than 1000 after transpilation. Additionally, we omit QFT, QAOA and QV circuits, since they have already been tested.

Table 2: Comparison of entanglement cost and time taken for QASM benchmark suite circuits.

Circuit	Partitions	P Cost	PE Cost	ESD Cost	MLFM-R Cost	P Time	PE Time	ESD Time	MLFM-R Time
adder28	2	4	3.6	3	2.2	0.079	0.205	0.699	1.257
adder28	3	2	2	5	3.3	0.07	0.118	0.712	2.039
adder28	4	12	10.8	9	7.3	0.08	0.226	0.694	3.102
bv30	2	1	1	1	1	0.008	0.01	0.02	0.203
bv30	3	2	2	2	2	0.01	0.011	0.02	0.299
bv30	4	2.6	2.6	3	2.8	0.011	0.013	0.023	0.433
knn31	2	2.4	2.4	4	1	0.049	0.075	0.719	0.956
knn31	3	4.4	4.4	5	2	0.066	0.093	0.429	1.754
knn31	4	10.8	10.8	21	3.5	0.098	0.127	0.643	2.345
cc32	2	1	1	1	1	0.008	0.012	0.034	0.758
cc32	3	2.2	2.2	2	2	0.018	0.022	0.031	1.255
cc32	4	3	3	3	3	0.013	0.016	0.033	1.734
dnn33	2	4	4	4	4	0.095	0.179	1.115	1.165
dnn33	3	11	11	11	7	0.107	0.201	0.987	1.908
dnn33	4	28.8	28.8	36	9.3	0.221	0.318	1.111	2.757
ising34	2	1	1	1	1	0.023	0.034	0.111	0.073
ising34	3	6.6	6.6	9	2.6	0.03	0.042	0.116	0.109
ising34	4	13.4	13.4	17	3.5	0.035	0.049	0.124	0.144
cat35	2	1	1	1	1	0.01	0.014	0.025	0.354
cat35	3	9	9	25	2	0.028	0.032	0.045	0.573
cat35	4	9	9	23	3	0.027	0.032	0.045	0.751
wstate36	2	2	2	2	1	0.028	0.039	0.119	0.414
wstate36	3	4	4	4	3	0.033	0.044	0.124	0.574
wstate36	4	6	6	6	3	0.036	0.047	0.129	0.88
qugan39	2	5	5	5	4	0.121	0.233	2.092	1.411
qugan39	3	12	12	12	8.4	0.13	0.255	1.901	2.152
qugan39	4	22.6	21.6	21	14.1	0.169	0.448	1.764	3.29
ghz40	2	1	1	1	1	0.012	0.017	0.032	0.462
ghz40	3	4.4	4.4	14	2	0.023	0.028	0.045	0.743
ghz40	4	3	3	3	3	0.017	0.022	0.037	0.981
knn41	2	1	1	1	1	0.073	0.118	2.044	2.028
knn41	3	5	5	5	2.1	0.092	0.14	1.252	3.441
knn41	4	10.8	10.8	15	3	0.159	0.209	0.94	5.069
swaptest41	2	1	1	1	1	0.07	0.113	2.021	2.028
swaptest41	3	5	5	5	2.1	0.089	0.135	1.233	3.441
swaptest41	4	9.2	9.2	15	3	0.125	0.173	0.904	5.069
ising42	2	1	1	1	1	0.031	0.048	0.176	0.101
ising42	3	2	2	2	2.2	0.035	0.053	0.182	0.166
ising42	4	14.4	14.4	19	3.6	0.049	0.068	0.184	0.179

Table 3: Comparison of entanglement cost and time taken for larger QASM benchmark suite circuits.

Circuit	Partitions	P Cost	PE Cost	ESD Cost	MLFM-R Cost	P Time	PE Time	ESD Time	MLFM-R Time
dnn51	2	4	4	4	3.2	0.199	0.395	5.024	2.66
dnn51	3	11	10.2	10	7.1	0.211	0.652	3.103	4.897
dnn51	4	15	15	15	14.1	0.229	0.456	3.278	6.53
adder64	2	4	3.2	3	4.6	0.276	0.858	5.765	7.269
adder64	3	9	8.4	8	9.8	0.284	0.883	5.78	13.66
adder64	4	12	10.8	9	13.1	0.295	0.919	5.73	18.543
cc64	2	1	1	1	1	0.025	0.039	0.163	3.6
cc64	3	2	2	2	2	0.032	0.046	0.15	6.424
cc64	4	3	3	3	3	0.033	0.047	0.133	8.608
cat65	2	1	1	1	1	0.028	0.041	0.075	1.4
cat65	3	3.8	3.8	4	2	0.04	0.053	0.084	2.453
cat65	4	8	8	21	3	0.051	0.064	0.123	3.1
ising66	2	1	1	1	1.2	0.071	0.113	0.505	0.153
ising66	3	2	2	2	2.4	0.076	0.118	0.502	0.232
ising66	4	3	3	3	4	0.078	0.12	0.506	0.297
knn67	2	1	1	1	1	0.187	0.309	11.154	6.218
knn67	3	2	2	2	2	0.199	0.321	8.785	12.465
knn67	4	6	6	6	5	0.215	0.339	4.692	16.765
bv70	2	1	1	1	1	0.017	0.025	0.072	0.957
bv70	3	1	1	1	2	0.019	0.027	0.079	1.669
bv70	4	2	2	3	3	0.022	0.03	0.061	2.221
qugan71	2	5	5	5	4	0.346	0.725	11.392	5.53
qugan71	3	10	10	10	11.9	0.375	0.769	8.146	9.694
qugan71	4	15	15	15	14.5	0.403	0.794	9.485	13.137
wstate76	2	2	2	2	1	0.105	0.156	0.666	2.114
wstate76	3	4	4	4	3	0.115	0.165	0.687	3.133
wstate76	4	6	6	6	3	0.117	0.167	0.703	4.777
ghz78	2	1	1	1	1	0.039	0.057	0.103	2.138
ghz78	3	2	2	2	2	0.047	0.069	0.126	3.785
ghz78	4	3	3	3	3	0.047	0.066	0.114	4.898
swaptest83	2	1	1	1	1	0.258	0.444	26.224	11.01
swaptest83	3	2	2	2	2	0.277	0.466	20.204	19.046
swaptest83	4	6	6	6	5	0.294	0.482	17.082	29.794
ising98	2	1	1	1	1.5	0.147	0.241	1.354	0.236
ising98	3	2	2	2	2.8	0.156	0.25	1.369	0.383
ising98	4	3	3	3	4	0.164	0.258	1.378	0.466