Quantum Science and Technology

PAPER • OPEN ACCESS

Distributing circuits over heterogeneous, modular quantum computing network architectures

Quantum Science and Technology

CrossMark

OPEN ACCESS

RECEIVED
15 January 2024

REVISED
19 June 2024

ACCEPTED FOR PUBLICATION
24 July 2024

PUBLISHED
6 August 2024

PAPER

Distributing circuits over heterogeneous, modular quantum computing network architectures

Pablo Andres-Martinez[1,8,*], Tim Forrer[2,8], Daniel Mills[1,8,*], Jun-Yi Wu[3,4], Luciana Henaut[1], Kentaro Yamamoto[1], Mio Murao[2,6] and Ross Duncan[1,5,7]

[1] Quantinuum, Terrington House, 13-15 Hills Road, Cambridge CB2 1NL, United Kingdom
[2] Department of Physics, Graduate School of Science, The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113-0033, Japan
[3] Physics Division, National Center for Theoretical Sciences, Taipei 10617, Taiwan, ROC
[4] Department of Physics and Center for Advanced Quantum Computing, Tamkang University, 151 Yingzhuan Rd., New Taipei City 25137, Taiwan, ROC
[5] Department of Physics and Astronomy, University College London, Gower Street, London WC1E 6BT, United Kingdom
[6] Trans-scale Quantum Science Institute, The University of Tokyo, Bunkyo-ku, Tokyo 113-0033, Japan
[7] Department of Computer and Information Sciences, University of Strathclyde, 26 Richmond Street, Glasgow G1 1XH, United Kingdom
[8] These authors contributed equally to this work.
[*] Authors to whom any correspondence should be addressed.

E-mail: pablo.andresmartinez@quantinuum.com and daniel.mills@quantinuum.com

Keywords: distributed quantum computing, quantum circuit compilation, quantum software, quantum internet, scalable quantum computing, quantum communication

Abstract

We consider a heterogeneous network of quantum computing modules, sparsely connected via Bell states. Operations across these connections constitute a computational bottleneck and they are likely to add more noise to the computation than operations performed within a module. We introduce several techniques for transforming a given quantum circuit into one implementable on such a network, minimising the number of Bell states required to do so. We extend previous works on circuit distribution to the case of heterogeneous networks. On the one hand, we extend the hypergraph approach of Andres-Martinez and Heunen (2019 *Phys. Rev.* A **100** 032308) to arbitrary network topologies, and we propose the use of Steiner trees to detect and reuse common connections, further reducing the cost of entanglement sharing within the network. On the other hand, we extend the embedding techniques of Wu *et al* (2023 *Quantum* **7** 1196) to networks with more than two modules. We show that, with careful manipulation of trade-offs, these two new approaches can be combined into a single automated framework. Our proposal is implemented and benchmarked; the results confirm that our contributions make noticeable improvements upon the aforementioned works and complement their weaknesses.

1. Introduction

Quantum computing providers are racing to scale up their systems, targeting qubit numbers and gate fidelities that would allow for demonstrations of quantum advantage on practical applications. As architectures scale up, their basic components grow farther apart, increasing the cost of communicating between them. Moreover, operations between distant components require more intermediary elements to be involved, thus making it challenging to maintain high fidelity as errors accumulate.

Distributed quantum computing [1] provides an alternative: once a quantum computing module pushing the limits of current technology is engineered, it may be more practical to produce copies of it and connect them together to create larger devices. Indeed, researchers in academia and industry have proposed both short and long-term distributed quantum computing projects.

Short-term. An emerging field of research studies the use of classical postprocessing to 'knit together' multiple quantum circuits [2, 3], with the goal of simulating circuits that are too large to be run in current quantum computers. The quantum circuit is 'cut' at different points, creating smaller subcircuits that can be run on current quantum computers.

The classical postprocessing may be done after the quantum computation has finished. Cutting circuits has the additional advantage of facilitating greater parallelism, and of preventing error propagation between portions of the circuit. However, the classical overhead scales exponentially with the number of cuts, so the technique is only applicable to circuits that can be split using few cuts. While practical applications can be found in the field of quantum chemistry, where knowledge of the symmetries of the system being modelled can be exploited to generate circuits in which two groups of qubits barely interact with each other [4], circuit cutting is not applicable to larger and more entangled circuit which might demonstrate quantum computation supremacy [5]. As such, circuit cutting does not provide a long term solution to the problem of distributing quantum circuits.

Long-term. There is a history of academics proposing modular quantum computers [6–8], and related technologies appear in the field of quantum internet [9, 10]. In such modular architectures, it is expected that different modules will interact with each other throughout the computation via entanglement sharing. This approach has a resource overhead scaling only linearly in the number of cuts to the circuit, making it a longer term solution.

Currently, the challenge of high-rate generation of entangled states between different modules is too great for the technology to become widely applicable. However we will eventually reach an inflexion point where the communication cost within a large enough module will be comparable to that of entanglement generation between separate modules [6]. The current road-map of IBM promises the release of the first prototype of a modular quantum computer (Flamingo) with quantum communication between modules by the end of 2024, and Quantinuum considers developing a modular quantum computer in its long term road-map[9].

Even in the long-term situation discussed above, communication of quantum information between modules will be a significant bottleneck of the computation. It is thus essential to develop circuit optimisation methods that minimise the amount of quantum communication required to distribute a circuit. This is the purpose of the present manuscript. The methods we discuss here are also applicable to the short-term applications of classically simulated circuit knitting [3, 4], since reducing the amount of communication between modules is equivalent to reducing the number of cuts and, hence, the exponential classical overhead.

In this work, we assume that all quantum communication is carried out by the consumption of Bell pairs shared between modules. Previous works on distributed quantum computing focus either on the minimisation of the circuit's depth [11] or attempt to minimise the number of Bell pairs consumed [12–15]. This manuscript falls into the second category, since we identify Bell pair generation and sharing as the main bottleneck of the computation. Among the works in this category, [12, 13, 15] assume a fully connected network of modules. Sundaram *et al* [14] studies heterogeneous networks, where not every pair of modules are connected to each other directly, and where each module may have different qubit register capacities.

The task of circuit distribution has some similarities with the qubit routing problem [16, 17], in that both are concerned with gate scheduling and assignment of qubits to hardware registers. The main distinction between them lies in that the goal of routing is to implement a circuit on a *single* module (with limited connectivity), whereas the distribution problem deals with the interaction between multiple modules. Thus, the distribution problem can be studied at a higher level of abstraction, where we may assume operations within a module to be comparatively free. This leads to distribution being naturally related to the mathematical problem of graph partitioning, whereas qubit routing is an instance of token swapping [17]. Moreover, this distinction leads to a desirable separation of concerns: once a circuit is distributed, the next step on a compilation stack is to solve the routing problem for each of its subcircuits, optimising its implementation for the specific hardware constraints of the module it is assigned to.

In section 2 we give a precise definition of the circuit distribution problem. We review the relevant literature in section 3, focusing on approaches that minimise the number of Bell pairs consumed [12–15]. The main contributions of our work are presented in section 4:

- In section 4.1 we develop the use of Steiner trees to reuse common connections in the network, reducing the entanglement cost.
- In section 4.2 we extend the results of [15] from bipartite to multipartite networks and develop a method to integrate the embedding approach of [15] with the entanglement distribution via Steiner trees from section 4.1.

[9] The road-map of these companies is publicly available at their respective web-pages at the time of writing.

- In section 4.3 we extend the results of [12] for gate and qubit distribution from homogeneous to heterogeneous networks.

In section 5 we explore different routes to combining these new methods in ways that make use of the advantages of both; presenting benchmarking results there. The proposed approach has been implemented as an open source project, pytket_dqc.

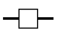
2. The distributing quantum circuits (DQC) problem

In this work we focus on the problem of DQC over general networks of quantum computers, minimising the number of entangled resources required to do so. A network is comprised of a collection of quantum computers that we refer to as *modules*. These modules are connected via quantum communication channels, with Local Operations and Classical Communication (LOCC) also available. A quantum communication channel may be used to generate maximally entangled bipartite states between two modules. We refer to a such shared state as an *ebit*, and take it to be a Bell state:

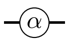
$$\frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right). \tag{1}$$

Formally, the network is specified by an undirected graph $G = (V, E)$. Each vertex $\mathtt{A} \in V$ corresponds to a module and each edge $(\mathtt{A}, \mathtt{B}) \in E$ indicates that ebits may be prepared and shared between modules $\mathtt{A}$ and $\mathtt{B}$. Each module $\mathtt{A} \in V$ is capable of managing $\omega(\mathtt{A})$ qubits dedicated to computation—its *computation register*—and $\epsilon(\mathtt{A})$ qubits dedicated to communication—its *link qubit register*. Thus, $\epsilon(\mathtt{A})$ determines the maximum number of connections that can be simultaneously maintained by module $\mathtt{A}$. These link qubits are disentangled from the rest of the computation at the end of the communication protocol described in section 3.1. Consequently, we may reuse the space in the link qubit register throughout the computation in order to establish new communications channels at different points in time[10].

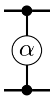
We assume each of the modules is capable of universal quantum computation and we consider no restrictions on the module's internal qubit connectivity. The particular universal gate set, and the actual internal connectivity of the modules, may be accounted for by a later stage of circuit compilation [22] acting individually on the local subcircuit assigned to each module. Our objective is to minimise the total number of ebits consumed, whose preparation and sharing is expected to be the bottleneck of any distributed quantum computation. Throughout the paper we consider that LOCC are comparatively free and assume that circuits are constructed using the gateset $\{H, R_Z, CR_Z\}$, which we depict using the following shorthand:



$$= \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix},$$



$$= \begin{pmatrix}1 & 0\\ 0 & e^{i\alpha}\end{pmatrix},$$



$$= \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & e^{i\alpha}\end{pmatrix}.$$

Evidently, CZ and Z gates are contained within this gateset, since they are particular instances of CR_Z gates and R_Z gates whose phase is π.

The DQC problem in the case of such a network can be divided into two subproblems:

- **Qubit allocation**. We must determine the allocation of each qubit of the circuit to a module $\mathtt{A} \in V$ in the network. The number of qubits allocated to each module $\mathtt{A} \in V$ must not exceed the module's computation register size $\omega(\mathtt{A})$.

Once the circuit's qubits are allocated, some two-qubit gates may act on qubits allocated to different modules; we call these *non-local* gates. We are interested in qubit allocations that reduce the number of ebits required to implement these non-local gates.

[10] In this work we abstract away details about inter-module entanglement generation and management. We refer the reader to [18–21] for details and reviews of methods of constructing a complete 'quantum internet protocol stack'.

- **Non-local gate distribution**. Once a qubit allocation is chosen, we must find a way to implement the non-local gates that arise. This may be done by consuming ebits and using LOCC. As in previous works [12–15], here we focus on the use of simultaneous gate teleportation—which we refer to as the EJPP protocol—as described in section 3.1.

The maximum number of EJPP protocols sharing data with module $\mathtt{A} \in V$ at a given point in time should not exceed the module's link qubit register size of $\epsilon(\mathtt{A})$ qubits. Approaches to enforce this capacity constraint have been explored in previous works [14, 15]. In this work, we present techniques that assume $\epsilon(\mathtt{A})$ is not bounded and, hence, may allow an arbitrary number of simultaneous quantum channels. Such an assumption is unreasonable in practice and in appendix C we discuss a simple algorithm that, given an already distributed circuit, modifies it so that the bound to $\epsilon(\mathtt{A})$ for each module $\mathtt{A} \in V$ is satisfied.

The solution to a DQC problem can be concisely characterised as follows:

Definition 1 (distribution). Let Q be a set of qubits and V a set of modules. A *distribution* of a quantum circuit on $|Q|$ qubits over a network of $|V|$ modules is characterised by:

- a *qubit allocation* map $\phi : Q \to V$ such that $|\phi^{-1}(\mathtt{A})| \leqslant \omega(\mathtt{A})$ for all $\mathtt{A} \in V$ and
- an equivalent circuit on $|Q| + \sum_{\mathtt{A}\in V} \epsilon(\mathtt{A})$ qubits that satisfies the qubit allocation map ϕ for the qubits in Q and where all multi-qubit gates between modules are realised via the generation and consumption of ebits.

We are interested in distributions that consume the fewest number of ebits.

3. Background

3.1. EJPP protocol and distributable packets

A non-local CR_Z gate can be implemented by consuming a single ebit. The distribution protocol we use originates from [23] and we refer to it as the EJPP protocol, using the initials of the latter paper's authors. Figure 1 provides an example of such a protocol. During the EJPP protocol, a qubit $\hat{q}$ is shared with a remote module $\mathtt{B}$, entangling it with an ancilla qubit within its link qubit register $\epsilon(\mathtt{B})$. The starting process of the EJPP protocol—boxed in grey in figure 1—generates and consumes an ebit to produce a link qubit that is entangled with $\hat{q}$. The *ending process* only uses LOCC and disentangles the link qubit. Crucially, multiple non-local gates can be implemented using a single EJPP protocol and, hence, consuming a single ebit.

Definition 2 (distributable packet). A *distributable packet* rooted on qubit $\hat{q}$ is a subset of a circuit's non-local CR_Z gates that act on $\hat{q}$ and can all be implemented simultaneously using a single EJPP protocol[11].

Lemma 3. *Given a circuit comprised of gates in $\{H, R_Z, CR_Z\}$ for which a qubit allocation map ϕ has been provided, let P be a subset of CR_Z gates in the circuit. If the following three conditions hold, then P is a distributable packet rooted on qubit $\hat{q}$.*

(a) *Each gate $g \in P$ acts on $\hat{q}$.*
(b) *For each $g \in P$ let q_g be the qubit g acts on such that $q_g \neq \hat{q}$; there is a module $B \in V$ such that $\phi(\hat{q}) \neq B$ and $\phi(q_g) = B$ for all $g \in P$.*
(c) *For every pair of gates $g, g' \in P$, there is no H gate in the circuit acting on $\hat{q}$ between g and g'.*

Proof. Conditions (a) and (b) ensure that sharing the state of qubit $\hat{q}$ with module $\mathtt{B}$ is sufficient to implement all of the gates in P within $\mathtt{B}$. A starting process creates a link qubit in module $\mathtt{B}$ that is entangled with $\hat{q}$. Then, each gate $g \in P$ is replaced by the same gate acting on q_g and said link qubit. The ending process is applied after the last gate in P, measuring out the link and correcting as necessary to guarantee determinism. Condition (c) along with the circuit's gateset implies that all gates between g and g' are R_Z or CR_Z gates, so they commute past g and g'. Closer inspection of the circuit for a starting process and ending process (see figure 1) reveals these also commute with R_Z and CR_Z gates. Therefore, their presence within the EJPP protocol does not affect its operation and we can apply all gates between g and g' unchanged, on their original qubits. Then, it only remains to check that the equivalence of the circuits in figure 1 generalises to the case of any number of consecutive CR_Z gates. This follows from combining all CR_Z gates into a single gate controlled on $\hat{q}$, which is then implemented using the standard EJPP protocol [23]. □

[11] This definition captures the essence of definition 16 from [15]. Here, we refer to the elements of a distributable packet P as gates $g \in P$, whereas in [15] the elements of P are pairs $(\hat{q}, t_g)$, where $\hat{q}$ is the qubit that P is rooted on and t_g is the layer in the circuit that gate g appears at. In our work, the data $(\hat{q}, t_g)$ is implicitly captured in g; that is, g identifies a particular gate position in the circuit.

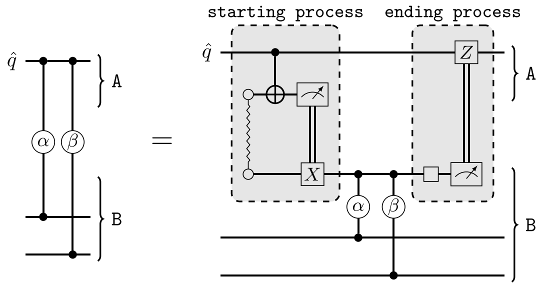


Figure 1. Distribution of two non-local CR_Z gates via an EJPP protocol. The gates act on two different modules A and B, the former containing a qubit $\hat{q}$ that both CR_Z gates are applied to. The starting process generates and consumes an ebit, depicted by a wavy line. Other than the ebit, all operations on the distributed circuit are LOCC.

Remark 4. While conditions (a) and (b) are necessary for all gates in P to be implementable using a single EJPP protocol, (c) can be replaced with a more general condition using a technique known as *embedding* [15].

(c*) For every pair of gates $g, g' \in P$, all gates in the circuit acting on $\hat{q}$ between g and g' are *embeddable*.

This leads to larger distributable packets. We will discuss what the term *embeddable* refers to in section 3.3. For now, it suffices to know that condition (c) implies (c*).

3.2. DQC via hypergraph partitioning

The qubit allocation subproblem introduced in section 2 is reminiscent of a graph partitioning problem. Indeed, we may define the connectivity graph of a circuit as follows: each qubit in the circuit corresponds to a vertex and each CR_Z gate creates an edge between the vertices of the pair of qubits it acts on. It is straightforward to see that partitioning such a graph into k blocks corresponds to allocating each of the qubits to one of k different modules, and cut edges correspond to non-local gates. Thus, the standard graph partitioning problem—whose goal is to minimise the number of edges cut—would produce qubit allocations that minimise the number of non-local gates.

However, such an approach would not consider the fact that a single EJPP protocol is capable of implementing multiple non-local gates consuming a single ebit. If our objective is to minimise the number of ebits consumed, a different partition that creates more non-local gates may be advantageous. An example of such a situation is shown in figure 2. Crucially, the optimal qubit allocation of figure 2(a) places qubits q_0 and q_1 in module A and qubits q_2 and q_3 in module B, but such an assignment differs from the optimal partition of the circuit's connectivity graph figure 2(b), which would instead place q_0 and q_2 in module A and qubits q_1 and q_3 in module B. The former allocation yields four non-local gates whereas the latter yields only three, however, the former can be distributed using two ebits (figure 2(e)), while the latter requires three.

In [12] it was shown that qubit allocation and non-local gate distribution could both be solved simultaneously via a reduction to hypergraph partitioning. Formally, the only difference between a hypergraph and a graph is that its edges need not be pairs, but subsets of vertices known as 'hyperedges'. The intuition behind why hypergraphs are better suited to describe the DQC problem is that, when multiple gates belong to the same distributable packet (definition 2), we may represent them as a single hyperedge. Then, if any number of these gates become non-local due to a qubit allocation, the corresponding effect is that a single hyperedge will be cut by the partition, thus precisely capturing the number of EJPP protocols required to implement the distributable packet. The algorithm that builds such a hypergraph from a given circuit is described in [12] and figure 2(c) shows the outcome of the process on a simple circuit. In [12] the authors proved the following theorem.

Theorem 5 ([12]). *Given a circuit in the $\{H, R_Z, CR_Z\}$ gateset, each of its possible distributed implementations corresponds to a unique partition of its hypergraph. Assuming a fully connected network of modules, the number of ebits required to implement such a distribution coincides with the cost of the partition, calculated using the connectivity metric*[12].

This implies that we may reduce the problem of distributing a quantum circuit to the problem of hypergraph partitioning as follows.

[12] For a given hypergraph, where H is its set of hyperedges, the connectivity metric [24] of a given partition is calculated as $\sum_{h\in H}\lambda(h) - 1$ where $\lambda(h)$ corresponds to the number of different partition blocks the hyperedge h has vertices on.

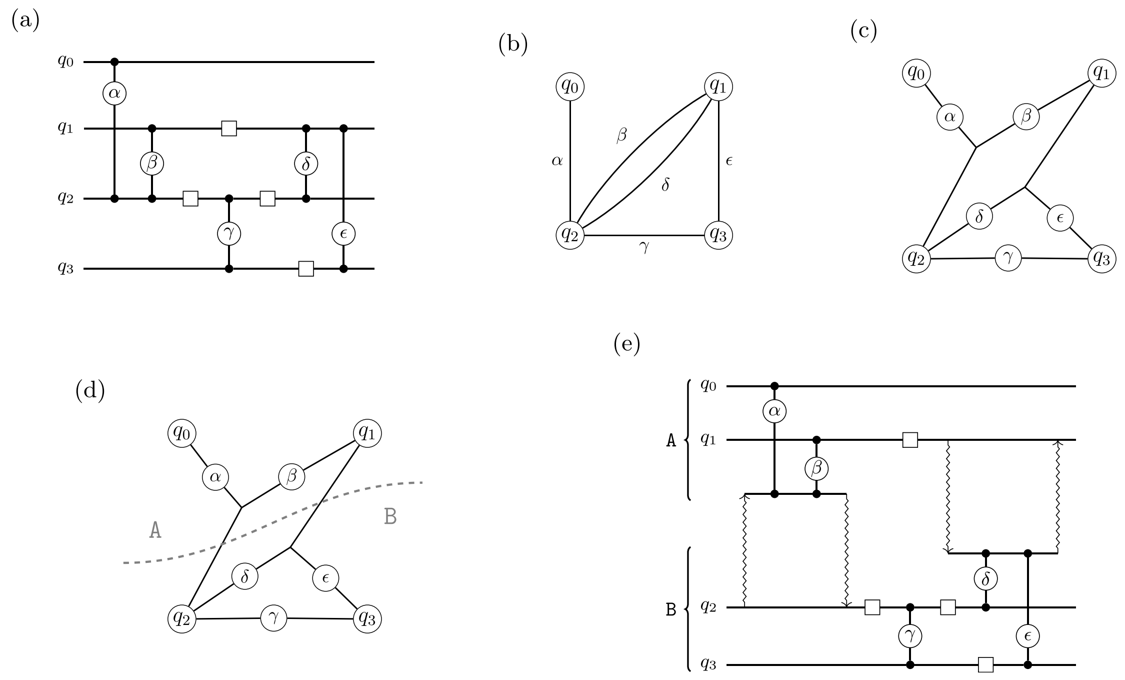


Figure 2. Example of correspondence between circuits, graphs and hypergraphs. (a) An input circuit, (b) its connectivity graph, (c) its hypergraph, as defined in [12], (d) an optimal partition of the hypergraph, only two hyperedges are cut, (e) the distributed circuit that arises from the hypergraph partition, the number of EJPP protocols matches the connectivity metric of the hypergraph partition: two cuts (theorem 5). Starting processes and ending processes are depicted as wavy arrows.

1. Build the hypergraph of the circuit as described in [12].
2. Use a state-of-art hypergraph partitioner to obtain an efficient partition.
3. Translate the partition into a distribution of the circuit.

Notice that in the hypergraph of figure 2(c) there are more vertices than qubits in the circuit. In fact, there is a vertex per qubit and a vertex per CR_Z gate; we call them *qubit-vertices* and *gate-vertices* respectively. When a partition assigns a qubit-vertex to block A it indicates that such a qubit is to be allocated to module A; similarly, when a gate-vertex is assigned to block A it indicates that the corresponding CR_Z gate ought to be implemented as a local CR_Z gate within module A, with the aid of an EJPP protocol if the hyperedge is cut. Figures 2(d) and (e) exemplify how a partition of the hypergraph gives rise to a distribution of the original circuit.

This approach, as presented in [12] has some shortcomings, identified below.

- Hypergraph partitioners often assume that all blocks of the partition should be filled with approximately the same number of vertices each. In our task, however, each module A may have a different capacity of workspace qubits $\omega(\mathtt{A})$. To account for this constraint, we may use hypergraph partitioners such as KaHyPar [24] which allow us to indicate the maximum capacity of each block; more details on section 4.3.
- In [12], only fully connected networks were considered (i.e. complete graphs), whereas in this work we consider heterogeneous networks (section 2). Creation and sharing of ebits between adjacent modules is directly supported by the network's hardware. Non-adjacent modules may still share an ebit, but producing it will require some entanglement distribution, consuming multiple hardware-supported ebits in the process. The framework summarised in this section is not capable of making such a distinction. Some techniques used to extend hypergraph partitioning to account for heterogeneous networks are discussed in section 4.3.
- In section 3.3 we discuss some advanced techniques to further reduce the ebit count of implementing non-local gates by merging multiple distributable packets. These techniques are beyond what can be captured in terms of hypergraph partitioning. Thus, in section 4 we use hypergraph partitioning to provide an initial solution to the DQC problem, whose non-local gate distribution is later refined using the techniques from sections 3.3, 3.4.1 and appendix A.

Approaches that solve the two subproblems of DQC separately—qubit allocation and non-local gate distribution—have been proposed in the literature. In [13], the authors solve the qubit allocation subproblem by partitioning a weighted graph describing the connectivity of the circuit, where the calculation

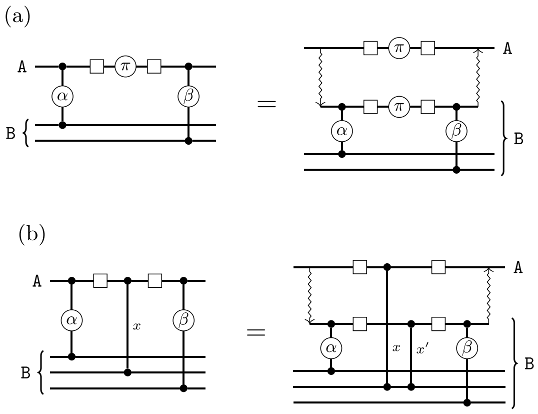


Figure 3. Examples of embedding. (a) Embedding $H\cdot Z\cdot H$, (b) embedding $H\cdot CZ\cdot H$, the correction CZ gate x' is local. In both examples, the CR_Z gates with phase α and β are implemented by the same EJPP protocol.

of the weights attempts to take into account cases where multiple non-local gates may be implemented using a single ebit. Such an approach has the same shortcomings listed above, with the additional drawback that the weights only provide an estimate for the ebit cost (rather than the exact value as in the case of hypergraph partitioning) and the advantage that graph partitioners are simpler and, hence, can be expected to perform better than hypergraph partitioners. A follow up paper by the same authors [14] solves the qubit allocation subproblem using a Tabu search algorithm. The latter work supports heterogeneous networks, solving one of the three shortcomings discussed above, but fails to take advantage of the optimisation opportunities we discuss in section 4.1. Both of these works solve non-local gate distribution on a second step, which we review in section 3.4.

3.3. Embedding

Lemma 3 provides sufficient conditions for a group of non-local CR_Z gates to belong to the same distributable packet. Remark 4 hinted at a more general condition involving the notion of *embedding* proposed in [15]. Figure 3 provides a couple of examples where the CR_Z gates of phase α and β belong to the same distributable packet even though there are H gates between them, violating condition (c) from lemma 3.

Definition 6 (Embedding unit). Consider an EJPP protocol with starting process $\mathcal{S}_{\hat{q},\mathtt{B}}$ sharing qubit $\hat{q}$ with module $\mathtt{B}$ and ending process $\mathcal{E}_{\hat{q},\mathtt{B}}$ (see figure 1). An embedding unit is a subcircuit C satisfying the following identity:

$$C = \mathcal{E}_{\hat{q},\mathtt{B}} \left(\bigotimes_{\mathtt{M}\in V} L_{\mathtt{M}}\right) C \left(\bigotimes_{\mathtt{M}\in V} K_{\mathtt{M}}\right) \mathcal{S}_{\hat{q},\mathtt{B}} \tag{2}$$

where V is the set of modules in the network and for each module $\mathtt{M}\in V$, $L_{\mathtt{M}}$ and $K_{\mathtt{M}}$ are local gates within $\mathtt{M}$. We refer to the gates $L_{\mathtt{M}}$ and $K_{\mathtt{M}}$ as the *correction gates* of the embedding.

In essence, an embedding unit is a subcircuit appearing between gates of a distributable packet P such that, if P is distributed, we only require local correction gates to maintain circuit equivalence. Importantly, notice that we do not require C to be local—it has not yet been distributed. Indeed, the embedded CZ gate labelled x' in figure 3(b) is non-local.

It follows from the above definition that any gate that commutes with a starting process $\mathcal{S}_{\hat{q},i}$ forms an embedding unit, which is the reason why condition (c) from lemma 3 implies (c*) from remark 4. More interesting embedding units containing H gates are captured by the following lemma.

Lemma 7. *Let C be a circuit built from $\{H, R_Z, CR_Z\}$ containing a qubit $\hat{q}$, let B be a module and let ϕ be a qubit allocation such that $\phi(\hat{q}) \neq B$. If each of the following conditions holds, then C is an embedding unit of an EJPP protocol sharing $\hat{q}$ with module B.*

(a) *The first gate and last gates in C are H gates acting on $\hat{q}$.*
(b) *All CR_Z gates within C that act on $\hat{q}$ have their other qubit allocated to module B.*
(c) *All CR_Z gates within C that act on $\hat{q}$ have π phase—i.e. they are CZ gates.*

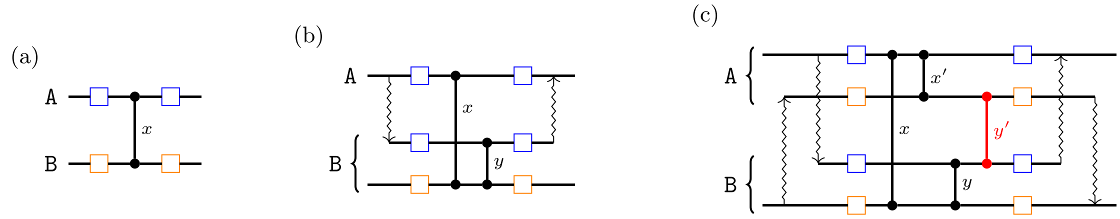


Figure 4. Embedding conflict. (a) A simple circuit with two embedding units: one containing the *CZ* gate and the blue *H* gates; the other containing the *CZ* gate and the orange *H* gates. (b) Embedding only one of the embedding units causes no issues: the correction gate y is local. However, notice that the other embedding unit (the one containing the orange *H* gates) contains y as well. (c) Since y does not satisfy lemma 7 (both of its qubits are in the same module), embedding it would create a non-local correction gate y', defeating the purpose of embedding.

(d) *All R_Z gates within C that act on $\hat{q}$ may be squashed together so that only R_Z gates with π phase remain—i.e. Pauli Z gates.*
(e) *There are no more than two H gates acting on $\hat{q}$ in C.*

Proof. Immediate from corollary 30 of [15]. Alternatively, this is a straightforward generalisation of the two embedding units shown in figure 3. □

Lemma 7 provides a sufficient condition for a subcircuit to be an embedding unit. In [15] a more detailed analysis shows that condition (e) can be relaxed, but the formalisation of this more general condition is too intricate to be presented in this summary. These more general conditions can be checked on a circuit using algorithm 35 from [15], which we implemented in the software `pytket_dqc` we present in this work.

Equipped with the notion of embedding units and the algorithm from [15] to identify them, we can now build larger distributable packets. Whenever a gate commutes with the packet's starting process, embedding it requires no correction gates. Whenever we encounter an embedding unit, we apply the embedding rules from corollary 30 of [15] to introduce the required local correction gates; more details are provided in appendix B.

Condition (b) from lemma 7 has a rather subtle implication: if two embedding units on different qubits contain the same *CZ* gate, only one of the two is embeddable, see figure 4. We then say that such a pair of embedding units have an *embedding conflict*; similarly, two distributable packets that contain embedding units in conflict are also said to have an embedding conflict. Resolving an embedding conflict requires choosing which of the two distributable packets should be distributed and splitting the other one into two separate packets so that embedding the conflicting *CZ* gate a second time is no longer necessary. An algorithm for non-local gate distribution that takes advantage of embedding and resolves embedding conflicts was proposed in [15]; we briefly review the algorithm in section 3.4.1.

For the sake of brevity, we have not discussed how to deal with situations where a certain embedding unit must be embedded within more than one distributed packet. Thanks to corollary 14 from [15], we know that these situations will never cause new conflicts. A more subtle situation arises when two distributable packets P and P' happen to be intertwined in the sense that some gate $g \in P$ needs to be embedded within P' while, at the same time, some other gate $g' \in P'$ needs to be embedded within P. We describe how we deal with such a situation in appendix B.

3.4. Non-local gate distribution via vertex cover

In this section we review the literature on the subproblem of non-local gate distribution. We focus on approaches that reduce it to finding the minimum vertex cover of a graph. We begin from a version of the problem with the following simplifications:

- we assume that the network of modules is fully connected,
- we ignore the optimisation opportunities embedding provides and
- we impose that a non-local gate must be implemented in either of the two modules it acts on—this prevents beneficial distributions such as the one in figure 5 from being considered.

This simplified problem is presented in [13] under the name `MS-HC`; we summarise their solution in this section. One of the contributions of the present work is the extension of their approach to the general problem where these three constraints are lifted. In particular, section 4.2 and appendix A.3 allows us to consider heterogeneous networks of modules, we use the approach of [15] (summarised in section 3.4.1) to exploit embedding and employ the method in appendix A.1 to lift the last of the constraints.

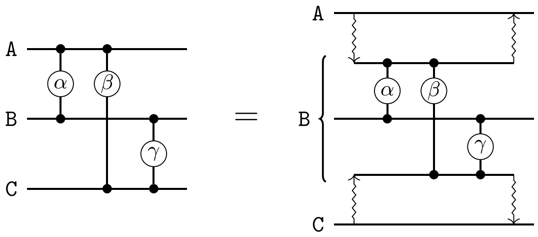


Figure 5. Distribution with detached gate. In the distributed circuit, the CR_Z gate with phase β is implemented within module B, but originally had none of its qubits allocated to it—we refer to it as a detached gate.

Once a qubit allocation has been chosen, all that remains to do is identify distributable packets and, for each non-local CR_Z gate, decide which of the packets it belongs to should be used to distribute it. In the absence of embedding, the first task—which we refer to as *gate packing*—is trivial: scan the circuit qubit by qubit, from beginning to end, and find sequences of gates satisfying lemma 3. We shall only consider the collection of largest distributable packets, i.e. those that are not a subset of any other distributable packet; as a consequence, each non-local CR_Z belongs to exactly two distributable packets—one per qubit.

The second task corresponds to finding the minimum vertex cover of a graph whose vertices represent the distributable packets and where an edge between two of these corresponds to the existence of at least one non-local CR_Z gate contained in both packets. A vertex cover of a graph is a subset of its vertices such that each edge is incident to at least one vertex in the subset; thus, a vertex cover of the previous graph selects which distributable packets ought to be realised so that all non-local CR_Z gates are distributed. A minimum vertex cover of the graph would select the fewest number of distributable packets and, hence, yield the optimal distribution under the given constraints. In the appendix of [13] its authors show that the previous graph is guaranteed to be bipartite, which implies the minimum vertex cover can be found efficiently.

Sundaram *et al* [13] then considered a more general problem where non-local gates may be implemented in a detached manner, i.e. so that distributions such as the one from figure 5 may be explored. This more general problem is not known to be reducible to a vertex cover problem on a bipartite graph. Nevertheless, the authors of [13] proposed an efficient algorithm that is guaranteed to provide a distribution only a logarithmic factor away from the best distribution achievable under the given constraints. However, said algorithm is still solving a simplified problem since it is omitting the following constraints and optimisation opportunities.

Network topology: In section 2 we let networks be described by arbitrary (connected) graphs. Thus, the approach should take into account the distance between modules when computing the cost of distributing non-local gates.

Bounded link qubit register: Each module A may have a bound to the size of its link qubit register $\epsilon(\mathtt{A})$ (see section 2). The resulting distribution should refrain from exceeding it.

Embedding: The embedding technique described in section 3.3 lets us create larger distributable packets. Thus, an algorithm using embedding is likely to cover all non-local gates using fewer packets and, hence, find a distribution that uses fewer ebits.

In [14] the authors propose an algorithm that is aware of the network topology and the bound to the size of the link qubit register. Rather than a minimum vertex cover problem, they consider the dual problem of maximising the number of non-local gates covered using a fixed number of ebits while satisfying a set of linear constraints. The linear constraints are used to capture the network topology and the bound to the link qubit register. The central element of the optimisation procedure is carried out by an integer linear programming subroutine. However, this approach does not take advantage of embedding.

In [15] an algorithm that exploits embedding is proposed. The algorithm is based on finding minimum vertex covers and it makes use of graph colouring to identify solutions that satisfy the bound to the link qubit register. However, the algorithm is targeted to networks containing only two modules and, consequently, has a trivial network topology. The approach to DQC we propose in the present paper takes multiple insights from this latter work, orchestrating them in a more general framework. Thus, we dedicate the following section to introduce the ideas from [15] that are relevant to us.

3.4.1. Embedding-aware approach

The approach to non-local gate distribution using minimum vertex cover can be extended to account for embedding. The means to do so were described in [15], where detailed algorithms were provided. Once again, the first step is to identify the largest distributable packets that can be realised without the use of embedding. Then, for each distributable packet P, we check whether the gates that come immediately after P form an embedding unit. If so, this would allow P to be merged with the packet appearing immediately after the embedding unit, creating a larger distributable packet. The algorithm identifies the largest distributable packets that can be achieved by such merging, and records the embeddings that are required to do so. This task simply requires us to carry out a scan over the circuit and, hence, it scales linearly with the dimensions of the circuit.

The resulting distributable packets are then arranged in a graph G as in the case of the standard vertex cover approach: its vertices correspond to each of the packets and its edges correspond to common non-local CR_Z gates between them. It may seem that it only remains to find a minimum vertex cover of G, but this would not account for embedding conflicts (see figure 4). Instead, we need to define an additional graph K whose vertices correspond to the embeddings that were used when merging distributable packets, where an edge between two such embeddings appears if and only if the embeddings are in conflict. With these two graphs G and K at hand, a sketch of the algorithm is presented below.

1. Find a minimum vertex cover $\mathcal{C}_G$ of G.
2. Find the subset κ of embeddings required to implement all of the distributable packets in $\mathcal{C}_G$.
3. Extract the subgraph K_κ of K whose vertex set is κ and whose edges are those from K that connect vertices in κ.
4. Obtain a minimum vertex cover $\mathcal{C}_K$ of K_κ: this is the smallest set of embeddings that we must give up in order to resolve all embedding conflicts incurred by $\mathcal{C}_G$.
5. For each element in $\mathcal{C}_K$—an embedding—, identify which distributable packet $P \in \mathcal{C}_G$ used it (there is exactly one) and update $\mathcal{C}_G$ replacing P with two distributable packets: one containing all of the gates in P that come before the embedding unit responsible for the embedding conflict and another with the gates in P that come afterwards.

The resulting set of distributable packets $\mathcal{C}_G$ is no longer a *minimum* vertex cover of G, but it is a vertex cover with no embedding conflicts. Thus, it can be used to generate a valid distribution. This approach is not guaranteed to return the overall optimal solution, but it does resolve the embedding conflicts of a given vertex cover of G in an optimal way. In an attempt to find better overall solutions, we may choose to repeat the routine above for multiple distinct vertex covers of G and pick the best among them [15].

The algorithms presented in [15] for the tasks just described were designed for networks with exactly two modules. Generalising these to networks of multiple modules is immediate: it is sufficient that our conditions for identifying distributable packets (lemma 3) and embedding units (lemma 7) required that all of their CR_Z gates acted on the same two modules. This guarantees that both G and K are bipartite graphs, so we may find a minimum vertex cover for them efficiently. The fact that these graphs are bipartite graphs is not trivial, but it follows from the same argument Sundaram *et al* [13] used for their bipartite graph for the `MS-HC` problem.

Wu *et al* [15] propose a way to take into account the bound to the link qubit register size. They identify offending distributable packets via graph colouring and split them so that the number of link qubits simultaneously in use is reduced, at the cost of increasing the total number of ebits consumed. Such an approach is beyond the scope of the present paper and we omit further details for the sake of brevity. For an alternative approach, see appendix C.

3.5. Intermediate representation of distribution

Throughout this section we have discussed multiple approaches aimed at optimising different aspects of the DQC problem. We have considered multiple abstractions—e.g. distributable packets, embedding units, embedding conflicts, hypergraphs, etc—each tailored to be as natural as possible to the approach at hand. Our goal in this paper is to propose an approach that can take advantage of the insights of each of these optimisation methods. To do so, we require an intermediate representation where the outcome of each of these optimisation methods can be represented. Such an intermediate representation could simply be a partially distributed circuit; however, a more abstract representation would be preferable to minimise the overhead of dealing with superfluous low level details—such as the correction gates required for an embedding unit, the exact placement of a starting process within a circuit, the reuse of link qubits, etc—that could easily be deferred to the final step of the workflow. Fortunately, all of what has been reviewed in this section can be captured within the framework of hypergraphs discussed in section 3.2, which makes it a natural choice for our intermediate representation of a distribution.

Definition 8 (IR of distributions). A `Distribution` contains the following information.

- A hypergraph of $|Q|+|\mathcal{G}|$ vertices, where Q is the set of qubits in the original circuit and $\mathcal{G}$ is its collection of CR_Z gates. We refer to these as qubit-vertices and gate-vertices respectively, as established in section 3.2.
- An allocation map $\phi : Q \cup \mathcal{G} \to V$, where V is the set of modules in the network.

Additionally, we include the original circuit and the network of modules, which remain unchanged throughout the workflow.

The purpose of including the original circuit and the network of modules within the `Distribution` is to be able to assess the ebit cost (see section 4.2). Furthermore, the information contained in `Distribution` is all we require to generate the corresponding distributed circuit; we explain how to do so in appendix B. Notice that the allocation map ϕ determines a partition of the hypergraph. Below, we briefly discuss how the different abstractions considered in this section can be captured within a `Distribution`. Notice that, by construction of the hypergraph in section 3.2, each hyperedge has a single qubit-vertex and each gate-vertex is present in exactly two hyperedges.

Non-local gate:a gate $g \in \mathcal{G}$ is non-local if and only if its adjacent qubit-vertices q and q' satisfy $\phi(q) \neq \phi(q')$.

Detached gate:a gate $g \in \mathcal{G}$ is detached if and only if its adjacent qubit-vertices q and q' satisfy $\phi(q) \neq \phi(g)$ and $\phi(q') \neq \phi(g)$.

Distributable packet: a distributable packet P rooted on $\hat{q}$ can be represented as a hyperedge with qubit-vertex $\hat{q}$ and the gate-vertices corresponding to the gates in P. In general, a hyperedge may contain the union of any number of distributable packets as long as they are all rooted on the same qubit $\hat{q}$. Whereas it is necessary for all gates of a distributable packet $g \in P$ to be allocated to the same module $\phi(g)$, this requirement does not apply to hyperedges. As a consequence, we can extract the distributable packets comprising a hyperedge by grouping its gate-vertices in terms of where they are allocated to.

Embedding unit: if two distributable packets may be merged together by embedding the gates between them, the same can be said about merging the hyperedges the packets belong to. As such, embedding techniques alter the hypergraph itself, increasing the size of hyperedges for the sake of reducing their number. Embedding units can be retrieved on demand by inspecting the subcircuit between any two gates on the same hyperedge. Since `Distribution` is meant to capture valid distributions, we assume no embedding conflicts are incurred; it is the responsibility of the optimising method to guarantee that this is satisfied.

Verifying that the bound to computation registers $\omega(\texttt{A})$ of each module $\texttt{A} \in V$ is satisfied is straightforward: simply count the number of $q \in Q$ such that $\phi(q) = \texttt{A}$. The cost in the number of ebits can be inferred using the methods presented in section 4.2. Unfortunately, the satisfaction of bound to link qubit registers $\epsilon(\texttt{A})$ cannot be easily checked using our intermediate representation; instead, we need to generate its corresponding distributed circuit (as detailed in appendix B) and count the number of link qubits used—recall that this is not the same as the number of ebits, since space in the link qubit registers may be reused. This is not an obstacle to our optimisation approaches since none of them consider the bound $\epsilon(\texttt{A})$ within their routines: satisfaction of this bound is deferred to a final pass at the end of the workflow that acts directly on the distributed circuit and is described in appendix C.

4. Distribution techniques

In this section we discuss the novel distribution techniques that we have implemented in pytket_dqc, our DQC tool, available at https://github.com/CQCL/pytket-dqc. Our tool is designed as an extension to `pytket`, the Python interface of the TKET compiler [22] and, as such, it may easily be integrated in a full compilation stack.

Our techniques are orchestrated together in the default workflows detailed on section 5.3. The user may choose to run these default workflows or create a custom one, combining the distribution techniques available as they prefer. Any DQC workflow making use of pytket_dqc should contain the following steps, in this precise order.

Rebase. Rewrite the circuit to an equivalent one in the gateset $\{H, R_Z, CR_Z\}$. Within pytket_dqc we provide an automated method to do so, based upon the rebase passes provided within `pytket`.

Qubit allocation. Assign to which module each qubit of the circuit should be allocated, adhering to the bound on the size of the computation register. Our techniques are based on the hypergraph representation discussed in section 3.5, and the user may choose between an annealing approach or a third-party hypergraph partitioner with a greedy refinement, both of which are detailed in section 4.3. Both of these take advantage of Steiner trees as discussed in section 4.1.

Gate packing. This step is meant to identify opportunities where embedding may be used, passing this information to the next step. In particular, we implemented the algorithm proposed in [15] for this task, whose core ideas are summarised in section 3.3.

Non-local gate distribution. Two options are available: either use the solution provided by the qubit allocation step—distribute gates according to which modules their gate-vertices are assigned to—or make use of the vertex cover approach proposed in [15] and summarised in section 3.4.1. The former option will not take advantage of embedding, but will make use of Steiner trees; conversely, the latter option will consider embedding but not Steiner trees. Neither of these guarantee satisfaction of the bound to the link qubit registers; this is deferred to the last step of the workflow.

Refinement. The previous step makes use of either the embedding technique or Steiner trees. During this refinement step, the user can choose to apply any number of the passes described in appendix A. These refinement passes further improve upon the current solution by taking advantage of readily available opportunities for optimisation using Steiner trees and embedding. The key insight that lets us combine these two seemingly mutually exclusive techniques is described in section 4.2. A refinement that lets us take advantage of detached gates (as in figure 5) is also provided.

Circuit generation. Our tool provides methods for the automatic generation of the distributed circuit as a `pytket` circuit or QASM file. We keep track of the occupancy of the link qubit register of each module and reuse link qubits after the EJPP protocol that employed them terminates. Thus, even though our methods do not guarantee satisfaction of a bound to communication memory, the required memory capacity is not directly dependent on the number of EJPP protocols carried out, but rather on the maximum number of EJPP protocols simultaneously active at any given time. As shown in appendix C, the size of the link qubit registers remains manageable, even if the user does not specify a bound. If the user does specify a bound to link qubit registers, we use the routine described in appendix C to update the distributed circuit as necessary to satisfy the bound, at the cost of increasing the number of ebits required.

Moreover, our tool provides some basic functions for analysing the distributed circuit, such as counting the number of ebits used and the qubit occupancy of the registers of each module. We also provide a method to verify the equivalence between the original circuit and the distributed one, based on [25], which is automatically called at the end of the circuit generation step.

4.1. Gate distribution using Steiner trees

One approach to implementing a distribution hyperedge between two non adjacent modules in a heterogeneous network would be to first construct a single ebit between the relevant modules. This could be done via entanglement swapping; consuming ebits between intermediate modules in the network to build the single required ebit. This single ebit can then be used to perform the EJPP protocol at a total cost in ebits equal to the shortest path in the network between the two modules. In the case where the hyperedge is distributed between three modules, which is to say two distributable packets, and so EJPP processes, are required, the e-bit cost of this approach is the sum of the cost of constructing two ebits. In this case this would be the sum of the shortest paths in the network between the module from which the hyperedge is being distributed, and the two other modules.

During the above described technique, the proxy link qubits in the intermediate modules are measured before the non-local gates have been applied. Alternatively, as these disentangling operation do not affect the qubits which are acted on by the non-local gate, they may be delayed until after the non-local gates have been enacted. Additionally, the starting and ending process commute with the controls of the distributed gates. This means that when non-local gates belong to the same hyperedge are distributed to separate modules, all starting processes can be performed before the gates are enacted, and all ending process may be performed after all gates are acted. This process is depicted in figure 6.

Reusing intermediate link qubits in the aforementioned way reduces the e-bit cost of the distribution to the size of the smallest subtree of the module network which includes the modules of concern. This subgraph

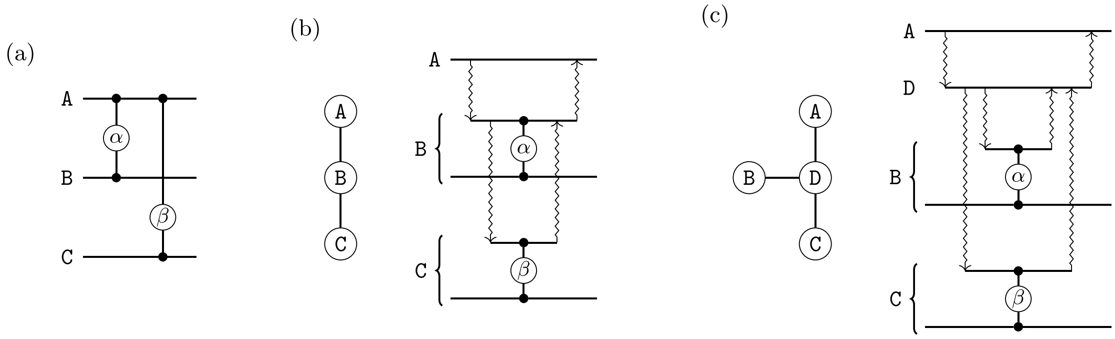


Figure 6. Gate distribution via EJPP embedding with Steiner trees. (a) An input circuit where each qubit is allocated to a different module. (b) The distribution of (a) onto a line network. This should be compared to the approach of using entanglement swapping, where the number of required ebits would be 3. In this case disentangling operation have been delayed until after the non-local gates have been enacted, and starting processes related to β have been commuted to the start of the computation. This results in a total ebit cost of 2 (c) The distribution of (a) in a T shaped network with entangling and disentangling operations acting along the edges of the Steiner tree connecting the relevant modules.

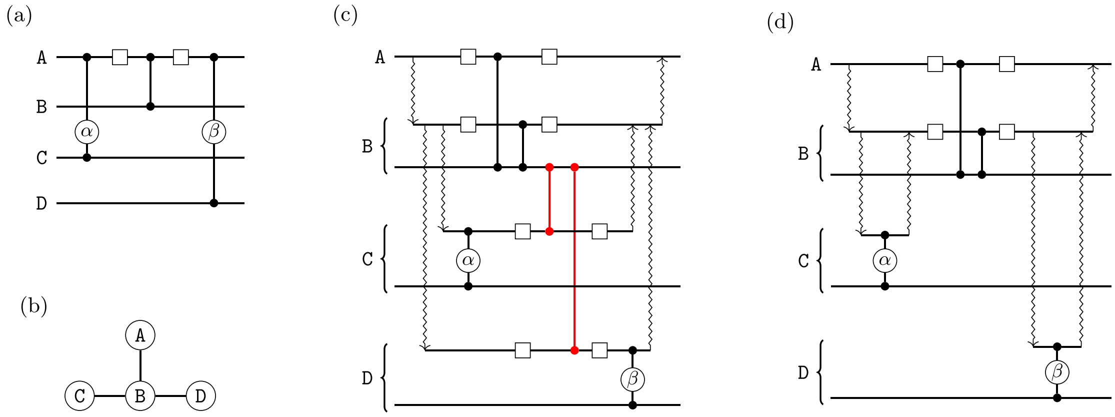


Figure 7. Combining embedding and Steiner trees. (a) An input circuit where each qubit is allocated to a different module, (b) the topology of the network of modules. (c) An equivalent circuit generated using the approach from section 4.1: two non-local correction gates arise, drawn in red. (d) An equivalent circuit generated by algorithm 1.

is known as a Steiner tree. This approach extends to Steiner trees of arbitrary shape, as exemplified in figure 6. Circuit distribution in pytket_dqc makes use of Steiner trees instead of entanglement swapping, allowing us to make savings upon a naive application of the EJPP protocol.

Note that it is not possible to safely commute entangling and disentangling operation as described in the case when Steiner trees are combined with embedding units containing H gates. This is discussed in section 4.2.

4.2. Combining embedding and Steiner trees

The approach proposed in section 4.1 efficiently generates the entanglement sharing required for the distribution of the gates in a hyperedge, using Steiner trees. To do so, we maintain the entanglement of some proxy link qubits throughout the whole duration of the collection of EJPP processes. Unfortunately, if the hyperedge includes any distributable packet that requires some embedding, such as the example in figure 7, maintaining the entanglement of these proxy link causes a problem: correction gates acting on them will be required. As shown in figure 7 these correction gates may be non-local, thus creating the need for extra ebits to implement them, defeating the purpose of embedding.

There is a simple solution to our compatibility issue: maintain the entanglement of these proxy link qubits for as long as possible to maximise the use of Steiner trees, but disentangle them right before an embedding unit so that they do not interfere with it. The implementation of such an intuition is sketched in algorithm 1.

Figure 7(d) shows the result of running algorithm 1 on a simple circuit. The proxy link qubit of module B is maintained throughout the circuit, whereas the link qubits of modules C and D are only maintained as long as necessary to implement the two CR_Z gates. Maintaining the link qubit of module B saves one ebit, whereas our management of the link qubits of modules C and D avoids the need for non-local correction gates that

Algorithm 1. Distribution with embedding and Steiner trees.

Input: hyperedge (hedge), allocation map (ϕ)
Output: partly distributed circuit (dist_circ).

```
tree ← hedge's Steiner tree (see section 4.1)
linked_modules ← ∅
hedge_circ ← extract as in remark 9
iter ← hedge_circ.iterator()

dist_circ ← empty circuit
while iter.current not null do
    gate ← iter.current

    if gate ∈ hedge then                                  ▷ Distribute
        if $\phi$(gate) ∉ linked_modules then
            dist_circ ← insert starting process
            linked_modules ← add $\phi$(gate)
        dist_circ ← insert distributed gate

    else if gate is $H$ then                              ▷ Embed
        embedding_unit ← gate's embedding unit
        remote_module ← module B from lemma 7
        for module ∈ linked_modules do
            if module ≠ remote_module then
                dist_circ ← insert ending process
        linked_modules ← {remote_module}
        dist_circ ← insert embedding_unit
        dist_circ ← insert correction gates
        iter ← move at the end of embedding_unit

    else                                                  ▷ Skip
        dist_circ ← insert gate unchanged
    iter.next()
```

would otherwise be required (see figure 7(c)). Thus, it is possible to define distributions that combine the techniques of embedding and Steiner trees, and algorithm 1 is capable of generating the corresponding circuit.

We can count the number of ebits consumed in the distributed circuit outputted by algorithm 1, thus obtaining the exact ebit cost of the distribution. This can be done for each cut hyperedge in our hypergraph, and it is straightforward to check that algorithm 1 runs in time $\mathcal{O}(g_d + g_e)$ where g_d is the number of gate-vertices in the hyperedge and g_e is the number of gates that need to be embedded to realise its distribution. Thus, this provides an efficient function to calculate the exact ebit cost of a given cut hyperedge, using both embedding and Steiner trees. This cost function will be used by the combinatorial optimisation approaches of appendix A which will be the ones to ultimately decide how each non-local gates should be distributed.

Remark 9. Algorithm 1 iterates over the hyperedge's subcircuit (`hedge_circ`): given a hyperedge whose qubit-vertex is $\hat{q}$, its subcircuit is the sequence of gates from the original circuit that contains all of the gates corresponding to gate-vertices of the hyperedge and every gate in between these that acts on $\hat{q}$. The hyperedge given to algorithm 1 as input is required to be valid, in the sense that every gate in its subcircuit is either distributable or embeddable. We can verify this ahead of time by checking the conditions from lemma 3 (with the amend from remark 4) and lemma 7 respectively.

4.3. Partitioning on heterogenous networks

In section 3.2 we reviewed an approach that reduces the DQC problem on fully connected networks to hypergraph partitioning [12]. In the case of heterogeneous networks, the DQC problem still reduces to (a version of) hypergraph partitioning, but the cost function of a partition is different—since we need to consider the distance between modules—and we must filter out invalid solutions where the module's computation register capacity is exceeded. In this section we propose two approaches to solve this alternative version of hypergraph partitioning and, thus, the DQC problem on heterogeneous networks.

Both of our approaches start from an initial partition and apply rounds of updates to it, guided by the cost function defined in section 4.2. On each round, vertices of the hypergraph are moved from their assigned module to a different one; then, the cost of every hyperedge containing a reallocated vertex is updated. We can calculate the gain of the moves as the difference between the new cost and the previous cost. Depending on the gain and the approach used, the moves will be committed or rolled back. Since calculation of the cost function from section 4.2 requires finding Steiner trees on the network's graph—which is a non-trivial computation—we keep a cache of already computed Steiner trees.

Recall that our hypergraphs have two kinds of vertices: qubit-vertices and gate-vertices. The allocation of a qubit-vertex to a module fills up one slot of the module's computation register, whereas the allocation of gate-vertices do not affect the computation register. Consequently, we assign weight 1 to qubit-vertices and weight 0 to gate-vertices and filter out partitions where the sum of weights in a module exceeds the corresponding module's computation register capacity. If a move would cause the capacity of a module to be exceeded, we select a qubit-vertex on the offending module and swap it with the vertex we intended to move. Our approaches assume unbounded link qubit registers, unlike [14]. In contrast, we make use of Steiner trees as discussed in section 4.1, tapping into optimisation opportunities not considered in the latter work.

4.3.1. Simulated annealing

Simulated annealing is a stochastic optimisation algorithm; modifying an existing solution by randomly searching its neighbourhood. This search process is repeated iteratively, with the working solution updated if a lower cost solution is found. The solution may also be updated with some probability if the cost is higher, which prevents the algorithm from becoming trapped in local optima. The probability of accepting a worse solution falls with each iteration, encouraging that the region of the global optimum be found early on, after which the optimum itself is isolated.

In particular, the initial circuit distribution we use assigns qubits to random modules which have space for them, and assigns gate-vertices to random modules as well. Each step moves a random vertex in the distribution hypergraph to a random module. In the case of qubit-vertices this may require that a qubit in the module be swapped out to make room. The new distribution is accordingly updated depending on the new cost of the distribution.

Each iterations of the annealing procedure makes use of the cost function defined in section 4.2 to accept or reject an update to the distribution. As such, the scheme considers heterogeneous networks and Steiner trees in the first instance. It will not however update the distributable packets, and so considers embedding only in so far as the initial distribution take it into account. In section 5 the initial distribution does not take embedding into consideration. Since annealing is a very general purpose tool and not well optimised to the problem of concern, we do not expect it to perform as well or as quickly as other specialised tool. The technique is however very versatile, and could be easily adapted to other similar problem. Additionally our implementation in `pytket_dqc` avoids dependencies on other third party libraries.

4.3.2. Boundary reallocation

The initial solution of this approach is computed using `KaHyPar` [24], a state-of-art hypergraph partitioner that has the option to fix the maximum vertex weight each partition block can hold. Thus, its solution already provides a valid distribution, in the sense that it does not exceed the computation register capacity of the modules. However, the solution is optimised according to the wrong cost function, since it is assuming an all-to-all network topology. We refine the solution applying a greedy algorithm guided by the cost function defined in section 4.2, improving the allocation of vertices on the boundary between partition blocks.

On each round, we collect all of the vertices in the hypergraph that belong to a hyperedge cut by the partition—the boundary of the partition. For each vertex v in said boundary we find all of the modules that v has a neighbour in; we then calculate the gain of moving v to each of these modules and pick the most advantageous move (with ties broken randomly) or, if all of them are detrimental, we choose not to move v. A round finishes when this routine has been run once for each vertex in the boundary. Thus, each round generates a new partition and the cost of its distribution is decreased monotonically.

There is no attempt to escape local minima. The initial solution provided by `KaHyPar`—which does have strategies to avoid local minima [24]—already identifies groups of qubits that should be allocated to the same module; such grouping is a property of the circuit and hence, equally valid in the context of heterogeneous networks. Unfortunately, our greedy refinement struggles to move vertices that have many neighbours within its allocated module but few in other modules. We expect this to be a noticeable limitation in the case of networks resembling a line graph, where some of these immobile vertices may be stuck on the ends of the network. In practice, however, we expect that modules will be arranged in a small-world network[13] such as a hypercube, where the allocation of a few immobile vertices is not crucial thanks to the network's small average distance. In such cases, the potential for optimisation would primarily come from making smart choices of where the vertices that do not strongly belong to any of the modules (i.e. those in the boundary of the partition), taking into account the topology of the network.

5. Benchmarks

Here we present the results of benchmarking the methods described in section 4, comparing them to [13]. We describe the networks, circuits, and distribution workflows used in sections 5.1–5.3 respectively. The results of the benchmarks are shown and discussed in section 5.4.

5.1. Networks

The following architectures are used in the experiments of section 5.4. Generator methods for these networks are available within `pytket_dqc`.

Homogeneous: All modules are directly connected to all other modules. All modules contain the same number of qubits, and no bound is set on the number of link qubits available in each module. This models an idealised network, and is exemplified in figure 8(a).

We refer to the following collectively as *heterogeneous networks*. We will generate random instances of heterogeneous networks, and they are designed to be representative of real world networks.

Unstructured: Modules are connected according to edges in random Erdós–Rényi graphs, where each possible edge in the graph is added with a fixed probability. In our case we post-select to generate only connected graphs. This is the most common notion of random networks, and is exemplified in figure 8(b).

Scale-free: The distribution of node degrees in a scale-free network follows a power law. Such networks have few nodes, called hubs, with high degree. This is a common model for networks, including the World Wide Web [28]. They can be generated using preferential attachment, where high degree nodes are more likely to receive new edges as nodes are added. This is the case for the Barabási–Albert model [28] of scale-free networks, which we use to generate them here[14]. Scale-free networks are exemplified in figure 8(c).

Small-world: The characteristic path lengths of small-world networks are small, while the clustering coefficient is large [27, 30]. This is compared to random Erdós–Rényi graphs which have small characteristic path and small clustering coefficient. Unlike Scale-free networks, small-world networks do not include hub nodes. Such networks are used to model social networks and are prevalent in engineering due to their communication efficiency [27]. We generate them using the Watts–Strogatz model [30], and exemplify them in figure 8(d).

The particular sizes of the networks we use are listed in the results of section 5.4. In the case of the heterogeneous networks, edge probabilities are set so that the average number of edges incident on each module is two, and qubits are assigned at random to each module. We take that the size of the link qubit register is the largest integer smaller than the average number of computational qubits per module. This means that one would not typically be able to fit the computational qubits of one network module into the link qubit register of another, and as such that networking the modules together results in an increase in the number of computation qubits. Bounds to the size of the link qubit register are not considered in section 5.4, but are explored in appendix C.

5.2. Circuits

The following classes of randomly generated circuits are considered during the experiments of section 5.4.

[13] In a small-world network of N nodes, few of them are adjacent to each other, but the path between any two nodes tends to be of length $\log N$. Small-world networks are common in engineering due to their logarithmic scaling average distance, which reduces communication bottlenecks [27].

[14] We find this broad class of networks to be a well motivated example for the purposes of our comparison. However, practical considerations give subdivisions of the class of scale-free networks [29]. A fine grained analysis of the resulting impact on quantum circuit distribution would be of interest.

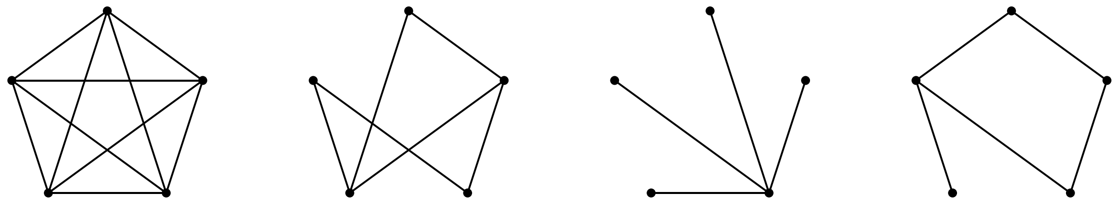

(a) Homogeneous (b) Unstructured (c) Scale-Free (d) Small-World

Figure 8. Example network architecture graphs. Vertices indicate modules. Edges indicate connections along which ebits can be established.

Algorithm 2. Building an instance of CZ Fraction.

input: Width, $n \in \mathbb{Z}$, depth, $d \in \mathbb{Z}$, fraction $p \in [0,1]$
output: Circuit, C_n

```
for each layer $t$ up to depth $d$ do
    for each qubit $q_i$ do
        With probability $1-p$ apply $H$.
    Randomly pair all qubits to which no $H$ was acted.
    To each pair apply $CZ$.
```

Algorithm 3. Building an instance of Quantum Volume.

input Width, $n \in \mathbb{Z}$, depth, $d \in \mathbb{Z}$
output Circuit, C_n

```
for each layer $t$ up to depth $d$ do
    Divide qubits into $\frac{n}{2}$ random pairs $\{q_{i,1}, q_{i,2}\}$.
    for all $i \in \mathbb{Z}, 0 \leqslant i \leqslant \frac{n}{2}$ do
        Generate $U_{i,t} \in \mathrm{SU}(4)$ uniformly at random according to the Haar measure.
        Enact the gate corresponding to the unitary $U_{i,t}$ on qubits $q_{i,1}$ and $q_{i,2}$.    ▷ Decompositions of this gate can be found in [37]
```

Algorithm 4. Building an instance of Pauli Gadget.

input Width, $n \in \mathbb{Z}$, depth, $d \in \mathbb{Z}$
output Circuit, C_n

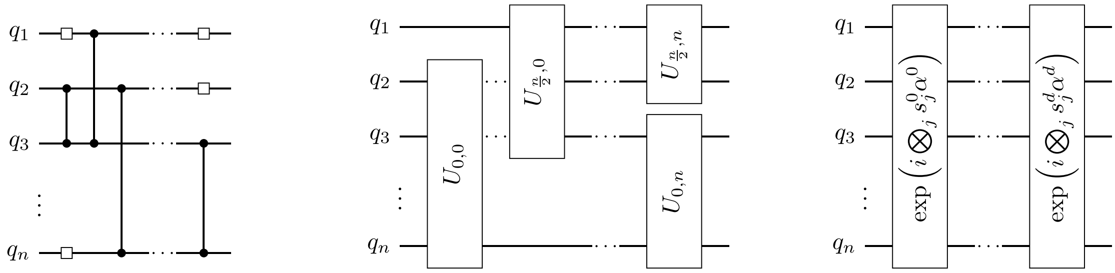
```
for each layer $t$ up to depth $d$ do
    Select a random string $s^t \in \{I, X, Y, Z\}^n$
    Generate random angle $\alpha^t \in [0, 2\pi]$
    Enact $\exp\left(i \bigotimes_j s_j^t \alpha^t\right)$ on qubits $q_1, \ldots, q_n$.    ▷ Decompositions of this gate can be found in [32]
```

CZ Fraction: Consisting of d layers of gates, with each layer built from H and CZ gates. A parameter `cz_fraction` determines the proportion of the qubits on which CZ gates are acted in each layer. These benchmark circuits are already in the gateset considered by the distribution workflows studied, and so provide a controlled way to study the performance of these workflows. CZ fraction circuits are introduced in [13], exemplified in figure 9(a), and detailed in algorithm 2.[15]

While CZ Fraction circuits were designed for the study of DQC workflows, the following are inspired by popular protocols.

Quantum Volume: Consists of d *layers* of random two-qubit gates, each acting on different bipartitions of the qubits, and similar to those used for the quantum volume benchmark [31]. By utilising uniformly random two-qubit unitaries and all-to-all connectivity, Quantum Volume Circuits provide a

[15] Notice that, when `cz_fraction` is close to 1, it will be possible to gather more non-local CZ gates in each distributable packet, since there are very few H gates to impose the constraint listed in lemma 3(c). Indeed, previous works [13] have reported a decrease in ebit count for values beyond a `cz_fraction` of 0.7, and we observe the same behaviour in our figures in section 5.4.


q_1
q_2
q_3
q_n
$U_{0,0}$
$U_{\frac{n}{2},0}$
$U_{\frac{n}{2},n}$
$U_{0,n}$
$\exp\left(i\bigotimes_j s_j^0\alpha^0\right)$
$\exp\left(i\bigotimes_j s_j^d\alpha^d\right)$

(a) CZ Fraction. (b) Quantum Volume. (c) Pauli Gadget.

Figure 9. Examples of circuits used for benchmarks.

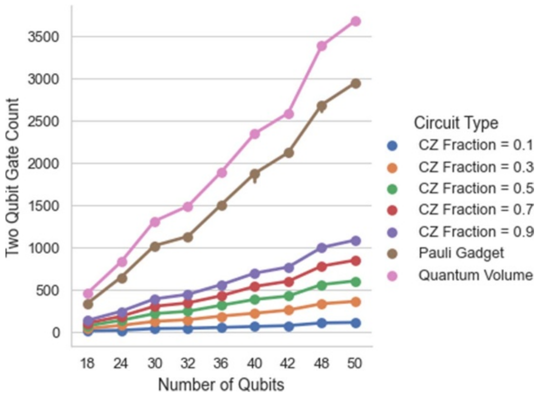


Figure 10. Average number of two-qubit gates for circuit of each type. Points indicate the mean number of two-qubit gates for circuits of the given type, covering the given number of qubits. Error bars indicate one standard deviation.

comprehensive benchmark. While *CZ* Fraction and Pauli Gadget circuits naturally decompose to contain *CZ* gates when rewritten in $\{H, R_Z, CR_Z\}$, Quantum Volume circuits will contain CR_Z gates of a variety of rotation angles. This exemplifies the capacity for `pytket_dqc` to distribute such gates. Quantum Volume circuits are exemplified in figure 9(b) and detailed in algorithm 3.

Pauli Gadget: Pauli gadgets [32] are quantum circuits implementing the exponential of a Pauli tensor. Sequences of Pauli gadgets acting on qubits form *product formula* circuits, most commonly used in Hamiltonian simulation and the variational quantum eigensolver[33–35]. Circuits from this particular class of Pauli Gadget circuits are constructed from several layers of random Pauli Gadgets, each acting on a random subset of n qubits [36]. Pauli Gadget circuits are exemplified in figure 9(c) and detailed in algorithm 4.

In the case of all benchmarks conducted in this work, the number of layers used is set to be equal to the number of qubits in the circuit.

The comparative size of the circuits in these classes is seen in figure 10. Note that *CZ* fraction circuits contain many fewer two-qubit gates than circuits from the other two classes. This is because each layer of the Quantum Volume and Pauli Gadget circuits corresponds to many gates when decomposed into the $\{H, R_Z, CR_Z\}$ gate set. Further, while circuits spanning the same number of qubits in the Quantum Volume class contain more two-qubit gates than those in the Pauli Gadget class, this number is comparable.

5.3. Distribution workflows

This section details the distribution workflows used in the experiments of section 5.4. Our novel distribution workflows improve upon the distributions output by the following schemes, presented in the literature [12, 15] and available through `pytket_dqc`.

`Embed`: Utilises the approach discussed in section 3.4.1 for distributing quantum circuits using vertex covering.

Partition: Utilises the approach discussed in section 3.2 for distributing quantum circuits using hypergraph partitioning.

The following workflows are novel to this work, and are available through pytket_dqc. The refinement passes referenced here are detailed further in appendix A.

EmbedSteiner: All gates in each hyperedge of distributions resulting from Embed act between the same two modules. EmbedSteiner improves upon the output of Embed by merging packets where doing so does not require additional embedding, as discussed in appendix A.3. This results in an ebit saving from reusing proxy link qubits when distributing entanglement according Steiner trees.

EmbedSteinerDetach: Non-local gates are allocated by Embed to either one of the two modules that contain the qubits the gate acts on. Gates are not reallocated by EmbedSteiner. EmbedSteinerDetach improves upon the latter by reallocating gates, making use of detached gates to save extra ebits (details in appendix A.1). Note that this improvement upon Embed is made possible by first refining by merging hyperedges, as in EmbedSteiner, as detached gates may be beneficially utilised when hyperedges contain gates acting between 3 or more modules.

PartitionEmbed: Refines the approach of Partition to make use of embedding (see appendix A.2). This does not consider heterogeneous network connectivity, and we will only use it on homogeneous networks.

PartitionHetero: Recreates the approach of section 4.3.2 to adapt the output of Partition to heterogeneous networks using boundary reallocation.

PartitionHeteroEmbed: Since PartitionHetero neglects the possibility of embedding gates, PartitionHeteroEmbed improves upon it by making use of embedding to merge distributable packets, as discussed in appendix A.2.

Annealing: Utilises the approach of section 4.3.1 for quantum circuit distribution using simulated annealing. Annealing optimises for heterogeneous networks in the first instance, including detached gates and Steiner trees, but does not consider embedding.

The following workflows correspond to existing approaches [13] discussed in section 3.4. The necessary implementations are not available in pytket_dqc but were provided by their authors upon request. These approaches perform qubit allocation by solving a balanced k-min-cut problem over an edge-weighted graph, where the weights capture the connectivity of the circuit. A greedy algorithm that iteratively fixes allocations of non-local gates is used. Each iteration requires solving an instance of the weighted densest subgraph problem in order to pick the allocations to fix at that round. The following two distribution workflows use different methods of solving the weighted densest subgraph problem.

FullG*-Simple:A simple greedy solution.

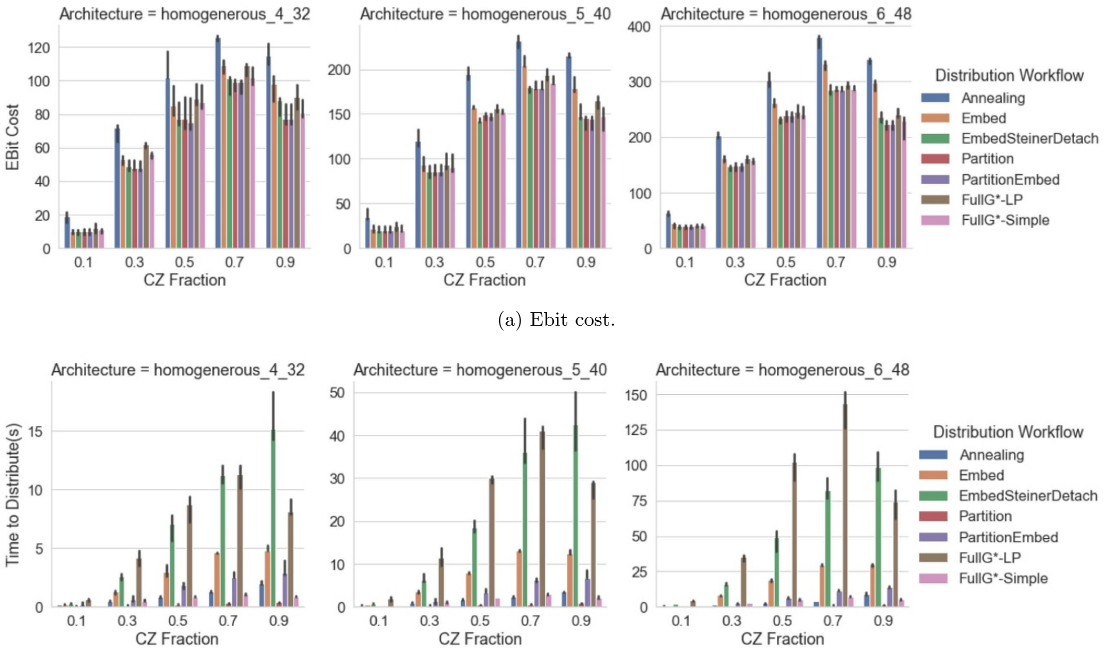
FullG*-LP:An optimal approach based on Integer Linear Programming.

5.4. Results

Here we present the results of the benchmarks described above. We explore homogeneous networks in section 5.4.1, heterogeneous networks in section 5.4.2, and the distribution of a particular circuit of practical interest over heterogeneous networks in section 5.4.3. Data supporting the findings of this section can be found at [26].

5.4.1. Homogeneous networks

We compare the techniques described in section 4 to the techniques of [13], namely FullG*-Simple and FullG*-LP. Aligning with the target scenario of [13], we consider homogeneous networks and CZ Fraction circuits. We consider networks with 4, 5 and 6 modules, each with 8 qubits per module, as well as 2 module networks with 16 and 25 qubits per module. For each network size we generate 5 random CZ fraction circuits of that size.



(a) Ebit cost.

(b) Time to generate distribution, measured in seconds.

Figure 11. Distribution techniques applied to homogeneous networks and *CZ* fraction circuits. Here we use the notation where `homogeneous_n_m` is a homogeneous network connecting `n` modules in a network with a total of `m` qubits. Bars indicate the median over 5 circuits. Error bars indicate 75% percentile range.

Results concerning networks with more than 2 modules can be seen in figure 11. Consistently, the unrefined distribution workflows producing the lowest cost distributions are `Partition` and `FullG*-Simple`[16]. For smaller networks `Partition` mildly outperforms `FullG*-Simple`.

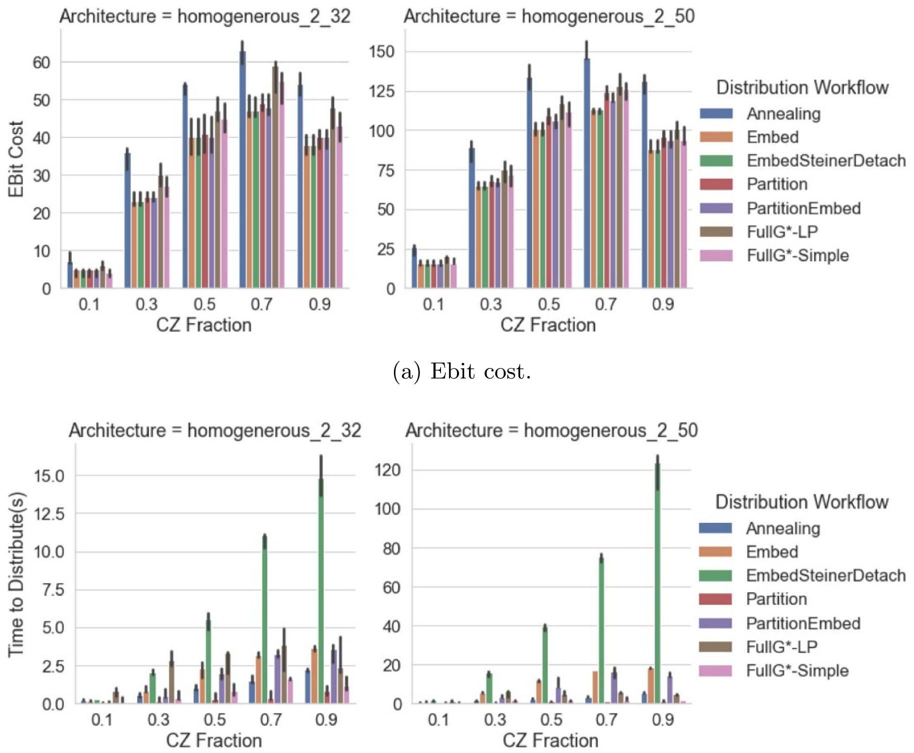
`Annealing` performs the worst overall, which is to be expected as the methods used are particularly general. However, `Annealing` is particularly sensitive to the values of hyper-parameters, particularly the number of annealing iterations performed. Hence, these results may be improved by increasing the number of iterations. Here, the number of iterations is chosen so that the time taken by `Annealing` is roughly comparable to those of the best performing unrefined distribution workflows, as seen in figure 11(b). `Partition` performs the quickest across circuit sizes and *CZ* fractions, while the scaling of `FullG*-LP` and `EmbedSteinerDetach` is the worst. However, as no workflow takes more than a few minutes to complete, the time taken is acceptable in all cases.

`Embed` performs poorly in the results of figure 11(a). This is unsurprising as it corresponds to the original work from [15] which was designed to work best with 2 modules, where detached gates need not be considered. However `EmbedSteinerDetach` significantly improves upon `Embed`, demonstrating the significant potential gains to be made from the use to detached gates. Indeed, in the case of 2 modules, as seen in figure 12, `Embed` performs the best (particularly in the regime of 50 qubits and *CZ* fraction of 0.5 and 0.7). In this case `EmbedSteinerDetach` does not improve the results, as is to be expected since in the 2 module case there is no opportunity for detached gates.

In the case of networks containing more than 2 modules, `PartitionEmbed` barely improves upon `Partition`. This may be because `Partition` produces many detached gates which cannot be embedded by the embedding refinement pass. In the case of 2 server networks, where no gates are detached, `PartitionEmbed` mildly improves upon `Partition`, but does not outperform `Embed`. This demonstrates that embedding can be beneficial when sequences of gates act between 2 modules, but implies that embedding should be considered in the first instance on such networks, rather than through refinement.

We consider the performance of these techniques on the Quantum Volume and Pauli Gadget circuit classes, giving the results in figure 13. Here we consider only network with greater than 2 modules, and so do not consider `Embed` which performs well only on 2 module networks. As these circuits have a significantly larger number of gates than the *CZ* Fraction circuits we consider only the quicker distribution workflows, namely `FullG*-Simple`, `Partition`, `PartitionEmbed`, and `Annealing`.

[16] Note that this contrasts with the results reported in [13]. This is the result of correcting a poor choice of default parameters in [12], which limited how large a hyperedge could be.



(a) Ebit cost.

(b) Time to generate distribution, measured in seconds.

Figure 12. Distribution techniques applied to homogeneous networks and *CZ* fraction circuits over 2 modules. Here we use the notation that `homogeneous_n_m` is a homogeneous network connecting `n` nodes in a network with a total of `m` qubits. Bars indicate the median over 5 circuits. Error bars indicate 75% percentile range.

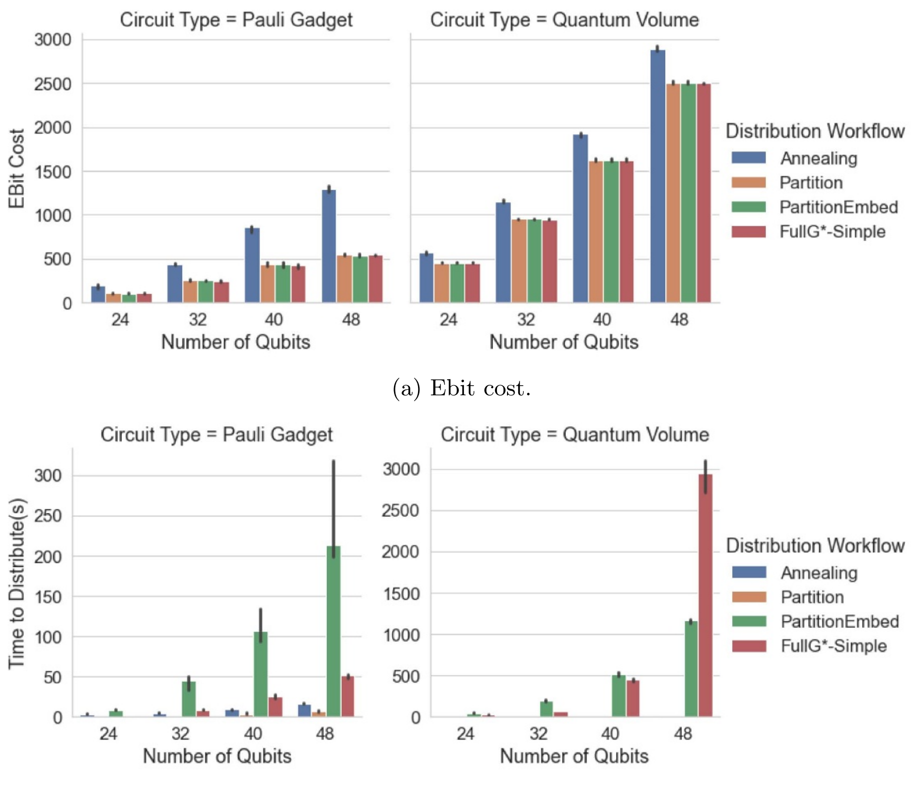
Note that the cost of distributing Pauli Gadget circuits is cheaper for a similar total number of two-qubit gates than the cost of distributing Quantum Volume circuits. Refer to figure 10 for details on comparative 2-qubit gate counts. This is to be expected since the structure of Pauli Gadget circuits, having long sequences of *CZ* gates, allows for the construction of larger distributable packets. For the same reason, Pauli Gadget circuits may be distributed more quickly.

In figure 13 we see a similar pattern to the relative performance of the schemes as we saw in figure 11, namely that there is no significant difference in the e-bit costs of the distributions produced by each workflow, apart from that `Annealing` has a higher cost. `Partition` performs best if both the ebit cost and time taken are considered. `FullG*-Simple` performs similarly well as measured by ebit cost, but the time required to distribute with `FullG*-Simple` scales worse as the number of distributable packets becomes very large, as is the case for the larger Quantum Volume circuits.

5.4.2. Heterogeneous networks

Here we compare the performance of `Embed`, `EmbedSteiner`, `EmbedSteinerDetach` `Partition`, `PartitionHetero`, `PartitionEmbed`, and `PartitionHeteroEmbed`, each of which is capable of performing circuit distribution over heterogeneous networks (although `Embed` and `Partition` are not designed for them). We do not include results for `Annealing` in the plots of this section, as in each case it is outperformed by `PartitionHetero`. We use networks with 3, 4, and 5 modules, each with an average of 6 computational qubits per module. Here we do not bound the size of the link qubit register, instead exploring these bounds in appendix C. For each network size we generate 5 random instances of each of the heterogeneous networks described in section 5.1. The results of these benchmarks can be found in figures 14 and 15, and our findings are detailed below.

`FullG*-Simple` and `FullG*-LP` are not suited to heterogeneous networks, and so the most relevant comparable result is that of [14]. Unfortunately, we were not able to access the implementation of the work of [14] for comparison. The latter work does not make use of Steiner trees for entanglement distribution, which are utilised by all the the schemes presented in this section except `Embed`. The work of [14] does not consider embedding either, which is considered by `Embed`, `EmbedSteiner`, `EmbedSteinerDetach`, `PartitionEmbed`, and `PartitionHeteroEmbed`, and is shown to provide a reduction in ebit cost. As such we expect our techniques to compare favourably to those of [14].



(a) Ebit cost.

(b) Time to generate distribution, measured in seconds.

Figure 13. Distribution techniques applied to homogeneous networks and Quantum Volume and Pauli Gadget circuits. Here we use homogeneous networks built of 3, 4, 5 and 6 modules, each with 8 qubits. Each sample in the experiment corresponds to a single circuit, with 5 samples per bar. Bars indicate the median over five circuits. Error bars indicate 75% percentile range.

5.4.2.1. Refinement has little effect on Quantum Volume circuits

We expect that distributable packets are unavoidably small in the case of Quantum Volume circuits since there are few consecutive CR_Z gates in the circuits and few valid embedding units: the phases of R_Z gates will rarely satisfy condition (d) from lemma 7. In figure 14(a) this manifests in there being no gain from using refinement passes targeted at the use of Steiner trees and embedding.

Additionally, no benefit is found in these circuits when performing boundary reallocation targeted at optimising for the network topology (`PartitionHetero`) and detached gates (`EmbedSteinerDetach`). This again reflects that the hyperedges are too small (often just edges from gate-vertex to qubit-vertex) which, combined with the uniformly random connectivity of the circuit, leads to no window for improvement of the vertex allocation.

5.4.2.2. Each refinement improves the median cost of Pauli Gadget circuits

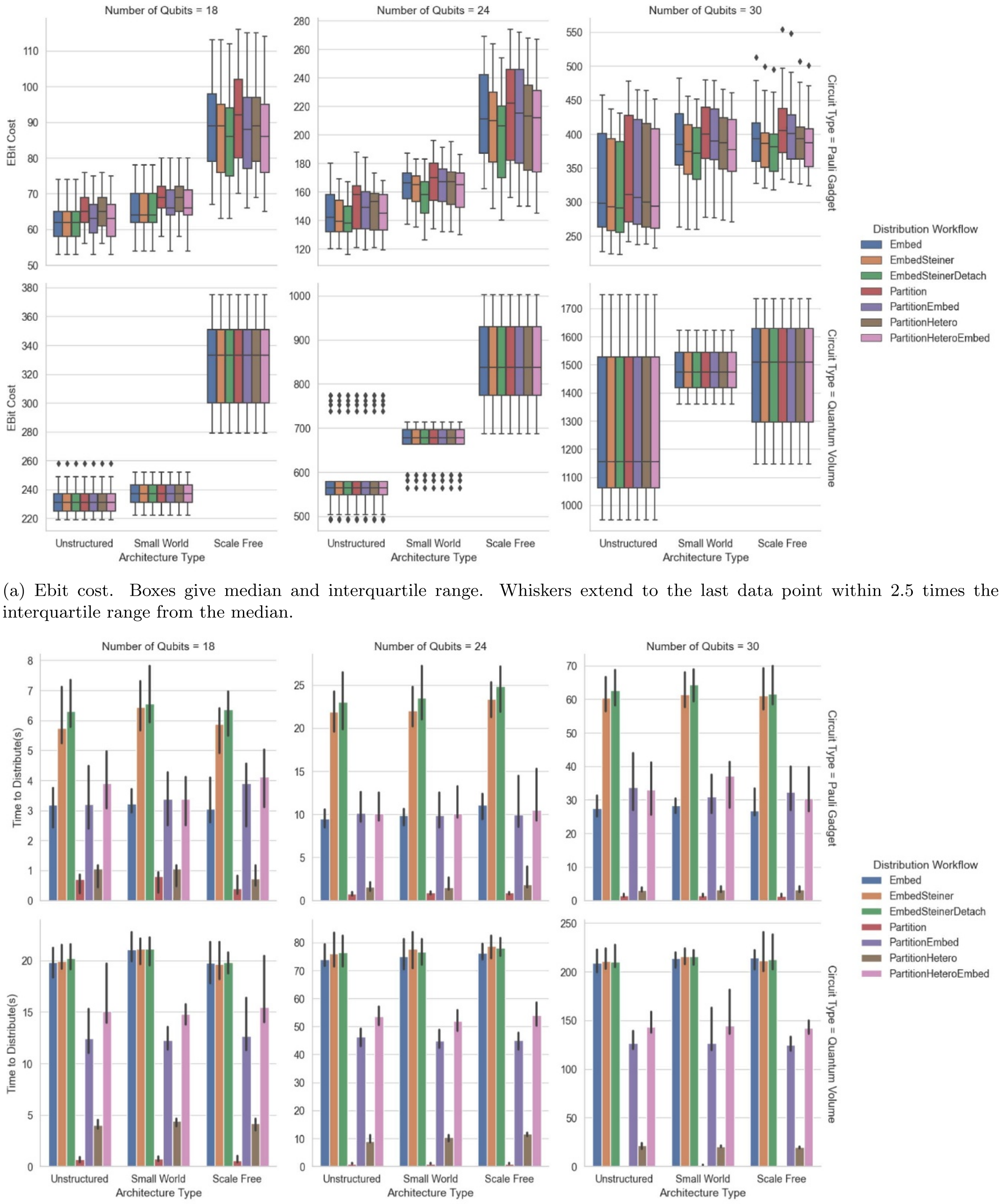
As opposed to Quantum Volume circuits, distributable packets in Pauli Gadget circuits are relatively large, and can be beneficially combined. This is shown in the improvement achieved in figures 14(a) and 15(a) by employing refinement passes making use of Steiner trees, detached gates and embedding.

5.4.2.3. Pauli Gadget circuits are cheaper and quicker to distribute

Figure 14(a) demonstrates that, as a result of Pauli Gadget circuits having larger distributable packets, the cost of distribution of Pauli Gadget circuits is much less than that of Quantum Volume circuits of similar size. Likewise, as seen in figure 14(b), the time required to distribute Pauli gadget circuits is shorter since run time scales primarily with respect to the number of packets, rather than the number of qubits or gates in the circuit.

5.4.2.4. CZ Fraction circuits on networks with more than 2 modules do not benefit greatly form embedding

As observed initially in figure 11(a), we see again in figure 15(a) that refinement to make use of embedding has little impact on the resulting cost of distributing *CZ* fraction circuits onto networks with more than 2 modules. This identifies a middle ground between the more structured Pauli Gadget circuits, which do benefit from embedding, and the larger gate set of the Quantum Volume circuits, which do not benefit from refinement of any kind.



(a) Ebit cost. Boxes give median and interquartile range. Whiskers extend to the last data point within 2.5 times the interquartile range from the median.

(b) Time to generate distribution (seconds). Bars give median over 5 circuits. Error bars are 75% percentile range.

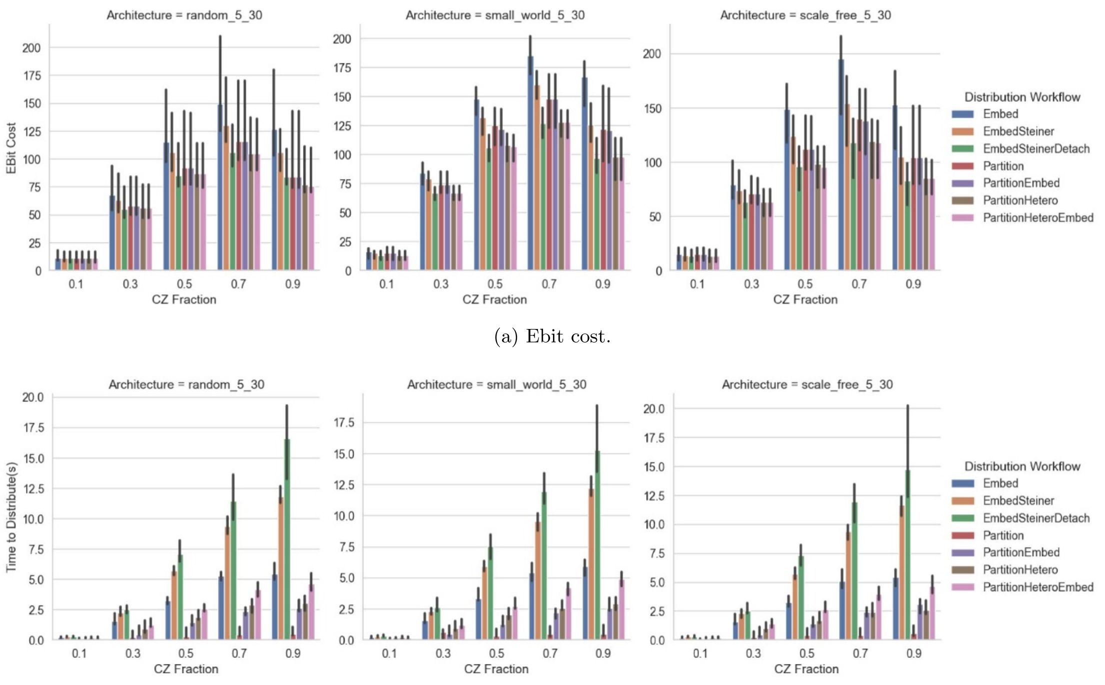
Figure 14. Distribution over heterogeneous networks. Here we use heterogeneous networks built of 3, 4, and 5 modules, each with an average of 6 qubits. Each sample in the experiment corresponds to a single circuit-network pair. Each bar/box considers 5 circuits and 5 networks, giving a total of 25 circuit-network pairs per bar/box.

5.4.2.5. Techniques combined perform best

We see that `EmbedSteinerDetach` typically perform as well or better than the other workflows. This demonstrates the benefit of combining the use of detached gates, Steiner trees, and embedding, and that no one or two alone would perform best. That `EmbedSteinerDetach` mildly outperforms `PartitionHeteroEmbed` on average—which also makes use of detached gates, Steiner trees and embedding—in the Pauli Gadget results of figure 14(a) indicates that embedding is hard to capture in a refinement pass, so it should instead be optimised for in the first instance.

5.4.3. Chemically-aware ansatz

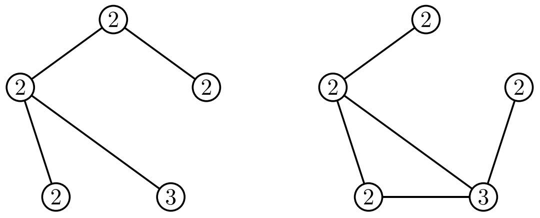
We explore the performance of our approaches in the particular case of a chemically-aware unitary coupled cluster singles and doubles ansatz [38]. We use the example of the minimal basis H_2O molecule with C_{2v} point group symmetry and the 6 electrons in 5 spatial orbital (6e, 5o) active space. The corresponding circuit



(a) Ebit cost.

(b) Time to generate distribution, measured in seconds.

Figure 15. Distribution techniques applied to heterogeneous networks and CZ fraction circuits. Here we use the notation where `type_n_m` is a network of type `type` connecting `n` modules in a network with a total of `m` qubits. Bars indicate the median over 5 circuits. Error bars indicate 75% percentile range.



(a) Network 1 (b) Network 2

Figure 16. Networks for chemistry aware experiments. Numbers in vertices indicate the number of qubits in each module; edges indicate connections alone which ebits can be established.

contains 10 qubits, and is built from Pauli gadgets selected to reflect the symmetries of the system. In the gateset $\{H, R_Z, CR_Z\}$ the circuit contains 463 2-qubit gates.

We distribute this circuit onto the networks of 11 qubits depicted in figure 16, without bounds on the link qubit register sizes. The results are listed in table 1. In the results of sections 5.4.1, 5.4.2 and appendix C the number of qubits in the circuit matches the total number of computation qubits in the network. However our tools are capable of managing situations where there are more computational qubits in the network than are required by the circuit, as demonstrated here.

As expected and indicated by the results of section 5.4.2, we see that the ebit cost decreases with additional refinement. Here it is noticeable that embedding is beneficial, both when introduced as part of a refinement pass, and when introduced during an initial circuit distribution. This shows that real application have circuit structures which benefit from embedding. Indeed it is the case that `EmbedSteinerDetach`, which introduces embedding in the first instance, performs best.

6. Conclusion and future work

In this work we consider the distribution of quantum circuits over heterogeneous networks. We propose a collection of methods for distributing a given quantum circuit over an arbitrary network in a way which minimises the number of ebits required. We make these methods available through pytket_dqc.

Table 1. Distributing chemistry aware ansatz circuits. Networks 1 and 2 are found in figure 16.

	Ebits		Detached		Non-local		Hyperedges	
Workflow	1	2	1	2	1	2	1	2
`Embed`	238	172	0	0	343	340	631	632
`EmbedSteiner`	233	164	0	0	343	340	355	355
`EmbedSteinerDetach`	230	160	13	19	344	342	355	355
`Partition`	295	196	28	28	342	342	397	397
`PartitionEmbed`	276	183	28	28	342	342	352	352
`PartitionHetero`	256	189	35	27	350	344	397	397
`PartitionHeteroEmbed`	253	182	35	27	350	344	353	357

Our first contribution is to introduce two workflows, `Annealing` and `Partition`, which perform quantum circuit distribution over heterogeneous networks in a way which makes use of detached gates. Secondly, where previous work had made use of either detached gates or embedding, we present approaches making use of both simultaneously. We do so by starting from distribution workflows that make use of either of them and then applying rounds of refinement to make the most use of the other approach. Finally, by detecting and optimising common entanglement sharing paths via Steiner trees, and by developing methods to combine their use along with embedding, we further improve our solutions.

We extensively benchmark our distribution workflows on a selection of random and application motivated circuits. We identify that the best workflow to utilise on bipartite networks is `Embed`, while for larger homogeneous networks `Partition` is best. For structured application motivated circuits on heterogeneous networks `EmbedSteinerDetach` is best, while for unstructured Quantum Volume circuits it is best to simply use the fastest workflow, in this case `Partition`.

In the future, optimisation strategies that can take into account the bound to the link qubit registers should be explored further. We are aware of two papers that do so, namely [14, 15]; however, the approach from [14] does not consider the embedding technique nor Steiner trees, while [15] targets networks with only two modules. Moreover, even though the approach from [14] tends to yield solutions that meet the specified bound to the link qubit register, this is not guaranteed—in certain cases, it is necessary to split some of the distributable packets in a similar way we discuss in appendix C.

A second way to more pragmatically treat the problem of quantum circuit distribution would be to include precise measures of the noise within and between QPUs. In this work we have used the number of ebits as a proxy measure for the noise induced by distributing circuits. While we expect the number of ebits to be strongly indicative of the expected noise, new insights may be gained from this more subtle treatment which would include precise noise characterisation information in the appropriate cost functions. This approach would also have the advantage of indicating the performance to be expected from existing technology, and of providing a route to explore the noise levels required to make distributed quantum computing practical. A thorough treatment of the noise internal to QPUs would also reveal the interplay between inter- and intra-QPU noise. We expect to be particularly notable in the case of QPUs with sparse internal connectivity.

Future work may also consider preprocessing of the circuit to facilitate less costly distributions. This is particularly applicable to Pauli Gadgets, which may be decomposed in a variety of ways [32], each of which may be more or less suited to distribution. We additionally encourage the investigation of dynamical quantum circuit distribution, which combines gate teleportation and qubit teleportation. In [12, 14] the authors propose approaches to doing so, suggesting the static distribution of segments of circuits, stitched together via qubit teleportation. The work of this paper can be straightforwardly used as a static distributor in this framework, obtaining similar gains as those reported in [12, 14]. We expect that approaches capable of freely interleaving qubit teleportation and EJPP processes be even more beneficial, and we suggest this is the most pressing line of further work.

Finally, it is relevant to point out that the EJPP protocol shown in figure 1 and discussed in section 3.1 allows for multi-qubit gates—controlled on qubit $\hat{q}$—to be included in distributable packets, as long as the constraints from lemma 3 are satisfied. In [12] it was shown that the hypergraph partitioning approach discussed in section 3.2 is compatible with CCZ gates, and that distributing these gates can provide a major benefit to the ebit count. The qubit allocation approaches for heterogeneous networks we propose in this work (section 4.3) can be readily generalised to multi-controlled gates, since they are based on the hypergraph representation. Similarly, our use of Steiner trees (section 4.1) is agnostic to the gates being implemented and need not be altered. The embedding technique displayed in figure 3(b) can be readily generalised to CCZ, but its treatment within the optimisation algorithm may not be straightforward.

Distribution of multi-qubit gates is a promising avenue of future work, and we believe the flexibility the hypergraph representation provides could be further utilised in such a setting.

Data availability statement

The techniques outlined in section 4 are implemented in pytket_dqc, which can be found in https://github.com/CQCL/pytket-dqc along with example notebooks. Documentation for pytket_dqc can found at https://cqcl.github.io/pytket-dqc/. The results of the benchmarks in section 5 can be found in https://github.com/CQCL/pytket-dqc_experiment_data.

Acknowledgments

The authors thank Ranjani Sundaram, Himanshu Gupta, and C R Ramakrishnan for their insights on their own work, and for sharing the related code. We also acknowledge the contributions made by Kosuke Matsui and Akihito Soeda during early conversations about the project. Thanks to Matty Hoban and Yao Tang for their careful proofreading of this manuscript.

J Y W is supported by Ministry of Science and Technology, Taiwan, R.O.C. under Grant Nos. NSTC 110-2112-M-032-005-MY3, 111-2923-M-032-002-MY5 and 111-2119-M-008-002. T F and M M are supported by: MEXT Quantum Leap Flag-ship Program (MEXT QLEAP) JPMXS0118069605, JPMXS0120351339, Japan Society for the Promotion of Science 21H03394.

Benchmark tools

The results in section 5.4 were obtained using a MacBook Pro with a 2.3 GHz Dual-Core Intel Core i5 processor and 8 GB 2133 MHz LPDDR3 memory. Time to generate distribution in each plot refers to the time taken by this machine.

Appendix A. Refinement

Our software, pytket_dqc, contains several `Refiners` which act on an already valid distribution of a circuit and further reduce its ebit cost. For instance, the approach described in section 4.3.2 is implemented as a refiner. Refiners that have not been explained in the main text are briefly described here; some of these are used in our default workflows listed in section 5.3. Note that sequencing and repeating these refiners may result in greater improvement than a single application, and there exist functionality in pytket_dqc for constructing such sequences.

A.1. Detached gate identification

The approach described in section 4.3.2 refines a hypergraph's partition taking into account the heterogeneous network. It can be repurposed for the task of identifying opportunities where non-local gates may be implemented in a detached manner, i.e. as in figure 5. For this purpose, we impose that qubit-vertices of the hypergraph cannot have their allocation changed, so that the set of non-local gates remains the same. Furthermore, we impose that gate-vertices corresponding to embedded gates are not reallocated either, since algorithm 1 is not capable of accurately estimating the cost of embedding a detached gate. As such, we can apply this approach as a refiner at the end of any workflow, reallocating gate-vertices that do not have a risk of detrimentally interfering with previous optimisations.

This refiner is meant to be applied at the end of any workflow that employs the vertex covering approach discussed in section 3.4.1. On its own, the vertex covering approach cannot take advantage of distribution via detached gates, but such opportunities can be easily identified by hypergraph partitioning approaches, since it is just a matter of allocating the corresponding gate-vertex to a module other than where its qubits are assigned to. Since the approach from section 4.3.2 only changes the allocation of a vertex when doing so reduces the ebit cost of the distribution (calculated using the approach from section 4.2) and non-local gates that may be implemented in a detached manner are necessarily in the boundary of the partition, the refiner will be able to identify opportunities for these and reduce the cost of the distribution accordingly.

A.2. Eager H-type merging

Eager H-type merging refers to the merging of distributable packets via embedding. The refiner scans the circuit qubit by qubit, packet by packet, from start to end, finding opportunities where embedding can be used to merge distributable packets of the given solution. For a given packet P_0 we first identify the next packet P_1 that can be merged via embedding (if any). We check whether an embedding conflict would be created by said merging and whether the embedding unit includes any detached gates. If neither, the refiner merges P_0 and P_1 and, otherwise the packets are not merged. Regardless of the outcome, the refiner continues the search until all pairs of packets have been considered.

This refiner allows for the use of the embedding technique after workflows that do not use it. Since it does not alter the way each of the non-local gates are distributed—it only extends the lifespan of link qubits—it can easily be applied at the end of any workflow. This comes at the disadvantage of not exploiting the potential of the embedding technique to the fullest. If embedding is expected to be the main source of ebit cost reduction on a given distribution, a workflow such as `Embed` or any of its derivatives would be preferable.

A.3. D-type merging

D-type merging refers to the merging of hyperedges when doing so does not require additional embedding. In particular, two hyperedges on the same qubit can be D-type merged when two CR_Z gates, one from each packet, act consecutively on said qubit with no H gates acting between them.

D-type merging has the effect of merging multiple distributable packets—that may share their qubits with different modules—into a single hyperedge. This has the advantage of allowing for greater opportunity to reduce ebit cost through the use of gate distribution via Steiner trees, as discussed in section 4.1. As such, we recommend the use of D-type merging on workflows employing the vertex covering approach of section 3.4 which, on its own, would produce hyperedges involving only two modules each, preventing the use of optimisations based on Steiner trees.

There are two D-type merging refiners in pytket_dqc: `NeighbouringDTypeMerge` and `IntertwinedDTypeMerge`, which differ only in the relative positioning of the distributable packets which they merge. Each refiner iterates through the packets acting on a qubit, merging them when a D-type merge is possible.

Appendix B. Building the distributed circuit

The outcome of each of the approaches discussed in this paper is a `Distribution` which, as established in definition 8, corresponds to a hypergraph along with an allocation of its vertices to modules. However, we ultimately want to convert this abstract data structure to an actual quantum circuit; a method to do so is provided within pytket_dqc. Algorithm 1 described how, given a hyperedge and its allocation of vertices to modules, we can distribute its corresponding subcircuit, implementing the non-local gates corresponding to its gate-vertices via EJPP protocols and embedding as appropriate. In order to distribute the whole circuit, we apply algorithm 1 on the input circuit once per hyperedge in the `Distribution`. There are some subtleties that were omitted in the main text for the sake of brevity and are detailed below.

B.1. Correction gates

These are extra gates that must be applied along with embedding units in order to preserve circuit equality. Recall that the input circuit has been rebased to the $\{H, R_Z, CR_Z\}$ gateset and that, if the embedding unit commutes with the starting process, no correction gates are required (this follows immediately from definition 6). Thus, correction gates are only required by embedding units that begin and end with an H gate. Figure 3 provides the two most simple examples where correction gates appear; from these, the general case can be inferred.

Due to conditions (c) and (d) of lemma 7 we only need to concern ourselves with correcting H, Z and CZ gates. In particular, whenever an H (or Z) gate acting on $\hat{q}$ is being embedded, we must apply an H (respectively, Z) gate on each link qubit that is currently entangled with $\hat{q}$. In the case of a CZ gate, let $\hat{q}$ be the qubit that is being shared and let q' be the other qubit the CZ gate acts on; we require one correction CZ gate per link qubit currently entangled with $\hat{q}$, with the gate acting on such a link qubit and q'. As shown in figure 3, the correction CZ gates are local; this is guaranteed by condition (b) of lemma 7. Repeating this process above for every gate within an embedding unit provides all of the correction gates that are required by algorithm 1.

B.2. Ending processes

When using algorithm 1, a starting process may first entangle a proxy link qubit q with another link qubit q', and then have the ending process disentangling q appear before the disentanglement of q'. In such a situation, q' has lost its immediate predecessor in the entanglement chain, and it may be unclear how to disentangle it. Fortunately, the only gate to be applied on q during the ending process of q' is a classically controlled Z gate (see figure 1) which may be equivalently applied on the root of the Steiner tree: the circuit qubit $\hat{q}$. Since ending processes only use LOCC, the network architecture does not pose an obstacle to their implementation and, hence, there is no dependency between ending processes.

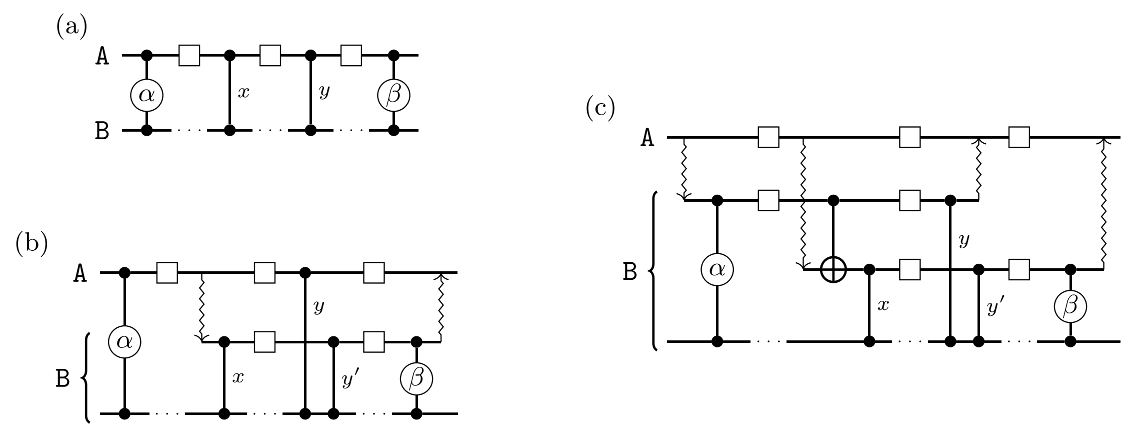
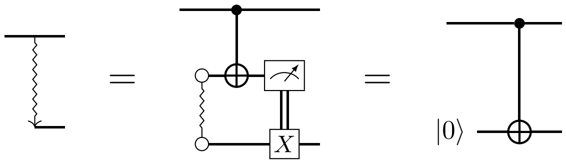


Figure 17. Distribution with intertwined embeddings. (a) Input circuit; two distributable packets are considered, both rooted on the qubit in A: $P_0 = \{\alpha, y\}$ and $P_1 = \{x, \beta\}$. (b) Circuit after distributing P_1; gate y is embedded. (c) Circuit after distributing P_0 as well; we need to embed the starting process of P_1, which requires a local CX correction gate.

B.3. Intertwined embeddings

In situations such as the one depicted in figure 17 where two distributable packets are rooted on the same qubit, applying algorithm 1 on one of them yields a circuit (depicted in figure 17(b) that would require embedding a starting process for the second packet to be distributed. The following equality can be derived by manipulating the circuit required to implement a starting process (see figure 1).



This means that—for matters of embedding—we can treat starting processes just as if they were non-local CX gates. Since a CX gate is equivalent to a CZ sandwiched by H gates, the approach discussed in the main text can be applied directly to embedding units containing starting processes. Doing so yields the distributed circuit from figure 17(c). Proving that the resulting CX correction gate is always local is nontrivial, but it follows from the fact that this intertwining of embeddings can only occur if there is a CZ gate such as y in figure 17(c) that is distributable in one packet and embedded in the other. Then, due to condition (b) of lemma 3 and condition (b) of lemma 7, the remote module B that the qubit is being shared with must be the same for both packets. Consequently, both link qubits live in the same module B and the correcting CX gate is local in B. In the case of ending processes, a similar argument holds, although a simpler approach is to realise that the gate that would need to be embedded is just a classically controlled Z gate and, consequently, its correction is straightforward.

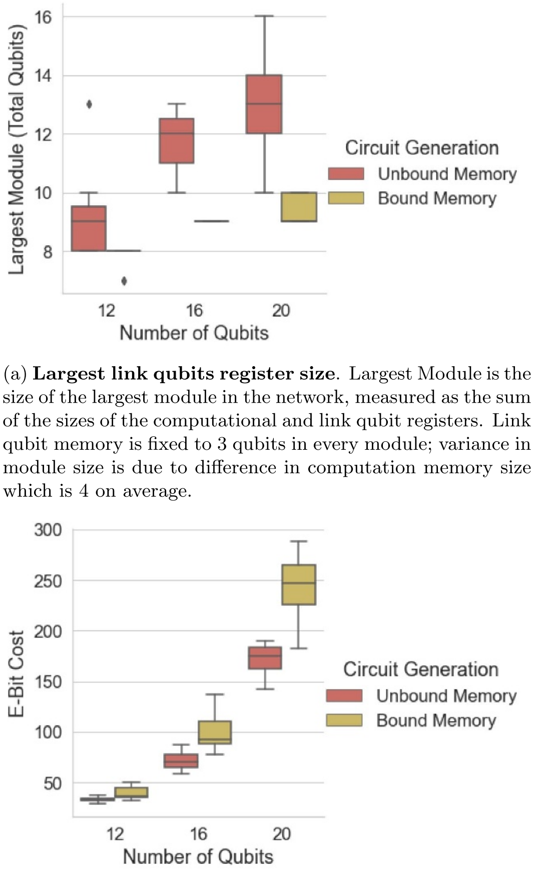
Appendix C. Limited link qubits

In section 5 no bound on the number of available link qubits was imposed. In practice, two considerations bound this quantity:

- Each module has a fixed total number of qubits, and a register of unlimited size dedicated to link qubits would be infeasible. The sum of the number of computation qubits and link qubits required by the distributed circuit should be less than the total number of qubits in each module.
- The sum of the number of computation qubits and the number of link qubits in the largest module should be strictly less than the number of qubits used by the original circuit. If this were not the case then the circuit could be run within the largest module, defeating the purpose of distribution.

In this section we firstly demonstrate that the methods introduced in section 4 do not produce distributed circuits requiring excessively large link qubit registers. Secondly, we introduce an approach to limiting the size of the link qubit register.

We explore the size of link qubit registers across a collection of distributed circuits. As the distributable packets are larger and longer lasting in the case of Pauli Gadget circuits, we will consider them here. As there



(a) **Largest link qubits register size**. Largest Module is the size of the largest module in the network, measured as the sum of the sizes of the computational and link qubit registers. Link qubit memory is fixed to 3 qubits in every module; variance in module size is due to difference in computation memory size which is 4 on average.

(b) **Ebit cost.**

Figure 18. Fixed link qubits register size. Number of Qubits gives the size of the Pauli Gadget circuit, and therefore the total number of computational qubits in the network. Link qubit memory is fixed to 3 qubits in every module. In each plot, boxes show the quartiles of the dataset, whiskers stretch to largest and smallest value within 1.5 of the interquartile range, and remaining points are outliers.

was no noticeable difference in performance between networks in section 5.4.2, we will consider only Small World networks. We take networks with an average module size of 4, considering both unbounded link qubit registers, and link qubit registers bounded to contain 3 qubits. We consider networks with 3, 4, and 5 modules, taking 3 networks of each size. The Pauli Gadget circuits are of the same size as the total number of computational qubits in the network, with 5 random circuits generated for each network. We use the `EmbedSteinerDetach` distributor as it was found to be the best performing in the results of section 5.4.2. The relevant results are presented in figure 18.

C.1. Link qubit register size

As seen in figure 18(a), and with few exception, the largest module required by circuits distributed onto networks with unbounded memory is smaller than the total number of qubits in the original circuit. This is more consistently the case as the number of modules in the network increases, and the size of the largest module appears to plateau. This suggests that the size of the link qubit registers is correlated with the average module size, rather than the number of qubits in the circuit.

C.2. Bounded link qubit registers

Our tool, pytket_dqc, checks whether the distributed circuit exceeds the link qubit register capacity of any module. If it does, the user may request pytket_dqc to amend it, at the cost of extra ebits. We now sketch the approach implemented in pytket_dqc to do so. As discussed in appendix B, the generation of the distributed circuit proceeds iteratively, distributing each of the hyperedges of the `Distribution` one at a time. As we do so, we keep track of the available space of the link qubit registers of each module. Whenever the realisation of a hyperedge would cause the capacity to be exceeded, we make note of the offending module `A` and the non-local gate *g* that this happened at. Then, we find the subset of hyperedges that share their qubit with module `A` (i.e. those that have some of its gate-vertices allocated to `A`) and whose distributable packets span over gate *g*—not necessarily distributing it. These are the hyperedges that require the existence of a link qubit in module A at the time g is distributed. If we split any of these hyperedges into two different ones—by separating the gate-vertices that come before g from those that come after—we remove the need to store its link qubit at the time of the bound violation. Thus, we simply need to use some heuristic to pick one of these hyperedges, update the `Distribution` splitting it, and run the circuit generation routine of appendix B again; this process is repeated as many times as necessary to satisfy the user's bound to the link qubit registers.

The heuristic we use to choose which of the hyperedges to split is simple: we pick the one whose gate-vertices immediately before and after g are furthest apart in the circuit. Intuitively, this identifies the hyperedge whose link qubit in module A is the most 'idle' at the time of the bound violation. Generally speaking, any circuit may be distributed using modules whose link qubit registers are only capable of storing a single link qubit (unless detached gates are used, in which case a minimum of two link qubits per module are required). The harsher the bound, the more ebits will be required to distribute the circuit.

The most relevant comparable result is that of [14], where a technique to bound link register size is introduced. Unlike ours, their main optimisation procedure already considers the bound to link qubit registers and, hence, their distributions tend to satisfy the bound more often than ours. However, as the authors explained, bound satisfaction is not guaranteed by their approach either, which means they sometimes need to apply a final pass similar to ours at the end of the optimisation. When comparing their pass with ours, we find their approach to be too strict: it picks one of the distributable packets causing the bound violation and opt to distribute each of its CZ gates separately, consuming one ebit for each. In contrast, our approach amends the distributed circuit with less ebit overhead, at the cost of requiring a non-trivial search and repeat-until-success approach, which will take longer to run.

Figure 18(a) shows that strictly capping the size of the link qubit register to 3 limits the size of the largest module below that produced by distributing onto networks with unbounded link qubit registers. As expected, and as seen in figure 18(b), the cost in ebits of the resulting distribution is increased in the case of bounded memory.

ORCID iDs

Pablo Andres-Martinez https://orcid.org/0000-0003-4456-7052
Tim Forrer https://orcid.org/0009-0007-1141-2755
Daniel Mills https://orcid.org/0000-0001-5902-3774
Jun-Yi Wu https://orcid.org/0000-0001-8843-9269
Luciana Henaut https://orcid.org/0000-0002-9783-7450
Kentaro Yamamoto https://orcid.org/0000-0002-9994-1200
Mio Murao https://orcid.org/0000-0001-7861-1774
Ross Duncan https://orcid.org/0000-0001-6758-1573

References

[1] Caleffi M, Amoretti M, Ferrari D, Cuomo D, Illiano J, Manzalini A and Cacciapuoti A S 2022 Distributed quantum computing: a survey (arXiv:2212.10609)

[2] Tang W, Tomesh T, Suchara M, Larson J and Martonosi M 2021 CutQC: using small quantum computers for large quantum circuit evaluations *Proc. 26th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)* (Association for Computing Machinery) pp 473–86

[3] Piveteau C and Sutter D 2022 Circuit knitting with classical communication (arXiv:2205.00016)

[4] Eddins A, Motta M, Gujarati T P, Bravyi S, Mezzacapo A, Hadfield C and Sheldon S 2022 Doubling the size of quantum simulators by entanglement forging *PRX Quantum* **3** 010309

[5] Arute F *et al* 2019 Quantum supremacy using a programmable superconducting processor *Nature* **574** 505

[6] van Meter R, Ladd T D, Fowler A G and Yamamoto Y 2010 Distributed quantum computation architecture using semiconductor nanophotonics *Int. J. Quantum Inf.* **8** 295

[7] Monroe C, Raussendorf R, Ruthven A, Brown K R, Maunz P, Duan L-M and Kim J 2014 Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects *Phys. Rev.* A **89** 022317

[8] Nigmatullin R, Ballance C J, de Beaudrap N and Benjamin S C 2016 Minimally complex ion traps as modules for quantum communication and computing *New J. Phys.* **18** 103028

[9] Wehner S, Elkouss D and Hanson R 2018 Quantum internet: a vision for the road ahead *Science* **362** eaam9288

[10] Cacciapuoti A S, Caleffi M, Tafuri F, Cataliotti F S, Gherardini S and Bianchi G 2020 Quantum internet: networking challenges in distributed quantum computing *IEEE Netw.* **34** 137

[11] Cuomo D, Caleffi M, Krsulich K, Tramonto F, Agliardi G, Prati E and Cacciapuoti A S 2023 Optimized compiler for distributed quantum computing *ACM Trans. Quantum Comput.* **4** 1–29

[12] Andrés-Martínez P and Heunen C 2019 Automated distribution of quantum circuits via hypergraph partitioning *Phys. Rev.* A **100** 032308

[13] Sundaram R G, Gupta H and Ramakrishnan C R 2021 Efficient distribution of quantum circuits *35th Int. Symp. on Distributed Computing (DISC 2021)* (*Leibniz International Proceedings in Informatics (LIPIcs)* vol 209) ed S Gilbert (Schloss Dagstuhl–Leibniz-Zentrum für Informatik) pp 41:1–41:20

[14] Sundaram R G, Gupta H and Ramakrishnan C R 2022 Distribution of quantum circuits over general quantum networks (arXiv:2206.06437)

[15] Wu J-Y, Matsui K, Forrer T, Soeda A, Andrés-Martínez P, Mills D, Henaut L and Murao M 2023 Entanglement-efficient bipartite-distributed quantum computing *Quantum* **7** 1196

[16] Cowtan A, Dilkes S, Duncan R, Krajenbrink A, Simmons W and Sivarajah S 2019 On the qubit routing problem *14th Conf. on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)* (*Leibniz International Proceedings in Informatics (LIPIcs)* vol 135) ed W van Dam and L Mancinska (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik) pp 5:1–5:32

[17] Childs A M, Schoute E and Unsal C M 2019 Circuit transformations for quantum architectures *14th Conf. on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)* (*Leibniz International Proceedings in Informatics (LIPIcs)* vol 135) ed W van Dam and L Mancinska (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik) pp 3:1–3:24

[18] Illiano J, Caleffi M, Manzalini A and Cacciapuoti A S 2022 Quantum internet protocol stack: a comprehensive survey *Comput. Netw.* **213** 109092

[19] Kozlowski W, Dahlberg A and Wehner S 2020 Designing a quantum network protocol *Proc. 16th Int. Conf. on Emerging Networking EXperiments and Technologies* (ACM)

[20] Pirker A and Dür W 2019 A quantum network stack and protocols for reliable entanglement-based networks *New J. Phys.* **21** 033003

[21] Meter R V, Satoh R, Benchasattabuse N, Teramoto K, Matsuo T, Hajdusek M, Satoh T, Nagayama S and Suzuki S 2022 A quantum internet architecture *IEEE Int. Conf. on Quantum Computing and Engineering (QCE)* (IEEE)

[22] Sivarajah S, Dilkes S, Cowtan A, Simmons W, Edgington A and Duncan R 2020 t|ket⟩: a retargetable compiler for NISQ devices *Quantum Sci. Technol.* **6** 014003

[23] Eisert J, Jacobs K, Papadopoulos P and Plenio M B 2000 Optimal local implementation of nonlocal quantum gates *Phys. Rev.* A **62** 052317

[24] Schlag S, Heuer T, Gottesbüren L, Akhremtsev Y, Schulz C and Sanders P 2022 High-quality hypergraph partitioning *ACM J. Exp. Algorithmics* **27** 1–39

[25] Kissinger A and van de Wetering J 2020 PyZX: large scale automated diagrammatic reasoning *Electron. Proc. Theor. Comput. Sci.* **318** 229

[26] Andres-Martinez P, Forrer T, Mills D, Wu J-Y, Henaut L, Yamamoto K, Murao M and Duncan R 2023 Distributing circuits over heterogeneous, modular quantum computing network architectures: experiment data (available at: https://github.com/CQCL/pytket-dqc_experiment_data)

[27] Latora V and Marchiori M 2001 Efficient behavior of small-world networks *Phys. Rev. Lett.* **87** 198701

[28] Barabási A-L and Albert R 1999 Emergence of scaling in random networks *Science* **286** 509

[29] Doyle J C, Alderson D L, Li L, Low S, Roughan M, Shalunov S, Tanaka R and Willinger W 2005 The 'robust yet fragile' nature of the internet *Proc. Natl Acad. Sci.* **102** 14497

[30] Watts D J and Strogatz S H 1998 Collective dynamics of 'small-world' networks *Nature* **393** 440

[31] Cross A W, Bishop L S, Sheldon S, Nation P D and Gambetta J M 2019 Validating quantum computers using randomized model circuits *Phys. Rev.* A **100** 032328

[32] Cowtan A, Dilkes S, Duncan R, Simmons W and Sivarajah S 2020 Phase gadget synthesis for shallow circuits *Electron. Proc. Theor. Comput. Sci.* **318** 213

[33] Berry D W, Ahokas G, Cleve R and Sanders B C 2006 Efficient quantum algorithms for simulating sparse Hamiltonians *Commun. Math. Phys.* **270** 359

[34] Peruzzo A, McClean J, Shadbolt P, Yung M-H, Zhou X-Q, Love P J, Aspuru-Guzik A and O'Brien J L 2014 A variational eigenvalue solver on a photonic quantum processor *Nat. Commun.* **5** 4213

[35] Barkoutsos P K *et al* 2018 Quantum algorithms for electronic structure calculations: particle-hole Hamiltonian and optimized wave-function expansions *Phys. Rev.* A **98** 022322

[36] Mills D, Sivarajah S, Scholten T L and Duncan R 2021 Application-motivated, holistic benchmarking of a full quantum computing stack *Quantum* **5** 415

[37] Tucci R R 2005 An introduction to cartan's kak decomposition for qc programmers (arXiv:quant-ph/0507171)

[38] Khan I T, Tudorovskaya M, Kirsopp J J M, Ramo D M, Warrier P W, Papanastasiou D K and Singh R 2022 Chemically aware unitary coupled cluster with ab initio calculations on system model H1: a refrigerant chemicals application (arXiv:2210.14834)