Quantum Information and Computation, Vol. 17, No. 13&14 (2017) 1096–1104
© Rinton Press

EFFICIENT IMPLEMENTATION OF QUANTUM CIRCUITS WITH LIMITED QUBIT INTERACTIONS

STEPHEN BRIERLEY
DAMTP, Centre for Mathematical Sciences, University of Cambridge
Wilberforce Road, Cambridge CB3 0WA, UK

Received September 14, 2016
Revised July 13, 2017

The quantum circuit model allows gates between *any* pair of qubits yet physical instantiations allow only limited interactions. We address this problem by providing an interaction graph together with an efficient method for compiling quantum circuits so that gates are applied only locally. The graph requires each qubit to interact with 4 other qubits and yet the time-overhead for implementing any n-qubit quantum circuit is $4 \log n$. Building a network of quantum computing nodes according to this graph enables the network to emulate a single monolithic device with minimal overhead.

Keywords:

Communicated by: R Cleve & C Williams

1 Introduction

Just as with their classical counterparts, quantum algorithms will be compiled into a sequence of elementary physical operations. Quantum algorithms use arbitrary two-qubit interactions since in the circuit model, gates can be applied to *any* pair of qubits. However, after quantum error correction the allowed logical interactions are limited to a graph that typically has low degree. Beals et al. give a sequence of SWAP gates permuting the qubits so that every interaction occurs between neighbours of the host graph [1]. The time overhead, T, depends on the properties of the graph. Two interesting examples being the k-dimensional lattice which for an n qubit device has overhead $T = O(n^{1/k})$ and the hypercube with overhead $T = O(\log^2 n)$ [2]. Comparing to the solution where each gate is implemented by a separate permutation, this means that the time to permute all n qubits is within a logarithmic factor of the time to move just one.

The hypercube is a powerful network with the ability to sort in time $O(\log^2 n)$. However, the degree of each node grows as $\log n$, which for large n could become difficult to implement and means that new components have to be designed as the device is scaled up. In addition, implementations of optical switches in a noisy network model typically suffer losses and so it is appealing to reduce the degree to a small constant. In this paper we present improvements to the approach taken by Beals et al. in two directions. We reduce the required degree of the network to a small constant and at the same time cut the overhead to $4 \log n$ (see Table 1 for a comparison to previous work). This lowers the cost of implementing arbitrary quantum algorithms on a physical device and makes the required networks more realistic. A device built using this architecture is truly scalable, additional nodes have the same small degree as the existing qubits. In addition, the lower degree means that we have reduced the *total* number of connections by a factor $O(\log n)$.

Graph	Degree	T	S	Emulation method
Complete graph	n	1	1	n/a
1D n.-n.	2	n^2	1	Move individually
		$2n-3$	1	Sorting network [1, 6]
		$O(1)$	n	Teleportation [7]
2D n.-n.	4	$O(\sqrt{n})$	1	Sorting network [1]
Hypercube	$\log n$	$O(\log^2 n)$	1	Sorting network [1]
Cyclic butterfly	4	$4\log n$	2	Theorem 1

Table 1. The time, T, and space, S, overhead of embedding a quantum circuit into the graph restricted by the physical implementation. A key limitation being the degree of the graph which corresponds to number of interactions per qubit. Previous results have applied to the 1D and 2D nearest-neighbour (n.-n.) and hypercube graph. The final line summarizes the main result of this paper. We show that using a cyclic butterfly network reduces both the degree and time overhead in emulating a quantum circuit on a physically realistic device.

In section 2, we introduce hypercube-like networks and in particular, the so called cyclic butterfly network. We then discuss the properties of a cyclic butterfly graph that we need for the main result which is presented in Section 3. Some alternative networks and the application of these ideas to near-term experiments on noisy network architectures are discussed in the conclusion.

2 Hypercubic networks

We represent a network of qubits as an undirected graph. Nodes correspond to single qubits, or qubit plus a single ancilla, and edges correspond to the allowed interactions. The problem of permuting qubits is then similar to routing packets of information in a synchronous parallel computer. SWAP gates exchange quantum information between two nodes or move a quantum state into a node provided there is an available ancilla qubit in the state $|0\rangle$. In comparison to parallel classical computing, the parameters we are interested in are somewhat different. For example, we will think of each node as a *single* (or pair of) qubit(s) rather than a computing node capable of complex operations. We clearly distinguish between the off-line classical computation which is essentially free (provided it is poly-time) from the on-line quantum computing. We also impose the restriction that no two ‘packets’ can be stored at a single node; there is no ‘buffering’ space in a single qubit.

The quantum computer is required to work synchronously at the logically level - of course at the physical scale, entanglement generation or magic state distillation will be probabilistic and gate times will vary. We do not address these issues here but rather assume that sufficient physical resources allow the system to effectively function as a synchronous device.

The computational power of a network is typically described in terms of its ability to emulate the complete graph. Hypercubic networks are variants of the hypercube that are designed to use nodes with constant degree yet maintain its computational power to within a small constant. Since we consider each node as a qubit, the low degree means that we do not require too many possible interactions with other qubits. In addition, hypercubic networks typically have a nice scaling property since we can use the same components in any size quantum computer (although the distance of the interactions may grow). There are many hypercubic networks with prominent examples being the butterfly, cube-connected cycles, Benes network, shuffle-exchange and the de Bruign network (see for example, ref. [8]). We will use the so called cyclic butterfly network (defined below) which has two useful properties; it embeds a Benes network and is invariant under cyclic permutations.

2.1 The cyclic butterfly network

The $n = r2^r$ nodes of an r-dimensional cyclic butterfly network (also called a wrapped butterfly) can be described in terms of the rows and columns of an $r \times 2^r$ array. Each node is labelled by a pair (w, i) where w is a r-bit word corresponding to one of the 2^r rows and i labels the column. Two nodes (w, i) and $(v, i + 1 \mod r)$ are connected by an edge if either they are in the same row, $w = v$ or if w and v differ by precisely one bit in position i. There are no other connections in the network so the degree of every node equals 4. An example of a $n = 3 \times 2^3$ node cyclic butterfly network is given in Fig. 1.

The cyclic butterfly network is closely related to the hypercube. Merging the r nodes in every row into a single node results in the 2^r node hypercube. Like the hypercube, the butterfly network has a simple recursive structure, one r-dimensional butterfly contains two $(r - 1)$-dimensional butterflies.

There are two properties of cyclic butterfly networks that we make use of in our efficient algorithm for moving qubits. The first property is that the graph embeds a so called Benes network [9], meaning that starting from any given node in column $i = 0$, if we advance through the columns in a full cycle $i = 0, 1, 2, \ldots, r$ and then return in the reverse order $i = r, r - 1, \ldots, 0$, we can arrive at any other node. Furthermore, all nodes in column $i = 0$ can be simultaneously permuted in this way without collision. The second property is that the graph is cyclic: reordering the rows $i \mapsto i + 1 \mod r$ results in the same cyclic butterfly graph. Combining these two properties means that every column can traverse a Benes network simultaneously. Thus on a cyclic butterfly graph, we can permute the 2^r row elements in every column without collisions. Note that this is trivially true on a square $\sqrt{n} \times \sqrt{n}$ lattice: we can simultaneously permute the $\sqrt{n}$ entries of every column independently. The crucial difference is that on a cyclic butterfly the time taken is only $2r \approx 2 \log n$ as opposed to $\sqrt{n}$ on a square lattice.

3 Algorithm for permuting qubits

We now present the main result of the paper, that the butterfly network can implement any quantum algorithm with an overhead of $4 \log n$.

Theorem 1 *On a n-qubit cyclic butterfly network, there is a sequence of local gates with depth $4 \log n$ such that the qubit at node a is sent to node $\pi(a)$ for all $a = 1, \ldots, n$ and any permutation $\pi : [1, n] \to [1, n]$.*

Proof We use the row and column structure of the graph. The destinations of every qubit are labeled by 2^r rows, w, and r columns indexed by $i = 0, \ldots, r - 1$. We implement a permutation of all nodes in three steps using this structure: we first permute the rows, then columns and finally the rows again. The only moves we are allowed to make are swapping two qubits or moving a qubit from one node into its neighbours ancilla. In particular, no two qubits can occupy the same node in a single step.

We first permute the entries in each row in such a way that the row destination of every qubit in each column become distinct i.e. after permuting rows, column i, contains every word $w = 0, \dots, 2^r - 1$ for all $i = 0, \dots, r-1$. This is made possible by Hall's Matching Theorem [10] - also called Hall's marriage theorem as it allows two groups of men and women to happily marry. A matching in a graph is a set of edges that have no common vertices. Hall's theorem gives a necessary and sufficient condition for finding a matching and is commonly used in routing problems.

We use the permutation π to construct a bipartite "routing graph" (U, V, E) containing 22^r nodes $U = \{u_1, \dots, u_{2^r}\}$ and $V = \{v_1, \dots, v_{2^r}\}$ and $r2^r$ edges $U = \{e_1, \dots, e_{r2^r}\}$. The U nodes represent the original row location of each qubit and the V nodes are their destination rows. If a qubit in row u_i has a destination row v_j we add the edge (u_i, v_j) so that there are r edges for every node in U and V.

Hall's Matching Theorem then tells us that we can $r-$colour the edges so that no colour is used twice at any node. We can use the Ford-Fulkerson algorithm to find the matching by reducing the problem to a maximum-flow problem [17]. We add two nodes s and t to the graph and connect s to everything in U and t to everything in V. Since each node has unit capacity, a matching is equivalent to the maximum flow from s to t. The classical computation of the Ford-Fulkerson algorithm is bounded by $O(|U||E|) = O(n^2)$ [18]. Having coloured the edges, we now know how to permute the row elements; an operation we can implement in time $r - 1$ using an odd-even sorting network since each row is a 1D nearest neighbour graph (see Appendix).

The $r-$colouring implies that in every column, i, each row label appears exactly once. Using the Benes and pipe-lining properties of the butterfly network discussed in Sec 2.1, we can sort every column according to the row labels in $2r$ steps. In the first r steps, the qubits increment $i \mapsto i + 1 \mod r$, then in the final r step the rows move in the opposite direction $i \mapsto i - 1 \mod r$. Using a single ancilla at each node the time cost is $2r$.

The final part of the algorithm is to permute the rows according to the column labels. Since the destination column labels are now all distinct, this is possible without collisions using odd-even sort.

The total time overhead is thus $T = (r - 1) + (2r) + (r - 1) < 4 \log n$ as claimed. $\square$

Corollary 2 *A quantum computer whose n logical qubits are connected according to the cyclic butterfly network can implement any quantum algorithm with a time and space overhead of $T = 4 \log n$ and $S = 2$ respectively.*

Proof Each time-step in a quantum circuit consists of up to $n/2$ two-qubit gates. The gates define the permutation, π, used in Theorem 1. We place the destination of each pair of qubits involved in a gate so that they are neighbours in the cyclic-butterfly graph. The proof of Theorem 1 provides an efficient method to construct a sequence of gates implementing the permutation. Every time step requires one permutation of the qubits so the time and space overhead is precisely that given in Theorem 1. $\square$

4 Conclusion

Fault-tolerant quantum computers based on the circuit model are highly parallel machines. Every qubit is effectively a processing node since the identity gate will be error corrected at a cost similar to other gates. Taking this view has led to the application of techniques developed for routing in synchronous parallel (classical) computers. We presented an efficient method for compiling a quantum circuit onto a cyclic-butterfly network. This improves on previous results in two respects. The interaction graph has constant degree and at the same time, the time overhead is a small constant away from the best possible (the time to move a single qubit).

There are two alterations to the cyclic butterfly graph one could make that achieve a trade-off between the cost of building the network and the time-overhead in emulating arbitrary circuits.

1. Replace each node by a ring of 4 nodes, each connected to one of the previous edges. This reduces the connectivity to 3, the minimum possible non-trivial degree, whilst increasing the time overhead by a factor 2.

2. Use the k-arry cyclic butterfly graph where the words labelling the k^r rows is given in radix-k. In this case, the degree increase to $2k$ whilst reducing the overhead to $T = 4 \log_k n$.

Combining these two ideas results in a slightly more efficient solution than the cyclic butterfly graph. The k-arry cyclic butterfly with each node expanded to a ring of $2k$ nodes has degree 3 and time overhead $T = 4k \log_k n$, thus taking $k = 3$ is optimal.

The ideas presented here can used when designing the communication architecture in a noisy network quantum computer. Individual nodes (or cells) correspond to a small number of physical qubits in a system such as NV centers in diamond, trapped ions or superconducting devices. Photonic channels mediate entanglement between two nodes which can then be distilled to allow inter-node communication (see, for example, recent experimental results in NV centers [11], superconducting qubits [12] and trapped ions [13]). Nickerson et al. show how these resources could be used to implement a fault tolerant computation via the surface code even in the presence of noisy photonic links [14]. An alternative approach would be to take advantage of the cyclic butterfly graph and use CSS block codes. Steane described how fault tolerant operations can be performed on separate CSS block codes via ancilla states [15, 16]. Thus nodes could correspond to a small number of logical qubits, each in a separate block. The ancilla states would then be distilled using the photonic channel in much the same way as 4-qubit GHZ states are required when using the surface code.

Acknowledgments

The author would like to thank Aram Harrow and Naomi Nickerson for suggesting the two alternative graphs given in the conclusion, and Tom McCourt for discussions when the idea was at an early stage.

References

1. R. Beals, S. Brierley, O. Gray, A. Harrow, S. Kutin, N. Linden, D. Shepherd and M. Stather, *Efficient Distributed Quantum Computing*, Proc. R. Soc. A 2013 469, 20120686. arXiv:1207.2307

2. There is an asymptotically better algorithm with overhead $O(\log n)$ on the hypercube. However, we will not consider it here since it is based on the AKS sorting algorithm [3, 4] which has constant $\approx 6,100$ [5].
3. M. Ajtai, J. Komlos and E. Szemeredi, *An $O(n \log n)$ sorting network,* Proc. 15th annual ACM symposium on Theory of computing, 1 (1983)
4. M. Paterson, *Improved sorting networks withO (logN) depth*, Algorithmica 5 (1-4), 75-92, (1990)
5. D. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, 1998
6. Y. Hirata, M. Nakanishi, S. Yamashita and Y Nakashima, *An efficient conversion of quantum circuits to a linear nearest neighbor architecture,* Quantum Information & Computation 11, 142 (2011)
7. D. Rosenbaum, *Optimal Quantum Circuits for Nearest-Neighbor Architectures*, pg 294, 8th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2013), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. arXiv:1205.0036
8. F. Leighton, *Introduction to parallel algorithms and architectures,* Morgan Kauffman Publishers, San Mateo, CA. (1992)
9. V. Benes, *Mathematical theory of connecting networks and telephone traffic*, Academic Press Inc., New Yrok, 1965.
10. P. Hall, Philip, *On Representatives of Subsets*, J. London Math. Soc. 10 (1): 26–30, (1935)
11. H. Bernien, B. Hensen, W. Pfaff, G. Koolstra, M. S. Blok, L. Robledo, T. H. Taminiau, M. Markham, D. J. Twitchen, L. Childress and R. Hanson, *Heralded entanglement between solid-state qubits separated by three metres,* Nature 497, 86–90 (2013)
12. N. Roch, M. E. Schwartz, F. Motzoi, C. Macklin, R. Vijay, A. W. Eddins, A. N. Korotkov, K. B. Whaley, M. Sarovar, and I. Siddiqi, *Observation of Measurement-Induced Entanglement and Quantum Trajectories of Remote Superconducting Qubits,* Phys. Rev. Lett. 112, 170501 (2014)
13. D. Hucul, I. Inlek, G. Vittorini, C. Crocker, S. Debnath, S. Clark and C. Monroe, *Modular entanglement of atomic qubits using photons and phonons,* Nature Physics 11, 37–42 (2015)
14. N. Nickerson, J. Fitzsimons and S. Benjamin, *Freely scalable quantum technologies using cells of 5-to-50 qubits with very lossy and noisy photonic links*, Phys. Rev. X 4, 041041. arXiv:1406.0880
15. A. Steane, *Efficient fault-tolerant quantum computing,* Nature 399, 124-126 (1999). arXiv:quant-ph/9809054
16. T. Brun, Y.-C. Zheng, K.-C. Hsu, J. Job and C.-Y. Lai, *Teleportation-based Fault-tolerant Quantum Computation in Multi-qubit Large Block Codes.* arXiv:1504.03913
17. T Cormen, C Leiserson, R Rivest and C Stein, *Introduction to Algorithms*, MIT Press Cambridge Massachusetts 2009
18. L. Ford and D. Fulkerson, *Maximal flow through a network*, Canadian Journal of Mathematics 8: 399 (1956)

Appendix A Sorting Networks

A sorting network is designed to sort all possible input sequences using only comparison gates acting on neighbouring nodes $(x, y) \in G$,

$$C(x,y) = \begin{cases} (x,y) & \text{if } x > y \\ (y,x) & \text{if } x < y. \end{cases}$$

That is, $C(x, y)$ swaps the inputs if $x < y$ and leaves them unchanged otherwise. Sorting networks have been well studied in the classical literature and examples are know over various graphs [5]. Two examples are odd-even and bitonic sort that sort over the 1D nearest-neighbour and hypercubic graphs respectively (see Fig A.1).

A sorting network over a graph, G, provides a method of compiling any circuit onto G. Each time-step in the original circuit defines a permutation; qubits are moved so that the gates become local in G. The classical compiler then inputs the destinations into the sorting network and each time the comparison gate implements a SWAP, the compiler applies a SWAP gate to the corresponding qubits. By construction, every operation is local in G and once the required gates from the sorting network have been added, the gates from the time-step in the original circuit can be enacted on neighbouring qubits. Note that it is not necessary to have a sorting network that correctly sorts all inputs, we only need to sort the inputs that appear in the circuit. In addition, one could use a different network for each time-step.

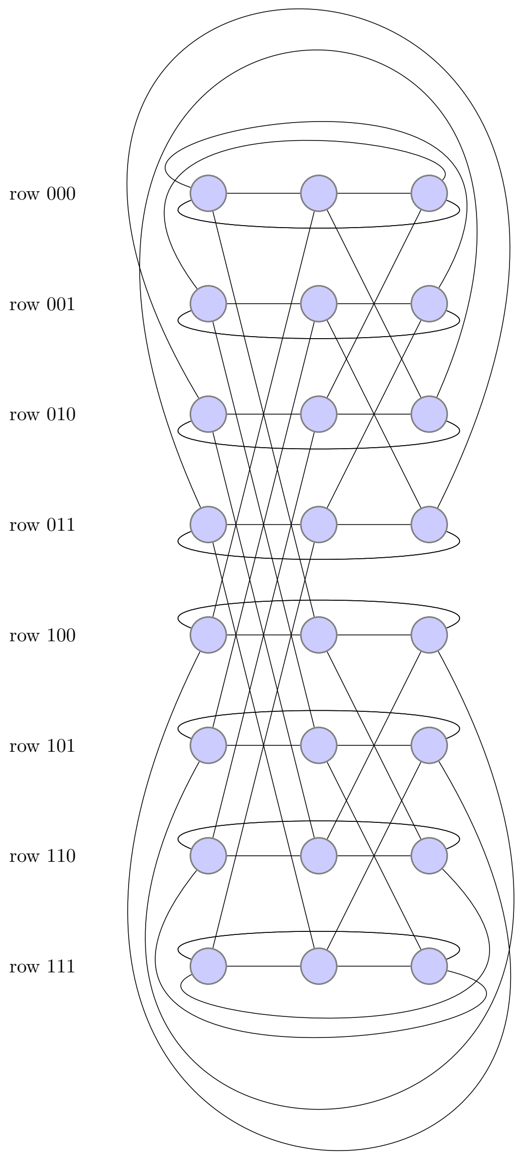
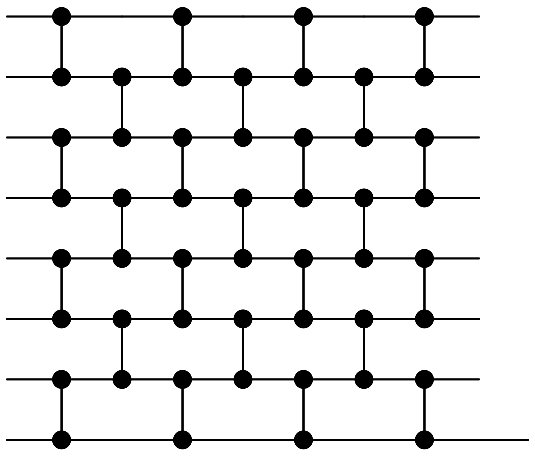
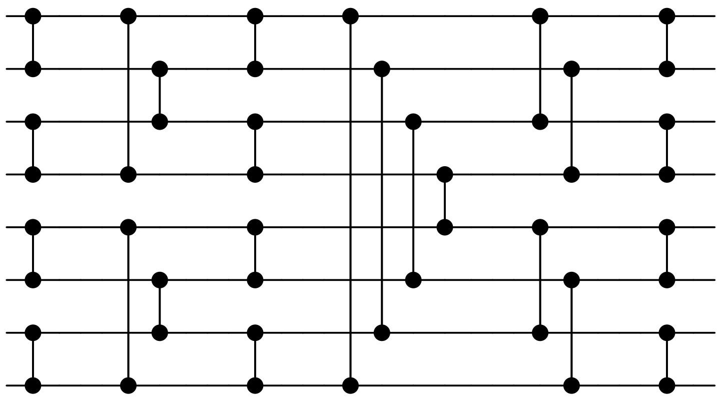


Fig. 1. A 3-dimensional cyclic butterfly graph with $n = 3 \times 2^3$ nodes representing a qubit plus its ancilla. The edges represent the allowed interactions between qubits.



(a)



(b)

Fig. A.1. Two examples of sorting networks on 8 inputs: (a) the odd-even sort over a 1D nearest neighbour graph which sorts in time $T = n - 1$, and (b) the bitonic sort over the hypercube that requires time $T = \frac{1}{2} \log n(\log n + 1)$.