Efficient distributed quantum computing

Robert Beals\textsuperscript{1}, Stephen Brierley\textsuperscript{2}, Oliver Gray\textsuperscript{2}, Aram W. Harrow\textsuperscript{3}, Samuel Kutin\textsuperscript{1}, Noah Linden\textsuperscript{4}, Dan Shepherd\textsuperscript{2,5} and Mark Stather\textsuperscript{5}

\textsuperscript{1}Center for Communications Research, 805 Bunn Drive, Princeton, NJ 08540, USA
\textsuperscript{2}Heilbronn Institute for Mathematical Research, Department of Mathematics, University of Bristol, Bristol BS8 1TW, UK
\textsuperscript{3}Center for Theoretical Physics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
\textsuperscript{4}Department of Mathematics, University of Bristol, Bristol BS8 1TW, UK
\textsuperscript{5}CESG, Hubble Road, Cheltenham GL51 0EX, UK

We provide algorithms for efficiently moving and addressing quantum memory in parallel. These imply that the standard circuit model can be simulated with a low overhead by a more realistic model of a distributed quantum computer. As a result, the circuit model can be used by algorithm designers without worrying whether the underlying architecture supports the connectivity of the circuit. In addition, we apply our results to existing memory-intensive quantum algorithms. We present a parallel quantum search algorithm and improve the time–space trade-off for the element distinctness and collision finding problems.

1. Introduction

There is a significant gap between the usual theoretical formulation of quantum algorithms and the way that quantum computers are likely to be implemented. Descriptions of quantum algorithms are often given in the quantum circuit model in which, for an $N$-qubit circuit, there are up to $N/2$ simultaneous two-qubit gates in any one time step. The model allows arbitrary interactions between qubits, even though any implementation is likely to be mostly local in two or three
dimensions, with a small number of long-range connections. This is unlike classical computers, whose algorithms and implementations are currently dominated by von Neumann architectures where a small number of cores share access to a large random access memory (RAM).

On the other hand, sometimes the circuit model is also less powerful than we would like. The concept of RAM or an analogue of the classical parallel RAM model would be useful for algorithm design. We introduce the quantum parallel RAM (Q PRAM) model that, in addition to any step of the circuit model, allows simultaneous queries to a shared quantum RAM. This allows us to design new quantum algorithms for parallel database searching, and the element distinctness and collision finding problems.

We demonstrate that a single idea—reversible sorting networks—can be used to efficiently relate the circuit and parallel RAM models of computation to one that is more physically realistic. We provide an efficient scheme for implementing a quantum circuit on a physical device where the possible pairwise qubit iterations are restricted. We then present a quantum circuit for implementing the (Q PRAM) model, establishing that quantum computers can efficiently access memory in a parallel and unrestricted way. Combining these two results means that quantum memory does not need to be in a single place, but can be distributed among many processors.

Our algorithms allow us to relate any quantum algorithm presented in terms of a quantum circuit to a distributed quantum computer in which each processor acts on a few well-functioning qubits connected to a small number of other memory sites (possibly via long-range interactions). At the most basic level, a ‘memory site’ could be a single qubit. Thus, experiments such as nitrogen vacancy centres in diamond, and trapped ions connected using optical cavities [1–3] or cavity electrodynamics for superconducting qubit networks [4–6], could be used to efficiently implement quantum algorithms presented in the circuit model.

Even in architectures strictly constrained to nearest-neighbour interactions in a constant number of dimensions, our results provide efficient simulations of the circuit model. For example, suppose we wish to implement a circuit with many concurrent operations on a one-dimensional nearest-neighbour machine. One approach would be to use SWAP gates to bring each pair of qubits together so that the gate can be performed locally. For highly parallel circuits, this would require as many as $O(N^2)$ time steps as there are $N/2$ gates involving qubits that could be as far as $O(N)$ qubits apart. Using sorting networks (in this case the insertion sort), Hirata et al. show how to perform all $N/2$ gates in only $O(N)$ time [7]. This is a special case of our results for the one-dimensional nearest-neighbour architecture. Additionally, we prove that this is the best you can do; a one-dimensional nearest-neighbour machine cannot do better than emulate an arbitrary circuit with an $O(N)$ overhead.

Our work suggests to experimentalists how to begin building architectures for quantum computers. There is little need to build a quantum computer in which every qubit has the ability to interact with every other qubit. The addition of only a few long-range (flying) qubits allows the connectivity of a hypercube. There are efficient sorting networks over the hypercube (such as the bitonic sorting network), so the overhead for emulating arbitrary quantum circuits on such a device is small.

Implicit in our work is the assumption of full parallelism. In classical computing, this is generally unrealistic because communication is expensive and storage elements can be much cheaper than computation elements. It may be that a passive fault-tolerant quantum memory will later be invented, which will make storage costs of quantum memory substantially lower than the costs of performing computation. However, it remains an open question whether such memories are physically possible in fewer than four spatial dimensions. Instead, every known scheme for fault-tolerant quantum computing requires essentially full parallelism, even if the goal is simply to store quantum information without performing any computations. Thus, in this work, we will consider ‘space’ and ‘number of computing elements’ to be roughly synonymous.

Returning to the more theoretical aspects of our work, the results can be summarized as relating the following three models of quantum computation presented in table 1: the well-known circuit model (Q circuit), Q PRAM and a more realistic distributed quantum computing (DQC) model (also called a quantum multi-computer by van Meter et al. [8,9]).
Table 1. Three models of quantum computation.

<table>
<thead>
<tr>
<th>model</th>
<th>in each time step</th>
<th>limitations/features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q circuit</td>
<td>up to (N/2) two-qubit gates are performed on arbitrary disjoint pairs of qubits</td>
<td>no physical implementation since it requires all qubits to interact not clear how to perform simultaneous memory look-ups</td>
</tr>
<tr>
<td>Q PRAM</td>
<td>one step of circuit model or many simultaneous queries to a shared RAM</td>
<td>RAM should be locally writeable (i.e. address (i) controlled by processor (i)) but globally accessible</td>
</tr>
<tr>
<td>DQC</td>
<td>(N) small processors interact in a fixed low-degree graph (see table 2 for examples)</td>
<td>can simulate the above two models with degree (O(\log N)) the overhead is (O(\log^2 N))</td>
</tr>
</tbody>
</table>

We prove that these three models are equivalent up to polylogarithmic depth overhead, as long as the DQC model is defined with respect to a sufficiently connected graph, such as the hypercube. See theorems 3.1 and 3.2 and corollary 3.3 for explanations of how the DQC model can simulate the circuit model. Theorems 4.2 and 4.4 give precise descriptions of how the circuit model can simulate a machine with Q PRAM.

In theorems 3.1 and 3.2, we explain how a distributed quantum computer can implement arbitrary circuits with low overhead. Our results are applied after quantum error correction so that the host graph, \(\mathcal{G}\), in the DQC model consists of logical qubits and gates (layer 4 in [10]). The emulation process plays a role similar to the Solvay–Kitaev theorem [11] for approximating arbitrary single qubit gates given a universal gate set. A general quantum algorithm in the circuit model may involve arbitrary two-qubit interactions. We efficiently map these gates to a sequence of logical gates restricted to \(\mathcal{G}\).

Our main result is a new tool for quantum algorithm design. Theorem 4.2 gives a quantum circuit for efficiently accessing quantum memory in parallel. The key primitive we need to implement is the following: given registers \(|i_1,\ldots,i_N, x_1,\ldots,x_N\rangle\) and \(|y_1,\ldots,y_N\rangle\) with \(|i_j, x_j, y_j\rangle\) controlled by processor \(j\), we would like to replace the \(|y_j\rangle\) register with \(|y_j \oplus x_j\rangle\). In other words, processor \(j\) has specified an index \(i_j\) and would like to query the \(x_1,\ldots,x_N\) string at this position, although this string is distributed across \(N\) processors. We achieve this using sorting networks, which are methods of sorting \(N\) objects via a series of pairwise reorderings of the form ‘if \(X < Y\) then swap \(X\) and \(Y\)’. Such networks exist on the \(N\)-vertex hypercube using \(O(\log^2 N)\) depth [12] and in many other graphs (see table 2 or Ajtai et al. [13]). We can implement a sorting network reversibly by saving a bit for each comparison to record whether a swap took place. We stress that no assumptions are made about the distribution of query indices, and that the algorithm works equally well if the queries are random or concentrated on a single node.

Our results apply when the processors are connected via a general graph with a topology-dependent overhead factor. The key consideration is the required depth of a sorting network. However, in order to reduce the number of qubits required to save ‘which path’ information, in some cases, it is advantageous for a weakly connected architecture (such as a two-dimensional grid) to first simulate a highly connected architecture (such as a hypercube). Doing so would allow a \(\sqrt{N} \times \sqrt{N}\) grid of qubits to simulate the Q PRAM model with \(O(\sqrt{N}\text{poly log} N)\) depth overhead, which is optimal up to the poly log \(N\) factor.

(a) Element distinctness

An important application of the Q PRAM model and the efficient circuit for its implementation (theorem 4.4) is to quantum algorithms that use large amounts of memory. For example, the element distinctness problem asks whether a function \(f:\{N\} \to \{N\}\) is one to one or has \(x \neq y\) such that \(f(x) = f(y)\). By guessing pairs of inputs, this can be solved with a classical computer with space \(S\) and time \(T\) constrained by \(ST = O(N^2)\). Observe that a naive
Table 2. Comparison of various $N$ node graphs giving the graph valency and examples of sorting networks over $G$. The depth of the sorting network, $D_G$, translates into the cost of running $V_h$ via theorem 2.2. AKS, Ajtai–Komlós–Szemerédi.

<table>
<thead>
<tr>
<th>graph</th>
<th>valency</th>
<th>sorting network</th>
<th>$D_G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-dimensional nearest neighbour</td>
<td>2</td>
<td>bubble/insertion sort</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>two-dimensional nearest neighbour</td>
<td>4</td>
<td>bubble/insertion sort</td>
<td>$O(\sqrt{N})$</td>
</tr>
<tr>
<td>hypercube</td>
<td>log $N$</td>
<td>bitonic sort [12]</td>
<td>$O(\log^2 N)$</td>
</tr>
<tr>
<td>complete graph</td>
<td>$N$</td>
<td>any sorting algorithm</td>
<td>$O(\log N)^4$</td>
</tr>
</tbody>
</table>

\(^2\)Includes a large constant $c \approx 6,1000$.

application of Grover’s algorithm [14] fails to achieve any improvement in the number of queries: there are $O(N^2)$ pairs to search, requiring $O(N)$ queries. There is an elegant quantum algorithm by Ambainis [15] that achieves the optimal query complexity of $O(N^{2/3})$. However, it also requires memory $N^{2/3} / \log N$; i.e. $S = O(N^{2/3})$, where the tilde notation implies that we have neglected factors polynomial in $\log N$.

When the function, $f$, is cheap to evaluate, but cannot be reverse engineered, the best method of finding a collision is to treat $f$ as a black box. In this case, the same performance as Ambainis’ algorithm can be achieved by a much simpler approach [16]. We divide the $N^2$ search space into $N^2 / S$ blocks and Grover search each one in time $O(\sqrt{N^2 / S})$ for a time–space trade-off of $ST^2 = \tilde{O}(N^2)$, which includes $S = T = \tilde{O}(N^{2/3})$ as a special case. Grover and Rudolph pose beating this trade-off as a challenge.

Our main result (theorem 4.4) allows us to meet this challenge. We choose a random set of $S$ inputs $x_1, \ldots, x_S$ to query, and store these along with $f(x_1), \ldots, f(x_S)$ in the distributed memory using a data structure that permits fast look-ups. Next, we use the $S$ processors to check the remaining $N - S$ inputs for a collision with one of the $x_1, \ldots, x_S$. Using Grover’s algorithm $S$ times in parallel (theorem 5.1) requires time $\tilde{O}(\sqrt{N / S})$. Note that each iteration of Grover requires the use of our algorithm for making simultaneous queries to a distributed memory.

This will succeed (with constant probability) in finding a collision if one of the colliding elements was in $\{x_1, \ldots, x_S\}$. Since this event happens with probability $\Omega(S/N)$ (as long as a collision exists at all), we can wrap this entire routine in amplitude amplification to boost the probability to two-thirds by repeating $O(\sqrt{N / S})$ times. The total time is $T = \tilde{O}(N/S)$, and so we have achieved a time–space trade-off of

$$ST = \tilde{O}(N).$$

Owing to the efficient circuit for parallel memory look-ups (theorem 4.4), the trade-off applies in the circuit model where space is the number of qubits (or width) and time is the circuit depth.

(b) Organization of the paper

The remainder of the paper is organized as follows. In §2, we provide an algorithm for efficiently moving quantum data (theorem 2.2) according to a classical permutation. Distinct indices are permuted, replacing $|x_1, \ldots, x_N\rangle$ with $|x_{j_1}, \ldots, x_{j_N}\rangle$. Armed with theorem 2.2, we relate the circuit model to the DQC model in §3. We prove that given (by an experimentalist) any layout of qubits grouped into memory sites with connections between fixed pairs of sites, we can efficiently implement algorithms presented in the circuit model. Of course, there is a price to pay: the overhead depends on the topology of the processors, but our algorithm is close to optimal.

In §4, we generalize the data-moving algorithm so that the destinations $j_1, \ldots, j_N$ are quantum data and no longer restricted to form a permutation. This results in an algorithm (circuit) that can look-up multiple memory entries in parallel. In addition, we prove that this memory look-up algorithm (measured in terms of circuit complexity) is scarcely more expensive than any
circuit capable of accurately accessing even a single entry. Thus, the Q PRAM model can be efficiently simulated by the circuit model that, in turn, can be simulated using a distributed quantum computer. We consider, in §5, various quantum search problems in the Q PRAM model. The application of our efficient algorithm for parallel memory access results in several improved quantum algorithms: the multi-Grover algorithm for parallel search of an unstructured database; an improved algorithm for the element distinctness problem and an improved algorithm for the collision finding problem.

2. Moving quantum data

The quantum state stored in a qubit can be moved across a graph, $G$, using a chain of SWAP gates. Alternatively, we could apply a teleportation scheme making use of existing entanglement and classical control [17]. However, if in a single phase of computation, we wish to perform a permutation of many or even all of the qubits, this routing problem is non-trivial.

We consider the problem of moving $N$ pieces of quantum data, initially stored at the $N$ nodes of $G$. In this section, we provide an algorithm, $V_N$, that performs an arbitrary permutation of qubits. Our only requirement being that the host graph, $G$, supports a sorting network. We make two simplifying assumptions: that the destinations are known at the outset (as classical information) and that they are all different. In §4, we relax these assumptions and generalize our algorithm.

**Definition 2.1.** Given the quantum register $|x_1, x_2, \ldots, x_N\rangle$ with $|x_j\rangle$ located at node $j$ of the graph $G$, an algorithm for data moving is a unitary map implementing

$$V_N : |x_1, x_2, \ldots, x_N\rangle \mapsto |x_{j_1}, x_{j_2}, \ldots, x_{j_N}\rangle,$$

for some permutation $(1, \ldots, N) \mapsto (j_1, \ldots, j_N)$. The $N$ data registers $x_1, \ldots, x_N$ each consist of $d$ qubits.

We now present an efficient algorithm for implementing $V_N$. The result applies to any connected graph, and we give some interesting examples and their respective properties in table 2.

**Theorem 2.2.** Let $G$ be a connected graph with $N$ nodes, each with $d$-qubit data registers and $Q$-qubit ancilla space. Let $D_G$ be the depth of a reversible sorting network over $G$ with $N$ inputs. Then, there exists an algorithm for $V_N$, restricted to $G$, with depth $O(D_G)$.

Before proving theorem 2.2, we first describe the main subroutine for the data-moving algorithm and subsequent algorithm for parallel memory access: a classical reversible sorting network. Both the data-moving and parallel memory access algorithms are in fact classical reversible circuits. Indeed, the tools we use were developed originally for classical parallel computing [18–20]. We were unable to find our theorem 2.2 or 4.4 in the literature (although [18] is somewhat similar), and we suspect that may be because quantum computing presents architectural issues that are different to those encountered in the field of traditional classical parallel computing.

(a) Sorting over a graph

A sorting network is a network on $T$ wires in which each wire represents one of $T$ elements and where the only gates are binary comparators. A binary comparator takes two elements on its input, and it outputs the same two elements but in the correct order, according to some specified comparison routine. Crucially, a sorting network is independent of the inputs, the same fixed set of gates correctly sorts all inputs.

For any connected graph $G$ of $T$ vertices, there must exist a lowest-depth sorting network for sorting $T$ elements, all of whose comparators lie along edges of $G$. For example, when $T = 4$ and $G$ is a 4-cycle, then the bitonic sorting network fits the graph, and, having six comparators in
depth 3, is optimal. But for \( T = 4 \) and \( G \), a graph of three edges in a line, the bitonic sort is not possible and a sort involving six comparators in depth 4 turns out to be optimal for this graph. Examples of sorting networks for popular architectures are given in table 2.

To make a sorting network reversible, we add an additional sorting-ancilla bit to each comparator that is initially set to \( |0\rangle \). This sorting-ancilla gets flipped to \( |1\rangle \) if and only if the comparator exchanges the order of the two elements input. In more detail, the comparator first compares the two elements, storing the resulting bit in the sorting-ancilla; then conditioned on the sorting-ancilla being set, it swaps over the two elements. These two steps are each clearly reversible in their own right. The sorting-ancillas are then all retained, to enable ‘unsorting’ later.

A reversible comparator for full lexicographic sort on \( b \)-bit objects was constructed by Thapliyal et al. [21]. Their algorithm is based on the binary tree and is efficient for our purposes, having width \( O(b) \) and depth \( O(\log b) \).

Write \( s(T) \) for the total number of comparators appearing in a given sorting network for \( T \) elements. A sorting operation is written as

\[
S_T : |0\rangle \otimes \bigotimes_{k=1}^{T} |x_k\rangle \mapsto |\sigma\rangle \otimes \bigotimes_{k=1}^{T} |x_{\sigma(k)}\rangle. \tag{2.1}
\]

Here, \( \sigma \in \text{Sym}(T) \) denotes a permutation that puts the \( T \) elements in order, and the first register (storing \( \sigma \)) actually consists of the \( s(T) \) sorting-ancilla bits that get fed into the comparators. It is clear from this observation that \( s(T) \geq \log_2(T!) = \Omega(T \log T) \) for any valid sorting network, since potentially any element of Sym(T) might be a uniquely correct one. If a sorting network has depth \( D \), then it uses a total of \( O(D \cdot T) \) comparators, and so depth must be \( D = \Omega(\log T) \) for any valid sorting network.

Because the sorting subroutine is reversible, it makes sense to run it backwards. When that happens, the comparators are encountered in reverse order, and each comparator swaps the order of its inputs according to whether its sorting-ancilla bit is set. That sorting-ancilla bit is then reversibly cleared (regardless of its value) by ‘uncomputing’ the comparison between the two elements.

(b) The quantum data-moving algorithm

We now describe the quantum data-moving circuit, \( V_N \). It is convenient to break \( V_N \) into three basic parts: formatting, sorting and applying the permutation. Both the formatting and the sorting need to be reversed after the permutation has been applied. This can all be annotated as follows, reading right to left:

\[
V_N = \hat{F} \circ S^{-1}_{2N} \circ P \circ S_{2N} \circ F. \tag{2.2}
\]

We must be careful to ensure that all operations can be performed with the graph locality restriction and to count the depth and additional ancilla space required by each of these subroutines. The format and permutation will both be entirely local operations, so their circuits are already admissible to the processor model. The gates of the sorting subroutine all belong to comparators of the sorting network over the graph, \( G \).

(i) Initial formatting

The subroutine \( F \) can be achieved using Pauli X gates and SWAP gates with no additional ancillas and in depth 1. It is not really a ‘computation’ at all, rather a rearrangement of the input data into a format amenable for describing the sorting that will follow.
Let there be $2N$ ancilla registers, which we call packets. Each packet contains an address ($\lceil \log_2 N \rceil$ bits), a flag (1 bit) and a data ($d$ bits) register. The initial format moves the input data out of the input registers and into the packets, ready for sorting,

$$F : |x_1, x_2, \ldots, x_N\rangle \bigotimes_{i=1}^{N} |(0,0,0)\rangle_{2i} |(0,0,0)\rangle_{2i+1}$$

$$\mapsto |0,0,\ldots,0\rangle \bigotimes_{i=1}^{N} |(i,a,x_i)\rangle_{2i} |(j_i,q,0)\rangle_{2i+1},$$  \hspace{1cm} (2.3)

where $j_i \neq j_s$ for all $r \neq s$. The special symbols $q$ and $a$ denote ‘question’ and ‘answer’, respectively.

One can think of the map $F$ in the following terms. Each processor, $i$, submits the ‘answer’ packet $(i,a,x_i)$, and if the data it would like to obtain is at index $j_i$, it also submits the ‘question’ packet $(j_i,q,0)$. The question–answer flag is used in the sort and unsort steps with the convention that $q < a$.

(ii) Sorting

Sorting was described in §2a. In this context, we want to sort the $2N$ packets, using a lexicographical ordering that reads only the address then the flag of each packet. In accordance with equation (2.1), the subroutine $S_{2N}$ must employ a register of $s(2N)$ sorting-ancilla bits, mapping as

$$S_{2N} : |0\rangle \bigotimes_{i=1}^{N} |(i,a,x_i)\rangle_{2i} |(j_i,q,0)\rangle_{2i+1}$$

$$\mapsto |\sigma\rangle \bigotimes_{i=1}^{N} |(i,q,0)\rangle_{2i} |(i,a,x_i)\rangle_{2i+1},$$  \hspace{1cm} (2.4)

where $\sigma \in \text{Sym}(2N)$ is the permutation implied by $2i = \sigma(2i + 1)$ and $2i + 1 = \sigma(2j_i)$, for $i$ from 1 to $N$.

The total depth of the circuit for the unitary map $S_{2N}$ is equal to the depth of the sorting network multiplied by the depth of a single comparator.

(iii) Locally SWAP

After the sort, we are left with a sequence of packets of the form

$$\ldots (i-1, q, 0) \ (i-1, a, x_{i-1}) \ (i, q, 0) \ (i, a, x_i) \ (i+1, q, 0) \ (i+1, a, x_{i+1}) \ldots,$$  \hspace{1cm} (2.5)

where the packets are sorted in lexicographical order according to the address and the flag. This ordering means that each answer is immediately to the right of the corresponding question and, furthermore, both lie on the same node. Without the need for any auxiliary bits, and in depth 1, SWAP gates can achieve the map

$$P : \bigotimes_{i=1}^{N} |(i,q,0)\rangle_{2i} |(i,a,x_i)\rangle_{2i+1} \mapsto \bigotimes_{i=1}^{N} |(i,q,x_i)\rangle_{2i} |(i,a,0)\rangle_{2i+1}.$$  \hspace{1cm} (2.6)

An important property of $P$ is that it does not change the address or the flag used in the sort.
(iv) Unsorting

The sorting network can be run in reverse to return the packets to their original positions. To achieve the unsort, $S^{-1}_{2N}$ acts on the sorting-ancilla register and the packets, mapping

$$
S^{-1}_{2N}: |\sigma\rangle \otimes \bigotimes_{i=1}^{N} |(i, a, 0)\rangle_{2i} \mapsto |0\rangle \otimes \bigotimes_{i=1}^{N} |(i, a, 0)\rangle_{2i} \quad (2.7)
$$

Since $S^{-1}_{2N}$ is the same as the sorting network, but with the order of the gates reversed, the cost for the unsort step is the same as for step 2.

(v) Final formatting

The final step in the algorithm is to write the data back to the original registers and clear the ancilla space. As with the initial formatting, $F$, the subroutine $\hat{F}$ has depth 1. Acting on the same registers as for its counterpart $F$, it works as follows:

$$
\hat{F}: |0, 0, \ldots, 0\rangle \bigotimes_{i=1}^{N} |(i, a, 0)\rangle_{2i} \mapsto |x_{j_1}, x_{j_2}, \ldots, x_{j_n}\rangle \bigotimes_{i=1}^{N} |0, 0, 0\rangle_{2i} \quad (2.8)
$$

(c) Low valency graphs

We require $O(D_G)$ qubits at each node to record the which-way information of the two sorting steps in $V_N$. Hence, for very low valency graphs where the sorting depth is large, there is insufficient local space to run the algorithm. For example, in the one-dimensional nearest-neighbour graph, we would need each node to have $O(N)$ qubits. We now explain how to fix this problem.

The sorting step in the $V_N$ algorithm involves only the classical information about the permutation known at the start of the algorithm; it is independent of the data register. We can classically compute all of the moves required by the sorting network before running the quantum algorithm. Hence, we can replace the reversible comparators (with their additional storage qubit) by a SWAP gate or no gate, as required in that step of the sort. Since the network remains the same, the depth of the algorithm is $O(D_G)$, as before, but with $O(d)$ qubits per node. This is the approach taken by Hirata et al., who consider the case of the one-dimensional nearest-neighbour graph with one qubit per site [7].

3. Efficient distributed quantum computing

The circuit model allows any pair of qubits to be connected by gates and so allows arbitrarily long-range interactions. This model is very far from any likely implementation of a quantum computer. We imagine that a small number of long-range interactions could be possible, but most gates will need to be local. Owing to the presumed requirements of fault tolerance [22], we expect implementations will allow the concurrent execution of one- and two-qubit gates with fast classical control.

It is well known that if the qubits are laid out in a line and each could only connect with its nearest neighbours (one either side), then the resulting model of computation would still be universal because it could emulate any circuit of the more general kind. The emulation proceeds in a straightforward fashion using SWAP gates to bring qubits next to each other so
that nearest-neighbour gates can be applied. The price to pay in this emulation is (in general) an overhead factor of $O(W)$ for each gate, where $W$ counts the total width (number of qubits) of the circuit being emulated. For highly parallel algorithms, there are $O(W)$ gates per time step so the cost of this emulation scheme is $O(W^2)$. Hirata et al. [7] use a sorting algorithm to emulate all $O(W)$ gates simultaneously, resulting in an improved emulation scheme for the one-dimensional nearest-neighbour architecture with cost $O(W)$.

More generally, we could envisage having memory larger than a single qubit at each ‘site’, with connectivity more generous than simply being connected each to two neighbours in a line. Then, the overhead depth factor (or emulation factor) of embedding circuit operations into the underlying hardware could be reduced to something smaller than $O(W)$. In this section, we demonstrate that reversible sorting networks provide an elegant way of embedding an arbitrary circuit into physical hardware. Up to logarithmic factors, the proposed solution is asymptotically optimal. Our scheme is very general and includes the case of each site being a single qubit.

Consider a quantum circuit of width $W$, and let there be $N$ quantum processors, each with its own local memory of $Q$ qubits. Suppose that $Q \cdot N \geq W$ and that the processors are interconnected as the $N$ nodes of a graph $\mathcal{G}$. We say that a circuit respects graph locality if every two-qubit gate in the circuit has those two qubits lying either in the same processor’s local memory or in a neighbouring memory with respect to $\mathcal{G}$. In addition, we require that each gate is assigned to a processor that holds at least one of its qubits, and each processor can only perform one gate per time step. Together, these restrictions on the ordinary circuit model define the DQC model formally.

We examine the best overhead depth factor for arbitrary circuit emulation in the worst case. Starting with a circuit of $W$ qubits, we wish to emulate it using a circuit that respects graph locality. We want the overhead depth factor of this emulation to be as small as possible. That is, we wish to minimize the function

$$F(\mathcal{G}, Q, W) := \max_C \frac{\text{depth}(C')}{\text{depth}(C)},$$

where $C'$ is a circuit for emulating $C$ subject to the constraints imposed by the host graph $\mathcal{G}$ and the number of qubits at each site, $Q$. Maximization is over all circuits $C$ of width $W$, so $F(\mathcal{G}, Q, W)$ is the worst-case cost of emulating arbitrary circuits. Normally, we are concerned with the case $W = |\mathcal{G}| = N$, so that an emulation has one processor per qubit being emulated, and where $Q$ is large enough to hold ancilla for basic computations.

In §2, we used sorting networks to efficiently move quantum data. We now show that scheduling tasks on a distributed quantum computer can be regarded as (more or less) equivalent to the problem of designing sorting networks commensurate with those topologies. We put these ideas on a firmer footing in the following theorem.

**Theorem 3.1.** Let $\mathcal{G}$ be any connected graph on $N$ nodes, and consider the DQC model with graph $\mathcal{G}$ and $Q = O(\log N)$ qubits per processor. Let $D_\mathcal{G}$ denote the depth of the best algorithm for sorting $2N$ arbitrary bit strings of length $\lceil \log_2 N \rceil + 2$ over $\mathcal{G}$. Then, there exists an algorithm for emulating arbitrary circuits with depth overhead given by

$$F(\mathcal{G}, Q, N) = O(D_\mathcal{G}).$$

**Proof.** The proof of the theorem uses our algorithm for $V_N$ with data registers of size $d = 1$ (cf. §2b). The circuit for $V_N$ can be directly embedded into the distributed quantum computer. Hence, we show how a circuit of width $N$ can be emulated in terms of gates that are local (with respect to $\mathcal{G}$) and renditions of $V_N$ only.

Each processor is large enough to hold two packets and some ancillas, since $Q = O(\log N)$. Given a general circuit of width $N$, in each time slice, there will be at most $N$ gates. The gates are ‘assigned’ one per processor when the time slice is emulated. When a processor comes to emulate a gate assigned to it, it will need access to the one or two qubits of that gate. The emulation of a time slice, therefore, requires two calls to the subroutine $V_N$: without loss of generality, we can assume that the first qubit of a gate already resides at the processor to which that gate has been
assigned; the first call to \( V_N \) brings the second qubit of each gate to its processor; the processor implements the gate locally on the two qubits; the second call to \( V_N \) restores the second qubit of each gate to its original home. Null ancilla qubits can be included within each \( V_N \) operation in order to make it a permutation.

Every time slice of the circuit being emulated now additionally requires two calls to \( V_N \), plus appropriate \( O(\log N) \)-sized circuity (per processor) to write and erase the indices \( j_i \) used within \( V_N \). Overall, this costs an overhead depth factor of \( O(D_G) \).

In theorem 3.1, we provided an algorithm for embedding a circuit into a physical host graph. Our aim was to minimize the depth overhead factor \( F(G, Q, N) \) of emulating the worst-case circuit. In the following theorem, we prove that this algorithm is optimal up to logarithmic factors.

**Theorem 3.2.** For a distributed quantum computer with host graph \( G \) consisting of \( N \) nodes each with \( Q \) qubits, the cost of emulating arbitrary circuits of width \( O(N \log N) \) is bounded by

\[
F_{\min}(G, Q, O(N \log N)) = \Omega \left( \frac{D_G}{\log N \log \log N} \right).
\]

**Proof.** The Ajtai–Komlós–Szemerédi (AKS) sorting network sorts \( T \) elements in \( O(\log T) \) comparator depth [13,23]. Suppose we use this network to sort \( 2N \) packets of bit length \( \lceil \log_2 N \rceil + 2 \). Let \( C \) be the (unconstrained) circuit for achieving this, so the width of \( C \) is \( W = O(N \log N) \) and its depth is \( O(\log N \log \log N) \). The cost of emulating \( C \) is bounded by \( D_G \) since the emulation is a sorting algorithm on the \( (G, Q) \)-distributed computer, and we defined \( D_G \) to be the depth of the best sorting algorithm. Hence, the cost of any emulation scheme is lower bounded by \( \Omega(D_G / \log N \log \log N) \), as required.

A lower bound on the cost of implementing an addition circuit on a \( k \)-dimensional (\( kD \)) nearest-neighbour graph was found by Choi & Van Meter [24]. Their bound translates into an emulation cost of

\[
F_{\min}(G, Q, N) = \Omega \left( N^{1/k} \right) = \Omega(D_G),
\]

in the special case of emulating a circuit for addition on a \( kD \) nearest-neighbour graph.

(a) The hypercube architecture

Theorem 3.1 provides an efficient algorithm for emulating circuits on *any* graph. We briefly consider a particularly nice architecture, the hypercube, demonstrating that the addition of a few flying qubits greatly improves the efficiency of a distributed quantum computer.

All of the comparators in any given sorting network are identical, and when any \( T \) elements are input to the sorting network, its overall effect should be to output them in totally sorted order, with certainty. The network can of course be designed completely independently of the comparator, since the details of what makes one element ‘greater’, ‘less’ or ‘equal’ to another is irrelevant from the perspective of which binary comparisons are needed to guarantee sorting. Thus, for any value of \( T \), one can ask for the lowest-depth sorting network for sorting \( T \) elements.

In [13,23], it is shown that \( T \) elements can be sorted in \( O(\log T) \) depth of comparators, and that a uniform family of sorting networks achieves this. Unfortunately, the constant for Paterson’s simplified version of the AKS sorting network [23] is around 6100 and so not practical for any realistic sizes of \( T \). However, the bitonic sorting network [12] sorts \( T = 2^t \) elements in depth \( 1 + 2 + \cdots + t = t(t + 1)/2 = O(\log^2 T) \).

The following corollary of theorem 3.2 demonstrates a nice balance between the cost of embedding arbitrary circuits into the graph, the size of the nodes, \( Q \), and the degree, \( \text{val}(G) \).

**Corollary 3.3.** A distributed quantum computer with \( N \) nodes of \( Q = O(\log N) \) qubits interconnected using a hypercube architecture has the following properties: the number of connections per node is \( \text{val}(G) = O(\log N) \), and yet the overhead depth factor is \( F(G, Q, N) = O(\log^2 N \log \log N) \) for emulating quantum circuits of width \( N \).
Ignoring constants, the AKS graph would be a better one to use, as it has an overhead depth factor $F(G, Q, N) = O(\log N \log \log N)$, the AKS graph would be a better one to use. Hence, asymptotically, the cost of emulating the circuit model on a distributed quantum computer with low degree is $O(\log N)$. Alternatively, there are sorting algorithms with an $O(\log N)$ complexity but which are probabilistic and so solve the routing problem on almost all circuits. However, such a probabilistic approach will not work if we require all permutations to be correctly routed. This is the case in the next section when we consider parallel memory access.

4. The cost of accessing quantum memory

In §2, we considered moving quantum data according to a permutation known at compile time. By using the permutation needed to embed a circuit into a host graph, this approach led to an efficient algorithm for DQC in §3. The algorithm for data moving can be easily modified to perform a permutation that is stored as quantum data, $|j_1, \ldots, j_N\rangle$. We simply amend the formatting steps of the algorithm, using SWAP rather than $X$ gates,

$$F : |j_1, \ldots, j_N\rangle|x_1, x_2, \ldots, x_N\rangle \mapsto |j_1, \ldots, j_N\rangle|x_1, x_2, \ldots, x_N\rangle \mapsto \frac{1}{\sqrt{2}}(|0, 0, 0\rangle + |(i, q, 0)\rangle)_{2i+1} |(i, a, x_i)\rangle_{2i}.$$ (4.1)

Such an algorithm would allow us to move data in a superposition over permutations. In this section, we generalize the idea further to the case where the indices do not necessarily form a permutation, i.e. the indices need not be distinct. This new algorithm allows efficient parallel access to quantum memory and leads to the new quantum algorithms presented in §5.

(a) The cost of a single look-up

Often when quantum algorithms are quoted in the query model, the concept of an oracle is used to abstract away those logical unitaries that are intended to make ‘random’ access to (quantum) memory. We start by examining the idea of accessing memory in more detail, without using oracles.

**Definition 4.1.** A logical unitary for accessing a single piece of data is a map $U_{(1,N)}$ that implements

$$U_{(1,N)} : |j\rangle|y\rangle|x_1, x_2, \ldots, x_N\rangle \mapsto |j\rangle|y \oplus x_j\rangle|x_1, x_2, \ldots, x_N\rangle,$$ (4.2)

where $\oplus$ denotes bitwise addition.

Here, we have depicted $2 + N$ registers. The first register (index register) holds an index large enough to ‘point’ to any one of $N$ data registers; its associated Hilbert space must be of dimension at least $N$. The second register is called the target register and holds the same kind of data as a data register. The other $N$ registers are data registers and could in principle be of any (equal) size.

We can derive a simple lower bound for the cost of accessing a single piece of memory based on two simple constraints on any circuit for $U_{(1,N)}$. The memory must hold the entire database and so within the circuit, there must be a causal chain from every data register to the target register. This observation has been made before [25], but we give it here to motivate the cost of our parallel look-up algorithm that follows.

**Theorem 4.2.** In the circuit model, if a circuit implements $U_{(1,N)}$ on $N$ data registers each consisting of $d$ qubits, then its width is $\Omega(dN)$ and its depth is $\Omega(d \log N)$.

There is a sense in which the gates in a typical circuit for $U_{(1,N)}$ can be said to be ‘not working very hard’ (although this idea is hard to quantify precisely), and this inefficiency points to the need for a parallel algorithm.
(b) The cost of parallel look-ups

Definition 4.3. A logical unitary for accessing \( N \) pieces of data is a map \( U_{(N,N)} \) that implements

\[
U_{(N,N)} : |j_1,j_2,\ldots,j_N\rangle|y_1,y_2,\ldots,y_N\rangle|x_1,x_2,\ldots,x_N\rangle
\rightarrow |j_1,j_2,\ldots,j_N\rangle|y_1 \oplus x_{j_1},y_2 \oplus x_{j_2},\ldots,y_N \oplus x_{j_N}\rangle|x_1,x_2,\ldots,x_N\rangle. \tag{4.3}
\]

Here, we have depicted \( N \) index registers, \( N \) target registers and \( N \) data registers, comprising a total of \( 3N \) input registers. As before, the index registers are each made up of \( \lceil \log_2 N \rceil \) qubits, while the target registers and data registers are each made up of \( d \) qubits.

We define the Q PRAM model as the circuit model plus the ability to access quantum memory in a parallel and unrestricted way. In any one time step, we can apply up to \( N \) input registers. As before, the index registers are each made up of only a logarithmic cost in the width and the depth. The input value, \( UN \), while the target registers and data registers are each made up of \( d \) qubits.

The initial formatting step is achieved in one time step. The map \( U_{(N,N)} \) is presented in \( \S4 \). The calculation of the width and depth can be found in the electronic supplementary material, appendix A.

Theorems 4.2 and 4.4 also tell us that in the circuit model, for parallel memory look-ups, we can achieve a factor \( N \) more ‘effect’ for only a small additional ‘effort’. This will be seen to have radical effects on certain ‘memory-intensive’ algorithms in \( \S5 \).

(c) The parallel look-up algorithm

We now present the algorithm for accessing memory in parallel, implementing the unitary \( U_{(N,N)} \) defined in equation (4.3). We will generalize the algorithm given for \( V_N \) to show how \( U_{(N,N)} \) may be efficiently implemented. We first construct packets that include the original data and a flag to be used in the sorting step. After a sorting network is applied, we transform the data. Instead of the permutation used in the data-moving algorithm, the transformation to use is composed of cascading, \( B \), copying, \( C \), and uncascading, \( B^{-1} \). Finally, we reverse the sort and map the data back to the original registers. Accordingly, the parallel look-up algorithm can be written as

\[ U_{(N,N)} = F \circ S_{2N}^{-1} \circ B^{-1} \circ C \circ B \circ S_{2N} \circ F. \tag{4.4} \]

(i) Initial formatting

For the parallel look-up algorithm, we use packets containing four items. Each packet contains an address and flag as before, but now we have two data registers of \( d \) bits; target data \( y_i \) and memory data \( x_i \). The initial formatting stage, \( F \), resembles the same step in equation (2.3), but where packet locations are initially stored as quantum data,

\[
F : |j_1,\ldots,j_N\rangle|y_1,\ldots,y_N\rangle|x_1,\ldots,x_N\rangle \bigotimes_{i=1}^{N} |(0,0,0,0)_{2i} ||(0,0,0,0)_{2i+1} \rightarrow |0,\ldots,0\rangle|0,\ldots,0\rangle|0,\ldots,0\rangle \bigotimes_{i=1}^{N} |(i,a,0,x_i)_{2i} ||(j,b,y_j,0)_{2i+1}. \tag{4.5}
\]

The map \( F \) moves the data into the packets, where it can be processed by the rest of the algorithm. The initial formatting step is achieved in one time step.
(ii) Sorting

The sorting step is the same as before: we sort lexicographically reading only the address and flag of each packet,

\[ S_{2N} : |0\rangle \otimes_{k=1}^{2N} (i_k, f_k, y_k, x_k) \mapsto |\sigma\rangle \otimes_{k=1}^{2N} (i_{\sigma(k)}, f_{\sigma(k)}, y_{\sigma(k)}, x_{\sigma(k)}) \]. (4.6)

At the end of the sort, we are left with a sequence of the form

\[ ... (i, q, y_*, 0) \ldots (i, q, y_*, 0) (i, a, 0, x_i) \ldots, \] (4.7)

where asterisks (\(*\)) denote the various processors that have queried processor \(i\).

(iii) Cascade

The goal of cascade is to send a copy of the data \(x_i\) into the empty data registers of packets on the left (cf. (4.7)). This can be done by performing controlled NOTs (CNOTs) first to packets at distance 1, then distance 2, 4, 8, etc. The CNOTs operate on the fourth site of each packet and are controlled to only act if the source and target packet both have the same value of \(j\). Since there is no way of knowing in advance how far the data will need to propagate, we need a method that works in all cases. For example, it could be the case that every processor requests data from a single processor, say \(j_1 = j_2 = \ldots = j_N = 1\). The cascade can be achieved reversibly in \(\lceil \log_2(2N) \rceil\) steps, the details of which can be found in the electronic supplementary material, appendix B.

(iv) Copying

\(C\) is a simple depth 1 local operation. Every packet simply CNOTs the contents of its memory data into its target data. This has the effect of mapping every \(|(j_i, q, y_i, x_i)\rangle\) to \(|(j_i, q, y_i \oplus x_i, x_i)\rangle\), while every \(|(i, a, 0, x_i)\rangle\) gets mapped to \(|(i, a, x_i, x_i)\rangle\).

(v) Reversing the cascade

This map reverses the effect of the cascade, cleaning up all the aux-phase ancillas, making the packets ready for unsorting. The total effect of \((B^{-1} \circ C \circ B)\) is to replace every instance of \(|(j_i, q, y_i, 0)\rangle\) with \(|(j_i, q, y_i \oplus x_i, 0)\rangle\), while replacing \(|(i, a, 0, x_i)\rangle\) with \(|(i, a, x_i, x_i)\rangle\) for all \(i\). This action does not change the ordering of the packets, which only depends on the address and flag of each packet. Therefore, the action is compatible with the sorting stages, as required.

(vi) Unsort

The unitary \(S_{2N}^{-1}\) unsorts just as before.

(vii) Final formatting

The final step is to apply a formatting map, \(\hat{F}\), which works as follows:

\[ |0, \ldots, 0\rangle |0, \ldots, 0\rangle |0, \ldots, 0\rangle \otimes_{i=1}^{N} (i_i, a_i, x_i, x_i)_{2i} |(j_i, q_i, y_i \oplus x_j, 0)\rangle_{2i+1} \mapsto |j_1, \ldots, j_N\rangle |y_1 \oplus x_j, \ldots, y_N \oplus x_j\rangle |x_1, \ldots, x_N\rangle \otimes_{i=1}^{N} (0, 0, 0, 0)_{2i} |(0, 0, 0, 0)\rangle_{2i+1}. \] (4.8)

Note that this is a depth 2 map rather than a depth 1 map because each \(x_i\) appears in two places, and these need to ‘relocalize’ as well as ‘move’.
5. Revisiting popular algorithms

Grover’s quantum search algorithm [14] and the generalization to amplitude amplification [28, 29] have the great advantage of being widely applicable. Problems such as element distinctness [15, 30], collision finding [31], triangle finding [32, 33], 3-SAT [34] and NP-hard tree search problems [35] all have faster quantum algorithms because they are able to make use of amplitude amplification. In this section, we revisit Grover’s quantum search algorithm and the element distinctness problem, resolving a challenge posed by Grover & Rudolph [16].

The theorems presented in this section use the circuit model so that they can be easily compared with existing literature. However, we developed the ideas using the Q PRAM model and indeed, we borrow some of the terminology in the theorem proofs. The results in this section follow from applying our main result (theorem 4.4) to existing quantum algorithms by Grover [14] and Buhrman et al. [30].

(a) A single quantum search of an unstructured database

Grover’s fast quantum search algorithm [14] is usually presented as solving an oracle problem: let \( f : \{1, \ldots, N\} \rightarrow \{0, 1\} \) be a function and suppose we wish to find solutions, \( s \), such that \( f(s) = 1 \). We are given an oracle with the ability to recognize solutions in the form of a unitary \( U_f : |x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle \). (5.1)

Setting the target register of \( U_f \) to the state \((|0\rangle - |1\rangle)/\sqrt{2}\) encodes the value of \( f(x) \) into a phase shift \( U_f : |x\rangle \mapsto (-1)^{f(x)}|x\rangle \), (5.2)

where we have suppressed the target register since it remains unchanged. Grover’s algorithm then makes \( \Theta(\sqrt{N/M}) \) calls to \( U_f \), and with probability \( 1 - O(M/N) \) outputs one of the \( M \) possible solutions uniformly at random.

Grover’s algorithm can be applied to search an unstructured database. We construct the oracle by first looking-up a database entry, and then testing if this entry is a solution to the function \( \alpha \). The initial state of the computer is \( |j\rangle|0\rangle|y\rangle|x_1, \ldots, x_N\rangle \), (5.4) where the state \( |j\rangle \) is the index of the database item we will look-up, \( |0\rangle \) is used to load the memory and \( |y\rangle \) is the target register.

An oracle is constructed by first looking-up a database entry, and then testing if this entry is a solution to the function \( \alpha \). The initial state of the computer is

\[
|j\rangle|0\rangle|y\rangle|x_1, \ldots, x_N\rangle,
\]

where the state \( |j\rangle \) is the index of the database item we will look-up, \( |0\rangle \) is used to load the memory and \( |y\rangle \) is the target register.

First, we apply the single memory look-up unitary, \( U_{(1,N)} \) (definition 4.1) so that the computer is in the state

\[
|j\rangle|x_j\rangle|y\rangle|x_1, \ldots, x_N\rangle,
\]

then calling the function \( \alpha \), using the corresponding unitary \( U_{\alpha} \), maps the state to

\[
|j\rangle|x_j\rangle|y \oplus \alpha(x_j)\rangle|x_1, \ldots, x_N\rangle.
\]
Finally, we restore the auxiliary state used to load the database item by applying $U^{i}_{(1,N)} = U_{(1,N)}$.

The final state of the computation is therefore

$$|j⟩|0⟩|y⟩\oplus α(x_j)⟩|x_1, \ldots, x_N⟩. \tag{5.7}$$

Hence, the unitary

$$O_a = U_{(1,N)} \circ U_a \circ U_{(1,N)} \tag{5.8}$$

can be used as an oracle for Grover’s algorithm. Setting the target register $|y⟩ = (|0⟩ - |1⟩)/√2$ encodes the value of $α(x_j)$ into a phase shift

$$O_a : |j⟩ \mapsto (-1)^{α(x_j)}|j⟩. \tag{5.9}$$

From theorem 4.2, the quantum circuit implementing Grover’s algorithm with $O_a$ as an oracle requires circuit width $\Omega(Nd)$ and depth $\Omega(\sqrt{N}/M(\log N + D_a))$ to find one solution with high probability. Here, $D_a$ is the depth of the circuit for $U_a$.

In many cases of interest (cf. §5c), we actually want to search over pairs (or more generally $r$-tuples) of database elements. We thus generalize the preceding algorithm by considering a function $α : X^r \to \{0, 1\}$ that takes as input an $r$-tuple of database entries. We wish to find solutions $s = (s^1, \ldots, s^r) \in \{1, \ldots, N\}^r$ such that $α(x_{s^1}, \ldots, x_{s^r}) = 1$. As with the case $r = 1$, we construct an oracle $O_a$ for Grover’s algorithm using $U_{(1,N)}$ and the unitary

$$U_a : |x_j⟩|b⟩ \mapsto |x_j⟩|b⟩\oplus α(x_j), \tag{5.10}$$

where $j = (j^1, \ldots, j^r) \in \{1, \ldots, N\}^r$ and $x_j = (x_{j^1}, \ldots, x_{j^r}) \in X^r$. The construction of the oracle runs just as in the previous case, but with the first register labelled with the multi-index $j$, while the second register contains $r$-tuples of database entries $x_j$. The initial state is

$$|j⟩|0⟩|0⟩|x_1, \ldots, x_N⟩. \tag{5.11}$$

Applying the unitary

$$O_a = U^{j}_{(1,N)} \circ U_a \circ U^{j}_{(1,N)}, \tag{5.12}$$

and setting the target register $|b⟩ = (|0⟩ - |1⟩)/√2$, we obtain the oracle

$$O_a : |j⟩ \mapsto (-1)^{α(x_j)}|j⟩. \tag{5.13}$$

The quantum circuit implementing Grover’s algorithm with $O_a$ as an oracle makes $Θ(\sqrt{N}/M)$ calls to $O_a$, where $M$ is the number of solutions. From theorem 4.2, the circuit width and depth are $Ω(Nd)$ and $Ω(\sqrt{N}/M(r \log N + D_a))$, respectively.

(b) Parallel quantum search of an unstructured database

Now that we have an efficient algorithm for performing parallel memory look-ups, we consider the effect of using the unitary $U_{(N,N)}$ together with (up to) $N$ functions as oracles for Grover’s algorithm. We show how multiple quantum processors can each run a Grover search on a single database in parallel. Since the first step of each instance of Grover’s algorithm is to query the entire database, it may come as a surprise that we do not encounter (unwanted) interference. It is the efficiency of our parallel memory look-up algorithm that allows us to interleave the Grover steps with the parallel database access.

Suppose we have (up to) $N$ functions that take $r$ database elements as inputs, $α_i : X^r \to \{0, 1\}$, for $i = 1, \ldots, N$. We wish to find solutions $s_i = (s^1_i, \ldots, s^r_i) \in \{1, \ldots, N\}^r$, such that $α_i(x_{s^1_i}, \ldots, x_{s^r_i}) = 1$ for all $i = 1, \ldots, N$. In exact analogy to the previous section, we assume that we are given circuits for the unitaries

$$U_{a_i} : |x_j⟩|b⟩ \mapsto |x_j⟩|b⟩\oplus α_i(x_j), \tag{5.14}$$

where again we write $j = (j^1, \ldots, j^r) \in \{1, \ldots, N\}^r$ and $x_j = (x_{j^1}, \ldots, x_{j^r}) \in X^r$. We shall assume that the depth and width of the circuits for $U_{a_i}$ are polynomial functions of $\log N, d$ and $r$. In effect, we are interested in functions that are easy to compute, but we do not know how to invert.
Thus, the best method for finding a solution is to treat the function as a black box. Each node has size $d$, which is $\Omega(\log N)$ since it must be large enough to perform basic operations, hold the database items and perform function evaluations. Thus, we assume that $d$ is a polynomial in $\log N$.

With these assumptions, the following theorem provides an algorithm that finds a solution for each function $\alpha_i$, using the same size quantum circuit (up to a factor polynomial in $\log N$ and $r$) required to find only one solution using the unitary $U_{(1,N)}$ to form an oracle (see §5a).

**Theorem 5.1 (Multi-Grover search algorithm).** Using the notation defined above, there is a quantum algorithm that, for each $i = 1, \ldots, N$, either returns the solution index $s_i \in \{1, \ldots, N\}$ such that $\alpha_i(x_{s_i}) = 1$, or, if there is no such solution, returns ‘no solution’. The algorithm succeeds with probability $\Theta(1)$ and can be implemented using a quantum circuit with width $\tilde{O}(N)$ and depth $\tilde{O}(\sqrt{N^2/M})$.

The value $M$ in the statement of theorem 5.1 is the minimum over $i \in \{1, \ldots, N\}$ of the number of solutions $s_i$ to $\alpha_i(x_{s_i}) = 1$. The notation $\tilde{O}$ means that we are neglecting a multiplicative factor that is polynomial in $r \log N$.

**Proof.** The proof is given in the electronic supplementary material, appendix C.

If $X$ were highly structured (such as being the numbers 1 to $N$), it would be straightforward to perform $N$ Grover searches in parallel, using a circuit of width $\tilde{O}(N)$, since we would not need to store $X$ explicitly. Now, making use of the efficient parallel memory look-up algorithm to access $X$, we are able to interlace the steps in the Grover algorithm with database look-ups. The end result is that we can indeed perform $N$ Grover searches in parallel, regardless of the structuring of $X$. We now examine the effect of theorem 4.4 on other memory-intensive quantum algorithms.

(c) **Element distinctness**

In this section, we present a quantum algorithm for the element distinctness problem: given a function $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$, determine whether there exist distinct $i, j \in \{0, 1\}^n$ with $f(i) = f(j)$. It is based on the algorithm of Buhrman et al. [30], but improved by making use of our efficient algorithm for parallel memory look-ups (theorem 4.4).

The size of the problem is parametrized by $N = 2^n$. Let $S$ and $T$ denote the available memory and time, respectively. Classically, we can search for a solution to the element distinctness problem using $O(N)$ function calls provided $ST = O(N^2)$. Applying Grover’s algorithm to this classical search fails to achieve an improvement in the number of function queries: there are $\Theta(N^2)$ pairs to search, so Grover’s algorithm requires $\Theta(N)$ queries. Ambainis [15] discovered an elegant quantum algorithm that improves upon this, having query complexity $O(N^{2/3})$. In fact, $\Omega(N^{2/3})$ queries is also a lower bound, so Ambainis’s algorithm is known to be optimal from this perspective. However, Ambainis requires memory proportional to $\tilde{O}(N^{2/3})$, i.e. $S = T = \tilde{O}(N^{2/3})$.

If we think of $f$ as something that is computed by a small (say poly log $N$ size) circuit, then the same performance as Ambainis’s algorithm can be achieved by a much more naıve approach [16]. We simply divide the $N^2$ search space into $N^2/S$ blocks and Grover search each one in time $\tilde{O}(\sqrt{N^2/S})$ for a time-space trade-off of $ST^2 = \tilde{O}(N^2)$, which includes both Ambainis’s special case $S = T = \tilde{O}(N^{2/3})$ and the naive Grover search $S = O(\text{poly log } N), T = \tilde{O}(N)$.

We now show how to improve on these results in the DQC model answering a challenge posed by Grover & Rudolph [16]. We are interested in the case where $f$ can be cheaply evaluated, but not reverse engineered (so that our best method of finding a collision is to treat $f$ as a black box).

**Theorem 5.2.** Given a poly log $N$ size circuit for computing the function $f$, we can solve the element distinctness problem in space $\tilde{O}(S)$ and time $T = \tilde{O}(N/S)$ for any $S \leq N$. This results in the space–time trade-off

$$ST = \tilde{O}(N).$$
Proof. First choose a random set of $S$ inputs $x_1, \ldots, x_S$ to query and store these along with $L = \{f(x_1), \ldots, f(x_S)\}$ in the distributed memory. That is, processor $j$ stores the pair $(x_j, f(x_j))$. Next, we use the $S$ processors to check the remaining $N - S$ inputs for a collision with one of the $x_1, \ldots, x_S$.

Using the AKS sorting network, the $S$ processors can sort $L$ in time $O(\log N)$. It is now easy to check if $L$ contains any pair of elements that are equal, $f(x_i) = f(x_j)$. If we find such a pair, then output $(i, j)$. Otherwise, use the sorted list, $L$, to construct a function, $g_L : \{1, \ldots, N - S\} \rightarrow \{0, 1\}$, defined by

$$g_L(i) = \begin{cases} 1 & \text{if there exists an element } j \in \{1, \ldots, S\} \text{ such that } f(x_i) = f(x_j), \\ 0 & \text{otherwise.} \end{cases}$$ (5.15)

Now we can split the remaining $N - S$ inputs into $S$ collections of size $O(N/S)$ and use the multi-Grover algorithm to perform $S$ parallel searches, each with the function $g_L$ as an oracle. This procedure requires $\Theta(\sqrt{N}/S)$ calls to $g_L$. The cost of calling the function $g_L$ in parallel is $O(\log S \log N \log \log N)$ using a binary search combined with our algorithm $\mathcal{U}_{(S,S)}$ for making simultaneous queries to a distributed memory. Hence, the total depth is $\tilde{O}(\sqrt{N}/S)$.

We will succeed (with constant probability) in finding a collision if one of the colliding elements was in $\{x_1, \ldots, x_S\}$. Since this event happens with probability $\Theta(S/N)$, we can wrap the entire routine in amplitude amplification to boost the probability to $\Theta(1)$ by repeating $\Theta(\sqrt{N}/S)$ times. Hence, the total run time is $T = \tilde{O}(N/S)$, and so we have achieved a time–space trade-off of $ST = \tilde{O}(N)$.

(d) Collision finding problem

Our results also apply to the collision finding problem: in which $f : \{N\} \rightarrow \{N\}$ is an efficiently computable function with the promise of being either $1$–$1$ or $2$–$1$, for which an $ST = O(N^{2/3})$ query algorithm is given in [31]. This problem may be solved with $ST = \tilde{O}((N)$, either by selecting $O(\sqrt{N})$ random elements and solving element distinctness, or by simply using the algorithm of [31] directly, augmented by using $S$ processors with shared memory together with our look-up algorithm.\footnote{Bernstein [25] discusses time–space trade-offs for a related problem in other computational models, including a two-dimensional nearest-neighbour architecture and a variant of the quantum RAM model that he refers to as ‘a naive model of communication’. Bernstein [25] also uses a distributed sorting algorithm to parallelize collision finding, which could be seen as a special case of our main result.} So we perform $S$ Grover searches in parallel on spaces of size $N/S^2$ instead of a single search on a space of size $N/S$.

6. Conclusion and discussion

In classical parallel computing, sorting networks provide an elegant solution to the routing problem and simulation of the parallel RAM model. In this paper, we have demonstrated that they can be applied to quantum computing too.

The information about the connectivity of a quantum circuit is available before we run the algorithm (at compile time). Using this classical information, we have designed an efficient scheme for routing quantum packets. The application of this data-moving algorithm is to DQC. We provide an efficient way of mapping arbitrary unconstrained circuits to limited circuits respecting the locality of a graph.

Our results already apply to nearest-neighbour architectures when the circuit is highly parallel. The case of emulating a circuit with many concurrent operations on a one-dimensional nearest-neighbour machine was covered by Hirata et al. [7]. The approach is to use the insertion/bubble sort to perform all of the operations in $O(N)$ time steps, which compares favourably to performing...
each gate in turn in $O(N^2)$ depth. We put this idea in a general framework applying to any (connected) graph. Along the way, we are able to prove that up to polylogarithmic factors, this approach is optimal.

We have shown how the addition of a few long-range (or flying) qubits dramatically increases the power of a distributed quantum computer. Using only $O(\log N)$ connections per node enables efficient sorting over the hypercube. A distributed quantum computer with nodes connected according to the hypercube graph would be able to emulate arbitrary quantum circuits with only $O(\log^2 N)$ overhead. One might expect that a quantum computer requires $O(N)$ connections per node so that each qubit can potentially interact with any other qubit. Our result demonstrates that this is not the case: for a small overhead, $O(\log N)$ connections suffice.

Theorem 3.2 provides a lower bound on the cost of emulating arbitrary circuits on a distributed quantum computer, which we use to argue that our algorithm is optimal up to logarithmic factors. The theorem comes with various assumptions that, if avoided, might reduce the emulation cost, the most obvious being that we want to emulate all possible circuits on our DQC. Any specific circuit could potentially be redesigned to respect the locality of the underlying architecture. In addition, one could try to optimize the choice of which physical qubit corresponds to which qubit in the circuit, called the qubit placement problem [37]. There are many other ways a quantum circuit could be optimized, as discussed in the study of Fowler & Devitt [38] and Maslov & Saeedi [39], for example.

We have presented a new algorithm for accessing quantum memory in parallel. The algorithm is a modification of the data-moving algorithm used in §§2 and 3, but where the destinations are quantum data and no longer restricted to form a permutation. The algorithm is extremely efficient; it has an overhead that is scarcely larger than any algorithm capable of accessing even a single entry from memory. Theorem 4.2 implies that $N$ processors can have unrestricted access to a shared quantum memory. It tells us that the Q PRAM and the circuit models are equivalent up to logarithmic factors.

Finally, we demonstrated that the parallel look-up algorithm can be used to optimize existing quantum algorithms. We provided an extension of Grover’s algorithm that efficiently performs multiple simultaneous searches over a physical database, and answered an open problem posed by Grover and Rudolph by demonstrating an improved space–time trade-off for the element distinctness problem. It seems likely that this framework for efficient communication in parallel quantum computing will be a useful subroutine in other memory-intensive quantum algorithms, such as triangle finding, or more generally for frameworks such as learning graphs.

The authors thank the Heilbronn Institute for Mathematical Research for hosting the discussions that led to this research. We are grateful to Frederic Magniez and Tony Short for catching bugs in a previous version, and an anonymous referee for highlighting relevant references. A.W.H. was funded by NSF grant nos 0916400, 0829937, 0803478, DARPA QuEST contract FA9550-09-1-0044 and IARPA via DoI NBC contract no. D11PC20167. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC or the US Government.

References


38. Fowler A, Devitt S. A bridge to lower overhead quantum computation. (http://arxiv.org/abs/1209.0510)