1-2003

Can Quantum Computers Be Useful When There are Not Yet Enough Qubits?

Luc Longpre
The University of Texas at El Paso, longpre@utep.edu

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-03-06
Published in *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 2003, Vol. 79, pp. 164-169.

Can Quantum Computers Be Useful When There Are Not Yet Enough Qubits?

Luc Longpré and Vladik Kreinovich

Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
{longpre,vladik}@cs.utep.edu

Formulation of the problem. Quantum computers (see, e.g., [5]) have a potential of speeding up computations in many important problems. For example:

- By using Grover's algorithm [3, 4], for any given error probability e_0, we can search for an item in an n-item (un-sorted) database in $\Theta(\sqrt{n})$ steps (where checking that a given element is the desired one is counted as a single step). By contrast, the best possible probabilistic algorithm that locates an item with an error probability $\leq e_0$ requires $\Theta(n)$ steps.
- By using Shor's algorithm [6], we can factorize integers in polynomial time instead of the best known exponential time for non-quantum deterministic algorithm (or even non-quantum probabilistic algorithms with any reasonable probability).

In view of the great potential for computation speedup, engineers and physicists are actively working on the design of actual quantum computers. There already exist working prototypes. However, at present, these computers can only solve trivial instances of the above problems, instances that have already been efficiently solved by non-quantum computers. Main reason: the existing quantum computers have only a few qubits, while known quantum algorithms require a lot of qubits; for example:

- Grover's algorithm requires a register with $q = \log(n)$ qubits for a search in a database of n elements;
- Shor's algorithm requires $O(\log(n))$ qubits to factor an integer n, etc.

Of course, while we only have 2 or 3 or 4 qubits, we cannot do much. However, due to the active research and development in quantum computer hardware, we will (hopefully) have computers with larger number of qubits reasonably soon.

A *natural question* is: while we are still waiting for the qubit register size that is necessary to implement the existing quantum computing algorithms (and thus, to achieve the theoretically possible speedup), can we somehow utilize the registers of smaller size to achieve a partial speed up?

In this paper, we start answering this question by showing the following: for quantum search, even when we do not have enough qubits, we can still get a partial speedup. The fact that we do

get a partial speedup for quantum search makes us hope that even when we do not have all the qubits, we can still get a partial speedup for other quantum computing algorithms as well.

We can put the above question in a more general perspective of *resource-bounded* computations. In classical computational complexity, algorithms are typically analyzed in terms of their time and space requirements. In many occasions, there is a space-time trade-off, meaning that, for example, if space is limited, the algorithm can still work, given more time resource. We consider the number of qubits in a quantum computation as an additional resource, and quantum algorithms efficiency should therefore be evaluated in terms of time, space, and number of qubits. Now, an interesting question is to consider whether an algorithm can trade-off some quantum qubits for time and space. We show below that Grover's algorithm offers such a trade-off with respect to qubits and time. We do not see how to implement any non-trivial trade-off in other well-known quantum algorithm, and so we offer this as an interesting open problem.

Grover's algorithm: a simple result. Let us assume that we are interested in searching in an unsorted database of n elements, and that instead of all $\log(n)$ qubits that are necessary for Grover's algorithm, we only have, say 90% or 50% of them. To be more precise, we only have a register consisting of $r = \alpha \cdot \log(n)$ qubits, where $0 < \alpha < 1$. How can we use this register to speed up the search?

Grover's algorithm enables us to use a register with r qubits to search in a database of $N = 2^r$ elements in time $C \cdot \sqrt{N}$. For our available register, $r = \alpha \cdot \log(n)$, hence $N = 2^r = n^\alpha$, so we can use Grover's algorithm with this qubit register to search in a database of size n^α in time $C \cdot \sqrt{N} = C \cdot n^{\alpha/2}$.

To search in the original database of size n, we can do the following:

- divide this original database into $n^{1-\alpha}$ pieces of size n^α; and then
- consequently apply Grover's algorithm with a given qubit register to look for the desired element in each piece.

Searching each piece requires $C \cdot n^{\alpha/2}$ steps, so the sequential search in all $n^{1-\alpha}$ pieces requires time $n^{1-\alpha} \cdot (C \cdot n^{\alpha/2}) = C \cdot n^{1-\alpha/2}$. Since $\alpha > 0$, we get a speedup.

Comment. In Grover's algorithm, we have a database of n elements, and we search for an element that has a certain (easy-to-test) property. At the end of Grover's algorithm, we apply measurement to the resulting quantum state to get an element (one of the original n elements), and then we check whether this element satisfies the desired property. If it does, we have solved the search problem; if it does not, we claim that there is no element with this property in the given database. There is a probability of an error: when an element actually exists, but the algorithm does not find it. Grover's algorithm returns the correct answer with a probability $\geq p_0$; the threshold probability $p_0 \approx 1$ must be fixed from the very beginning, as one of the parameters of this algorithm.

When we divide the original list into $n^{1-\alpha}$ pieces and use the same probability of error $P(er \,|\, In_k) \leq 1 - p_0$ for search in each piece k, then the probability of an overall error can be estimated as $P(er) \leq P(In_1) \cdot P(er \,|\, In_1) + \ldots P(In_k) \cdot P(er \,|\, In_k) + \ldots$, where $P(In_k)$ is the probability that the first element with the desired property is in k-th piece. Since $P(er \,|\, In_k) \leq 1 - p_0$ and $\sum P(In_k) = 1$, we conclude that $P(er) \leq 1 - p_0$. In other words, our proposed modification of Grover's algorithm returns the correct answer with the probability $\geq p_0$.

Grover's algorithm: discussion. The larger α, the better the speedup:

- when α tends to 0, the computation time tends to $C \cdot n$, i.e., to the time of non-quantum search;

- when α tends to 1, the computation time tends to $C \cdot n^{1/2}$, i.e., to the time of quantum search.

A curious thing is that in our estimates, for all values α, we get a *power law*. We believe that this fact is not a mathematical coincidence, there is a deep reason behind it.

Indeed, let us fix a computational setting – be it non-quantum computing, quantum computing with potentially unlimited number of qubits, or quantum computing with a restricted number of qubits, and let $t(n)$ be the "optimal" (smallest possible) computation time that is necessary, within this setting, to search for an element in a database of n elements. We want to show that under certain reasonable conditions, $t(n) \approx n^{\beta}$ for some real number β.

Informally, these reasonable conditions are that this algorithm should be *optimal* and that it should work on real-life databases. Let us describe what these two conditions mean.

Definition 1. *We say that a function $t(n)$ from natural numbers to real numbers ≥ 1* describes an optimal algorithm *if it satisfies the following two conditions:*

- *the function $t(n)$ is (non-strictly) increasing, and*
- *for some constant $C^+ > 0$, we have $t(n_1 \cdot n_2) \leq C^+ \cdot t(n_1) \cdot t(n_2)$ for all natural numbers n_1 and n_2.*

Let us present motivations for these conditions. First, if $m > n$, then, to each database of size m, we can add $n - m$ extra fictitious elements and thus search it in $t(n)$ steps. Since $t(m)$ is the smallest possible time for searching databases of m elements, we conclude that $t(m) \leq t(n)$, i.e., that the function $t(n)$ is non-strictly increasing.

The second condition comes from the fact that, if $n = n_1 \cdot n_2$, then we can subdivide the original database into n_1 pieces of size n_2. For each piece, we can check, in $t(n_2)$ steps, whether this piece contains the desired element; we thus have an auxiliary algorithm $\mathcal{A}$ that checks (in $t(n_2)$ steps) whether each piece contains the desired element. Now, we can view the original database as a database of n_1 pieces. We can apply the same (optimal) algorithm to this database of n_1 "elements", and in $t(n_1)$ calls to $\mathcal{A}$, find the piece that contains the desired element. Each call to $\mathcal{A}$ requires $t(n_2)$ steps; thus, overall, we need $t(n_1) \cdot t(n_2)$ steps.

If there was no overhead, then, since $t(n)$ $(= t(n_1 \cdot n_2))$ is the smallest possible time for searching databases of n elements, we would be able to conclude that $t(n) \leq t(n_1) \cdot t(n_2)$. In reality, there is a possibility of an overhead, so we require that $t(n) \leq C^+ \cdot t(n_1) \cdot t(n_2)$, where a constant C^+ describes the necessary amount of overhead.

Definition 2. *We say that a function $t(n)$* describes an algorithm that works on real-life databases *if for some constant $C^- > 0$, we have $C^- \cdot t(n_1) \cdot t(n_2) \leq t(n_1 \cdot n_2)$ for all natural numbers n_1 and n_2.*

The motivation for this definition is provided by the following argument. The need for a speedup is the most important for large databases, when n is large. However, large databases are rarely located on a single homogeneous medium, they usually have a hierarchical structure. Let us consider the simplest case when we have the simplest possible two-level hierarchy: database $\rightarrow$ pieces $\rightarrow$ elements, and all the pieces are of the same size n_2. In other words, the database consists of n_1 pieces, and each piece has n_2 elements in it.

In some of such cases, the pieces are physically separated to the extent that we cannot directly run our search algorithm on the database as a whole. Instead, we can use our algorithm to search into each piece, and then use the same algorithm to search for the right piece. We have already shown that this arrangement requires time $t(n_1) \cdot t(n_2)$. Since we require that the original algorithm

finish its search in time $t(n)$ for all databases – including hierarchical ones, in the idealized no-overhead situation, we would conclude that this number of steps should not exceed $t(n)$ $(= t(n_1 \cdot n_2))$, i.e., that $t(n_1) \cdot t(n_2) \le t(n)$. Similarly to the previous definition, the constant C^- take care of the possible overhead.

Proposition. *If a function $t(n)$ describes an optimal algorithm that works on real-life databases, then $t(n) = \Theta(n^\beta)$ for some $\beta \ge 0$.*

Proof. Since the function $t(n)$ is increasing, it is either bounded by a constant $\lim_{n\to\infty} t(n)$, or it tends to ∞ as $n \to \infty$. If the function $t(n)$ is bounded, then the proposition is true for $\beta = 0$. So, it is sufficient to consider the case when $t(n) \to \infty$.

Due to our assumptions, $t(n)$ is a positive-valued non-strictly increasing function for which

$$C^- \cdot t(n_1) \cdot t(n_2) \le t(n_1 \cdot n_2) \le C^+ \cdot t(n_1) \cdot t(n_2) \tag{1}$$

for all natural numbers n_1 and n_2.

Since $t(n) \ge 1$ for all $n > 0$, we can take logarithms of both sides and conclude that

$$c^- + T(n_1) + T(n_2) \le T(n_1 \cdot n_2) \le c^+ + T(n_1) + T(n_2), \tag{2}$$

where we denoted $c^- \stackrel{\text{def}}{=} \ln(C^-)$, $c^+ \stackrel{\text{def}}{=} \ln(C^+)$, and $T(n) \stackrel{\text{def}}{=} \ln(t(n))$. Since the values $t(n)$ are non-strictly increasing, their logarithms $T(n) \ge 0$ also form an increasing function.

Let us fix two natural numbers n_1 and n_2. For every two positive integers k_1 and k_2, if $n_1^{k_1} \le n_2^{k_2}$, then, due to the monotonicity of $T(n)$, we have $T(n_1^{k_1}) \le T(n_2^{k_2})$. Due to (2), we have $T(n_2^{k_2}) \le k_2 \cdot T(n_2) + k_2 \cdot c^+$ and $T(n_1^{k_1}) \ge k_1 \cdot T(n_1) + k_1 \cdot c^-$. Thus, $k_1 \cdot (T(n_1) + c^-) \le k_2 \cdot (T(n_2) + c^+)$.

Since $T(n) = \ln(t(n)) \to \infty$, for large enough n_i, we get $T(n_i) + c^- > 0$ and $T(n_i) + c^+ > 0$; so we can conclude that $k_1/k_2 \le r$, where $r \stackrel{\text{def}}{=} (T(n_2) + c^+)/(T(n_1) + c^-)$.

The inequality $n_1^{k_1} \le n_2^{k_2}$ is equivalent to $k_1 \cdot \ln(n_1) \le k_2 \cdot \ln(n_2)$, i.e., to $k_1/k_2 \le R$, where $R \stackrel{\text{def}}{=} \ln(n_2)/\ln(n_1)$. Thus, every rational number k_1/k_2 that is smaller than R is also smaller than r. By taking rational numbers that are arbitrarily close to R, we conclude that $R \le r$, i.e., after rearranging terms, that

$$\frac{T(n_1) + c^-}{\ln(n_1)} \le \frac{T(n_2) + c^+}{\ln(n_2)} \tag{3}$$

for all n_1 and n_2. For the ratio $r(n) \stackrel{\text{def}}{=} T(n)/\ln(n)$, we conclude that

$$|r(n_1) - r(n_2)| \le c \cdot \left(\frac{1}{\ln(n_1)} + \frac{1}{\ln(n_2)} \right),$$

where $c \stackrel{\text{def}}{=} \max(|c^-|, |c^+|)$; therefore, the sequence $r(n)$ converges. Let us denote its limit by β.

If we tend $n_2 \to \infty$ in (3), we conclude that $(T(n_1) + c^-)/\ln(n_1) \le \beta$ for every natural number n_1, i.e., that for every n, we have $T(n) \le \beta \cdot \ln(n) - c^-$.

Similarly, when we take $n_1 \to \infty$, we conclude that for every n, we have $\beta \cdot \ln(n) - c^+ \le T(n)$; so,

$$\beta \cdot \ln(n) - c^+ \le T(n) \le \beta \cdot \ln(n) - c^-. \tag{4}$$

Since $T(n) = \ln(t(n))$, we have $t(n) = \exp(T(n))$. If we apply $\exp(x)$ to all three terms in the inequality (4), we conclude that $C_- \cdot n^\beta \le t(n) \le C_+ \cdot n^\beta$ for some positive constants C_- and C_+ – i.e., that $t(n) = \Theta(n^\beta)$. The proposition is proven.

Comment. When $c^- = c^+ = 0$, a function satisfying the formula (2) is called a *totally additive number theoretic* function; see, e.g., [1]. It is known (see, e.g., [1, 2]) that every monotonic totally additive number theoretic function has the form $T(n) = \beta \cdot \ln(n)$. Our result uses the main idea from the proof of this statement, but our result extends it to the general case when the values c^- and c^+ may be different from 0.

Shor's algorithm: open problem. Shor's algorithm enables us to factorize an integer n in polynomial time by using $\lceil \log(n) \rceil$ qubits. What if we have fewer qubits? For example:

- What if we have $\log(n) - C$ qubits, for some constant C? is a polynomial-time algorithm still possible? our conjecture: yes (see general open problem below).
- What if we have $\log(n) - C \cdot \log(\log(n))$ qubits, for some constant C? is a polynomial-time algorithm still possible? our conjecture: yes (similarly, see general open problem below);
- What if we have $\alpha \cdot \log(n)$ qubits, for some $\alpha < 1$? is a polynomial-time algorithm still possible? our conjecture: probably not. Can we still have a speedup compared to the best classical algorithm? our conjecture: probably yes, but perhaps difficult to prove.

Deutsch-Josza algorithm: open problem. Since Shor's algorithm is rather complex, it may be instructive to analyze the effect of using fewer qubits on this algorithm without trying other (simpler) quantum computing algorithms first. A natural starting point may be the historically first quantum algorithm – Deutsch-Josza algorithm.

This algorithm solves the following problem: we have a function $f(x_1, \ldots, x_n)$ of n Boolean variables, and we know that this function is either constant or "balanced" (takes value "true" on exactly half of 2^n possible Boolean vectors); we need to know whether it is constant or balanced. Quantum algorithm solves this problem in one call to f (i.e., one computation of f) by using $n+1$ qubits (with probability 1).

Although this algorithm may not have as much practical promise as Grover's and Shor's algorithms, Deutsch-Josza algorithm is a spectacular example of the power of quantum computing, because the best possible deterministic algorithm for solving this problem requires at least $2^n/2+1$ calls to f. (A non-quantum probabilistic algorithm solves this problem in finitely many steps, but this number of steps increases as we decrease the allowed probability of an error.)

What if we have only n qubits? only $n - C$ qubits for some constant C? only $n - C \cdot \log(n)$ qubits? only $\alpha \cdot n$ qubits, for some $\alpha < 1$?

General open problem. We can ask a *general question:* Suppose that for some problem, we have a quantum algorithm that, for every input of length n, uses $q(n)$ qubits to solve the problem in time $\leq t(n)$ (either solves it with probability 1, or with a probability exceeding a certain threshold $p_0 \approx 1$). If, for some n, instead of $q \stackrel{\text{def}}{=} q(n)$ qubits, we only have $q' < q$ qubits, what is the smallest time t' that we need to solve the same problem? Can we have a general approach that could be applied to all such algorithms?

A *similar situation* occurs if, instead of qubits, we had "guessed" bits (as in nondeterministic algorithms for solving problems from the class NP). Then, if we have a program that runs in time t with q guess bits, and we are only allowed $q' < q$ bits, then we can try all $2^{q-q'}$ possible combinations of missing bits and get the same result in time $t' = t \cdot 2^{q-q'}$.

A *natural question* is: is t' equal to $t \cdot 2^{q-q'}$ for quantum computations as well?

Acknowledgments. This work was supported in part by NASA under cooperative agreement NCC5-209 and grant NCC2-1232, by the Future Aerospace Science and Technology Program

(FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant F49620-00-1-0365, by NSF grants CDA-9522207, EAR-0112968, EAR-0225670, and 9710940 Mexico/Conacyt, and by IEEE/ACM SC2001 and SC2002 Minority Serving Institutions Participation Grants. This work was also partly supported by a research grant from the Army Research Laboratories.

References

[1] J. Aczél and J. Dhombres, *Functional equations in several variables, with applications to mathematics, information theory, and to the natural and social sciences*, Cambridge University Press, Cambridge, 1991.

[2] P. Erdős, "On the distribution of additive functions", *Ann. of Math.*, 1946, Vol. 2, pp. 1–20.

[3] L. Grover, "A fast quantum mechanical algorithm for database search", *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, 1996, pp. 212–219.

[4] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack", *Phys. Rev. Lett.*, 1997, Vol. 79, No. 2, pp. 325–328.

[5] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, Cambridge, U.K., 2000.

[6] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring", *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, 1994, pp. 124–134.