УДК 512.6, 517.9, 519.6

STRUCTURE DESIGN TOOLKIT OF QUANTUM ALGORITHMS. PT 2.

Reshetnikov Andrey[1] , Tyatyushkina Olga[2], Ulyanov Sergey[3], Degli Antonio Giovanni[4]

[1]PhD, Associate professor;
Dubna State University, Institute of the system analysis and management;
141980, Dubna, Moscow reg., Universitetskaya str., 19;
e-mail: agreshetnikov@gmail.com.

[2]PhD, Associate professor;
Dubna State University, Institute of the system analysis and management;
141980, Dubna, Moscow reg., Universitetskaya str., 19;
e-mail: tyatyushkina@mail.ru.

[3]Doctor of Science in Physics and Mathematics, professor;
Dubna State University, Institute of the system analysis and management;
141980, Dubna, Moscow reg., Universitetskaya str., 19;
e-mail: ulyanovsv@mail.ru.

[4]PhD, professor;
Polo Didattico e di Ricerca di Crema;
Via Bramante, 65-26013, Crema (CR), Italy;
e-mail: gda@dsi.unimi.it.

Principles and methodologies of quantum algorithmic gates design are considered. The possibilities of quantum algorithmic gates simulation on classical computers are discussed. Applications of quantum gate of nanotechnology in intelligent control are introduced.

Keywords: Quantum computing, universal quantum gates, quantum operators, matrix transformation

ИНСТРУМЕНТАРИЙ ПРОЕКТИРОВАНИЯ КВАНТОВЫХ АЛГОРИТМОВ. Ч. 2.

Решетников Андрей Геннадьевич[1] , Тятюшкина Ольга Юрьевна[2], Ульянов Сергей Викторович[3], Джиованни дели Антонио[4]

[1]Кандидат технических наук, доцент;
ГБОУ ВО МО «Университет «Дубна»,
Институт системного анализа и управления;
141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: agreshetnikov@gmail.com.

[2]Кандидат технических наук, доцент;
ГБОУ ВО МО «Университет «Дубна»,
Институт системного анализа и управления;
141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: tyatyushkina@mail.ru.

[3]Доктор физико-математических наук, профессор;
ГБОУ ВО МО «Университет «Дубна»,
Институт системного анализа и управления;
141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: ulyanovsv@mail.ru.

[4]PhD, professor;
Polo Didattico e di Ricerca di Crema;
Via Bramante, 65-26013, Crema (CR), Italy;
e-mail: gda@dsi.unimi.it.

В работе рассмотрены принципы и методология проектирования квантовых алгоритмических ячеек. Дано описание возможностей моделирования квантовых алгоритмических ячеек на классических компьютерах для применения в проектировании интеллектуальном управлении на основе нанотехнологий

Ключевые слова: квантовые вычисления, универсальные квантовые вентели, квантовые операторы и матрицы преобразования

Introduction

The Hadamard gate creates the superpositions of classical states, and CNOT gate create the entangled states for robustness quantum computation. We consider the possibility of creation the interference with Quantum Discrete Fast Fourier Transformation (QFFT). Well-known examples of such transforms include the discrete Fourier transform (DFT), the Walsh-Hadamard transform, the trigonometric transforms such as the Sine and Cosine transform, the Hartley transform, and the Slant transform. All these different transforms find applications in signal and image processing, because the great variety of signal class, occurring in practice, cannot be handled by a single transform.

Quantum Parallelism, Interference, and QFFT.

We now have sufficient ingredients to understand how a quantum computation can perform logical operations and compute just like an ordinary computer [1-18]. We describe an algorithm, which makes use of quantum parallelism that we have hinted already: finding the period of long sequences.

Quantum Parallelism for Computation of Classical Functions. We use the fact that the efficiently implementable classical functions can be implemented with comparable complexity on a quantum computer using standard blocks. We assume perfect operations, so we do not deal with error control.

Remark. Let $f(x)$ be a classical polynomially computable function. Quantum parallelism can be used to compute all the values of $f(x)$ for all x at the same time. We will ignore any temporal work space, which returns to its original state by the end of the computation that might be needed to compute $f(x)$. Knowing that arbitrary classical function $f(x)$ can be implemented on quantum computer, we assume the existence of a quantum array U_f that implements f. What happens if U_f is applied to input in a superposition? The answer is easy but powerful; since U_f is a linear transformation, it is applied to all basis vectors in the superposition simultaneously and will generate a superposition of the results. In this way, it is possible to compute $f(x)$ for all n values of x in a single application of U_f. This effect is called quantum parallelism and was in detail in Part 1 described.

We use the following standard transformation to implement the quantum parallel computation of $f(x)$, $U_f : |x, y\rangle \xrightarrow{U_f} |x, y \oplus f(x)\rangle$ where $\oplus$ does not denote the direct sum vectors, but rather the bitwise exclusive – OR. Operator U_f defined in this way is unitary for any function $f(x)$. To compute $f(x)$ we apply U_f to $|x,0\rangle$. Since $f(x) \oplus f(x) = 0$, we have $U_f U_f = I$.

Graphically the transformation U_f is depicted as

$$\begin{array}{l} |x\rangle \rightarrow \\ |y\rangle \rightarrow \end{array} \boxed{U_f} \begin{array}{l} \rightarrow \quad |x\rangle \\ \rightarrow |y \oplus f(x)\rangle \end{array}$$

Consider a superposition of x values, $\sum_x a_x|x\rangle$. Then U_f transforms $\sum_x a_x|x\rangle\otimes|0\rangle$ as $\sum_x a_x|x,0\rangle \to \sum_x a_x|x, f(x)\rangle$.

Example: *Changing of sign*. We still have to explain how to invert the amplitude of desired result using operator U_f. Let $f(x)=1$ if the sign of x is to change, and $f(x)=1$ otherwise. We show, more generally, a simple and surprising way to invert the amplitude of exactly those states with $f(x)=1$ for a general f. Let U_f be the gate array that performs the computation $U_f : |x,b\rangle \to |x, b\oplus f(x)\rangle$. The additional register is set to the superposition $|b\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. The operation U_f then gives a superposition in which the phase of those x with $f(x)=1$ are inverted and $|b\rangle$ remains unchanged. This means that applying of U_f to the superposition $\sum_x a_x|x\rangle$ and choose $|b\rangle$ as above to end up in a state where the sign of all x with $f(x)=1$ has been change, and $|b\rangle$ remains unchanged. This is readily seen as follows:

$$U_f\left(\sum_x a_x|x\rangle\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)\right)=\boxed{\begin{array}{l}\frac{1}{\sqrt{2}}\left(\sum_{x\in X_0} a_x|x,0\rangle-\sum_{x\in X_0} a_x|x,1\rangle+\sum_{x\in X_1} a_x|x,1\rangle-\sum_{x\in X_1} a_x|x,0\rangle\right)\\ =\left(\sum_{x\in X_0} a_x|x\rangle-\sum_{x\in X_1} a_x|x\rangle\right)\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)\end{array}}$$

where $X_0=\{x|f(x)=0\}$ and $X_1=\{x|f(x)=1\}$. The operation introduces a phase factor of – 1 for exactly those $x\in X_1$, as desired. It also leaves $|b\rangle$ unchanged. In particular the extra register is not entangled with the x values.

Remark. This technique requires only one call to U_f, but restrict the phase to 1 and – 1.

Example: Suppose we want to change the phase of all elements of X_1 by γ. Instead of using $|b\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ we use $|b\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \gamma|1\rangle)$. The result of applying U_f is

$$U_f\left(\sum_x a_x|x\rangle\otimes\frac{1}{\sqrt{2}}(|0\rangle+\gamma|1\rangle)\right)=\boxed{\frac{1}{\sqrt{2}}\left(\sum_{x\in X_0} a_x|x,0\rangle+\gamma\sum_{x\in X_0} a_x|x,1\rangle+\sum_{x\in X_1} a_x|x,1\rangle+\gamma\sum_{x\in X_1} a_x|x,0\rangle\right)}$$

In general, the resulting state is not simply a tensor product of x and b with some additional phase shift. Usually, x and b become *entangled*. A possible approach to extracting the desired state from *entanglement* is to measure the last bit. The state in last equation becomes either

$$\boxed{\left(\sum_{x\in X_0} a_x|x,0\rangle+\gamma\sum_{x\in X_1} a_x|x,0\rangle\right)} \text{ or } \boxed{\left(\gamma\sum_{x\in X_0} a_x|x,1\rangle+\sum_{x\in X_1} a_x|x,1\rangle\right)}$$

If the measurement returns 0, we have achieved the desired phase shift. To get the desired result when the measured value is 1, we try multiplying the state by γ to get

$$\boxed{\left(\gamma^2\sum_{x\in X_0} a_x|x,1\rangle+\sum_{x\in X_1} a_x|x,1\rangle\right)}$$

We get the desired result only when $\gamma^2 = 1$.

Example: While the preceding calculation shows that general phase changes cannot be implemented with the technique for changing signs, the behavior when the last bit is measured suggest a way to change the phase of the elements of X_1 by a $2^m th$ root of unity. This trick can be to rotate part of the state by i or $-i$. Let $\gamma = i$. Perform U_f and measure the last bit. If the result is 0, the state will be

$$\boxed{\left(\sum_{x\in X_0} a_x |x,0\rangle + i\sum_{x\in X_1} a_x |x,0\rangle\right)}$$

and if the result is 1, the result will be

$$\left(i\sum_{x\in X_0} a_x |x,1\rangle + \sum_{x\in X_1} a_x |x,1\rangle\right) = \boxed{i\left(\sum_{x\in X_0} a_x |x,1\rangle - i\sum_{x\in X_1} a_x |x,1\rangle\right)}$$

except for a constant factor, the two states differ only in the phase of $x \in X_1$ and one can be transformed into the other by applying a phase change of – 1 to X_1. Thus half the time, when 0 is measured, only one call to $f(x)$ is needed. Otherwise a second phase change is needed, which requires an additional call to $f(x)$ for a total of two calls. By iterating this process, one can achieve arbitrary rotations by $2^m th$ roots of unity. Let $\gamma = e^{2\pi i/2^m}$. The transformation and measurement of the last bit give

$$\boxed{\left(\sum_{x\in X_0} a_x |x,0\rangle + e^{2\pi i/2^m}\sum_{x\in X_1} a_x |x,0\rangle\right)} \text{ or } \boxed{\left(e^{2\pi i/2^m}\sum_{x\in X_0} a_x |x,1\rangle + \sum_{x\in X_1} a_x |x,1\rangle\right)}$$

when the last bit is measured to be 0 or 1, respectively. In the latter case the state is, up to a constant overall phase,

$$\boxed{\left(\sum_{x\in X_0} a_x |x,1\rangle + e^{-2\pi i/2^m}\sum_{x\in X_1} a_x |x,1\rangle\right)}.$$

Essentially X_1 has been rotated by the right amount, but in the wrong direction. The desired state can be achieved by rotating X_1 by $\exp\{2\pi i/2^{m-1}\}$, twice the original amount, using the same process. In the worst case, rotating elements in X_1 by $\exp\{2\pi i/2^m\}$ requires $O(m)$ invocations of U_f. Surprisingly, the average number of calls to $f(x)$ for this rotation is only $\frac{2^{m-1}-1}{2^{m-2}}$.

This average is always less than two, so on average this technique requires fewer calls.

A different generalization of the sign change technique allows additional function calls to be avoided completely. Furthermore, multiple phases even up to 2^n of them, can be achieved in this way as long as they are all multiplies of the same underlying phase $\omega = e^{2\pi i/k}$. This technique requires only one function call plus $\log_2 k$ additional qubits.

Example: In this case, the bitwise XOR $\otimes$ is replaced by modular addition. Specifically, we use $U_f : |x,a\rangle \to |x, a + f(x) \bmod k\rangle$. Here, $f(x)$ maps states to the set $\{0,\ldots,k-1\}$ and the desired phase adjustment for state x is $\omega^{f(x)}$, where $\omega = e^{2\pi i/k}$. To perform this adjustment with a single evaluation of $f(x)$, we set the extra register in the superposition $R = \frac{1}{\sqrt{k}}\sum_{h=0}^{k-1} \omega^{k-h}|h\rangle$. The superposition R can be

constructed in $\log_2 k$ steps. To see behavior of U_f acting on $S \otimes R$, write $S = \sum_{j=0}^{k-1} \sum_{x \in X_j} a_x |x\rangle$, where X_j is the set of states for which $f(x) = j$.

Then, $S \otimes R = \frac{1}{\sqrt{k}} \sum_h \sum_j \sum_{x \in X_j} a_x \omega^{k-h} |x, h\rangle$.

Operating with $U_f : |x, a\rangle \to |x, a + f(x) \bmod k\rangle$ then gives $\frac{1}{\sqrt{k}} \sum_h \sum_j \sum_{x \in X_j} a_x \omega^{k-h} |x, h + j \bmod k\rangle$. For any j as h ranges from 0 to $k-1$, $m = h + j \bmod k$ ranges over these values as well. In terms of $m, h = m - j \bmod k$ and $k - h = j + (k-m) \bmod k$. Furthermore, since $\omega^k = 1$, we can write the sum as $\frac{1}{\sqrt{k}} \sum_m \sum_j \sum_{x \in X_j} a_x \omega^j \omega^{k-m} |x, m\rangle$ or $\boxed{\frac{1}{\sqrt{k}} \sum_j \sum_{x \in X_j} a_x \omega^j |x\rangle \otimes \sum_m \omega^{k-m} |m\rangle}$, which is just $DS \otimes R$, where D is diagonal matrix.

Remark: *Approximation of Phase Changes*. An arbitrary phase can be approximated by a series of shifts by roots of unity. For instance, consider $\phi = e^{p 2\pi i}$ for $0 \le p < 1$. Let $p = 0.b_1 b_2 \ldots b_k$ be the binary expansion of p to the desired precision.

The $\phi = \exp\left(2\pi i \sum_{j=1}^{k} b_j 2^{-j} \right) = \prod_{j \in B} e^{2\pi i 2^{-j}}$, where $B = \{ j | b_j = 1 \}$. Arbitrary unitary transformation cannot be efficiently approximated. However, if the phase changes can be concisely described, then they can be approximated to k bit precision using this relations. Let the phase change be represented by a diagonal matrix D with phases $D_{mm} = p_m$, and let f_j for each $j < k$ be such that $f_j(m)$ is the $j-th$ bit of p_m. Then D can be done by using one of the techniques for each f_j using the 2×2 phase matrices $\begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i / 2^j} \end{pmatrix}$. Thus, an arbitrary diagonal matrix D can be approximated to $e^{2\pi i / 2^k}$ in $O(k)$ steps plus the time it takes to compute each of the $f_j's$.

Discrete Fourier Transform (DFT). The N bit Discrete Fourier Transform (DFT) matrix F_N is defined by $(F_N)_{a,b} = \frac{1}{\sqrt{N_S}} \omega^{ab}$, where $a, b \in Z_{0, N_S - 1}$ and $\omega = e^{i \frac{2\pi}{N_S}}$. Note that $(F_N)_{a,b}$ is a symmetric and unitary matrix. If $\tilde{v}$ and v are complex N_S dimensional vectors such that $\tilde{v} = F_{N_B} v$ then we call $\tilde{v}$ the *DFT* of v. Calculating $\tilde{v}$ the naive way, by multiplying v by F_{N_B}, would take $O(N_S^2)$ classical elementary operations (complex multiplication mostly). Instead, it is possible to calculate $\tilde{v}$ from v in order $O(N_S \ln N_S)$ classical elementary operations using the well-known *Fast Fourier Transform (FFT) algorithm*. The FFT algorithm can be realized as a product of matrices each of which acts on at most 2 bits at a time. This way of expressing it is often called "*quantum FFT algorithm*" because it is ideal for quantum computation. For simplicity, we will only consider the case $N_B = 4$. What follows can be easily generalized to arbitrary $N_B \ge 1$. As usual, let $\omega = \exp\left(i \frac{2\pi}{N_S} \right) = \exp\left(i \frac{2\pi}{2^4} \right)$. Define matrices $\Omega, \Omega^2, \ldots$ by $\Omega = diag(1, \omega)$, $\Omega = diag(1, \omega^2)$, …. The FFT algorithm might been stated like this

F_{N_B}		*Expression*
F_1	=	$\frac{1}{\sqrt{2}}H$
F_2	=	$\frac{1}{\sqrt{2}}(H\otimes I_2)\left[I_2\oplus(\Omega^4)\right](I_2\otimes F_1)P_2$
F_3	=	$\frac{1}{\sqrt{2}}(H\otimes I_4)\left[I_4\oplus(\Omega^4\otimes\Omega^2)\right](I_2\otimes F_2)P_3$
F_4	=	$\frac{1}{\sqrt{2}}(H\otimes I_8)\left[I_8\oplus(\Omega^4\otimes\Omega^2\otimes\Omega^1)\right](I_2\otimes F_3)P_4$

The matrices P_2, P_3, P_4 are permutation matrices to be specified later. Although we could have written just a single equation that combined all four equations for F_i, $i=1,2,3,4$, we have chosen not to do this, so as to make explicit the recursive nature of the beast. Note

$$\Omega^4\otimes\Omega^2\otimes\Omega^1 = diag\left(1,\omega,\omega^2,\omega^3,\omega^4,\omega^5,\omega^6,\omega^7\right)=\omega^{2^2n(2)+2^1n(1)+2^0n(0)}$$

The last equation becomes clear when one realizes that $2^2n(2)+2^1n(1)+2^0n(0)$ operating on a state $|a_3,a_2,a_1,a_0\rangle$ gives the binary expansion of $d(0,a_2,a_1,a_0)$. Once the last equation sinks into the old bean, it is only a short step to the realization that:

$I_2\oplus(\Omega^4)$	=	$(\omega^4)^{n(1)\left[2^0n(0)\right]}$	=	$e^{i\left[n(1)n(0)\phi_2\right]}$
$I_4\oplus(\Omega^4\otimes\Omega^2)$	=	$(\omega^2)^{n(2)\left[2^1n(1)+2^0n(0)\right]}$		$e^{i\left[n(2)n(1)\phi_2+n(2)n(0)\phi_3\right]}$
$I_8\oplus(\Omega^4\otimes\Omega^2\otimes\Omega^1)$	=	$(\omega^1)^{n(3)\left[2^2n(2)+2^1n(1)+2^0n(0)\right]}$		$e^{i\left[n(3)n(2)\phi_2+n(3)n(1)\phi_3+n(3)n(0)\phi_4\right]}$

The final step is to replace all tensor product that contain H by bit operators:

$H\otimes I_8$	=	$H(3)$
$I_2\otimes H\otimes I_4$	=	$H(2)$
$I_4\otimes H\otimes I_2$	=	$H(1)$
$I_8\otimes H$	=	$H(0)$

If all the permutation matrices for F_j, $j=1,2,3,4$ are combined into a single permutation P_{BR}, then $P_{BR}=(I_4\otimes P_2)(I_2\otimes P_3)P_4$. Let $\vec{a}\in Bool^4$ label the columns of a 16×16 matrix.

As was known, the matrices P_4, P_3, P_2 acts as shown in Fig. 1.

$(\underline{a_3},a_2,a_1,\underline{a_0})$	$a_0\to a_3$
$\Downarrow$	P_4
$(\underline{a_0},a_3,a_2,\underline{a_1})$	$a_1\to a_0$
$\Downarrow$	$(I_2\otimes P_3)$
$(a_0,a_1,\underline{a_3},\underline{a_2})$	$a_2\to a_3$
$\Downarrow$	$(I_4\otimes P_2)$
(a_0,a_1,a_2,a_3)	P_{BR}

Figure 1. Permutation matrices P_4, P_3, P_2, P_{BR} that arise in the FFT algorithm for $N_B=4$

Therefore, their product P_{BR} takes $\vec{a}=(a_3,a_2,a_1,a_0)$ to $\vec{a}=(a_0,a_1,a_2,a_3)$; i.e., P_{BR} reverses the bits of $\vec{a}$. In general form the FFT algorithm can written as following $F_{N_B}=\frac{1}{\sqrt{N_S}}H(N_B-1)\cdots\Delta(2)H(2)\Delta(1)H(1)\Delta(0)H(0)P_{BR}$, where $H(\alpha)$ is the 1- bit Hadamard matrix operating on bit $\alpha\in Z_{0,N_B-1}$, P_{BR} is the bit reversible matrix for N_B bits, and $\Delta(\beta)=\Delta(\beta+1,\beta)\Delta(\beta+2,\beta)\cdots\Delta(N_B-1,\beta)$, where

$$\Delta(\alpha,\beta)=\exp\left[i\phi_{|\alpha-\beta|+1}n(\alpha)n(\beta)\right] \quad \phi_\gamma=\frac{2\pi}{2^\gamma}$$

Thus, $\Delta(\alpha,\beta)$ is diagonal matrix whose diagonal entries are either 1 or a phase factor.

Example. For example, $N_B=3$, $n(0)n(2)|a_2,a_1,a_0\rangle=\begin{cases}1|a_2,a_1,a_0\rangle & \text{if } a_2=a_0=0\\ 0 & \text{otherwise}\end{cases}$ so $\Delta(0,2)=e^{i\phi_3 n(0)n(2)}=diag\left(\overset{000}{1},\overset{001}{1},\overset{010}{1},\overset{011}{1},\overset{100}{1},\overset{101}{e^{i\phi_3}},\overset{110}{1},\overset{111}{e^{i\phi_3}}\right)$.

For $N_B=3$, reversing the bits of the numbers contained in $Z_{0,7}$ exchanges $1=d(001)$ with $4=d(100)$ and $3=d(011)$ with $6=d(110)$, and it leaves all other numbers in $Z_{0,7}$ the same. Thus, for $N_B=3$, P_{BR} is the 8×8 permutation matrix which corresponds to the following product of transpositions: $(1,4)(3,6)$.

Note that $F_{N_B}, H(\alpha), \Delta(\alpha)$ and P_{BR} are all symmetric matrices. Hence, taking the transpose of both sides of F_{N_B}, one gets $F_{N_B}=\frac{1}{\sqrt{N_S}}P_{BR}H(0)\Delta(0)H(1)\Delta(1)H(2)\Delta(2)\cdots H(N_B-1)$.

Both equations are called the *quantum FFT algorithm*.

Remark: *Quantum Cooley-Tukey FFT Algorithm and Bit – Reversible Permutation Matrix*. The classical Cooley-Tukey FTT factorization for a 2^n- dimensional vector is given by $F_{2^n}=A_nA_{n-1}\cdots A_1P_{2^n}=\underline{F}_{2^n}P_{2^n}$, where $A_i=I_{2^{n-1}}\otimes B_{2^i}$, $B_{2^i}=\frac{1}{\sqrt{2}}\begin{pmatrix}I_{2^{i-1}} & \Omega_{2^{i-1}}\\ I_{2^{i-1}} & -\Omega_{2^{i-1}}\end{pmatrix}$ and $\Omega_{2^{i-1}}=diag\left(1,\omega_{2^i},\omega_{2^i}^2,\ldots,\omega_{2^i}^{2^{i-1}-1}\right)$ with $\omega_{2^i}=e^{-\frac{2\pi}{2^i}}$. We have that $F_2=W=H=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$. The operator $\underline{F}_{2^n}=A_nA_{n-1}\cdots A_1$ represent the computational kernel of Cooley-Tukey FFT while P_{2^n} represents the permutation, which needs to be performed on the elements of the input vector before feeding that vector into the computational kernel. The Genteleman – Sande FFT factorization is obtained by exploiting the symmetry of F_{2^n} and transporting the Cooley-Tukey factorization leading to $F_{2^n}=P_{2^n}A_1^T\cdots A_{n-1}^TA_n^T=P_{2^n}\underline{F}_{2^n}^T$, where $A_1^T\cdots A_{n-1}^TA_n^T=\underline{F}_{2^n}^T$ represent the computational kernel of the Genteleman – Sande FFT while P_{2^n} represents the permutation which needs to be performed to obtain the elements of the output vector in the correct order. A quantum circuit for the implementation of $\underline{F}_{2^n}^T$ is presented by developing a factorization of the operator B_{2^i} as

$$B_{2^i}=\frac{1}{\sqrt{2}}\begin{pmatrix}I_{2^{i-1}} & \Omega_{2^{i-1}}\\ I_{2^{i-1}} & -\Omega_{2^{i-1}}\end{pmatrix}=\frac{1}{\sqrt{2}}\begin{pmatrix}I_{2^{i-1}} & I_{2^{i-1}}\\ I_{2^{i-1}} & -I_{2^{i-1}}\end{pmatrix}\begin{pmatrix}I_{2^{i-1}} & 0\\ 0 & \Omega_{2^{i-1}}\end{pmatrix}$$

Let $C_{2^i} = \begin{pmatrix} I_{2^{i-1}} & 0 \\ 0 & \Omega_{2^{i-1}} \end{pmatrix}$. It then follows that

$$\boxed{B_{2^i} = \left(H \otimes I_{2^{i-1}}\right) C_{2^i}} \quad \textit{and} \quad \boxed{A_1 = I_{2^{n-i}} \otimes B_{2^i} = \left(I_{2^{n-i}} \otimes H \otimes I_{2^{i-i}}\right)\left(I_{2^{n-i}} \otimes C_{2^i}\right)}$$

Remark: *Unitary of Quantum Fourier Transform.* For a system of n qubits QFFT is defined as

$$\boxed{F = \frac{1}{\sqrt{2^n}} \sum_{x,k=0}^{2^n-1} |k\rangle e^{2\pi i k x / 2^n} \langle x|}.$$

Verify that F is unitary. Write down the matrix representation of F for $n = 2$. One has

$$\begin{array}{|c|c|l|} \hline F^* F & = & \frac{1}{\sqrt{2^n}} \sum_{x,k,x',k'} |x'\rangle e^{-2\pi i' k x' / 2^n} \langle k'|k\rangle e^{2\pi i k x / 2^n} \langle x| \\ \hline & = & \sum_{x,x'} |x'\rangle \underbrace{\left(\frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} \exp\left\{2\pi i k \left(x - x'\right)/ 2^n\right\} \right)}_{\langle x'|F^* F|x\rangle} \langle x| \\ \hline \end{array}$$

If $x = x'$, then $\langle x'|F^* F|x\rangle = 1$. Otherwise, $\langle x'|F^* F|x\rangle = \frac{1}{2^n} \frac{1 - e^{2\pi i (x - x')}}{1 - e^{2\pi i (x - x')/2^n}} = 0$, since $\text{N э}\left(x - x'\right) < 2^n$. Thus $F^* F = 1$. For $n = 2$ one has $F = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$.

Example. Let us consider a quantum computation consisting of *n+l+m* qubits, where a total of n qubits (to be called the index bits) are used for an FFT, a total of *l* qubits describe the Hilbert space in which the operator U_ψ acts, and m extra working qubits are required for temporary storage. Let $Q = 2^n$. The accuracy of result will grow as 1/Q. Assume that the n index qubits are initially in the state $|0\rangle$ and that the l qubits are initially in the state V_a. The state vector V_a may be generated using a quantum algorithm, such as quantum annealing (hence, the need for V_a to be generated in quantum polynomial time). That is, the initial state is $|\psi\rangle = |0\rangle|V_a\rangle$, where the m work qubits are assumed to be $|0\rangle$ unless specified performs otherwise. We a $\pi/2$ rotation on each of the n idex qubuts to obtain the state $|\psi\rangle = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} |j\rangle|V_a\rangle$.

Next, one performs a series of quantum logic operations that transform the computation into the state $|\psi\rangle = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} |j\rangle \left(U_\psi\right)^j |V_a\rangle$. This transformation is accomplished by applying the operation U_ψ to the second set of l qubits (which are initially in the state V_a) j times. It can be implemented easily by performing a loop (indexed by i) from 1 to M.

Remark. Using standard quantum logic operations, set a flag qubit to the value $|1\rangle$ if and only if $i < j$ and perform the operation U_ψ conditioned on the value of this flag. Thus only those components of the above superposition for which $i < j$ are effected. Finally, undo the flag qubit and continue with the next iteration. After M iterations, the state above is obtained.

At this point, it is helpful to rewrite the state in a slightly different manner. Label the eigenvectors of U_ψ by the state $|\phi_k\rangle$ and the corresponding eigenvalues with λ_k. We can then write $|V_a\rangle = \sum_k c_k |\phi_k\rangle$ in which case the state $|\psi\rangle = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} |j\rangle (U_\psi)^j |V_a\rangle$ can be rewritten as

$$|\psi\rangle = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} |j\rangle (U_\psi)^j \sum_k c_k |\phi_k\rangle$$

$$|\psi\rangle = \frac{1}{\sqrt{M}} \sum_k c_k \sum_{j=0}^{M-1} |j\rangle (\lambda_k)^j |\phi_k\rangle$$

If we write λ_k as $e^{i\omega_k}$ and change the order of the qubits that the labels $|\phi_k\rangle$ appears first, the result is seen then most clearly:

$$|\psi\rangle = \frac{1}{\sqrt{M}} \sum_k c_k |\phi_k\rangle \sum_{j=0}^{M-1} e^{i\omega_k j} |j\rangle$$

It is now self-evident that a quantum FFT performed on the m index qubits will reveal the phases ω_k and thereby the eigenvalues λ_k. The quantum FFT requires only poly(n) operations, whereas the accuracy of the result will scale linearity with $M = 2^n$. Each frequency is seen to occur with the amplitude $c_k = \langle V_a | \phi_k \rangle$; by performing a measurement on the n index qubits, one thus obtains each eigenvalue with probability $|c_k|^2$. Only polynomial number of trials is therefore required to obtain any eigenvalue for which c_k is not exponentially small. If the initial state $|V_a\rangle$ is close to the desired state (i.e., $\langle V_a | V_a \rangle$ is close to 1), then only a few trials may be necessary.

Remark. The number of qubits required for the FFT is not as large as one might at first suppose, based on the earlier statement that the accuracy scales linearly with the size of the FFT. This statement is true only for fixed U_ψ. By increasing the length of time in $U_\psi(t)$ one can obtain the eigenvalues to arbitrary precision using a fixed number of FFT points. However, the number of points in the FFT must be sufficiently large so as to seperate the frequencies corresponding to distinct eigenvectors. This is how the estimate of 6 or 7 qubits (64 or 128 FFT points) is made.

Quantum Parallelism, QFFT and Interference. Consider the sequence $f(0), f(1), \ldots, f(Q-1)$, where $Q = 2^k$; we shall use quantum parallelism to find its period. We start with a set of initially spin-down particles which we group into sets (two quantum registers, or quantum variables): $|0,0\rangle = |\downarrow,\downarrow,\ldots;\downarrow,\downarrow,\ldots\rangle$, the first set having k bits; the next having sufficient for our need. (In fact other registers are required, but by applying Bennett`s solution to space management they may be suppressed in our discussion here.) On each bit of the first register we perform the $U_{-\pi/2}$ one-bit operation, yielding a superposition of every possible bit-string of length k in this register: $\rightarrow \frac{1}{\sqrt{Q}} \sum_{a=0}^{Q-1} |a;0\rangle$. The next stage is to break down the computation, corresponding to the function $f(a)$, into a set of one-bit and two-bit unitary operations. The sequence of operations is designed to map the state $|a;0\rangle$ to the state $|a; f(a)\rangle$ for any input a. Now we see that the number of bits required for this second register must be at least sufficient to store the longest result $f(a)$ for any of these computations. When, however, this sequence of operations is applied to our exponentially large superposition, instead of the single input, we obtain

$\boxed{\rightarrow \frac{1}{\sqrt{Q}}\sum_{a=0}^{Q-1}|a;f(a)\rangle}$. An exponentially large amount of computation has been performed essentially for free. The final computational step, like the first, is again a purely quantum mechanical one.

Consider a discrete quantum FFT on the first register: $\boxed{|a\rangle \rightarrow \frac{1}{\sqrt{Q}}\sum_{c=0}^{Q-1}e^{2\pi iac/Q}|c\rangle}$. It is easy to see that this is reversible via the inverse transform and indeed it is readily verified to be unitary. Further, an efficient way to compute this transform with one-bit and two-bit gates has been described by Coopersmith (Fig. 2). When this quantum FFT is applied to our superposition, we obtain $\boxed{\rightarrow \frac{1}{Q}\sum_{a=0}^{Q-1}\sum_{c=0}^{Q-1}e^{2\pi iac/Q}|c;f(a)\rangle}$. The computation is now complete and we retrieve the state output from the quantum computation by measuring the state of all spins in the first register (the first k bits). Indeed, once the FFT has been performed the second register may even discarded.

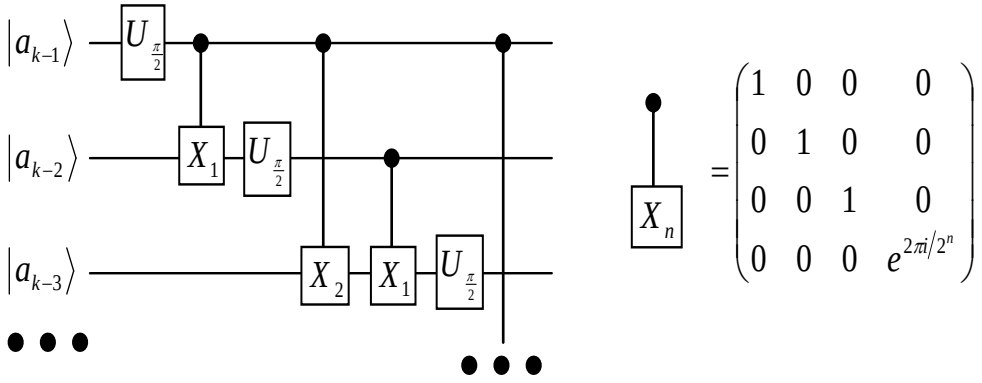

Figure 2. Circuit for QFFT of the variable $|a_{k-1}\ldots a_1 a_0\rangle$ using Coppersmith`s FFT approach. The two-bit "X_n" gate may itself be decomposed into one-bit and XOR gates

Suppose $f(a)$ has period r so $f(a+r)=f(a)$. The sum over a will yield *constructive interference* from the coefficients $\exp\left\{\frac{2\pi iac}{Q}\right\}$ only when $\frac{c}{Q}$ is a multiple of the reciprocal period $\frac{1}{r}$. All other values of $\frac{c}{Q}$ will produce *destructive interference* to a greater or lesser extent. Thus, the probability distribution for finding the first register with various values is shown schematically by Fig. 3.

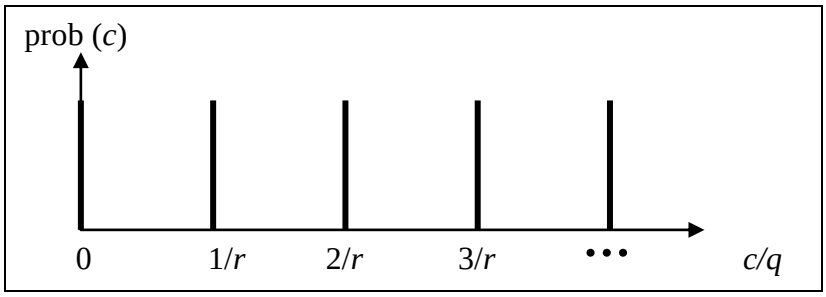

Figure 3. Plot of the probability of each result prob (c) versus c/q. Constructive interference produces narrow peaks of the inverse period of the sequence 1/r

One complete run of the quantum computation yields a random value of c/q underneath one of the peaks in the probability of each result *prob(c)*. That is, we obtain a random multiple of the inverse period. To extract the period itself we need only repeat this quantum computation roughly $\log\log\frac{r}{k}$ times in order to

have a high probability for at least one of the multiplies to be relatively prime to the period r - uniquely determining it. Thus, this algorithm yields only a probabilistic result. Fortunately, we can make this probability as high as we like.

Quantum Computing: Unified Approach to Fast Unitary Transforms

Discrete orthogonal transforms and discrete unitary transforms have found various applications in signal, image, and video processing, in pattern recognition, in bio-computing, and in numerous other areas. Well-known examples of such transforms include the discrete Fourier transform (DFT), the Walsh-Hadamard transform, the trigonometric transforms such as the Sine and Cosine transform, the Hartley transform, and the Slant transform. All these different transforms find applications in signal and image processing, because the great variety of signal class, occurring in practice, cannot be handled by a single transform.

On a classical computer, the straightforward way to compute a discrete orthogonal transform of a signal vector of length N takes in general $O(N^2)$ operations. An important aspect in many applications is to achieve the best possible computational efficiency. The examples mentioned above allow an evaluation with as few as $O(N \log N)$ operations or – in the case of the wavelet transforms – even with as little as $O(N)$ operations. In view of the trivial lower bound of $\Omega(N)$ operations for matrix-vector-products, we notice that these algorithms are optimal or nearly optimal.

Remark: The rules of the game change dramatically when the ultimate limit of computational integration is approached, that is, when information is stored in single atoms, photons, or other quantum mechanical systems. The operation manipulating the state of such a computer have to follow the dictum of quantum mechanics. However, this is not necessarily a limitation.

A striking example of the potential speed-up of quantum computation has been given by Shor in 1994. He showed that integers can be factored in polynomial time on a quantum computer. In contrast, there are no polynomial time algorithms known for this problem on a classical computer. The quantum computing model does not provide a uniform speed-up for all computational tasks. In fact, there are a number of problems, which do not allow any speed-up at all. For instance, it can be shown that a quantum computer searching a sorted database will not have any advantage over a classical computer. On the other hand, if we use the classical algorithms on a quantum computer, then it will simply perform the calculation in a similar manner to a classical computer. In order for a quantum computer to show its superiority one needs to design new algorithms, which take advantage of quantum parallelism.

A quantum algorithm may be thought of as a discrete unitary transform, which is followed by some I/O operations. This observation partially explains why signal transforms play a dominant role in numerous quantum algorithms. Another reason is that it is often possible to find extremely efficient quantum algorithms for the discrete orthogonal transforms mentioned above. For instance, the discrete Fourier transform (DFT) of length $N = 2^n$ can be implemented with $O(\log^2 N)$ operations on a quantum computer.

A quantum computer directly manipulates information stored in the state of quantum mechanical systems. The available operations have many attractive features but also underlie severe restrictions, which complicate the design of quantum algorithms. We are present a divide-and-conquer approach to the design of various quantum algorithms. The class of algorithm includes many transforms, which are well known in classical signal processing applications. We show how fast quantum algorithms can be derived for the DFT, the Walsh-Hadamard transform, the Slant transform, and the Hartley transform. All these algorithms use at most $O(\log^2 N)$ operations to transform a state vector of a quantum computer of length N.

Divide-and-Conquer methods. We have seen that a number of powerful operations are available on a quantum computer. Suppose that we want to implement a unitary or orthogonal transform $U \in U(2^n)$ on a quantum computer. The goal will be to find an implementation of U in terms of elementary quantum gates. Usually, our aim will be to find first a factorization of U in terms of sparse structured unitary matrices U_i, $U \cong U_1 U_2 \cdots U_k$, where, of course, k should be small. It is often very easy to derive quantum circuits for structured sparse matrices. For example, if we can find an implementation with few multiply controlled unitary gates for each factor U_i, then the overall circuit will be extremely efficient.

Remark. The success of this method depends of course very much on the availability suitable factorization of *U*. However, in the case orthogonal transforms used in signal processing, there are typically numerous classical algorithms available, which provide the suitable factorizations. It should be noted that, in principle, an exponential number of gates might be needed to implement even a diagonal unitary matrix. Fortunately, we will see that most structured matrices occurring in practice have very efficient implementations.

In fact, we will see that all the transforms of size $2^n \times 2^n$ discussed in the following can be implemented with merely $O(\log^2 2^n) = O(n^2)$ elementary quantum gates. We discuss a simple – but novel – approach to derive such efficient implementations. This approach is based on a divide-and-conquer technique. Assume that we want to implement a family of unitary transforms U_N, where $N = 2^n$ denotes the length of the signal. Suppose further the family U_N can be recursively generated by a recursive circuit construction. We will give a generic construction for the family of pre-computation circuits Pre- and the family of post-computation circuits Post. This way, we obtain a fairly economic description of the algorithms.

Example: Assume that total of *P*(*N*) elementary operations are necessary to implement the pre-computation circuit $\text{Pre}_{N/2}$. Then the overall number *T*(*N*) of elementary operations can be estimated from the recurrence equation T(N) = T(N/2) + P(N). The number of operations T(N) for the recursive implementation can be estimated as follows:

LEMMA: If $P(N) \in \Theta(\log^p N)$, then $T(N) \in O(\log^{p+1} N)$.

Fourier transform. We will illustrate the general approach by way of some examples. The first example is the DFT. A quantum algorithm implementing this transform found a most famous application in Shor's integer factorization algorithm.

Example: Algorithm and Physical Interpretation of Fast Fourier Transform (FFT). Consider a *signal* (any quantity that is a function of time, perhaps pressure in the air) affecting a *system* (anything that responds to a signal, like a microphone). The outcome of this interaction is called, naturally, *the response of the system to the signal.* The question is. How are we to compute such response digitally? In *many* applications (in which often the horizontal axis is not time, but also space, e. g. in image processing) it is of great interest to compute the system response, thereby simulating the system. The following is a rough description of how this is done.

First we *digitize* the signal, by *sampling it* often enough. Then we must know how the system behaves. It turns out that, if the system is *time invariant* (does not change its behavior over time) and *linear* (roughly, gives twice the response to twice the signal), then its behavior is completely captured by its *impulse response* - a description of what it does over time if it is given a sudden unit "jerk" at time zero. If we know that, since any signal is the sum of such jerks happening at various times, and since we know that the system is time-invariant and linear, then all we have to do is calculate the responses at various times, and add them to get the total system response.

Remark: To do the algebra, suppose that the signal is the sequence $(a_0,\ldots,a_{T-1})$ of real numbers, and the system impulse response is $(b_0,\ldots,b_{T-1})$ (assume that they both have the same time horizon, that is, they both die after *T* time unit. Then at time 0 we have the system response $a_0 b_0$ - only the first pulse of the signal has arrived, and has only gotten the immediate response b_0. At time 1 we have the system response $a_0 b_1 + a_1 b_0$, because now a_1 gets the immediate response, while a_0 causes the delay-1 response of the system, and the two are added. After two steps, then the system response is $a_0 b_2 + a_1 b_1 + a_2 b_0$.

What is the system response after *t* time units? If $t < T$, that is, if the signal keeps arriving at time *t*, then it is $c_t = \sum_{i=1}^{t} a_i b_{t-i}$. If $t \geq T$ – that is, if the signal has died out – then the system response keeps coming, since the system keeps responding to the signal in the past: The system response is then $c_t =$

$\sum_{i=t-T+1}^{T-1} a_i b_{t-i}$. Finally, at $t = 2T - 1$ we have $c_{2T-2} = a_{T-1} b_{T-1}$, and thereafter $c_t = 0$: the system response has died out.

This sequence of formulas for $c_0,\ldots,c_{2T-2}$ *are precisely the formulas for the coefficients of the product of product of two polynomials.*

$$\left(\sum_{i=0}^{T-1} a_i x^i\right)\cdot\left(\sum_{i=0}^{T-1} b_i x^i\right) = \sum_{i=0}^{2T-2} c_i x^i$$

This is not a coincidence: We can think of both the signal and the system response as two polynomials *in x, where x can be thought of as a unit delay,* x^2 two delays, etc. Then the product of the two polynomials is, naturally enough, the system response.

Conclusion: It is of great interest to compute very fast the coefficients c_t, t = 0,…, 2T – 2, of the product of two given polynomials of degree $T - 1$.

Unfortunately, just by looking at the formulas, the number of operations required to compute the c_t's seems to be $\Omega(T^2)$: The number of terms increases from 1 to *T* and then down to 1 again, for a sum of about T^2. How can we calculate the c_t's much faster?

Remark: The c_t's are the coefficient of the polynomial *C(x)* = $\sum_{i=0}^{2T-2} c_t x^t$ = *A(x)B(x),* where *A(x)* = $\sum_{i=0}^{T-1} a_t x^t$, and *B(x)* = $\sum_{i=0}^{T-1} b_t x^t$. And here is a scheme for calculating these coefficients:

1	*Calculate the values of A(x) and B(x) at enough points* $x_1,\ldots, x_n$ *where* $n \geq 2T - 1$.
2	*Calculate the values of C(x) at these points as* $C(x_i) = A(x_i)\cdot B(x_i)$, *i=1,…, n*
3	*Now that we know at least 2T – 1 values of the 2T – 2-degree polynomial C(x), we can interpolate and recover the coefficients. (Recall that there is a unique d-degree polynomial that goes through d+1 points.)*

But there are problems with this approach: Although step (2) is easy (it only requires *n* multiplications), step (1) seems to still require $\Omega(n^2)$ operations, and step (3) seems even harder. Have we accomplished nothing with this clever manoeuvre?

It turns out that we need another trick: Pick the points $x_1,\ldots, x_n$ on which to evaluate A(x) and B(x) so that the n evaluations can be done together very fast. It turns out that the most clever way to choose these points is to find n different points $x_1,\ldots, x_n$ such that the equation $x^n = 1$ holds for all of them.

Remark: Obviously, there are at most two real numbers that satisfy this equation: 1, if *n* is odd, and perhaps –1, if *n* is even. But there are exactly *n complex* numbers that do: *The n complex roots of unity*. They are *n* points lying, in the complex plane, on the unit circle. And since on the unit circle rising to the *n*th power means multiplying the angle by *n*, all *n* of these numbers are mapped to the real unity when raised to the *n*th power. Let us call then $x_1 = 1$, $x_2 = w$, $x_3 = w^2$, $x_4 = w^3$,…, $x_n = w^{n-1}$. What we need to remember from now on about these numbers is that *w* is some number satisfying $w^n = 1$ – nothing else. Except one thing: If we take a root of unity like w^i, and add its powers $1+w^i+w^{2i}+\ldots+w^{(n-1)i}$ then we get 0 – because the powers of w^i are just points around the origin, "pulling it in all different directions." With one exception: If $i = 0$, then of course the sum is 1+1+…+1 = *n*.

We conclude that we want to find a fast way to compute these *n* values:

$$A(w^i) = \sum_{i=1}^{n-1} a_i w^{ij},\ j = 0,\ldots, n-1. \qquad (1)$$

In this equation (1), j varies over 0,…, $n-1$ to define the n points $x_1,\ldots, x_n$ on which to evaluate A(x), and the summation is the value $A(x_{j+1}) = A(w^j)$. We need to remember all the time that $w^n = 1$ – this is the fact that is going to save us from the $\Omega(n^2)$ algorithm.

We are going to compute all n values in (1) together, *by divide-and-conquer.* Assume that n is a power of two – presumably the next power of two above $2T-2$. Then, we *divide* the sequence of coefficients $a_0,\ldots, a_{n-1}$ into two subsequences: The *even subsequence* $a_0, a_2, a_4, \ldots, a_{n-2}$, and the *odd subsequence* $a_1, a_3, a_5, \ldots, a_{n-1}$. Then we can write (1) as

$A(w^j)$	=	$\sum_{i=0}^{\frac{n}{2}-1} a_{2i} w^{2ij} + \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w^{2ij+j}$
	=	$\sum_{i=0}^{\frac{n}{2}-1} a_{2i} (w^2)^{ij} + w^j \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} (w^2)^{ij}$

Now this equation is just two problems of the same kind applied to sequences of length $\frac{n}{2}$ (compute the values of a degree $\frac{n}{2}$ - 1 – polynomial at the $\frac{n}{2}$-nd roots of 1 – notice that is 1, w^2, w^4,…, w^{2n-2} are precisely the $\frac{n}{2}$-nd roots of unity). To obtain the $A(w^j)$ from the results of the conquered subproblems, we just multiply the j mod $\frac{n}{2}$-th result of the second evaluation by w^j and add it to the corresponding j mod $\frac{n}{2}$-th result of the first. The points on which we need to evaluate the subproblems are just $\frac{n}{2}$, because this is the number of the possible values of $(w^2)^j$. This is quintessential divide-and-conquer, with recurrence

T(n)	=	$2T(\frac{n}{2}) + n$

Here the *O(n)* term is the work required to "put together" the results of the conquered parts (*n* multiplications and additions). The total complexity of the algorithm is, as we know, *O(n* log *n)*.

Remark: This algorithm, which computes the values of an *n*-degree polynomial at the *n n*-th roots of unity in *O(n* log *n)* time by divide-and-conquer is called *the Fast Fourier Transform (FFT).* It is perhaps one of the most important and widely used algorithms; it was discovered by Cooley and Tuckey in the 1950's.

Example: Notice that the FFT, as described so far, only takes care of Step 1 of this scheme: Evaluating *A(x)* and *B(x)* at *n* points. How are we to carry out Step (3) – recovering the coefficients of *C(x)* from these *n* values? The amazing fact is that *this can be can be done with another FFT* – but this time w^{-1} playing the role of *w*.

This is just algebra. Suppose that we apply this "inverse" FFT to the values of $C(w^j)$, $j=1,\ldots,n-1$,

$\sum_{j=0}^{n-1} C(w^j) w^{-jk}$	=	$\sum_{j=0}^{n-1} (\sum_{i=0}^{n-1} c_i w^{ij}) w^{-jk}$
	=	$\sum_{i=0}^{n-1} \left[\sum_{j=0}^{n-1} c_i w^{j(i-k)} \right]$

Consider however the inner sum of the last line, for some fixed values of *i* and *k*. If these values are the same, then the parenthesis contributes *n*. If they are not, then the powers of $w^{i-k} \neq 1$ cancel each other. So,

the above sum is just $\sum_{j=0}^{n-1} C(w^j)w^{-jk} = c_k$ - it recovers the coefficients of *C(x)*. To summarize, the three steps of polynomial multiplication algorithm now are these:

Step1:	Compute the FFT of $a_0,\ldots,a_{n-1}$ top obtain the sequence $A_0,\ldots,A_{n-1}$. Repeat with $b_0,\ldots,b_{n-1}$ top obtain the sequence $B_0,\ldots,B_{n-1}$.
Step2:	Compute $C_i := A_i \cdot B_i$, $i = 0,\ldots, n-1$ (note that these are complex number multiplications.)
Step3:	Compute the inverse FFT (FFT with w^{-1} in the place of *w*), and then divide the results by *n*, to obtain $c_0,\ldots, c_{n-1}$.

Remark: We often need to solve the system response problem when the input signal is *periodic* – that is, it is repeated after *n* time units. The system response then is found by an FFT with *half* the points (*n* = *T*).

Remark: Also, since all we needed from *w* is the equations $w^n = 1$ and $\sum w^i = 0$, we could use any arithmetical domain where these equations hold – not necessarily the complex numbers with their slow multiplications. We shall see in good time that there are domains in which we are computing modulo a large integer, in which such equations hold.

Example: The FFT Circuit. As with all divide-and-conquer algorithms, it is worthwhile to *unravel* the recursion, to see what the algorithm *really* does. If we do this, the algorithm becomes the *circuit* shown in Fig. 4.

Remark: A word of explanation: The nodes are complex variables. The nodes on the left are the inputs (but in a funny order), and those on the right are the outputs. An arrow labeled with the integer *j* (unlabeled arrows are labeled "0") from *x* to *y* can be though of as carrying the value xw^j to *y*. The two arrows coming into a node (other than the input nodes) are added together. Under this interpretation, Figure 4 shows the FFT of eight points.

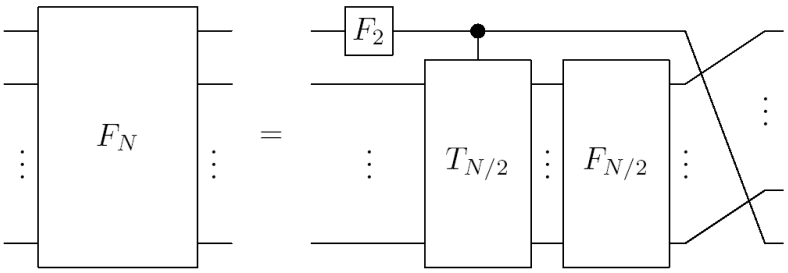

Figure 4. The recursive structure of the quantum Fourier transform

Notice these properties of this circuit:

•	There are 3 = log *n* levels, with *n* variables each, and four complex operations per variable (actually, seven real operations), for a total of 7*n* log *n* operations.
•	There is a unique path between every input node and every output node.
•	The path between a_i and $A(w^j)$ has label sum equal to *ij* modulo 8 (and it makes sense to take powers of *w* modulo 8, since we know that $w^8 = 1$).
•	The previous two facts ensure that the circuit correctly computes the FFT.
•	The inputs are mixed up this weird order. (Compare the *binary representations* of the indices of an input and an output that are opposite one another).
•	Notice how neatly arranged this circuit is for *parallel evaluation.* Indeed, the FFT is a natural for parallelism, and can be carried out in log *n* short parallel stages. Often the FFT is computed by specialized embedded parallel hardware.

•	Each stage of the FFT consists of $\frac{n}{2}$ "*butterfly*" operations – a typical butterfly is the subcircuit shown in bold in Fig. 4.

Recall that the DFT F_N of length $N = 2^n$ can be described by the matrix

$$F_N = \frac{1}{\sqrt{N}} (\omega^{jk})_{j,k=0,\ldots,N-1'}$$

Where ω denotes a primitive N-th root of unity, $\omega = \exp(2\pi i/N)$. And i denotes a square root of –1.

Remark: The main observation behind the fast quantum algorithm dates at least back to work by Danielson and Lanczos in 1942 (and is implicitly contained in numerous earlier works). They noticed that the matrix F_N might be written as

$$F_N = \frac{1}{\sqrt{2}} P_N \begin{pmatrix} F_{N/2} & F_{N/2} \\ F_{N/2} T_{N/2} & -F_{N/2} T_{N/2} \end{pmatrix}$$

Where P_N denotes the permutation of rows given by $P_N |bx\rangle = |xb\rangle$ with x an n – 1-bit integer, and b a single bit, and $T_{N/2} := \mathrm{diag}(1, \omega, \omega^2, \ldots, \omega^{N/2-1})$ denotes the matrix of twiddle factors.

This observation allows to represent F_N by the following product of matrices:

$$F_N = \underbrace{P_N} \overbrace{\begin{pmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{pmatrix}} \begin{pmatrix} 1_{N/2} & 0 \\ 0 & T_{N/2} \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1_{N/2} & 1_{N/2} \\ \underline{\underline{1_{N/2}}} & -1_{N/2} \end{pmatrix} =$$

$$= P_N (1_2 \otimes F_{N/2}) \begin{pmatrix} 1_{N/2} & 0 \\ 0 & T_{N/2} \end{pmatrix} \underline{(F_2 \otimes 1_{N/2})}$$

This factorization yields an outline of an implementation on a quantum computer.

Remark: It remains to detail the different steps in this implementation. The first step is a single qubit operation, implementing a butterfly structure. The next step is slightly more complicated. We observe that $T_{N/2}$ is a tensor product of diagonal matrices $D_j = \mathrm{diag}(1, \omega^{2^{j-1}})$. Indeed, $T_{N/2} = D_{n-1} \otimes \ldots \otimes D_2 \otimes D_1$.

Thus, $1_{N/2} \oplus \mathrm{T}_{N/2}$ can be realized by controlled phase shift operations (see Fig. 5 for an example).

We then recurse to implement the FT of smaller size. The final permutation implements the cyrlic rotation of the quantum wires.

Example: The complexity of the QFT can be estimated as follows. If we denote by *R(N)* the number of gates necessary to implement the DFT of length $N = 2^n$ on a quantum computer, then implies the recurrence relation $R(N) = R(N/2) + \Theta(\log N)$, which leads to the estimate $R(N) = O(\log^2 N)$.

It should be noted that all permutations $P_N (1_2 \otimes P_{N/2}) \ldots (1_{N-2} \otimes P_4)$ at the end can be combined into a single permutation of quantum wires.

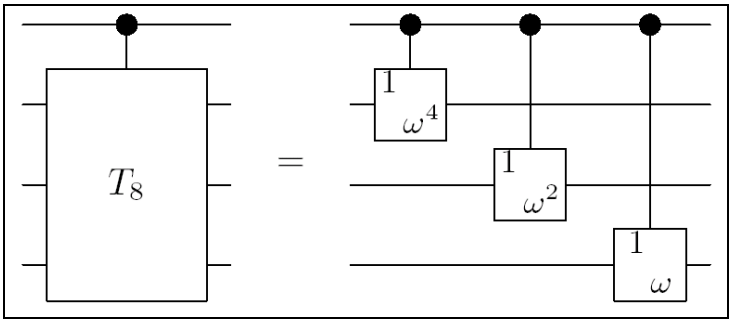

Figure 5. Implementation of the twiddle matrix $1_8 \oplus T_8$

The resulting permutation is the bit reversal, see Fig. 6.

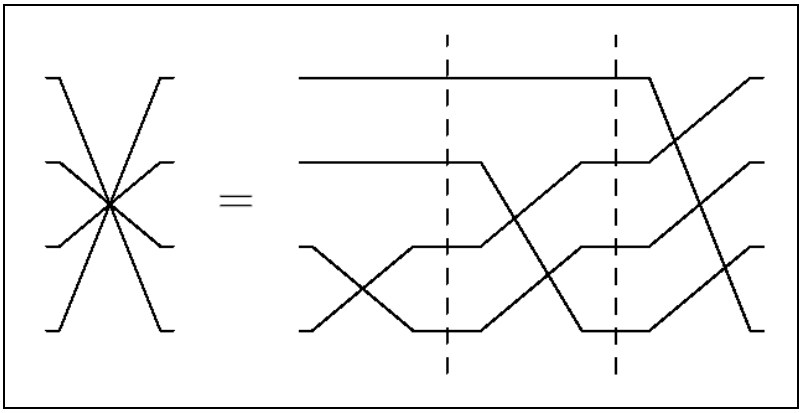

Figure 6. The bit reversal permutation resulting from $P_8(1_2 \otimes P_4)(1_4 \otimes P_2)$

The Walsh – Hadamar transform

Example: The Walsh-Hadamard transform W_N is maybe the simplest instance of the recursive approach. This transform is defined by the Hadamard gates W_2 = H in the case of signals of length 2. For signals of larger length, the transform is defined by $W_N = (1_2 \otimes W_{N/2})(H \otimes 1_{N/2})$.

This yields the recursive implementation shown in Fig. 7.

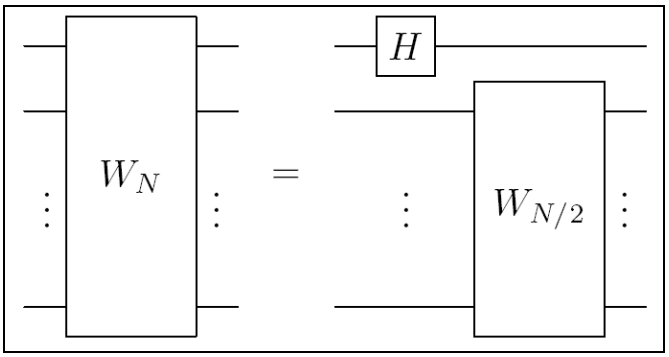

Figure 7. Recursive implementation of the Walsh-Hadamard transform

Since $P(N) = \Theta(1)$, the Lemma 3 shows that the number of operations $T(N) \in O(\log N)$. It is of course trivial to see that in this case exactly log N operations are needed.

The Slant transform

Example: The Slant transform is used in image processing for the representation of images with many constant or uniformly changing gray levels.

Remark: The transform has good energy compaction properties. It is used in Intel's 'Indeo' video compression and in numerous still image compression algorithms.

The slant transform S_N is defined for signals of length N= 2 by the Hadamard matrix

$S_2 = H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, and for signals of length $N = 2^k$, N >2, by

$$S_N = Q_N \begin{pmatrix} S_{N/2} & 0_{N/2} \\ 0_{N/2} & S_{N/2} \end{pmatrix}, \tag{2}$$

where $0_{N/2}$ denotes the all-zero matrix, and Q_N is given by the matrix product

$$Q_N = P_N^a (1_{N/2} \oplus \hat{Q}_N)(H \otimes 1_{N/2}) P_N^b . \tag{3}$$

The matrices in (3) are defined as follows: $1_{N/2}$ is the identity matrix, H is the Hadamard matrix, and P_N^a realizes the transposition (1,N/2), that is,

$P_N^a \lvert 1\rangle = \lvert N/2\rangle$	$P_N^a \lvert N/2\rangle = \lvert 1\rangle$	and	$P_N^a \lvert x\rangle = \lvert x\rangle$ otherwise.

The matrix P_N^b is defined by $\mathrm{P}_N^b \lvert x\rangle = \lvert x\rangle$ for all x except in the case x= N/2+1, where it yields the phase change $\mathrm{P}_N^b \lvert N/2+1\rangle = -\lvert N/2+1\rangle$. Finally $\hat{Q}_N = \begin{pmatrix} A_N & 0 \\ 0 & 1_{\left(\frac{N}{2}-2\right)} \end{pmatrix}$, $A_N = \begin{pmatrix} a_N & b_N \\ -b_N & a_N \end{pmatrix}$, where a_N and b_N are recursively defined by $a_2 = 1$ and $b_N = \frac{1}{\sqrt{1+4(a_{N/2})^2}}$ and $a_N = 2b_{N^{a_{N/2}}}$. It is easy to check that A_N is a unitary matrix.

Remark: The definition of the Slant transform suggests the following implementation. Equation (2) tells us that the input signal of a Slant transform of length N is first processed by two Slant transforms of size $N/2$, followed by a circuit implementing Q_N. We can write equation (2) in the form

$$S_N = Q_N \begin{pmatrix} S_{N/2} & 0_{N/2} \\ 0_{N/2} & S_{N/2} \end{pmatrix} = Q_N \left(1_2 \otimes S_{N/2}\right).$$

The tensor product structure $1_2 \otimes S_{N/2}$ is compatible with our decomposition into quantum bits. This means that a single copy of the circuit $S_{N/2}$ acting on the lower significant bits will realize this part. It remains to give an implementation for Q_N. Equation (3) describes as a product of for sparse matrices, which are easy to implement. Indeed, the matrix P_N^b is realized by conditionally excerting the phase gate Z. The matrix $H \otimes 1_{N/2}$ is implemented by a Hadamard gate H acting on the most significant bit. A conditional application of A_N implements the matrix $1_{N/2} \otimes \hat{Q}_N$. A conditional swap of the least and the most significant qubit realizes P_N^a, that is, three multiply controlled NOT gates implement P_N^a.

THEOREM: *The Slant transform of length $N = 2^k$ can be realized on a quantum computer with at most $O(\log^2 N)$ elementary operations (that is, controlled NOT gates and single qubit gates), assuming that additional workbits are available.*

Proof. Recall that a multiplied controlled gate can be expressed with at most O(log *N*) elementary operations as long as additional workbits are available. It follows from the Lemma that at most $O(\log^2 N)$ elementary operations are needed to implement the Slant transform.

The quantum circuit realizing this implementation is depicted in Fig. 8.

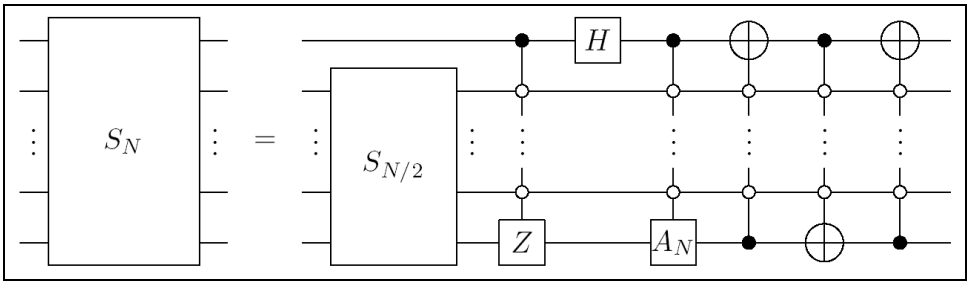

Figure 8. Implementation of the Slant transform

The recursive step is realized by a single Slant transform of size $S_{N/2}$. The next three gates implement P_N^b, $H \otimes 1_{N/2}$, and $1_{N/2} \oplus \hat{Q}_N$, respectively. The last three gates implement P_N^a. Thus, the implementation of Q_N totals five multiply controlled gates and one single qubit gate.

The Hartley transform

Example: the discrete Hartley transform H_N is defined for signals of length $N=2^n$ by the matrix $H_N = \frac{1}{\sqrt{N}}\left(\cos(2\pi kl) + \sin(2\pi kl)\right)_{k,l=0,\ldots,N-1}$.

Remark: The discrete Hartley transform is very popular in classical signal processing, since it requires only real arithmetic but has similar properties. In particular, there are classical algorithms available, which outperform the FFT algorithms. We derive a fast quantum algorithm for this transform, again based on a recursive divide-and-conquer algorithm. A fast algorithm for the discrete Hartley transform based on a completely different approach has been discussed by Klappenecker and Rotteler.

The Hartley transform can be recursively represented as

$$H_N = \frac{1}{\sqrt{2}} \begin{pmatrix} 1_{N/2} & 1_{N/2} \\ 1_{N/2} & -1_{N/2} \end{pmatrix} \begin{pmatrix} 1 & \\ & BC_{N/2} \end{pmatrix} \begin{pmatrix} H_{N/2} & \\ & H_{N/2} \end{pmatrix} Q_N, \tag{4}$$

where Q_N is the permutation $Q_N |xb\rangle = |bx\rangle$, with b a single bit, separating the even indexed samples and the odd indexed samples; for instance $Q_8 (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)^t = (x_0, x_2, x_4, x_6, x_1, x_3, x_5, x_7)^t$.

The matrix $\mathrm{BC}_{N/2}$ is given by

$$BC_{N/2} = \begin{pmatrix} 1 & \\ & CS_{N/2-1} \end{pmatrix}, \quad \text{with} \quad \begin{pmatrix} c_N^1 & & & & & & s_N^1 \\ & \ddots & & & & \iddots & \\ & & c_N^{N/4-1} & & s_N^{N/4-1} & & \\ & & & 1 & & & \\ & & s_N^{N/4-1} & & -c_N^{N/4-1} & & \\ & \iddots & & & & \ddots & \\ s_N^1 & & & & & & -c_N^1 \end{pmatrix}$$

The equation (4) leads to the implementation sketched in Fig. 9.

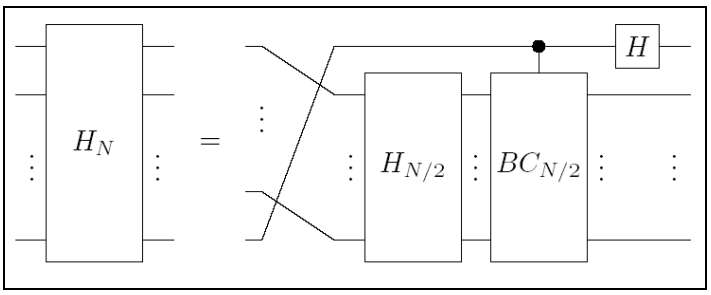

Figure 9. Recursive implementation of the Hartley transform.

Remark: It remains to describe the implementation of $BC_{N/2}$. It will be instructive to detail the action of the matrix $BC_{N/2}$ on a state vector of $n-1$ qubits. We will need a few notations first. Denote by $|bx\rangle$ a state vector of $n-1$ qubits, where *b* denotes a single bit and *x* an $n-2$ bit integer. We denote by x' the two's complement of *x*. We mean by $x=0$ the number 0 and by1 the number $2^{2n-2}-1$, that is, 1 has all bits set and 0 has no bit set. Then the action of $BC_{N/2}$ on $|bx\rangle$ is given by

$BC_{N/2}\|00\rangle = \|00\rangle,$	$BC_{N/2}\|0y\rangle = c_N^y\|0y\rangle + s_N^y\|1y'\rangle,$
$BC_{N/2}\|01\rangle = \|01\rangle,$	$BC_{N/2}\|1y\rangle = s_N^{y'}\|0y'\rangle - c_N^{y'}\|1y\rangle,$

where $s_N^k = \sin(2\pi k/N)$ and $c_N^k = \cos(2\pi k/N)$.

We are now in the position to describe the implementations of $BC_{N/2}$ shown in Fig. 10. In the first step, the least $n-2$ qubits are conditionally mapped to their two's complement. More precisely, the input signal $|bx\rangle$ is mapped to $|bx'\rangle$ if *b* =1, and does not change otherwise. Thus, the circuit *TC* implements the involuntary permutation corresponding to the two's complement operation. This can be done with $O(n)$ elementary gates, provided that sufficient workspace is available. In the next step, a sign change is done if $b-1$, that is, $|1x\rangle \mapsto -|1x\rangle$, unless the input *x* was equal to zero, $|10\rangle \mapsto |10\rangle$. The next step is a conditioned cascade of rotations. The least significant bits determine the angle of the rotation on the $(n-1st)$ most significant qubit. The *k*-th qubits exerts a rotation,

$$R_{2^k} = \begin{pmatrix} \cos(2\pi 2^k/N) & -\sin(2\pi 2^k/N) \\ \sin(2\pi 2^k/N) & \cos(2\pi 2^k/N) \end{pmatrix},$$

on the most significant qubit. Finally, another two's complement circuit traditionally applied to the state.

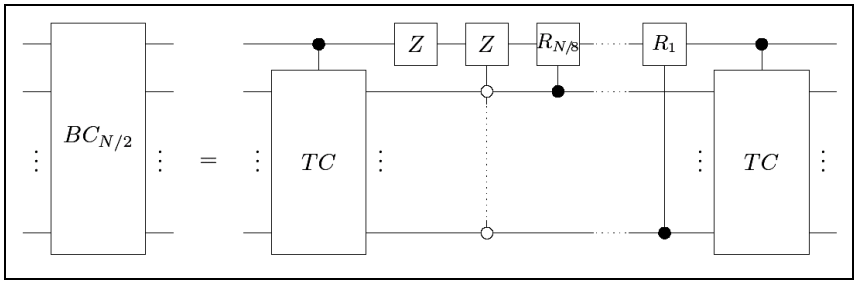

Figure 10. Implementation of the matrix $BC_{N/2}$

One readily checks that the implementation indeed maps $BC_{N/2}|00\rangle$ to $|00\rangle$ and $BC_{N/2}|10\rangle$ to $|10\rangle$. The input $|0x\rangle$ is mapped to $c_N^x|0x\rangle + s_N^x|1x'\rangle$, as desired. Assume that the input is $|1x\rangle$ with $x \neq 0$. Then the state is changed to $|1x'\rangle$ by the circuit *TC*, and after that its sign is changed, which yields $-|1x'\rangle$. The

rotations map this state to $s_N^{x'}|0x'\rangle - c_N^{x'}|1x'\rangle$. The final conditional two's complement operation yields the state $s_N^{x'}|0x'\rangle - c_N^{x'}|1x\rangle$, which is exactly what we want.

The initial permutation, the circuit $BC_{N/2}$ and the Hartley gate in Fig. 9 can be implemented with $\Theta(\log^2 N)$ elementary gates. It is crucial that additional work bits are available, otherwise the complexity will increase to $\Theta(\log^2 N)$. The Lemma completes the proof of the following theorem:

THEOREM: *There exists a recursive implementation of the discrete Hartley transform H_N on a quantum computer with $O(\log^2 N)$ elementary gates (that is, controlled NOT gates and single qubit gates), assuming that additional work bits are available.*

It should be emphasized that the divide-and-conquer approach is completely general. It can be applied to a much larger class of circuits, and is of course not restricted to signal processing applications. Moreover, it should be emphasized that many variations of this method are possible.

The method of the design of quantum algorithms takes advantage of a divide-and-conquer approach [1-18]. We have illustrated the method in the design of quantum algorithms for the Fourier, Walsh, Slant and Hartley transforms. The same method can be applied to derive fast algorithms for various discrete Cosine transforms. One reason might be that the quantum circuit model implements only straight-line programs. We defined recursions on top of that model, similar to macro expansions in many classical programming languages. The benefit is that many circuits can be specified in a very lucid way.

References

1. Gruska J. Quantum computing. – Advanced Topics in Computer Science Series, McGraw-Hill Companies, London. – 1999.
2. Nielsen M.A. and Chuang I.L. Quantum computation and quantum information. – Cambridge University Press, Cambridge, Englandю – 2000.
3. Hirvensalo M. Quantum computing. – Natural Computing Series, Springer-Verlag, Berlinю – 2001.
4. Hardy Y. and Steeb W.-H. Classical and quantum computing with C++ and Java Simulations. – Birkhauser Verlag, Basel. – 2001.
5. Hirota O. The foundation of quantum information science: Approach to quantum computer (in Japanese). – Japan. – 2002.
6. Pittenberg A.O. An introduction to quantum computing and algorithms. – Progress in Computer Sciences and Applied Logic. – Vol. 19. – Birkhauser. – 1999.
7. Brylinski F.K. and Chen G. (Eds). Mathematics of quantum computation. – Computational Mathematics Series. – CRC Press Co. – 2002.
8. Lo H.-K., Popescu S. and Spiller T. (Eds). Introduction to quantum computing and information. – World Scientific Publ. Co. – 1998.
9. Berman G.P., Doolen G.D., Mainieri R. and Tsifrinovich V.I. Introduction to quantum computers. – World Scientific Publ. Co. – 1999.
10. Rieffel E. and Polak W. An introduction to quantum computing for non-physicists // ACM Computing Surveys. – 2000. – Vol. 32. – No 3. – pp. 300 – 335.
11. Hogg T., Mochon C., Polak W. and Rieffel E. Tools for quantum algorithms // International Journal of Modern Physics. – 1999. – Vol. C10. – No 7. – pp. 1347 – 1361.
12. Uesaka Y. Mathematical principle of quantum computation (in Japanese). – Corona Publ. Co. Ltd. – 2000.
13. Marinescu D.C. and Marinescu G.M. Approaching quantum computing. – Pearson Prentice Hall, New Jersey. – 2005.

14. Benenti G., Casati G. and Strini G. Principles of quantum computation and information. –Singapore: World Scientific. – Vol. I. – 2004; – Vol. II. – 2007.
15. Nakahara M. and Ohmi T. Quantum computing: From Linear Algebra to Physical Realizations. – Taylor & Francis. – 2008.
16. Stenholm S. and Suominen K.-A. Quantum approach to informatics. – Wiley- Interscience. A J. Wiley&Sons, Inc. – 2005.
17. Jaeger G. Quantum Information: An Overview. – N.Y.: Springer Verlag. – 2007.
18. McMahon D.Quantum computing explained. – Wiley- Interscience. A J. Wiley&Sons, Inc. – 2008.