Computer Physics Communications 237 (2019) 47–61

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Massively parallel quantum computer simulator, eleven years later

Hans De Raedt [a], Fengping Jin [b], Dennis Willsch [b,c], Madita Willsch [b,c], Naoki Yoshioka [d], Nobuyasu Ito [d,e], Shengjun Yuan [f,*], Kristel Michielsen [b,c,**]

[a] *Zernike Institute for Advanced Materials, University of Groningen, Nijenborgh 4, NL-9747 AG Groningen, The Netherlands*
[b] *Institute for Advanced Simulation, Jülich Supercomputing Center, Forschungzentrum Jülich, D-52425 Jülich, Germany*
[c] *RWTH Aachen University, D-52056 Aachen, Germany*
[d] *RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan*
[e] *Department of Applied Physics, School of Engineering, The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113-8656, Japan*
[f] *School of Physics and Technology, Wuhan University, Wuhan 430072, China*

ARTICLE INFO

Article history:
Received 12 May 2018
Received in revised form 30 October 2018
Accepted 8 November 2018
Available online 29 November 2018

Keywords:
Quantum computation
Computer simulation
High performance computing
Parallelization
Benchmarking

ABSTRACT

A revised version of the massively parallel simulator of a universal quantum computer, described in this journal eleven years ago, is used to benchmark various gate-based quantum algorithms on some of the most powerful supercomputers that exist today. Adaptive encoding of the wave function reduces the memory requirement by a factor of eight, making it possible to simulate universal quantum computers with up to 48 qubits on the Sunway TaihuLight and on the K computer. The simulator exhibits close-to-ideal weak-scaling behavior on the Sunway TaihuLight, on the K computer, on an IBM Blue Gene/Q, and on Intel Xeon based clusters, implying that the combination of parallelization and hardware can track the exponential scaling due to the increasing number of qubits. Results of executing simple quantum circuits and Shor's factorization algorithm on quantum computers containing up to 48 qubits are presented.

1. Introduction

Simulating universal quantum computers on conventional, classical digital computers is a great challenge. Increasing the number of qubits (denoted by N) of the quantum computer by one requires a doubling of the amount of memory of the digital computer. For instance, to accurately simulate the operation of a universal quantum computer with 45 qubits, one needs a digital computer with slightly more than 1/2 Petabytes ($10^{15}/2$ bytes) of memory. There are only a few digital computers in the world which have the amount of memory, number of compute nodes, and a sufficiently powerful network connecting all the compute nodes to perform such simulations. Performing computations with such a large amount of memory and processors requires a simulator that can efficiently use the parallel architecture of present day supercomputers.

We report on novel algorithms and techniques implemented in the Jülich universal quantum computer simulator (JUQCS). In this paper, "universal quantum computer" refers to the theoretical, pen-and-paper, gate-based model of a quantum computer [1] in which the time evolution of the machine is defined in terms of a sequence of simple, sparse unitary matrices, with no reference to the real time evolution of a physical system. An article about an earlier version of the same simulator was published in this journal eleven years ago [2]. Since then, supercomputer hardware has evolved significantly and therefore we thought it was time to review and improve the computationally critical parts of the simulator and use it to benchmark some of the most powerful supercomputers that are operational today. In Table 1 we collect the main characteristics of the computer systems that we have used for our benchmarks.

JUQCS runs on digital computers ranging from personal computers to the largest supercomputers that are available today. The present version of the simulator comes in two flavors. One version, referred to as JUQCS-E (E referring to numerically exact, see below), uses double precision (8-byte) floating point arithmetic and has been used to simulate a universal quantum computer with up to 45 qubits. The 45 qubit limit is set by the amount of RAM memory available on the supercomputers that we have access to, see Table 1. For a system of N qubits and using 16 bytes per complex coefficient of the 2^N different basis states, the amount of memory required to store the wave function is 2^{N+4}, i.e., 1/2 PB are needed to store the wave function of $N = 45$ qubits. Adding storage for communication buffers (default is to use 2^{N-3} bytes) and an insignificant amount of bytes for the code itself, simulating a $N = 45$ qubit universal quantum computer requires a little more than 1/2 PB but certainly less than 1 PB of RAM memory.

* Corresponding author.
** Corresponding author at: Institute for Advanced Simulation, Jülich Supercomputing Center, Forschungzentrum Jülich, D-52425 Jülich, Germany.
E-mail addresses: s.yuan@whu.edu.cn (S. Yuan), k.michielsen@fz-juelich.de (K. Michielsen).

https://doi.org/10.1016/j.cpc.2018.11.005

Table 1
Overview of the computer systems used for benchmarking. The IBM Blue Gene/Q JUQUEEN [3] (decommissioned), JURECA [4] and JUWELS are located at the Jülich Supercomputing Center in Germany, the K computer of the RIKEN Center for Computational Science in Kobe, Japan, and the Sunway TaihuLight [5] at the National Supercomputer Center in Wuxi, China. The row "# qubits" gives the maximum number of qubits N that can be simulated with JUQCS-A (JUQCS-E). At the time of running the benchmarks on JUWELS, the maximum number of qubits N was limited to 43 (40)

	JUQUEEN	K computer	Sunway TaihuLight	JURECA-CLUSTER	JUWELS
CPU	IBM PowerPC A2	eight-core SPARC64 VIIIfx	SW26010 manycore 64-bit RISC	Intel Xeon E5-2680 v3	Dual Intel Xeon Platinum 8168
clock frequency	1.6 GHz	2.0 Ghz	1.45 GHz	2.5 GHz	2.7 GHz
memory/node	16 GB	16 GB	32 GB	128 GB	96 GB
# threads/core used	1 – 2	8	1	1 – 2	1 – 2
# cores used	1 – 262144	2 – 65536	1 – 131072	1 – 6144	1 – 98304
# nodes used	1 – 16384	2 – 65536	1 – 32768	1 – 256	1 – 2048
# MPI processes used	1 – 524288	2 – 65536	1 – 131072	1 – 1024	1 – 2048
# qubits	46 (43)	48 (45)	48 (45)	43 (40)	46 (43)

A second version, referred to as JUQCS-A (A referring to approximate), trades memory for CPU time and can be used to simulate a universal quantum computer with up to 48 qubits on digital computers with less than 1 PB of RAM memory, with a somewhat reduced numerical precision relative to the other version of the simulator. JUQCS-A employs adaptive coding to represent the quantum state in terms of 2-byte numbers, effectively reducing the memory requirements by a factor of eight relative to the one of JUQCS-E (see Section 4.1 for more details). The adaptive coding requires additional computation such that for some of the quantum gates, JUQCS-A takes a longer time to complete than JUQCS-E. The reduced precision (about 3 digits) has been found more than sufficient for all quantum circuits that have been tested so far.

From the quantum computer user perspective, JUQCS-E and JUQCS-A are fully compatible. In this document, the acronym JUQCS refers to both versions while JUQCS-E and JUQCS-A are used specifically to refer to the numerically exact version and the adaptive-coding version of the simulator, respectively. The only difference, if any, between JUQCS-E and JUQCS-A is in the accuracy of the results.

A quantum gate circuit for a universal quantum computer is a representation of a sequence of matrix–vector operations involving matrices that are extremely sparse. Only a few arithmetic operations are required to update one coefficient of the wave function. Therefore, in practice, simulating universal quantum computers is rather simple as long as there is no need to use distributed memory or many cores and the access to the shared memory is sufficiently fast [6–9]. The elapsed time it takes to perform such operations is mainly limited by the bandwidth to (cache) memory. However, for a large number of qubits, the only viable way to alleviate the memory access problem is to use distributed memory, which comes at the expense of overhead due to communication between nodes, each of which can have several cores that share the memory (as is the case on all machines listed in Table 1). Evidently, the key is to reduce this overhead by minimizing the transfer of data between nodes, which is exactly what JUQCS does [2].

Another road to circumvent the memory bottleneck is to use the well-known fact that propagators involving two-body interactions (two qubits in the case at hand) can be replaced by single-particle propagators by means of a Hubbard–Stratonovich transformation, that is by introducing auxiliary fields. A discrete version of this trick proved to be very useful in quantum Monte Carlo simulations of interacting fermions [10]. In Section 4, we show that the same trick can be used in the present context to great advantage as well, provided that the number of two-qubit gates is not too large and that it is sufficient to compute only a small fraction of the matrix elements between basis states and the final state. The latter condition considerably reduces the usefulness of this approach because for an algorithm such as Shor's, it is a-priori unknown which of the basis states will be of interest. Nevertheless, this trick of trading memory for CPU time is interesting in itself and has recently been used, in various forms and apparently without recognizing the relation to the auxiliary field approach to many-body physics, to simulate large random circuits with low depth [11–14].

JUQCS is a revised and extended version of the simulator, written in Fortran, developed about eleven years ago [2]. Depending on the hardware, the source code can be compiled to make use of OpenMP, the Message Passing Interface (MPI), or a combination of both. Apart from a few technical improvements, the "complicated" part of the simulator, i.e. the MPI communication scheme, is based on the same approach as the one introduced eleven years ago [2]. JUQCS-E and JUQCS-A use the same MPI communication scheme. During the revision, we have taken the opportunity to add some new elementary operations for implementing error-correction schemes and a translator that accepts circuits expressed in OpenQASM, i.e., the language used by the IBM Q Experience [15,16]. The executable code of JUQCS has been built using a variety of Fortran compilers such as Intel's ifort, GNU's gfortran, IBM's XLF, and others. Using JUQCS-A (JUQCS-E), a notebook with 16GB of memory can readily simulate a universal quantum computer with 32 (29) qubits. Since portability is an important design objective, we have not engaged in optimizing the code on the level of machine-specific programming to make use of, e.g., the accelerator hardware in the Sunway TaihuLight. We leave this endeavor to future work.

A JUQCS program looks very much like a conventional assembler program, a sequence of mnemonics with a short list of arguments. JUQCS converts a quantum circuit into a form that is suitable as input for the simulation of the real-time dynamics of physical qubit models, such as NMR quantum computing [17] using the massively-parallel quantum spin dynamics simulator SPI12MPI [6], or quantum computer hardware based on superconducting circuits [18]. A description of the instruction set that JUQCS accepts is given in Appendix A.

The primary design objective of the original JUQCS simulator [2] was to provide an environment for testing and optimizing the MPI communication part of SPI12MPI. The efficient simulation of spin-1/2 models (e.g. physical models of quantum computers) requires elementary operations that are significantly more complex than those typically used in universal quantum computation [6]. Therefore, to test the MPI communication part properly, JUQCS does not exploit the special structure of the CNOT and the Toffoli gate and also does not modify the input circuit using quantum gate circuit optimization techniques.

JUQCS is found to scale very well as a function of the number of compute nodes, beating the exponential growth in time that is characteristic for simulating universal quantum computers [1]. Such simulations can be very demanding in terms of processing power, memory usage, and network communication. Therefore, JUQCS can also serve as a benchmark tool for supercomputers.

We cover this aspect by reporting weak scaling plots obtained by running quantum algorithms on the supercomputers listed in Table 1.

The paper is structured as follows. In Section 2, we briefly review the basics of gate-based universal quantum computing, emphasizing the aspects which are important for the design of a simulator. Section 3 addresses techniques for distributing the workload of a simulation over many compute cores. The primary bottleneck of simulating a gate-based universal quantum computer is the amount of memory required to store the wave function representing the quantum state of the machine. Section 4 discusses two very different methods for alleviating this problem. In Section 5, we present results obtained by executing a variety of quantum circuits on JUQCS, running on five different supercomputers. Conclusions are given in Section 6.

2. Basic operation

A quantum computer is, by definition, a device described by quantum theory. The elementary storage unit of a quantum computer is, in its simplest form, represented by a two-level system, called qubit [1]. The state of the qubit is represented by a two-dimensional vector

$$|\Phi\rangle = a(0)|0\rangle + a(1)|1\rangle, \tag{1}$$

where $|0\rangle$ and $|1\rangle$ denote two orthogonal basis vectors of the two-dimensional vector space and $a_0 \equiv a(0)$ and $a_1 \equiv a(1)$ are complex numbers, normalized such that $|a_0|^2 + |a_1|^2 = 1$.

The internal state of a quantum computer comprising N qubits is described by a 2^N-dimensional unit vector of complex numbers

$$\begin{aligned}|\Phi\rangle &= a(0\ldots00)|0\ldots00\rangle + a(0\ldots01)|0\ldots01\rangle + \cdots \\ &\quad + a(1\ldots10)|1\ldots10\rangle + a(1\ldots11)|1\ldots11\rangle,\end{aligned} \tag{2}$$

where

$$\sum_{i=0}^{2^L-1} |a_i|^2 = 1, \tag{3}$$

i.e. by rescaling the complex-valued amplitudes a_i, we normalize the vector $|\Phi\rangle$ such that $\langle\Phi|\Phi\rangle = 1$.

Unlike in many-body physics where the leftmost (rightmost) bit of the basis state represents quantum spin number 1 (N), in the quantum computer literature it is common to label the qubits from 0 to $N-1$, that is the rightmost (leftmost) bit corresponds to qubit 0 ($N-1$) [1].

Executing a quantum algorithm on a universal, gate-based quantum computer consists of performing a sequence of unitary operations on the vector $|\Phi\rangle$. As an arbitrary unitary operation can be decomposed into a sequence of single-qubit operations and the CNOT operation on two qubits [1], it is sufficient to implement these specific operations as a sequence of arithmetic operations on the vector $\mathbf{v} = (a(0\ldots00), \ldots, a(1\ldots11))^{\mathrm{T}}$.

We illustrate the procedure for the Hadamard gate on qubit $0 \le j \le N-1$. The $2^N \times 2^N$ matrix $\mathscr{H}$ multiplying the vector $\mathbf{v}$ is given by $\mathscr{H} = \mathbb{1}_0 \otimes \cdots \otimes \mathbb{1}_{j-1} \otimes H \otimes \mathbb{1}_{j+1} \otimes \cdots \otimes \mathbb{1}_{N-1}$ where H denotes the 2×2 Hadamard matrix

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{4}$$

Obviously, the matrix $\mathscr{H}$ is very sparse. The matrix–vector multiplication $\mathbf{v} \leftarrow \mathscr{H}\mathbf{v}$ decomposes into 2^{N-1} matrix–vector multiplications involving H and vectors of length two only. The whole operation can be carried out in place (i.e. by overwriting $\mathbf{v}$), requiring additional memory of $\mathscr{O}(1)$ only.

In more detail, the rule to update the amplitudes reads

$$\begin{aligned} a(*\ldots*0_j*\ldots*) &\leftarrow \frac{1}{\sqrt{2}}\Big(a(*\ldots*0_j*\ldots*) \\ &\qquad + a(*\ldots*1_j*\ldots*)\Big) \\ a(*\ldots*1_j*\ldots*) &\leftarrow \frac{1}{\sqrt{2}}\Big(a(*\ldots*0_j*\ldots*) \\ &\qquad - a(*\ldots*1_j*\ldots*)\Big), \end{aligned} \tag{5}$$

where the $*$'s are placeholders for the bits $0,\ldots,j-1$ and $j+1,\ldots,N-1$. From Eq. (5), it follows that the update process consists of selected pair of amplitudes using bit j as index, replace the two amplitudes by the right-hand side of Eq. (5), and repeat the calculation for all possible pairs. Obviously, the update process exhibits a very high degree of intrinsic parallelism.

Implementing quantum gates involving two qubits, e.g. the CNOT gate, or three qubits, e.g. the Toffoli gate, requires selecting groups of four or eight amplitudes, respectively. In other words, instead of the two rules in Eq. (5), we have 2^{N-2} (2^{N-3}) groups of four (eight) amplitudes that need to be updated. Note that as the CNOT (Toffoli) gate only exchanges two of the four (eight) amplitudes, the computational work involved is less than in the case of say the Hadamard gate.

The result of executing a quantum gate circuit on JUQCS is an array of amplitudes $\mathbf{v}$ which represents the final state $|\Phi\rangle$ of the pen-and-paper quantum computer. According to quantum theory, measuring a single qubit j yields an outcome that is either 0 or 1 with probability $\sum_* |a(*\ldots*0_j*\ldots*)|^2$ or $\sum_* |a(*\ldots*1_j*\ldots*)|^2$, respectively. JUQCS provides methods for computing these probabilities as well as for generating events.

3. Parallelization techniques

As explained in Section 2, the sparse matrix structure of the quantum gates translates into an algorithm for updating the amplitudes which exhibits a very high degree of intrinsic parallelism. In this section, we discuss two different techniques to exploit this parallelism.

3.1. OpenMP

Assuming memory is not an issue, a platform independent method to distribute the computational work over several compute cores is to use OpenMP directives. Thereby, care has to be taken that the order in which the groups of amplitudes are processed is "cache friendly". The excerpt of the Fortran code given below shows how we have implemented a single-qubit gate in qubit j.

```
      nstates=2**(N-1)
      i=2**j
      if( nstates/(i+i) >= i ) then
!$OMP parallel do private ...
      do k=0,nstates-1,i+i
      do l=0,i-1
      i0 = ior(l, k)  ! *...* 0 *... *
      i1 = ior(i0, i) ! *...* 1 *... *
      ...
      end do
      end do
!$OMP end parallel do
      else
      do k=0,nstates-1,i+i
!$OMP parallel do private ...
```

```
        do l=0,i-1
        ...
        enddo
!$OMP end parallel do
        enddo
```

The indices k and l run over all possible values of the bits $(N-1),\ldots,j+1$ and $(j-1),\ldots,0$, respectively. We use the test "nstates/(i+i) $\geq$ i" to decide whether it is more efficient to distribute both the outer and inner loop or only the inner loop over all compute cores. Our numerical experiments show that the test "nstates/(i+i) $\geq$ i" is not optimal. The best choice depends on N and on the particular hardware in a seemingly complicated manner but the reduction of computation time is marginal. Therefore, we opted for the simple, universal test "nstates/(i+i) $\geq$ i".

The implementation of two-qubit gates involving qubits j_0 and j_1 requires three instead of two loops to generate the "*"'s in the bit string "$*\ldots*j_1*\ldots*j_0*\ldots*$", a simple generalization of the code used to implement single-qubit operations. This scheme straightforwardly extends to three-qubit gates.

3.2. MPI

As the number of qubits N increases, there is a point at which a single compute node does not have enough memory to store the whole vector of amplitudes such that it become necessary to use memory distributed over several compute nodes. A simple scheme to distribute the amplitudes over a number of nodes is to use the high-order bits as the integer representation of the index of the node. Let us denote the number of high-order bits that will be used for this purpose by N_{h}, the corresponding number of compute nodes by $K_{\mathrm{h}} = 2^{N_{\mathrm{h}}}$, and the number of amplitudes per compute node by $K_{\mathrm{l}} = 2^{N_{\mathrm{l}}}$ where $N_{\mathrm{l}} = N - N_{\mathrm{h}}$.

Obviously, there is no need to exchange data between compute nodes if we perform a single-qubit operation on qubit $0 \leq j < N_{\mathrm{l}}$ because each pair of amplitudes which need to be updated resides in the memory of the same compute node. However, if $N_{\mathrm{l}} \leq j < N-1$, it is necessary to exchange data between compute nodes before the two amplitudes can be multiplied by the 2×2 matrix which represents the quantum gate. For an operation such as the Hadamard gate, this implies that half of all amplitudes have to be transferred to another compute node. In JUQCS this exchange is implemented by swapping nonlocal qubits and local ones and keeping track of these swaps by updating the permutation of the N bit indices [2]. Nevertheless, if N is large, this swapping is a time-consuming operation, even if the inter-node communication network is very fast.

Operations such as the CNOT and Toffoli gate that only exchange two amplitudes can be implemented without having to exchange data through the inter-node communication network. As explained earlier, the primary design objective of the original JUQCS simulator [2] was to provide an environment for testing and optimizing the MPI communication scheme for a quantum spin dynamics simulator which requires the implementation of more complicated many-qubit gates. Therefore, the current version of JUQCS does not exploit the special structure of the CNOT or Toffoli gate. The MPI communication scheme that we use is, apart from its actual implementation, identical to the one described in Ref. [2] and will therefore not be discussed in detail here.

4. Trading memory for CPU time

The main factor limiting the size of the pen-and-paper quantum computer that can be simulated is the memory required to store the 2^N amplitudes of the vector $|\Phi\rangle$. In this section, we discuss two different methods to reduce the amount of memory needed. Evidently, this reduction comes at the price of an increase of computation time.

4.1. Double precision versus byte encoding

In quantum theory, the state of a single qubit is represented by two complex numbers ψ_0 and ψ_1 which are normalized such that $|\psi_0|^2 + |\psi_1|^2 = 1$ [1]. A gate operation on the qubit changes these numbers according to

$$\begin{pmatrix} \psi_0 \\ \psi_1 \end{pmatrix} \leftarrow U \begin{pmatrix} \psi_0 \\ \psi_1 \end{pmatrix}, \tag{6}$$

where U is a 2×2 unitary matrix. Gate operations involving n qubits correspond to (repeated) matrix–vector multiplications involving $2^n \times 2^n$ unitary matrices. As the number of arithmetic operations on the vector of complex amplitudes representing the state of the N-qubit systems grows exponentially with N, it may seem necessary to perform these operations with high numerical precision. Our implementation of JUQCS-E uses two 8-byte floating point numbers to encode one complex amplitude.

However, not all gates change the numerical representation of the state amplitudes. For instance, the X and CNOT gates only swap amplitudes, while the Hadamard gate arithmetically combines the two amplitudes. Therefore, we have explored various ways to encode the complex numbers with less than 16 bytes. An adaptive encoding scheme that we have found to perform quite well for quantum gate circuits is based on the polar representation $z = re^{i\theta}$ of the complex number z. We use one byte variable $-128 \leq b_1 < 128$ to encode the angle $-\pi \leq \theta < \pi$, i.e. $\theta = \pi b_1/128$. Another byte variable $-128 \leq b_0 < 128$ is used to represent r in the following manner. The special values $r=0$ and $r=1$ correspond to $b_0 = -128$ and $b_0 = 127$, respectively. The remaining values of $-127 \leq b_0 \leq 126$ are used to compute r according to $r = (b_0+127)(r_1-r_0)/253 + r_0$, where r_0 and r_1 are the minimum and maximum value of the z's with $0 < |z| < 1$ over all elements of the state vector. The values of r_0 and r_1 need to be updated to adaptively tune the encoding scheme to the particular quantum circuit being executed. Obviously, our encoding scheme reduces the amount of memory required to store the state by a factor of 8 at the expense of additional CPU time to perform the decoding–encoding procedure. The amount of additional CPU time depends on the gate and varies from very little for e.g. the X or CNOT gate to a factor of 3–4 for gates such as the Hadamard or +X gate.

4.2. Auxiliary variable method

An appealing feature of the universal quantum computation model is that only a few single-qubit gates and the CNOT gate suffice to perform an arbitrary quantum computation [1]. In other words, in principle, any unitary matrix can be written as a product of unitary matrices that involve only single-qubit and two-qubit operations.

This subsection demonstrates that any circuit involving single-qubit gates, controlled-phase-shifts, and CNOT operations can be expressed as a string of single-qubit operations, summed over a set of discrete, two-valued auxiliary variables. Each term in this sum can be computed in $\mathcal{O}(N)$ arithmetic operations. The number of auxiliary variables is exactly the same as the number P of controlled-phase-shifts or CNOT gates in the circuit. The worst case run time and memory usage of this algorithm are $\mathcal{O}(NM2^P)$ and $\mathcal{O}(N+M)$, respectively, where M is the number of output amplitudes desired. Clearly, if $M \ll 2^N$, the memory reduction from $\mathcal{O}(2^N)$ to $\mathcal{O}(N+M)$ bytes becomes very significant as the number of qubits N increases. To be effective, this approach requires that the input state to the circuit is a product state. However, this is hardly an obstacle because in the gate-based model of quantum computation, it is standard to assume that the N-qubit device can be prepared in the product state [1]

$$|0\rangle = |0\rangle_0 |0\rangle_1 \ldots |0\rangle_{N-1}, \tag{7}$$

where the subscripts refer to the individual qubits.

First, let us consider a string of single-qubit gates acting on qubit $j = 0, \ldots, N-1$ and denote the product of all the unitary matrices corresponding to these single-qubit gates by V_j. The application of these gates changes the initial state of qubit 0 into

$$V_j|0\rangle_j = \alpha_j|0\rangle_j + \beta_j|1\rangle_j, \tag{8}$$

where α_j and β_j are complex-valued numbers satisfying $|\alpha_j|^2 + |\beta_j|^2 = 1$. If $V = V_0 \otimes \cdots \otimes V_{N-1}$ represents a circuit that consists of single-qubit gates only, we have

$$V|0\rangle = \prod_{j=0}^{N-1} \left(\alpha_j|0\rangle_j + \beta_j|1\rangle_j\right). \tag{9}$$

From Eq. (9), it follows immediately that in practice, the right-hand side can be computed in $\mathscr{O}(N)$ arithmetic operations on a digital computer. More importantly, the amount of memory required to store the product state Eq. (9) is only 2^5N bytes (assuming 8-byte floating point arithmetic), much less (if $N > 7$) than the exponentially growing number 2^{N+4} required to store an arbitrary state.

Second, consider the results of applying to the state Eq. (9), a CNOT gate with control qubit 0 and target qubit 1. We have

$$\begin{aligned}\mathrm{CNOT}_{01}V|0\rangle = \Big(&\alpha_0\alpha_1|0\rangle_0|0\rangle_1 + \beta_0\alpha_1|1\rangle_0|1\rangle_1 \\ &+\alpha_0\beta_1|0\rangle_0|1\rangle_1 + \beta_0\beta_1|1\rangle_0|0\rangle_1\Big) \\ &\times\left(\prod_{j=2}^{N-1}\left(\alpha_j|0\rangle_j + \beta_j|1\rangle_j\right)\right),\end{aligned} \tag{10}$$

such that it is no longer possible to treat the coefficients of qubit 0 and 1 independently from each other. Of course, this is just a restatement, in computational terms, that the CNOT gate is a so-called "entangling" gate. It is not difficult to imagine that a circuit containing several CNOT (or controlled phase shift, Toffoli) gates that involve different qubits can create a state, such as the one created by the sequence of CNOT gates mentioned in Section 5.2, in which a single-qubit operation on one particular qubit changes the amplitudes of all basis states. Thus, any strategy to reduce the memory usage must deal with this aspect and must therefore "eliminate" the entangling gates.

A simple, effective method to express controlled phase shifts and CNOT gates in terms of single-qubit gates is to make use of the discrete Hubbard–Stratonovich transformation, originally introduced to perform quantum Monte Carlo simulations of the Hubbard model [10]. Consider the controlled phase shift operation defined by the unitary matrix

$$U_{01}(a) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{ia} \end{pmatrix} = e^{ia(1+\sigma_0^z\sigma_1^z-\sigma_0^z-\sigma_1^z)/4}, \tag{11}$$

where σ_0^z and σ_1^z are the z-components of the Pauli matrices representing qubit 0 and 1, respectively. Note that by convention, the computational basis is built from eigenstates of the z-components of the Pauli matrices [1]. When a takes values $\pm 2\pi/2^k$, the matrix in Eq. (11) is exactly the one which performs the conditional phase shifts in the quantum Fourier transform circuit; and when $a = \pi$, we have $H_1U_{01}(\pi)H_1 = \mathrm{CNOT}_{01}$ such that all CNOT gates can be expressed as a product of Hadamard gates and $U_{01}(\pi)$. Thus, without loss of generality, it is sufficient to consider $U_{01}(a)$ only.

Denoting the eigenvalues of σ_0^z and σ_1^z by σ_0 and σ_1, respectively, we note that $e^{ia\sigma_0\sigma_1/4}$ can only take two values and can be written as

$$e^{ia\sigma_0\sigma_1/4} = \frac{e^{-ia/4}}{2}\sum_{s=\pm 1} e^{ix(\sigma_0+\sigma_1)s} \quad , \quad \sigma_0, \sigma_1 = \pm 1, \tag{12}$$

where x is given by $\cos 2x = e^{ia/2}$. Therefore, we have

$$\begin{aligned}U_{01}(a) &= e^{ia(1+\sigma_0^z\sigma_1^z-\sigma_0^z-\sigma_1^z)/4} \\ &= \frac{1}{2}\sum_{s=\pm 1} e^{i(\sigma_0^z+\sigma_1^z)(xs-a/4)},\end{aligned} \tag{13}$$

and we have accomplished the task of writing the controlled phase shift Eq. (11) as a sum of products of two single-qubit operations each.

The final step is to introduce auxiliary variables $s_p = \pm 1$ for each of the $p = 1, \ldots, P$ controlled phase shifts (including those that originate from rewriting the CNOTs) that appear in the quantum circuit. Then, the result of applying the whole circuit to the initial state $|0\rangle$ can be written as

$$|\psi\rangle = \frac{1}{2^P}\sum_{s_1\ldots s_P=\pm 1}\prod_{j=1}^{N} W_j(s_1, \ldots, s_P)|0\rangle, \tag{14}$$

where $W_j(s_1, \ldots, s_p)$ is a concatenation of single-qubit operations on qubit j. The action of $W_j(s_1, \ldots, s_p)$ can be computed independently (and in parallel if desired) of the action of all other $W_{j'}(s_1, \ldots, s_p)$'s. In practice, for large N, the advantage of the auxiliary variable approach in terms of memory usage disappears if the application requires knowledge of the full state $|\psi\rangle$ but can be very substantial if knowledge of only a few of the 2^N amplitudes of $|\psi\rangle$ suffices.

5. Validation and benchmarking

The first step in validating the operation of JUQCS is to execute all kinds of quantum circuits, including circuits randomly generated from the set of all the gates in the instruction set, for a small ($N = 2$) to moderate ($N \approx 30$) number of qubits on PCs running Windows (7,10) and on Linux workstations. Validating the operation of JUQCS when it makes use of MPI, OpenMP or both is less trivial, in particular if the number of qubits is close to the limit of what can be simulated on a particular hardware platform. Of course, validation of real quantum computing devices is much more difficult. In contrast to a simulation on a digital computer where the full state of the quantum computer is known with high accuracy, the correct operation of real quantum computing devices must be inferred by sampling the amplitudes of the computational basis states, a daunting task if the number of qubits increases.

On a PC/workstation with, say 16 GB of memory, one can run small problems, i.e. those that involve not more than 29 (32 when JUQCS-A is used) qubits. Quantum circuits involving 45 or more qubits can only be tested on supercomputers such as the IBM Blue Gene/Q of the Jülich Supercomputing Center in Germany, the K computer of the RIKEN Center for Computational Science in Kobe, Japan, or the Sunway TaihuLight at the National Supercomputer Center in Wuxi in China.

Validating the operation of JUQCS requires circuits for which the exact input–output relation is known such that the correctness of the outcome can be easily verified. This section presents JUQCS results obtained by executing quantum circuits for which this is the case and, at the same time, illustrates the scaling and performance of JUQCS on the supercomputers listed in Table 1.

5.1. Uniform superposition

A common first step of a gate-based quantum algorithm is to turn the initial state (all qubits in state $|0\rangle$) into a uniform superposition by a sequence of Hadamard operations. Such a sequence has the nice feature that it can be trivially extended to more and more qubits and is therefore well-suited to test the weak scaling behavior of a universal quantum computer simulator. Note that

Table 2
The expectation values of the individual qubits, measured after performing a Hadamard operation on each of the N qubits as obtained by JUQCS-A and JUQCS-E. Recall that JUQCS-A uses a factor of 8 less memory than JUQCS-E but still yields the same numerically exact results as those produced by JUQCS-E for these tests. The JUQCS-A calculations were performed on JUQUEEN (up to $N = 46$ qubits), Sunway TaihuLight (up to $N = 48$ qubits), the K computer (up to $N = 48$ qubits), and JURECA (up to $N = 43$ qubits). The JUQCS-E calculations were performed on JUQUEEN (up to $N = 43$ qubits), Sunway TaihuLight (up to $N = 45$ qubits), the K computer (up to 45 qubits), JURECA (up to $N = 40$ qubits), and JUWELS (up to $N = 40$ qubits). The line beginning with '...' is a placeholder for the results of measuring qubits $1, \dots, N-2$.

	JUQCS-A			JUQCS-E		
qubit	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$
0	0.000	0.500	0.500	0.000	0.500	0.500
...	0.000	0.500	0.500	0.000	0.500	0.500
$N-1$	0.000	0.500	0.500	0.000	0.500	0.500

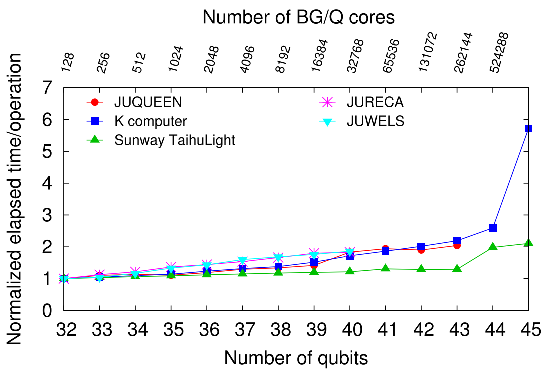


Fig. 1. The elapsed time per gate operation (normalized by the values 1.2 s (JUQUEEN), 1.0 s (K), 7.7 s (Sunway TaihuLight), 1.9 s (JURECA), and 1.3 s (JUWELS) for running the 32 qubit circuit) as a function of the number of qubits, as obtained by JUQCS-E executing a Hadamard operation on each qubit. This weak scaling plot shows that JUQCS-E beats the exponential scaling of the computational work by doubling the size of the machine with each added qubit.

independent of the number of qubits and computer architecture used, it is possible to construct the uniform superposition without any form of communication between nodes (a technique used by the SHORBOX instruction, see Section 5.4) but, as explained in Section 1, one of the design objectives of JUQCS was to test and benchmark the MPI communication, not to construct the most efficient simulator of a universal quantum computer tuned to specific hardware. Therefore, on purpose, we do not "optimize" the quantum circuit at this level.

Table 2 summarizes the results of executing such sequences of Hadamard operations on JUQCS-E for $N \leq 45$ and on JUQCS-A for $N \leq 48$.

In Figs. 1 and 2 we present the results of a weak scaling analysis of the elapsed times required to execute a Hadamard operation on each of the N qubits. Clearly, by doubling the size of the machine with each added qubit, JUQCS beats the exponential scaling of the computational work with the number of qubits.

5.2. Sequence of CNOT gates

Table 3 and Figs. 3 and 4 summarize the results obtained by executing the sequence of gate operations (H 0), (CNOT 0 1), (CNOT 1 2), ..., (CNOT N-2, N-1). The result of this quantum circuit is to put the quantum computer in the maximally entangled state $(|0\dots0\rangle + |1\dots1\rangle)/\sqrt{2}$. Also for these circuits, by doubling the size of the machine with each added qubit, JUQCS beats the exponential scaling of the computational work with the number of qubits, the salient feature of a gate-based universal quantum computer.

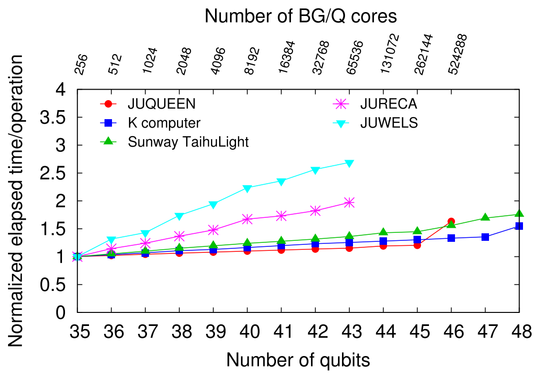


Fig. 2. The elapsed time per gate operation (normalized by the values 10.7 s (JUQUEEN), 30.8 s (K), 101.2 s (Sunway TaihuLight), 9.7 s (JURECA) and 4.7 s (JUWELS) for running the 35 qubit circuit) as a function of the number of qubits, as obtained by JUQCS-A executing a Hadamard operation on each qubit. This weak scaling plot shows that JUQCS-A beats the exponential scaling of the computational work by doubling the size of the machine with each qubit added. Note that JUQCS-A not only uses a factor of 8 less memory than JUQCS-E but also uses a factor of 8 less cores to run the same circuit.

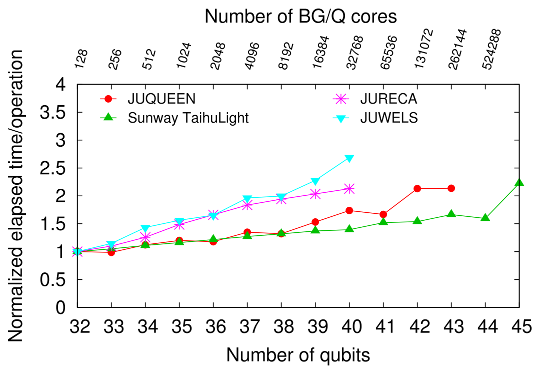


Fig. 3. The elapsed time per gate operation (normalized by the values 0.98 s (JUQUEEN), 5.1 s (Sunway TaihuLight), 1.4 s (JURECA), and 0.9 s (JUWELS) to run the 32 qubits circuit) as a function of the number of qubits, as obtained by JUQCS-E executing a Hadamard operation on qubit 0 and the sequence (CNOT 0 1), (CNOT 1 2), ..., (CNOT N-2, N-1), followed by a measurement of the expectation values of all the qubits. This weak scaling plot shows that JUQCS-E beats the exponential scaling of the computational work by doubling the size of the machine with each added qubit.

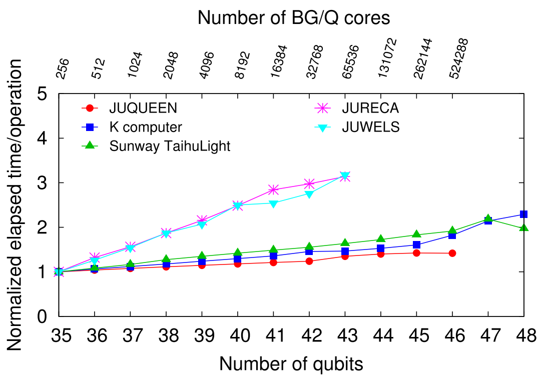


Fig. 4. The elapsed time per gate operation (normalized by the values 2.7 s (JUQUEEN), 3.8 s (K), 19.9 s (Sunway TaihuLight), 2.4 s (JURECA), and 2.2 s (JUWELS) to run the 35 qubits circuit) as a function of the number of qubits, obtained by JUQCS-A executing a Hadamard operation on qubit 0 and the sequence (CNOT 0 1), (CNOT 1 2), ..., (CNOT N-2, N-1). This weak scaling plot shows that JUQCS-A beats the exponential scaling of the computational work by doubling the size of the machine with each added qubit.

Table 3
The expectation values of the individual qubits, measured after performing the sequence (H 0), (CNOT 0 1), (CNOT 1 2), ..., (CNOT N-2, N-1), followed by a measurement of all N qubits. Recall that JUQCS-A uses a factor of 8 less memory than JUQCS-E but, for these tests, also yields the same numerically exact results as those produced by JUQCS-E. The JUQCS-A calculations were performed on JUQUEEN (up to $N = 46$ qubits), Sunway TaihuLight (up to $N = 48$ qubits), the K computer (up to $N = 48$ qubits), and JURECA (up to $N = 43$ qubits). The JUQCS-E calculations were performed on JUQUEEN (up to $N = 43$ qubits), Sunway TaihuLight (up to $N = 45$ qubits), the K computer (up to 45 qubits), JURECA (up to $N = 40$ qubits), and JUWELS (up to $N = 40$ qubits). The line beginning with '...' is a place holder for the results of measuring qubits $1, \ldots, N-2$.

	JUQCS-A			JUQCS-E		
qubit	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$
0	0.500	0.500	0.500	0.500	0.500	0.500
...	0.500	0.500	0.500	0.500	0.500	0.500
$N-1$	0.500	0.500	0.500	0.500	0.500	0.500

Performing single- and two-qubit gates requires only a little amount of computation per two or four basis states, respectively. Some gates, such as CNOT and X, only perform a permutation of the elements of the state. In other words, the arithmetic intensity of these operations is very low and the performance is mainly limited by the memory bandwidth. This explains that the weak scaling behavior of the circuit with the many CNOT's is slightly worse than that of the circuit involving Hadamard gates only. One strategy to overcome this limitation is to increase the arithmetic intensity by combining single- and two-qubit gates to multi-qubit (say 5-qubit) gates. As JUQCS is also a test bed for the simulator SPI12MPI for spin-1/2 models, we refrained from implementing this rather specialized strategy in the present version of JUQCS. While the weak scaling behavior on JURECA and JUWELS is rather good by itself, it is not as good as the ones on the other supercomputers used. This suggests that there may be some limitations in the bandwidth to the memory and network on JURECA and JUWELS.

The 2-byte encoding/decoding used by JUQCS-A to reduce the amount of required memory comes at the cost of larger computation time, affecting the ratio between computation and communications. This extra time depends on the type of quantum gate and ranges from almost zero (e.g. CNOT gate) to a factor of 3–4 (e.g Hadamard gate). As a result, comparing elapsed times of JUQCS-A and JUQCS-E only makes sense if we execute the same quantum circuit and even then, because of the factor-of-eight difference in memory usage, interpreting the differences in these elapsed times is not straightforward.

5.3. Adder circuit

The quantum circuit that performs the addition (modulo 2^K) of M integers [2,19,20], each represented by K qubits, provides a simple, scalable, and easy-to-verify algorithm to validate universal quantum computer simulators [20]. It involves a quantum Fourier transform [1] and primarily performs controlled-phase gates. We have constructed and executed quantum circuits that add up to five 9-bit integers and have run a sample of these circuits on the K computer and BG/Q.

Table 4 shows some representative results of a quantum circuit that adds two 19-bit integers. These results have been obtained by JUQUEEN using 8192 cores and 8192 MPI processes and took 1446 s for JUQCS-A and 388 s for JUQCS-E to complete. In this example, the values of the integers (210018 and 314269) are chosen such that their sum $(2^{19} - 1)$ corresponds to a binary number with 19 bits equal to one, which makes it very easy to verify the correctness of the result. If the quantum circuit works properly, the expectation values of the corresponding qubits should be equal to one. Clearly, Table 4 confirms that JUQCS-E works properly and also shows that the results of JUQCS-A are close but, in contrast to what the examples presented earlier might suggest, not always equal to the numerically exact results.

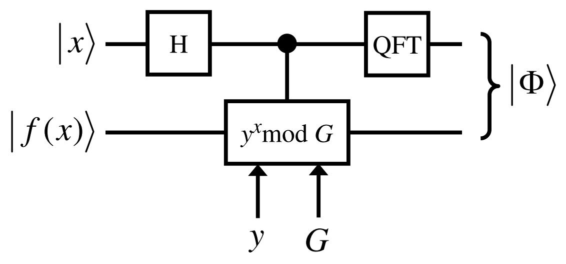


Fig. 5. Schematic diagram of Shor's algorithm.

5.4. Shor's algorithm on a 48 qubit quantum computer

For a detailed description of this algorithm, see Ref. [1,21]. Briefly, Shor's algorithm finds the prime factors p and q of a composite integer $G = p \times q$ by determining the period of the function $f(x) = y^x \bmod G$ for $x = 0, 1, \ldots$ Here, $1 < y < G$ should be coprime (greatest common divisor of y and G is 1) to G. If, by accident, y and G were not coprimes then $y = p$ or $y = q$ and there is no need to continue with the algorithm. Let r denote the period of $f(x)$, that is $f(x) = f(x+r)$. If the chosen value of y yields an odd period r, we repeat the algorithm with another choice for y, until we find an r that is even. Once we have found an even period r, we compute $y^{r/2} \bmod G$. If $y^{r/2} \neq \pm 1 \bmod G$, then we find the factors of G by calculating the greatest common divisors of $y^{r/2} \pm 1$ and G.

The schematic diagram of Shor's algorithm is shown in Fig. 5. The quantum computer has N qubits. There are two qubit registers: An x-register with X qubits to hold the values of x and a f-register with $F = N - X$ qubits to hold the values of $f(x) = y^x \bmod G$.

What is the largest number Shor's original algorithm can factorize on a quantum computer with N qubits? The number of qubits to represent $y^x \bmod G$ is $F = \lceil \log_2 G \rceil$. For Shor's algorithm to work properly, that is to find the correct period r of $f(x)$, the number of qubits X in the x-register should satisfy $G^2 \leq 2^X < 2G^2$ [21]. Omitting numbers G that can be written as a power of two (which are trivial to factorize), the minimum number of qubits of the x-register is $X = \lceil \log_2 G^2 \rceil$, so $N = X + F$ is either $3F$ or $3F - 1$. It follows that the maximum number of qubits that can be reserved for the f-register is given by $F = \lfloor (N+1)/3 \rfloor$, which determines the largest value of G. For example, on a 45-, or 46-qubit quantum computer $G = 32765$ is the largest integer composed of two primes that can be factorized by Shor's algorithm.

The SHORBOX instruction of JUQCS takes G and y as input, performs the Hadamard operations on all qubits of the x-register and also computes $f(x) = y^x \bmod G$ conditional on the qubits in the x-register and stores the result in the f-register. Application of the quantum Fourier transform on the x-register and sampling the state in the x-register produces numbers of basis states which can then be used to determine the period r and the factors p and q [1,21]. The task of JUQCS is to execute SHORBOX and perform the quantum Fourier transform.

In Table 5, we present the results for the case $G = 1007$ and $y = 529$, using 30 qubits, as obtained by running the JUQCS-A on a Lenovo W520 notebook. For comparison, we show the expectation values of the three components of the qubits as given by JUQCS together with the exact results calculated from the exact closed-form expression [2]. The results of JUQCS-A agree very well with the exact ones.

In Table 6, we present the results for the case $G = 32399$ and $y = 4295$, using 45 qubits, as obtained by running the JUQCS-A on JUQUEEN. For comparison, we show the expectation values of

Table 4
Results of summing two 19-bit integers (210018 and 314269) using a quantum adder circuit [2,19]. The BIT ASSIGNMENT instruction is used to interchange qubits (0–18) and (19–37) such that the sum of the integers (all 19 bits equal to one) is returned in qubits (0–18). This operation also reduces the amount of MPI communication. Recall that JUQCS-A uses a factor of 8 less memory than JUQCS-E and, as some of the numbers in the left three columns show, returns results that deviate slightly from the numerically exact results produced by JUQCS-E. Calculations were performed on JUQUEEN using 8192 cores and 8192 MPI processes and took 1446 s for JUQCS-A and 388 s for JUQCS-E to complete.

	JUQCS-A			JUQCS-E		
qubit	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$
0	0.500	0.500	0.999	0.500	0.500	1.000
1	0.500	0.504	0.999	0.500	0.500	1.000
2	0.504	0.502	0.999	0.500	0.500	1.000
3	0.500	0.500	0.999	0.500	0.500	1.000
4	0.504	0.501	0.999	0.500	0.500	1.000
5	0.504	0.499	1.000	0.500	0.500	1.000
6	0.500	0.500	0.999	0.500	0.500	1.000
7	0.500	0.500	1.000	0.500	0.500	1.000
8	0.500	0.500	1.000	0.500	0.500	1.000
9	0.500	0.500	1.000	0.500	0.500	1.000
10	0.500	0.500	1.000	0.500	0.500	1.000
11	0.500	0.500	1.000	0.500	0.500	1.000
12	0.500	0.500	1.000	0.500	0.500	1.000
13	0.500	0.500	1.000	0.500	0.500	1.000
14	0.500	0.500	1.000	0.500	0.500	1.000
15	0.500	0.500	1.000	0.500	0.500	1.000
16	0.489	0.497	1.000	0.500	0.500	1.000
17	0.500	0.500	1.000	0.500	0.500	1.000
18	0.500	0.500	1.000	0.500	0.500	1.000
19	0.500	0.500	0.000	0.500	0.500	0.000
20	0.500	0.500	1.000	0.500	0.500	1.000
21	0.500	0.500	1.000	0.500	0.500	1.000
22	0.500	0.500	0.000	0.500	0.500	0.000
23	0.500	0.500	0.000	0.500	0.500	0.000
24	0.500	0.500	1.000	0.500	0.500	1.000
25	0.500	0.500	1.000	0.500	0.500	1.000
26	0.500	0.500	0.000	0.500	0.500	0.000
27	0.500	0.500	1.000	0.500	0.500	1.000
28	0.500	0.500	0.000	0.500	0.500	0.000
29	0.500	0.500	0.000	0.500	0.500	0.000
30	0.500	0.500	0.000	0.500	0.500	0.000
31	0.500	0.500	1.000	0.500	0.500	1.000
32	0.500	0.500	1.000	0.500	0.500	1.000
33	0.500	0.500	0.000	0.500	0.500	0.000
34	0.500	0.500	0.000	0.500	0.500	0.000
35	0.500	0.500	0.000	0.500	0.500	0.000
36	0.500	0.500	1.000	0.500	0.500	1.000
37	0.500	0.500	0.000	0.500	0.500	0.000

Table 5
Representative results of running Shor's algorithm with JUQCS-A for a 30 qubit QC, $G = 1007 = 19 \times 53$ and $y = 529$, yielding a period $r = 18$. The three rightmost columns give the exact results, obtained from the closed-form expression [2] of the expectation values of the individual qubits. The expectation values produced by JUQCS-E are numerically exact and are therefore not shown. The number of qubits in the x-register is 20. The calculation used all 8 cores of a Lenovo W520 notebook (Windows 10) and took 348 s (elapsed time) to complete.

	JUQCS-A			Exact		
qubit	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$
0	0.504	0.495	0.500	0.500	0.500	0.500
1	0.502	0.497	0.500	0.500	0.500	0.500
2	0.502	0.499	0.500	0.500	0.500	0.500
3	0.502	0.499	0.445	0.500	0.500	0.445
4	0.504	0.500	0.445	0.500	0.500	0.445
5	0.506	0.500	0.445	0.500	0.500	0.445
6	0.501	0.500	0.444	0.500	0.500	0.444
7	0.500	0.500	0.444	0.500	0.500	0.444
8	0.500	0.500	0.444	0.500	0.500	0.444
9	0.500	0.500	0.444	0.500	0.500	0.444
10	0.500	0.500	0.444	0.500	0.500	0.444
11	0.500	0.500	0.444	0.500	0.500	0.444
12	0.500	0.500	0.444	0.500	0.500	0.444
13	0.501	0.500	0.444	0.500	0.500	0.444
14	0.501	0.500	0.444	0.500	0.500	0.444
15	0.500	0.500	0.444	0.500	0.500	0.444
16	0.501	0.500	0.444	0.500	0.500	0.444
17	0.500	0.500	0.444	0.500	0.500	0.444
18	0.501	0.499	0.444	0.500	0.500	0.444
19	0.500	0.500	0.500	0.500	0.500	0.500

the three components of the qubits as given by JUQCS together with the exact results calculated from the exact closed-form expression [2]. The results of JUQCS-A agree very well with the exact ones. We conjecture that a comparable accuracy of about 3 digits on the expectation values of single qubits ($\langle Qz(i)\rangle$) is beyond the reach of present [20] and future hardware realizations of gate-based quantum computers.

On a 48- or 49-qubit quantum computer, the largest composite integer $G = p \times q$ that can be factorized with Shor's original algorithm is $G = 65531 = 19 \times 3449$. We employed JUQCS-A to run Shor's algorithm on a 48 qubit universal quantum computer (simulator) for $G = 64507$ and $G = 65531$, requiring 32 qubits for the x-register and 16 qubits for the F register. On the Sunway TaihuLight, we made a run with $y = 21587$ and, after about 347 min of elapsed time, obtained a result that shows a period $r = 2$, which according to Shor's algorithm, yields the factorization $64507 = 251 \times 257$. On the K computer, a run with $y = 34888$ yielded, after 299 min of elapsed time, a result with period $r = 4$. According to Shor's algorithm this implies that $64507 = 251 \times 257$, in concert with the result obtained on the Sunway TaihuLight. Running Shor's algorithm with $G = 65531$ and $y = 1122$ on the K computer returned after 300 min of elapsed time, a result with period $r = 4$, in agreement with $G = 65531 = 19 \times 3449$.

6. Conclusion

The revised version of the massively parallel quantum computer simulator has been used to run a variety of quantum circuits on the Sunway TaihuLight, on the K computer, on an IBM Blue Gene/Q, and on Intel Xeon based clusters. Close-to-linear weak scaling of the elapsed time as a function of the number of qubits was observed on all computers used. This implies that the combination of software, many cores, and a fast communication network beats the exponential increase in memory and CPU time that is the characteristic of simulating quantum systems on a digital computer.

Two techniques for alleviating the memory problem have been discussed. The first employs an adaptive coding scheme to represent the quantum state in terms of 2-byte instead of 16-byte numbers. Benchmarks including Shor's algorithm, adders, quantum Fourier transforms, Hadamard and CNOT operations show that the factor-of-eight reduction in memory has no significant impact on the accuracy of the outcomes. This version can simulate a 32-qubit universal quantum computer on a notebook with 16 GB of memory.

The second technique resorts to a well-known method of Quantum Monte Carlo simulations to express two-qubit gates in terms of single-qubit gates and auxiliary variables. The worst case run time and memory usage of this algorithm was shown to be $\mathcal{O}(NM2^P)$ and $\mathcal{O}(N + M)$, respectively, where N is the number of qubits, P is the number of two-qubit gates and M is the number of output amplitudes desired. Although the reduction in memory can be huge if $M \ll N$, the technique is of limited practical use unless one knows how to choose M basis states of interest.

Through the specification of the sequence of quantum gates, the input to JUQCS can easily be tailored to put a heavy burden on the communication network, memory, processor or any combination of them. Therefore, the simulator described in this paper may be a useful addition to the suite of benchmarks for new high-performance computers. As mentioned in the introduction, the current version was designed to be portable over a wide range of computing platforms. However, the new generation of high-performance computers rely on accelerators or GPUs to deliver higher performance. For instance, the Sunway TaihuLight requires machine-specific programming to make use of the accelerator hardware. Adapting the code to make efficient use of GPUs or other kinds of accelerators is a challenging project that we leave for future work.

Acknowledgments

The authors thank Koen De Raedt for his help in improving the simulation code. The authors acknowledge the computing time granted by the JARA-HPC Vergabegremium and provided on the JARA-HPC Partition part of the supercomputer JUQUEEN [3] at the Forschungszentrum Jülich. D.W. is supported by the Initiative and Networking Fund of the Helmholtz Association, Germany through the Strategic Future Field of Research project “Scalable solid state quantum computing (ZT-0013)”. Part of the simulations reported in this paper were carried out on the K computer at RIKEN. S. Yuan gratefully acknowledges financial support from the Thousand Young Talent Plan and computational resources provided by National Supercomputing Center in Wuxi (China).

Appendix A. Instruction set

This appendix gives a detailed specification of each of the gate operations that are implemented in JUQCS. JUQCS will become accessible through a Jülich cloud service in 2019. A dockerized version of the four executables (JUQCS-E and JUQCS-A, MPI/OpenMP and MPI only) is available on request.

I gate

Description the I gate performs an identity operation on qubit n.

Syntax I n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

Operation $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

Graphical symbol $-\boxed{I}-$

Note The I gate is implemented as a “no operation” and is provided for compatibility with some other assembler-like language only.

H gate

Description the H gate performs a Hadamard operation on qubit n.

Syntax H n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

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

Graphical symbol $-\boxed{H}-$

X gate

Description the X gate performs a bit flip operation on qubit n.

Syntax X n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

Table 6
Representative results of running Shor's algorithm with JUQCS-A for a 45 qubit QC, $G = 32399 = 179 \times 181$ and $y = 4295$, yielding a period $r = 6$. The three rightmost columns give the exact results, obtained from the closed-form expression [2] of the expectation values of the individual qubits. The number of qubits on the x-register is 30. The calculation was performed on JUQUEEN, using 262144 cores, and took 5669 s of elapsed time to complete.

	JUQCS-A			Exact		
qubit	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$	$\langle Qx(i)\rangle$	$\langle Qy(i)\rangle$	$\langle Qz(i)\rangle$
0	0.499	0.480	0.502	0.500	0.500	0.500
1	0.579	0.486	0.375	0.500	0.500	0.375
2	0.494	0.494	0.345	0.500	0.500	0.344
3	0.518	0.498	0.336	0.500	0.500	0.336
4	0.496	0.498	0.335	0.500	0.500	0.334
5	0.503	0.500	0.333	0.500	0.500	0.333
6	0.499	0.499	0.335	0.500	0.500	0.333
7	0.502	0.500	0.333	0.500	0.500	0.333
8	0.500	0.500	0.334	0.500	0.500	0.333
9	0.500	0.500	0.333	0.500	0.500	0.333
10	0.500	0.500	0.334	0.500	0.500	0.333
11	0.501	0.500	0.333	0.500	0.500	0.333
12	0.500	0.500	0.335	0.500	0.500	0.333
13	0.499	0.500	0.333	0.500	0.500	0.333
14	0.501	0.500	0.335	0.500	0.500	0.333
15	0.499	0.500	0.333	0.500	0.500	0.333
16	0.501	0.500	0.335	0.500	0.500	0.333
17	0.499	0.500	0.333	0.500	0.500	0.333
18	0.500	0.500	0.335	0.500	0.500	0.333
19	0.499	0.500	0.333	0.500	0.500	0.333
20	0.501	0.500	0.335	0.500	0.500	0.333
21	0.499	0.500	0.333	0.500	0.500	0.333
22	0.501	0.500	0.335	0.500	0.500	0.333
23	0.499	0.500	0.333	0.500	0.500	0.333
24	0.500	0.500	0.335	0.500	0.500	0.333
25	0.499	0.500	0.332	0.500	0.500	0.333
26	0.501	0.500	0.335	0.500	0.500	0.333
27	0.500	0.500	0.332	0.500	0.500	0.333
28	0.500	0.500	0.335	0.500	0.500	0.333
29	0.500	0.500	0.500	0.500	0.500	0.500

Operation $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Graphical symbol $-\boxed{X}-$

Y gate

Description the Y gate performs a bit and phase flip operation on qubit n.

Syntax Y n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

Operation $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$

Graphical symbol $-\boxed{Y}-$

Z gate

Description the Z gate performs a phase flip operation on qubit n.

Syntax Z n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

Operation $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

Graphical symbol $-\boxed{Z}-$

S gate

Description the S gate rotates qubit n about the z-axis by $\pi/4$.

Syntax S n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

Operation $S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$

Graphical symbol $-\boxed{S}-$

S† gate

Description the S† gate rotates qubit n about the z-axis by $-\pi/4$

Syntax S+ n

Argument n is in the range $0, \ldots, N-1$ where N is the number of qubits.

Operation $S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$

Graphical symbol $-\boxed{S^\dagger}-$

T gate

Description the T gate rotates qubit n about the z-axis by $\pi/8$

Syntax T n

Argument n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation $T = \begin{pmatrix} 1 & 0 \\ 0 & (1+i)/\sqrt{2} \end{pmatrix}$

Graphical symbol $-\boxed{T}-$

T† gate

Description the T† gate rotates qubit n about the z-axis by $-\pi/8$

Syntax T+ n

Argument n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation $T^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & (1-i)/\sqrt{2} \end{pmatrix}$

Graphical symbol $-\boxed{T^\dagger}-$

U1 gate

Description the U1 gate performs a U1(λ) operation [16] on qubit n.

Syntax U1 n λ

Arguments n is an integer in the range $0, \dots, N-1$ where N is the number of qubits and λ is a number (floating point or integer) that represents an angle expressed in radians.

Operation $U1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$

Graphical symbol $-\boxed{U1(\lambda)}-$

U2 gate

Description the U2 gate performs a U2(ϕ, λ) operation [16] on qubit n.

Syntax U2 n ϕ λ

Arguments n is an integer in the range $0, \dots, N-1$ where N is the number of qubits and ϕ and λ are numbers (floating point or integer) that represent angles expressed in radians.

Operation $U2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix}$

Graphical symbol $-\boxed{U2(\phi, \lambda)}-$

U3 gate

Description the U3 gate performs a U3(θ, ϕ, λ) operation [16] on qubit n.

Syntax U3 n θ ϕ λ

Arguments n is an integer in the range $0, \dots, N-1$ where N is the number of qubits and θ, ϕ and λ are numbers (floating point or integer) that represent angles expressed in radians.

Operation $U3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix}$

Graphical symbol $-\boxed{U3(\theta, \phi, \lambda)}-$

+X gate

Description the +X gate rotates qubit n by $-\pi/2$ about the x-axis.

Syntax +X n

Argument n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation $+X = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix}$

Graphical symbol $-\boxed{+X}-$

-X gate

Description the -X gate rotates qubit n by $+\pi/2$ about the x-axis.

Syntax -X n

Argument n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation $-X = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}$

Graphical symbol $-\boxed{-X}-$

+Y gate

Description the +Y gate rotates qubit n by $-\pi/2$ about the y-axis.

Syntax +Y n

Argument n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation $+Y = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$

Graphical symbol $-\boxed{+Y}-$

-Y gate

Description the -Y gate rotates qubit n by $+\pi/2$ about the y-axis.

Syntax -Y n

Argument n is in the range $0, \dots, N-1$ where N is the number of qubits.

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

Graphical symbol $-\boxed{-Y}-$

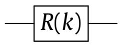
R(k) gate

Description the R(k) gate changes the phase of qubit n by an angle $2\pi/2^k$.

Syntax R n k

Arguments n is in the range $0, \dots, N-1$ where N is the number of qubits and k is a non-negative integer.

Operation $R(k) = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}$

Graphical symbol 

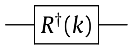
R†(k) gate

Description the inverse R(k) gate changes the phase of qubit n by an angle $-2\pi/2^k$.

Syntax R n $-k$

Arguments n is in the range $0,\ldots,N-1$ where N is the number of qubits and k is a non-negative integer.

Operation $R^\dagger(k) = \begin{pmatrix} 1 & 0 \\ 0 & e^{-2\pi i/2^k} \end{pmatrix}$

Graphical symbol 

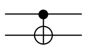
CNOT gate

Description the controlled-NOT gate flips the target qubit if the control qubit is 1.

Syntax CNOT *control target*

Arguments *control* $\neq$ *target* are integers in the range $0,\ldots,N-1$ where N is the number of qubits.

Operation $\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$; in the computational basis $|control, target\rangle$.

Graphical symbol 

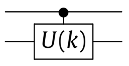
U(k) gate

Description the controlled-phase gate shifts the phase of the target qubit by an angle $2\pi/2^k$ if the control qubit is 1.

Syntax U *control target k*

Arguments *control* $\neq$ *target* are integers in the range $0,\ldots,N-1$ where N is the number of qubits and k is a non-negative integer.

Operation $U(k) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{2\pi i/2^k} \end{pmatrix}$; in the computational basis $|control, target\rangle$.

Graphical symbol 

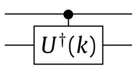
U†(k) gate

Description the U† gate shifts the phase of the target qubit by an angle $-2\pi/2^k$ if the control qubit is 1.

Syntax U *control target* $-k$

Arguments *control* $\neq$ *target* are integers in the range $0,\ldots,N-1$ where N is the number of qubits and k is a non-negative integer.

Operation $U^\dagger(k) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-2\pi i/2^k} \end{pmatrix}$ in the computational basis $|control, target\rangle$.

Graphical symbol 

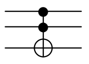
Toffoli gate

Description the TOFFOLI gate flips the target qubit if both control qubits are 1.

Syntax TOFFOLI *control*$_1$ *control*$_2$ *target*

Arguments *control*$_1$ $\neq$ *control*$_2$ $\neq$ *target* $\neq$ *control*$_1$ are integers in the range $0,\ldots,N-1$ where N is the number of qubits.

Operation $\text{TOFFOLI} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$ in the computational basis $|control_1, control_2, target\rangle$.

Graphical symbol 

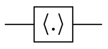
BEGIN MEASUREMENT

Description BEGIN MEASUREMENT computes and prints out the expectation values of all N qubits. This operation does not change the state of the quantum computer.

Syntax BEGIN MEASUREMENT

Arguments None

Operation In terms of their representation in terms of Pauli matrices, JUQCS computes $\langle Qx(n)\rangle = \langle\Psi|(1-\sigma_n^x)|\Psi\rangle/2$, $\langle Qy(n)\rangle = \langle\Psi|(1-\sigma_n^y)|\Psi\rangle/2$, and $\langle Qz(n)\rangle = \langle\Psi|(1-\sigma_n^z)|\Psi\rangle/2$ for $n = 0,\ldots,N-1$, where $|\Psi\rangle$ is the state of the quantum computer at the time that BEGIN MEASUREMENT was issued.

Graphical symbol 

GENERATE EVENTS

Description GENERATE EVENTS computes the probabilities of each of the basis states. It then uses random numbers to generate and print out the states according to these probabilities. This operation destroys the state of the quantum computer. It will force JUQCS to exit.

Syntax GENERATE EVENTS *events seed*

Arguments *events* is a positive integer, determining the number of events that will be generated and *seed* is an integer that is used as the initial seed for the random number generator if *seed* > 0. If *seed* ≤ 0, JUQCS uses as seed the value provided by the operating system.

Operation GENERATE EVENTS produces a list of *events* states, all sampled from the probability distribution computed from the current state of the quantum computer.

Graphical symbol none

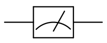
M

Description M performs a projective measurement on qubit n.

Syntax M n

Arguments n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation JUQCS first computes the probabilities p_0 and p_1 to observe qubit n in the state $|0\rangle$ and $|1\rangle$, respectively, Then JUQCS selects the measurement outcome 0 or 1 at random according to these probabilities and projects the qubit onto the corresponding state.

Graphical symbol 

QUBITS

Description QUBITS specifies the number of qubits of the universal quantum computer.

Syntax QUBITS N

Arguments N is an integer which must be larger than 1 and smaller than 64 (the actual number is limited by the available memory).

Note QUBITS N must be the first instruction.

BIT ASSIGNMENT

Description Applications that require MPI to run JUQCS on a distributed memory machine may benefit from renumbering the qubits such that the amount of MPI communications is reduced. The box below shows how this can be done without changing the original quantum circuit, for a simulation involving 4 qubits.

Syntax BIT ASSIGNMENT Permutation(0,1,...,N-1), see Example 3.

Arguments A list of integers in the range $0, \dots, N-1$ that is a permutation of the set $\{0, 1, \dots, N-1\}$.

```
QUBITS 4
BIT ASSIGNMENT 2 3 1 0
```

Note This instruction should appear after QUBITS N and before the first gate instruction.

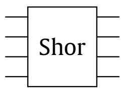
SHORBOX

Description SHORBOX initializes the x-register and the f-register in Shor's algorithm [1] to the state of uniform superposition and $y^x \bmod G$, respectively. Here G is the number to be factorized and $1 < y < G$ is chosen to be coprime to G. Subsequent application of the quantum Fourier transform to the x-register allows for the determination of the period of the function $f(x) = y^x \bmod G$ from which the factors of G may be calculated [1,2].

Syntax SHORBOX n_x G y

Arguments $n_x < N$ is the number of qubits reserved for the x-register, and G and y are integers. See Ref. [2] for details.

Operation SHORBOX $= 2^{-n_x/2} \sum_{x=0}^{2^{n_x}-1} |x\rangle |y^x \bmod G\rangle$.

Graphical symbol example for $n_x = 4$ 

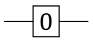
CLEAR

Description The CLEAR instruction projects the state of qubit n to $|0\rangle$.

Syntax CLEAR n

Arguments n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation CLEAR $= |0\rangle_n \langle 0|_n$

Graphical symbol 

Note This instruction fails if the projection results in to a state with amplitude zero.

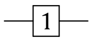
SET

Description The SET instruction projects the state of qubit n to $|1\rangle$.

Syntax SET n

Arguments n is in the range $0, \dots, N-1$ where N is the number of qubits.

Operation SET $= |1\rangle_n \langle 1|_n$

Graphical symbol 

Note This instruction fails if the projection results in to a state with amplitude zero.

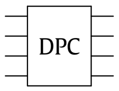
DEPOLARIZING CHANNEL

Description Insert X, Y, or Z gates with specified probabilities to mimic gate errors.

Syntax DEPOLARIZING CHANNEL P_X = p_x , P_Y = p_y , P_Z = p_z , SEED = k.

Arguments may appear in any order and any of them is optional. Missing arguments are assumed to have value zero. The values of the arguments should satisfy $0 \le p_x,\ p_y,\ p_z \le 1$ and $0 \le p_x+p_y+p_z \le 1$ and k must be a number smaller than $2^{31}-1$. If k is zero or negative, JUQCS takes the value provided by the operating system as the seed for the random number generator.

Operation After each gate operation, JUQCS performs an X gate on each qubit with probability p_x, a Y gate on each qubit with probability p_y, and a Z gate on each qubit with probability p_z.

Graphical symbol example for $N = 4$ 

Note 1 This instruction should appear after QUBITS N and before the first gate instruction.

Note 2 For an example see "**Example: input**". Removing '! ' from the second line instructs JUQCS to insert X, Y, or Z gates with specified probabilities.

EXIT

Description EXIT instruction terminates execution.

Syntax EXIT

Arguments None.

Operation The EXIT instruction forces JUQCS to measure all qubits and terminate. It can appear at any point in the instruction list. It is useful for debugging.

Appendix B. Illustrative example

In this section, we give a simple example that shows how JUQCS processes the input file with assembler-like instructions representing the quantum-gate circuit, in Fig. B.6 we show a circuit that uses qubits 3 and 4 to perform error-correction [1,22] on the logical qubit encoded in the qubits 0, 1, and 2. The list of JUQCS instructions is given in box "**Example: input**". For the definition of the mnemonics see Appendix A.

Example: input

```
 1  QUBITS        5
 2  ! DEPOLARIZING CHANNEL p_X =0.01, p_Y =0.01
 3
 4  H 0 ! initial state
 5  T 0
 6  H 0 ! initial state
 7
 8  CNOT 0 1 ! encode
 9  CNOT 0 2
10  BEGIN MEASUREMENT
11
12  !x 1 ! error
13  x 0 ! error
14
15  CNOT 0 3 ! correct
16  CNOT 1 3
17  CNOT 0 4
18  CNOT 2 4
19  M 3
20  M 4
21  TOFFOLI 3 4 0
22  X 4
23  TOFFOLI 3 4 1
24  X 3
25  X 4
26  TOFFOLI 3 4 2
27  X 3
28
29  H 3
30  CLEAR 3   ! must be preceeded by Hadamard
31  H 4
32  SET 4 ! must be preceeded by Hadamard
33
34  BEGIN MEASUREMENT
35  GENERATE EVENTS 8192   1
```

Lines 4 to 6 prepare (starting from the state with all qubits in state $|0\rangle$) a nontrivial input state to the error-correction circuit. Lines 8 and 9 encode the two physical qubits into logical ones. Line 10 instructs JUQCS to measure and print out the expectation values of the three components of the qubit (i.e. the expectation values of the Pauli matrices). Line 13 introduces a single-qubit error. Lines 19 and 20 instruct JUQCS to perform a projective measurement on qubits 3 and 4, respectively. Lines 15 to 27 are the instructions to detect and correct the error, if any. Lines 29 (31) and 30 (32) show the instructions to prepare qubit 3 (4), which has undergone a projective measurement in line 19 (20), for later re-use. Line 35 instructs JUQCS to generate an output file with 8192 events, meaning bit strings representing basis states, sampled from the final state of the quantum computer.

The relevant parts of the output, i.e. the expectation values of the three components of the five qubits in the initial and final state, when JUQCS is run with the file "**Example: input**" as input, (i) with line 13 commented out (no single-qubit error), (ii) with one single-qubit error on qubit 0, and (iii) with the '!' in line 12 removed (errors on qubits 0 and 1), are shown in "**Example: output (i)**", "**Example: output (ii)**", and "**Example: output (iii)**", respectively. Box "**Example: output (ii)**" demonstrates that the error-correction code indeed detects and corrects a single-qubit error while "**Example: output (iii)**" shows that it fails to correct two-qubit errors.

Example: output (i)

```
——i——<Qx( i )>————<Qy( i )>————<Qz( i )>—
  0  0.500E+00  0.500E+00  0.146E+00
  1  0.500E+00  0.500E+00  0.146E+00
  2  0.500E+00  0.500E+00  0.146E+00
  3  0.500E+00  0.500E+00  0.000E+00
  4  0.500E+00  0.500E+00  0.000E+00
—————————————————————————————————————————

——i——<Qx( i )>————<Qy( i )>————<Qz( i )>—
  0  0.500E+00  0.500E+00  0.146E+00
  1  0.500E+00  0.500E+00  0.146E+00
  2  0.500E+00  0.500E+00  0.146E+00
  3  0.500E+00  0.500E+00  0.000E+00
  4  0.500E+00  0.500E+00  0.100E+01
—————————————————————————————————————————
```

Example: output (ii)

```
——i——<Qx( i )>————<Qy( i )>————<Qz( i )>—
  0  0.500E+00  0.500E+00  0.146E+00
  1  0.500E+00  0.500E+00  0.146E+00
  2  0.500E+00  0.500E+00  0.146E+00
  3  0.500E+00  0.500E+00  0.000E+00
  4  0.500E+00  0.500E+00  0.000E+00
—————————————————————————————————————————

——i——<Qx( i )>————<Qy( i )>————<Qz( i )>—
  0  0.500E+00  0.500E+00  0.146E+00
  1  0.500E+00  0.500E+00  0.146E+00
  2  0.500E+00  0.500E+00  0.146E+00
  3  0.500E+00  0.500E+00  0.000E+00
  4  0.500E+00  0.500E+00  0.100E+01
—————————————————————————————————————————
```

Example: output (iii)

```
——i——<Qx( i )>————<Qy( i )>————<Qz( i )>—
  0  0.500E+00  0.500E+00  0.146E+00
  1  0.500E+00  0.500E+00  0.146E+00
  2  0.500E+00  0.500E+00  0.146E+00
  3  0.500E+00  0.500E+00  0.000E+00
  4  0.500E+00  0.500E+00  0.000E+00
—————————————————————————————————————————

——i——<Qx( i )>————<Qy( i )>————<Qz( i )>—
  0  0.500E+00  0.500E+00  0.854E+00
  1  0.500E+00  0.500E+00  0.854E+00
  2  0.500E+00  0.500E+00  0.854E+00
  3  0.500E+00  0.500E+00  0.000E+00
  4  0.500E+00  0.500E+00  0.100E+01
—————————————————————————————————————————
```

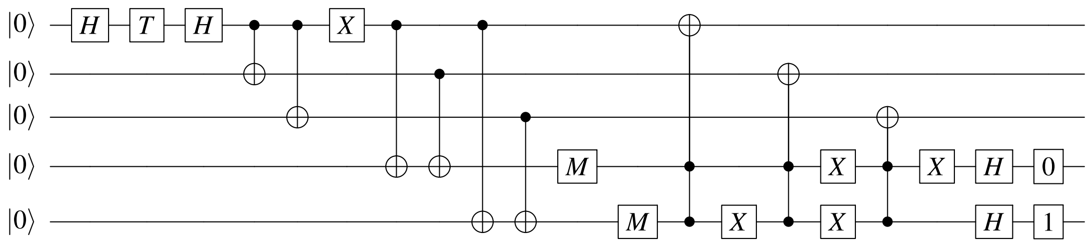


Fig. B.6. Quantum circuit performing error correction on the top three qubits. The corresponding JUQCS input file is listed in Example: input. Qubits are numbered from zero (top) to four (bottom). Reading from left to right, the first three gates prepare the initial state, the next two (CNOT) gates perform the encoding, the *X* gate on qubit 0 introduces a spin flip error, the next 11 gates detect and correct the error and the last 3 (2) gates on qubit 3 (4) illustrate how to reset a qubit to 0 (1).

References

[1] M. Nielsen, I. Chuang, Quantum Computation and Quantum Information, tenth ed., Cambridge University Press, Cambridge, 2010.
[2] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe, N. Ito, Comput. Phys. Comm. 176 (2007) 121–136.
[3] M. Stephan, J. Doctor, J. Large-Scale Res. Facil. 1 (2015) A1.
[4] D. Krause, J. Large-Scale Res. Facil. 2 (2016) A62.
[5] W. Zhang, J. Lin, W. Xu, H. Fu, G. Yang, Tsinghua Sci. Technol. 22 (2017) 675–681.
[6] H. De Raedt, K. Michielsen, in: M. Rieth, W. Schommers (Eds.), Handbook of Theoretical and Computational Nanotechnology, American Scientific Publishers, Los Angeles, 2006, pp. 2–48.
[7] A.A.-G.M. Smelyanskiy, N.P.D. Sawaya, qHiPSTER: The quantum high performance software testing environment, arXiv:1601.07195.
[8] N. Khammassi, I. Ashraf, X. Fu, C.G. Almudever, K. Bertels, Design, Automation Test in Europe Conference Exhibition, 2017, pp. 464–469.
[9] T. Häner, D.S. Steiger, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, 2017, pp. 33:1–33:10.
[10] J.E. Hirsch, Phys. Rev. B 28 (1983) 4059–4061.
[11] E. Pednault, J.A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, R. Wisnieff, Breaking the 49-qubit barrier in the simulation of quantum circuits, arXiv:1710.05867.
[12] S. Boixo, S.V. Isakov, V.N. Smelyanskiy, H. Neven, Simulation of low-depth quantum circuits as complex undirected graphical models, arXiv:1712.05384.
[13] Z. Chen, Q. Zhou, C. Xue, X. Yang, G. Guo, G. Guo, Sci. Bull. 63 (2018) 964–971, http://dx.doi.org/10.1016/j.scib.2018.06.007.
[14] S. Boixo, S.V. Isakov, V.N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M.J. Bremner, J.M. Martinis, H. Neven, Nat. Phys..
[15] IBM, Q Experience, 2016, http://www.research.ibm.com/quantum/.
[16] A.W. Cross, L.S. Bishop, J.A. Smolin, J.M. Gambetta, Open quantum assembly language, arXiv:1707.03429.
[17] H. De Raedt, K. Michielsen, A. Hams, S. Miyashita, K. Saito, Eur. Phys. J. B 27 (2002) 15–28.
[18] D. Willsch, M. Nocon, F. Jin, H. De Raedt, K. Michielsen, Phys. Rev. A 96 (2017) 062302.
[19] T.G. Draper, Addition on a quantum computer, arXiv:quant-ph/0008033.
[20] K. Michielsen, M. Nocon, D. Willsch, F. Jin, Th. Lippert, H. De Raedt, Comput. Phys. Comm. 220 (2017) 44–55.
[21] P. Shor, SIAM Rev. 41 (1999) 303.
[22] S.J. Devitt, W. Munro, K. Nemoto, Rep. Progr. Phys. 76 (2013) 076001.