nature research

https://doi.org/10.1038/s42254-020-00245-7

Supplementary information

Quantum programming languages

In the format provided by the authors and unedited

Q#

Listing 1. Q# code implementing quantum teleportation between two qubits. qsharp/teleport.qs

```
namespace Microsoft.Quantum.Samples {

    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Intrinsic;

    /// # Summary
    /// Prepares a Bell state given two qubits in a computational basis state.
    operation PrepareBellPair(left : Qubit, right : Qubit) : Unit is Adj + Ctl {
        H(left);
        CNOT(left, right);
    }

    /// # Summary
    /// Teleports the state of the 'msg' qubit to the given 'target' qubit,
    /// by temporarily using a 'helper' qubit as a resource.
    operation Teleport(msg : Qubit, target : Qubit) : Unit {
        // Q# supports local allocations of qubits.
        using (helper = Qubit()) {
            PrepareBellPair(helper, target);
            Adjoint PrepareBellPair(msg, helper);
            // We apply a Pauli correction conditional on the outcomes of
            // single-qubit measurements in computational basis.
            (M(msg)    == One ? Z | I)(target);
            (M(helper) == One ? X | I)(target);
        }
    }

    /// # Summary
    /// Executes a teleportation experiment.
    /// If it succeeds, then the returned measurement result is Zero.
    operation TeleportationExperiment(prep : (Qubit => Unit is Adj + Ctl)) : Result {
        // We allocate new qubits for the duration of the block.
        using ((msg, target) = (Qubit(), Qubit())){
            // We prepare the message to teleport using the given
            // preparation routine and teleport the message.
            prep(msg);
            Teleport(msg, target);
            // If the target qubit is in the intended state, this will map it to |0⟩.
            Adjoint prep(target);
            return M(target);
        }
    }

    /// # Summary
    /// Unit test to check that various states are teleported correctly.
    // The Test attribute defines the target on which the test will be executed.
    // "QuantumSimulator" indicates that the test will be executed on the full
    // state simulator.
    @Test("QuantumSimulator")
    operation TeleportTest() : Unit {
        // We want to execute the teleportation experiment for various messages.
        let messages = [H, X, T];
        for(rep in 1 .. 100) {
```

```
            // We run the teleporation experiment for each message.
            let results = ForEach(TeleportationExperiment, messages);
            // We check that each run returned Zero using unit testing tools.
            // 'Fact' will fail and print the give string if success is false.
            let success = All(IsResultZero, results);
            Fact(success, "Teleportation failed.");
        }
    }
}
```

OpenQASM and Qiskit

Listing 2. OpenQASM code implementing quantum teleportation between two qubits. `openqasm/teleport.qasm`

```
// All OpenQASM programs begin with a header indicating the language version.
OPENQASM 2.0;
include "qelib1.inc";

// We need to declare all quantum and classical registers that are going to be used.
// We will use qubits[0] as the qubit containing the message ('msg'), qubits[1] as
// the helper qubit ('helper'), and qubits[2] as the target qubit ('target').
qreg qubits[3];
// Branching conditional on measurements outcomes requires passing in a classical
// register, hence we construct a separate register for the x- and z-correction.
creg corrz[1];
creg corrx[1];
// We also need a classical register to store the measurement outcome that
// indicates whether the teleportation succeeded.
creg final[1];

// We prepare the state |+⟩ to teleport.
h qubits[0];

// Prepares a Bell state between the helper and the target qubit.
h qubits[1];
cx qubits[1], qubits[2];

// Perform the inverse operation on the message and helper qubit.
cx qubits[0], qubits[1];
h qubits[0];

// We measure the message and the helper qubit in the computational basis
// and store the results in the classical registers corrz and corrx.
measure qubits[0] -> corrz[0];
measure qubits[1] -> corrx[0];

// We apply a Pauli correction conditional on the stored outcomes of
// the single-qubit measurements in computational basis.
if (corrz==1) z qubits[2];
if (corrx==1) x qubits[2];

// If the target qubit is indeed in a |+⟩ state, this will map it to |0⟩.
h qubits[2];

// If the message is teleported successfully this measurement should always be 0.
measure qubits[2] -> final[0];
```

Listing 3. Execute OpenQASM programs using Qiskit. `openqasm/run_teleportation.py`

```
from qiskit import QuantumCircuit, BasicAer, execute
import argparse # used to process command line arguments
from teleportation_circuit import * # custom teleportation implementation in Qiskit

"""
Executes the given QuantumCircuit on the qasm_simulator and returns the histogram.
"""
def run_experiment(circuit):
    # Qiskit Aer allows to simulate circuits using one of several different backends.
```

```
    backend = BasicAer.get_backend('qasm_simulator')

    # We can send the constructed circuit to an Aer backend to get an object
    # that represents the asynchronous execution of our circuit.
    # We repeat the simulation 100 times, and return the histogram.
    job = execute(circuit, backend, shots=100)

    # The result of a job tells us whether the job completed successfully,
    # and all measurement results obtained during the job.
    result = job.result()
    return result.get_counts(circuit)


if __name__ == "__main__":
    # We do some minimal command line parsing to allow giving a qasm file as input.
    parser = argparse.ArgumentParser()
    parser.add_argument("—qasm", dest="qasm")
    args = parser.parse_args()

    if args.qasm:
        # Qiskit Terra allows to load an OpenQASM file and compile it to a circuit.
        print("Loading␣circuit␣from␣file␣" + args.qasm + ".")
        circuit = QuantumCircuit.from_qasm_file(args.qasm)
    else:
        # Alternatively, we can also construct the circuit in Qiskit.
        print("Running␣the␣circuit␣returned␣by␣teleportation_experiment.")
        # We choose to teleport a \ket{+} state,
        # but any function for the preparation routine could be passed in here.
        circuit = teleportation_experiment(lambda prep, q: prep.h(q))

    data = run_experiment(circuit)

    # We check whether the teleportation succeeded for all shots,
    # i.e. we check if the final measurement was always 0.
    is_correct = lambda key: key.startswith('0')
    success = all(map(is_correct, data.keys()))

    if success: print("\nTeleportation␣succeeded!")
    else: print("\nTeleportation␣failed.")

    # We additionally print the the histogram of the measured values for all shots.
    # and emit the executed circuit. A matrix representation cannot be printed
    # since the circuit is not unitary.
    print("\nFull␣histogram:")
    print(data)
    print("\nCircuit:")
    print(circuit)
```

Listing 4. Build circuit directly in Qiskit. openqasm/teleportation_circuit.py

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, BasicAer, execute

"""
Adds the gates to the given circuit to prepare a Bell state between the two qubits.
"""
def prepare_bell_pair(circuit, left, right):
    circuit.h(left)
    circuit.cx(left, right)
```

```
"""
Adds the gates to the given circuit that implement the adjoint of the sequence of
gates added by prepare_bell_pair.
"""
def adjoint_prepare_bell_pair(circuit, left, right):
    # Qiskit supports generating the adjoint for QuantumCircuit instances, but
    # in this case it is more convenient to just implement it as python function.
    circuit.cx(left, right)
    circuit.h(left)

"""
Adds the gates to the given circuit that teleport the state of the 'msg' qubit
to the given 'target' qubit using the 'helper' qubit as a resource, and
using the classical registers corrz and corrx to store the measurement results.
"""
def teleport(circuit, msg, helper, target, corrz, corrx):
    prepare_bell_pair(circuit, helper, target)
    adjoint_prepare_bell_pair(circuit, msg, helper)
    # Add the gates to measure the message and the helper qubit in the computational
    # basis and store the results in the classical registers corrz and corrx.
    circuit.measure(msg, corrz[0])
    circuit.measure(helper, corrx[0])
    # Add the gates to apply a Pauli correction conditional on the stored outcomes
    # of the single-qubit measurements in computational basis.
    # Note that c_if expects a ClassicalRegister, which is why we need two of them.
    circuit.z(target).c_if(corrz, 1)
    circuit.x(target).c_if(corrx, 1)

"""
Constructs and returns a QuantumCircuit instance that applies the given state
preparation routine to a single qubit.
"""
def prepare_message(prep):
    qubits = QuantumRegister(1, "qubits")
    circuit = QuantumCircuit(qubits, name="prep")
    prep(circuit, qubits[0])
    return circuit

"""
Constructs and returns a QuantumCircuit instance containing a teleportation experiment.
If the experiment succeeds, then the last classical bit in the circuit will be 0.
"""
def teleportation_experiment(prep):
    # We need to declare all quantum and classical registers that are going to be
    # used as part of the circuit before we can construct the circuit instance.
    qubits = QuantumRegister(3, "qubits")
    msg, helper, target = map(lambda idx: qubits[idx], range(3))
    # Branching conditional on measurements outcomes requires passing in a classical
    # register, hence we construct a separate register for the x- and z-correction.
    corrz = ClassicalRegister(1, "corrz")
    corrx = ClassicalRegister(1, "corrx")
    # We also need a classical register to store the measurement outcome that
    # indicates whether the teleportation succeeded.
    final = ClassicalRegister(1, "final")

    # We construct a separate subcircuit implementing the passed in state preparation
    # function such that we can easily invert it later.
    prep = prepare_message(prep)

    # We create a QuantumCircuit instance to which we then add gates.
```

```
circuit = QuantumCircuit(qubits, corrz, corrx, final, name="teleport")
# We plug the constructed subcircuit for preparing the message to teleport
# into the circuit and teleport the message.
circuit.append(prep.to_instruction(), [msg])
teleport(circuit, msg, helper, target, corrz, corrx)
# We apply the inverse of the message preparation circuit to the target qubit.
# If the target qubit is in the intended state, this will map it to |0⟩.
circuit.append(prep.inverse().to_instruction(), [target])
# If the message is teleported successfully this measurement should always be 0.
circuit.measure(target, final[0])
return circuit
```

Cirq

Listing 5. Cirq code implementing quantum teleportation. `cirq/teleport.py`

```
import cirq
from cirq.ops import *

"""
Prepares a Bell state given two qubits in a computational basis state.
"""
def prepare_bell_pair(left, right):
    yield H(left)
    yield CNOT(left, right)

"""
Teleports the state of the 'msg' qubit to the given 'target' qubit
using the 'helper' qubit as a resource.
"""
def teleport(msg, helper, target):
    yield prepare_bell_pair(helper, target)
    yield cirq.inverse(prepare_bell_pair(msg, helper))
    # We apply a Pauli correction in a coherent manner, as Cirq does not
    # support applying operations conditional on a measurements outcome.
    yield CZ(msg, target)
    yield CNOT(helper, target)

"""
Constructs and returns a Circuit instance containing a teleportation experiment.
If the experiment succeeds, then the final measurement of the target qubit will be 0.
"""
def teleportation_experiment(prep):
    # We will use three qubits on a line.
    msg, helper, target = map(cirq.LineQubit, range(3))

    # We create a Circuit instance to which we then add gates.
    circuit = cirq.Circuit()
    # We add the gates to prepare the message to teleport using the given
    # preparation routine and teleport the message.
    circuit.append(prep(msg))
    circuit.append(teleport(msg, helper, target))
    # If the target qubit is in the intended state, this will map it to |0⟩.
    circuit.append(cirq.inverse(prep(target)))
    circuit.append(measure(target, key="final"))
    return circuit

"""
Executes the given Circuit on the simulator and returns the histogram.
"""
def run_experiment(circuit):
    # We simulate circuits using the local simulator.
    backend = cirq.Simulator()
    # We repeat the simulation 100 times, and return the histogram.
    # The returned histogram contains the result of all measurements in all runs.
    result = backend.run(circuit, repetitions=100)
    return result.histogram(key="final")


if __name__ == "__main__":
    # We choose to teleport a \ket{+} state,
```

```
# but any adjointable state preparation routine could be passed in here.
circuit = teleportation_experiment(H)
data = run_experiment(circuit)

# We check whether the teleportation succeeded for all shots,
# i.e. we check if the final measurement was always 0.
is_correct = lambda key: key == 0
success = all(map(is_correct, data.keys()))

if success: print("\nTeleportation succeeded!")
else: print("\nTeleportation failed.")

# We additionally print the the histogram of the measured value for all shots,
# emit the executed circuit, and print the matrix representation of the circuit.
print("\nFull histogram:")
print(data)
print("\nCircuit:")
print(circuit)
print("\nMatrix representation:")
print(cirq.unitary(circuit))
```

Quipper

Listing 6. Quipper code implementing quantum teleportation between two qubits. `quipper/teleport.hs`

```
module Teleport where
import Quipper

-- Prepares a Bell state given two qubits in a computational basis state.
prepareEntangledPair :: (Qubit, Qubit) -> Circ (Qubit, Qubit)
prepareEntangledPair (left, right) = do
    gate_H_at left
    right <- qnot right `controlled` left
    return (left, right)

-- Teleports the state of the 'msg' qubit to the given 'target' qubit
-- using the 'helper' qubit as a resource.
teleport :: (Qubit, Qubit, Qubit) -> Circ (Qubit)
teleport (msg, helper, target) = do
    (helper, target) <- prepareEntangledPair (helper, target)
    -- The Adjoint of a Bell preparation is a Bell measurement.
    (msg, helper) <- reverse_simple prepareEntangledPair (msg, helper)
    -- We apply a Pauli correction based on the Bell measurement.
    (z, x) <- measure (msg, helper) -- measuring two qubits in computational basis
    target <- gate_X target `controlled` x -- X classically controlled on bit x
    target <- gate_Z target `controlled` z -- Z classically controlled on bit z
    cdiscard (z, x) -- discarding classical bits
    return target

-- Checks that the Teleport operation correctly teleports
-- half of an entangled pair.
teleportTest :: Circ Qubit
teleportTest = do
    with_ancilla $ \reference -> do
        (msg, helper, target) <- qinit (False, False, False)
        (reference, msg) <- prepareEntangledPair (reference, msg)
        target <- teleport (msg, helper, target)
        -- If the teleportation circuit is correct joint state
        -- of `reference` and `target` must be a Bell pair
        (reference, target) <-
            reverse_simple prepareEntangledPair (reference, target)
        -- Line below asserts that `target` is in state zero,
        qterm False target
        -- using with_ancilla asserts that `reference` is in state zero
        return target
```

Listing 7. Haskell code that instantiates the target machine for Listing 6. `quipper/run_teleportation.hs`

```
import Teleport
import System.Random
-- The core Quipper module provides functions like print_simple.
import Quipper
-- This module provides simulators for use with Quipper.
import Quipper.Libraries.Simulation -- simulation functions including run_generic

main :: IO ()
main = do

  -- Print out the circuit using the print_simple function.
```

```
print_simple ASCII teleportTest

-- Show the circuit in the previewer.
-- We haven't enabled any GUI features for compatibility with the Docker image,
-- but the following line can be uncommented when running outside the container.
-- print_simple Preview teleportTest

-- Show gate counts for the circuit.
print_simple GateCount teleportTest

-- Efficiently simulate the circuit using the Clifford simulator.
q <- run_clifford_generic teleportTest
print q

-- Simulate the circuit using the full state-vector simulator.
random_number_generator <- newStdGen
print $ run_generic random_number_generator (0.0 :: Double) teleportTest
```

Scaffold

Listing 8. Scaffold code implementing quantum teleportation. `scaffold/teleport.scaffold`

```
// Prepares a Bell state given two qubits in a computational basis state.
module PrepareBellPair(qbit left, qbit right) {
    H(left);
    CNOT(left, right);
}

// Implement the adjoint of the sequence of gates added by PrepareBellPair.
module AdjointPrepareBellPair(qbit left, qbit right) {
    CNOT(left, right);
    H(left);
}

// Teleports the state of the 'msg' qubit to the given 'target' qubit
// using the 'helper' qubit as a resource.
module Teleport(qbit msg, qbit helper, qbit target) {
    PrepareBellPair(helper, target);
    AdjointPrepareBellPair(msg, helper);
    if (MeasZ(msg))    { Z(target); }
    if (MeasZ(helper)) { X(target); }
}

int main() {
    // We will use qubits[0] as the qubit containing the message ('msg'), qubits[1]
    // as the helper qubit ('helper'), and qubits[2] as the target qubit ('target').
    // We assume that newly allocated qubits are in the |0⟩ state.
    qbit qubits[3];

    // We choose to teleport a \ket{+} state.
    // We initialize the message and teleport it to the target qubit.
    H(qubits[0]);
    Teleport(qubits[0], qubits[1], qubits[2]);

    // We measure the target qubit in X basis to verify if the teleportation worked.
    // If the teleportation succeeded, we return 1 and we return -1 otherwise.
    if (MeasX(qubits[2])) { return 1; }
    else { return -1; }
}
```