Multiple circuits in one flow

Contents

1 Implementation in qSOA®

As explained in QPath's® User Guide, ref.[1], one flow can be attached to multiple circuits at the same time. Lets see an example of how that works in qSOA®.

The process of implementing any algorithm in qSOA® is comprised of four steps:

1. Setting up qSOA® and selecting the quantum solution
2. Creating a circuit, in this case two, with the algorithm and assigning it to the solution
3. Introducing a circuit flow to control the number of launches of the algorithm
4. Executing the flow on different quantum devices.

As can be seen in qSOA's® manual, ref.[1], there are multiple ways to secure the connection depending on the context. Following the best practices of qSOA®, in this tutorial we are integrating the security in the code. Similarly, one can work with asynchronous or synchronous programming. Keeping in mind the purpose of this tutorial we will use the synchronous version.

1.1 Setting up qSOA®

Firstly, we import the SDK that has been previously installed, see ref.[1], and create the qSOA® workspace to work with.

[1]:
```
from QuantumPathQSOAPySDK import QSOAPlatform  # Import SDK
```

[2]:
```
# Create qSOA workspace, login manually
qsoa = QSOAPlatform()

username = 'username'
password = 'password' # password encrypted in SHA-256

authenticated = qsoa.authenticateEx(username, password)

print('Authentication completed:', authenticated)
```

```
Authentication completed: True
```

Then we review the existing solutions and select the one we are interested in.

[3]:
```
# Get catalogs
solutionList = qsoa.getQuantumSolutionList()
print("    ",solutionList)
idSolution = int(input("Select idSolution: "))
```

```
    {'12345': 'QS_GateTutorials'}
Select idSolution: 12345
```

1.1.1 Securing the connection

As has been said, qSOA® allows multiple business development contexts. Therefore, the user can secure the connection through a configuration file, *.qpath*, useful at a personal level, or in parameterized way, as it is done here.

1.2 Assigning a circuit to the solution

Once the solution has been selected, we must link it to the circuits that we are interested on implementing. Therefore, we first need to define the circuits and create them.

1.2.1 Defining the circuits

As we are working with a gates circuit, we can either enter the circuit in *visual language*, VL, or *intermediate language*, IL. In this case we are going to work with VL and, therefore, we can either enter the circuit as a *String* or as a *CircuitGates* object, which is what we are going to go for.

In order to define each circuit, we just need to write the gates we want to use, in the correspondent qubits.

For this tutorial the circuits are going to be very simple circuits. The first one will introduce a NOT gate to every qubit and measure it, and the second one will only measure. It is easy to see that the expected results are a set of ones, and a set of zeros.

[4]:
```
## Defining 1st circuit w/ Circuit Gates
circuit1 = qsoa.CircuitGates()
circuit1.h(list(range(4)))
circuit1.measure()

print(circuit1.getCircuitBody())
```

```
[['X', 'X', 'X', 'X'], ['Measure'], [1, 'Measure'], [1, 1, 'Measure'], [1, 1,
1, 'Measure']]
```

[5]:
```
## Defining 2st circuit w/ Circuit Gates
circuit2 = qsoa.CircuitGates()
circuit2.measure()

print(circuit2.getCircuitBody())
```

```
[['Measure'], [1, 'Measure'], [1, 1, 'Measure'], [1, 1, 1, 'Measure']]
```

Note: Every gate can be applied to a single qubit, a list of them, introduced as a list, or to every qubit in the circuit, using ().

1.2.2 Creating the circuits

In order to create the circuits we are going to use the *createAssetSync* function. This function receives the following fields as inputs:

- idSolution: to associate the circuit with the solution we have selected before.
- assetName: to set the name of the circuit.
- assetNamespace: to associate the circuit with a class of circuits that share something in common. In this case, we associate the circuit with a set of basic circuits.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the circuit we have previously defined.
- assetType: to select if we are working with a gates circuit or an annealing one.
- assetLevel: to select either visual language or intermediate language, according to the definition of the circuit.

[6]:
```
## Circuit creation 1
assetName = 'QC_qSOA_Circ1'
assetNamespace = 'Manual.Gates.MC1F'
assetDescription = 'Creating the first circuit from a flow, from qSOA'

assetBody = circuit1
assetType = 'GATES'
assetLevel = 'VL'

CircuitManagementResult1 = qsoa.createAssetSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetType, assetLevel)
```

[7]:
```
## Circuit creation 2
assetName = 'QC_qSOA_Circ2'
assetNamespace = 'Manual.Gates.MC1F'
assetDescription = 'Creating the second circuit from a flow, from qSOA'

assetBody = circuit2
assetType = 'GATES'
assetLevel = 'VL'

CircuitManagementResult2 = qsoa.createAssetSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetType, assetLevel)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAsset* function creates, compiles and transpiles the asset, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions work, we recommend using the asynchronous version, *createAsset*, and the *getAssetManagementResult* function.

1.3 Assigning a circuit flow to the circuits

The quantum flow is a box diagram that will allow us to control the number of runs of our algorithm. This is very useful, as the results in quantum computing have a certain probability associated with them. That means that the more times we run the algorithm, the more robust the results will be.

1.3.1 Defining the flow

As happened with the circuit, the flow can be written in VL or IL. If we choose to do it with VL then it can be entered as a *String* or a *CircuitFlow* object. Otherwise, it can only be entered as a *String*. For this example we are choosing VL and *CircuitFlow*.

In order to define a flow we need:

1. Starting node
2. Initializing node: usually set to 0
3. Circuit node: where we write the circuit we want lo launch
4. Repeat node: where the number of repetitions can be establish
5. End node
6. Links between each node we have created

[9]:
```
# Defining flow w/ CircuitFlow
flow = qsoa.CircuitFlow()
startNode = flow.startNode()
initNode = flow.initNode(0)
circ1 = flow.circuitNode('Manual.Gates.MC1F.QC_qSOA_Circ1')
circ2 = flow.circuitNode('Manual.Gates.MC1F.QC_qSOA_Circ2')
# Namespace + CircuitName
repeatNode = flow.repeatNode(1000)
endNode = flow.endNode()

flow.linkNodes(startNode, initNode)
flow.linkNodes(initNode, circ1)
flow.linkNodes(circ1, circ2)
flow.linkNodes(circ2, repeatNode)
flow.linkNodes(repeatNode, endNode)

print(flow.getFlowBody()['nodeDataArray'])
```

[9]:

```
{'class': 'go.GraphLinksModel', 'nodeDataArray': [{'category': 'Start', 'text':␣
↪'Start', 'key': -1, 'loc': ''}, {'category': 'Init', 'text': '0', 'key': -2,␣
↪'loc': ''}, {'category': 'Circuit', 'text': 'Manual.Gates.MC1F.QC_qSOA_Circ1',␣
↪'key': -3, 'loc': ''}, {'category': 'Circuit', 'text': 'Manual.Gates.MC1F.
↪QC_qSOA_Circ2', 'key': -4, 'loc': ''}, {'category': 'Repeat', 'text': '1000',␣
↪'key': -5, 'loc': ''}, {'category': 'End', 'text': 'End', 'key': -6, 'loc':␣
↪''}], 'linkDataArray': [{'from': -1, 'to': -2, 'points': []}, {'from': -2,␣
↪'to': -3, 'points': []}, {'from': -3, 'to': -4, 'points': []}, {'from': -4,␣
↪'to': -5, 'points': []}, {'from': -5, 'to': -6, 'points': []}]}
```

1.3.2 Creating the flow

In order to create the flow we are using *createAssetFlowSync*, although the *createAssetSync* function would also work. The inputs that this function requires are:

- idSolution: to associate the flow with the solution we have selected before.
- assetName: to set the name of the flow.
- assetNamespace: to associate the flow with a class of flows that share something in common. In this case, we associate the flow with a set of basic flows.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the flow we have previously defined.
- assetLevel: to select either VL or IL, according to the definition of the flow.
- publish: to select if we want to publish the flow on qSOA® or not.

[10]:
```
## Flow creation
assetName = 'QF_qSOA_MC1F'
assetNamespace = 'Manual.Gates.MC1F'
assetDescription = 'Creating the flow with multiple circuits from qSOA'
assetPublication = True

assetBody = flow
assetType = 'FLOW'
assetLevel = 'VL'

FlowManagementResult = qsoa.createAssetFlowSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetLevel,assetPublication)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAssetFlow* function creates, compiles and transpiles the flow, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAssetFlow*, and the *getAssetManagementResult* function.

1.4 Implementation of the algorithm

1.4.1 Executing the algorithm

We are now ready to execute the algorithm, so lets see on what platforms we can do so, and select the ones we are interested in.

[13]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
DeviceID = input("Select a device to run the flow in: ")
```

```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
Select a device to run the flow in: 14
```

Now, we proceed to run the quantum algorithm with the *runQuantumApplicationSync* function.

[14]:
```
exe_application = qsoa.runQuantumApplicationSync('NameTheTask', idSolution,␣
 ↪FlowID, DeviceID)
```

Note: Note that the synchronous version of this function is being used. This is because *runQuantumApplication* launches an execution, and the synchronous function waits for the execution to be done before moving on. If we do not know how long the execution is going to take (due to the execution itself or to the queue), or if it takes too long, the asynchronous version presents a clear advantage. However, for a better understanding of how this functions work, we recommend using the asynchronous version, *runQuantumApplication*.

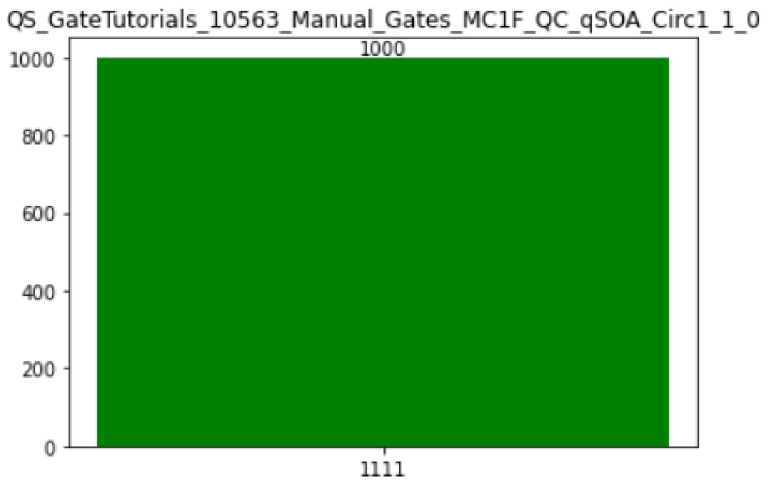
The function *runQuantumApplicationSync* gives an application object as output. For us to manage the results we need the *getQuantumExecutionResponse* function that returns the results as an execution object.

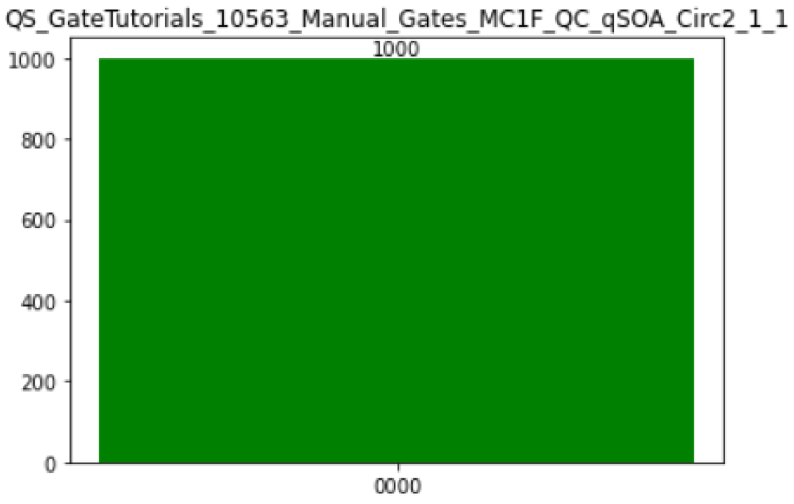
[15]:
```
restok_execution = qsoa.getQuantumExecutionResponse(exe_application.
 ↪getExecutionToken(), idSolution, FlowID)
restok_histogram = restok_execution.getHistogram()
print(restok_histogram)
```

```
{'QS_GateTutorials_12345_Manual_Gates_MC1F_QC_qSOA_Circ1_1_0':{'1111': 1000},
{'QS_GateTutorials_12345_Manual_Gates_MC1F_QC_qSOA_Circ2_1_1':{'0000': 1000}}
```

Now that we have the results we asked for, we can proceed to represent them with the function *representResults*.

[16]:
```
# Circuit gate representation
gates_representation = qsoa.representResults(restok_execution)
```

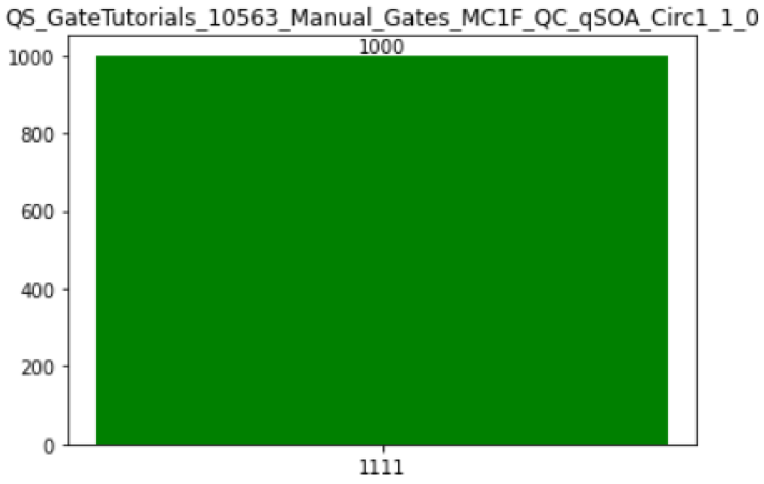




As expected, the obtained results from the circuits are 1111 and 0000, respectively.

Further more, the *representResults* function allows us to select which result we want to represent.

[17]:
```
# Circuit gate representation
gates_representation = qsoa.representResults(restok_execution, 0)
```



Executing the flow on multiple devices at the same time can be done by creating an array with all the information needed, and proceeding the same way.

References

[1] aQuantum, *QPath® Python SDK User Guide.* Available on QPath®.

[2] M. A. N. & I. L. Chuang, *Quantum Computation and Quantum Information,* Cambridge, 2010.