Coding a Quantum-HPC Hybrid Application with Python-based Workflow Orchestrator System on Supercomputer Fugaku

Soratouch Pornmaneerattanatri[1], Miwako Tsuji[1], Ketan Maheshwari[2], Mitsuhisa Sato[1]

[1]RIKEN R-CCS, Kobe, Japan,
[2]Oak Ridge National Laboratory, Tennessee, USA

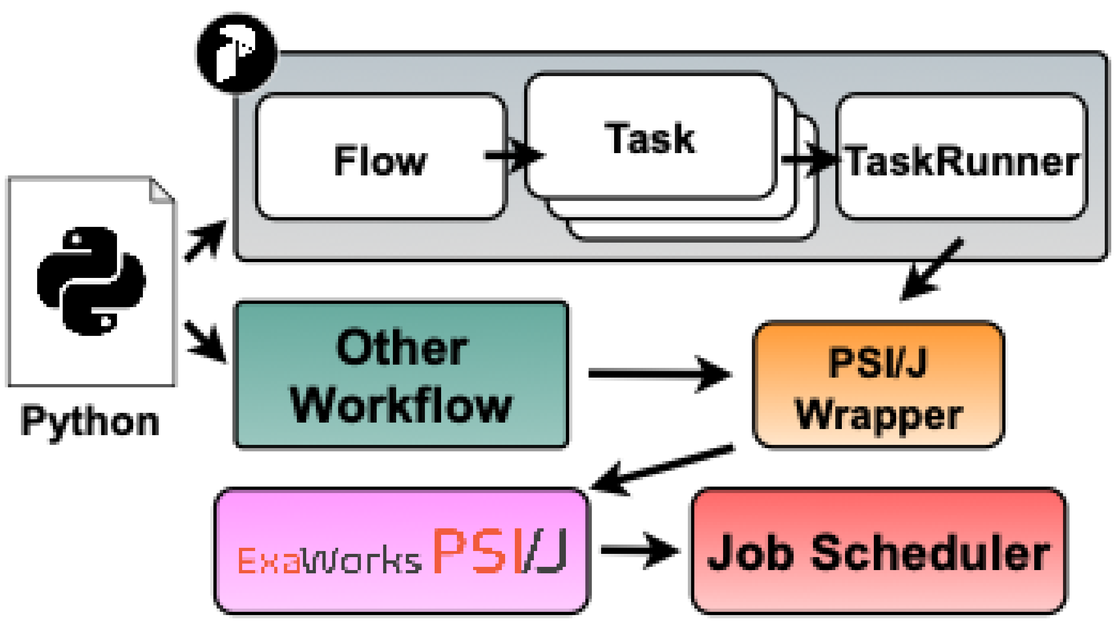
This poster is based on results obtained from a project, JPNP20017, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

Introduction

The improvement of Quantum Computer (QC) hardware in recent years has offered alternative solutions to some application previously limited by the classical algorithm. Quantum application cannot run purely on QC alone, it still cooperates with classical computer by preparing the input or quantum circuit, interpreting the output, and if the algorithm has further computing steps, the results from the QC will utilize in later steps. All quantum applications are quantum-classical hybrid applications in nature. However, running any simulator, it required the computing power of High-Performance Computer (HPC). Likewise, large quantum applications also require HPC to prepare a larger input, more complex quantum circuits, and calculate the post-processing task if necessary.

Normally, HPC resource management can only allocate one type of resource for one job. However, a Quantum-HPC hybrid application requires two types of computing resources. We developed a Python-based workflow orchestrator system to uncomplicated the job submission with multiple resource types. We developed this system for the Python-based application because major quantum libraries are Python-based, but the system is able to add the function to execute the non-Python-base binary and cooperate with the Python code. This workflow orchestrator system connects to the job scheduler of the supercomputer Fugaku via the PSI/J and the wrapper that we developed.

Python-based Workflow for Quantum-HPC Hybrid Application



Coding Quantum-HPC Hybrid Application with Python-based Workflow

```python
from prefect import task, flow
from prefect_psij import PSIJTaskRunner
from datetime import timedelta
import os

from ... import ...
import ...
```

```python
spec = {
  'executable': '/path/to/python/binary',
  'queue_name': 'resource_queue',
  'duration': timedelta( minutes = 30 ),
  'custom_attributes': {
      'group': 'group_name',
      'node_shape.node': '1'
}}
```

SPEC

```python
if __name__ == '__main__':
  df = main_flow_shor_algorithm()
  ...
```

1

```python
@flow
def main_flow_shor_algorithm();
  # Run task with 'with' keyword
  with PSIJTaskRunner( instance='pjsub', job_spec=spec ) as tr:
    qc_job = tr.submit(
      task = qc_task_func,
      parameters = {}
    )
    qc_result = qc_job.result()
  # Run task without 'with' keyword
  hpc_result = hpc_flow( qc_result, 8 )
  return hpc_result
```

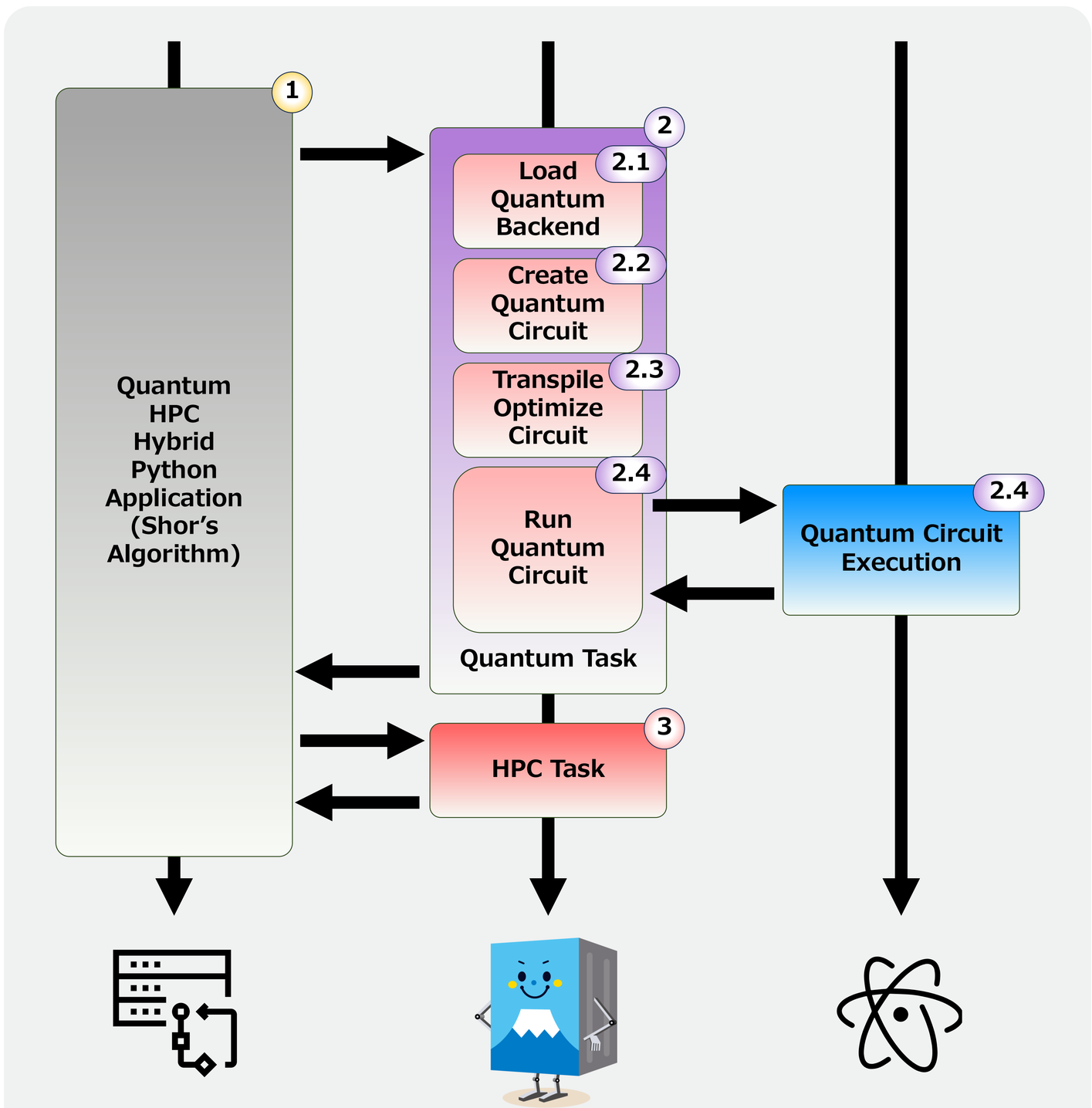
1 SPEC 2 3

```python
def shor_circuit( q, qubits ):
  qc = QuantumCircuit( qubits + 4, qubits )
  for i in range( qubits ):
    qc.h( i )
  qc.x( qubits )
  for i in range( qubits ):
    qc.append(c_amod15(a,2**i),[i]+[k + qubits for k in range(4)])
  qc.append( qft_dagger( qubits ), range( qubits ) )
  qc.measure( range( qubits ), range( qubits ) )
  return qc

@task
def qc_task_func():
  # Load quantum service
  service = QiskitRuntimeService( ... )
  backend = service.backend( 'ibm_xxx' )

  # Create quantum circuit
  qc = shor_circuit( a = 7, qubits = 8 )

  # Transpile and optimize the circuit
  pm = generate_preset_pass_manager(
    backend = backend, optimization_level = 1 )
  isa_qc = pm.run( qc )

  # Run circuit
  sampler = SamplerV2( mode = backend, options = {
'default_shots': 10000 } )
  result = samler.run( [ isa_qc ] ).result()
  return result
```

2 2.1 2.2 2.3 2.4

```python
@flow( task_runner = PSIJTaskRunner(
  instance = 'pjsub',
  job_spec = spec,
  work_directory = os.getcwd()
))
def hpc_flow( qc_result, qubits ):
  return hpc_task_func.submit( qc_result, qubits ).result()

@task
def hpc_task_func( qc_result, qubits ):
  counts = qc_result[0].join_data().get_counts()
  rows = []
  for output in counts:
    decimal = int( output, 2 )
    phase = decimal / ( 2**qubits )
    frac = Fraction( phase ).limit_denominator( 15 )
    rows.append([
      f"{output}(bin) = {decimal}(dec)",
      f"{decimal}/{2**qubits} = {phase:.5f}",
      f"{frac.numerator}/{frac.denominator}",
      f"{frac.denominator}"
    ])
    headers = ["Register Output","Phase","Fraction","Guess for r"]
    return pd.DataFrame( rows, columns=headers )
```

SPEC 3



Workflow Initialize Script GitHub Repository

Shor's Algorithm Source code