When Functional Programming Meets Parsl

Osama Almurshed

What is functional programming (FP)

- Focusing on the transformations that need to be applied to the data in order to produce the desired output.
  - Rather than focusing on the individual steps of an algorithm.
- Functional programming,
  - where less procedural syntax occurs in the implementation.
  - Python allows use to use FP, OOP, procedural
- Concepts
  - Function as a variable
  - Create a new function out of other functions (Higher order functions)
  - Functions pipeline

Why we should use FP with Parsl

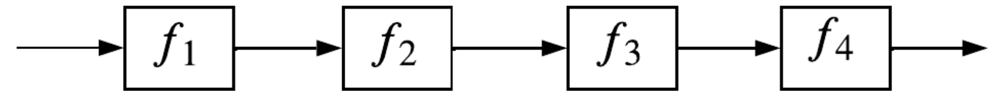
- `@python_app` of Parsl is a function that accept a function in the arguments.
- Parsl write task logics within python functions

```
@python_app(executors=['location_1'])
def test():
  return 0
```

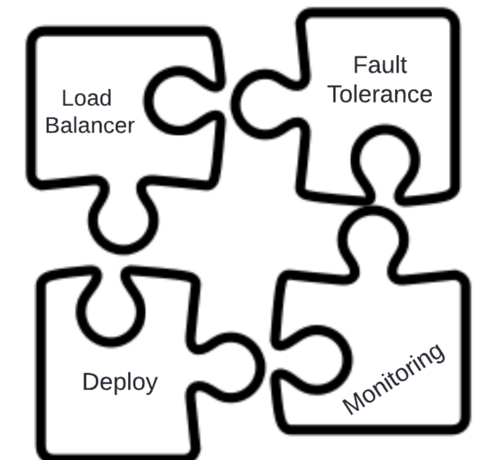
```
def test_():
  return 0

test = python_app(test_, executors=['location_1'])
```

Problem we try to solve

- Handle workflow inform of graph
  - A data model represent the graph in Python



- Dealing with problems that has graph in their implementation
  - Software Engendering
    - Install software packages with respect to the dependency between them
  - Distributed Deep Neural Network
    - Neural Network layers need to be running in orders
- Custom operations
  - Create a new operation
  - Utilize `parsl.python_app` to create utility function.



Functions’ type

- Types
  - Utility functions
  - Application functions
- Utility functions takes application functions as variables
  - Applications’ functions passed in data model then operate on it.

Utility Functions

- Is an extension of `parsl.python_app`
- Does the operation managements,
  - such as mapping function to resources
  - Or, monitoring the function deployment

```
# `*args`, `**kwargs` are the arguments of `function`
def place_function(location, function, *args, **kwargs):
  """ Deploy any python function as `parsl.python_app` """

  # create python_app that executes `function` in `location`
  execute_function = python_app(function, executors=[ location ])
  return execute_function(*args, **kwargs)
```

```
@python_app(executors=['ThreadPoolExecutor'])
def time_it(place_function, function, *args, **kwargs):
  locations = kwargs['locations']
  start = time()
  result = place_function(locations, function, *args, **kwargs).results()
  end = time()
  total_time = end - start
  return total_time, result
```

Application Functions Pipeline

```
# create functions
def adding(xs):
  return sum(xs)

def doubling(x):
  return x*2

def tripling(x):
  return x*3

# workflow: --> adding -->  doubling --> tripling --> doubling --> tripling -->
workflow = [adding, doubling, tripling, doubling, tripling]
```

Round Robin Load balancer

```
def round_robin(workflow, locations, place_function):
  """  distributing a set of functions over a set of locations using """

  number_location = len(locations)
  curr_location_index = 0

  for curr_function in workflow:
    # curr_location is hold the label of the parsl executor, i.e., location,
    curr_location = locations[ curr_location_index ]
    curr_input = place_function( curr_location, curr_function, curr_input )
    # `%` operation is Modular arithmetic returns the remainder or signed remainder of a division
      # numbers "wrap around" when reaching a certain value which is in this case `number_location - 1`
    curr_location_index = (curr_location_index + 1) % number_location

  last_function_output = curr_input
  return last_function_output
```

Conclusion

- Build a distributed operators that manages a graph of tasks
  - Build new functionality on top of a previous one.
- Writing the logic of our system in less code line.
  - Make it easer to debug.
- Separate application’s functions logic for the management logic
  - For example, separate function deployment from the act of deployment.

Thank you

AlmurshedO@Cardiff.ac.uk