Virtual Earthquake and seismology Research Community e-science environment in Europe
Project 283543 – FP7-INFRASTRUCTURES-2011-2 – www.verce.eu – info@verce.eu

dispel4py: A Python Framework for Data-Intensive Scientific Computing

(dispel4py training for OSDC workshop)
session 6 - part I

9 June 2015, Amsterdam

Outline

- What is dispel4py
- What is a stream
- What is a processing element (PE)
- What is a instance
- What is a graph
- What I need for constructing a dispel4py workflow
- Extra material

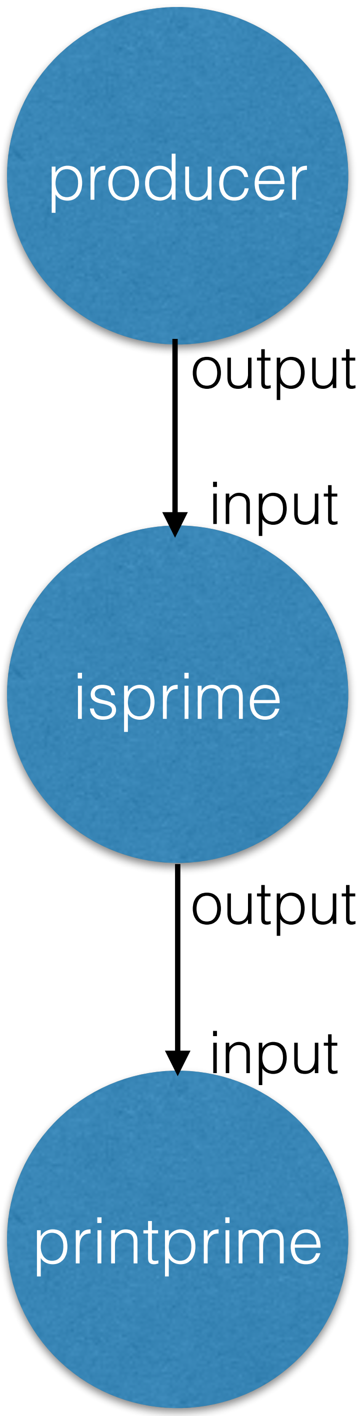
What is dispel4py

- dispel4py for **distributed data-intensive applications**
- Describes data-flow and processing elements using **Python**
- It enables abstract description of methods
- dispel4py **maps** to multiple enactment systems
- Applications **scale** automatically
  - exploiting parallel processing, clusters, grids and clouds
- dispel4py is **dataflow-oriented**
  - rather than control-oriented
  - no explicit specification of data movement
  - light-weight composition of data operations

What is a data stream

- A **stream** is a sequence of data units:
    - from external source
    - between data operations - Processing Elements (PEs)
    - to external destination
- Flow of input or output data between PEs
- Processes data from a source and delivers data to one or more destinations

What is a processing element (PE)



- Computational activity encapsulates
  - algorithm
  - services
  - data transformation processes
- Basic computational elements of dispel4py workflows
- PEs have:
  - inputs & outputs
  - computational activity.
- PEs are connected by streams
  - saves computational costs

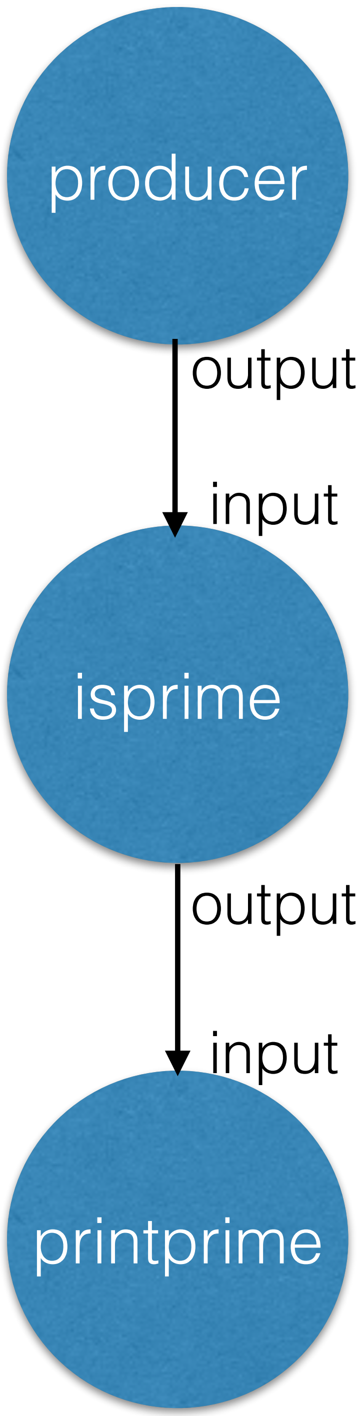
What is a graph

- How the PEs are connected
- How data is streamed
- The topology of the data flow
- No limitations on the type of graphs

What I need for constructing a dispel4py workflow

- You **only** have to **implement PEs** (in Python) and connect them:
  - Learn how to implement PEs.
  - Learn how to connect them.

Learning dispel4py by an example



- dispel4py workflow that generates a random integer number
- the number is checked if it is prime or not
- in case the number is prime a message is printed out

How to implement a PE

- Each PE specifies:
  - input & output connections
  - computational activity for processing data units
    - implement the "_process" method.

Types of PEs

Type	Inputs	Outputs	When to use it
GenericPE	*n* inputs	*m* outputs	many inputs and/or many outputs
IterativePE	1 input named 'input'	1 output named 'output'	process one and produce one data unit in each iteration
Consumer PE	1 input named 'input'	no output	no output and one input
ProducerPE	no input	1 input named 'output'	no inputs and one output; usually the root in a graph
Simple FunctionPE	1 input named 'input'	1 output named 'output'	only implement _process method; it can not store state between calls
create_iterative_chain	1 input named 'input'	1 output named 'output'	pipeline of functions processing sequentially; creates a composite PE

ProducerPE example

```
import random
from dispel4py.base import ProducerPE

class NumberProducer(ProducerPE):
    def __init__(self):
        ProducerPE.__init__(self)

    def _process(self):
        # this PE produces one input
        num= random.randint(1, 1000)
        return num
```

This PE produces a random number from 1 to 1000 and writes it to the output stream (‘output’)

What we have learnt:

- We don’t need to specify the output
- It does not receive any input.
- _process returns the value that is written to the output stream

IterativePE example

```
from dispel4py.base import IterativePE

class IsPrime(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, num):
        # this PE consumes one input and produces one output
        if all(num % i != 0 for i in range(2, num)):
            return num
```

This PE also receives a number (‘input’) and returns one output (‘output’) in case the number is prime.

What we have learnt:

- We don’t need to specify the input and output
- The parameter to the _process method is the data (the number)
- _process returns the value that is written to the output stream

ConsumerPE example

```
from dispel4py.base import ConsumerPE

class PrintPrime(ConsumerPE):
    def __init__(self):
        ConsumerPE.__init__(self)
    def _process(self, num):
        # this PE consumes one input
        self.log("the num %s is prime" % num)
```

This PE receives one input and prints it.

What we have learnt:

- We don’t need to specify the input
- It does not return any output.

How to connect PEs: Create a graph

- **Create the PEs**

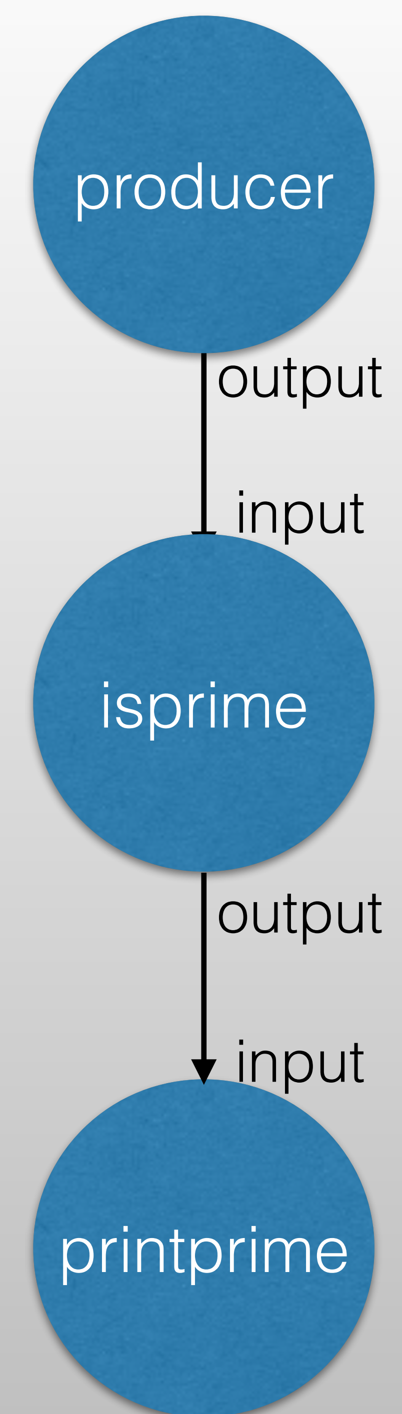
```
producer = NumberProducer()
isprime = IsPrime()
printprime = PrintPrime()
```

- **Create the graph and connect the PEs**

```
from dispel4py.workflow_graph import WorkflowGraph

graph = WorkflowGraph()
graph.connect(producer, 'output', isprime, ‘input')
graph.connect(isprime, 'output', printprime, 'input')
```

Example- Summary

```
from dispel4py.base import ProducerPE, IterativePE, ConsumerPE
from dispel4py.workflow_graph import WorkflowGraph
import random

class NumberProducer(ProducerPE):
    def __init__(self):
        ProducerPE.__init__(self)

    def _process(self):
        # this PE produces one input
        result= random.randint(1, 1000)
        return result

class IsPrime(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, num):
        # this PE consumes one input and produces one output
        self.log(“before checking data - %s - is prime or not“ % num)
        if all(num % i != 0 for i in range(2, num)):
            return num

class PrintPrime(ConsumerPE):
    def __init__(self):
        ConsumerPE.__init__(self)
    def _process(self, num):
        # this PE consumes one input
        self.log("the num %s is prime" % num)

producer = NumberProducer()
isprime = IsPrime()
printprime = PrintPrime()

graph = WorkflowGraph()
graph.connect(producer, 'output', isprime, ‘input')
graph.connect(isprime, 'output', printprime, ‘input')
```



Material

- Exercises: http://effort.is.ed.ac.uk/osdc2015/osdc2015_dispel4py_exercises.tar
- Slides: http://effort.is.ed.ac.uk/osdc2015/osdc2015_dispel4py_slides.tar

Extra Material

- GenericPE example
- SimpleFunctionPE example
- creative_iterative_chain
- How to create those three new PE types
- What does connecting PEs really mean?

GenericPE example

```
from dispel4py.core import GenericPE

class IsPrimeBis(GenericPE):
    def __init__(self):
        GenericPE.__init__(self)
        self._add_input('input')
        self._add_output('output_prime')
        self._add_output('output_total')
        self.cont = 0
    def process(self, inputs):
        num = inputs['input']
        # this PE consumes one input and can return two outputs or nothing.
        if all(num%i!=0 for i in range(2,num)):
            self.cont +=1
            return {'output_prime':num, 'output_total':self.cont}
```

This PE reads a number (input) and can returns two outputs: number (output_prime) and the total number of primes (output_total).

What we have learnt:

- We can add several outputs with different names
- The process method gets values from the input streams
- The process method returns both streams

SimpleFunctionPE example

```
from dispel4py.base import create_iterative_chain

def is_prime(num):
    if all(num%i!=0 for i in range(2,num)):
            return num
#For using this function as a PE we need to use 'SimpleFunctionPE' before
defining the graph:

isPrime = SimpleFunctionPE(is_prime)
```

This PE will emit a number if it is prime.

What we have learnt:

- Only implement the processing function
- The easiest but the most restrictive way
- The **function cannot store state between calls**; for example you can't implement SUM or AVG with it
- 1 input called 'input', 1 output called 'output'.

create_iterative_chain example

```
from dispel4py.base import create_iterative_chain
def add_value(num, value):
    num += value
    return num
def subtract_value(num, value):
     num -= value
     return num
def change_polarity(num):
    num *= -1
    return num

# For using this function as a PE we need to use 'creative_iterative_chain' before defining the graph.
preprocessData = create_iterative_chain([(add_value,{'value':33}), (subtract_value, {'value':5}), change_polarity])
```

What we have learnt:

- We can create a *composite* PE which processes several function in a sequence - more on composite PEs later!
- It creates a pipeline of SimpleFunctionPEs
- It's the easiest way to create a pipeline but the most restrictive
- 1 input called 'input', 1 output called 'output'.

How to connect PEs: Create a PE object

- Create a PE (could be GenericPE, IterativePE, ConsumerPE, ProducerPE)

```
isPrime = IsPrime()
```

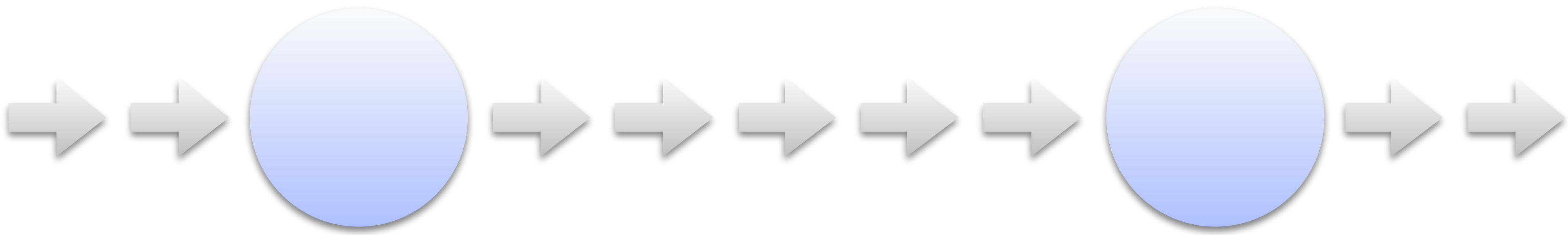
- Create a function wrapped in a simple PE

```
isPrime = SimpleFunctionPE(is_prime)
```

- Create a composite PE with a pipeline

```
preprocessData =
    create_iterative_chain([(add_value,{'value':33}), (subtract_value, {'value':5}), change_polarity])
```

How to connect PEs What does it mean



- PEs process a small amount of data at a time
- Data need not be explicitly stored
- PEs may store a small amount of result data (e.g. stacking) or big amount (if you have the resources)