Introduction to python for HPC

11th Advanced School in parallel computing

Bologna 2015

m.cestari@cineca.it

In/out

- What's in this course...
  - Basic of python language
  - Basic of mpi4py
  - Learning through examples

- What is not...
  - Any type of python “acceleration” (yes python can be really slow)

In/out

- Often, with HPC, people mean improving python performance
- Our python course cover this topic
- We will focus on python as an instrument to be used in massively parallel system

Why python

- Python has gained a lot of momentum in scientific computation
  - It's easy to learn the basics
  - It's very powerful (modern language)
  - can be coupled with good plotting tool

- In your scientific work sooner or later you'll come across to a python script

Why python / 2

- In HPC, python:
  - can be used as a glue for traditional (compiled) languages
  - can be used for quick prototyping
  - can be used to create ad hoc work-flows (i.e. by interfacing with the scheduling system)

- future employment in massiviley parallel system:
  - managing ensamble simulations
  - fault tolerance (layer between scheduler and simulations

Goal

Develop a small python program that runs multiple serial execution with different load balancing techniques applied

Goal / 2

“Hey I can do that!!”

(learn how to start with python development)

python

language introduction

Interpreter

- **Ipython:**
  - **enhance** the prompt capabilities
    - tab completion for functions, modules, variables, files
    - works neatly with **matplotlib**
    - filesystem navigation (cd, ls, pwd)
    - has access to the standard Python help and ?/?? information
    - Search commands (Ctrl-n, Ctrl-p, Ctrl-r)
    - the output of the nth command is in _n
    - magic commands: type % → (tab) to list them all
      - %whos
      - %run script.py
      - %timeit
      - %logstart name
  - improves the **interactive mode** usage

Python types

- Python is strongly typed and dynamically typed
  - Everything is a **type**: type(x)
  - a = 4 ; a = 4.5

- Operator “=” means a reference to a space in memory that contains an object
  - id(x)

- Objects are mutable (once created can be changed or updated) or immutable

strings

- Strings can be created using quotes (single, double or triple)
  - a = 'home'
  - b = “new home”
- Triple quotes are used for string that contains single or double quotes or that span over more than a single line
- Escape characters are similar to C (\n \t)

strings / 2

- Multiple actions on strings

```
>>> a = 'my new home'
>>> a.upper()
>>> a.split()
```

- Single elements of strings can be accessed

```
>>> a[0:2]    # python index starts from 0
>>> a[-4:]    # no values means beginning or end
```

- Concatenation of strings

```
>>> a+” is beautiful”        >>> a*3
```

Containers (sequences)

- List

```
>>> a = [1, 1, 2, 'home']   # mutable
```

- Tuple

```
>>> a = (1, 4, 'seven', 6)  #  immutable
```

- Dict

```
>>> a = {'a': 2, 'b':4, 4:5}  # mutable
```

- Set

```
>>> a = set([1, 1, 3, 5])
```

List

- Can be not homogeneous

```
>>> a = [1, 1, 2, 'home']
```

- Index ranges from 0 to len(*list*)

- Slicing

```
>>> a[0:2]   # from first to third element   [i:j:k]  k = stride
>>> a[-1:] + a[:-1]    # ['home', 1, 1, 2]
```

- Mutable (in-place)

```
>>> a[0] = 4     # [4, 1, 2, 'home']
```

List / 2

- append

```
>>> a = [1, 1, 2, 'home']
>>> a.append(3)    # [1, 1, 2, 'home', 3]
```

- pop

```
>>> a.pop()   # remove rightmost element
```

- Function “range” can be used to create list of integers

```
>>> a = range(3)    #  [0, 1, 2]
>>> b = range(2, 10, 3)   #  first, last (excluded), step
                          #  [2, 5, 8]
```

Dictionaries

- Map keys to values (mappings)

```
>>> a = {'b':2, 'c': 3}   #  'b', 'c' keys   2,3 values
>>> a['b']   # returns 3
```

- There is no left to right order, only mapping

```
>>> a[-1]   # does not work
```

- a.keys(), a.values(), a.items()

Control-flow statements

- **Indentation** matters

```
>>> if a > 3:  # mind the colon
        print a
        print 'still in the if statement'
    elif a == 5:
        print 'a is 5'
    else:
         print 'a is less than 3'
```

for loop

- Any sequence object is iterable

```
>>> for i in range(5):
        print i   #  prints 0, 1, 2, 3, 4
```

- More common in python

```
>>> a = [1, 1, 4, 'home']
>>> for i in a:
        print i   #  prints 1, 1, 4, 'home'
```

- break # exit from inner loop
- continue # go to next iteration

Bool conversion

- Built-in types can be converted in bool, i.e. they can be used as condition expressions

int 0 # False

int != 0 # True

float 0.0 # False

float != 0.0 # True

empty string “” # False

empty sequence # False

Let's go with a live example

(serial) Python program that runs simple simulations

cash_flow

- It's a toy model that simulates cash flows of insurance company deriving from life policies
- Each month the insured (policy holder) pay the company a sum of money (*premium*)
- In case of death (random, look up actuarial table) of the insured, the company pays a sum of money to the beneficiary of the life policy
- If the insured is still alive, it receives some money back

cash_flow / 2

Input file

```
300    # (number of policies, thousands)
15     # (policy length years)
150    # (premium Euro)
2.5    # (beneficiary return factor)
25     # (insured initial age)
       # 25,30,35,40
```

mpi4py

mpi4py: philosophy

- Provides python bindings to MPI libraries
- Often only a small portion of the code is time-critical
- Use python for everything, apart from heavy work calculation
  - Memory management
  - Input / Output
  - User interface
  - Error handling

mpi4py

- OO Interface similar to MPI C++
- You can communicate Python objects
- Optimized communications of Python objects that expose single-segment buffer interface (contiguous memory buffer), i.e. Numpy arrays
  - Performance close to C speed

mpi4py / 2

- No need to call MPI_Init() or MPI_Finalize()

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

point to point

- Send(), Recv(), Sendrecv() can communicate memory buffers
- send(), recv(), sendrecv() can communicate generic Python objects
- Nonblocking communications are also available

```
#!/usr/bin/env python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

buf = [ ]

if rank == 0:
    comm.send([rank, 1000], dest=1, tag=10)
    buf = comm.recv(source=1, tag=20)
else:
    buf = comm.recv(source=0, tag=10)
    comm.send([rank, 1000] , dest=0, tag=20)

print "my rank is %d, I received %s from %d" % (rank, buf, buf[0])
```

Collective communications

- Barrier() # synchronization
- Global communications
  - Broadcast
  - Gather
  - Scatter
- Global reduction operations

```
#!/usr/bin/env python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],  'key2' : ( 'abc', 'xyz')}
else:
    data = None

data = comm.bcast(data, root=0)   # broadcast of a dict

print rank, data
```

More info on:

http://mpi4py.scipy.org/docs/apiref/index.html