GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Hendrik Nolte

Parallelisation with Python

Practical Course on Parallel Computing

Parallelization in Python

- Python offers two ways of parallelization:
  - **Multi-Threading:**

    Sub-tasks on several CPUs share the same memory

    Requires proper memory synchronisation

  - **Multi-Processing:**

    Each process has own memory, processes run completely independent from each other

    More stable, but additional overhead in in memory consumption

Multi-Threading

- Included modules in Python:
  - `thread` (deprecated, not supported any more in Python3)
  - `threading`
- The threading module:
  - Provides the following functions:
    - `threading.activeCount()` – Returns the number of thread objects that are active.
    - `threading.currentThread()` – Returns information of the thread from where it is called.
    - `threading.enumerate()` – Returns a list of all thread objects that are currently active.
- Threads can be initialized and started with

```
thread      = threading.Thread(target=f, args=(var1, var2,))
thread.start()
```

(executes function *f(var1,var2)* in a thread)

Simple example

- Example `ThreadTest_0.py`:

```
#!/usr/bin/python

import threading

def myfunction(a,b):
   print a*b

# Create new threads
thread1 = threading.Thread(target=myfunction, args=(2,3))
thread2 = threading.Thread(target=myfunction, args=(4,6))

# Start the threads
thread1.start()
thread2.start()
```

The Thread Class

- Everything that you want to execute in a parallel thread can also be written in a `Thread` class:
- Methods of the class:
  - `__init__ (self [,args])` - Initialization
  - `run()` - The method contains the code to be ran in parallel.
  - `start()` - The method starts a thread by calling the run method.
  - `join(timeout=None)` - Waits for threads to terminate.
  - `isAlive()` - The method checks whether a thread is still executing.
  - `getName()` - The method returns the name of a thread.
  - `setName()` - The method sets the name of a thread.
- To use threading, you have to write your own sub-class of the `Thread` class:
  - Define new sub-class
  - Override the `__init__()` and `run()` methods

Multi-Threading Example

- Example ThreadTest_1.py:

```
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print "Starting " + self.name
      print_time(self.name, 5, self.counter)
      print "Exiting " + self.name
```

…

Multi-Threading Example

…

```
def print_time(threadName, counter, delay):
   while counter:
      time.sleep(delay)
      print "%s: %s" % (threadName, time.ctime(time.time()))
      counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

Multi-Threading Example

- Output:

```
Starting  Thread-1
Starting  Thread-2
 Exiting Main Thread
Thread-1: Wed May  8 16:37:06 2019
Thread-1: Wed May  8 16:37:07 2019
Thread-2: Wed May  8 16:37:07 2019
Thread-1: Wed May  8 16:37:08 2019
Thread-1: Wed May  8 16:37:09 2019
Thread-2: Wed May  8 16:37:09 2019
Thread-1: Wed May  8 16:37:10 2019
Exiting Thread-1
Thread-2: Wed May  8 16:37:11 2019
Thread-2: Wed May  8 16:37:13 2019
Thread-2: Wed May  8 16:37:15 2019
Exiting Thread-2
```

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Monitoring Threads

- Add the following lines to your code after you started the threads:

```
for t in threading.enumerate():
    print t
```

- Output:

```
<_MainThread(MainThread,    started 139672845326144)>
<myThread(Thread-1, started 139672823293696)>
<myThread(Thread-2, started 139672814900992)>
```

- There will always be a `MainThread`

Synchronizing Threads

- Synchronization needed for threads that depend on each other
  → Requires communication between threads

- Easiest way for communication: the `threading.Event()` object:

  Provided functions:

  - `set()` and `clear()`: set an internal flag to true or false
  - `isSet()`: checks if the event has been set by the `set()` method
  - `wait()`: blocks further processing until another thread calls the set() method; can be used with `wait(t)` to time-out after `t` second

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Synchronizing: Example

- Example ThreadTest_2.py:

```
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
   def __init__(self, threadID, name, e):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.e = e

   def run(self):

      if(self.threadID==1):
         print "Start blocking"
         time.sleep(10)
         self.e.set()
         print "End blocking"
```

```
      if(self.threadID==2):
         print "Start waiting process"
         while not self.e.isSet():
            event_is_set = self.e.wait(2)
            print "event set: ", event_is_set
            if event_is_set:
               print "processing event"
            else:
               print "doing other things"

e = threading.Event()

# Create new threads
thread1 = myThread(1, "blocking", e)
thread2 = myThread(2, "non-blocking", e)

# Start new Threads
thread1.start()
thread2.start()

#Wait for threads to finish
thread1.join()
thread2.join()

print "Exiting"
```

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Synchronizing: Example

- Output:

```
Start blocking  waiting process
Start
event set:  False
doing other things
event  set:  False
doing other things
event  set:  False
doing other things
event  set:  False
doing other things
event  set:  False
End blocking
event set:  True
processing
 Exiting   event
```

Synchronizing: Locking

- Locking is alternative method to synchronize threads

- The `threading.Lock()` object:
  - `acquire()`: change the state to locked
  - `release()`: unlock the state

- If one thread calls `acquire()` for a lock object, all other threads calling `acquire()` have to wait until the first thread calls `release()`.

Locking: Example

- Modify previous example (see `ThreadTest_3.py`):
  - Add definition of Lock object to main part of the program:

```
myLock = threading.Lock()
```

  - Modify `run()` function of first example:

```
def run(self):
    print "Starting " + self.name
    # Get lock to synchronize threads
    myLock.acquire()
    print_time(self.name, self.counter, 3)
    # Free lock to release next thread
    myLock.release()
```

Locking: Example

- Output:

```
Starting Thread-1
Starting Thread-2
Thread-1: Fri May 10 17:18:50 2019
Thread-1: Fri May 10 17:18:51 2019
Thread-1: Fri May 10 17:18:52 2019
Thread-2: Fri May 10 17:18:54 2019
Thread-2: Fri May 10 17:18:56 2019
Thread-2: Fri May 10 17:18:58 2019
```

→ Thread 2 has to wait for thread 1 to finish

Multi-Processing

- Multi-core processing configuration is very similar to threading
- The `multiprocessing` class:
  - Similar methods as the `threading` class, but:
    - `Thread` → `Process`
    - `threading` → `multiprocessing`
  - Example (`MPTest_0.py`):

```
#!/usr/bin/python

import multiprocessing

def myfunction(a,b):
   print a*b

# Create new processs
process1 = multiprocessing.Process(target=myfunction, args=(2,3))
process2 = multiprocessing.Process(target=myfunction, args=(4,6))

# Start the processs
process1.start()
process2.start()
```

Difference to Threading

- In multi-threading, each thread accesses the same memory
- In multi-processing, memory is completely separated

- Example (`MPTest_1.py`):

```
#!/usr/bin/python

import multiprocessing
import threading
import random

result =[]

def myfunction():
    result.append(random.randint(1,10))
```

Now run `myfunction()` in multiprocessing and threading mode

Difference to Threading

```
# Create new processes
process1 = multiprocessing.Process(target=myfunction)
process2 = multiprocessing.Process(target=myfunction)

# Start the processes
process1.start()
process2.start()

#Wait for processes to finish
process1.join()
process2.join()

print "Multi-processing result: " ,  result

result=[]

# Create new threads
thread1 = threading.Thread(target=myfunction)
thread2 = threading.Thread(target=myfunction)

# Start the threads
thread1.start()
thread2.start()

#Wait for threads to finish
thread1.join()
thread2.join()

print "Multi-threading result: " ,  result
```

- Output:

```
Multi-processing result:   []
Multi-threading result: [9, 2]
```

Multiprocessing Pool

- Multi-processing can be done by creating a `Pool`:
  - How to create a pool:

```
Import multiprocessing as mp

#create a pool with all  available CPUs
 pool= mp.Pool(mp.cpu_count())
```

  - The Pool class provides the following functions:
    - `apply`
    - `map`
    - `starmap`
    - `close()`: stops the pool cluster

  `apply` and `map` are similar to the default functions apply and map, but arguments are executed in parallel on the pool.

Pool: Example 1

- Calling `apply` executes a single function on the pool, see `MPTest_2.py`:

```
#!/usr/bin/python

import multiprocessing as mp

def myfunction(a, b):
   return a*b

pool = mp.Pool(mp.cpu_count())

results = [ pool.apply(myfunction, args=(a,2)) for a in range(1,100) ]

pool.close()

print results
```

```
 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58,
60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108  ...
                                               ,
```

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Pool: Example 2

- `map` takes one iterable as argument and executes processes for each iterable object, see `MPTest_3.py`:

```
#!/usr/bin/python

import multiprocessing as mp

def myfunction(a):
   return a*2

pool = mp.Pool(mp.cpu_count())

results = pool.map(myfunction, [a for a in range(1,100)])

pool.close()

print results
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58,
60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108 ...
,
```

Pool: Example 3

- `starmap` (available only since Python 3.3) also takes one iterable as argument, but each iterable object can be iterable again, see `MPTest_4.py`:

```
#!/usr/bin/python3

import multiprocessing as mp

def myfunction(a,b):
    return a*b

pool = mp.Pool(mp.cpu_count())

results = pool.starmap(myfunction, [(a,2) for a in range(1,100)])

pool.close()

print(results)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58,
60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108 ...
,
```

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Asynchronous Execution

- Asynchronous execution is often faster then the default synchronized way
- Output of individual jobs is not ordered
- Need specific functions to retrieve the unordered output
- In python:

  - `Pool.apply_async`
  - `Pool.map_async`

  Methods provide just list of output objects, not the output of the parallel calculation

  Output can be collected via:

  - Call the `pool.ApplyResult.get()`
  - Define `callback` routine as argument of `apply_async` or `map_async` that is called after all jobs finished

Asynchronous: Example 1

- Example with `get()`, see `MPTest_5.py`:

```
#!/usr/bin/python

import multiprocessing as mp

def myfunction(a, b):
    return a*b

pool = mp.Pool(mp.cpu_count())

results_objects = [pool.apply_async(myfunction, args=(a,2)) for a in range(1,100)]

results = [r.get() for r in results_objects]

pool.close()

print results
```

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Asynchronous: Example 2

- Example with `callback` function, see

```
#!/usr/bin/python

import multiprocessing as mp

results = []

def myfunction(a, b):
   return a*b

# Define callback function to collect the output:
def collect_result(result):
    global results
    results.append(result)

pool = mp.Pool(mp.cpu_count())

for a in range(1,100):
   pool.apply_async(myfunction, args=(a,2), callback=collect_result)

pool.close()

print results
```