UL HPC Users' session: Python multiprocessing

Raymond Bisdorff

Université du Luxembourg
FSTC/ILAS

Luxembourg, June 2015

Using GNU parallel for python nose tests

Running python3 nosetests in parallel on a SMP machine like the Intel Core i7 processors:

```
...$ make tests
tests:
    cp digraphs.py test/
    cp perfTabs.py test/
     ...
    (cd test; nosetests3 -v\
        noseTestsDigraph.py)
    (cd test; nosetests3 -v\
        noseTestsPerfTab.py)
     ...
```

```
...$ make pTests
pTests:
   parallel --gnu cp {}.py\
          test/ ::: digraphs\
                    perfTabs\
                    ...
   (cd test; parallel --gnu -k\
         nosetests3 -v ::: noseTests*.py )
```

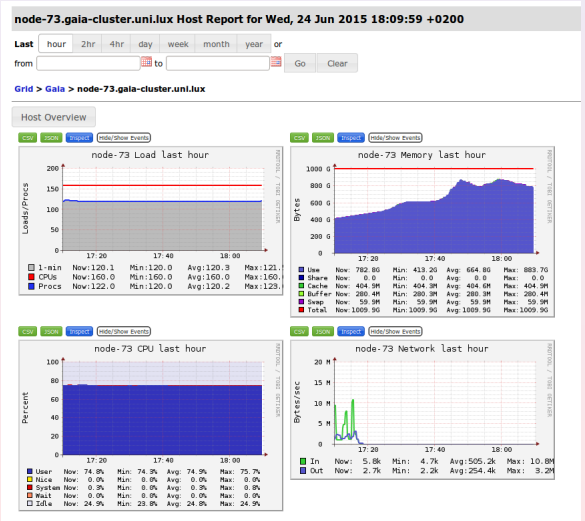

Using a private virtual python 3.4 environmant

```
...(gaia-cluster) ~ $ cd $WORK
...(gaia-cluster) rbisdorff $ source myP34/bin/activate
(myP34)...(gaia-cluster) rbisdorff $ cd Digraph3
(myP34)...(gaia-cluster) (svn:1379M) Digraph3 $\
   svn update
   U    randomPerfTabs.py
   Updated to revision 1380.
(myP34)...(gaia-cluster) (svn:1380M) Digraph3 $\
   make installVenv
   ...
(myP34)...(gaia-cluster) (svn:1380M) Digraph3 $\
   make pTests
...
...
noseTestsWeakOrders.testQuantilesRankingDigraphWithOutThreading ... ok
noseTestsWeakOrders.testQuantilesRankingDigraphWithThreading ... ok
noseTestsWeakOrders.testKohlerArrowRaynaudFusionDigraph ... ok
----------------------------------------------------------------------
Ran 8 tests in 15.615s
OK
```


Use case 1: running independent experiments in parallel

```
from multiprocessing import Pool
Nsim = 1000
resultFileName = 'results.csv'
Nproc = 120
## task description
def jobTask(s):
    print('simulation = %d into %s ' %(s+1,resultFileName))
    # params
    ...
    # task computations
    ...
    return(writestr)
if __name__ == '__main__':
    ### prepare result csv file
    fo = open(resultFileName,'w')
    fo.write(... header row ...)
    fo.close()
    ### starting the pool of workers
    with Pool(processes=Nproc) as pool:
        for res in pool.imap_unordered(jobTask,range(Nsim)):
            print(res)
            fo = open(resultFileName,'a')
            fo.write(res)
            fo.close()
```

Monitoring the used memory on the connected node



Use case 2: multithreading algorithmic design

```
from multiprocessing import Process, active_children
    class myThread(Process):
        def __init__(self, threadID, ...)
                    Process.__init__(self)
                    self.threadID = threadID
                    ...
        def run(self):
            ... task description
            ...

nbrOfJobs = ...
for j in range(nbrOfJobs):
    ... pre-threading tasks per job
    print('iteration = ',j+1,end=" ")
    splitThread = myThread(j, ...)
    splitThread.start()
while active_children() != []:
    pass
print('Exiting computing threads')
for j in range(nbrOfJobs):
    ... post-threading tasks per job
```


Use case 2: choosing the right granularity ?

Is it more efficient: ?

- to run many simple jobs in parallel
- to run a in parallel a small number of complex jobs
- to align the number of parallel jobs to the number of reserved cores
- to start more parallel jobs than reserved cores