Calculations using remote computers

This is a second part in the tutorial series. Read the first tutorial to understand all material presented here.

Using remote computers requires some additional configuration and slightly different approach to the process. The initial setup is identical and its description will not be repeated here. We will move directly to the configuration of the QE-util package. Most of the configuration is pre-packaged in the `RemoteQE` calculator object and for typical supercomputing setups does not need any changes. You can review the variables in the `RemoteQE` if you need any modifications to be made.

In [1]:
```
# Import the basic libraries

# ASE system
import ase
from ase import Atom, Atoms
from ase import io
from ase.lattice.spacegroup import crystal

# Spacegroup/symmetry library
from pyspglib import spglib

# iPython utility function
from IPython.core.display import Image
```

In [2]:
```
# Import the remote execution tools from the qe-util package
from qeutil import RemoteQE
```

Configuring for remote execution

The basic idea of using a remote computing system for a calculation revolves around submitting jobs to the queue and collecting the results. The `RemoteQE` calculator is pre-configured for the most common case, which is not the most effective one but is almost guaranteed to work on any modern Linux installation and most of other Unix-based systems. The only external requirements are availability of the `ssh` and `rsync` tools (common in modern distributions) and proper setup of key-authenticated login between the local and remote machine.

Remember to transfer the pseudopotential directory to the remote machine!

Private Key authentication

For the data transfer to work you need to connect your accounts on local and remote machine. The procedure is quite simple. You need to generate your private-public key pair. You may have done it in the past. Check the `.ssh` directory (note the dot in the front) for `id_rsa` and `id_rsa.pub` files. If they are present skip the key generation part.

1. **Key pair generation**

   ```
   ssh-keygen
   ```

   Accept default answers to the questions with the `enter` key.

1. **Account connection**

   ```
   ssh-copy-id user@host
   ```

   Replace the `user@host` by your account data. After authenticating to the remote system you should be able to log in from your local machine to the remote one with private key authentication.

   Remember that from now on the security of your remote account is connected with the security of your local machine. You need to have proper password authentication on the local machine and keep it secure.

Adaptation for your setup

The rest of the configuration is easy. Just follow the example below. Usually adaptation for your particular setup will be quite minimal. Just put your user name into `RemoteQE.user` variable, and change the `RemoteQE.host` variable to the name of the machine you are working with. Copy your pseudopotential directory to the remote machine and create working directory locally and remotely. The only non-trivial part of the configuration is a PBS script for the particular system present on your machine. For example for the `bugaboo.westgrid.ca` this script will look like this:

```
#!/bin/bash
#PBS -r n
#PBS -l walltime=140:00:00
#PBS -l pmem=8000m
#######################
cd $PBS_O_WORKDIR
echo "PWscf started at: `date`"
%(command)s
echo "Run finished at: `date`"
```

Further, you need to set up your access and host configuration info. To avoid repeating this procedure in each notebook you can put all definitions regarding one machine into a single file: `host.py` and just import this file at the start of your session. The `host.py` file would look similar to the following configuration:

```
#!/usr/bin/python

from qeutil import RemoteQE

# Access info
RemoteQE.user='user'
RemoteQE.host='host.example.ca'

# Working directories: local and remote
RemoteQE.wdir='/home/user/calc/'
RemoteQE.rdir='calc/'

# The job running script.
# This is specific to the particular host
# You need to replace it by the proper script for your machine.
# Usually only module loding part changes, but sometimes the modifications need to be more extensive.
RemoteQE.pbs_template='''#!/bin/bash

cd $PBS_O_WORKDIR

echo "QE started at: `date`"
%(command)s
echo "QE finished at: `date`
'''
```

In [3]:
```
# Now simply import the configuration.
# The name of the file is host.py in this case.
import host
```

Single calculation

We start with a single calculation. The same stress tensor calculation from the first example. We will just execute it on the remote machine.

In [4]:
```
# Stup the SiC crystal
# Create a cubic crystal with a spacegroup F-43m (216)
a=4.36
cryst = crystal(['Si', 'C'],
                [(0, 0, 0), (0.25, 0.25, 0.25)],
                spacegroup=216,
                cellpar=[a, a, a, 90, 90, 90])
```

In [5]:
```
# Check the spacegroup (symmetry) of our creation
spglib.get_spacegroup(cryst)
```

Out[5]:
```
'F-43m      (216)'
```

In [6]:
```
# Create a Quantum Espresso calculator for our work.
# This object encapsulates all parameters of the calculation,
# not the system we are investigating.
qe=RemoteQE(label='Remote',
              kpts=[4,4,4],
              xc='pz',         # Exchange functional type in the name of the pseudopotentials
              pp_type='vbc',   # Variant of the pseudopotential
              pp_format='UPF', # Format of the pseudopotential files
              ecutwfc=70,
              pseudo_dir='../pspot',
              use_symmetry=True,
              procs=8)         # Use 8 cores for the calculation

# Check where the calculation files will reside on the local machine.
print qe.directory
```

```
calc/Remote.h87LBH
```

In [7]:
```
# Assign the calculator to our system
cryst.set_calculator(qe)
```

In [8]:
```
# Run the calculation to get stress tensor (in Voigt notation, in eV/A^3) and pressure (in kBar)
# Do the same but get the results in GPa
# Note that this time we get the results immediately.
# We did not change the system, so it is not necessary to repeat the calculation.
print "Stress tensor  (Voigt notation, GPa):", cryst.get_stress()/ase.units.GPa
print
print "Stress tensor (Tensor notation, GPa):"
print cryst.get_stress(voigt=False)/ase.units.GPa
print
print "External pressure              (GPa):", cryst.get_isotropic_pressure(cryst.get_stress())*1e-4
```

```
Stress tensor  (Voigt notation, GPa): [ 3.549  3.549  3.549 -0.     0.     0.   ]

Stress tensor (Tensor notation, GPa):
[[ 3.549  0.     0.   ]
 [ 0.     3.549 -0.   ]
 [ 0.    -0.     3.549]]

External pressure              (GPa): -3.549
```

Series of calculations

The real power of the remote execution in the computing center comes with ability of using parallel computation of many jobs on many processors. Using RemoteQE we can run a series of calculations in parallel.

For example we can again find a minimum of energy of our crystal. To do this we need to generate a series of structures with various lattice constants, calculate the energy and stress for each and collect the results. Since we will submit all jobs at once the calculation will be finish quicker.

First we need to create the list of structures.

In [9]:
```
# The list for crystals
systems=[]

# Loop over scales from 98% to 102% lattice constant.
# 11 structures
for x in linspace(0.98,1.02,11):

    # Structure scaled by the x factor
    cr=Atoms(cell=x*cryst.get_cell(),
             numbers=cryst.get_atomic_numbers(),
             scaled_positions=cryst.get_scaled_positions(), pbc=True)

    # Remote calculator for each structure
    cqe=RemoteQE(label='CldP',
                  kpts=[8,8,8],
                  xc='pz',         # Exchange functional type in the name of the pseudopotentials
                  pp_type='vbc',   # Variant of the pseudopotential
                  pp_format='UPF', # Format of the pseudopotential files
                  ecutwfc=70,
                  pseudo_dir='../pspot',
                  use_symmetry=True,
                  procs=8)         # Use 8 cores for the calculation
    cr.set_calculator(qe.copy())
    systems.append(cr)
```

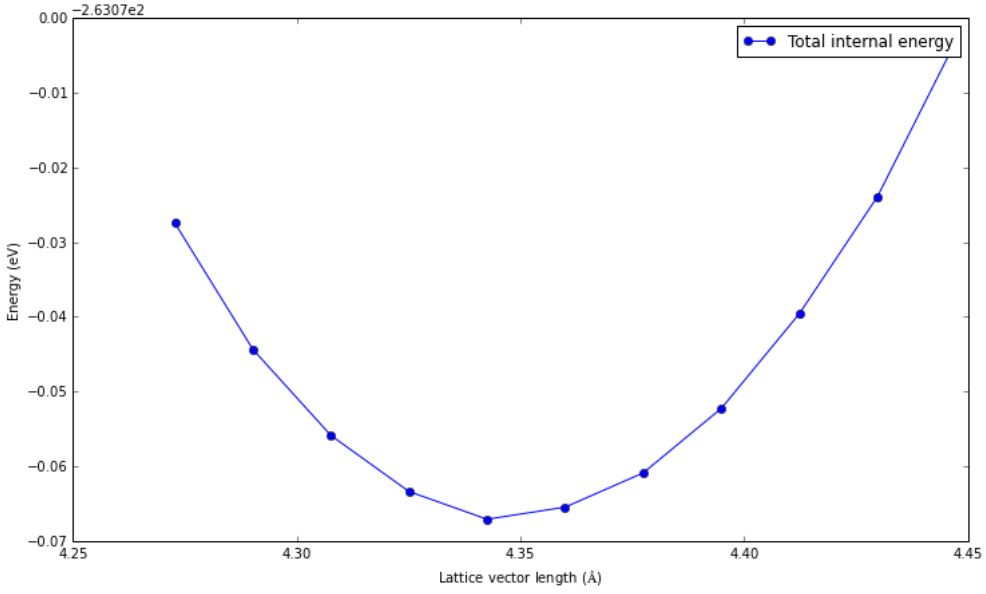
In [10]:
```
# Compute stresses for all systems in parallel
qe.ParallelCalculate(systems,properties=['stress']);
```

```
Launching: 1 2 3 4 5 6 7 8 9 10 11
     Done: 1 2 3 4 5 6 7 8 9 10 11
```

In [11]:
```
# Extract the results from the objects in the systems list
result=array([[s.get_cell()[0,0], s.get_potential_energy(), 1e-4*s.get_isotropic_pressure(s.get_stress())] for s in

# Print out the results
print "   A      Energy (eV)  Pressure (GPa) "
print "--------------------------------------"
for v in result:
    print "% 5.03f     %+6.4f      % +8.3f " % tuple(v)

# Prepare the collected data for plotting
# This will make an array (matrix) out of a list and transpose it for easier access
result=array(result).T

# Let us save our calculated data to a file.
# To have data in columns we need to transpose the array.
savetxt('e+p-vs-a.dat',result.T)
```

```
   A      Energy (eV)  Pressure (GPa)
--------------------------------------
 4.273     -263.0975       +10.952
 4.290     -263.1144        +7.770
 4.308     -263.1259        +4.723
 4.325     -263.1334        +1.828
 4.343     -263.1371        -0.915
 4.360     -263.1355        -3.549
 4.377     -263.1309        -6.033
 4.395     -263.1223        -8.402
 4.412     -263.1096       -10.664
 4.430     -263.0939       -12.788
 4.447     -263.0732       -14.849
```

In [12]:
```
# Let us plot the results and save the figure
figsize(12,7)
plot(result[0],result[1],'o-',label='Total internal energy')
legend()
xlabel('Lattice vector length ($\AA$)')
ylabel('Energy (eV)')
savefig('e-vs-a.pdf')
```



This replicates our result from the first tutorial.