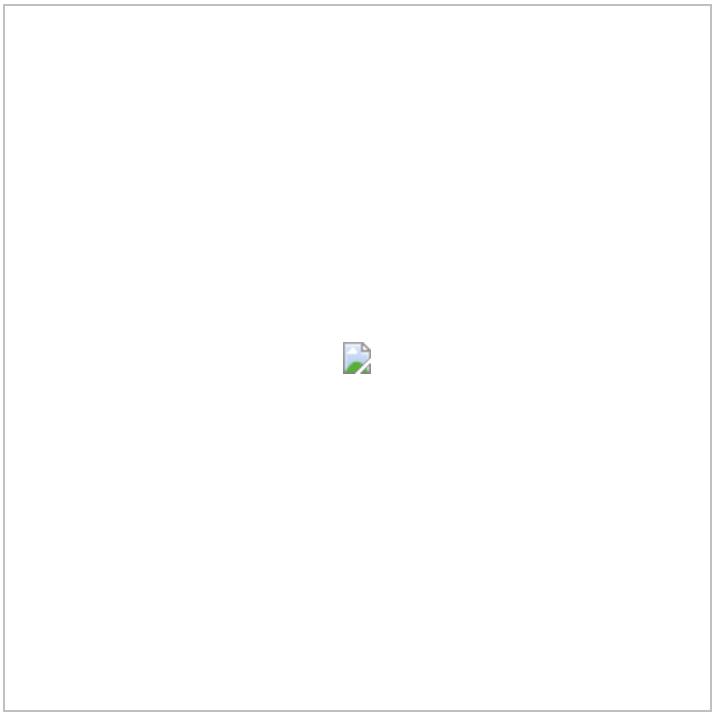
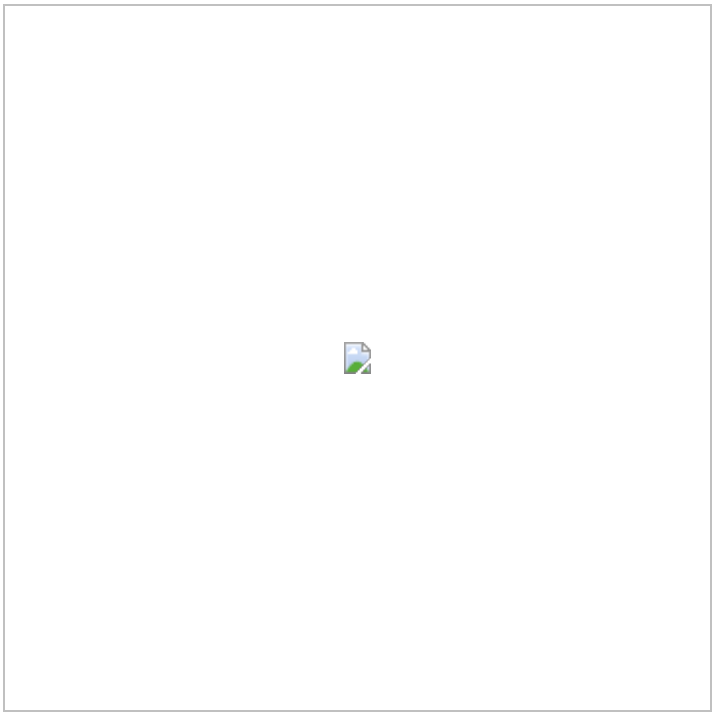
Distributed Model Selection and Assessment

Outline of the session:

- Introduction to **IPython.parallel**
- Sharing Data Between Processes with **Memory Mapping**
- **Parallel Grid Search** and Model Selection
- **Parallel** Computation of **Learning Curves** (TODO)
- **Distributed** Computation on **EC2 Spot Instances with StarCluster**

Motivation

When doing model evaluations and parameters tuning, many models must be trained independently on the same data. This is an embarrassingly parallel problem but having a copy of the dataset in memory for each process is waste of RAM:

When doing 3 folds cross validation on a 9 parameters grid, a naive implementation could read the data from the disk and load it in memory 27 times. If this happens concurrently (e.g. on a compute node with 32 cores) the RAM might blow up hence breaking the potential linear speed up.

IPython.parallel, a Primer

This section gives a primer on some tools best utilizing computational resources when doing predictive modeling in the Python / NumPy ecosystem namely:

- optimal usage of available CPUs and cluster nodes with **`IPython.parallel`**
- optimal memory re-use using shared memory between Python processes using **`numpy.memmap`** and **`joblib`**

What is so great about `IPython.parallel`:

- Single node multi-CPUs
- Multiple node multi-CPUs
- Interactive In-memory computing
- IPython notebook integration with `%px` and `%%px` magics
- Possibility to interactively connect to individual computing processes to launch interactive debugger (`#priceless`)

Let's get started:

Go to the "Cluster" tab of the notebook and **start a local cluster with 2 engines**. Then come back here:

In [1]:
```
from IPython.parallel import Client
client = Client()
```

In [2]:
```
len(client)
```

Out[2]: 2

The %px and %%px magics
All the engines of the client can be accessed imperatively using the `%px` and `%%px` IPython cell magics:

In [3]:
```
%%px
# all cell with %%px as the start will execute code in parallel in this cell

import os
import socket

print("This is running in process with pid {0} on host '{1}'.".format(
      os.getpid(), socket.gethostname()))
```

```
[stdout:0] This is running in process with pid 8530 on host 'Alexander-Chens-
MacBook-Pro.local'.
[stdout:1] This is running in process with pid 8531 on host 'Alexander-Chens-
MacBook-Pro.local'.
[stderr:0]
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/site-
packages/IPython/utils/rlineimpl.py:111: RuntimeWarning:
*******************************************************************************
libedit detected - readline will not be well behaved, including but not limited to:
   * crashes on tab completion
   * incorrect history navigation
   * corrupting long-lines
   * failure to wrap or indent lines properly
It is highly recommended that you install readline, which is easy_installable:
     easy_install readline
Note that `pip install readline` generally DOES NOT WORK, because
it installs to site-packages, which come *after* lib-dynload in sys.path,
where readline is located.  It must be `easy_install readline`, or to a custom
location on your PYTHONPATH (even --user comes after lib-dyload).
*******************************************************************************
  RuntimeWarning)
[stderr:1]
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/site-
```

```
packages/IPython/utils/rlineimpl.py:111: RuntimeWarning:
*************************************************************************
libedit detected - readline will not be well behaved, including but not limited to:
   * crashes on tab completion
   * incorrect history navigation
   * corrupting long-lines
   * failure to wrap or indent lines properly
It is highly recommended that you install readline, which is easy_installable:
     easy_install readline
Note that `pip install readline` generally DOES NOT WORK, because
it installs to site-packages, which come *after* lib-dynload in sys.path,
where readline is located.  It must be `easy_install readline`, or to a custom
location on your PYTHONPATH (even --user comes after lib-dyload).
*************************************************************************
  RuntimeWarning)
```

The content of the `__main__` namespace can also be read and written via the `%px` magic:

In [4]:
```
%px a = 1
```

In [5]:
```
%px print(a)
```

```
[stdout:0] 1
[stdout:1] 1
```

In [6]:
```
%%px
# double one value and get 2 of them run in different engines
a *= 2
print(a)
```

```
[stdout:0] 2
[stdout:1] 2
```

It is possible to restrict the `%px` and `%%px` magic instructions to specific engines:

In [7]:
```
%%px --targets=-1
# this means targeting a specific machine
# -1 last started engine
# better -1 then 0, 0 is the first engine, might have died
a *= 2
print(a)
```

```
4
```

In [8]:
```
%px print(a)
# latest engine a has doubled, the other one didn't change
```

```
[stdout:0] 2
[stdout:1] 4
```

The DirectView objects

Cell magics are very nice to work interactively from the notebook but it's also possible to replicate their behavior programmatically with more flexibility with a `DirectView` instance. A `DirectView` can be created by slicing the client object:

In [9]:
```
all_engines = client[:]
all_engines
```

Out[9]: <DirectView [0, 1]>

Out[9]: <DirectView [0, 1]>

The namespace of the `__main__` module of each running python engine can be accessed in read and write mode as a python dictionary:

In [10]:
```
all_engines['a'] = 1
```

In [11]:
```
all_engines['a']
```

Out[11]: [1, 1]

Direct views can also execute the same code in parallel on each engine of the view:

In [12]:
```
def my_sum(a, b):
    return a + b

my_sum_apply_results = all_engines.apply(my_sum, 11, 31)
my_sum_apply_results
```

Out[12]: <AsyncResult: my_sum>

The ouput of the `apply` method is an asynchronous handle returned immediately without waiting for the end of the computation. To block until the results are ready use:

In [13]:
```
my_sum_apply_results.get()
```

Out[13]: [42, 42]

Here is a more useful example to fetch the network hostname of each engine in the cluster. Let's study it in more details:

In [21]:
```
def hostname():
    """Return the name of the host where the function is being called"""
    # do the import in the function, because we are loading this on a
    # particular cluster
    import socket
    return socket.gethostname()

hostname_apply_result = all_engines.apply(hostname)
```

When doing the above, the `hostname` function is first defined locally (the client python process). The `DirectView.apply` method introspects it, serializes its name and bytecode and ships it to each engine of the cluster where it is reconstructed as local function on each engine. This function is then called on each engine of the view with the optionally provided arguments.

In return, the client gets a python object that serves as an handle to asynchronously fetch the list of the results of the calls:

In [22]:
```
hostname_apply_result
```

Out[22]: <AsyncResult: hostname>

In [23]:
```
hostname_apply_result.get()
```

Out[23]: ['Alexander-Chens-MacBook-Pro.local', 'Alexander-Chens-MacBook-Pro.local']

It is also possible to key the results explicitly with the engine ids with the `AsyncResult.get_dict` method. This is a very simple idiom to fetch metadata on the runtime environment of each engine of the direct view:

In [24]:
```
hostnames = hostname_apply_result.get_dict()
hostnames
```

Out[24]:
```
{0: 'Alexander-Chens-MacBook-Pro.local',
 1: 'Alexander-Chens-MacBook-Pro.local'}
```

It can be handy to invert this mapping to find one engine id per host in the cluster so as to execute host specific operation:

In [25]:
```
# which engine runs on which host
one_engine_by_host = dict((hostname, engine_id) for engine_id, hostname
                          in hostnames.items())
one_engine_by_host
```

Out[25]:
```
{'Alexander-Chens-MacBook-Pro.local': 1}
```

In [26]:
```
one_engine_per_host_view = client[one_engine_by_host.values()]
one_engine_per_host_view
```

Out[26]:
```
<DirectView [1]>
```

Trick: you can even use those engines ids to execute shell commands in parallel on each host of the cluster:

In [27]:
```
one_engine_by_host.values()
```

Out[27]:
```
[1]
```

In [28]:
```
%%px --targets=[1]
# tell the machine which cluster do you want to run it on
# so this is installing something at clustering one

!pip install flask
```

```
[stdout:1] /bin/sh: pip: command not found
```

Note on Importing Modules on Remote Engines
In the previous example we put the `import socket` statement inside the body of the `hostname` function to make sure to make sure that is is available when the rest of the function is executed in the python processes of the remote engines.

Alternatively it is possible to import the required modules ahead of time on all the engines of a directview using a context manager / with syntax:

In [29]:
```
with all_engines.sync_imports():
    import numpy
```

```
importing numpy on engine(s)
```

However this method does **not** support alternative import syntaxes:

```
>>> import numpy as np
```

```
>>> from numpy import linalg
```

Hence the method of importing in the body of the "applied" functions is more flexible. Additionally, this does not pollute the `__main__` namespace of the engines as it only impact the local namespace of the function itself.

Exercise:

- Write a function that returns the memory usage of each engine process in the cluster.
- Allocate a largish numpy array of zeros of known size (e.g. 100MB) on each engine of the cluster.

Hints:

Use the `psutil` module to collect the runtime info on a specific process or host. For instance to fetch the memory usage of the currently running process in MB:

```
>>> import os
>>> import psutil
>>> psutil.Process(os.getpid()).get_memory_info().rss / 1e6
```

To allocate a numpy array with 1000 zeros stored as 64bit floats you can use:

```
>>> import numpy as np
>>> z = np.zeros(1000, dtype=np.float64)
```

The size in bytes of such a numpy array can then be fetched with `z.nbytes`:

```
>>> z.nbytes / 1e6
0.008
```

In [30]:
```
def get_engines_memory(client):
    def memory_mb():
        import os, psutil
        return psutil.Process(os.getpid()).get_memory_info().rss / 1e6

    return client[:].apply(memory_mb).get_dict()
```

In [31]:
```
get_engines_memory(client)
```

Out[31]:
```
{0: 15.9744, 1: 16.15872}
```

In [32]:
```
sum(get_engines_memory(client).values())
```

Out[32]:
```
32.681984
```

In [33]:
```
%%px
import numpy as np
z = np.zeros(int(1e7), dtype=np.float64)
print("Allocated {0}MB on engine.".format(z.nbytes / 1e6))
```

```
[stdout:0] Allocated 80.0MB on engine.
[stdout:1] Allocated 80.0MB on engine.
```

In [34]:
```
get_engines_memory(client)
```

Out[34]:
```
{0: 97.3824, 1: 97.50528}
```

Load Balanced View

`LoadBalancedView` is an alternative to the `DirectView` to run one function call at a time on a free engine.

In [35]:
```
lv = client.load_balanced_view()
```

In [36]:
```
def slow_square(x):
    import time
    time.sleep(2)
    return x ** 2
```

In [37]:
```
result = lv.apply(slow_square, 4)
```

In [38]:
```
result
```

Out[38]:
```
<AsyncResult: slow_square>
```

In [39]:
```
result.ready()
```

Out[39]:
```
True
```

In [40]:
```
result.get()  # blocking call
```

Out[40]:
```
16
```

It is possible to spread some tasks among the engines of the LB view by passing a callable and an iterable of task arguments to the `LoadBalancedView.map` method:

In [41]:
```
results = lv.map(slow_square, [0, 1, 2, 3])
results
```

Out[41]:
```
<AsyncMapResult: slow_square>
```

In [42]:
```
results.ready()
```

Out[42]:
```
False
```

In [43]:
```
results.progress
```

Out[43]:
```
2
```

In [44]:
```
# results.abort()
```

In [45]:
```
# Iteration on AsyncMapResult is blocking
for r in results:
    print(r)
```

```
0
1
4
9
```

The load balanced view will be used in the following to schedule work on the cluster while being able to monitor progress and occasionally add new computing nodes to the cluster while computing to speed up the processing when using EC2 and StarCluster (see later).

Sharing Read-only Data between Processes on the Same Host with Memmapping

Let's **restart the cluster** to kill the existing python processes and restart with a new client instances to be able to monitor the memory usage in details:

In [48]:
```
from IPython.parallel import Client
client = Client()
```

In [49]:
```
print len(client)
```

```
2
```

The numpy package makes it possible to memory map large contiguous chunks of binary files as shared memory for all the Python processes running on a given host:

In [50]:
```
%px import numpy as np
```

```
[stderr:0]
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/site-
packages/IPython/utils/rlineimpl.py:111: RuntimeWarning:
*******************************************************************************
libedit detected - readline will not be well behaved, including but not limited to:
   * crashes on tab completion
   * incorrect history navigation
   * corrupting long-lines
   * failure to wrap or indent lines properly
It is highly recommended that you install readline, which is easy_installable:
     easy_install readline
Note that `pip install readline` generally DOES NOT WORK, because
it installs to site-packages, which come *after* lib-dynload in sys.path,
where readline is located.  It must be `easy_install readline`, or to a custom
location on your PYTHONPATH (even --user comes after lib-dyload).
*******************************************************************************
  RuntimeWarning)
[stderr:1]
/Library/Frameworks/EPD64.framework/Versions/7.3/lib/python2.7/site-
packages/IPython/utils/rlineimpl.py:111: RuntimeWarning:
*******************************************************************************
libedit detected - readline will not be well behaved, including but not limited to:
   * crashes on tab completion
   * incorrect history navigation
   * corrupting long-lines
   * failure to wrap or indent lines properly
It is highly recommended that you install readline, which is easy_installable:
     easy_install readline
Note that `pip install readline` generally DOES NOT WORK, because
it installs to site-packages, which come *after* lib-dynload in sys.path,
where readline is located.  It must be `easy_install readline`, or to a custom
location on your PYTHONPATH (even --user comes after lib-dyload).
```

```
********************************************************************************
  RuntimeWarning)
```

Creating a `numpy.memmap` instance with the `w+` mode creates a file on the filesystem and zeros its content. Let's do it from the first engine process or our current IPython cluster:

In [51]:
```
%%px --targets=-1

# Cleanup any existing file from past session (necessary for windows)
import os
if os.path.exists('small.mmap'):
    os.unlink('small.mmap')

mm_w = np.memmap('small.mmap', shape=10, dtype=np.float32, mode='w+')
print(mm_w)
# now this memory is shared among all clusters
```

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Assuming the notebook process was launched with:

```
cd solutions
ipython notebook
```

or:

```
cd notebooks
ipython notebook
```

and the cluster was launched from the ipython notebook UI, the engines will have a the same current working directory as the notebook process, hence we can find the `small.mmap` file the current folder:

In [53]:
```
!ls -lh small.mmap
# Basically save the data in the (ipython notebook folder, so all cluster can access
```

```
-rw-r--r--  1 alexwchen  staff    40B  9 Jun 16:28 small.mmap
```

This binary file can then be mapped as a new numpy array by all the engines having access to the same filesystem. The `mode='r+'` opens this shared memory area in read write mode:

In [54]:
```
%%px

mm_r = np.memmap('small.mmap', dtype=np.float32, mode='r+')
print(mm_r)
```

```
[stdout:0] [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
[stdout:1] [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

In [55]:
```
%%px --targets=-1

mm_w[0] = 42
print(mm_w)
print(mm_r)
# point to the same memory
```

```
[ 42.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[ 42.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
```

In [56]:
```
%px print(mm_r)
```

```
[stdout:0] [ 42.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[stdout:1] [ 42.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
```

Memory mapped arrays created with `mode='r+'` can be modified and the modifications are shared with all the engines:

In [57]:
```
%%px --targets=1

mm_r[1] = 43
```

In [58]:
```
%%px
print(mm_r)
```

```
[stdout:0] [ 42.  43.   0.   0.   0.   0.   0.   0.   0.   0.]
[stdout:1] [ 42.  43.   0.   0.   0.   0.   0.   0.   0.   0.]
```

Be careful those, there is no builtin read nor write lock available on this such datastructures so it's better to avoid concurrent read & write operations on the same array segments unless there engine operations are made to cooperate with some synchronization or scheduling orchestrator.

Memmap arrays generally behave very much like regular in-memory numpy arrays:

In [59]:
```
%%px
print("sum={0:.3}, mean={1:.3}, std={2:.3}".format(
    mm_r.sum(), np.mean(mm_r), np.std(mm_r)))
```

```
[stdout:0] sum=85., mean=8.5, std=17.
[stdout:1] sum=85., mean=8.5, std=17.
```

Before allocating more data in memory on the cluster let us define a couple of utility functions from the previous exercise (and more) to monitor what is used by which engine and what is still free on the cluster as a whole:

In [60]:
```
def get_engines_memory(client):
    """Gather the memory allocated by each engine in MB"""
    def memory_mb():
        import os
        import psutil
        return psutil.Process(os.getpid()).get_memory_info().rss / 1e6

    return client[:].apply(memory_mb).get_dict()

def get_host_free_memory(client):
    """Free memory on each host of the cluster in MB."""
    all_engines = client[:]
    def hostname():
        import socket
        return socket.gethostname()

    hostnames = all_engines.apply(hostname).get_dict()
    one_engine_per_host = dict((hostname, engine_id)
                               for engine_id, hostname
                               in hostnames.items())

    def host_free_memory():
        import psutil
        return psutil.virtual_memory().free / 1e6
```

```
    host_mem = client[one_engine_per_host.values()].apply(
        host_free_memory).get_dict()

    return dict((hostnames[eid], m) for eid, m in host_mem.items())
```

In [61]:
```
get_engines_memory(client)
```

Out[61]:
```
{0: 44.371968, 1: 44.404736}
```

In [62]:
```
get_host_free_memory(client)
```

Out[62]:
```
{'Alexander-Chens-MacBook-Pro.local': 820.256768}
```

Let's allocate a 80MB memmap array in the first engine and load it in readwrite mode in all the engines:

In [63]:
```
%%px --targets=-1

# Cleanup any existing file from past session (necessary for windows)
import os
if os.path.exists('big.mmap'):
    os.unlink('big.mmap')

np.memmap('big.mmap', shape=10 * int(1e6), dtype=np.float64, mode='w+')
```

Out[1:9]:
```
memmap([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

In [64]:
```
ls -lh big.mmap
```

```
-rw-r--r--  1 alexwchen  staff    76M  9 Jun 16:39 big.mmap
```

In [65]:
```
get_host_free_memory(client)
```

Out[65]:
```
{'Alexander-Chens-MacBook-Pro.local': 810.94656}
```

No significant memory was used in this operation as we just asked the OS to allocate the buffer on the hard drive and just maitain a virtual memory area as a cheap reference to this buffer.

Let's open new references to the same buffer from all the engines at once:

In [66]:
```
%px %time big_mmap = np.memmap('big.mmap', dtype=np.float64, mode='r+')
```

```
[stdout:0]
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.16 s
[stdout:1]
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.28 s
```

In [67]:
```
%px big_mmap
```

Out[0:7]:
```
memmap([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

Out[1:11]:
```
memmap([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

In [68]:
```
get_host_free_memory(client)
```

Out[68]:
```
{'Alexander-Chens-MacBook-Pro.local': 824.881152}
```

No physical memory was allocated in the operation as it just took a couple of ms to do so. This is also confirmed by the engines process stats:

In [69]:
```
get_engines_memory(client)
```

Out[69]:
```
{0: 44.474368, 1: 44.548096}
```

Let's trigger an actual load of the data from the drive into the in-memory disk cache of the OS, this can take some time depending on the speed of the hard drive (on the order of 100MB/s to 300MB/s hence 3s to 8s for this dataset):

In [70]:
```
%%px --targets=-1

%time np.sum(big_mmap)
```

```
CPU times: user 0.03 s, sys: 0.05 s, total: 0.08 s
Wall time: 1.03 s
Out[1:12]: memmap(0.0)
```

In [71]:
```
get_engines_memory(client) # mem used of last engine has increased
```

Out[71]:
```
{0: 44.478464, 1: 124.547072}
```

In [72]:
```
get_host_free_memory(client) # free mem of host decreased
```

Out[72]:
```
{'Alexander-Chens-MacBook-Pro.local': 729.829376}
```

We can see that the first engine has now access to the data in memory and the free memory on the host has decreased by the same amount.

We can now access this data from all the engines at once much faster as the disk will no longer be used: the shared memory buffer will instead accessed directly by all the engines:

In [73]:
```
%px %time np.sum(big_mmap)
```

```
[stdout:0]
CPU times: user 0.03 s, sys: 0.04 s, total: 0.07 s
Wall time: 0.17 s
[stdout:1]
CPU times: user 0.03 s, sys: 0.00 s, total: 0.03 s
Wall time: 0.06 s
Out[0:8]: memmap(0.0)

Out[1:13]: memmap(0.0)
```

In [74]:
```
get_engines_memory(client)
```

Out[74]:
```
{0: 124.489728, 1: 124.547072}
```

In [76]:
```
get_host_free_memory(client) # this shouldn't decrease much becus mem is shared
```

Out[76]:
```
{'Alexander-Chens-MacBook-Pro.local': 584.613888}
```

So it seems that the engines have loaded a whole copy of the data but this actually not the case as the total amount of free memory was not impacted by the parallel access to the shared buffer. Furthermore, once the data has been

preloaded from the hard drive using one process, all the of the other processes on the same host can access it almost instantly saving a lot of IO wait.

This strategy makes it very interesting to load the readonly datasets of machine learning problems, especially when the same data is reused over and over by concurrent processes as can be the case when doing learning curves analysis or grid search.

Memmaping Nested Numpy-based Data Structures with Joblib

joblib is a utility library included in the sklearn package. Among other things it provides tools to serialize objects that comprise large numpy arrays and reload them as memmap backed datastructures.

To demonstrate it, let's create an arbitrary python datastructure involving numpy arrays:

In [77]:
```
import numpy as np

class MyDataStructure(object):

    def __init__(self, shape):
        self.float_zeros = np.zeros(shape, dtype=np.float32)
        self.integer_ones = np.ones(shape, dtype=np.int64)

data_structure = MyDataStructure((3, 4))
data_structure.float_zeros, data_structure.integer_ones
```

Out[77]:
```
(array([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]], dtype=float32),
 array([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]))
```

We can now persist this datastructure to disk:

In [78]:
```
from sklearn.externals import joblib
# all these numpy var are mem mapped automatically becus of joblib load
joblib.dump(data_structure, 'data_structure.pkl')
```

Out[78]:
```
['data_structure.pkl',
 'data_structure.pkl_01.npy',
 'data_structure.pkl_02.npy']
```

In [79]:
```
!ls -l data_structure*
```

```
-rw-r--r--  1 alexwchen  staff  255  9 Jun 16:48 data_structure.pkl
-rw-r--r--  1 alexwchen  staff  128  9 Jun 16:48 data_structure.pkl_01.npy
-rw-r--r--  1 alexwchen  staff  176  9 Jun 16:48 data_structure.pkl_02.npy
```

A memmapped copy of this datastructure can then be loaded:

In [80]:
```
# mem mapped automatically
memmaped_data_structure = joblib.load('data_structure.pkl', mmap_mode='r+')
memmaped_data_structure.float_zeros, memmaped_data_structure.integer_ones
```

Out[80]:
```
(memmap([[ 0.,  0.,  0.,  0.],
```

```
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]], dtype=float32),
 memmap([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]]))
```

Memmaping CV Splits for Multiprocess Dataset Sharing

We can leverage the previous tools to build a utility function that extracts Cross Validation splits ahead of time to persist them on the hard drive in a format suitable for memmaping by IPython engine processes.

In [81]:
```
from sklearn.externals import joblib
from sklearn.cross_validation import ShuffleSplit
import os

def persist_cv_splits(X, y, n_cv_iter=5, name='data',
    suffix="_cv_%03d.pkl", test_size=0.25, random_state=None):
    """Materialize randomized train test splits of a dataset."""

    cv = ShuffleSplit(X.shape[0], n_iter=n_cv_iter,
        test_size=test_size, random_state=random_state)
    cv_split_filenames = []

    for i, (train, test) in enumerate(cv):
        cv_fold = (X[train], y[train], X[test], y[test])
        cv_split_filename = name + suffix % i
        cv_split_filename = os.path.abspath(cv_split_filename)
        joblib.dump(cv_fold, cv_split_filename)
        cv_split_filenames.append(cv_split_filename)

    return cv_split_filenames
```

Let's try it on the digits dataset, we can run this from the :

In [82]:
```
from sklearn.datasets import load_digits

digits = load_digits()
digits_split_filenames = persist_cv_splits(digits.data, digits.target,
    name='digits', random_state=42)
digits_split_filenames
```

Out[82]:
```
['/Users/alexwchen/Desktop/fastml_tutorial/parallel_ml_tutorial-
master/solutions/digits_cv_000.pkl',
 '/Users/alexwchen/Desktop/fastml_tutorial/parallel_ml_tutorial-
master/solutions/digits_cv_001.pkl',
 '/Users/alexwchen/Desktop/fastml_tutorial/parallel_ml_tutorial-
master/solutions/digits_cv_002.pkl',
 '/Users/alexwchen/Desktop/fastml_tutorial/parallel_ml_tutorial-
master/solutions/digits_cv_003.pkl',
 '/Users/alexwchen/Desktop/fastml_tutorial/parallel_ml_tutorial-
master/solutions/digits_cv_004.pkl']
```

In [83]:
```
ls -lh digits*
```

```
-rw-r--r--  1 alexwchen  staff   280B  9 Jun 16:51 digits_cv_000.pkl
```

```
-rw-r--r--  1 alexwchen  staff   674K  9 Jun 16:51 digits_cv_000.pkl_01.npy
-rw-r--r--  1 alexwchen  staff    11K  9 Jun 16:51 digits_cv_000.pkl_02.npy
-rw-r--r--  1 alexwchen  staff   225K  9 Jun 16:51 digits_cv_000.pkl_03.npy
-rw-r--r--  1 alexwchen  staff   3.6K  9 Jun 16:51 digits_cv_000.pkl_04.npy
-rw-r--r--  1 alexwchen  staff   280B  9 Jun 16:51 digits_cv_001.pkl
-rw-r--r--  1 alexwchen  staff   674K  9 Jun 16:51 digits_cv_001.pkl_01.npy
-rw-r--r--  1 alexwchen  staff    11K  9 Jun 16:51 digits_cv_001.pkl_02.npy
-rw-r--r--  1 alexwchen  staff   225K  9 Jun 16:51 digits_cv_001.pkl_03.npy
-rw-r--r--  1 alexwchen  staff   3.6K  9 Jun 16:51 digits_cv_001.pkl_04.npy
-rw-r--r--  1 alexwchen  staff   280B  9 Jun 16:51 digits_cv_002.pkl
-rw-r--r--  1 alexwchen  staff   674K  9 Jun 16:51 digits_cv_002.pkl_01.npy
-rw-r--r--  1 alexwchen  staff    11K  9 Jun 16:51 digits_cv_002.pkl_02.npy
-rw-r--r--  1 alexwchen  staff   225K  9 Jun 16:51 digits_cv_002.pkl_03.npy
-rw-r--r--  1 alexwchen  staff   3.6K  9 Jun 16:51 digits_cv_002.pkl_04.npy
-rw-r--r--  1 alexwchen  staff   280B  9 Jun 16:51 digits_cv_003.pkl
-rw-r--r--  1 alexwchen  staff   674K  9 Jun 16:51 digits_cv_003.pkl_01.npy
-rw-r--r--  1 alexwchen  staff    11K  9 Jun 16:51 digits_cv_003.pkl_02.npy
-rw-r--r--  1 alexwchen  staff   225K  9 Jun 16:51 digits_cv_003.pkl_03.npy
-rw-r--r--  1 alexwchen  staff   3.6K  9 Jun 16:51 digits_cv_003.pkl_04.npy
-rw-r--r--  1 alexwchen  staff   280B  9 Jun 16:51 digits_cv_004.pkl
-rw-r--r--  1 alexwchen  staff   674K  9 Jun 16:51 digits_cv_004.pkl_01.npy
-rw-r--r--  1 alexwchen  staff    11K  9 Jun 16:51 digits_cv_004.pkl_02.npy
-rw-r--r--  1 alexwchen  staff   225K  9 Jun 16:51 digits_cv_004.pkl_03.npy
-rw-r--r--  1 alexwchen  staff   3.6K  9 Jun 16:51 digits_cv_004.pkl_04.npy
```

Each of the persisted CV splits can then be loaded back again using memmaping:

In [84]:
```
# the above comment is super useful and impressive!
X_train, y_train, X_test, y_test = joblib.load(
    'digits_cv_002.pkl',  mmap_mode='r+')
```

In [85]:
```
X_train
```

Out[85]:
```
memmap([[  0.,   1.,  13., ...,   1.,   0.,   0.],
        [  0.,   0.,   7., ...,   9.,   0.,   0.],
        [  0.,   0.,   0., ...,  13.,   1.,   0.],
        ...,
        [  0.,   0.,   4., ...,  16.,   1.,   0.],
        [  0.,   0.,   2., ...,  15.,   8.,   0.],
        [  0.,   0.,   0., ...,   3.,   0.,   0.]])
```

In [86]:
```
y_train
```

Out[86]:
```
memmap([5, 3, 1, ..., 8, 6, 4])
```

Parallel Model Selection and Grid Search

Let's leverage IPython.parallel and the Memory Mapping features of joblib to write a custom grid search utility that runs on cluster in a memory efficient manner.

Assume that we want to reproduce the grid search from the previous session:

In [87]:
```
import numpy as np
from pprint import pprint
```

```
svc_params = {
    'C': np.logspace(-1, 2, 4),
    'gamma': np.logspace(-4, 0, 5),
}
pprint(svc_params)
```

```
{'C': array([   0.1,    1. ,   10. ,  100. ]),
 'gamma': array([  1.00000000e-04,   1.00000000e-03,   1.00000000e-02,
         1.00000000e-01,   1.00000000e+00])}
```

`GridSearchCV` internally uses the following `IterGrid` utility iterator class to build the possible combinations of parameters:

In [88]:
```
from sklearn.grid_search import IterGrid

list(IterGrid(svc_params))
```

Out[88]:
```
[{'C': 0.10000000000000001, 'gamma': 0.0001},
 {'C': 0.10000000000000001, 'gamma': 0.001},
 {'C': 0.10000000000000001, 'gamma': 0.01},
 {'C': 0.10000000000000001, 'gamma': 0.10000000000000001},
 {'C': 0.10000000000000001, 'gamma': 1.0},
 {'C': 1.0, 'gamma': 0.0001},
 {'C': 1.0, 'gamma': 0.001},
 {'C': 1.0, 'gamma': 0.01},
 {'C': 1.0, 'gamma': 0.10000000000000001},
 {'C': 1.0, 'gamma': 1.0},
 {'C': 10.0, 'gamma': 0.0001},
 {'C': 10.0, 'gamma': 0.001},
 {'C': 10.0, 'gamma': 0.01},
 {'C': 10.0, 'gamma': 0.10000000000000001},
 {'C': 10.0, 'gamma': 1.0},
 {'C': 100.0, 'gamma': 0.0001},
 {'C': 100.0, 'gamma': 0.001},
 {'C': 100.0, 'gamma': 0.01},
 {'C': 100.0, 'gamma': 0.10000000000000001},
 {'C': 100.0, 'gamma': 1.0}]
```

Let's write a function to load the data from a CV split file and compute the validation score for a given parameter set and model:

In [94]:
```
def compute_evaluation(cv_split_filename, model, params):
    """Function executed by a worker to evaluate a model on a CV split"""
    # All module imports should be executed in the worker namespace
    from sklearn.externals import joblib

    X_train, y_train, X_validation, y_validation = joblib.load(
        cv_split_filename, mmap_mode='c')

    model.set_params(**params)
    model.fit(X_train, y_train)
    validation_score = model.score(X_validation, y_validation)
    return validation_score
```

In [95]:
```
def grid_search(lb_view, model, cv_split_filenames, param_grid):
    """Launch all grid search evaluation tasks."""
    all_tasks = []
    all_parameters = list(IterGrid(param_grid))
```

```
    for i, params in enumerate(all_parameters):
        task_for_params = []

        for j, cv_split_filename in enumerate(cv_split_filenames):
            t = lb_view.apply(
                compute_evaluation, cv_split_filename, model, params)
            task_for_params.append(t)

        all_tasks.append(task_for_params)

    return all_parameters, all_tasks
```

Let's try on the digits dataset that we splitted previously as memmapable files:

In [96]:
```
from sklearn.svm import SVC
from IPython.parallel import Client

client = Client()
lb_view = client.load_balanced_view()
model = SVC()
svc_params = {
    'C': np.logspace(-1, 2, 4),
    'gamma': np.logspace(-4, 0, 5),
}

all_parameters, all_tasks = grid_search(
   lb_view, model, digits_split_filenames, svc_params)
```

The `grid_search` function is using the asynchronous API of the `LoadBalancedView`, we can hence monitor the progress:

In [97]:
```
def progress(tasks):
    return np.mean([task.ready() for task_group in tasks
                                 for task in task_group])
```

In [101]:
```
print("Tasks completed: {0}%".format(100 * progress(all_tasks)))
```

```
Tasks completed: 100.0%
```

Even better, we can introspect the completed task to find the best parameters set so far:

In [102]:
```
def find_bests(all_parameters, all_tasks, n_top=5):
    """Compute the mean score of the completed tasks"""
    mean_scores = []

    for param, task_group in zip(all_parameters, all_tasks):
        scores = [t.get() for t in task_group if t.ready()]
        if len(scores) == 0:
            continue
        mean_scores.append((np.mean(scores), param))

    return sorted(mean_scores, reverse=True)[:n_top]
```

In [103]:
```
from pprint import pprint
```

```
print("Tasks completed: {0}%".format(100 * progress(all_tasks)))
pprint(find_bests(all_parameters, all_tasks))
```

```
Tasks completed: 100.0%
[(0.99022222222222211, {'C': 1.0, 'gamma': 0.001}),
 (0.98888888888888893, {'C': 100.0, 'gamma': 0.001}),
 (0.98888888888888893, {'C': 10.0, 'gamma': 0.001}),
 (0.98755555555555552, {'C': 10.0, 'gamma': 0.0001}),
 (0.98711111111111127, {'C': 100.0, 'gamma': 0.0001})]
```

Optimization Trick: Truncated Randomized Search

It is often wasteful to search all the possible combinations of parameters as done previously, especially if the number of parameters is large (e.g. more than 3).

To speed up the discovery of good parameters combinations, it is often faster to randomized the search order and allocate a budget of evaluations, e.g. 10 or 100 combinations.

See this JMLR paper by James Bergstra for an empirical analysis of the problem. The interested reader should also have a look at hyperopt that further refines this parameter search method using meta-optimizers.

Randomized Parameter Search has just been implemented in the master branch of scikit-learn be part of the 0.14 release.

A More Complete Parallel Model Selection and Assessment Example

In [104]:
```
%pylab inline
import pylab as pl
import numpy as np

# Some nice default configuration for plots
pl.rcParams['figure.figsize'] = 10, 7.5
pl.rcParams['axes.grid'] = True
pl.gray()
```

```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

In [105]:
```
lb_view = client.load_balanced_view()
model = SVC()
```

In [106]:
```
import sys
from collections import OrderedDict
sys.path.append('..')
import model_selection, mmap_utils
reload(model_selection), reload(mmap_utils)

lb_view.abort()

svc_params = OrderedDict([
    ('gamma', np.logspace(-4, 0, 5)),
    ('C', np.logspace(-1, 2, 4)),
```

```
])

search = model_selection.RandomizedGridSeach(lb_view)
search.launch_for_splits(model, svc_params, digits_split_filenames)
```

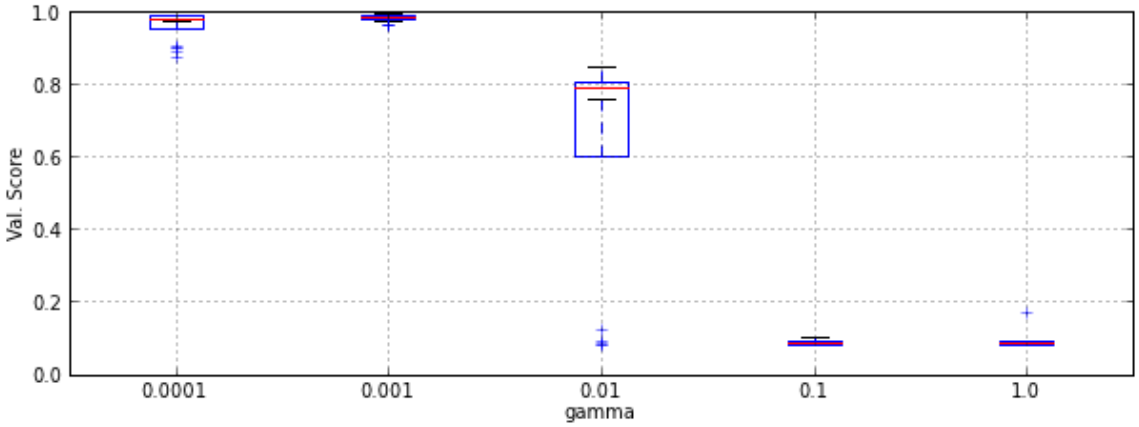
Out[106]: Progress: 00% (000/100)

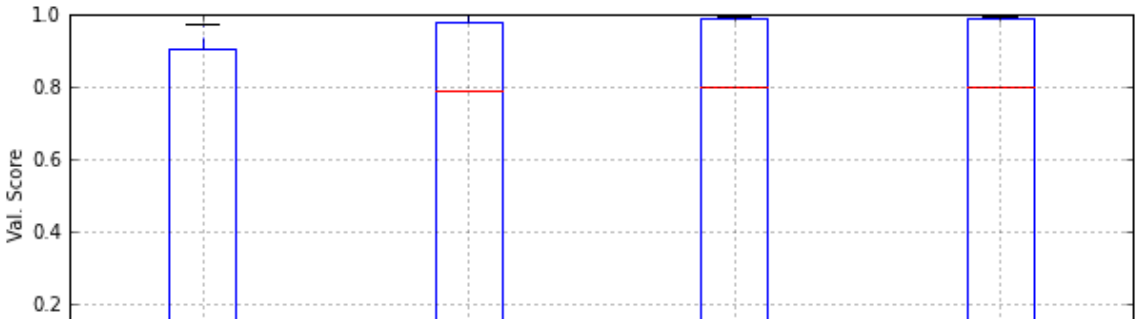
In [109]:
```
print(search.report())
```

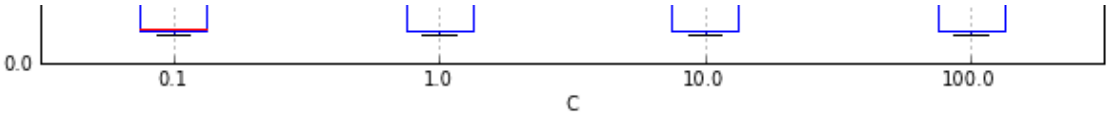
```
Progress: 100% (100/100)

Rank 1: validation: 0.99022 (+/-0.00151) train: 0.99896 (+/-0.00038):
 {'C': 1.0, 'gamma': 0.001}
Rank 2: validation: 0.98889 (+/-0.00157) train: 1.00000 (+/-0.00000):
 {'C': 100.0, 'gamma': 0.001}
Rank 3: validation: 0.98889 (+/-0.00157) train: 1.00000 (+/-0.00000):
 {'C': 10.0, 'gamma': 0.001}
Rank 4: validation: 0.98756 (+/-0.00194) train: 0.99733 (+/-0.00018):
 {'C': 10.0, 'gamma': 0.0001}
Rank 5: validation: 0.98711 (+/-0.00178) train: 1.00000 (+/-0.00000):
 {'C': 100.0, 'gamma': 0.0001}
```

In [110]:
```
print(search.report())
search.boxplot_parameters(display_train=False)
```

```
Progress: 100% (100/100)

Rank 1: validation: 0.99022 (+/-0.00151) train: 0.99896 (+/-0.00038):
 {'C': 1.0, 'gamma': 0.001}
Rank 2: validation: 0.98889 (+/-0.00157) train: 1.00000 (+/-0.00000):
 {'C': 100.0, 'gamma': 0.001}
Rank 3: validation: 0.98889 (+/-0.00157) train: 1.00000 (+/-0.00000):
 {'C': 10.0, 'gamma': 0.001}
Rank 4: validation: 0.98756 (+/-0.00194) train: 0.99733 (+/-0.00018):
 {'C': 10.0, 'gamma': 0.0001}
Rank 5: validation: 0.98711 (+/-0.00178) train: 1.00000 (+/-0.00000):
 {'C': 100.0, 'gamma': 0.0001}
```







In [111]:
```
#search.abort()
```

Distributing the Computation on EC2 Spot Instances with StarCluster

Installation

To provision a cheap transient compute cluster on Amazon EC2, the first step is to register on EC2 with a credit card and put your EC2 credentials as environment variables. For instance under Linux / OSX:

```
[laptop]% export AWS_ACCESS_KEY_ID=XXXXXXXXXXXXXXXXXXXX
[laptop]% export AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

You can put those exports in your `~/.bashrc` to automatically get those credentials loaded in new shell sessions.

Then proceed to the installation of StarCluster it-self. We will need plugins that have been contributed to the `develop` branch after the release of StarCluster 0.93.3. So if 0.94 or later is released you can do:

```
[laptop]% pip install StarCluster
```

In the mean time, install from the development branch:

```
[laptop]% pip install git+git://github.com/jtriley/StarCluster.git
```

Configuration

Let's run the help command a first time and create a template configuration file:

```
[laptop]% starcluster help
StarCluster - (http://star.mit.edu/cluster)
Software Tools for Academics and Researchers (STAR)
Please submit bug reports to starcluster@mit.edu

cli.py:87 - ERROR - config file /home/user/.starcluster/config does not exist

Options:
--------
[1] Show the StarCluster config template
[2] Write config template to /home/user/.starcluster/config
[q] Quit

Please enter your selection:
2
```

and create a password-less ssh key that will be dedicated to this transient cluster:

```
[laptop]% starcluster createkey mykey -o ~/.ssh/mykey.rsa
```

You can now edit the file `/home/user/.starcluster/config` and remplace its content with the following sample configuration:

```
[global]
DEFAULT_TEMPLATE=iptemplate
REFRESH_INTERVAL=5

[key mykey]
KEY_LOCATION=~/.ssh/mykey.rsa

[plugin ipcluster]
SETUP_CLASS = starcluster.plugins.ipcluster.IPCluster
ENABLE_NOTEBOOK = True
NOTEBOOK_PASSWD = aaaa

[plugin ipclusterstop]
SETUP_CLASS = starcluster.plugins.ipcluster.IPClusterStop

[plugin ipclusterrestart]
SETUP_CLASS = starcluster.plugins.ipcluster.IPClusterRestartEngines

[plugin pypackages]
setup_class = starcluster.plugins.pypkginstaller.PyPkgInstaller
packages = scikit-learn, psutil

# Base configuration for IPython.parallel cluster
[cluster iptemplate]
KEYNAME = mykey
CLUSTER_SIZE = 1
CLUSTER_USER = ipuser
CLUSTER_SHELL = bash
REGION = us-east-1
NODE_IMAGE_ID = ami-5b3fb632     # REGION and NODE_IMAGE_ID go in pair
NODE_INSTANCE_TYPE = c1.xlarge   # 8 CPUs
DISABLE_QUEUE = True             # We don't need SGE, faster cluster startup
PLUGINS = pypackages, ipcluster
```

Launching a Cluster

Start a new cluster using the `myclustertemplate` section of the `~/.startcluster/config` file:

```
[laptop]% starcluster start -c iptemplate -s 3 -b 0.5 mycluster
```

- The `-s` option makes it possible to select the number of EC2 instance to start.
- The `-b` option makes it possible to provision non-master instances on the Spot Instance market
- To also provision the master node on the Spot Instance market you can further add the `--force-spot-master` flag to the previous commandline.
- Provisioning Spot Instances is typically up to 5x cheaper than regular instances for largish instance types such as `c1.xlarge` but you run the risk of having your instances shut down if the price goes up. Also provisioning new instances on the Spot market can be slower: often a couple of minutes instead of 30s for On Demand instances.
- You can access the price history of spot instances of a specific region with:

```
[laptop]% starcluster -r us-west-1 spothistory c1.xlarge
StarCluster - (http://star.mit.edu/cluster) (v. 0.9999)
Software Tools for Academics and Researchers (STAR)
Please submit bug reports to starcluster@mit.edu
```

```
>>> Current price: $0.11
>>> Max price: $0.75
>>> Average price: $0.13
```

Connect to the master node via ssh:

```
[laptop]% starcluster sshmaster -A -u ipuser
```

- The `-A` flag makes it possible to use your local ssh agent to manage your keys: makes it possible to `git clone` / `git push` github repositories from the master node as you would from your local folder.
- The StarCluster AMI comes with `tmux` installed by default.

It is possible to ssh into other cluster nodes from the master using local DNS aliases such as:

```
[myuser@master]% ssh node001
```

Dynamically Resizing the Cluster

When using the `LoadBalancedView` API of `IPython.parallel.Client` is it possible to dynamically grow the cluster to shorten the duration of the processing of a queue of task without having to restart from scratch.

This can be achieved using the `addnode` command, for instance to add 3 more nodes using $0.50 bid price on the Spot Instance market:

```
[laptop]% starcluster addnode -s 3 -b 0.5 mycluster
```

Each node will automatically run the `IPCluster` plugin and register new `IPEngine` processes to the existing `IPController` process running on master.

It is also possible to terminate individual running nodes of the cluster with `removenode` command but this will kill any task running on that node and IPython.parallel will **not** restart the failed task automatically.

Terminating a Cluster

Once your are done with your computation, don't forget to shutdown the whole cluster and EBS volume so as to only pay for the resources you used.

Before doing so, don't forget to backup any result file you would like to keep, by either pushing them to the S3 storage service (recommended for large files that you would want to reuse on EC2 later) or fetching them locally using the `starcluster get` command.

The cluster shutdown itself can be achieved with a single command:

```
[laptop]% starcluster terminate mycluster
```

Alternatively to can also keep your data by preserving the EBS volume attached to the master node by remplacing the `terminate` command with the `stop` command:

```
[laptop]% starcluster stop mycluster
```

You can then later restart the same cluster again with the `start` command to automatically remount the EBS volume.