Introduction to IPython Parallel Computing

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Topic Overview

- The AsyncResult object
- Using MPI with IPython
- Starting the IPython controller and engines
- A true cluster

Load Balanced View –revisit

- We now go over an example obatined from `https://blog.liang2.tw/2015Talk-IPython-Parallel/?full#cover`
- A deliberately setup task is created, which determines each task's work load by numbering of the task.
- Light load takes 200ms while heavy one takes a second.
- However only the first bunch of them are heavy load.
- We assign 5 heavy tasks and 22 light tasks among 5 engines.
- While tasks are mapped acrossed engines, engine 0 will take all heavy tasks, i.e., distributed in $\{6, 6, 5, 5, 5\}$ fashion.
- The differences of direct view and load balanced view are then obvious.

Load Balanced View –revisit

- We assign 5 heavy tasks and 22 light tasks among 5 engines.
- While tasks are mapped acrossed engines, engine 0 will take all heavy tasks, i.e., distributed in {6, 6, 5, 5, 5} fashion.

[7]:

```
num_engines = len(rc.ids)
total_works = num_engines * 5 + 2

def high_variated_work(i):
    start_time = datetime.now()
    base_workload = 10
    if i  < (total_works // num_engines):
        base_workload *= 5
    # so short task: 200ms long task: 1s
    sleep(base_workload * 20 / 1000)
    end_time = datetime.now()
    return pid, start_time, end_time
```

Timing

- For use after the tasks are done:
  - `ar.serial_time` is the sum of the computation time of all of the tasks done in parallel.
  - `ar.wall_time` is the time between the first task submitted and last result received. This is the actual cost of computation, including IPython overhead.

- An often used metric is the time it cost to do the work in parallel relative to the serial computation, and this can be given with

```
[7]: speedup = ar.serial_time / ar.wall_time
```

Load Balanced View –revisit

[8]:

```
dview['num_engines'] = num_engines
dview['total_works'] = total_works
```

[9]:

```
# lb: load balance, ar: async result
lb_ar = lbview.map_async(high_variated_work,␣
 ↪range(total_works))
```

[11]:

```
# d: direct
d_ar = dview.map_async(high_variated_work,␣
 ↪range(total_works))
```

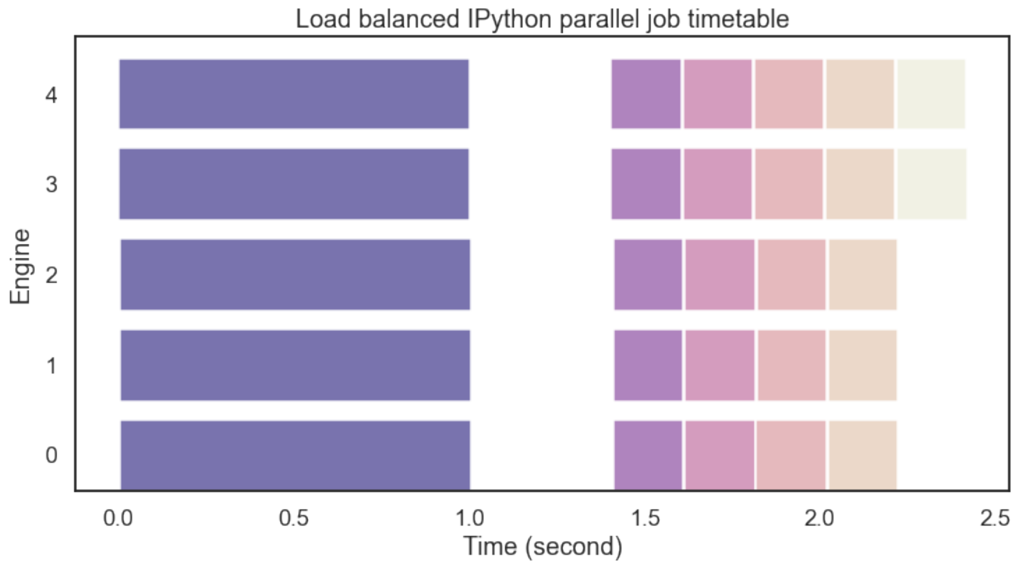
[19]:

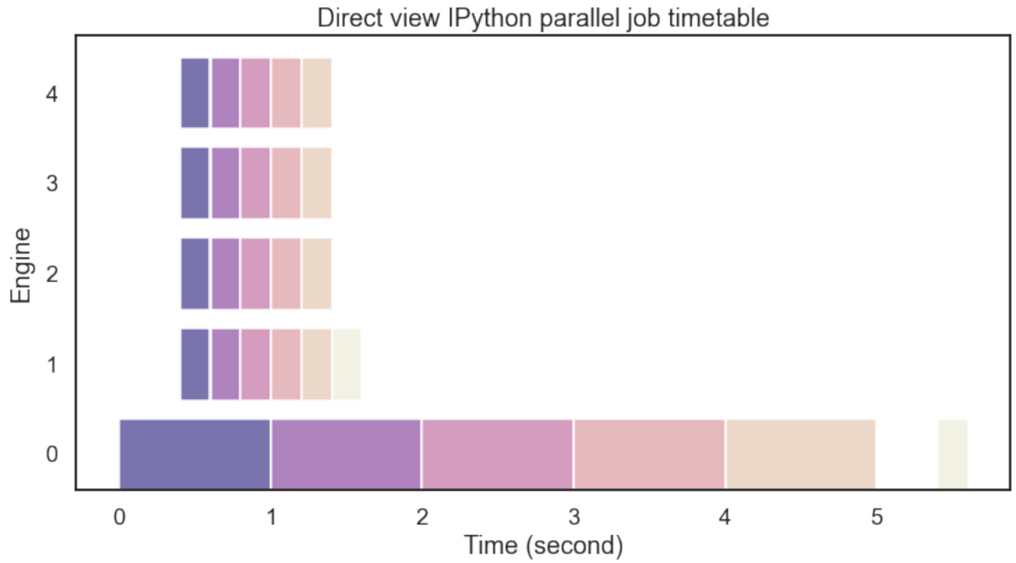
```
speedup = lambda ar: ar.serial_time / ar.wall_time
print(
    'Speedup for load-blanced and direct view␣
 ↪(ideal: {:d}x): {:.2f} and {:.2f}'
    .format(num_engines, *map(speedup, [lb_ar,␣
 ↪d_ar]))
)
```

```
Speedup for load-blanced and direct view (ideal:␣
 ↪5x): 4.64 and 1.81
```

Load Balanced View –revisit

- The spaces between adjacent tasks reflect the time needed to schedule the next tasks.





A parallel example – 50 million digits of π

- In this example we would like to study the distribution of digits in the number π (in base 10).

- While it is not known if π is a normal number (a number is normal in base 10 if 0-9 occur with equal likelihood) numerical investigations suggest that it is.

- We begin with a serial calculation on 10,000 digits of π.

- For the serial calculation, we use SymPy to calculate 10,000 digits of π and then look at the frequencies of the digits 0-9. Out of 10,000 digits, we expect each digit to occur 1,000 times.

- SymPy is a Python library for symbolic mathematics.

- While SymPy is capable of calculating many more digits of π, our purpose here is to set the stage for the much larger parallel calculation.

A serial calculation on 10,000 digits of π

We enable matplotlib integration for plotting.

[2]:
```
%matplotlib inline
import sympy
import numpy as np
from matplotlib import pyplot as plt

pi = sympy.pi.evalf(40)
pi
```

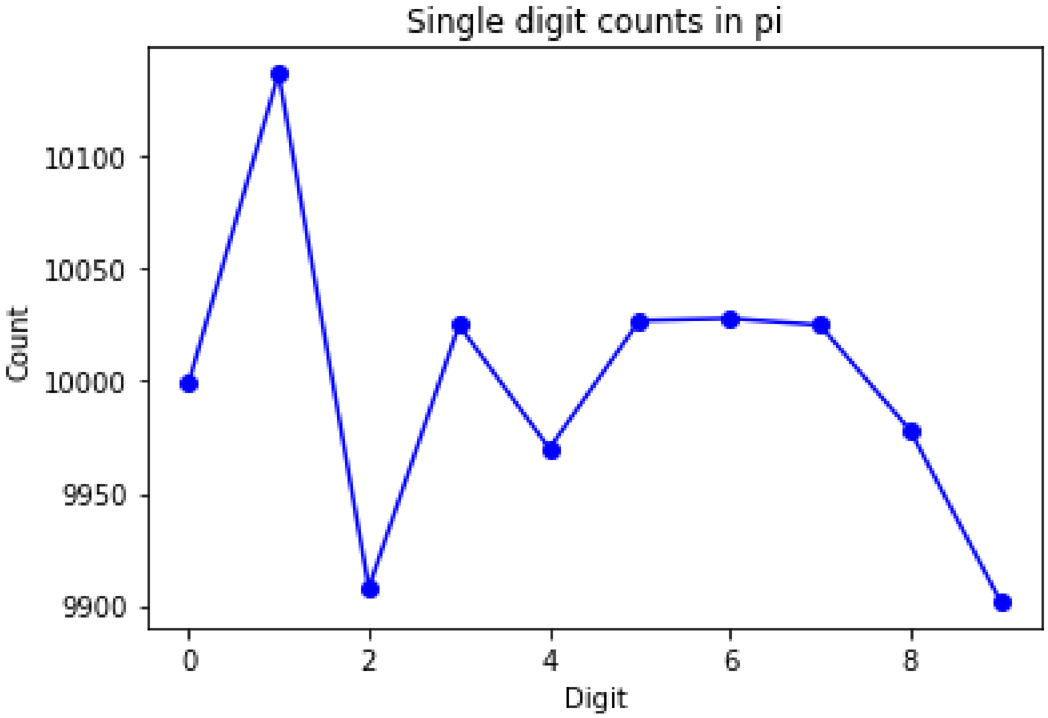
```
Using matplotlib backend: Qt5Agg
```

[2]: 3.141592653589793238462643383279502884197

A serial calculation on 10,000 digits of π –cont.

```
[4]: pi = sympy.pi.evalf(100000)
     digits = (d for d in str(pi)[2:])   # create a␣
      ↪sequence of digits (excluding 3.)
     freqs = one_digit_freqs(digits)
     plot_one_digit_freqs(freqs)
```

```
[4]: [<matplotlib.lines.Line2D at 0x2dd3bd0d860>]
```



A parallel calculation on 50 million digits of π

- We place 5 files, each with 10 million digits and file name `pi1000m.ascii.xxof100`.

- First, we use 1 engine to read digits of π, and compute the 2 digit frequencies.

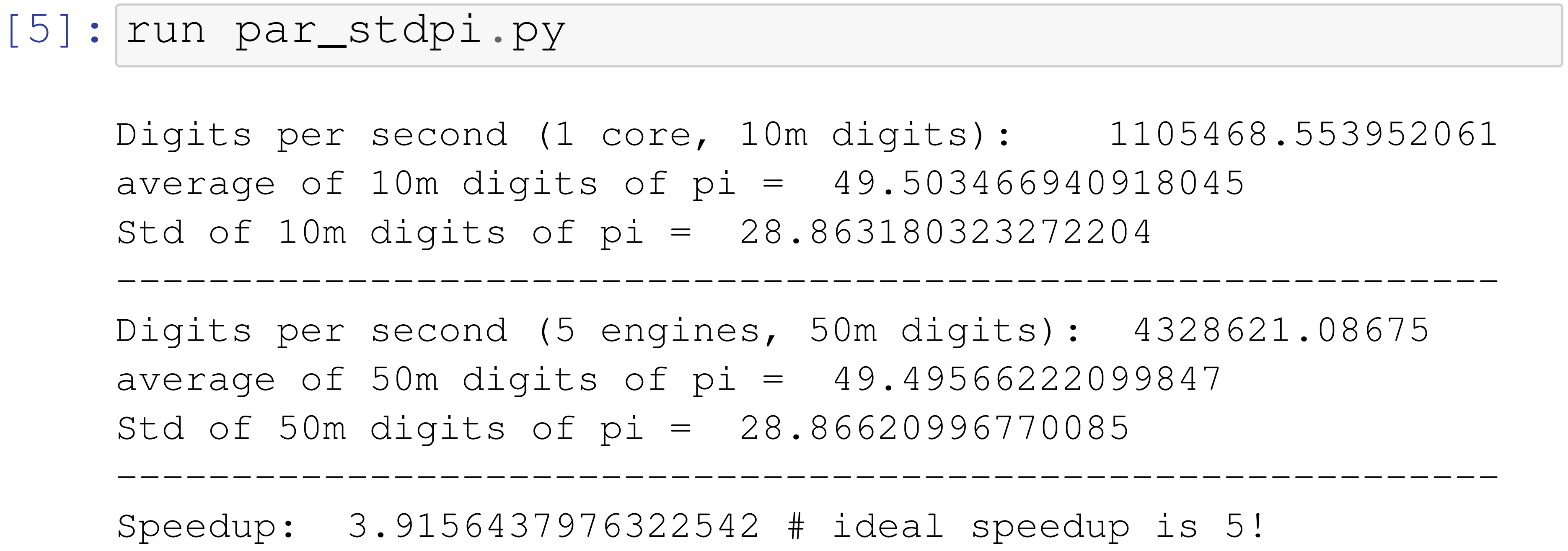
- We then compute the average and standard deviation for the 10m digits.

- For parallel run, we use 5 engines to read digits of π seperately, and compute the 2 digit frequencies.

- There are 5 engines + 1 controller = 6 cores (total cores in my machine).

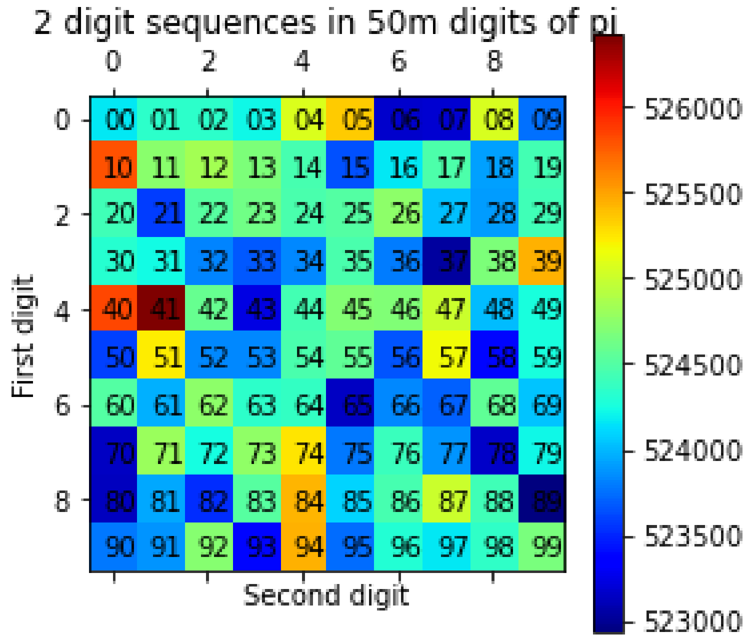
- We aggregate the frequencies from 50m digits and compute their average and standard deviation.

A parallel calculation on 50 million digits of π –cont.



```
[5]: run par_stdpi.py
```

```
Digits per second (1 core, 10m digits):    1105468.553952061
average of 10m digits of pi =  49.50346940918045
Std of 10m digits of pi =  28.86318032327204
---------------------------------------------------------------
Digits per second (5 engines, 50m digits):  4328621.08675
average of 50m digits of pi =  49.49566222099847
Std of 50m digits of pi =  28.86620996770085
---------------------------------------------------------------
Speedup:  3.9156437976322542 # ideal speedup is 5!
```



A parallel calculation on 50 million digits of π –cont.

```
# par_stdpi.py, stdpidigits.py
# One core code, we use apply
freqs10m = c[id0].apply_sync(compute_two_digit_freqs,␣
 ↪files[0])
Sum = freqs10m*LL

# freqs10m keeps the freqs of the occurance of each number
# from 00 to 99. freqs10m.sum of it gives the total number
# of digits.

avg10m = Sum.sum()/freqs10m.sum()
# compute the STD after mean is obtained
stdL = (LL - avg10m) ** 2
Stdsum = freqs10m*stdL
std10m = math.sqrt(Stdsum.sum()/freqs10m.sum())

# 5 engines code, we use map
freqs_all = v.map(compute_two_digit_freqs, files[:n])
# aggregation occurs in IPython session
# using function reduce_freqs
freqs150m = reduce_freqs(freqs_all)
```

A parallel calculation on 50 million digits of π –cont.

- Homework: Write the code to compute the frequencies of $\{0, 1, 2, 3, \cdots 9\}$ from 50 million digits and include the code in the par_stdpi.py

Using MPI with IPython

- Often, a parallel algorithm will require moving data between the engines. One way of accomplishing this is by doing a `pull` and then a `push` using the direct view, i.e. communcating through IPython session.

- However, this will be slow as all the data has to go through the controller to the client and then back through the controller, to its final destination.

- A much better way of moving data between engines is to use a message passing library, such as the Message Passing Interface (MPI).

- IPython's parallel computing architecture has been designed from the ground up to integrate with MPI.

Using MPI with IPython–cont.

- If you want to use MPI with IPython, you will need to install:
  - A standard MPI implementation such as OpenMPI (OpenMPI) or MPICH. We use MS-MPI. See `HowToRunMSMPICode.doc`.
  - The mpi4py (mpi4py) package. This could be done through Anaconda. Make sure you use Python 3.7 (currectly, 3.8 is not supported).

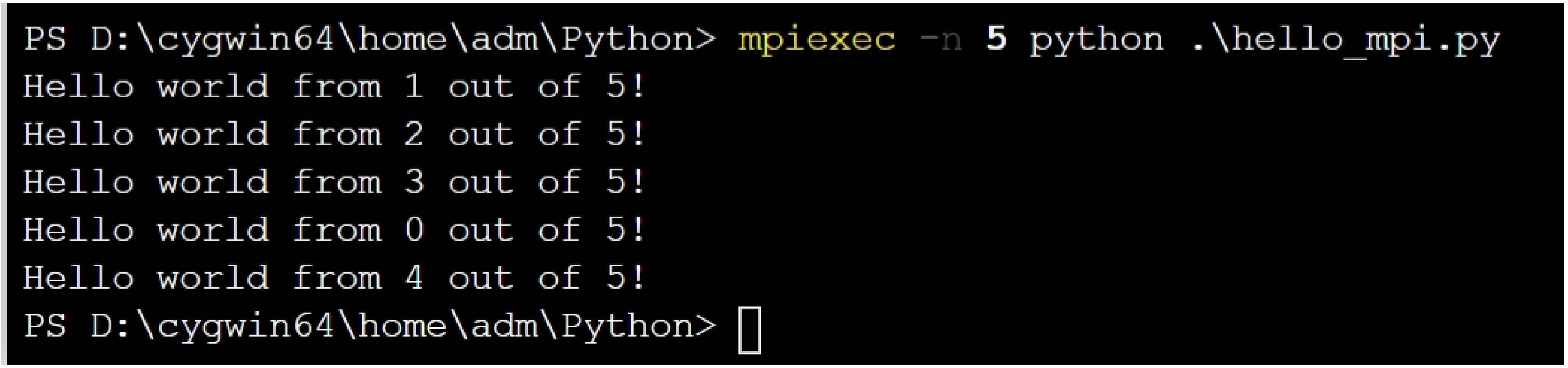
- Hello World! Again & again.

```
# hello_mpi.py
# Hello World ! Again
from mpi4py import MPI

comm   = MPI.COMM_WORLD
npes   = comm.Get_size()
myrank = comm.Get_rank()
print("Hello world from %i out of %i!"
      %(myrank, npes))
```

Using MPI with IPython–cont.

- We submit the job by `mpiexec -n 5 python .\hello_mpi.py`, here we request 5 engines.
- It appears that you could ask as many (virtual) engines as you wish.



```
PS D:\cygwin64\home\adm\Python> mpiexec -n 5 python .\hello_mpi.py
Hello world from 1 out of 5!
Hello world from 2 out of 5!
Hello world from 3 out of 5!
Hello world from 0 out of 5!
Hello world from 4 out of 5!
PS D:\cygwin64\home\adm\Python> 
```

Starting the engines with MPI enabled

- We now wish to invoke MPI code within IPython session, i.e. we wish to start a cluster equipped with MPI support.

- To use code that calls MPI, there are typically two things that MPI requires.
  - The process that wants to call MPI must be started using `mpiexec` that has MPI support.
  - Once the process starts, it must call `MPI_Init()`.

- The easiest approach is to use the MPI Launchers in `ipcluster`, which will first start a controller and then a set of engines using `mpiexec`.

Configuring an IPython cluster

- Cluster configurations are stored as profiles. You can create a new profile (e.g. mpi3) with:

```
$ ipython profile create --parallel --profile=mpi
```

- This will create the directory `IPYTHONDIR/profile_mpi3`, and populate it with the default configuration files for the three IPython cluster commands.

- Once you edit those files, you can continue to call `ipcluster/ipcontroller/ipengine` with no arguments beyond `profile=mpi3`, and any configuration will be maintained.

- Edit the file `IPYTHONDIR/profile_mpi/ipcluster_config.py`.

- Instruct ipcluster to use the MPI launchers by adding the lines:

```
  c.IPClusterEngines.engine_launcher_class =␣
↪'MPIEngineSetLauncher'
  c.BaseParallelApplication.cluster_id = 'mpi3'
```

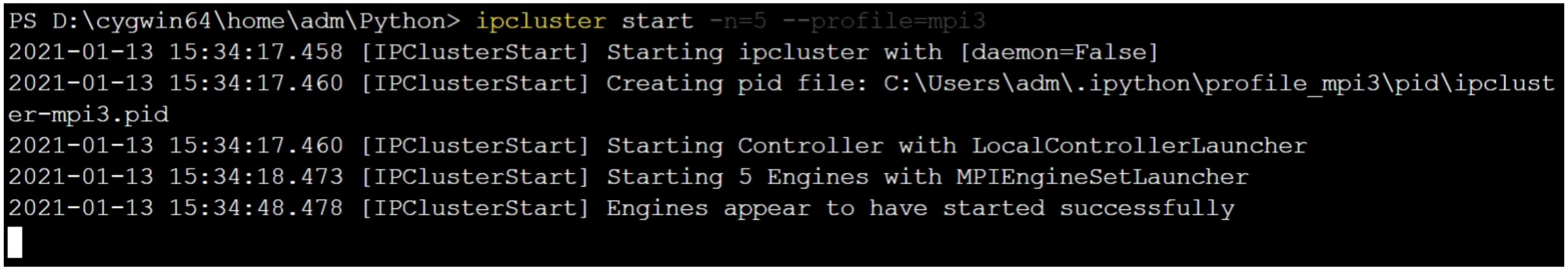
Configuring an IPython cluster

- The issue of cluster_id is not present when local clusters are launched, i.e. void is acceptable.

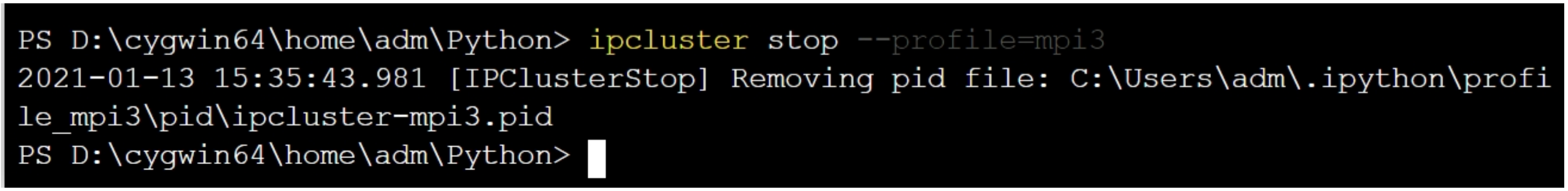
- Now, you can start a cluster with 5 engines by

  ```
  $ ipcluster start --n=5 --profile=mpi3
  ```

- You could start a cluster with as many (virtual) engines as you wish, however, it is better to have one core for controller and one core for each engine. The sum of them should not be grather than the number of cores in your CPU.



```
PS D:\cygwin64\home\adm\Python> ipcluster start -n=5 --profile=mpi3
2021-01-13 15:34:17.458 [IPClusterStart] Starting ipcluster with [daemon=False]
2021-01-13 15:34:17.460 [IPClusterStart] Creating pid file: C:\Users\adm\.ipython\profile_mpi3\pid\ipclust
er-mpi3.pid
2021-01-13 15:34:17.460 [IPClusterStart] Starting Controller with LocalControllerLauncher
2021-01-13 15:34:18.473 [IPClusterStart] Starting 5 Engines with MPIEngineSetLauncher
2021-01-13 15:34:48.478 [IPClusterStart] Engines appear to have started successfully
```



```
PS D:\cygwin64\home\adm\Python> ipcluster stop --profile=mpi3
2021-01-13 15:35:43.981 [IPClusterStop] Removing pid file: C:\Users\adm\.ipython\profi
le_mpi3\pid\ipcluster-mpi3.pid
PS D:\cygwin64\home\adm\Python>
```

When MPI meets IPython

```
[1]: import ipyparallel as ipp
     import numpy as np
     c = ipp.Client(profile='mpi3', cluster_id='mpi3')
     print(c.ids)
     dview = c[:]
```

```
     [0, 1, 2, 3, 4]
```

We deploy 32 double-floating numbers on purpose, so that the length of a in each engine is different.

```
[2]: dview.scatter('a',np.arange(32,dtype='float64'))
```

```
[2]: <AsyncResult: scatter>
```

```
[3]: dview['a']
```

```
[3]: [array([0., 1., 2., 3., 4., 5., 6.]),
      array([ 7.,  8.,  9., 10., 11., 12., 13.]),
      array([14., 15., 16., 17., 18., 19.]),
      array([20., 21., 22., 23., 24., 25.]),
      array([26., 27., 28., 29., 30., 31.])]
```

When MPI meets IPython–cont.

```
[4]: # run the contents of the file on each engine:
     dview.run('psum.py')
```

```
[4]: <AsyncResult: execute>
```

```
[5]: %px totalsum = psum(a)
```

```
[6]: dview['totalsum']
```

```
[6]: [array(496.), array(496.), array(496.), array(496.
      ↪), array(496.)]
```

When MPI meets IPython–cont.

- Although psum.py is a Python code, it actually calls the MPI library. So it follows the syntax of mpi, i.e. (address, count, datatype)
- `np.array(0.0, 'float64')` means a double precision array.

```
from mpi4py import MPI
import numpy as np

def psum(a):
    locsum = np.sum(a)
    rcvBuf = np.array(0.0,'float64')
    MPI.COMM_WORLD.Allreduce([locsum, MPI.DOUBLE],
        [rcvBuf, MPI.DOUBLE], op=MPI.SUM)
    return rcvBuf
```

When MPI meets IPython–cont.

- Let's get more information!

```
[7]: dview.run('rank_mpi.py')
```

```
[7]: <AsyncResult: execute>
```

```
# rank_mpi.py
from mpi4py import MPI
import numpy as np; import socket; import os

pid  = os.getpid()
comm = MPI.COMM_WORLD
host = socket.gethostname()

def myrank():
    return comm.rank # NOT comm.Get_rank()

def hostname():
    return host

def mypid():
    return pid
```

When MPI meets IPython–cont.

- Note that we use `comm.rank` in this example, lower case commands are Python commands, not MPI origional.

```
[8]: %px totalrank = myrank()
```

```
[9]: dview['totalrank']
```

```
[9]: [0, 1, 2, 3, 4]
```

```
[10]: %px totalhost = hostname()
```

```
[11]: dview['totalhost']
```

```
[11]: ['DESKTOP-10711',
       'DESKTOP-10711',
       'DESKTOP-10711',
       'DESKTOP-10711',
       'DESKTOP-10711']
```

When MPI meets IPython–cont.

```
[12]: %px totalpid = mypid()
```

```
[13]: dview['totalpid']
```

```
[13]: [19812, 5968, 21256, 22916, 24328]
```

```
# rank_mpi.py
d = np.zeros(1, dtype='float64')

def mysendOneToZero():
    if comm.rank == 1:
        comm.Send([a, 1, MPI.DOUBLE], dest = 0, tag=1)
    if comm.rank == 0:
        comm.Recv(d, source = 1, tag=1)
    return True
```

```
[15]: dview['d']
```

```
[15]: [array([7.]), array([0.]), array([0.]), array([0.
 ↪]), array([0.])]
```

- Note that we are not allowed to update variable ′a′ in engine.

When MPI meets IPython–cont.

- There are fundamental differences in implementation models between `mpiexec -n 5 python .\hello_mpi.py` and `dview.run('psum.py')`.

- Using `mpiexec` to implement a Python code is more like running a c/fortran mpi code, but now the code is in Python.

- However, `dview.run('psum.py')` is executing `psum.py` code parallelly in each engine from IPython session. But the results might not show up in the IPython session unless there are return values in the called functions.

  The printing result did not show up in the IPython session.

```
[16]: dview.run('mpi1_py/blocking_send.py')
```

```
[16]: <AsyncResult: execute:finished>
```

When MPI meets IPython–cont.

Even though, the results did not show up in the IPython session, but indeed, the code is executed.

```
[17]: dview['bb']
```

```
[17]: [array([3]), array([3]), array([0]), array([0]),␣
       ↪array([0])]
```

```
# blocking_send.py
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD; npes = comm.Get_size(); myrank = comm.Get_rank()

aa = numpy.zeros(1, dtype='i')
bb = numpy.zeros(1, dtype='i')
if myrank == 1:
    aa[0] = 2; bb[0] = 3
    comm.Send(aa, dest = 0, tag=1)
    comm.Send([bb, 1, MPI.INT], dest = 0, tag=2)
if myrank == 0:
    aa[0]  = 0; bb[0]  = 1
    comm.Recv(bb, source = 1, tag=2)
    comm.Recv(aa, source = 1, tag=1)
    print("Process", myrank, "received  aa[0]", aa[0] )
    print("Process", myrank, "received  bb[0]", bb[0] )
```

When MPI meets IPython–cont.

- There are two sets from mpi calls, the ones with upper-case like `comm.Send` and `comm.Recv` are for communication of buffer-like objects (e.g. arrays). These are more like c/fortran mpi calls.

- Buffer arguments to these calls must be explicitly specified by using a 2/3-list/tuple like `[data, MPI.DOUBLE]`, or `[data, count, MPI.DOUBLE]`.

- There is one more issue that I do NOT have answers yet! The `MPI.COMM_WORLD` is fixed when we launch the cluster.

- There are ways to launch new engines, and we can include the new engines to the `Client` as long as we rerun the `ipp.Client()`. However, there will be different `MPI.COMM_WORLD` among those engines groups. We will come back to this point later.

When MPI meets IPython–cont.

- The ones with lower case like `comm.send` and `comm.recv` are for communication of generic Python objects.

```
if  myrank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif  myrank == 1:
    data = comm.recv(source=0, tag=11)
    print("Process", myrank, "data = ", data )
```

```
PS D: mpiexec -n 2 python ./mpi1_py/sendrecv.py
Process 1 data =  {'a': 7, 'b': 3.14}
```

mpi4py

```
a = numpy.zeros(4, dtype='i')
b = numpy.zeros(1, dtype='i')
c = numpy.zeros(2, dtype='i')
d = numpy.zeros(4, dtype='i')

rec = numpy.zeros(8, dtype='i')

a[0]=a[2]=myrank+1
a[1]=a[3]=myrank*myrank+1

c[0]=myrank+1
c[1]=myrank*myrank+1
```

```
comm.Reduce([a, 2, MPI.INT], rec, MPI.SUM, 0)
if myrank == 0:
   print("Process", myrank, "rec", rec[0], rec[1], rec[2], rec[3])
```

```
cpu|    0    |    1    |    2    |    3      |
----------------------------------------------
a  | 1 1 1 1 | 2 2 2 2 | 3 5 3 5 | 4 10 4 10 |
rec|10 18 ? ?
```

mpi4py–cont.

- We need to be very careful about the variable sizes.

```
# We need a.size = 4 = npes*b.size
comm.Scatter([a, 1, MPI.INT], b, root = 3)
print("Process", myrank, "b", b[0])

cpu|    0    |    1    |    2    |    3      |
----------------------------------------------
a  | 1 1 1 1 | 2 2 2 2 | 3 5 3 5 | 4 10 4 10 |
b  | 4       |10       | 4       |10         |
```

- We need to be very careful about the variable sizes.

```
comm.Gather([c, 2, MPI.INT], rec, root = 0)
if myrank == 0:
    print("Process", myrank, "rec", rec[0], rec[1], \
         rec[2], rec[3], rec[4], rec[5], rec[6], rec[7])
cpu|    0    |    1    |    2    |    3      |
----------------------------------------------
c  | 1 1     | 2 2     | 3 5     | 4 10      |
rec| 1 1 2 2 3 5 4 10
```

mpi4py–cont.

```
comm.Alltoall([a, 1, MPI.INT], d)
print("Process", myrank, " Alltoall d", d[0], d[1],\
       d[2], d[3])

cpu|    0    |    1    |    2    |    3      |
----------------------------------------------
a  | 1 1 1 1 | 2 2 2 2 | 3 5 3 5 | 4 10 4 10 |
d  | 1 2 3 4 | 1 2 5 10| 1 2 3 4 | 1  2 5 10 |

if myrank == 0:
    b[0]= 1000

comm.Bcast(b, 0)
print("Process", myrank, "Bcast b", b[0])

cpu|    0    |    1    |    2    |    3      |
----------------------------------------------
b  |  1000   |    ?    |    ?    |    ?      |
b  |  1000   |  1000   |   1000  |    1000   |
```

mpi4py–cont.

- There are more commands!, e.q. `comm.Barrier()`.

- MPI2 commands:
  `MPI.File.Open(comm, filename, amode, info))`,
  `MPI.File.Write_all(offset, buffer)`.

```
amode = MPI.MODE_WRONLY|MPI.MODE_CREATE
comm = MPI.COMM_WORLD
fh = MPI.File.Open(comm, "./mpi_IO", amode)

buffer = np.empty(10, dtype=np.int)
buffer[:] = comm.Get_rank()

offset = comm.Get_rank()*buffer.nbytes
fh.Write_at_all(offset, buffer)

fh.Close()
```

mpi4py–cont.

```
PS D: mpiexec -n 4 python ./mpi1_py/mpi_io.py
```

- We have 10 integers with their rank from each engine, and use a MPI collect I/O to write them out as a “single” file!

```
[18]: f = open('mpi_IO', 'rb')
      a = np.fromfile(f, dtype=np.int)
      print(a)
```

```
[0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2␣
 ↪2 2 2 2 2 2 3 3 3 3 3 3 3
 3 3 3]
```

mpi4py for Python Objects

- Details, see `mpi_pyObj.py`
- It is more flexible to communicate generic Python objects comparing with buffer-like objetcs, i.e. c/fortran objetcs (numpy objects).
- `comm.send` & `comm.recv`

```
if  myrank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif  myrank == 1:
    data = comm.recv(source=0, tag=11)
    print("Process", myrank, "data = ", data )

Process 1 data =  {'a': 7, 'b': 3.14}
```

mpi4py for Python Objects–cont.

- comm.bcast

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

data = comm.bcast(data, root=0)
print("Process", myrank, "data = ", data )
```

```
Process 0 data =  'key1': [7, 2.72, (2+3j)],␣
 ↪'key2': ('abc', 'xyz')
Process 1 data =  'key1': [7, 2.72, (2+3j)],␣
 ↪'key2': ('abc', 'xyz')
Process 3 data =  'key1': [7, 2.72, (2+3j)],␣
 ↪'key2': ('abc', 'xyz')
Process 2 data =  'key1': [7, 2.72, (2+3j)],␣
 ↪'key2': ('abc', 'xyz')
```

mpi4py for Python Objects–cont.

- `comm.scatter`

```
if myrank == 0:
    data = [(i+1)**2 for i in range(npes)]
else:
    data = None

data = comm.scatter(data, root=0)
print("Process", myrank, "data = ", data )
```

```
Process 0 data =  1
Process 1 data =  4
Process 2 data =  9
Process 3 data =  16
```

mpi4py for Python Objects–cont.

- `comm.gather`

  ```
  data = (myrank+1)**2
  data = comm.gather(data, root=0)

  print("Process", myrank, "data = ", data )
  ```

  ```
  Process 2 data =  None
  Process 3 data =  None
  Process 1 data =  None
  Process 0 data =  [1, 4, 9, 16]
  ```

- Homework: Write a mpi Python code to compute variance of a set of numbers.

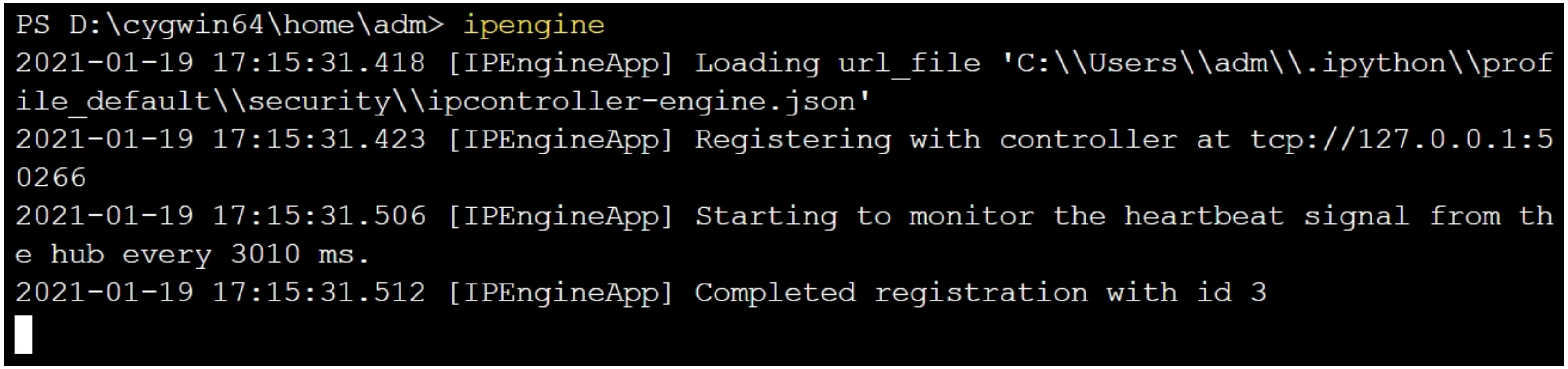
ipcluster revisit

- When we use `$ ipcluster start -n x` commands, we start the IPython controller on current host.

- We could manually add more engines to the current cluster using

  ```
  $ ipengine
  ```

- We start one more engine with id 3.



```
PS D:\cygwin64\home\adm> ipengine
2021-01-19 17:15:31.418 [IPEngineApp] Loading url_file 'C:\\Users\\adm\\.ipython\\prof
ile_default\\security\\ipcontroller-engine.json'
2021-01-19 17:15:31.423 [IPEngineApp] Registering with controller at tcp://127.0.0.1:5
0266
2021-01-19 17:15:31.506 [IPEngineApp] Starting to monitor the heartbeat signal from th
e hub every 3010 ms.
2021-01-19 17:15:31.512 [IPEngineApp] Completed registration with id 3
```

- You could use `ipcluster engines --n=x` to have more engines. (I did not try this, you should.)

More Engines

We start a cluster with 3 engines

```
[1]: import ipyparallel as ipp
     import numpy as np
     c = ipp.Client()
     print(c.ids)
```

```
[0, 1, 2]
```

Now, add one engine using ipengine

```
[2]: c = ipp.Client()
     print(c.ids)
```

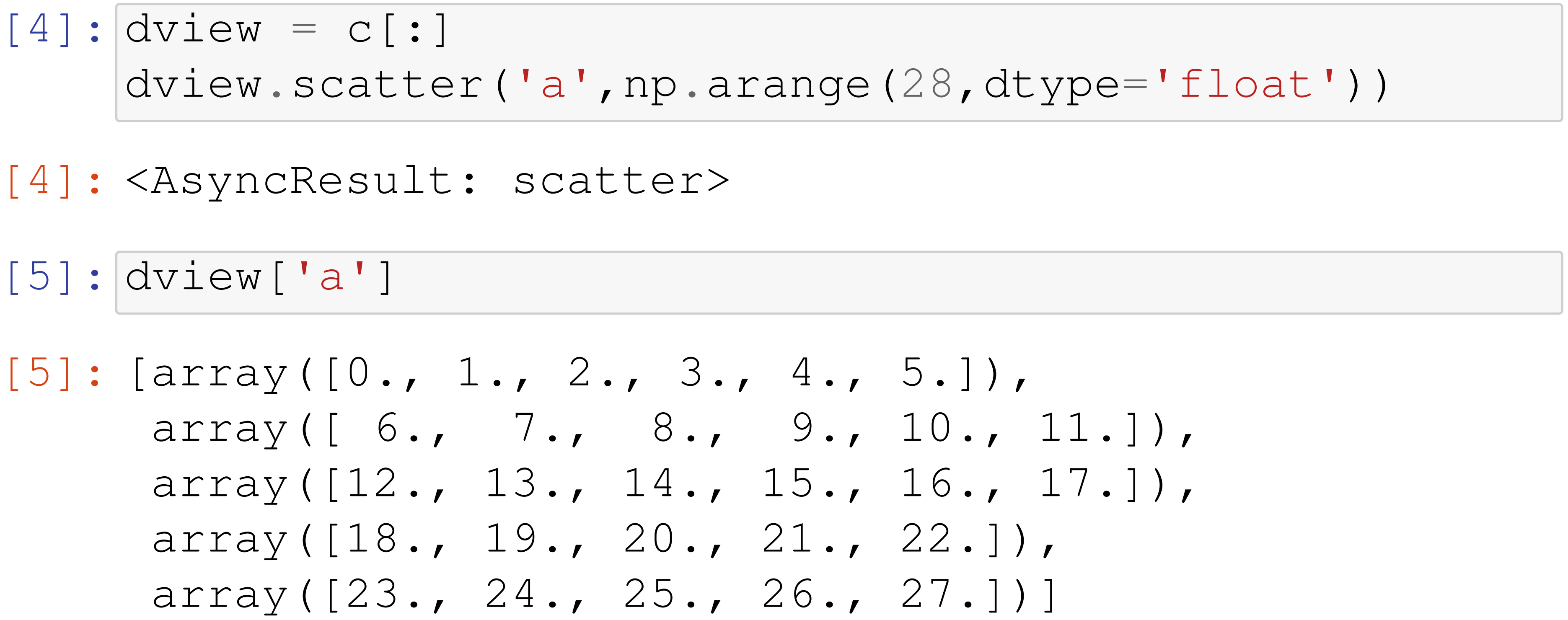
```
[0, 1, 2, 3]
```

We have now 4 engines!

```
[3]: c = ipp.Client()
     print(c.ids)
```

```
[0, 1, 2, 3, 4]
```

More Engines–cont.



```
[4]: dview = c[:]
     dview.scatter('a',np.arange(28,dtype='float'))

[4]: <AsyncResult: scatter>

[5]: dview['a']

[5]: [array([0., 1., 2., 3., 4., 5.]),
      array([ 6.,  7.,  8.,  9., 10., 11.]),
      array([12., 13., 14., 15., 16., 17.]),
      array([18., 19., 20., 21., 22.]),
      array([23., 24., 25., 26., 27.])]
```

Remote Engines

- When the controller and engines are running on different hosts, things are slightly more complicated, but the underlying ideas are the same:

1. Setup the controller (ex: remote)

2. Setup the "slave" machines

3. Create as many slave machines as you wish as in step 2.

4. Request Client from the cluster "remote"

Setup the local controller

- Create an IPython profile in local machine (ex: remote)

  ```
  $ ipython profile create --parallel --profile=remote
  ```

- Now, each time we want to start a parallel computation, we begin by starting the controller:

  ```
  $ ipcontroller --profile=remote --ip=140.117.35.xxx
  ```

  where the address is the controller's IP address.

- This command creates a file `ipcontroller-engine.json` that contains the connection info that the remote machines need in order to connect to the controller.

- The file is located in `IPYTHONDIR/profile_remote/security`. We need to copy `ipcontroller-engine.json` to the slave (remote) machines after the slave machine has been setted up.

Setup the local controller–cont.

- We then add local engines by

  ```
  $ ipengine --profile=remote
  ```

- In fact, we could use the following command to start x engines and create the needed information to the file `ipcontroller-engine.json`.

  ```
  $ ipcluster start -n x --profile=remote --ip=140.117.35.xxx
  ```

Setup the remote "slave" machines

- Create an IPython profile (ex: remote) at slave machine.

```
$ ipython profile create --parallel --profile=remote
```

- Paste `ipcontroller-engine.json` under `IPYTHONDIR/profile_remote/security` at the slave machine

- Start a computation engine with the ipengine command

```
$ ipengine --profile=remote
```

- Or you can use the following way to provide the file directly.

```
ipengine --profile=remote --file=C:IPYTHONDIR/
 ↪profile_remote/security/ipcontroller-engine.json
```

- Create as many slave machines as you wish as in step 2, or use `ipcluster engines --n=x --profile=remote` to have more engines.

A true cluster machine

- Finally, request Client from the cluster "remote".

  We start local cluster with 3 engines. Then add one engine remotely.

```
[1]: import ipyparallel as ipp
     import numpy as np
     c = ipp.Client(profile='remote')
     print(c.ids)
```

```
[0, 1, 2, 3]
```

```
[2]: dview = c[:]
```

We have now 4 engines!
Make sure you paste `rank_mpi.py` to remote machine.

```
[3]: dview.run('rank_mpi.py')
```

```
[3]: <AsyncResult: execute>
```

A true cluster machine–cont.

```
[4]: %px totalhost = hostname()
```

```
[5]: dview['totalhost']
```

We have “two hostnames” now indicating that we are running from two machines.

```
[5]: ['DESKTOP-10711', 'DESKTOP-10711',␣
      ↪'DESKTOP-10711', 'LAPTOP-213RFLLM']
```

```
[6]: dview.scatter('a', range(10))
     dview['a']
```

We distribute 0-9 to 4 engines among two machines!

```
[6]: [range(0, 3), range(3, 6), range(6, 8), range(8,␣
      ↪10)]
```

MPI ipcluster revisit

- Recall that we need to configure `ipcluster_config.py` file by specifying `cluster_id` and `engine_launcher_class`.

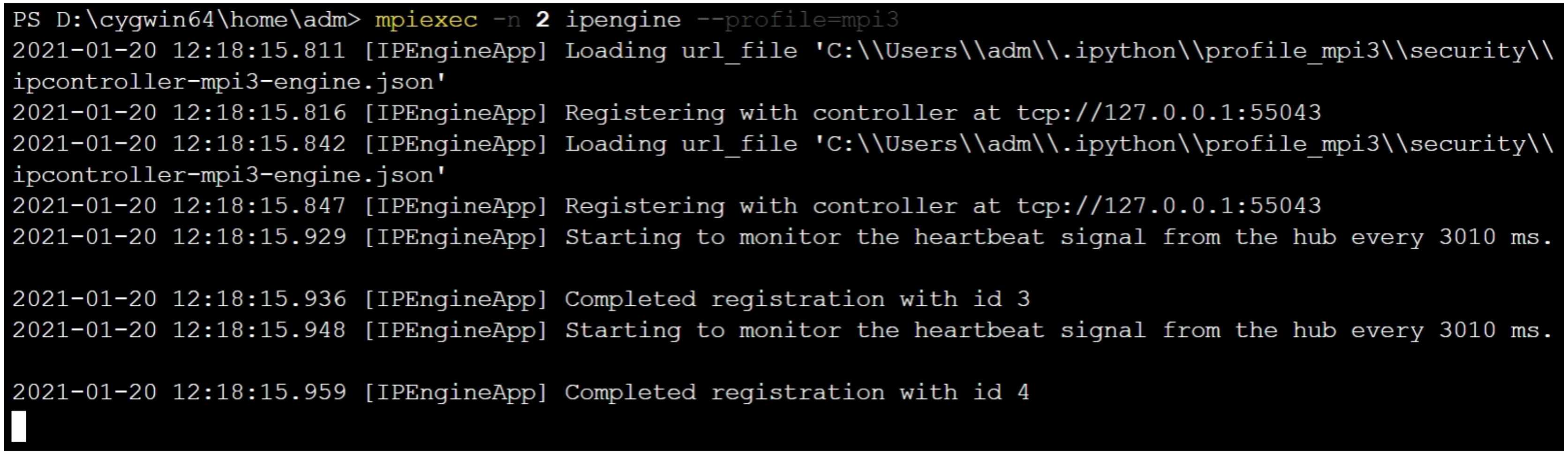
- After configuring IPython mpi profiles, we use `$ ipcluster start -n=x --profile=mpi3` command to launch a mpi cluster, we in fact did the following
  - Starts the IPython controller on current host.
  - Uses mpiexec to start x engines.

MPI ipcluster revisit–cont.

- Therefore, we could manually add more engines to the current cluster.

```
$ mpiexec -n 2 ipengine --profile=mpi3
```

- We starts two more engines. (id 3 and 4 in the following example)



```
PS D:\cygwin64\home\adm> mpiexec -n 2 ipengine --profile=mpi3
2021-01-20 12:18:15.811 [IPEngineApp] Loading url_file 'C:\\Users\\adm\\.ipython\\profile_mpi3\\security\\
ipcontroller-mpi3-engine.json'
2021-01-20 12:18:15.816 [IPEngineApp] Registering with controller at tcp://127.0.0.1:55043
2021-01-20 12:18:15.842 [IPEngineApp] Loading url_file 'C:\\Users\\adm\\.ipython\\profile_mpi3\\security\\
ipcontroller-mpi3-engine.json'
2021-01-20 12:18:15.847 [IPEngineApp] Registering with controller at tcp://127.0.0.1:55043
2021-01-20 12:18:15.929 [IPEngineApp] Starting to monitor the heartbeat signal from the hub every 3010 ms.

2021-01-20 12:18:15.936 [IPEngineApp] Completed registration with id 3
2021-01-20 12:18:15.948 [IPEngineApp] Starting to monitor the heartbeat signal from the hub every 3010 ms.

2021-01-20 12:18:15.959 [IPEngineApp] Completed registration with id 4
```

MPI ipcluster revisit–cont.

We start a cluster with 3 MPI engines.

```
$ ipcluster start -n 3 --profile=mpi3
```

```
[1]: import ipyparallel as ipp
     import numpy as np
     c = ipp.Client(profile='mpi3', cluster_id='mpi3')
     print(c.ids)
```

```
[0, 1, 2]
```

Now, start two more MPI engines!

```
$ mpiexec -n 2 ipengine --profile=mpi3
```

MPI ipcluster revisit–cont.

```
[2]: c = ipp.Client(profile='mpi3', cluster_id='mpi3')
     print(c.ids)
```

We have now 5 MPI engines!

```
[0, 1, 2, 3, 4]
```

```
[3]: dview = c[:]
     dview.scatter('a',np.arange(28,dtype='float'))
```

```
[3]: <AsyncResult: scatter>
```

```
[4]: dview['a']
```

```
[4]: [array([0., 1., 2., 3., 4., 5.]),
      array([ 6.,  7.,  8.,  9., 10., 11.]),
      array([12., 13., 14., 15., 16., 17.]),
      array([18., 19., 20., 21., 22.]),
      array([23., 24., 25., 26., 27.])]
```

MPI ipcluster revisit–cont.

```
[5]: dview.run('psum.py')
```

```
[5]: <AsyncResult: execute>
```

```
[6]: %px totalsum = psum(a)
```

Too bad, they are appaprently belong to two seperated MPI_COMM_WORLD!

```
[7]: dview['totalsum']
```

```
[7]: [array(153.), array(153.), array(153.), array(225.
   ↪), array(225.)]
```

MPI ipcluster revisit–cont.

```
[8]: dview.run('rank_mpi.py')
```

```
[8]: <AsyncResult: execute>
```

```
[9]: %px totalrank = myrank()
```

The ranks are duplicated. Evidently, we have two MPI_COMM_WORLDs!

```
[10]: dview['totalrank']
```

```
[10]: [0, 1, 2, 1, 0]
```

MPI ipcluster with remote nodes

- All MPI nodes have to be brought up at once.
  - `DynamicNodesinOpenMPI` suggests that it is not possible to add nodes post-launch.
  - Processes spawn by `MPI_Comm_spawn` are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents.
- Therefore, using `mpiexec` seems to be the only way to launch all engines at once.
- We have learned `ipython/ipyparallel` is capable to handle remote engines, how about MPI engines?

MPI ipcluster with remote nodes–cont.

- Bottom line is, you need to have a MPI capable cluster first.

- Unfortunatly, my local machine does NOT support this. MS provides a way to bulid cluster machine through MS HPC pack. However, the head node needs to run Windows Server OS, which I don't have.

- Alternatively, I will only demonstrate to start a local MPI cluster using ipcluster, then add in the remote MPI engines, but now they belong to different MPI_COMM_WORLD.

- The procedures are exactly the same as adding regular engines, but now with MPI engine configuration.

Setup the local MPI controller

- Create an IPython profile in local machine (ex: remote_mpi)

  ```
  $ ipython profile create --parallel --profile=remote_mpi
  ```

- Instruct ipcluster to use the MPI launchers by adding the lines to `ipcluster_config.py` file

  ```
    c.IPClusterEngines.engine_launcher_class =␣
   ↪'MPIEngineSetLauncher'
    c.BaseParallelApplication.cluster_id = 'mpi'
  ```

- Now, each time we want to start a parallel MPI computation, we begin by launching the MPI cluster:

  ```
  $ ipcluster start -n x --profile=remote_mpi --ip=140.117.35.xxx
  ```

  where the address is the local controller's IP address.

Start remote MPI engines

- This command creates a file `ipcontroller-mpi-engine.json` that contains the connection info that the remote machines need in order to connect to the controller.

- The file is located in `IPYTHONDIR/profile_remote_mpi/security`. We need to copy `ipcontroller-mpi-engine.json` to the same directory of the slave (remote) machines after the slave machine has been set up.

- Start remote MPI engines. The following will start two engines which belong to their own MPI_COMM_WORLD.

```
mpiexec -n 2 ipengine --profile=remote_mpi
 --file=C:\Users\adm\.ipython\profile_remote_mpi
        \security\ipcontroller-mpi-engine.json
```

- You could use `ipengine --profile=remote_mpi ...`, then you will have one engine and an independent MPI_COMM_WORLD.

Start remote MPI engines–cont.

- Document says that you could also specify the `xxx.json` in `ipengine_config.py`, e.g.

```
c.IPEngineApp.url_file = 'C:\Users\adm\.ipython\profile_remote_mpi
    \security\ipcontroller-mpi-engine.json'
```

- Note that `ipcontroller-mpi-engine.json` is from the local machine when mpi cluster is launched. There is extra "mpi" in it.

- BUT IT DOES NOT WORK FOR ME!

- If you don't wish to specify the location od the `xxx.json` file in the command line when use mpiexec, you will have to modify (at least in my case) `the xxx.json` file name to the default name `ipengine_config.py`, i.e. no "mpi".

A semi-ture MPI cluster

We start a MPI cluster locally with 3 engines.

[1]:

```
import ipyparallel as ipp
import numpy as np
c = ipp.Client(profile='remote_mpi',␣
 ↪cluster_id='mpi')
print(c.ids)
```

```
[0, 1, 2]
```

Launch another 2 MPI engines remotely.

[2]:

```
c = ipp.Client(profile='remote_mpi',␣
 ↪cluster_id='mpi')
print(c.ids)
```

```
[0, 1, 2, 3, 4]
```

We have now 5 engines!

A semi-ture MPI cluster–cont.

```
[3]: dview = c[:]
     dview.scatter('a',np.arange(32,dtype='float64'))

[3]: <AsyncResult: scatter>

[4]: dview['a']

[4]: [array([0., 1., 2., 3., 4., 5., 6.]),
      array([ 7.,  8.,  9., 10., 11., 12., 13.]),
      array([14., 15., 16., 17., 18., 19.]),
      array([20., 21., 22., 23., 24., 25.]),
      array([26., 27., 28., 29., 30., 31.])]

[5]: dview.run('psum.py')

[5]: <AsyncResult: execute>

[6]: %px totalsum = psum(a)

[7]: dview['totalsum']

[7]: [array(190.), array(190.), array(190.), array(306.
      ↪), array(306.)]
```

- There are two sums, one local one remote.

A semi-ture MPI cluster–cont.

- Homework: Build a mpi cluster from two seperated machines and use it compute a variance of a set of numbers. Note that we will have seperated MPI.COMM_WORLD, you will have to communicate the information through iPython session.