# Introduction to IPython Parallel Computing

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

# Topic Overview

- Introduction to ipyparallel

- Architecture overview

- Getting Started

- IPython's Direct interface

- Parallel Magic Commands

# Introduction to ipyparallel

- This section gives an overview of IPython's architecture for parallel and distributed computing. This architecture abstracts out parallelism in a general way, enabling IPython to support many different styles of parallelism, including:

1. Single program, multiple data (SPMD) parallelism

2. Multiple program, multiple data (MPMD) parallelism

3. Message passing using MPI

4. Data parallel

5. Combinations of these approaches
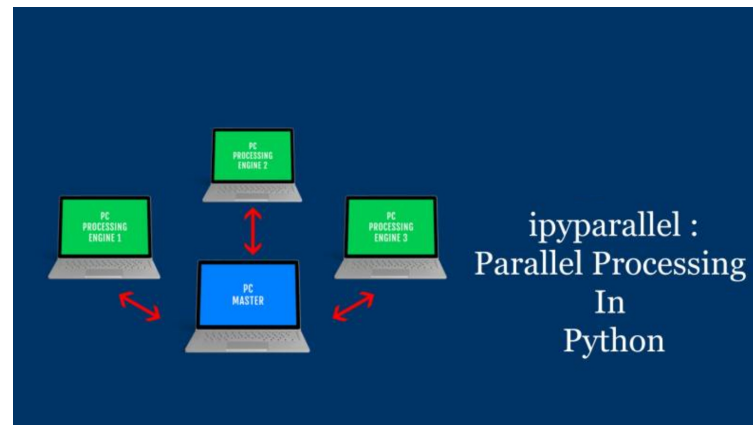
# ipyparallel project

- The IPython.parallel package has moved to the ipyparallel project since ipyparallel 4.0. IPython stands for interactive Python.

- The ipyparallel architecture is a natural extension of the serial IPython architecture.
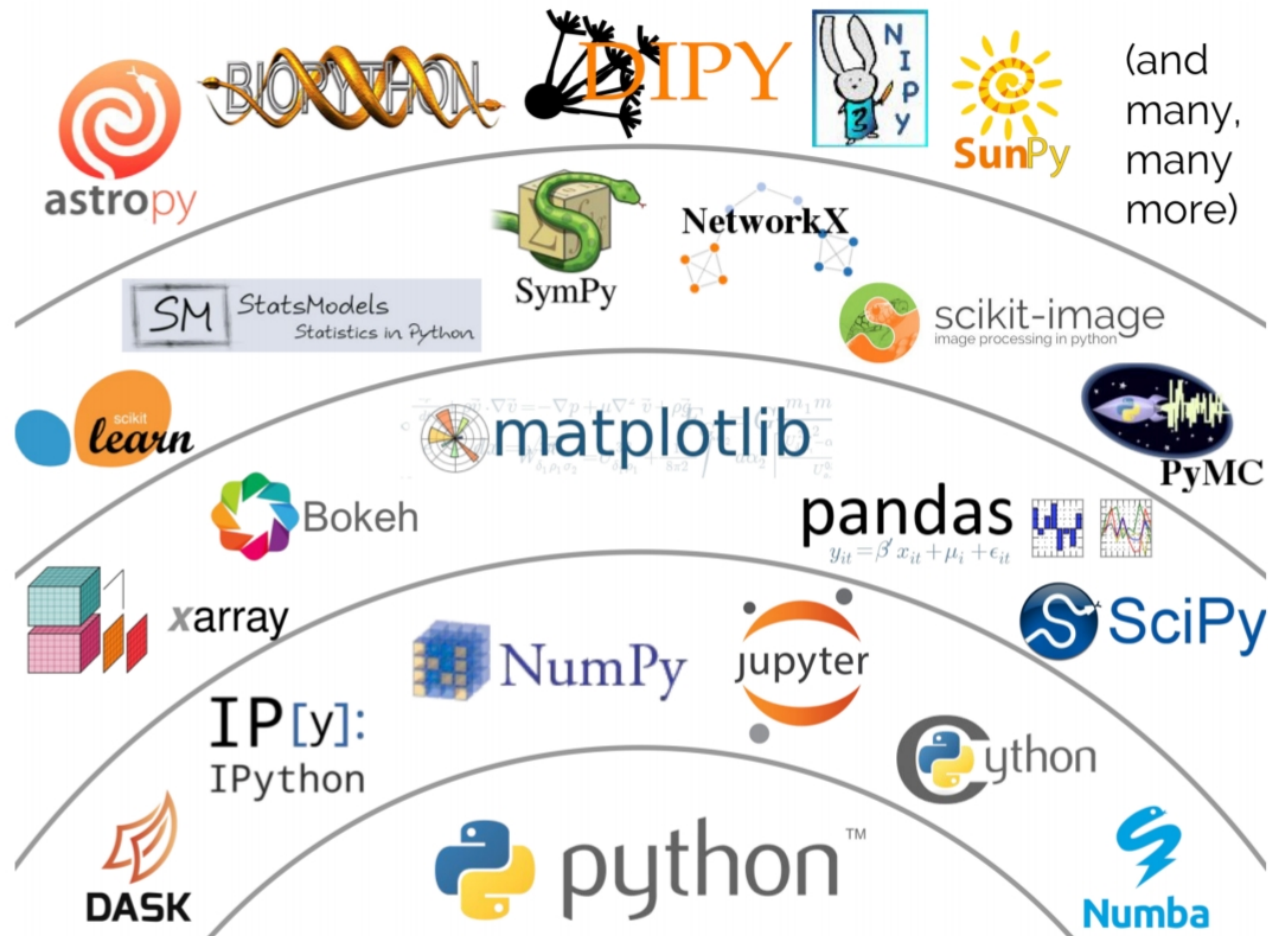
# ipyparallel Features

- Quickly parallelize algorithms that are embarrassingly parallel using a number of simple approaches.

- Steer traditional MPI applications on a supercomputer from an IPython session on your laptop.

- Analyze and visualize large datasets (that could be remote and/or distributed) interactively using IPython and tools like matplotlib.

- Tie together multiple MPI jobs running on different systems into one giant distributed and parallel system.

- Start a parallel job on your cluster and pull back data into their local IPython session for plotting and analysis.

- Run a set of tasks on a set of CPUs using dynamic load balancing.

# Architecture overview

- The IPython architecture consists of four components:

  1. The IPython client

  2. The IPython hub

  3. The IPython schedulers

  4. The IPython engine
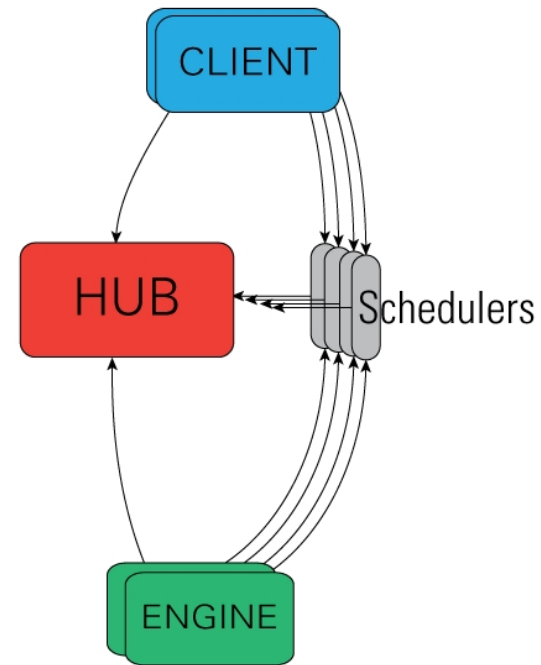


- These components live in the ipyparallel package.

# IPython client and views

- There is one primary object, the `Client`, for connecting from local IPython session to a cluster.

- For each execution model, there is a corresponding `View`.

- These views allow users to interact with a set of engines through the interface. Here are the two default views:

- The `DirectView` class for explicit addressing.

- The `LoadBalancedView` class for destination-agnostic scheduling.

---

In computing, destination-agnostic means that system does not know or does not care where the destination of the execution.

# IPython engine

- The IPython engine is an extension of the IPython kernel for Jupyter.

- The engine listens for requests over the network, runs code, and returns results.

- When multiple engines are started, parallel and distributed computing becomes possible.

# IPython controller

- The IPython controller processes provide an interface for working with a set of engines.

- At a general level, the controller is a collection of processes to which IPython engines and clients can connect.

- The controller is composed of a Hub and a collection of Schedulers.

- These Schedulers are typically run in separate processes on the same machine as the Hub.

- The controller also provides a single point of contact for users who wish to access the engines connected to the controller.

- The standard model is to designate that the controller listen on localhost, and use ssh-tunnels to connect clients and/or engines.

# IPython controller –cont.

- To connect to the controller an engine or client needs some information that the controller has stored in a JSON file.

- Typically, this is the /.ipython/profile_default/security directory on the host where the client/engine is running. Once the JSON files are copied over, everything should work fine.

- Currently, there are two JSON files that the controller creates:

  – ipcontroller-engine.json This JSON file has the information necessary for an engine to connect to a controller.

  – ipcontroller-client.json The client's connection information.

- There are different ways of working with a controller. In IPython, all of these models are implemented via the View.apply() method. The two primary models for interacting with engines are:

  1. A `Direct` interface, where engines are addressed explicitly

  2. A `LoadBalanced` interface, where the Scheduler is entrusted with assigning work to appropriate engines

# The Hub & Schedulers

## The Hub

- The center of an IPython cluster is the Hub.

- This is the process that keeps track of engine connections, schedulers, clients, as well as all task requests and results.

- The primary role of the Hub is to facilitate queries of the cluster state, and minimize the necessary information required to establish the many connections involved in connecting new clients and engines.
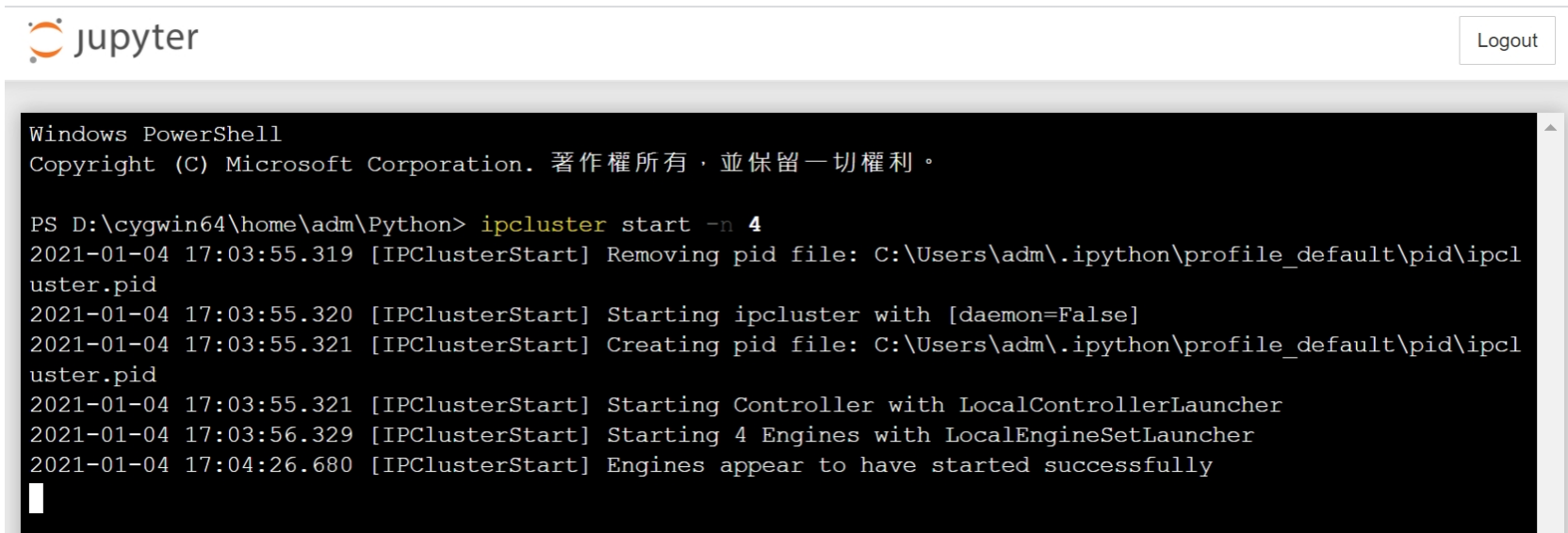
## Schedulers

- All actions that can be performed on the engine go through a Scheduler.

- While the engines themselves block when user code is run, the schedulers hide that from the user to provide a fully asynchronous interface to a set of engines.

# Getting Started

- To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine.

- Initially, it is best to start a controller and engines on a single host using the ipcluster command.

- To start a controller and 4 engines on your local machine:

```
PS D:\cygwin64\home\adm\Python> ipcluster start -n 4
```

# Creating a Client object

```
[1]: import ipyparallel as ipp
```

We create a `Client` instance named `c`.

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

To make sure there are engines connected to the controller, users can get a list of engine ids:

```
[3]: c.ids
```

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

Here we see that there are four engines ready to do work for us.

- The initially active View will have attributes `targets='all'`, `block=True`, which is a blocking view of all engines, evaluated at request time.

# Creating a Client object –cont.

We, then, make calls happen remotely by turning function calls into calls to apply.

For instance: we define a `lambda` function which returns "Hello World!".

```
[4]: c[:].apply_sync(lambda :  "Hello World!")
```

```
[4]: ['Hello World!', 'Hello World!', 'Hello World!',
      'Hello World!']
```

- To end the controller and engines, use

```
ipcluster stop
```

```
請嘗試新的跨平台 PowerShell https://aka.ms/pscore6

PS D:\cygwin64\home\adm\Python> ipcluster stop
2021-01-04 18:05:50.404 [IPClusterStop] Removing pid file: C:\Users\adm\.ipython\profi
le_default\pid\ipcluster.pid
PS D:\cygwin64\home\adm\Python> []
```

Reference:
https://ipyparallel.readthedocs.io/en/latest/api/ipyparallel.html

# Apply and Map Functions

- `apply_sync` is a method of `Client`. It returns the result of a function called with supplied arguments.

- `_syn` refers to waiting for the results.

- `map` is a bit different from `apply`.

- The `map` iterator takes a function and applies it to the values in an iterator. Therefore it is necessary to have an iterator as argumgment. We will see this soon in

```
[8]: dview.map_sync(lambda x: x ** 10, range(7))
```

# Anonymous (lambda) Function

- In python, the most common way of defining functions is the `def` statement.

- `lambda` statement is another way of defining short, one-off (happening only once) functions.

```
[5]: add = lambda x, y: x + y
     add(1, 2)
```

[5]: 3

```
[6]: def add(x, y):
         return x + y
```

- When a client is created with no arguments, the client tries to find the corresponding JSON file in the local ~/.ipython/profile/default/security directory.

- You could specify a profile, we shall come back to this later.

# IPython's Direct interface

- The direct interface represents one possible way of working with a set of IPython engines.

- The basic idea behind the direct interface is that the capabilities of each engine are directly and explicitly exposed to the user.

- Each engine is given an id that is used to identify the engine and give it work to do.

- This interface is very intuitive and is designed with interactive usage in mind, and is the best place for new users of IPython to begin.

# Creating a DirectView

We construct a DirectView object via list-access to the client and make use of it later.

```
[7]: dview = c[:]
```

- In many cases, you want to call a Python function on a sequence of objects, but in parallel. IPython Parallel provides a simple way of accomplishing this: using the DirectView's `map()` method.

  Python's builtin `map()` functions allows a function to be applied to a sequence element-by-element. We compute $\{x^{10} : x = 0,\ldots,6\}$.

```
[8]: dview.map_sync(lambda x: x ** 10, range(7))
```

```
[8]: [0, 1, 1024, 59049, 1048576, 9765625, 60466176]
```

- Question: How do we know how the sequence of elements (jobs) were distributed among the engines?

# Process ID (pid)

- The OS module in python provides functions for interacting with the operating system. This module provides a portable way of using operating system dependent functionality.

  We use `os.getpid()` to obatin the pid of each engine.

```
[9]: %%px
import os
pid = os.getpid()   # every engines will be
 ↪different
print(pid)
```

```
[stdout:0] 19464
[stdout:1] 14952
[stdout:2] 1992
[stdout:3] 12472
```

- `%%px` is a Cell Magic, which accepts some arguments for controlling the execution. We shall come back to this later.

Now, we ask the lambda function to return the pid and we can see how the jobs are being distributed among engines.

```
[10]: dview.map_sync(lambda x: pid, range(7))
```

```
[10]: [19464, 19464, 14952, 14952, 1992, 1992, 12472]
```

We see that 7 jobs are distributed in $\{2, 2, 2, 1\}$. You should try different numbers of jobs.

# Calling Python functions

- The most basic type of operation that can be performed on the engines is to execute Python code or call Python functions.

- Executing Python code can be done in blocking or non-blocking mode (non-blocking is default) using the View.execute() method, and calling functions can be done via the View.apply() method.

- `apply` is the main method for doing remote execution (in fact, all methods that communicate with the engines are built on top of it).

```
view.apply(f, *args, **kwargs)
```

## Flags of DirectView

- dv.block : bool, default: False
  whether to wait for the result, or return an AsyncResult object immediately

- dv.targets : int, list of ints
  The engines associated with this View.

# Calling Python functions –cont.

We distribute `a=6, b=7` to all engines.

```
[11]: dview['a'] = 6
      dview['b'] = 7
```

We ask all engines to call a lambda function with input `x= 27` and add it to a and b.

However, the default value is `dv.block = False,` so no results will be shown.

```
[12]: dview.apply(lambda x: a + b + x, 27)
```

`[12]: <AsyncResult: <lambda>>`

We set `dv.block = True` and the interavtive session waits for the results.

```
[13]: dview.block = True
      dview.apply(lambda x: a + b + x, 27)
```

`[13]: [40, 40, 40, 40]`

- `dview.apply_sync` = $\begin{cases} \text{dview.block = True} \\ \text{dview.apply} \end{cases}$

# Calling Python functions –cont.

`dv.targets` returns the list of ints which shows the engines associated with this View.

```
[14]: dview.targets
```

```
[14]: [0, 1, 2, 3]
```

We create a new DirectView by index-accessing on a `Client`.

```
[15]: dview2 = c[1:3]
```

```
[16]: dview2.targets
```

```
[16]: [1, 2]
```

# Calling Python functions –cont.

- Python commands can be executed as strings on specific engines by using a View's execute method:

  We execute $a + b$ on all "even" engines.

```
[17]: c[::2].execute('c=a+b')
```

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

  We execute $a - b$ on all "even" engines.

```
[18]: c[1::2].execute('c=a-b')
```

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

```
[19]: dview['c'] # shorthand for dview.pull('c',␣
      ↪block=True)
```

```
[19]: [13, -1, 13, -1]
```

# Calling Python functions –cont.

In non-blocking mode, the AsyncResult object gives you a way of getting a result at a later time through its `get()` method.

```
[20]: dview.block = False
       ar = dview.apply(lambda x: a+b+x, 27)
```

```
[21]: # Poll to see if the result is ready
       ar.ready()
```

[21]: False

```
[22]: # ask for the result
       ar.get()
```

[22]: [40, 40, 40, 40]

# Calling Python functions –cont.

```python
[23]: # define our function
      def wait(t):
          import time
          tic = time.time()
          time.sleep(t)
          return time.time() - tic
```

```python
[24]: ar = dview.apply_async(wait, 10)
```

```python
[25]: ar.ready()
```

```
[25]: False
```

- This allows you to quickly submit long running commands without blocking your local IPython session

# Moving Python objects around

- You can transfer Python objects between your IPython session and the engines.

- In IPython, these operations are called

  - `push(ns, targets=None, block=None, track=None)`

    * update remote namespace with dictionary ns

    * ns - dict of keys with which to update engine namespace(s)

  - `pull(names, targets=None, block=None)`

    * get object(s) by name (key) from remote namespace

    * will return one object if it is a key

    * can also take a list of keys, in which case it will return a list of objects.

# Moving Python objects around – cont.

```
[5]: # In IPython, these operations are called push()
      ↪(sending an object to the engines)
     # and pull() (getting an object from the engines).
     dview.push(dict(a=1.03234, b=3453))
```

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

```
[6]: dview.pull('a', block=True)
```

```
[6]: [1.03234, 1.03234, 1.03234, 1.03234]
```

```
[7]: dview.pull(['a', 'b'], block=True)
```

```
[7]: [[1.03234, 3453], [1.03234, 3453], [1.03234,
      ↪3453], [1.03234, 3453]]
```

The `targets` argument allows us to transfer object to specific engines.

```
[8]: dview.pull('b', targets=0, block=True)
```

```
[8]: 3453
```

# Moving Python objects around – cont.

We could be clever on distributing objects.

```
[9]: for i in range(4):
         dview.push(dict(a=i, b=i ** 2 + 1), targets=i)
```

```
[10]: dview.pull('a', block=True)
```

[10]: [0, 1, 2, 3]

```
[11]: dview.pull('b', block=True)
```

[11]: [1, 2, 5, 10]

# A bit on Dictionary

- Dictionaries are used to store data values in key:value pairs.

- A dictionary is a collection which is unordered, changeable and does not allow duplicates.

- Dictionaries are written with curly brackets, and have keys and values:

```
[1]: thisdict = {'brand':"Ford",  "model":"Mustang",
    ↪"year":1964}
     print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year':
 ↪1964}
```

```
[2]: dict(name = "John", age = 36, country = "Norway")
```

```
[2]: {'name': 'John', 'age': 36, 'country': 'Norway'}
```

# Moving Python objects around – cont.

- It is useful to partition a sequence and push the partitions to different engines.

- In MPI language, this is know as scatter/gather and python follow that terminology.

- However, it is important to remember that in IPython's Client class,

  - `scatter()` is from the interactive IPython session to the engines

  - `gather()` is from the engines back to the interactive IPython session

- For scatter/gather operations between engines, MPI, pyzmq, or some other direct interconnect should be used.

# Moving Python objects around – cont.

We distribute $\{0, \cdots, 27\}$ to 4 engines.

```
[12]: # scatter() is from the interactive IPython
      ↪session to the engines
      dview.scatter('c', range(28))
```

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

```
[13]: dview['c']
```

```
[13]: [range(0, 7), range(7, 14), range(14, 21),
      ↪range(21, 28)]
```

- You should try a number which can NOT be evenly distributed and see what happened.

```
[14]: # gather() is from the engines back to the
      ↪interactive IPython session.
      ar = dview.gather('c')
```

```
[15]: ar[1][2]
```

```
[15]: 9
```

# Remote function decorators

- Python has an interesting feature called decorators. It takes in a function, adds some functionality and returns it.

- Remote functions are like normal functions, but when they are called they execute on one or more engines rather than locally.

- IPython provides two decorators for producing parallel functions. The first is @remote, which calls the function on every engine of a view.

  We define a remote function which returns the engine's pid

```
[16]: # @remote, which calls the function on every
       ↪engine of a view.
      @dview.remote(block=True)
      def getpid():
          import os
          return os.getpid()
```

```
[17]: getpid()
```

```
[17]: [19464, 14952, 1992, 12472]
```

# Hello world! Again

We pass the npes and myrank to each engine.

```python
# Hello world! Again
n = len(c)
for i in range(n):
    dview.push(dict(npes=n, myrank=i), targets=i)

@dview.remote(block=True)
def hello():
    return "Hello world from %i out of %i!" \
    % (myrank, npes)

hello()
```

```
[18]: ['Hello world from 0 out of 4!',
       'Hello world from 1 out of 4!',
       'Hello world from 2 out of 4!',
       'Hello world from 3 out of 4!']
```

We could pass a longer dict to engines

```
[19]: # A bit on dictionary
number = {'three': 3, 'ninety': 90, 'two': 2,
 ↪'one': 1}
print(number)
dview.push(number)


print(type(number))



@dview.remote(block=True)
def dict_num():
    return one, ninety

dict_num()
```

```
{'three': 3, 'ninety': 90, 'two': 2, 'one': 1}
<class 'dict'>
```

```
[19]: [(1, 90), (1, 90), (1, 90), (1, 90)]
```

# Let's compute the variance

```
[20]: # We are adding 'c' which we need to scatter
      # among all engines before we call this function.
      @dview.remote(block=True)
      def local_sum():
          return sum(c)


      # mu is the mean
      @dview.remote(block=True)
      def local_var_sum_arg(mu):
          l_sum = 0
          for i in c:
              l_sum += (i - mu)**2
          return l_sum
```

# Let's compute the variance –cont.

```
[21]: l_sum = local_sum()
      print('Partial sum', l_sum)
      g_sum = sum(l_sum)

      print('Gobal sum', g_sum)


      gmu = g_sum/28
      print('Mean', gmu)


      l_sum = local_var_sum_arg(gmu)
      print('Partial variance sum', l_sum)
      g_sum  = sum(l_sum)
      g_var = math.sqrt(g_sum/28)
      print('Variance = ', g_var)
```

```
Partial sum [21, 70, 119, 168]
Gobal sum 378
Mean 13.5
Partial variance sum [799.75, 113.75, 113.75, 799.
 ↪75]
Variance =  8.077747210701755
```

# How about synchronization?

- Client is a synchronous object, if you set `block=False` in a remote function, system will synchronize all engine if you ask for the output from the remote function.

- If you need to wait for particular results to finish, you can use the `wait()` method.

```
[22]: # How about synchronization?
      @dview.remote(block=False)
      def eng_wait():
          import time
          time.sleep(myrank*10)
          a = myrank
          return a
```

# How about synchronization? – cont.

```
[23]: dview.pull('a', block=True)
```

```
[23]: [0, 1, 2, 3]
```

```
[24]: ar = eng_wait()
```

```
[*]: ar.get()
```

Later .....

```
[24]: [0, 1, 2, 3]
```

or

```
[24]: ar.wait()
```

```
[24]: True
```

# Remote function decorators –cont.

- The @parallel decorator creates parallel functions, that break up an element-wise operations and distribute them, reconstructing the result.

- We import NumPy which is a library for the Python, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

  We create an array of random numbers with size of $640 \times 48$.

```python
[25]: import numpy as np
      import time
      A = np.random.random((640,48))
      # 640x48
      A.size
```

[25]: 30720

# Remote function decorators –cont.

We define a parallel function. Then, we wish to see some speedup!.
Eureka !

```
[26]: @dview.parallel(block=True)
      def pmul(A,B):
          return A*B
```

```
[27]: t1 = time.time()
      C_local = A*A
      t2 = time.time()
      print('serial run time: ' + str(t2-t1) + '
       ↪seconds')
```

```
serial run time: 0.000997304916381836 seconds
```

```
[28]: t1 = time.time()
      C_remote = pmul(A,A)
      t2 = time.time()
      print('parallel run time: ' + str(t2-t1) + '
       ↪seconds')
```

```
parallel run time: 0.012965679168701172 seconds
```

# Here is the speedup

We prolong (waste) the compuational time !

```python
[29]: dview.block=False
      dview.scatter('x',range(32000))

      t1 = time.time()
      %px y = sum(x)
      %px for i in range(10 ** 8): i+=1

      y = dview.gather('y')
      global_sum = sum(y)

      t2 = time.time()
      print(global_sum)

      print('parallel run time: ' + str(t2-t1) + '
      ↪seconds')
```

```
511984000
parallel run time: 12.10887145996093B seconds
```

There should be 4 times of loops in a serial code!

```
[30]: xx = list(range(32000))
      # print(x)
      t1 = time.time()
      for i in range(4* 10 ** 8): i+=1
      global_sum =  sum(xx)
      # print(y)
      t2 = time.time()
      print(global_sum)
      print('serial run time: ' + str(t2-t1) + '␣
       ↪seconds')
```

```
511984000
serial run time: 36.25384283065796 seconds
```

- Now we are seeing a speedup around 3!

# Magic commands

- Magic commands are one of the important enhancements that IPython offers compared to the standard Python shell.

- Magic commands act as convenient functions where Python syntax is not the most natural one.

- There are two types of magic commands

  - Line magics

    * They are similar to command line calls. They start with % character.

    * Rest of the line is its argument passed without parentheses or quotes.

  - Cell magics

    * They have %% character prefix.

    * Unlike line magic functions, they can operate on multiple lines below their call. They receive the whole block as a single string.

# Parallel Magic commands

- IPython magic commands make it a bit more pleasant to execute Python commands on the engines interactively.

- These are mainly shortcuts to `DirectView.execute()` and `AsyncResult.display_outputs()` methods respectively.

- These magics will automatically become available when you create a Client:

- `%px y = sum(x)` simply means `y = sum(x)` in all choosen engines.

```
[31]: %px print('Hello world!')
```

```
[stdout:0] Hello world!
[stdout:1] Hello world!
[stdout:2] Hello world!
[stdout:3] Hello world!
```

We request a $2 \times 2$ random matrix in each engine.

```
[32]: %px a = numpy.random.rand(2,2)
```

# Remote imports

- Sometimes you may want to import packages both in your interactive session and on your remote engines.

- This can be done with the context manager created by a DirectView's `sync_imports()` method.

```
[33]: with c[:].sync_imports():
          import numpy
```

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

# Parallel Magic commands – cont.

```
[34]: %px print(a)
      %px numpy.linalg.eigvals(a)
```

```
[stdout:0]
[[0.94200896 0.38667172]
 [0.03502654 0.06301613]]
[stdout:1]
[[0.38151927 0.54669028]
 [0.59758071 0.15833394]]
[stdout:2]
[[0.85407804 0.69845007]
 [0.16777561 0.01239195]]
[stdout:3]
[[0.8242177  0.66605824]
 [0.24867175 0.38486579]]
```

```
Out[0:25]: array([0.95715622, 0.04786887])
Out[1:25]: array([ 0.85228774, -0.31243452])
Out[2:25]: array([ 0.97572163, -0.10925165])
Out[3:25]: array([1.06702136, 0.14206213])
```

- I do not know the meaning of 25 in Out(3:25):. Perhaps the numbers of operations?

- Since engines are IPython as well, you can even run magics remotely:

```
[35]: # The %pylab inline specifies that figures should
      # be   shown inline, directly in the notebook.
      %px %pylab inline
```

```
[stdout:0] Populating the interactive namespace␣
 ↪from numpy and matplotlib
[stdout:1] Populating the interactive namespace␣
 ↪from numpy and matplotlib
[stdout:2] Populating the interactive namespace␣
 ↪from numpy and matplotlib
[stdout:3] Populating the interactive namespace␣
 ↪from numpy and matplotlib
```
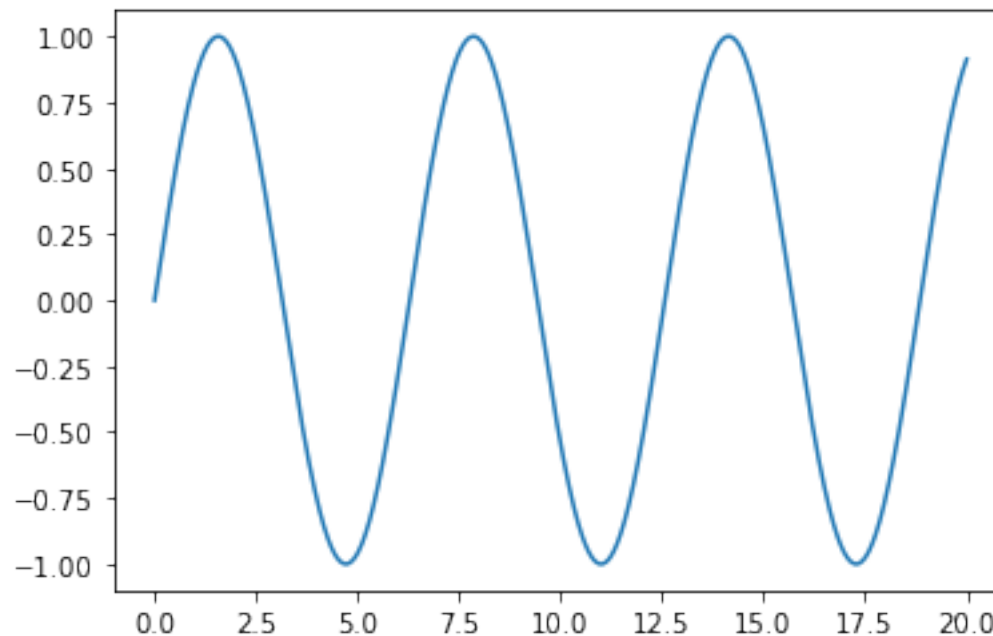
# Parallel Magic commands – cont.

```
[36]: %px x = np.linspace(0, 20, 1000)
      # 1000 evenly-spaced values from 0 to 20
      %px y = np.sin(x)
```

```
[37]: %%px --target 0
      pylab.plot(x, y)
```



```
Out[0:35]: [<matplotlib.lines.Line2D at
           →0x12a069dae10>]
```

# Parallel Magic commands – cont.

- %%px accepts `--targets` for controlling which engines on which to run

```
[38]: %%px --targets ::2
print("I am even")
```

```
[stdout:0]  I am even
[stdout:2]  I am even
```

- `--[no]block` for specifying the blocking behavior of this cell, independent of the defaults for the View.

- If you are using %px in non-blocking mode, you won't get output.

- You can use %pxresult to display the outputs of the latest command, as is done when %px is blocking:

# Parallel Magic commands – cont.

```
[39]: %%px --noblock
import time
time.sleep(1)
time.time()
```

[39]: <AsyncResult: execute>

```
[40]: %pxresult
```

Out[0:39]: 1610344552.924335

Out[1:34]: 1610344552.924335

Out[2:37]: 1610344552.924335

Out[3:34]: 1610344552.9253082

# Load Balanced View

- The `LoadBalancedView` interface to the cluster presents the engines as a fault tolerant, dynamic load-balanced system of workers.

- Unlike the direct interface, in the task interface the user have no direct access to individual engines.

- By allowing the IPython scheduler to assign work, this interface is simultaneously simpler and more powerful.

- When the user can break up the user's work into segments that do not depend on previous execution, the task interface is ideal.

- For load-balanced execution, we will make use of a LoadBalancedView object, which can be constructed via the client's `load_balanced_view()` method:

```
[1]: import ipyparallel as ipp; import time
     c = ipp.Client()
```

```
[2]: # default load-balanced view
     lview = c.load_balanced_view()
     dview = c[:]
```

```
[3]: # I have no idea why load balance takes longer␣
      ↪time !
     lview.block = True
     t1 = time.time()
     parallel_result = lview.map(lambda x:x,␣
      ↪range(3200))
     pp = sum(parallel_result)
     t2 = time.time()
     print('load balance run time: ' + str(t2-t1) + '␣
      ↪seconds')
     print(pp)
```

```
load balance run time: 5.851065158843994 seconds
5118400
```

```
[4]: dview.block = True
     t1 = time.time()
     parallel_result = dview.map(lambda x:x,
      ↪range(3200))
     pp = sum(parallel_result)
     t2 = time.time()
     print('direct view run time: ' + str(t2-t1) + '
      ↪seconds')
     print(pp)
```

```
direct view run time: 0.0159609317779541 seconds
5118400
```

- Similar to direct view, we could define parallel function through a decorator that turns any Python function into a parallel function:

```
[5]: @lview.parallel()
     def Lf(x):
         return 10.0*x*4

     @dview.parallel()
     def Df(x):
         return 10.0*x*4
```

```
[6]: t1 = time.time()
     gg = sum(Lf.map(range(32)))
     t2 = time.time()
     print('load balance run time: ' + str(t2-t1) + '
      ↪seconds')
     print(gg)

     t1 = time.time()
     gg = sum(Df.map(range(32)))
     t2 = time.time()
     print('direct view run time: ' + str(t2-t1) + '
      ↪seconds')
     print(gg)
```

```
load balance run time: 0.05824756622314453 seconds
19840.0
direct view run time: 0.010972023010253906 seconds
19840.0
```

- We still did not see the advantages of Load Balanced View.