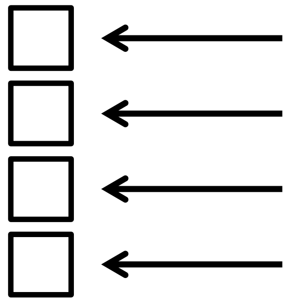


Parallel Computing in Python

PRACE Winter School 2017 @ Tel-Aviv/Israel

Mordechai Butrashvily

All material is under

All Rights Reserved

09/02/2017

Few Words

- I’m a graduate student @ Geophysics, Tel-Aviv University
- Dealing with physics, geophysics & applied mathematics

- Parallel computing is essential in my work

- Participated in PRACE Summer of HPC 2013
- **Don’t miss a chance to take part**
- Registration ends by 19/2
- **Apply here**:
  - https://summerofhpc.prace-ri.eu/apply

- You would enjoy it, and if not I’m here to complain…

All Rights Reserved

Why Are We Here?

- Python is very popular, simple and intuitive
- Increasing use in many fields of research & industry
- Open-source and free
- Many toolboxes available in almost every field
  - NumPy/SciPy are prerequisite for scientists
  - Replacement for MATLAB/Mathematica (to some extent)

- Natural to ask if parallel computing can be simpler too
  - Indeed yes!
  - And also take advantage of existing C/Fortran code

All Rights Reserved

(My) Audience Assumptions

- Some may know python
  - But not much of its multiprocessing capabilities
- Some may know parallel computing
  - But not quite familiar with python
- Working in Linux environments
  - Not mandatory but highly advised
- Knowing a descent programming language

All Rights Reserved

(My) Workshop Goals

- Don’t worry

- The workshop was designed for introductory purposes
- First basic theory, then practical hands-on session

- Hands-on will be in different levels (3 as usual)
  - **Basic** – for those who make their first steps
  - **Intermediate** – for those with moderate programming familiarity
  - **Advanced** – for those who are proficient in python

All Rights Reserved

Agenda

- Introduction To Python
  - The Interpreter Concept
  - Scientific Computing
- Python <-> C Interoperability Overview
- Node-Level Parallel Processing
  - "**multiprocessing**" Module
- Cluster-Level Parallel Processing
  - IPython "**ipyparallel**" Package
  - MPI Support Using "**mpi4py**"

All Rights Reserved 09/02/2017

- Basic Python & NumPy/SciPy
- Motivation

INTRODUCTION

All Rights Reserved

Python in High-Level

- Is a script based, interpreted language
- Very common in technology (websites etc.)
- Highly popular in recent years for many purposes

- Shares very intuitive syntax
- Still with advanced features (~OOP etc.)

- Two versions:
  - 2.7 – Providing legacy support and actively maintained
  - 3 – New features, new python, better target for it

- All our examples will work on both versions

All Rights Reserved

Meet the Interpreter

- The interpreter is responsible for running our code
    - Either if it’s interactive
    - Or running a script from a shell
- Issue the interpreter (Python 2) by running:

\# python <*optional_script_file*>

- Or for Python 3:

\# python3 <*optional_script_file*>

All Rights Reserved 09/02/2017

Meet the Interpreter (Cont.)

- A much nicer interpreter is “**idle**” or even “**idlespork**”
  - Remembers past statements
  - Auto statement completion
  - Code editing and running

- Some good IDE’s:
  - PyCharm (the best so far)
  - Microsoft Visual Studio Code (supports Linux & Mac as well!)

All Rights Reserved

Python Syntax & Reference

- The web is full of perfect guides to python
- Including the main website: http://www.python.org

- You may consult for any issue

All Rights Reserved

Scientific Computing

- An essential suite of packages for researchers:
  - **NumPy** (http://www.numpy.org)
  - **SciPy** (http://www.scipy.org)
  - **matplotlib** (http://www.matplotlib.org)
  - **SymPy** (Symbolic expressions, equations, integrals etc.)

- Can be used immediately after downloading and installing

All Rights Reserved

Scientific Computing (Cont.)

- For those who consider migrating from MATLAB
- Most functions are available with similar names and semantics
- Support for Linear Algebra, Statistics, Optimization…
  - Just name it

- Again, the web is full of examples and great documentation
- Including full source code

All Rights Reserved

Motivation (restated)

- Considering all that…
- We have state-of-the-art simulations
- Usually written to run on a single CPU core (serial)
- But this is not enough

- We would like to:
  1. Take advantage of all CPU cores
  2. Perhaps even utilize other computers as well
  3. Pass **NumPy** and python objects seamlessly
  4. Maintain simple & scalable code
  5. And much more (don't get greedy though)

All Rights Reserved

“<u>**No Free Lunch**</u>” Theorem

- Parallel programming is not easy at all
- In fact it is the most complex task in programming
  - We will deal with task parallelization mostly
  - **Not** the usual “***OpenMP***” approach (***parallel for***)

- Python is a very high-level language
- Performance is sacrificed for faster coding
  - Don’t expect same performance as true C/C++/Fortran code
  - But writing and debugging it will be about 10x faster if not more

  - We’ll see in a minute that python can interface native code
  - So performance isn’t much sacrificed

All Rights Reserved

PYTHON <-> C

- Python <-> C/Fortran Interoperability
- “ctypes” module

All Rights Reserved

Motivation

- A significant amount of code is still written in C/Fortran
- Scientific code runs much faster in native code
  - Keep performance-critical code in C/Fortran
  - Manage the rest of simulation including I/O in python
- Pass NumPy/SciPy data structures to native code
  - But avoid data transfer overhead
- Keep elegant code
- All this can be accomplished using "**ctypes**" module

All Rights Reserved

Fortran Disclaimer

- C and Fortran are very much alike
- We'll focus on C interoperability
- Although it can all be accomplished with Fortran as well

All Rights Reserved

"ctypes" Module

- A standard built-in python module

- Provides python data types that match C:
  - Primitive types: c_int, c_int16, c_float, c_size_t …
  - Define structures, unions, arrays, pointers

- Functions:
  - Call C functions (system, e.g. printf, and custom)
  - Access to Windows API (on Windows)
  - Pass python callbacks to C

- And much more

All Rights Reserved

Immediate Benefits

- The calling code is pure python
- No CPython nightmare

- NumPy arrays are C allocated pointers
- Pass NumPy array to C transparently
- Minimal performance overhead

- A note to Fortran programmers:
  - Possible to allocate column-major NumPy arrays
  - Native processing in Fortran
  - No transposing etc.

All Rights Reserved

Example

- Assume the following C function:

```c
#include <stddef.h>

double sum_my_numbers(double* numbers, size_t N)
{
		double result = 0.0;
		size_t i;

		for (i = 0; i < N; i++)
				result += numbers[i];

		// Nasty, change one element.
		numbers[N/2] = - numbers[N/2];

		return result;
}
```

All Rights Reserved

Example (Cont.)

- Let’s save the function in *funcs.c*

- Compile it to SO (Unix **S**hared **O**bject, aka DLL):

gcc -fPIC funcs.c -shared -o libfuncs.so

- Note that since the file has "**c**" extension the function is automatically exposed
- And given the actual name we chose (name mangling)

- With "**cpp**" files, must prefix the function with **extern "C"**
- Otherwise it will be cumbersome to load it in python

All Rights Reserved

Example (Cont.)

- Now we want to call it from python…

```
from ctypes import CDLL, POINTER, c_double, c_size_t
import numpy as np

# Open the SO.
my_lib = CDLL('./libfuncs.so')
# Configure function parameters and return type.
my_lib.sum_my_numbers.argtypes = [POINTER(c_double), c_size_t]
my_lib.sum_my_numbers.restype = c_double

# Create a native array of 100 elements, fill with values and get pointer.
arr = np.linspace(5, 50, 100, dtype=np.double)
arr_ptr = arr.ctypes.data_as(POINTER(c_double))

# Call function and print result.
print(my_lib.sum_my_numbers(arr_ptr, len(arr)))
```

All Rights Reserved

CDLL SO Handling in Linux

- **CDLL** cannot find custom SO files automatically

- Options:
  1. Specify exact path in **CDLL** creation
  2. Specify path in **LD_LIBRARY_PATH** prior to invoking python

All Rights Reserved

Conclusion

- We've seen direct integration of NumPy and C
- **ctypes** is very mature and provides advanced features
- Consult online documentation for further topics

- Sometimes it might still be necessary to use CPython
  - (cases of interpreter lock "GIL" etc.)
- Hope you will not get to that point

All Rights Reserved

- “multiprocessing” module

NODE-LEVEL PARALLEL COMPUTING

All Rights Reserved

“**multiprocessing**” Module

- A standard built-in python module

- Provides **process** level parallelization (SPMD class)
- Simply distributing functions among CPU cores
- Includes many advanced constructs:
  - Synchronization
  - Message Passing
  - Shared memory

- **Important** technical caveat:
  - Can only be used with scripts that include a “main”
  - => Cannot be used from the interpreter ☹

All Rights Reserved

Example Task

- Let's write a function that computes the STD and VAR of randomly generated samples:
  - Accepts a number **N** as input (number of samples to generate)
  - Generates N random samples (of some distribution)
  - Computes the STD and VAR of the samples
  - Returns the results

- We'll write a script that loops over the function with different values of N

All Rights Reserved

Sample Code

```python
import numpy as np
from time import clock

def my_statistics(N):
    samples = np.random.randn(N)
    return (N, np.std(samples), np.var(samples))

if __name__ == '__main__':
    tic = clock()
    # Randomize about 1M samples in 10 iterations.
    for N in range(1000000, 1000010):
        print(my_statistics(N))
    toc = clock()
    print('Total runtime: %f s' % (toc - tic))
```

All Rights Reserved

Parallelization with **Pool**

- **Pool** is a special class in the “**multiprocessing**” module

- Allocates a predetermined number of workers:

```
pool = Pool(6)  # Create a pool of 6 workers
```

- Or dynamically by the number of CPU cores:

```
pool = Pool(cpu_count() - 1)
```

- Distributes function calls among the workers
- A result is returned per function call (if needed)

All Rights Reserved

Example

- Most of the sample code remains the same

```
…
from multiprocessing import Pool, cpu_count

if __name__ = '__main__':
    tic = clock()
    # Create a list of arguments for distribution.
    args = [N for N in range(1000000, 1000010)]
    # Create a pool, leaving 1 core free for the OS.
    pool = Pool(cpu_count() - 1)
    # Run my_statistics in parallel, each worker receives single argument.
    print(pool.map(my_statistics, args))
    toc = clock()
    print('Total runtime: %f s' % (toc - tic))
```

All Rights Reserved

Some Remarks

- Replacing the **for** loop with **Pool** was very simple
- A single **Pool** can be reused many times (**close** when finished)
- More advanced features we didn't review

- ***map*** passes a single call to each worker
- Causes overhead if the function finishes fast
- Pass additional "**chunksize**" to handle multiple calls per worker

- ***map*** blocks until all results arrive
- Use ***map_async*** to run computations in background
- Allows later blocking until results arrive

All Rights Reserved

Some Remarks (Cont.)

- Each item in “**args**” should fully describe a function call
- But map can pass only a single argument

```
def my_func(args):
    <rest_of_function_code>
```

- Solution 1:
  1. Pack arguments in a **tuple/list** before calling: (arg1, arg2,…)
  2. Unpack them inside the function: arg1 = args[0] etc.

- Solution 2:
  1. Pass arguments as **dictionary**: dict(arg1=value, arg2=value…)
  2. Unpacking is trivial: args['arg1'] etc.

All Rights Reserved

Synchronization

- Sometimes a worker needs exclusive access to a resource
  - A file, DB or so
- All other workers must wait to get access too

- This can be accomplished by the **Lock** class
- The same **Lock** object **must** be passed to all workers

- Use lock ***acquire*** function to gain exclusive access
- Call ***release*** to let other workers lock in their turn

All Rights Reserved

Message Passing

- It is possible to exchange synchronized messages and objects:
  - Between workers
  - Between the master and workers

- **Queue** class provides the simplest means to do so
- The sender calls ***put*** to place an object (message)
- The receiver calls ***get*** to get an object (message)
  - If queue is empty the call is blocking until a message arrives

- For those concerned, there are options for asynchronous and non-blocking messaging within this class

All Rights Reserved

Short Summary

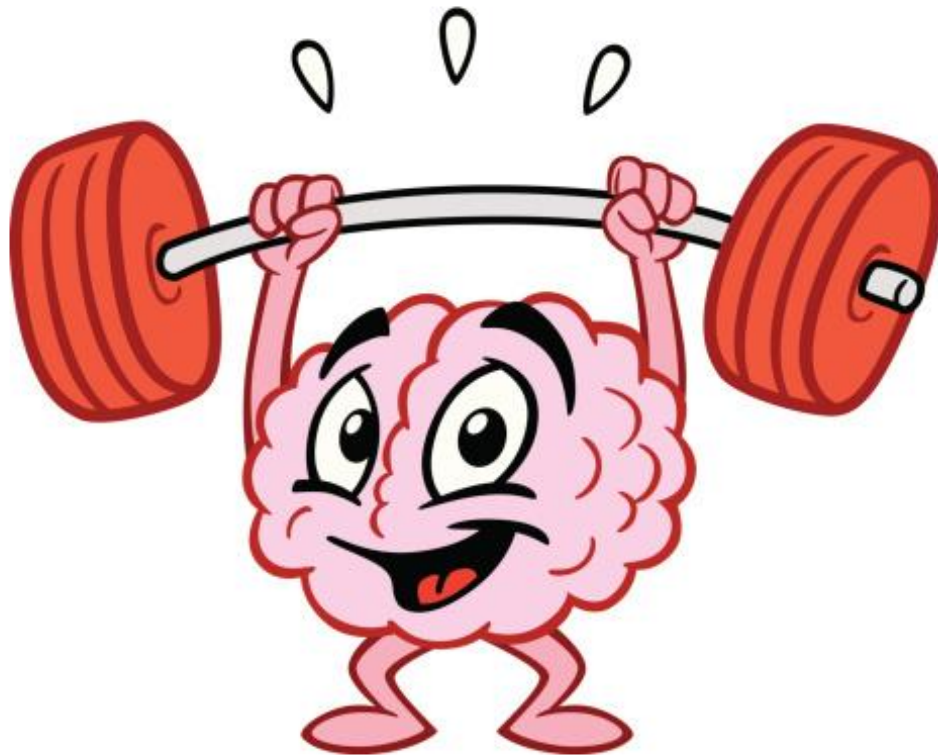
- "**multiprocessing**" provides advanced parallel constructs
- Very simple and intuitive to use

- We got a small taste of what it can offer
- There is more:
  - Shared memory with synchronization
  - More synchronization objects
  - Data passing using pipes
  - Distributing work on remote machines (through **Manager**)
  - …

All Rights Reserved

Hands-On Time

- 30-45 minutes of work



All Rights Reserved

Exercise Benefit

- Registration to PRACE Summer of HPC 2017 requires to submit a simple code test

- You probably guessed it right
- **Computing $\pi(N)$ is a code test for SoHPC 2017**

- You are now ready to apply ☺
- But not during class
- And there is no requirement for parallel solutions!

All Rights Reserved

A Numerical Note

- Floating point numbers are approximated in our computers
- Operations are usually not commutative or associative
  - Due to roundoff errors mostly

- Example 1:
  - 0.1 + 0.2 + 0.3 != 0.3 + 0.2 + 0.1
- Example 2:
  - (0.1 + 0.2) + 0.3 != 0.1 + (0.2 + 0.3)

- Order of operations matters for deterministic values
- Especially true in a parallel world, where order is not guaranteed
- If possible, start summing from smaller values to larger ones

All Rights Reserved

- IPython
- “mpi4py” module

CLUSTER-LEVEL PARALLEL COMPUTING

All Rights Reserved

IPython

- Why another python? It's actually the same but better
  - I stands for **i**nteractive
  - Much more user friendly
  - Reminds Mathematica by Wolfram
  - Equipped with many useful packages (e.g. data visualization)
  - A backend for Jupyter (advanced GUI)

- Take a look here: http://ipython.org
- And for Jupyter: http://jupyter.org

All Rights Reserved

Parallel Processing in IPython

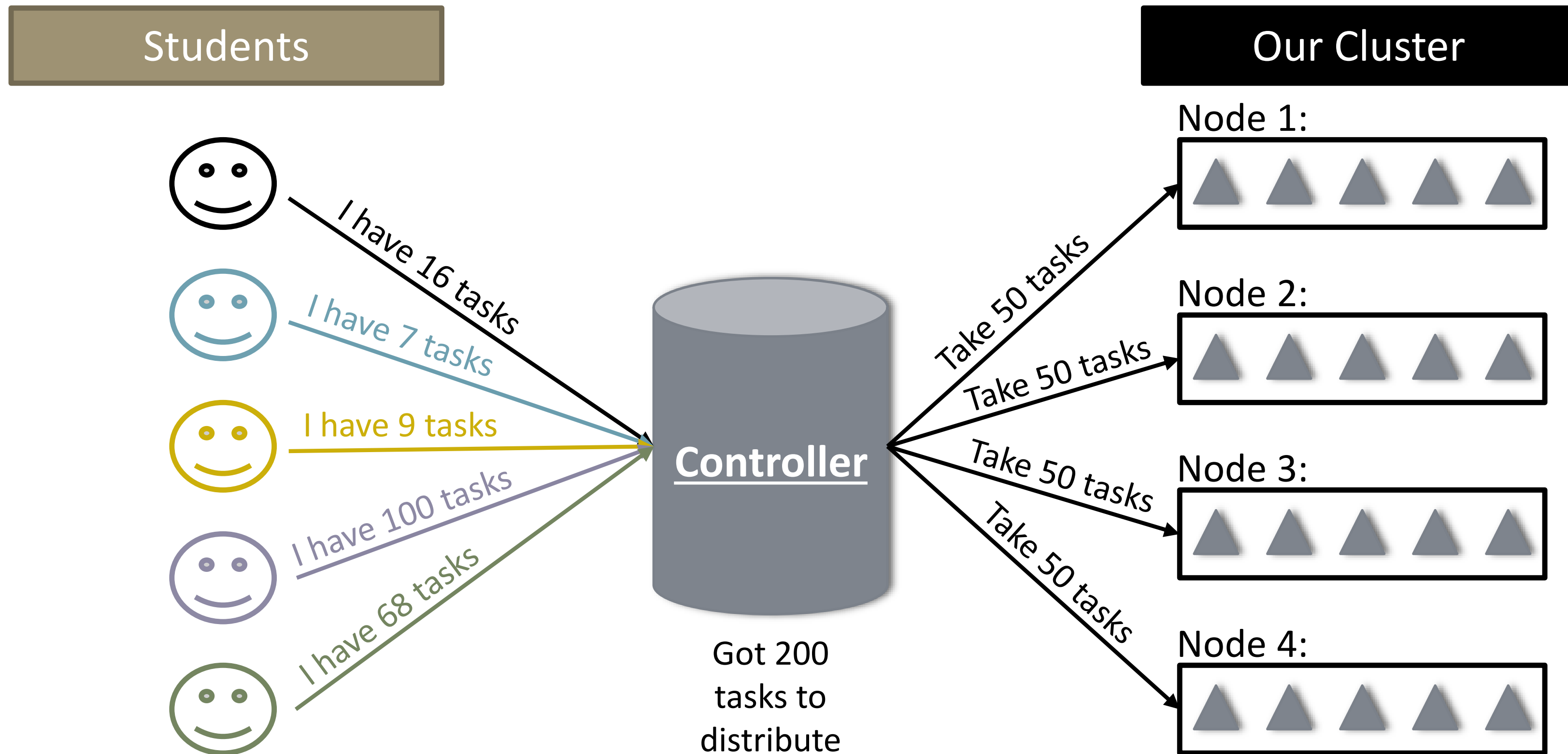
- The main package is **ipyparallel** (download separately)
- Advanced framework for parallel processing in python
- Can be **interactive**!
- Very intuitive parallelization constructs

- Built-in engine for node & cluster scale computations
- Supports many other cluster engines (LSF, SGE, Condor, MPI etc.)

- See: https://ipyparallel.readthedocs.io

All Rights Reserved

ipyparallel Basic Concepts



Arrows are **bi-directional**, also returning results.

= CPU Core
= Compute **engine**

All Rights Reserved 09/02/2017

Starting a Controller

- We are running on a single computer

- To start a controller with 4 compute engines:

ipcluster start -n 4 &

- Or equivalently in separate commands:

ipcontroller &

ipengine &
ipengine &
ipengine &
ipengine &

All Rights Reserved

Starting a Controller (Cont.)

- It is possible to spawn engines on a cluster or remote computers (--ip and --url options)
- See online documentation for further configuration details

All Rights Reserved

Connecting a Controller

- Let’s connect to a controller and submit some work

- Start ipython:

```
# ipython
```

- Connect:

```
import ipyparallel as ipp
c = ipp.Client()
```

All Rights Reserved

Example

- Recall my_statistics function from **multiprocessing**:

```
print(c[:].map_sync(my_statistics, args))
```

- c[:] above means we target all connected engines
- Also called a **DirectView**:
  - We view all engines directly
  - No automatic load balancing
  - It is possible to have a **LoadBalancedView**
  - See online documentation

All Rights Reserved

DirectView At Glance

- In previous slide we mentioned that c[:] is a **DirectView**

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

- It gives some very nice features

- Share data with engines:

```
dview['const1'] = 12.1
dview.push(dict(const1=12.1))
```

- Or get data from engines:

```
print(dview['const1'])
print(dview.pull('const1'))
```

All Rights Reserved

DirectView At Glance (Cont.)

- Similar to MPI we can use scatter operations:

```
dview.scatter('const1', [i**2 for i in range(100)])
```

- And gather operations:

```
const1 = dview.gather('const1')
```

All Rights Reserved

Synchronizing Imports

- **ipyparallel** doesn't know in advance which imports are used by functions
- Contrast to **multiprocessing**

- So before parallelizing functions remember to synchronize imports:

```
with dview.sync_imports():
    import numpy as np
```

- This performs the necessary imports on all engines

- Or use the **@require** decorator with dependencies (preferred)

All Rights Reserved

Synchronizing Imports (Cont.)

- Another elegant option is to embed imports within functions:

```
def my_parallel_function(some_params):
    import numpy as np
    <rest_of_function_code>
```

All Rights Reserved

Hands-On Time

- 30-45 minutes of work

All Rights Reserved

Back to MPI

- Until now we managed to simplify our life
- Starting with **multiprocessing** and then **ipyparallel**

- MPI is a low-level interface for parallel computing
- Standard, supported on Unix-like and Windows platforms
- A lecture given by Dr. Guy Tel-Zur few days ago

- Our goal is not to review MPI again
- Instead we'll see the MPI wrapper provided in Python

All Rights Reserved

mpi4py

- A python module you have to install by request
- Preferably using pip (# *pip install mpi4py*)

- It provides access to almost all MPI 1/2/3 APIs in pythonic way
- Also in object-oriented way
- Allows passing NumPy/SciPy arrays between processes
- And also pure python objects! (if picklable, even custom ones)

- **ipyparallel** is also built on top of **mpi4py**

- See: http://pythonhosted.org/mpi4py
- Also: https://mpi4py.readthedocs.io

All Rights Reserved

mpi4py (Cont.)

- A script runs as a standalone program (SPMD)
- No interactive prompt
- Similar to **multiprocessing**

- Use the usual ***mpiexec*** command to run a program in parallel:

mpiexec –n 4 python my_mpi4py_program.py

All Rights Reserved

mpi4py Example #1

```
from mpi4py import MPI

# Get communicator.
comm = MPI.COMM_WORLD
# Get process ID.
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 'my-string', 'b': 3.14}
    print(‘Master, sending: ', data)
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(‘Worker, received: ', data)
```

All Rights Reserved

Few Notes…

- No need to call the familiar ***MPI_Init***!
- Automatically called once MPI is imported from **mpi4py**

- No need to define "***main***"
- Implicit when importing **MPI**
- The script is encapsulated in a "***main***"

- We are passing a pure python dictionary

- No need to call the familiar ***MPI_Finalize***!
- Automatically called once the python process exits

All Rights Reserved

mpi4py Example #2

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(1000, dtype='d')
    comm.Send([data, MPI.DOUBLE], dest=1, tag=77)
elif rank == 1:
    data = np.empty(1000, dtype='d')
    comm.Recv([data, MPI.DOUBLE], source=0, tag=77)
```

All Rights Reserved

mpi4py Example #3

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(1000, dtype=np.float64)
    comm.Send(data, dest=1, tag=77)
elif rank == 1:
    data = np.empty(1000, dtype=np.float64)
    comm.Recv(data, source=0, tag=77)
```

All Rights Reserved

Few Notes…

- Now we managed to pass NumPy arrays

- It can infer the underlying datatype explicitly (#2)
- Or implicitly (#3)

All Rights Reserved

mpi4py Brief Reference

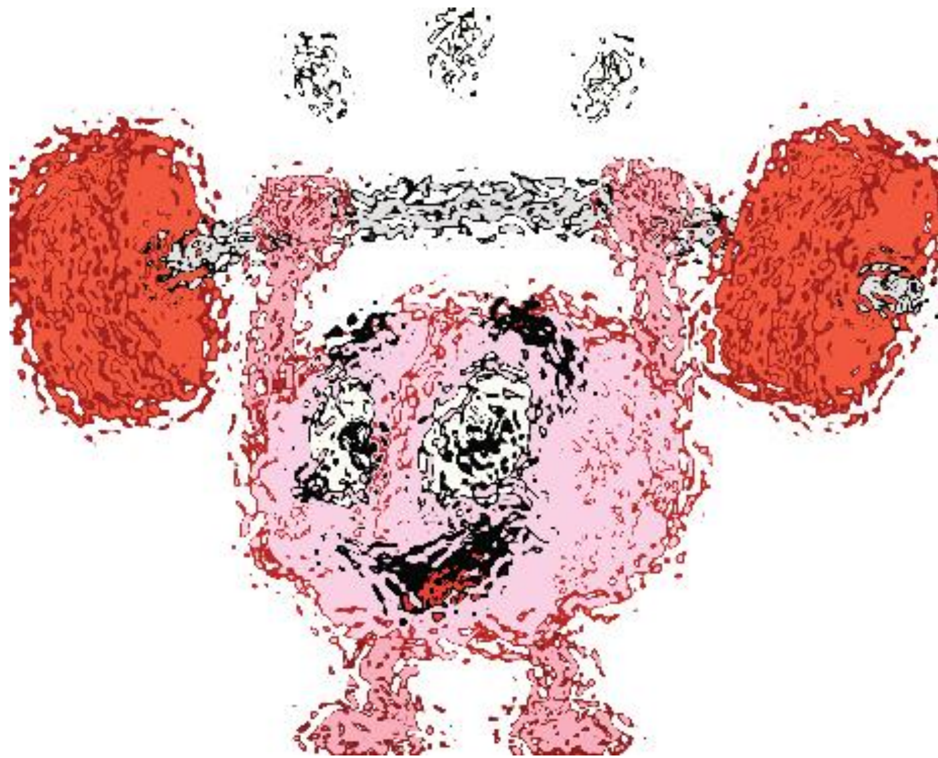
- The module provides support for almost all MPI functions
  - Communication (P2P, Collective, Map-Reduce, Scatter-Gather etc.)
  - Blocking and non-blocking communications
  - I/O
  - Dynamic process management (de-attaching workers)

- Communication functions have two variants:
  1. Buffer based methods – starting with uppercase (e.g. **Send**)
     - These are the familiar MPI functions
  2. Accept generic python objects – all lowercase (e.g. **send**)

All Rights Reserved

Hands-On Time

- 30-45 minutes of work



All Rights Reserved 09/02/2017

Final Remarks

- Had an overview of parallel toolkits/modules in python
- Very useful, simplifying programming tasks
- See online resources for more information

- We covered about 10% of available features
- But a necessary fraction to get started

Hope you enjoyed it ☺

Go Parallel !

All Rights Reserved