Python for High Performance Computing

Andy Gittings
ARC - Scientific Research Software Advisor

Yassamine Mather
ARC - Scientific Software Support Officer

Introduction

Course Overview

Intended audience

Intermediate to advanced Python programmers who develop performance-critical applications

Aims

- Explore the use of Python core packages for scientific computing
- Gain experience with a number of Python solutions for parallel processing

Scientific Computing

The typical research workflow:

- **Generate data**
  - From simulation
  - From experiment
- **Process data**
  - Compute or extract appropriate results
- **Visualise data**
  - Understand the results and gain scientific insight
- **Communicate results**
  - Publications, presentations, web, etc.

All stages above involve software

- The first **two** need scientific software
- This is expensive to run (both in terms of time and hardware)

Scientific Computing

Development of scientific applications

- **Make it run**
  - Software in working state, and produces correct results
  - Explore the scientific problem and spot (major) early-stage software design issues
- **Make it right**
  - Achieve a solid software design
  - Separate application into independent and cohesive units that are easy to maintain
- **Make it fast**
  - Optimize the parts of the program that are not fast enough

We are concerned with the **Make it fast** phase

Scientific computing requires **High Performance Computing (HPC)**

High Performance Computing

HPC is the practice of programming and running intensive applications efficiently on performance hardware.

This can be servers and clusters but not always…

HPC usually refers to **parallel computing**, but in reality the actual processing performance is determined by:

- CPU
- I/O
- Network

The computing performance (how fast simulations can run) determines the scale and complexity of the scientific numerical problems we can solve.

High Performance Computing

Okay, so how fast is fast enough?

A typical desktop PC can deliver tens of Gflops (flops = floating point operations per second)

Tens of millions is a lot of flops but that may or may not be enough

- Extreme example: short range weather forecast
  - The forecast for tomorrow's weather must be delivered in less than one day
    - Each of the MetOffice current models translate into a requirement for ~1 Pflops
    - That is 1 million times more that a desktop PC
    - The simulation has to run on many CPUs, working together on the same model

Why Python?

- **Python is a high level programming language**
  - Conventional syntax (close to pseudocode and mathematics)
  - Safe semantics (any violation produces errors)
  - Supports both procedural and object-oriented programming
  - Easy to prototype
- **Python is a high-productivity language**
  - Simple to learn and use
  - You spend more time thinking about what code does rather than how to write it
  - Relatively easy to do relatively hard things
- **Python has rich scientific computing functionality**
  - Numerical and scientific libraries
  - Domain-specific packages
  - Powerful plotting capabilities
  - Excellent for processing existing Fortran/C/C++ code output to create a pipeline

Python has **excellent** community support

Python and HPC?

- Python is **fast**
  - For writing, testing and developing code

- Python is **slow**
  - For executing research/scientific code, especially repeated execution of low-level tasks

The Performance Gap

What makes Python fast (for development) makes Python slow (for execution)

Python is slow because it is

- Interpreted
- Dynamically-typed
- Serial

The aim of this course is to explore ways in which the performance gap can be bridged.

Cause 1: Python code is interpreted

"Interpreted languages" are slow compared with "compiled languages"

- In fact, interpretation and compilation are properties of the implementation of a language rather than of the language itself

Compiled languages

- A compiler translates the source code directly into machine code
- The machine code is directly executed by the target machine and is thus specific to
  - A target processor
  - An operating system
- The compiler applies a host of optimisations in the translation to machine code

Interpreted languages

- The interpreter executes the source code
- The interpreter is specific to
  - A target processor
  - An operating system
- The interpreter has a "narrow" view of the code (one line at a time) and cannot apply optimisations.

Cause 1: Python code is interpreted

How much slower?

Quite a lot slower - a pure Python implementation could be orders of magnitude slower than its Fortran or C equivalent.

Even the best pure Python implementation (using NumPy) can be slower

Speed test example

We will define a Python function using NumPy arrays (fast Python) that is bound to perform well and compare this with compiled Fortran code.

Cause 1: Python code is interpreted

Here is the Python code:

```
import numpy as np
def my_func_python (x, y):
    return np.sin (x**2 + y**2)
```

Here is the Fortran code (using Fortran Magic):

```
%%fortran -v --opt="-O3"
      subroutine my_func_fortran (x, y, z)
           real, intent(in) :: x(:), y(:)
           real, intent(out) :: z(size(x))
           ! using vector operations
           z = sin (x**2 + y**2)
      end subroutine
```

Cause 1: Python code is interpreted

And now, we time both approaches for large arrays:

```
x = np.random.normal (size=1000000)
y = np.random.normal (size=1000000)
```

Python version:

```
% timeit my_func_python (x, y)
10 loops, best of 3: 38.9 ms per loop
```

Fortran version:

```
% timeit my_func_fortran (x, y)
100 loops, best of 3: 17.4 ms per loop
```

Cause 1: Python code is interpreted

The NumPy implementation

- Is slower
- ..and limited to a single thread of execution

Furthermore

Pure Python (not using NumPy but using loops) is even slower... a lot slower

Plus - the Fortran can be made even faster…

- serial execution: extra compiler optimisation
- parallel execution: OpenMP threading

Cause 2: Python is dynamic

With Static typing

- **The type of a variable declared before the variable is used**
- **All type checking is done at compile time**

```
int num;  // explicit declaration
num = 10; // use the variable
```

With Dynamic typing

- **The type of a variable is inferred from its use**
- **All type checking is done at run time**

```
num = 10  # just use the variable
```

Cause 2: Python is dynamic

Python type-checking overhead

- Every time a variable is used there is a check
- This overhead becomes significant in repeated operations (e.g. a loop)

Python treats every variable and function call like a box of chocolates

- You never know what you're gonna get
- So checks are applied at all steps

Cause 2: Python is dynamic

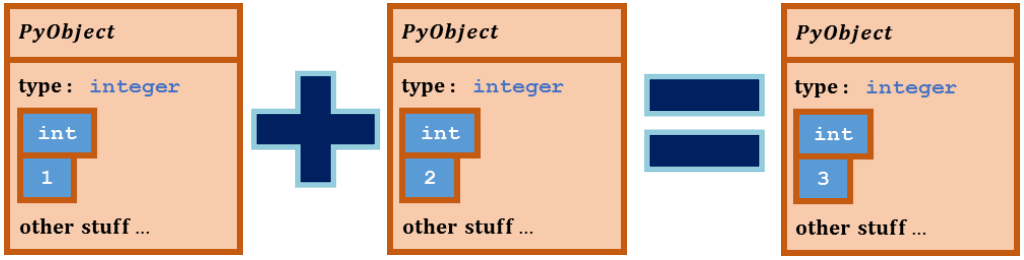
Consider a simple integer addition…

```
>>> 1+2
3
```

Cause 2: Python is dynamic

The way it's interpreted and executed

The operation:



- Is operand 1 of same type as operand 2?
  - Yes: try to do something with them
    - Is the operation requested valid and callable?
      - Yes: call "add" on the two integer arguments and generate a new Python object to contain the result
      - No: Throw an error
  - No: Throw an error

Interpreted computing is safe but there is a lot of overhead

Cause 2: Python is dynamic

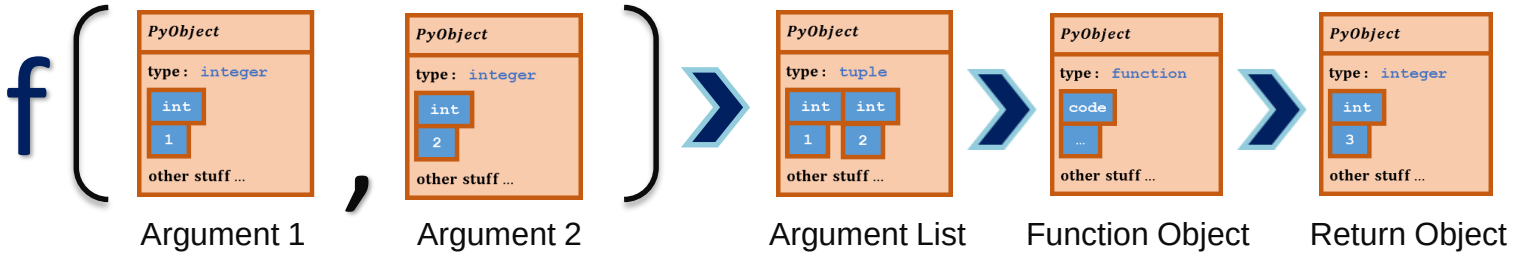
Consider a simple function call…

```
>>> def f(a, b):
    return a+b

>>> f(1, 2)
3
```

Cause 2: Python is dynamic

The way it's interpreted and executed

The function call:



- Is f a callable object?
  - Yes: Form list from arguments
    - Pass argument list to callable
    - The callable object unpacks the argument list and applies code (see previous slide)
    - If successful, form new object and return it
  - No: throw an error

Function calls are safe but there is more overhead (than C or Fortran)

Cause 2: Python is dynamic

```
import time

def f():
    pass # null operation: do nothing

n = 10**6 # a large number of times
npass = 10

t1 = time.time()
for ipass in xrange (npass):
    for i in xrange (n):
        pass
t1 = time.time() - t1
t1/= float (npass)
print "time (no function call) = ", t1, "sec"

t2 = time.time()
for ipass in xrange (npass):
    for i in xrange (n):
        f()
t2 = time.time() - t2
t2/= float(npass)

print "time (function call)     = ", t2, "sec"
print "function call overhead  = ", (t2-t1) / float(n)*1.e+9, "nsec"
```

Cause 2: Python is dynamic

The result…

```
time (no function call) =  0.107432198524 sec
time (function call)    =  0.259676289558 sec

function call overhead  =  152.244091034 nsec
```

Cause 3: Python is serial

The culprit?

The Global Interpreter Lock (GIL) is a major limitation of cPython

The GIL:

- Is a mutual exclusion property with the purpose of preventing race conditions
- It prevents native threads from executing Python bytecodes concurrently…
  - Only a single thread can acquire a lock on a Python object or C API at any one time
- It is necessary because the interpreter is not thread-safe and so the GIL protects access to current thread state and heap allocated objects
- It also prevents Python programs from taking full advantage of multiprocessor systems
- Effectively the language's most difficult technical challenge

Making Python Faster

Make the critical parts faster

- Use the right Python module
  - General purpose, e.g. NumPy, SciPy
  - Specialised modules, e.g. Pandas, SciKits
- Extend Python
  - Write functions in C/C++/Fortran for speed
  - Use existing C/C++/Fortran libraries for functionality
  - Extensions/libraries can be parallel, e.g. NumPy uses Intel MKL
- Parallelise Python code
- Use Just In Time (JIT) compiling

Making Python Faster

Make the critical parts faster

- Pros
  - Fine control over the technical solution
  - Potentially sustainable solution
  - Use existing software
  - Benefit from community support
  - Spread risks
- Cons
  - Development effort
  - Requires learning new techniques

Python Revisited

Using Python

Python can be run interactively

- Directly executing the Python interpreter `python`
  - Run Python interpreter without an input script file
  - Interpreter runs as a Python shell (interactive Python runtime environment)

- Indirectly via a notebook
  - IPython -- enhanced Python shell, ideal for data manipulation and visualisation
  - Jupyter -- web browser-based interactive document

The interactive mode is good for presentations, data inspection, etc.

Using Python

Python can be run non-interactively

Supply the Python interpreter with an input script file, e.g.

```
python myscript.py
```

The non-interactive mode is the normal way to execute production code.

Interactive Shell

Interactive access to interpreter via the `python` command.

Type commands directly at the prompt:

```
bash-prompt> python
>>> print "Hello World!"
>>> import numpy
>>> numpy.__version__
```

Use CTRL-d to exit

Good for testing simple operations, but can be rather limited

- History lost on exit
- No completion or integrated help
- Certain things do not work
- Not suitable for production code

Interactive IPython Shell

IPython extends the standard Python shell by adding a number of useful things

- TAB completion
- help (e.g. `help` or `?int`)
- "magic" commands, e.g.
  - `%run` (run python script within shell)
  - `%hist` (history of commands)
  - `%save` (record state to return later)
- Shell escape to operating system, e.g.

```
!ls -l
```

A Python script

You can use a text editor to create a script file called `hello.py`

```
print "Hello World!"
```

This script file may be executed…

By running from the command-line: `python hello.py`

By running from within an IPython shell using: `%run hello.py`

Scripts are ideal for persistent, reusable, large (complex) code.

jupyter-notebook

- jupyter-notebook (formerly ipython-notebook)
  - Front-end for IPython, a browser-based application
  - Very useful for presentations and interactive use
  - But not so useful for production code
  - Definitely **not** useful in batch processing
  - Run from the command line: `jupyter-notebook`

- Notebook presentations
  - Input is split into cells
  - Cells are "Markdown" (text) and "Code" (Python commands)
  - Cells are "executed" with SHIFT+ENTER
  - "Markdown" cells are displayed
  - "Code" cells execute code

Python Data Types

As we have already seen, Python is dynamically typed.

This means:

- No explicit type declarations
- Data types are worked out from context when code is executed
- Basic types include integers, floating point numbers, strings..

Python Data Types

Assigning values to variables:

```
nmax = 11
pi = 3.14159
string1 = "double quotes"
string2 = 'or single quotes‘

print nmax, pi
11 3.14159
print string1 + " " + string2
double quotes or single quotes
```

Python Data Types

Check variable type using the inbuilt type function:

```
nmax = 11
pi = 3.14159
string1 = "double quotes"
string2 = 'or single quotes'

print type (nmax)
<type 'int'>
print type (pi)
<type 'float'>
print type (string1)
<type 'str'>
```

Data Type Warning

Dynamic typing requires some care, use explicit type casting when needed…

```
nmax = 11
pi = 3.14159

nmax = pi
```

We have just redefined `nmax` as a float..

```
print nmax, type (nmax)
```

```
3.14159 <type 'float'>
```

Instead, use an explicit cast (if at all unsure):

```
nmax = int(pi)
print nmax, type (nmax)
```

```
3 <type 'int'>
```

Integer Division

The Problem

1/2

Returns 0 rather than 0.5

Solutions:

- Cast divisor:

```
1/float(2)
```

- Cast all division operations to floating point operations by importing the Python 3 division:

```
from __future__ import division
```

Integer Division

Another example:

```
nmax = 100  # integer
xmax = 2    # meant to be float but forgot to write 2.0

dx   = xmax / nmax
print dx, type(dx)
0 <type 'int'>

dx   = xmax / float(nmax)
print dx, type(dx)
0.02 <type 'float'>
```

Python Data Structures

Lists

Defining a list

```
mylist = [3, "a", 3.14, False]
```

Access a list element

```
print mylist[2]
```

```
3.14
```

Modify a list element

```
mylist[2] = 1
print mylist[2]
```

```
1
```

Python Data Structures

Dictionary

Defining a dictionary

```
mydict = {"key1" : 1, "key2" : 2}
```

Access a dictionary key

```
print mydict["key2"]
```

2

Modify a key value

```
mydict["key2"] = 1
print mydict["key2"]
```

1

Python Data Structures

Tuple

Defining a tuple

```
mytuple = (1,2,3)
```

Access a tuple

```
print mytuple[2]
```

```
3
```

Tuples are immutable objects

```
mytuple[2] = 1
```

```
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-13-9757188abe0d> in <module>()
----> 1 mytuple[2] = 1

TypeError: 'tuple' object does not support item assignment
```

Python code indentation

Code blocks are indented

- loops
- conditionals
- functions

Myths

- Indentation should be exactly 4 spaces (they can be anything, as long as everything is consistent, see below)
- Tabs and spaces cannot be mixed (they can, although its not a good idea)

Example:

```
for n in range (4):
    print "Iteration", n
print "End of iteration"
```

Python modules

Extending Python functionality

Suppose you need to find the square root of a number Python does not have a **sqrt** function you need the standard library module **math** for that…

So, we need to import the **math** module…

```
import math
```

We can then use the **sqrt** function… (note we reference the method from its class/object)

```
print math.sqrt(25.0)
```

```
5.0
```

Importing modules

Two basic options

`import module`

- Brings an entire module into the current namespace

`from module import name`

- Brings the name object into the current namespace as a local reference
- Use the variant from module import name as name2 in order to
  - Refer object using a shorter name (e.g. np instead of numpy)
  - Avoid namespace conflicts (e.g. two objects with the same name belong to two separate packages)

Note: avoid universal imports, e.g. from module import *

- Diminished code readability (not clear method belongs to what class)
- Polluted namespace (overriding variable and function names)

Python module structure

General: a directory containing

- setup.py
  - package and distribution management
- __init__.py
  - module initialiser e.g. loads other modules
- Python files .py
- directories: tests, documentation, etc.
- sub-modules

Python module structure

Particular: single file

- A file with extension .py that can be imported from Python
- Python environment specified at the top of the file:

```
#!/usr/bin/env python
```

- Python main at the bottom of the file (this allows use as "stand-alone" script or as imported module)

```
if __name__ == "__main__":
    import sys
    some_function_defined_above(sys.argv)
```

Python functions

- In-built functions exist, e.g. int(), type(), range()
- Object methods are accessed with the dot operator (object.method()), e.g. `list.sort()`
- Module functions also accessed with dot operator (module.function()), e.g. `math.sqrt()`
- Defining your own function is easy, e.g.

```
def my_square(x):
    return x*x
```

More about in-built functions:
https://docs.python.org/2/library/functions.html

Functions and variable scope

This can be confusing…

Variables are local to the scope assigned to them e.g.

```
def cat_change ():
    cat = "purring"
    print "inside: cat is " + cat

cat = "meowing"
cat_change ()
print "outside: cat is " + cat
```

```
inside: cat is purring
outside: cat is meowing
```

Functions and variable scope

A variable defined in the main body of a file is global

- Visible throughout the file
- Also visible inside any file which imports that file

A variable defined inside a function is local to that function

- It is accessible from the point at which it is defined until the end of the function
- It exists for as long as the function is executing

Functions and variable scope

Trying to change global variable:

```
def cat_change ():
    cat_local = "purring"
    cat_global = "also purring"
    print "inside: global cat is " + cat_global
    print "inside: local cat is " + cat_local

cat_global = "meowing"
cat_change ()
print "outside: global cat is " + cat_global
```

```
inside: global cat is also purring
inside: local cat is purring
outside: global cat is meowing
```

Functions and variable scope

Trying to change global variable using `global`:

```
def cat_change ():
    global cat_global
    cat_local = "purring"
    cat_global = "also purring"
    print "inside: global cat is " + cat_global
    print "inside: local cat is " + cat_local

cat_global = "meowing"
cat_change ()
print "outside: global cat is " + cat_global
```

```
inside: global cat is also purring
inside: local cat is purring
outside: global cat is also purring
```

Functions and variable scope

This confusion is avoidable…

Global variables are not a good idea

- Functions are passed arguments
- Object-oriented programming encapsulates data with code

Performance: Measuring time

Two ways of measuring processing time

From outside code - measure overall walltime of an entire application or function

From within code - measure execution time of a critical part of the application

Timing from outside code

Linux timing function time

Can be use to measure the execution time of small code snippets, e.g.

```
$ time python example.py
real 0m1.051s
user 0m1.022s
sys  0m0.028s
```

real actual (wallclock) time

user cumulative time spent by all threads of the application

sys cumulative time spent by all cores during system tasks (e.g. memory allocation)

Note: user + sys ≥ real (if threads work in parallel)

Timing from outside code

Python function `timeit`

Usage

- Import module timeit and use timeit.timeit or
- Use the magic IPython command %timeit

By default, `timeit`

- Loops over your code 3 times and outputs the best time
- Reports how many iterations it ran the code per loop
- The number of iterations per loop can be specified, e.g.

```
%timeit -n <iterations> -r <repeats>  <code_snippet>
```

It measures timing by running several times this gives statistical significance to measurements, especially small runtimes

Timing from inside code

The `time` module has two functions

- clock() - returns real time since previous call to clock()
- time() - returns current time in seconds since the Epoch (e.g. 01/01/1970 on Linux, 01/01/2001 on Mac OS)

Both depend on the system time and underlying system time functions

Which one to use?

Timing from inside code

Linux and Mac OS

`time.time()` gives the better precision (microsecond accuracy)

Windows

`time.clock()` gives the better precision

Timing from inside code

Conclusion

Make good use of `timeit` for benchmarking and testing

- Pros: automatically measures a mean value
- Cons: restrictive when used on isolated portions of code from within scripts

Use time when benchmarking portions of code in scripts

- Ideally, run the same piece of code a number of times and average
- Choose between `time.time()` and `time.clock()` depending on the system
- Platform-independent solution: `timeit.default_timer()` points to one of the above

NumPy

NumPy (Numerical Python)

advanced research computing

Best of two worlds: fast development & fast execution

- The de facto standard for scientific computing in Python
- Extends Python with `ndarray`, multi-dimensional arrays classes and methods
- Allows for efficient and fast math calculations
  - Operations "expressed" in NumPy are faster than pure Python
  - Some functionality based on multi-threaded libraries (BLAS, LAPACK, FFTW)
  - Works with the `numexpr` module to exploit multi-core architectures
- Interfaces easily with C and Fortran code
  - Bridge to legacy code
  - A way to write fast executing functions
- **NumPy is the cornerstone of scientific Python computing**
  - A lot of other packages are built on NumPy and rely on it for performance
  - Almost everything in this course depends on NumPy

NumPy (Numerical Python)

Essential NumPy methods

- Multidimensional arrays objects (`numpy.ndarray`)

Other

- Matrices and linear algebra operations
- Random number generation
- Fourier transforms

Complete documentation at https://www.numpy.org/

NumPy & SciPy

SciPy

- Loads a lot of NumPy functionality in its own namespace, e.g. randn, fft
- Overrides the linear algebra and fftw routines from NumPy with more sophisticated versions

The overlap exists because of…

- Development history (NumPy evolved from `numeric`)
- Backward compatibility

The good practice

- From NumPy use only the array classes and methods
- Anything beyond that (e.g. linear algebra, fft, optimisation, ...) belongs to SciPy

NumPy core functionality

Overarching idea: optimise operations on arrays

- Use the `numpy.ndarray` data type and associated methods - (n-dimensional array)
- Get rid of loops by "vectorising operations"

Features leading to array-based performance (NumPy arrays only)

- Ufuncs (universal functions) - these operate on `ndarrays` on an element by element basis
- Aggregations
- Broadcasting
- Slicing, masking, fancy indexing
- Vectorisation

NumPy Arrays

The standard Python library provides lists and 1D arrays (`array.array`)

- Lists are general containers for objects
- Arrays are 1D containers for objects of the same type
- Functionality is limited
- Memory and performance overheads

NumPy provides multidimensional arrays (`numpy.ndarray`)

- Stores elements of the same data type in multiple dimensions
- Efficient storage (and memory access) of data - similar and compatible with Fortran/C/C++ arrays
- More functionality than standard library
- Efficient and fast mathematical operations

Documentation:

http://docs.scipy.org/doc/numpy1.10.0/reference/generated/numpy.ndarray.html

NumPy 1D Arrays

First we import numpy as an alias to shorten code and improve readability:

We can create an array from a list:

```
% a = np.array( [-1, 0, 1] )
% print a
[-1  0  1]
```

Arrays can be copied:

```
% b = np.array( a )
% print b
[-1  0  1]
```

NumPy 1D Arrays

NumPy arrays are of type `ndarray` containing other data types.

We can use the `type()` function and examine the `dtype` property to show this:

```
% print type(b), b.dtype
<type 'numpy.ndarray'> int64
```

NumPy 1D Arrays

We can use `arange` for arrays, in the same way as `range` for lists…

Simple range 0-3:

```
a = np.arange(4)
print a
[0 1 2 3]
```

Range of -2 to 6 in steps of 2:

```
a = np.arange( -2, 6, 2 )
print a, a.dtype
[-2  0  2  4] int64
```

NumPy 1D Arrays

We can use `linspace` to create sample step points in an interval…

Use `linspace` to produce and array containing evenly spaced numbers from -10 to 10 in steps of 5:

```
a = np.linspace(-10, 10, 5)
print a, a.dtype
[-10.  -5.   0.   5.  10.] float64
```

NumPy 1D Arrays

Zeros and Ones

The `zeros` and `ones` routines create an `ndarray` of the specified size initialised with zeros or ones respectively.

```
b = np.zeros(3)
print b, b.dtype
[ 0.  0.  0.] float64

c = np.ones(5)
print c, c.dtype
[ 1.  1.  1.  1.  1.] float64
```

NumPy 1D Arrays

Useful functions

Both `zeros` and `ones` are also useful for creating arrays before their contents are known.

Another function is `empty` :

```
d = np.empty(3)
print d, d.dtype
[ 0.  0.  0.] float64
```

Note: `empty` just allocates memory and does not guarantee entries are zero.

Array Attributes

NumPy keeps track of array metadata as "attributes" of the array structure.

```
a = np.linspace (-10, 10, 5)
```

Lets examine key array attributes…

```
print a
[-10.  -5.   0.   5.  10.]
print "Dimensions ", a.ndim
Dimensions  1
print "Shape      ", a.shape
Shape       (5,)
print "Size       ", a.size
Size        5
print "Data type  ", a.dtype
Data type   float64
print "Object type", type(a)
Object type <type 'numpy.ndarray'>
```

Array Attributes

We can also specify array data at creation time…

An array of single precision floats:

```
a = np.array( [1.1,2.2,3.3], np.float32)
print a
[ 1.10000002  2.20000005  3.29999995 ]
print "Data type", a.dtype
Data type float32
```

NumPy Multi-dimensional arrays

Creation from lists…

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays etc…

For example: a 2D array or matrix can be created from a list of lists

```
mat = np.array( [[1,2,3], [4,5,6]] )
print mat
print "Dimensions: ", mat.ndim
print "Size:       ", mat.size
Print "Shape:      ", mat.shape
[[1 2 3]
 [4 5 6]]
Dimensions:  2
Size:        6
Shape:       (2, 3)
```

Arrays and ufuncs

Universal functions (ufunc)

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features.

A ufunc is a **vectorised** wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

Arrays and ufuncs

Pure Python

```
mylist= range(8)
new_list = [ item + 5 for item in mylist ]
```

NumPy ufunc

```
myarray = arange(8)
new_array = myarray + 5
```

The NumPy ufunc version is more concise and loopless.

Arrays and ufuncs

Example

```
mylist = range (10)

print "Original list", mylist
Original list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print "List iteration", [item + 5 for item in mylist]
List iteration [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

myarray = np.array(mylist)

print "Numpy ndarray", myarray + 5
Numpy ndarray [ 5  6  7  8  9 10 11 12 13 14]
```

Arrays and ufuncs

Performance measurement

```
mylist = range (100000)

%timeit [item + 5 for item in mylist]
100 loops, best of 3: 13.5 ms per loop

myarray = np.array(mylist)
%timeit myarray + 5
10000 loops, best of 3: 33.9 µs per loop
```

Arrays and ufuncs

Overloaded functions

Arithmetic operators: **+ - / %**

Bitwise operators: **& | etc…**

Comparison operators: **< > <= >= == !=**

Math functions: **np.exp np.log np.sin etc…**

Aggregations

Aggregations are array reduction operations

For example, finding the **max**, **sum**, **mean** of an array, for which NumPy has dedicated and fast methods.

Create a random vector

```
x = np.random.random (100000)
```

Standard min function has a lot of type checking

```
%timeit min (x)
10 loops, best of 3: 23.6 ms per loop
```

NumPy min method has minimal checking

```
%timeit x.min ()
10000 loops, best of 3: 33.2 µs per loop
```

Aggregations

Aggregations are array reduction operations

Many functions are available – for example:

np.min(), np.max()

np.sum(), np.prod()

np.mean(), np.std(), np.median(), np.percentile()

There are more…

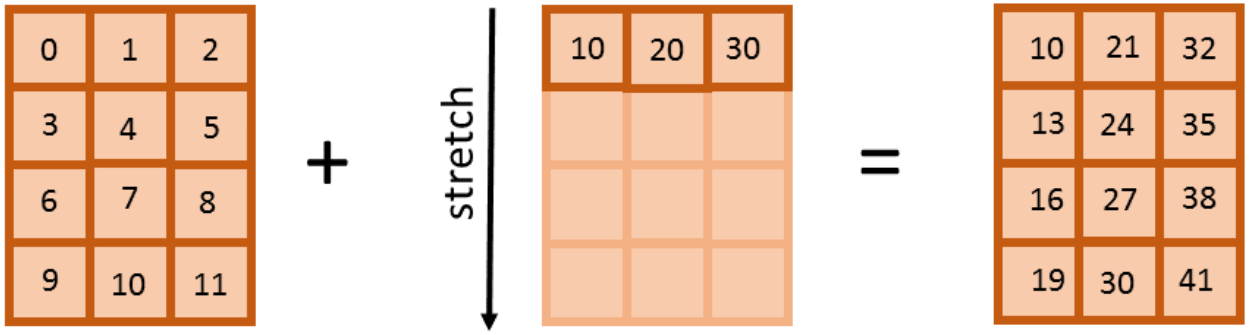
Array Broadcasting

These are a set of rules which determine how ufuncs operate on arrays of different sizes and dimensions.

- **An operation involves two arrays**
- **The arrays must have the same number of dimensions and the length of each dimension is either a common length or one. If neither, an error is raised.**
- **If an input has a dimension size of one in its shape, the first data entry in that dimension will be used for all calculations along that dimension.**
- **The size of the result is the maximum size along each dimension from the input arrays**

Array Broadcasting

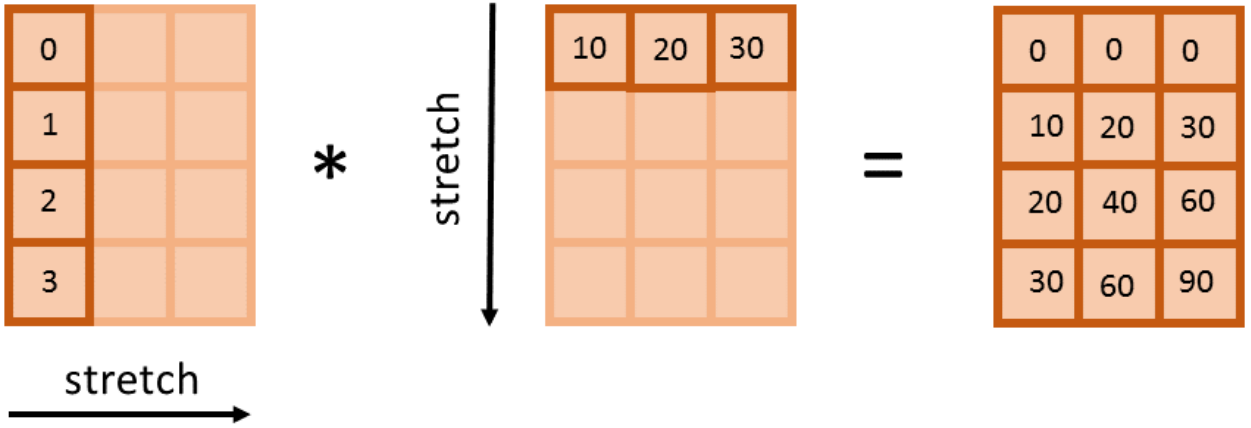
Example 1:




```
x43 = np.arange(12).reshape(4,3)
x13 = np.array([10, 20, 30])

print x43
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
print x13
[10 20 30]

print x43+x13
[[10 21 32]
 [13 24 35]
 [16 27 38]
 [19 30 41]]
```

Array Broadcasting

Example 2:




```
x41 = np.arange(4).reshape(4,1)
x13 = np.array([10, 20, 30])
print x41
[[0]
 [1]
 [2]
 [3]]
print x13
[10 20 30]
print x41*x13
[[ 0  0  0]
 [10 20 30]
 [20 40 60]
 [30 60 90]]
```

Accessing Array Elements

Slicing

[start:stop:step] --> [**start**, and select every '**step**' element until **stop**)

```
a = np.arange(8)
print "a", a
a [0 1 2 3 4 5 6 7]
```

First 4 elements (no step - default 1)

```
print a[0:4]
[0 1 2 3]
```

First 7 elements, step 2

```
print a[0:7:2]
[0 2 4 6]
```

Start at first element, end at last in steps of 2

```
print a[0::2]
[0 2 4 6]
```

Accessing Array Elements

Negative indices are valid

This feature is called **wraparound** and is useful for accessing the last element

```
print "a", a
a [0 1 2 3 4 5 6 7]
```

Last element of array “a”

```
print a[-1]
7
```

Start at element 2, end at third from last element, in steps of 2

```
print a[2:-3:2]
[2 4]
```

Accessing Array Elements

To access multi-dimensional arrays, we can use tuples or index notation:

Example: Index a 3D array - we want to access the value "6" from the following:

```
c = np.array( [ [[1,2],[3,4]] , [[5,6],[7,8]] ] )
```

Index Notation

```
print c[1][0][1]
```

```
6
```

Using a tuple (this is quicker)

```
print c[1,0,1]
```

```
6
```

Accessing Array Elements

To access multi-dimensional arrays, we can use tuples or index notation:

Example: Index a 3D array - we want to access the value "6" from the following:

```
c = np.array( [ [[1,2],[3,4]] , [[5,6],[7,8]] ] )
```

Index Notation

```
print c[1][0][1]
6
timeit c[1][0][1]
1000000 loops, best of 3: 778 ns per loop
```

Using a tuple (this is quicker)

```
print c[1,0,1]
6
timeit c[1,0,1]
1000000 loops, best of 3: 328 ns per loop
```

Views and Slices

A view is an array that refers to data in an existing array

- **You can create a view by selecting a slice of an existing array.**
- **No data is copied when a view is created – a view is just a reference to the data and not a copy.**

```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print a
a =  [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Let's create a view based on a slice starting at row 3 column 2 (the values 8 and 9)

```
s = a[2:, 1:]
print s
[[8 9]]
```

Views and Slices

We can prove the data is only a reference and not a copy

Example: Using our view from the previous slide, assign all the elements in this slice to -2

```
s[:,:] = -2
print "s = ", s
s =  [[-2 -2]]
```

As expected the elements of array s contain -2 but as this is just a view, so do the referred elements of array a

```
print "a = ", a
a =  [[ 1  2  3]
 [ 4  5  6]
 [ 7 -2 -2]]
```

Views and Slices

We can prove the data is only a reference and not a copy

We can test the `base` attribute of the `ndarray` to show if memory is shared between objects.

From our previous example:

```
print s.base is a
True
```

Reshaping arrays

An array can have its size and shape modified

```
a = np.arange(6)
print "a = ", a
a =  [0 1 2 3 4 5]
print "shape of a:", a.shape
shape of a: (6,)
```

Changing shape using `shape` attribute of `ndarray` requires that the size remains the same:

```
a.shape = (3,2)
print "a = ", a
a =  [[0 1]
 [2 3]
 [4 5]]
print "shape of a is now:", a.shape
shape of a is now: (3, 2)
```

Reshaping arrays

An array can have its size and shape modified

We can use the resize() method this may copy or pad depending on shape – it **always** returns a new array:

```
a1 = np.resize(a, (3, 2))
print a1
[[0 1]
 [2 3]
 [4 5]]

print a1.base is a
False
```

In order to perform the next resize, data is repeated to pad out…

```
a2 = np.resize(a, (4, 3))
print a2
[[0 1 2]
 [3 4 5]
 [0 1 2]
 [3 4 5]]
```

Reshaping arrays

An array can have its size and shape modified

We the reshape() method works differently. It gives a new shape to the array without changing its data or size – similar to changing the `shape` attribute.

```
a1 = np.reshape(a, (3, 2))
print a1
[[0 1]
 [2 3]
 [4 5]]

print a1.base is a
True
```

The following won't work… due to the size being too small.

```
a2 = np.reshape(a, (4, 3))
ValueError: cannot reshape array of size 6 into shape (4,3)
```

Fancy Indexing

Fancy indexing is indexing using another NumPy array containing integer or Boolean values

This is advanced indexing, which lets you do more than simple indexing.

```
p = np.array([[ 0,  1,  2],[ 3,  4,  5],
              [ 6,  7,  8],[ 9, 10, 11]])
print p
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

r = [0,0,3,3]
c = [0,2,0,2]
q = p[r,c]

print q
[ 0  2  9 11]
```

q is a new array (not a view) containing elements from array p indexed by rows array r and columns array c.

Fancy Indexing

Logical expressions and Boolean masks can be used to find indices of elements of interest

For example find the indices of elements with a value less than zero:

```
m = np.array( [[0,-1,4,20,99],[-3,-5,6,7,-10]] )
print m
[[  0  -1   4  20  99]
 [ -3  -5   6   7 -10]]

print m[ m < 0 ]
[ -1  -3  -5 -10]

print m<0
[[False  True False False False]
 [ True  True False False  True]]
```

Array copies

Simple assignment creates references or shallow copies of arrays.

```
a = np.array( [-2,6,2] )
print a
[-2  6  2]

b = a
a[0]=20

print a
[20  6  2]

print b
[20  6  2]
```

Array copies

If you need to make a copy of an array, use copy() to create a true or deep copy.

```
a = np.array( [-2,6,2] )
print a
[-2   6   2]

b = a.copy()
a[0]=20

print a
[20   6   2]

print b
[-2   6   2]
```

Vectorisation

Vectorisation is the replacement of explicit loops by array expressions.

- Allows element-wise operations on arrays
- No loops involved
- Efficient element-wise operations
- Uses all the features we have seen previously (indexing, broadcasting, etc.)

Vectorisation is powerful

- One or two (or more) orders of magnitude faster than the pure Python equivalent

Vectorisation needs extra memory

- Trade off between time and space (i.e. memory)
- Replacing loops may need extra variables for intermediary results

Vectorisation

Example: operations on arrays of matching size:

```
x = np.random.randn(1000)
y = np.random.randn(1000)

timeit [x[i]*x[i] + y[i]*y[i] for i in xrange(1000)]
1000 loops, best of 3: 1.6 ms per loop

timeit x*x + y*y
100000 loops, best of 3: 4.17 µs per loop
```

Vectorisation

Example: many kinds of operations are possible

```
a = np.arange(10).reshape([2,5])
b = np.arange(10).reshape([2,5])

print a
[[0 1 2 3 4]
 [5 6 7 8 9]]

print b
b= [[0 1 2 3 4]
 [5 6 7 8 9]]
```

Multiply a and b, entry by entry (note this is NOT matrix multiplication)

```
print a*b
[[ 0  1  4  9 16]
 [25 36 49 64 81]]
```

Vectorisation

Example: efficient element-wise math functions on arrays

Efficient vectorised computation

```
print np.sqrt(np.array([4, 9, 16]))
```

```
[ 2.  3.  4.]
```

Math operations

NumPy provides a host of math functions – but so does the Python `math` library.

Which one should be used?

- `math`
  - Mathematical functions defined by the C standard
  - Fast operations on regular Python types
- NumPy
  - Mathematical functions defined as ufuncs
  - Slower than `math` on Python scalars
  - Faster than `math` on NumPy arrays

NumExpr

- **Fast numerical expression evaluator for NumPy**
  - **An easy-to-use way to compute array-based operations**
  - **Faster than pure NumPy by up to an order of magnitude**

- **Accelerates expressions operating on NumPy arrays**
  - **Faster execution**
    - **Expression parsed**
    - **Array operands are split into small chunks that easily fit in the cache of the CPU**
  - **Less memory**
    - **Allocating memory for intermediate results is avoided**
- **Parallelism**
  - **The operations are multi-threaded**
  - **Support from Intel's VML (Vector Math Library), part of Math Kernel Library (MKL)**

NumExpr

Performance Example:

Generate large arrays

```
n = 10000000;
x = np.random.rand(n)
y = np.random.rand(n)
z = np.random.rand(n)
```

Normal NumPy version

```
%timeit np.sqrt(np.sum(x**2 + y**2 + z**2))
10 loops, best of 3: 116 ms per loop
```

NumExpr version

```
%timeit ev = ne.evaluate('sum(x**2 + y**2 + z**2)');
ne.evaluate('sqrt(ev)')
10 loops, best of 3: 45.4 ms per loop
```

NumExpr

More information

https://numexpr.readthedocs.io/en/latest/

Random Number Generation

NumPy provides functions for Random Number Generation (RNG)

For example: `np.random.rand()`

However, RNG in parallel can be a source of error:

What happens when you need to generate random numbers for different Python processes running concurrently on the same system? (This is a frequent occurrence, for example in Monte Carlo simulations.)

The default seed for the RNG is /dev/urandom (or the Windows analogue). Parallel processes that use the default may easily end up generating identical streams of numbers.

Random Number Generation

Seeding the RNG

Clearly, each process must use something other than the default to seed the RNG. But what?

Sequential seeds

A good quality RNG produces uncorrelated output for any two different seeds (including consecutive seeds). Therefore we can simply use the process number or rank to seed the RNG.

Numba

Overview

Numba

- Project sponsored by Continuum Analytics
- Core is a compiler for Python array and numerical functions
- Just In Time (JIT) compilation
- Exploits the LLVM compiler to generate optimised machine code

Features

- On-the-fly code generation (at import time or runtime, according to user choice)
- Native code generation for the CPU (default) and GPU
- Integration with the Python scientific software stack (thanks to NumPy)

Numba – Example 1

Fibonacci sequence (Python NumPy):

```
import numpy



def fib (n):
    """Compute Fibonacci sequence"""
    s = numpy.zeros(n, dtype=numpy.float64)
    if n >= 1:
        s[0] = 1.0

    if n >= 2:
        s[1] = 1.0

    for i in range (2, n):
        s[i] = s[i-1] + s[i-2]

    return s
```

Numba – Example 1

Fibonacci sequence (Python NumPy):

```
timeit fib (1024)
1000 loops, best of 3: 930 µs per loop
```

Numba – Example 1

Fibonacci sequence (Numba):

```
import numpy
from numba import jit

@jit
def fibJIT (n):
    """Compute Fibonacci sequence"""
    s = numpy.zeros(n, dtype=numpy.float64)
    if n >= 1:
        s[0] = 1.0

    if n >= 2:
        s[1] = 1.0

    for i in range (2, n):
        s[i] = s[i-1] + s[i-2]

    return s
```

Numba – Example 1

Fibonacci sequence (Numba):

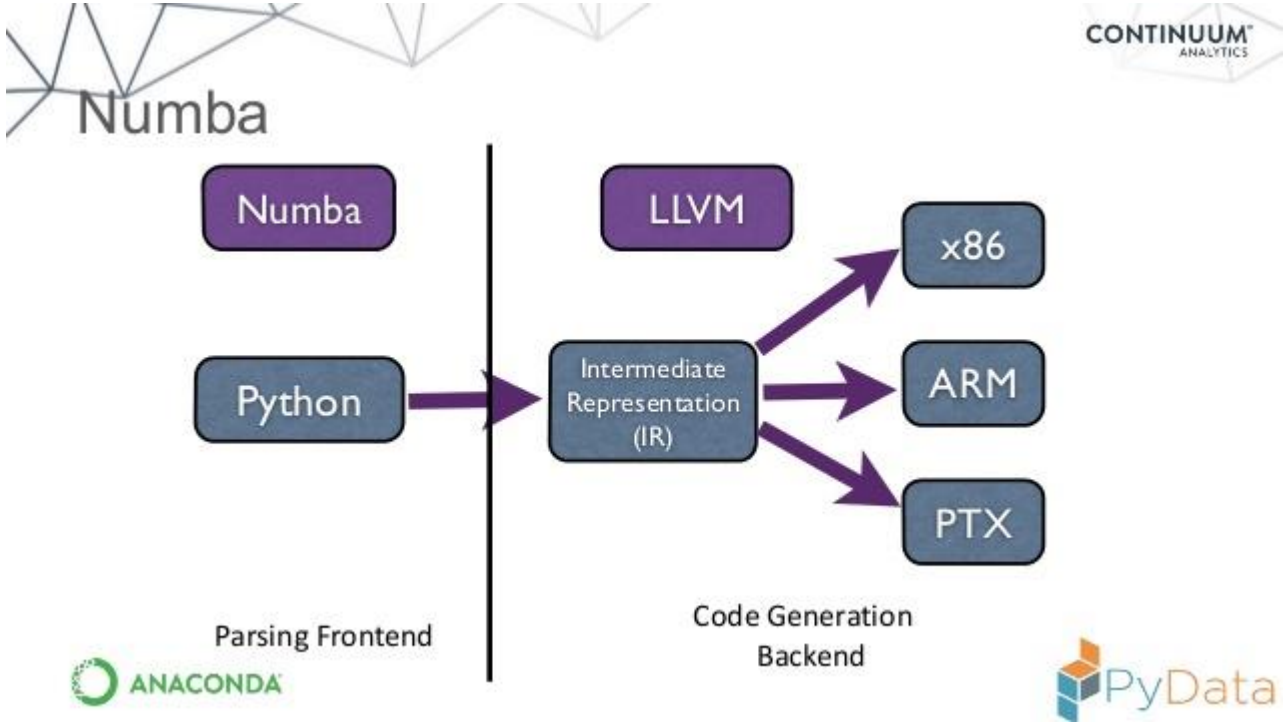
Run once just to generate machine code and cache the results…

```
fibJIT (16)
```

Perform timing run…

```
timeit fibJIT (1024)
100000 loops, best of 3: 3.95 µs per loop
```

How Numba works



Other features

- Powerful type inference
- Multi-dispatch compilation
- A function called with two different types generates two cached versions

Numba – Example 2

Task

- Compute the p-norm of a matrix
- Reference implementation is `numpy.linalg.norm`

Steps

- Write Python code, decorate it and execute it once (to cache results)
- Compare accuracy and performance against the NumPy equivalent

Numba – Example 2

Our Numba decorated code:

```
import math
import numpy
from numba import jit

@jit
def p_norm_JIT (u, p):
    n = u.size
    s = 0.0
    for i in range (n):
        s += math.pow (math.fabs(u[i]), p)

    return math.pow (s, 1.0 / float(p))
```

Numba – Example 2

Test Accuracy (against numpy.linalg.norm**):**

```
% n = 1024
% u = numpy.random.rand (n)

% nrm0 = numpy.linalg.norm (u, 3)
% nrm1 = p_norm_JIT (u, 3)


% print "relative error =", math.fabs ( (nrm0 - nrm1) / nrm0 )
relative error = 0.0
```

Numba – Example 2

Test Performance (against numpy.linalg.norm**):**

```
% n = 25000000
% u = numpy.random.rand (n)

% timeit numpy.linalg.norm (u, 3)
1 loop, best of 3: 1.94 s per loop

% timeit p_norm_JIT (u, 3)
10 loops, best of 3: 39 ms per loop
```

An impressive increase in performance...

- One import
- One decoration

Numba compares well with C/Fortran

Numba

More useful features

- Automatic function inlining
- Creating fast ufuncs using `@numba.vectorize`
- Writing fast generators (functions that return a sequence, such as xrange)
- Targeting GPU execution

Limitations

Numba is under rapid development, some features are missing

Parallel execution is a possibility – best implementation using `multiprocessing`

Summary

Numba

- An exciting project with promising future
- Extremely easy to use (essentially the API has only one feature; the decorator @jit)
- Extremely easy to install
- As performant as C/Fortran or Cython (on a single core)

Cython

Cython - Overview

Cython: you can have your cake and eat it... almost

- Enjoy the benefits of fast executing C code without practically leaving Python
- Requires a less Pythonic thinking and a more C-like thinking (familiarity with C pays)
- Bonus feature: multithreaded parallelism

Cython is

- An open-source project (http://cython.org)
- A Python compiler (nearly)
- An extension to the Python language for…
  - Writing fast-executing extension modules
  - Interfacing Python with C libraries

Cython is stable and mature

Establishing itself as a pillar of the scientific Python ecosystem

Example

Fibonacci sequence

```
import numpy
def fib (n):
    """Compute Fibonacci sequence"""
    s = numpy.zeros(n, dtype=numpy.float64)
    if n >= 1:
        s[0] = 1.0

    if n >= 2:
        s[1] = 1.0

    for i in range (2, n):
        s[i] = s[i-1] + s[i-2]

    return s
```

Example…

```
% timeit fib(1024)

1000 loops, best of 3: 926 µs per loop
```

Example…

```
%%cython

import numpy
cimport numpy

# same function as before but with data types
cpdef double[:] cfib (int n):
    """Compute Fibonacci sequence"""
    # declare types of output data
    cdef double[:] s
    # declare types of function local data
    cdef int i

    # the rest is the same
    s = numpy.zeros(n, dtype=numpy.float64)
    if n >= 1:
        s[0] = 1.0

    if n >= 2:
        s[1] = 1.0

    for i in range (2, n):
        s[i] = s[i-1] + s[i-2]

    return s
```

Example…

```
% timeit cfib(1024)
```

```
100000 loops, best of 3: 7.08 µs per loop
```

A massive speed up from the 926µs of pure Python

- 3 extra lines of code
- 2 types added in function definition

Cython Pros and Cons

Pros

- 99% Python (versions 2 and 3 compatible)
- Supports functionality both ways
  - Running C extensions from Python
  - Using Cython functions from C
- Incremental development
  - Standard Python is valid Cython
  - Speed code up by adding C features

Cython Pros and Cons

Cons

- Requires compilation
- CPython specific (does not work with other implementations, e.g. PyPy)

Documentation

http://cython.readthedocs.io/en/latest/

http://cython.readthedocs.io/en/latest/src/userguide/parallelism.html

How Cython works…

It writes C code so you don't have to…

- Cython generates C code from Python-like code
- A C compiler compiles the generated C code
  - All major compilers supported on all major platforms
- Most of Python syntax can be Cythonized
  - Top-level classes and functions
  - Control structures: loops, with, try-except/finally, ...
  - Object operations, arithmetic, ...

How Cython works…

But the more you can help, the better

- The more specific you can be about variables and functions
  - The more the generated C code uses C instead of Python API and…
  - The more the compiler can apply optimisations
- *aim: getting away from Python safety towards C optimised performance*

3 Steps to performance…

Step 1:

Remove Python object overheads

- **Tell Cython the types of variables…**

```
cdef int i, j, k
cdef float x, y[10], *z
```

3 Steps to performance…

Step 2:

Remove Python function call overheads

- **Tell Cython**
  - **how to turn functions into C or**
  - **how to use C functions directly**

```
# Define Python function
def foo (int i, char *s):
# C function, not visible to Python code that imports the module
cdef int foo2 (int i, char *s):
# use function from C library directly
from libc.math cimport sin
cdef double x
s = sin(x)
```

3 Steps to performance…

Step 3:

Remove Python check overheads

- **use compiler directives: tell Cython about any Python checks that can be skipped**

```
# True = avoid division checks (e.g. ZeroDivisionError)
@cython.cdivision(True)
# False = do not check that indexing operations raise IndexErrors
@cython.boundscheck(False)
# False = do not check for negative index handling (possibly causing
            segfaults or data corruption)
@cython.wraparound(False)
```

Cython workflow…

Write Cython code:

- **.pyx files: Python-like code**
- **.pxd files: Cython header files (optional)**

Cython translates this into C code and a compiler builds a shared library

Option #1: use a translate-compile two step procedure, e.g.

```
cython mycode.pyx
gcc -O2 -Wall -shared -fPIC $(pkg-config --cflags --libs python) -o
    myext.so myext.c
```

Option #2: build and install using distutils

```
python setup.py build_ext --inplace
python setup.py install --prefix=./
```

Cython workflow…

Python imports the module and uses it…

```
import mycode
…
…
mycode.func ()
```

Worked Example

Task

Compute the p-norm of a matrix

Reference implementation is `numpy.linalg.norm`

Steps

- Write cython code
- Build module
- Test module

Worked Example

Incremental Cython code development

- **Start with pure Python**
- **Type the variables**
- **Use external C functions**
- **Remove the GIL and add multithreading**
- **Add Cython compiler directives to eliminate checks**

Worked Example

Start with a pure Python function

```
import math

def p_norm (u, p):
    n = u.size
    s = 0.0
    for i in range (n):
        s += math.pow (math.fabs(u[i]), p)

    return math.pow (s, 1.0 / float(p))
```

Note:

- Python code is legal Cython code

Worked Example

Add types

```
import cython
cimport cython

cpdef double p_norm_types (double [:] u, int n, int p):
    cdef:
        int i
        double s

    s = 0.0
    for i in range (n):
        s += math.pow (math.fabs(u[i]), p)

    return math.pow (s, 1.0 / float (p))
```

Worked Example

Add external C functions

```
from libc.math cimport pow, fabs

cpdef double p_norm_types_better (double [:] u, int n, int p):
    cdef:
        int i
        double s

    s = 0.0

    for i in range (n):
        s += pow (fabs(u[i]), p)

    return pow (s, 1.0 / float (p))
```

Worked Example

Add multithreading and release the GIL

```
from cython.parallel import prange, parallel
cimport openmp

cpdef double p_norm_openmp (double [:] u, int n, int p, int nt=2):
    cdef:
        int i
        double s

    s = 0.0
    openmp.omp_set_num_threads (nt)
    with nogil:
        for i in prange (n):
            s += pow (fabs (u[i]), p)

    return pow (s, 1.0 / float (p))
```

Worked Example

Notes:

- **Parallel region defined via `prange`**
- **The parallel region releases the GIL:**
  - `with nogil:`
- **Number of threads to use in parallel region set via function argument…**
  - **Passed on to the OpenMP `omp_set_num_threads ()` function**
- **The reduction on variable s is inferred from context**
  - `s +=` **works**
  - `s = s +` **does not**

Worked Example

Final touches

```
@cython.cdivision(True)
@cython.boundscheck(False)
@cython.wraparound(False)

cpdef double p_norm_openmp_better (double [:] u, int n, int p, int nt=2):
    cdef:
        int i
        double s
    s = 0.0
    openmp.omp_set_num_threads (nt)
    with nogil:
        for i in prange (n):
            s += pow (fabs (u[i]), p)

    return pow (s, 1.0 / float (p))
```

Worked Example

Notes:

The Cython generator for C code is given additional hints to simplify code using compiler directives

```
boundscheck=False
wraparound=False
cdivision=True
```

-- guarantee array bounds are respected

-- guarantee negative indices are not used

-- guarantee division is safe, avoid expensive checks (e.g. division by 0)

Worked Example

Build module

```
python setup.py install --prefix=$PWD
```

```
running install
running build
running build_ext
running install_lib
running install_egg_info
Removing /home/conda/work/dtc-2019-python-master/lecture10-
cython/lib/python2.7/site-packages/UNKNOWN-0.0.0-py2.7.egg-info
Writing /home/conda/work/dtc-2019-python-master/lecture10-
cython/lib/python2.7/site-packages/UNKNOWN-0.0.0-py2.7.egg-inf
```

Worked Example

Test

First, test accuracy against `numpy.linalg.norm` then, test performance

```
python test.py
norm is  6.39805642224
relative errors [%] [  1.38820036e-14   1.38820036e-14   0.00000000e+00
0.00000000e+00]

 performance ...
                                        linalg.norm:  2.022527
                          pure python code cythonized 22.321308
                                       adding C types 16.937845
                                   adding C functions 1.788117
                        using types + OpenMP (1 thread) 2.519696
                       using types + OpenMP (2 threads) 1.317199
                       using types + OpenMP (4 threads) 0.876701
           using types + OpenMP + no checks (4 threads) 0.590176
```

Three types of function declarations

def

- **Python, basically**
  - **Called directly from Python**
  - **Python objects as arguments**
  - **Returns a Python object**

Three types of function declarations

cdef

- **Pure C functions**
  - **C code effectively, all types must be declared**
  - **Cython optimises aggressively**
  - **Pros: fastest executing code**
  - **Cons: declared functions not visible to code that imports the module**

Three types of function declarations

cpdef

- **Both C and Python**
  - **gets compiled to two functions**
    - **a cdef for C types (for fast execution)**
    - **a def for Python types (for compatibility)**
  - **Pros: visible to code that imports the module**
  - **Cons: version using Python objects can be as slow as def version**

Functions and type coercion

Argument type checks are automatic where variables are typed…

```
%%cython

def func (x):
    return x + 1
```

This works as intended

```
print func (1)
2
```

This raises a type error from the "add" operation itself

```
print func ("abc")
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-5156d55267bc> in <module>()
----> 1 print func ("abc")

_cython_magic_ec0feaddd93f4ac03dafa59c2ae701b2.pyx in
_cython_magic_ec0feaddd93f4ac03dafa59c2ae701b2.func()

TypeError: cannot concatenate 'str' and 'int' objects
```

Functions and type coercion

```
%%cython

def func_int (int x):
    return x + 1
```

This function has a typed argument and the error is from an argument type check

```
print func_int ("abc")
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-7ae0a2b6f4bd> in <module>()
----> 1 print func_int ("abc")

_cython_magic_ec0feaddd93f4ac03dafa59c2ae701b2.pyx in
_cython_magic_ec0feaddd93f4ac03dafa59c2ae701b2.func_int()

TypeError: an integer is required
```

Functions and type coercion

```
%%cython

# low-level C function, callable from C
cdef cdef_func_int (int x):
    return x + 1
```

Low-level C function, callable from C, not visible from Python

```
print cdef_func_int ("abc")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-a527c26877c7> in <module>()
      1 # low-level C function, callable from C, not visible from Python
----> 2 print cdef_func_int ("abc")

NameError: name 'cdef_func_int' is not defined
```

Functions and type coercion

```
%%cython

# low-level C function, callable from C + wrapper, callable from Python
cpdef cpdef_func_int (int x):
    return x + 1
```

Hybrid function, visible from Python, arg type check

```
print cpdef_func_int ("abc")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-3c4942895ae2> in <module>()
      1 # hybrid function, visible from Python, arg type check
----> 2 print cpdef_func_int ("abc")

_cython_magic_ec0feaddd93f4ac03dafa59c2ae701b2.pyx in
_cython_magic_ec0feaddd93f4ac03dafa59c2ae701b2.cpdef_func_int()

TypeError: an integer is required
```

Summary

Cython magic

Python code is enhanced with

- **Variable types**
- **External libraries functions**
- **Multi-threading constructs**
- **Extra directives for the Cython compiler**

Nevertheless, code remains mostly Python

Summary

Cython performance achieved by…

- **Compiling code**
  - **Applying standard compiler optimisation (which the interpreter cannot)**
  - **Pure Python cythonized cuts runtime by 20-50%**

- **Pruning the original Python down to essentials**
  - **Eliminate dynamic typing**
  - **Eliminate checks (e.g. bound checks)**
    - **Note: 300x speedup on pure Python Cythonized**
  - **Overcoming the GIL and using OpenMP to multithread**
    - **Note: very easy programming**

Multiprocessing

Models of parallelism: **Distributed Memory**

Distributed Memory Programming Model:

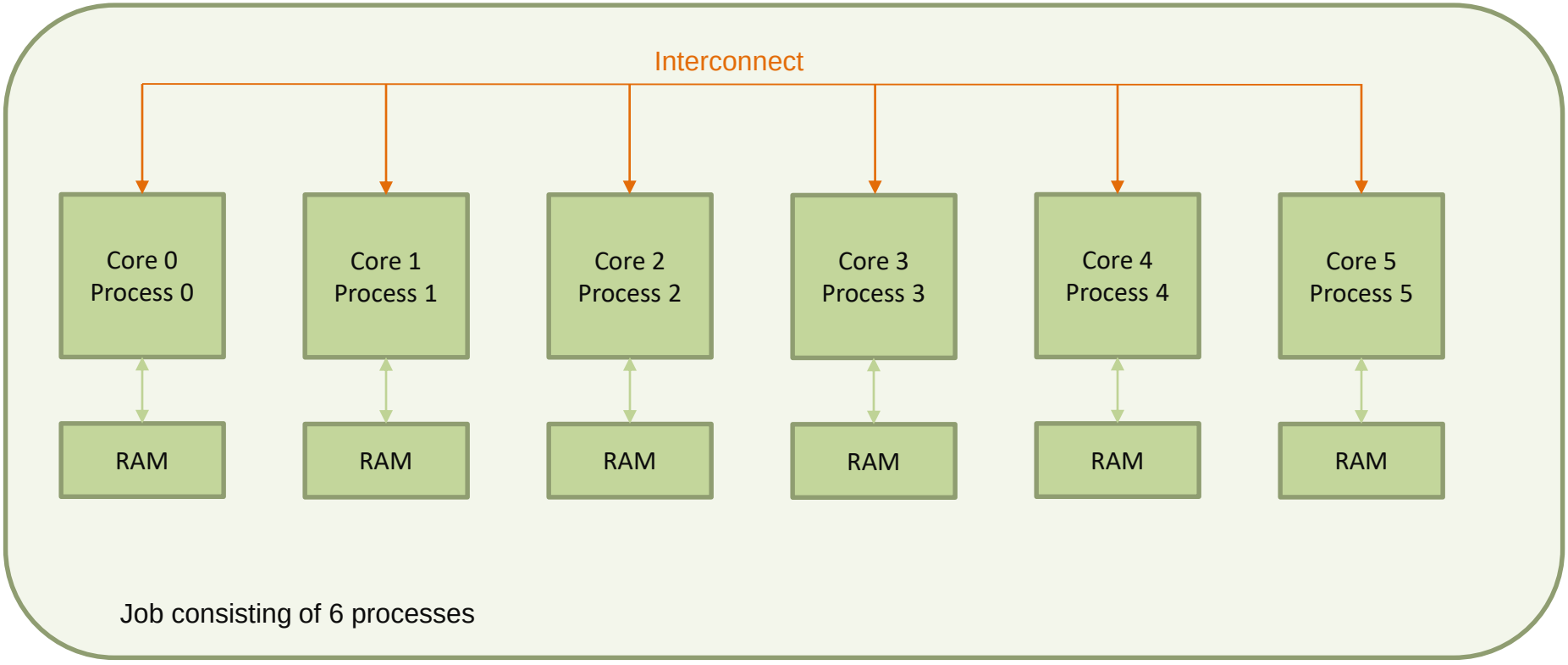
- multi-core system, each core has its own private memory
- local core memory is invisible to all other processors
- agent of parallelism: the **process** (program = collection of processes)
- exchanging information between processes requires **explicit** message passing
- the dominant programming standard: **MPI**

Distributed Memory Hardware:

- conceptually, many PCs connected together (traditional Beowulf cluster)
- current approach:
  - multi-core computer nodes (high-density blades) with own memory
  - high-bandwidth, low-latency network connection
  - off-the shelf modular technology (high-end CPUs, standard hard disk)
  - accounts for the largest HPC systems

Distributed Memory ARC systems: the **arcus-b** cluster (but any machine can be programmed using this model)

Distributed Memory View



Models of parallelism:
Shared Memory

Shared Memory Programming Model:

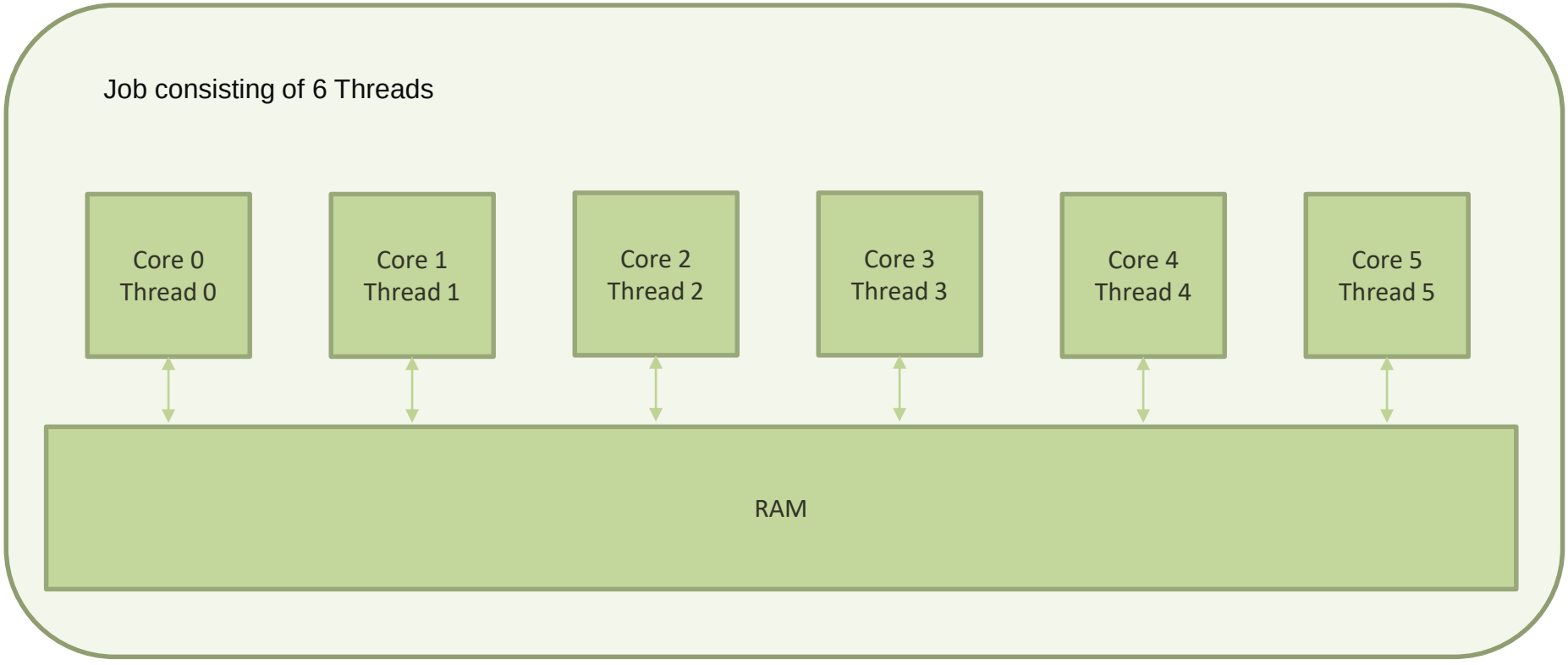
- multi-core system
- each core has access to a shared memory space
- agent of parallelism: the **thread** (program = collection of threads)
- threads exchange information **implicitly** by reading/writing shared variables
- the dominant programming standard: **OpenMP**

Shared Memory Hardware:

- conceptually, a single PC, with a large memory and many cores
- accounts for both small and inexpensive systems (desktops) and very large and expensive system (with very expensive high bandwidth memory access)

Shared Memory ARC Systems: **arcus-HTC** and any *single node* of the **arcus-b** cluster.

Shared Memory View



Distributed Memory v. Shared Memory

- Distributed Memory:
  - **Can scale to any number of cores**
  - Requires special tools to compile and run the code
    - Typically `mpicc` or `mpif90` to compile, `mpirun` to run it
  - Can be harder to program that shared memory
  - But will generally perform better if done well
  - And it teaches good parallel programming “habits”
- Shared memory
  - **Is usually limited to the number of cores in a node**
    - Can overpopulate, good for debug, bad idea for performance
  - Generally just requires an extra flag on the compiler
  - Can be easier to program than distributed memory
  - It is often hard to get good parallel performance
    - Sharing things is not good for parallelism …
  - Can easily let people be a bit sloppy when programming …

Multiprocessing

The `multiprocessing` library (from Python 2.6 on)

- Designed to exploit multiprocessor shared memory systems
- Copies (in part) the functionality of the threading library
- Uses processes instead of threads, sidestepping the Global Interpreter Lock (GIL) limitations of cPython

Remember the GIL?

- GIL is a mutex that prevents native threads from executing Python bytecodes concurrently
  - Only a single thread can acquire a lock on a Python object or C API at any one time
- It is necessary because the interpreter is not thread-safe
  - The GIL protects access to current thread state and heap allocated objects
- However it also prevents Python programs from taking full advantage of multiprocessor systems
- Effectively, it is the language's most difficult technical challenge

Multiprocessing

The `multiprocessing` library

- Works on any system that runs Python
- Spawns processes that can execute tasks concurrently (each process has a separate GIL)
- Abstracts the detail of parallel scheduling and execution

- Pros
  - Straightforward to use
  - Can achieve good parallel speedups on trivially parallel problems

- Cons
  - Not suited well for tightly coupled parallelism (which needs a lot of inter-process communication)
  - Not designed for distributed computing

Multiprocessing

The `multiprocessing` library works by:

- Assigning tasks to processes
- Executing the processes

- Ideal use: trivial parallelism
  - Tasks assigned to processes are independent and
  - Task data is local to the process

- There is more...
  - The library offers a lot more (e.g. communication between processes)
  - However, tightly coupled parallelism is better tackled by other solutions, e.g. mpi4py which we will look at later

Trivial Parallelism

Trivially parallel workloads

- independent tasks that only need minimal communication or synchronisation
- ideally, tasks are balanced - all tasks have a fixed computational cost

- Other examples
  - Mapping a single function to multiple data
  - Parameter sweep
  - Map-reduce

Process Example

The simplest way to use the library is via the Process class:

A “Hello world” example:

- Initialise a number of processes
- Each processes is assigned a task
- Each process has an associated argument (passed on to the task function)
- Each process is started independently

Process Example

A “Hello world” example:

```
import multiprocessing

def func (num):
    print "hello from worker", num

if __name__ == "__main__":
    for i in range(4):
        p = multiprocessing.Process (target=func, args=(i,))
        p.start ()
```

```
hello from worker 0
hello from worker 1
hello from worker 3
hello from worker 2
```

Process Example

An enhanced “Hello world” example:

- Print process information: from OS as well as from library
- Number of processes equals the number of cores available (including hyper-threading)

Process Example

An enhanced “Hello world” example:

```
import multiprocessing
import os

def func (num):
    print "hello from worker", num, "process", os.getpid(), "name",
multiprocessing.current_process().name
    return

if __name__ == "__main__":
    for i in range(multiprocessing.cpu_count()):
        p = multiprocessing.Process (target=func, args=(i,))
        p.start ()
```

```
hello from worker 1 process 7037 name Process-22
hello from worker 3 process 7043 name Process-24
hello from worker 2 process 7038 name Process-23
hello from worker 0 process 7036 name Process-21
```

Multiprocessing classes

Two very important classes

- `Process` - represents activity that is run in a separate process
- `pool` - controls a pool of "worker" processes to which tasks are assigned

Other classes exist

- Examples include `Queue`, `Pipe`, `Lock`, `Manager`, `Connection`
- Allow quite sophisticated programming (including inter-process communication)
- Usefulness for scientific computing can be limited

The `pool` class

`pool`

- The most important (and useful class)
- Particularly useful for trivial parallelism

Caveats

- The pool class methods should only be used by the process which created the pool
- The __main__ module should be importable by the children processes
  - Some examples involving pool will not work in the interactive interpreter

The `pool` class: task offloading

Using the `pool` class hides the complexity of concurrency programming

- You do not have to worry about managing processes or shared data
- You do not have to worry about distributing the work between workers

All that is needed is to…

- Open a pool of processes
- Offload tasks to processes in the pool

There are two types of methods to offload tasks

- `map` (and `map_async`) - map a function to arguments from an iterable
- `apply` (and `apply_async`) - apply a function to a set of arguments

The `pool` class: `map`

`map` and `map_async`

- Map a function to arguments from an iterable
- Similar to Python built-in map
- The iterable is automatically divided into "chunks" of tasks that are scheduled to be executed by separate processes in the pool
- Each process executes more than one task, going through its "chunk" of work sequentially

Usage:

A single same function / task needs to be mapped to a set of multiple similar arguments

The `pool` class: `map`

Example: Unique function to multiple data – `map` version

- `map` is simpler than `apply`
- `data` is a list (iterable)

```
import multiprocessing

def myfunc (x):
    return x*x

if __name__ == '__main__':
    pool = multiprocessing.Pool(2)
    data = [1, 2, 3, 4, 5, 6, 7]
    print (pool.map (myfunc, data))
```

```
[1, 4, 9, 16, 25, 36, 49]
```

The `pool` class: `map`

Example: Unique function to multiple data – `apply` version

- `map` is simpler than `apply`
- `data` is a list (iterable)

```
import multiprocessing

def myfunc (x):
    return x*x

if __name__ == '__main__':
    pool = multiprocessing.Pool(4)
    data = [1, 2, 3, 4, 5, 6, 7]
    results = [ pool.apply (myfunc, args = (d, )) for d in data ]
    print results
```

```
[1, 4, 9, 16, 25, 36, 49]
```

The `pool` class: properties

Four choices

- `map` and `map_async`
- `apply` and `apply_async`

Three important properties

- Scheduling
  - `map` / `map_async` are mapped directly
  - `apply` / `apply_async` use a task queue
- Blocking
  - The asynchronous variants are non-blocking; processing carry on after their call, without waiting for returned results
- Ordering
  - The asynchronous variants are mapped / applied in arbitrary order to the "targets"

The `pool` class: more methods

`close()`

- Prevents any more tasks from being submitted to the pool
- Once all the tasks completed, the worker processes exit and resources freed

`join()`

- Wait for the worker processes to exit
- `close()` must be called before using `join()`

`close()` and `join()` are usually used together

- Typically called at end of the parallelisable part of the program
- Ensure the tasks are completed before results can be used

`get()`

- Return results from the objects returned by map or apply (or their asynchronous variants)
- It is blocking

The asynchronous methods accept callback functions

- These are useful to perform useful actions on the results
- Can be used instead of `get()` to retrieve results

Performance note…

Minimise quantity of data passed to processes

- map small number of function arguments, little data
- apply tasks with little input data

Spawned processes are fed data by the process that owns the pool

- Amount of data has to be minimal, otherwise performance suffers
- Data is **pickled** before it is fed to spawned processes

Pickling

- The process whereby a Python object hierarchy is converted into a byte stream, or "serialised"
- Used by multiprocessing to pass data around spawned processes

Mapping functions with multiple parameters

- Use tuples

multiprocessing

Summary

Pros

- Easy to use, particularly on trivially parallel problems
- Good speedups can be achieved with little work

Cons

- Limited to a single node

MPI4PY

What is MPI?

One Standard

- Message Passing Interface
- The standard defines the syntax and semantics of a core of library routines designed for writing portable message-passing programs (in C and Fortran)
- The *de facto* standard for distributed parallel programming

Many implementations

- Open source: e.g. OpenMPI, mvapich2, MPICH
- Commercial: e.g. IntelMPI

Many Interfaces

- C, Fortran (the MPI primary targets)
- Java, Python (late additions)

Learning MPI

This presentation aims to give you an appreciation of how MPI works and how it can be used in Python (not to teach MPI)

Many options for learning MPI

- Online material
  - **http://www.archer.ac.uk/training/online/**
- Traditional courses
  - Archer training

Our advice: familiarise yourself with MPI using C or Fortran

- Material is generally a lot more detailed than for Python documentation (for MPI this is terse)
- A lot more online tutorials and good textbooks than for Python
- Clearer learning path than for Python (MPI for Python has several interfaces)

Message passing paradigm

Distributed memory programming model

- Independent processes running concurrently
- All processes execute the same program
- Each process has its own memory address space
- Any data that needs to be shared between processes needs to be passed explicitly

Distributed computing and distributed memory

- The problem data is partitioned, each processes owns a partition and works on it
- Computation becomes possible because large memory requirements are distributed
- Computation becomes faster because large computational requirements are distributed

Message passing paradigm

Advantages

Universality & portability

- Works everywhere and matches all hardware
  - Separate processors connected by any network (e.g. modern clusters, ad hoc networks of computers)
  - Shared memory systems (e.g. your laptop)
- Source code is universally portable (with few exceptions, e.g. parallel I/O)
  - Between computer systems
  - Between implementations

Performance & scalability

- The most compelling reason why MPI remains a permanent component of HPC
- With future core count increase, distributed memory computing is the key to high performance

MPI for Python

Combining the advantages of both

- The MPI standard says nothing about Python
- Any MPI solution for Python mimics the C/C++ standard

Several options

- Old
  - pypar, pyMPI, Scientific Python
- Newer and better
  - Mpi4py

mpi4py

- An interface very similar to the MPI-2 standard C++ interface
- Focus is in translating MPI syntax and semantics (if you know MPI, mpi4py is "obvious")
- Can communicate memory-contiguous data (as C/Fortran) and can communicate Python objects
- Performance not the same as for C and Fortran but what is lost in performance is gained in code development time

MPI for Python

Observations

- Some appreciation of the object-oriented nature of Python programming is useful
- The C++ bindings in the MPI-2 standard can provide a useful quick reference for mpi4py
- The C++ function names can be slightly different from the corresponding C/Fortran bindings
  - In fact, often they are reversed, e.g. MPI_Buffer_attach() becomes MPI.Attach_buffer()
- The iPython information `help(mpi4py.MPI)` is rather long
  - Useful to narrow it down, e.g. `help(mpi4py.MPI.Intracomm)`
  - This requires that you know what you are looking for

MPI for Python

Example “Hello World”

```
import sys
from mpi4py import MPI

def report (communicator):
    """Report rank and size of this communicator"""
    rank = communicator.rank
    size = communicator.size
    sys.stdout.write ("Hello from rank {:2d} of {:2d}\n".format(rank, size))

if __name__ == "__main__":
    """Execute in MPI.COMM_WORLD"""
    report (MPI.COMM_WORLD)
```

```
$ mpirun -np 4 python helloworld.py
```

```
Hello from rank  2 of  4
Hello from rank  3 of  4
Hello from rank  1 of  4
Hello from rank  0 of  4
```

mpi4py basics

MPI Initialisation

- Import the mpi4py module

```
from mpi4py import MPI
```

- The import mpi4py statement is responsible for initialising MPI (if not already initialised)
  - There is no analogue of calls to `MPI_Init()` and `MPI_Finalize()` as used in C/Fortran
  - These calls are implicit from the underlying MPI library when importing the mpi4py module and when the Python process ends

The class MPI

- Loaded from module mpi4py
- Provides the pre-defined communicator COMM_WORLD

Size and rank

`size = MPI.COMM_WORLD.size` (or MPI.COMM_WORLD.Get_size()) is the number of the processes started through mpirun

`rank = MPI.COMM_WORLD.rank` (or MPI.COMM_WORLD.Get_rank()) is the ID of the process: `0 ≤ rank ≤ size-1`

`mpi4py` basics

The Python script is started through the MPI launcher `mpirun`

- The `mpirun` command is part of the MPI library implementation
- Used to launch any MPI code, e.g. code compiled from C/C++/Fortran source
- Two command line arguments
  - The number of processes to launch
  - (optional) the hosts on which to run these processes (default: local host)

The launcher starts and coordinates a number of processes

- All these processes run concurrently
- All processes execute the same Python code
- Programming differentiates between what processes do

Communicators

A communicator provides the context within which communication takes place

- Two or more processes can pass messages between each other only if they belong to the same communicator
- All communicators are Python objects
- The pre-defined communicator COMM_WORLD contains all processes started by `mpirun` and is accessed via the class MPI

MPI Communication

Message Passing

- The transaction by which one process accesses data from another
- All communication has one (or several) sender(s) and one (or several) receiver(s)

Pattern of communication

- Point-to-point message passing - communication between only two processes
- Collective message passing - communication between all processes
  - one-to-many
  - many-to-one
  - all-to-all

Blocking or non-blocking?

- Blocking messaging: sender and receiver wait till the receive/send transaction finishes
- Non-blocking messaging: immediate send, no waiting, need to probe if a message arrived

mpi4py Communication

mpi4py

- supports fast (near C-speed) communication of contiguous memory objects (typically, NumPy arrays)
- supports convenient communication of generic Python objects (pickling behind the scenes)

Communication of contiguous memory objects

- function names start with upper-case: `Send()`, `Recv()`, `Bcast()`, `Scatter()`, etc.
- arguments must be explicitly specified, e.g. [data, MPI.DOUBLE] or [data, count, MPI.DOUBLE]
- automatic MPI datatype discovery for NumPy arrays is supported, but limited to basic C types (all C/C99-native signed/unsigned integer types and single/double precision real/complex floating types)

Communication of generic Python objects

- Function names start with lower-case: `send()`, `recv()`, `bcast()`, `scatter()`, etc.
- Any object to be communicated is passed as a parameter to the function
- The received object is the return value of the function

Point-to-point communication: blocking Send() and Recv()

Send() and Recv()

- Communicate buffer-like (contiguous memory) objects
- `Comm.Send (self, buf, int dest, int tag=0)`
  - Returns only when the data in the send buffer can be safely changed
  - That does not mean the data arrived at destination
- `Comm.Recv (self, buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)`
  - Returns only when receive buffer contains data expected

Observations

- The interface makes significant use of optional arguments
- Unlike in the C/Fortran standard, count or data type arguments associated with the message buffer are optional as they can be inferred

Point-to-point communication: Example

```
from mpi4py import MPI
import numpy

def main (comm):
    """Send a message between ranks 0 and 1"""

    # buffer length
    blen = 4
    # process rank
    rank = comm.Get_rank()

    if rank == 0:
        # create send buffer
        buf = numpy.ones (blen, numpy.double)
        # send buffer
        comm.Send ( [buf, blen, MPI.DOUBLE], dest = 1, tag = 999 )
    elif rank == 1:
        # allocate space for recv buffer
        buf = numpy.empty (blen, numpy.double)
        # receive buffer
        comm.Recv ( [buf, blen, MPI.DOUBLE], source = 0, tag = 999 )
        print " rank 1 received: %s" % (buf)

if __name__ == "__main__":
    main (MPI.COMM_WORLD)
```

Point-to-point communication: Example

```
$ mpirun -np 2 python ./send_recv_blocking.py

rank 1 received: [ 1.  1.  1.  1.]
```

Point-to-point communication: non-blocking `Isend()` and `Irecv()`

What is the point?

Hiding latencies by overlapping communication with processing

- Start non-blocking communication using `Isend()` and `Irecv()`
- Carry out some processing (which does not depend on data communicated)
- Handle the requests generated by `Isend()` and `Irecv()` to ensure communication takes place
- Carry on with processing that does depend on data communicated

Point-to-point communication: non-blocking Example

```
"""Exchange messages with 'ajoining' ranks"""
    # right-hand adjoining rank (wraparound)
    right = comm.rank + 1
    if right >= comm.size: right = 0
    # left-hand adjoining rank (wraparound)
    left = comm.rank - 1
    if left < 0: left = comm.size - 1

    # send and recv buffers
    smsg = numpy.array ( [comm.rank], numpy.int )
    rmsg = numpy.zeros ( 2, numpy.int )

    # initiate communication
    reqs1 = comm.Isend (smsg, left)
    reqs2 = comm.Isend (smsg, right)
    reqr1 = comm.Irecv (rmsg[0:], source = left)
    reqr2 = comm.Irecv (rmsg[1:], source = right)

    # receive requests handled by Wait()
    reqr1.Wait()
    reqr2.Wait()

    # all processes print
    # NB: safe to print as messages were received already
    print "Rank%d smsg=%s rmsg1=%s rmsg0=%s" % (comm.rank, smsg, rmsg[1], rmsg[0])

    # send requests handled by Waitall()
    MPI.Request.Waitall ( [reqs1, reqs2] )
```

Point-to-point communication: non-blocking Example

```
$ mpirun -np 4 python isend_irecv_nonblocking.py

Rank2 smsg=[2] rmsg1=3 rmsg0=1
Rank1 smsg=[1] rmsg1=2 rmsg0=0
Rank0 smsg=[0] rmsg1=1 rmsg0=3
Rank3 smsg=[3] rmsg1=0 rmsg0=2
```

Point-to-point communication: Python Objects

Message passing of serialised (pickled) Python objects

```
Comm.send (self, obj, int dest, int tag=0)
Comm.recv (self, buf=None, int source=ANY_SOURCE, int
tag=ANY_TAG, Status status=None)
```

Observations

- serialisation / de-serialisation carry an overhead (memory & time)
- Careful: complex / large objects are slow to communicate

Note:

The incoming message is received as the return value

```
msg = comm.recv(...)
```

Point-to-point communication: Python Objects Example

```
def main (comm):
    """Send a list from rank 0 to rank 1"""

    if comm.size != 2:
        sys.stdout.write ("Only two processes allowed\n")
        comm.Abort(1)

    if comm.rank == 0:
        msg = ["Any", "old", "thing", comm.rank, {"size" : comm.size}]
        comm.send (msg, dest=1, tag = 999)
    elif comm.rank == 1:
        msg = comm.recv (source=0, tag = 999)
        print " rank 1 received: %s" % (msg)
```

```
$ mpirun -np 2 python send_recv_list.py

 rank 1 received: ['Any', 'old', 'thing', 0, {'size': 2}]
```

Collective Communication

Multiple processes within same communicator exchange messages

- Always blocking
- All processes in the communicator must call the collective
- Failing that, deadlocks occur

Most useful

- Broadcasts
- Scatter / Gather
- Reductions

Collective Communication

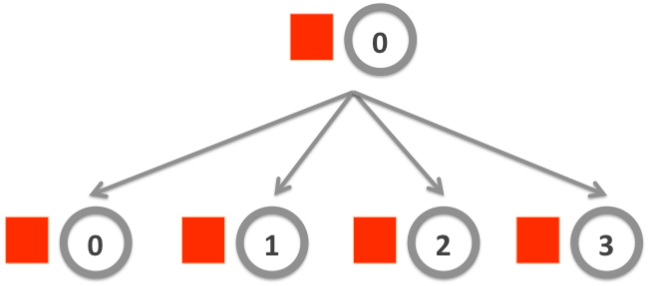
What is the point of collectives?

- specialised operations than can be replicated using point-to-point functions
- example: a broadcast from rank 0 can be written as:

```
if comm.rank == 0:
  for i in range (1, comm.size):
      comm.Send ([buf, sz, MPI.DOUBLE], 1, tag = 99)
else:
  comm.Recv ([buf, sz, MPI.DOUBLE], source = 0, tag = 99)
```

Collective Communication: Broadcast

One process sends the same message to all other processes in the communicator



Continuous buffer (e.g. numpy.ndarray)

```
comm.Bcast (self, buf, int root=0)
```

Python objects (pickling behind the scenes)

```
comm.bcast (self, buf, int root=0)
```

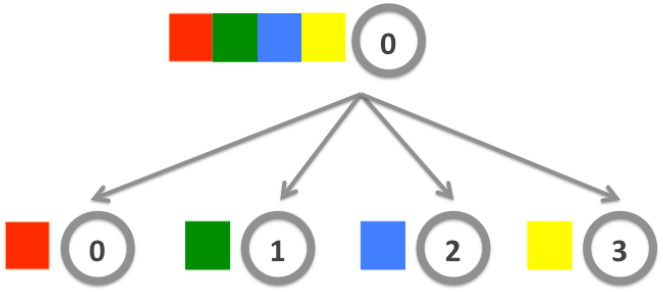
Collective Communication: Broadcast Example

```
def main (comm):
    if comm.rank == 0:
        # rank 0 has ndarray data
        u = numpy.arange (6, dtype=numpy.float64)
        # and dict data
        d = {"x": 1, "y": 3.14, "z": 1-2j}
    else:
        # all other have an empty array
        u = numpy.empty (6, dtype=numpy.float64)
        # and a place-holder
        d = None

    # broadcast from rank 0 to everybody
    comm.Bcast ( [u, MPI.DOUBLE] , root=0 )
    d = comm.bcast ( d, root=0 )

    # all processes print data
    print "[%d] %s %s" % (comm.rank, u, d["y"])
```

```
$ mpirun -np 4 python bcast.py

[2] [ 0.  1.  2.  3.  4.  5.] 3.14
[0] [ 0.  1.  2.  3.  4.  5.] 3.14
[1] [ 0.  1.  2.  3.  4.  5.] 3.14
[3] [ 0.  1.  2.  3.  4.  5.] 3.14
```

Collective Communication: Scatter

One process sends a different message to each other processes in the communicator



Continuous buffer (e.g. numpy.ndarray)

```
comm.Scatter (self, sendbuf, recvbuf, int root=0)
```

Python objects (pickling behind the scenes)

```
comm.scatter (self, buf, int root=0)
```

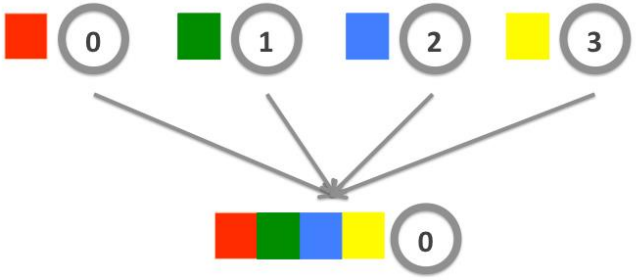
Collective Communication: Scatter Example

```
def main (comm):
    if comm.rank == 0:
        # rank 0 has ndarray data
        sendBuf = numpy.empty ([comm.size, 8], dtype=int)
        sendBuf.T[:,:] = range(comm.size)
        # and list data
        obj = [(i+1)**2 for i in range(comm.size)]
    else:
        # all other ranks have an empty send buffer
        sendBuf = None
        # and a place-holder
        obj = None

    # all ranks have a receiving buffer
    recvBuf = numpy.empty (8, dtype=int)

    # broadcast from rank 0 to everybody
    comm.Scatter (sendBuf, recvBuf, root=0)
    obj = comm.scatter ( obj, root=0 )

    # check: all processes verify data
    print "[%d] %s %s" % (comm.rank, recvBuf, obj)
```

```
$ mpirun -np 4 python scatter.py
[0] [0 0 0 0 0 0 0 0] 1
[1] [1 1 1 1 1 1 1 1] 4
[3] [3 3 3 3 3 3 3 3] 16
[2] [2 2 2 2 2 2 2 2] 9
```

Collective Communication: Gather

One process sends a different message to each other processes in the communicator



Continuous buffer (e.g. numpy.ndarray)

```
comm.Gather (self, sendbuf, recvbuf, int root=0)
```

Python objects (pickling behind the scenes)

```
comm.gather (self, buf, int root=0)
```

Collective Communication: Gather Example

```
def main (comm):
    # ndarray data to send
    sendBuf = numpy.zeros(8, dtype=int) + comm.rank
    # ndarray data to receive
    if comm.rank == 0:
        recvBuf = numpy.empty ([comm.size, 8], dtype=int)
    else:
        recvBuf = None
    # Python object
    obj = (comm.rank+1)**2

    # gather ndarray
    comm.Gather (sendBuf, recvBuf, root=0)
    # gather Python objects
    obj = comm.gather(obj, root=0)

    # check: all ranks print verify data
    print "[%d] %s %s" % (comm.rank, recvBuf, obj)
```

```
$ mpirun -np 4 python gather.py
[1] None None
[3] None None
[2] None None
[0] [[0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1]
 [2 2 2 2 2 2 2 2]
 [3 3 3 3 3 3 3 3]] [1, 4, 9, 16]
```

Collective Communication: Reduction

Data in all processes is "reduced" to a single process according to an operation

- Operations are defined by the `Op` class in mpi4py
  - Pre-defined operations: MIN, MAX, SUM, etc.
  - User defined operations can be programmed for

- Example: find the maximum value on all the data across all processes
  - Each process finds a "local" maximum across its own data
  - All local maxima are "reduced" to a single value

Continuous buffer (e.g. numpy.ndarray)

```
Comm.Reduce (self, sendbuf, recvbuf, Op op=SUM,
                    int root=0)
```

Python objects (pickling behind the scenes)

```
Comm.reduce (self, sendobj, Op=SUM, int root=0)
```

Collective Communication: Reduce Example

```
def main(comm):
    # ndarray data
    buf  = ( numpy.ones (6, dtype=int) + 1 ) * comm.rank
    buf2 = numpy.empty (6, dtype=int)

    # Python object data
    obj  = [ comm.rank ]

    # reduction on ndarray: buf reduced to buf2
    comm.Reduce ( [buf, MPI.INT], [buf2, MPI.INT], op=MPI.SUM, root=0 )

    # reduction of Python object
    obj  = comm.reduce ( [comm.rank], op=MPI.SUM, root=0 )

    # all processes print data
    if comm.rank == 0:
        print "[%d] %s %s <- reduced" % (comm.rank, buf2, obj)
    else:
        print "[%d] %s %s" % (comm.rank, buf, obj)
```

```
$ mpirun -np 4 python reduce.py

[3] [6 6 6 6 6 6] None
[2] [4 4 4 4 4 4] None
[1] [2 2 2 2 2 2] None
[0] [12 12 12 12 12 12] [0, 1, 2, 3] <- reduced
```

mpi4py Summary

We have briefly looked at

- The message passing paradigm and the MPI standard
- The mpi4py module in Python

mpi4py for production code?

- Yes, if
  - Communication is not very frequent
  - Communication does not involve a lot of data
  - Performance is not the primary concern
- No, if
  - The algorithm requires a lot of communication
  - The plan is to scale to large core counts

A good idea (?)

- mpi4py may be a good tool for teaching the basic concepts of distributed (MPI) computing

Measuring Performance

The most important measure

- In this section we will introduce a few measures of performance for computer codes, both serial and parallel
- But never forget that what you ultimately want to maximise is the amount of *science per second* that you generate
- This may be as simple as minimising the run time of your program
- But it may involve other factors
  - Using a more familiar application
  - Getting the best out of your computer budget
  - Turnaround on the cluster
    - **Higher core counts tend to turn around more slowly**

Parallel measures of performance

- Measuring the performance of parallel codes generally asks questions related to how much better are things running on multiple cores when compared to running on a single core or node
- We'll look at
  - Speed Up
  - Cost

Speed Up

- Speed up answers the question “How much faster does my program run if I use P cores”

$S(P)=T_1/T(P)$

- So if I use 100 processors it will run 100 times faster, right?
- And it can’t run more than 100 times faster, right?
- Also, I have 100 times as much memory so I can run 100 times bigger a problem, right?

Absolute Speed Up

- What we should really measure is Absolute Speed Up

$S(P)=T_s/T(P)$

- Where T_s is the time to run the best implementation of the serial program, and T(P) is the time to run the parallel code on P cores
  - You may use a different algorithm in the parallel code from the serial code

Relative Speed Up

- However what is almost always measured is Relative Speed Up

$S(P)=T(1)/T(P)$

- i.e. we compare the speed against the time taken on 1 core by the parallel program
- Saves writing both the serial and parallel code

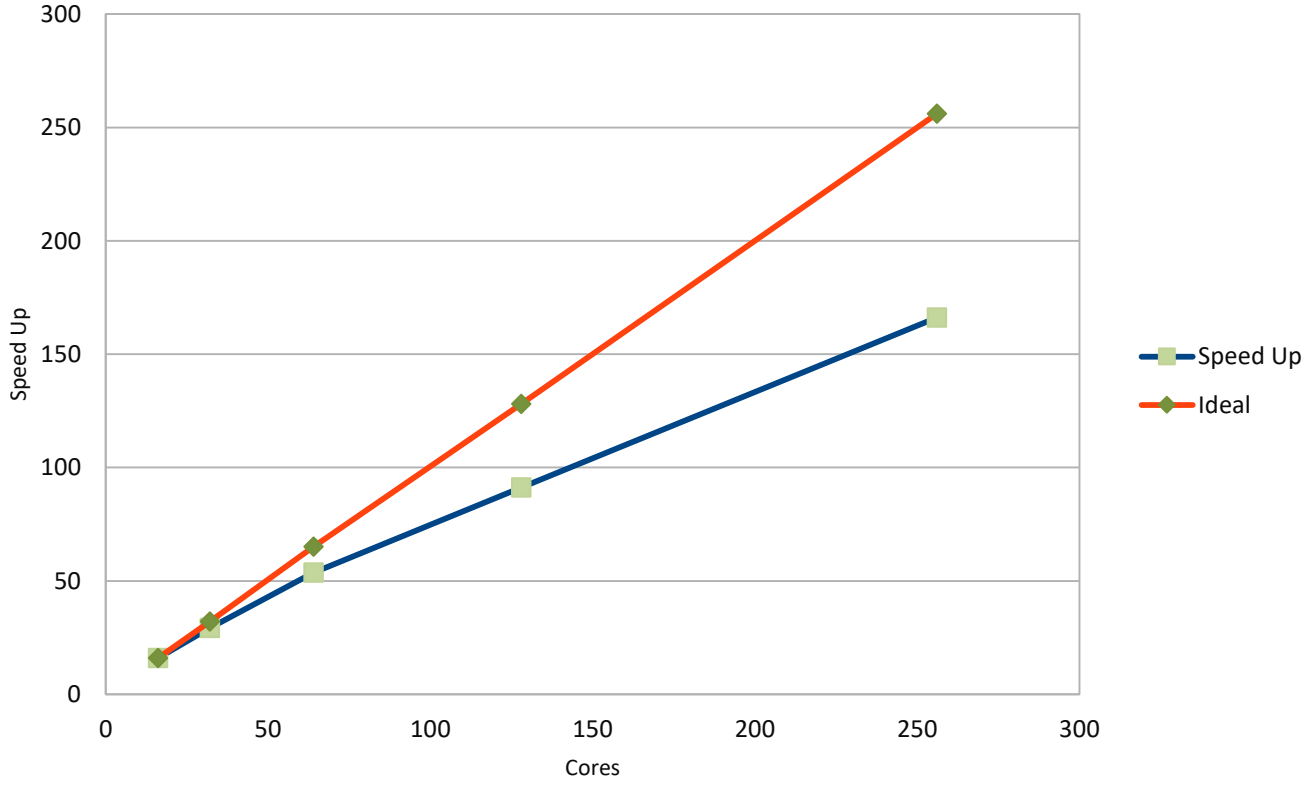
Linear Speed Up

- Linear speed up is simply

$S(P)=P$

- So if you run on P processors, it runs P times quicker
- This is the ideal situation
- Also called perfect scaling

What Does it Look Like



So Why is it Not Perfect!?

- Many Possible Reasons!
- We've touched on load balance
- To go beyond this again is beyond what we are trying to cover here
- The main thing is NOT to expect perfect speed up
- And to be aware that using too many cores can actually DEGRADE your performance

Cost

- On some computers you will be CHARGED(!!) for use
- The usual situation on national resources
- Typically you get a finite budget which gets deducted from every time you use the machine
- Quite how the charging works will vary from machine to machine. Typically you get charged a fixed rate per node that you use, and for every second you use
  - Cost=nodes*time
  - This is the model on ARC machines
- i.e. If you leave cores idle in a node you get charged for them

So How Many Cores Should I Use

- Well, firstly don't put more processes or threads on a node than there are cores!!
- It depends …
- It depends on the code you are running
  - It may have special parallel options which you should learn about
- It depends on the case you are running
- It depends on the computer you are running on
- More cores will cost you more
- More cores may even slow you calculation down
- More cores will probably mean slower turn around
- It depends upon you and how important cost and turnaround are
- BUT DON'T JUST GUESS!
- One thing you can do is to run a little experiment

An Experiment

- Many Scientific codes do the same kind of thing many times
  - E.g. time steps
  - E.g. iterations in a solver
- So plot the speed up curve for a few iterations and from that decide on a good number of cores for you
- For instance a simulation may require at the very least many thousands of time steps
- So first run it for 100 time steps on 1, 2, 4, 8, 16 … cores and see what the speed up curve looks like
- And then use that number of cores for the full run

Conclusion

Summary

We have examined

- The sources of Python slow performance for scientific computation
- NumPy
  - Array types that are suited for scientific codes
  - The basis for other Python modules
- Single host parallel execution
  - Serial: NumPy, numba,
  - Parallel: Cython, multiprocessing
- Distributed parallel execution
  - mpi4py

Summary

What to use? Assuming greenfield programming

- NumPy should be the first port of call
  - Vectorised operations
  - ufuncs and specialised class methods
  - The basis of further approaches

- numba and Cython are very easy
  - Should be tried next
  - Large gains for almost no effort

- Multiprocessing is relatively easy
  - Optimal for mapping functions to data in parallel
  - Best case: expensive functions and little data to map to
  - Limited to single host execution

- mpi4py is relatively difficult
  - May pay off if multi-host execution dominates
  - Inter-process communication should be avoided

But... First profile, then re-program

Reuse of this material