Requirements for running the Jupyter notebook

conda create -n pyladiesMarch python=2.7

conda activate pyladiesMarch

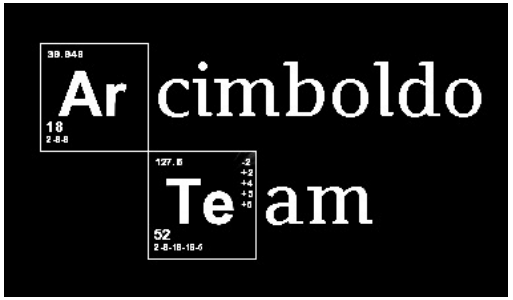
conda install nb_conda

python -m ipykernel install --user --name pyladiesMarch --display-name "Python2.7 pyladiesMarch "

Multiprocessing versus threading in a single workstation in Python 2.7.x

Barcelona, March 2019

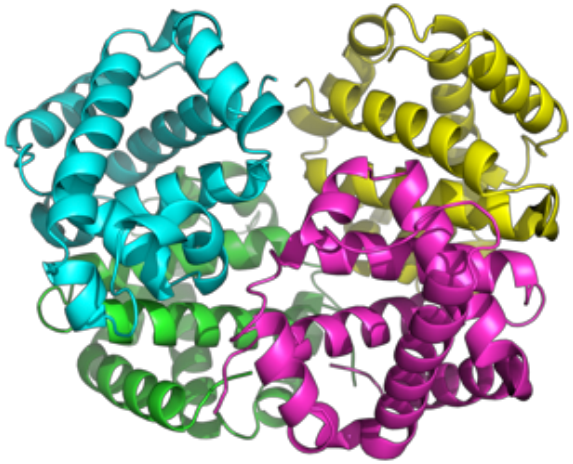
@cheshireminima
#ArcimboldoTeam
@PyLadiesBCN



Claudia Millán

cmncri@ibmb.csic.es

- Five years degree in **Biotechnology**
- Master's degree in **Crystallography** and Crystallisation
- For the last 6 years, developing software in the **Crystallographic Methods group** of Dr. Isabel Usón
- Since September 2018, Dr Claudia Millán!
- Self-taught (with some great help ;)) **Python** programmer, with some basic R and Fortran knowledge, and notions of C and Perl from the degree.



Why parallelization on single workstations?

- All modern computers have an underlying parallel architecture, and for a reasonably "cheap" price you get a "large" number of CPUs

- If your code is parallelizable and your choice of parallelization strategy is correct, you will get a considerable speed increment in your running time

Why Python?

Python has some features that affect its behaviour on parallel programming that we will discuss. BUT:

- Is a high level, interpreted, dynamic programming language, with a design philosophy that emphasizes code readability. There are many scientific programmers who are using it and developing libraries and extensions for a huge number of applications.
- CPU-demanding computations can be made faster by boosting python with C, C++ or Fortran code, and input/output overhead is there anyway.

Why Python 2.7.x?

- In my field, the move towards Python 3 is still not yet there, though many of the large libraries are being ported now

- In my lab, we will be doing the move in the next months together with a major rewriting of some of the programs, so any suggestions are welcomed

Parallel programming basics

- You want to:
  - Solve a particular problem with parallel computing
  - Transform your sequential program to a parallel, potentially faster version
- You need to:
  - Understand which kind of problem you are facing in detail
  - Identify the bottlenecks in your code

Parallel programming basics: step by step

1. Find concurrency
   - Task and data decomposition
   - Dependency analysis
2. Structure the algorithm so that concurrency can be exploited
3. Implement the algorithm in a programming environment
4. Execute and tune the performance of the code on a parallel system

Parallel programming basics: design patterns

- A software design pattern is **a general reusable solution** to a commonly occurring problem within a given context.

- Is not something that you can directly transform to source code, but rather a **template for how to solve a problem**.

Parallel programming basics: design patterns

- By tasks → You need to perform the same tasks on different sets of data
  - Task parallelism
  - Divide and conquer
- By data decomposition → You need to process large data that can be decomposed
  - Geometric decomposition
  - Recursive data
- By flow of data → You have forking steps in which each the sequential tasks on data can be performed in parallel
  - Pipeline
  - Event-based coordination

Design pattern: Task parallelism

- The problem is best decomposed into a **collection of tasks** that can execute concurrently

- Examples:
  - Ray-tracing (completely independent tasks)
  - Molecular dynamics (some dependencies)

- To consider in solution:
  - The tasks (how many, and how much they compute)
  - The dependencies among the tasks
  - The schedule (static or dynamic depending on your computing system)

Design pattern: Divide and conquer

- The problem can be formulated to be **solved in sub-problems** independently, and **merge** their solutions later on

- Examples:
  - Fast Fourier Transform
  - Merge-sort

- To consider in solution:
  - At very deep level of recursion might happen that the computation required is so small that is better to change to a sequential approach

Design pattern: Geometric decomposition

- The problem has **dependencies**, but they communicate in a predictable (geometric) **neighbour-to-neighbour path**

- Examples:
  - Matrix multiplication
  - One dimensional heat diffusion

- To consider in solution:
  - How to partition the global data into chunks
  - Give access for the tasks to all the data they need
  - Perform the computation on that chunk
  - How to map the chunks to execution units in order to achieve good performance

Design pattern: Pipeline

- The problem requires a **series of ordered but independent computation stages** to be applied on data, where **each output of a computation becomes input** of subsequent communication

- Examples:
  - Processing a sequence of images
  - Shell programs in UNIX

- To consider in solution:
  - Defining the stages of the pipeline
  - Structuring the computation
  - Representing the dataflow among pipeline elements
  - Handling errors
  - Processor allocation and task scheduling
  - Throughput and latency

Python Threads

Threads

- A thread is an independent portion of the program running inside the main process, and sharing resources with it.

- Multiple threads can exist in a single process. The threads that belong to the same process share memory (can read and write to the very same variables, and can interfere with one another)

- In Python, threads are managed using the **threading module**

Python Threads: Definition

- Definition by a class statement

```
import time
import threading

class CountdownThread(threading.Thread):

    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count

    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

Inherits from the threading.Thread class

You override the __init__ method in order to include the count attribute, and the run method

The code inside run is what the thread will execute

Python Threads: Definition

- start() will launch the thread

```
t1 = CountdownThread(10) # Create the thread object
t1.start() # Launch the thread

t2 = CountdownThread(20) # Create another thread
t2.start() # Launch
```

- A different way to define a thread

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
```

The function given as target argument will be used as run() method of the defined thread

Python Threads: Join a thread

join([timeout]) will block the calling thread until the thread whose join() method is called terminates or until and optional timeout occurs

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

import time
import threading

def printer():
    for _ in range(3):
        time.sleep(1.0)
        print "Hello!"

thread=threading.Thread(target=printer)
thread.start()
#thread.join()
print "goodbye"
```

What effect will have commenting or not the join statement?

If we comment it, we say goodbye even before saying hello! ;)

In a real world example, you might want to wait a thread to finish a task before continuing with a sequential part of your code

Python Threads: Daemonic Threads

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

import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,format='(%(threadName)-9s) %(message)s',)

def n():
    logging.debug('Starting')
    logging.debug('Exiting')

def d():
    logging.debug('Starting')
    time.sleep(5)
    # time.sleep(3)
    logging.debug('Exiting')

if __name__ == '__main__':

    t = threading.Thread(name='non-daemon', target=n)

    d = threading.Thread(name='daemon', target=d)
    d.setDaemon(True)

    d.start()
    t.start()

    #d.join()
    #t.join()

    #d.join(3.0)
    d.join(7.0)
    print 'd.isAlive()', d.isAlive()
    t.join()
```

Daemon threads are threads that will be killed automatically when your program dies. Or putting it the other way: a program terminates when all its non-daemonic threads are finished

Explicit declaration as daemon

They are used commonly used to perform background tasks: checking load average, maintaining connections, etc. Things that are performed iteratively and that will not be needed after the program ends

Python Threads: Communication

- Thread scheduling is non-deterministic, therefore, access to any shared data is also non-deterministic! So you do not want two threads modifying shared data at the same time, producing an unpredictable result and generating racing conditions in your program. The solution to that problem is **thread synchronization**

- The threading module defines a number of objects for thread synchronization:
  - Lock
  - Rlock
  - Semaphore
  - BoundedSemaphore
  - Event
  - Condition

Python Threads: Lock (mutex lock)

```
# Start logging
logging.basicConfig(level=logging.DEBUG,format='(%(threadName)-9s) %(message)s',)

class BestSolution(object):
    def __init__(self):
        self.cc = 0
    def setCCvalue(self,cc_value):
        self.cc = cc_value

def worker(best,solution):
    logging.debug('Launching shelxe job ........ ')
    time.sleep(10)
    solution_file=open(solution,'r')
    solution_content=solution_file.read()
    cc_from_solution=extract_best_CC_shelxe(solution_content)
    logging.debug('CC read from solution '+str(cc_from_solution))
    lock.acquire()
    logging.debug('Acquire the lock')
    if cc_from_solution>best.cc:
        logging.debug('New best CC found, saving it to BestSolution '+str(cc_from_solution))
        best.setCCvalue(cc_from_solution)
    lock.release()
    logging.debug('Release the lock')

if __name__ == '__main__':
    best = BestSolution()
    lock=threading.Lock()
    list_files=os.listdir(os.getcwd())
    for fich in list_files:
        if fich.endswith('.lst'):
            solution=fich
            t = threading.Thread(target=worker, args=(best,solution,))
            t.start()

    logging.debug('Waiting for worker threads....')
    main_thread = threading.currentThread()
    for t in threading.enumerate():
        if t is not main_thread:
            t.join()
    logging.debug('Maximum final CC value found is '+str(best.cc))
```

- Only one thread can acquire it at a time
- Can not be acquired more than once
- Can be released by any thread

This returns a list of all the threads

Python Threads: Rlock (Reentrant Mutex Lock)

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

import threading
import random
import time

class ABManager:
    def __init__(self):
        self.a = 1
        self.b = 2
        self.lock = threading.RLock()

    def changeA(self):
        print "Lock acquired"
        with self.lock:
            print "Thread ",(threading.currentThread()).name," is changing A"
            self.a = self.a + 1
        print "Lock released"
```

The use of a with statement will ensure the lock is released

```
with some_lock:
    # do something...
```

is equivalent to:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

```
def

if __name__ == '__main__':
    manager = ABManager()
    for i in range(10):
        t = threading.Thread(target=worker,args=(manager,))
        t.start()
    main_thread = threading.currentThread()
    for t in threading.enumerate():
        if t is not main_thread:
            t.join()
```

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

import threading
import random
import time

class ABManager:
    def __init__(self):
        self.a = 1
        self.b = 2
        self.lock = threading.RLock()

    def changeA(self):
        print "Lock acquired"
        with self.lock:
            print "Thread ",(threading.currentThread()).name," is changing A"
            self.a = self.a + 1
        print "Lock released"

    def changeB(self):
        print "Lock acquired"
        with self.lock:
            print "Thread ",(threading.currentThread()).name," is changing B"
            self.b = self.b + self.a
        print "Lock released"

    def changeAandB(self):
        print "Lock acquired"
        with self.lock:
            print "Thread ",(threading.currentThread()).name," is changing A and B"
            self.changeA()
            self.changeB()
        print "Lock released"

def worker(managerAB):
    r = random.random()
    time.sleep(r)
    if 0.5 < r < 1.0:
        managerAB.changeAandB()
    elif 0.3 < r < 0.5:
        managerAB.changeA()
    else:
        managerAB.changeB()

if __name__ == '__main__':
    manager = ABManager()
    for i in range(10):
        t = threading.Thread(target=worker,args=(manager,))
        t.start()
    main_thread = threading.currentThread()
    for t in threading.enumerate():
        if t is not main_thread:
            t.join()
```

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

import threading
import random
import time

class ABManager:
    def __init__(self):
        self.a = 1
        self.b = 2
        self.lock = threading.Lock()

    def changeA(self):
        if not self.lock.locked():
            print "Lock acquired"
            self.lock.acquire()
        print "Thread ",(threading.currentThread()).name," is changing A"
        self.a = self.a + 1
        if self.lock.locked():
            self.lock.release()
            print "Lock released"

    def changeB(self):
        if not self.lock.locked():
            print "Lock acquired"
            self.lock.acquire()
        print "Thread ",(threading.currentThread()).name," is changing B"
        self.b = self.b + self.a
        if self.lock.locked():
            self.lock.release()
            print "Lock released"

    def changeAandB(self):
        if not self.lock.locked():
            print "Lock acquired"
            self.lock.acquire()
        print "Thread ",(threading.currentThread()).name," is changing A and B"
        self.changeA()
        self.changeB()
        if self.lock.locked():
            self.lock.release()
            print "Lock released"

def worker(managerAB):
    r = random.random()
    time.sleep(r)
    if 0.5 < r < 1.0:
        managerAB.changeAandB()
    elif 0.3 < r < 0.5:
        managerAB.changeA()
    else:
        managerAB.changeB()

if __name__ == '__main__':
    manager = ABManager()
    for i in range(10):
        t = threading.Thread(target=worker,args=(manager,))
        t.start()
    main_thread = threading.currentThread()
    for t in threading.enumerate():
        if t is not main_thread:
            t.join()
```

Python Threads: Other synchronization objects

- Semaphores:
  - Lock that can be acquired and released by any thread, but only a n limited number of times
  - A semaphore manages an internal counter *n* which is decremented by each acquire() call and incremented by each release() call. There will not be more than *n* threads at the same time.
  - Semaphores are often used to guard resources with limited capacity (e.g. open network connections)

- Event:
  - synchronize two or more threads' operations
  - An Event manages an internal flag that the callers can either set() or clear(). Other threads can wait() (blocked) for the flag to be set()

- Condition
  - Not only synchronize but signalling.
  - You can notify collectively to threads that a certain condition is satisfied (e.g. when a list with items to process is ready)

Python Threads: Killing a thread

No official API for doing that, because is not generally safe and you should not need it, but can be achieved in a number of different ways

Python Queues:
Queue module

- Queues allow data sharing between threads
- You can get() data from or put() data in a queue
- Queues have a join() and a task_done() method

```
def process_hkl_array(array):
    print "Sorting the ",len(array),"reflections"
    sorted_array=sort_reflections_phs(array)
    print "Doing some more useful stuff with my sorted array"
    time.sleep(5)

def worker():
    while True:
        (name_array,array_to_process) = q.get()
        print "\nThread ",threading.current_thread().name, "is processing ",name_array
        process_hkl_array(array_to_process)
        q.task_done()


if __name__ == '__main__':
    q = Queue.Queue()
    for i in range(3):
        t = threading.Thread(target=worker)
        t.daemon = True
        t.start()
    for fich in os.listdir(os.getcwd()):
        if fich.endswith('.hkl'):
            array_hkl=read_hkl_file(fich)# Read the hkl files and put them into the queue
            q.put((fich,array_hkl))
    q.join()
```

When the queue is finished, it will return and join in the main thread, therefore killing all daemons

The Global Interpreter Lock (GIL)

- Is a feature from the python interpreter that **prevents that two threads run python code at the same time**

- Why? Long history here. Short: Python is optimized for single-threaded execution.

- Whenever a thread runs it will be holding the GIL, and it will release it on input/output blocking operations. For that reason, CPU-bound threads do have low performance on python, but **threads can still be convenient for I/O bound programs**

Python Processes

Python Processes: multiprocessing module

- In contrast to threads, different processes live in different memory areas, and each of them has its own variables (and runs in multiple independent copies of the Python interpreter). In order to communicate, processes have to use other channels (files, pipes or sockets)

- In Python, processes are managed with the **multiprocessing module**. At the high level, this module is mirroring the threading interface, just you now work with Process objects instead of Thread objects

Python Processes: Definition and launching

```
class CountdownProcess(multiprocessing.Process):

    def __init__(self,count):
        multiprocessing. Process.__init__(self)
        self.count = count

    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

In windows, the subprocesses will import the main module at start, so be sure to put this statement

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

if __name__ == '__main__':
    p1 = multiprocessing.Process(target=countdown,args=(10,))
    p1.start()
```

Wait! multiprocessing or subprocess?

60

The `subprocess` module lets you run and control other programs. Anything you can start with the command line on the computer, can be run and controlled with this module. Use this to integrate external programs into your Python code.

The `multiprocessing` module lets you divide tasks written in python over multiple processes to help improve performance. It provides an API very similar to the `threading` module; it provides methods to share data across the processes it creates, and makes the task of managing multiple processes to run Python code (much) easier. In other words, `multiprocessing` lets you take advantage of multiple processes to get your tasks done faster by executing code in parallel.

share edit

edited Nov 28 '12 at 14:15

glglgl

64.7k • 7 • 88 • 163

Python Processes: subprocess pipes

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

import subprocess

p1 = subprocess.Popen(["ls","-1"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
p2 = subprocess.Popen(["grep", "example"], stdin=p1.stdout, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
p3 = subprocess.Popen(["wc", "-l"], stdin=p2.stdout, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out, err = p3.communicate()
print "Found ",int(out)," code examples"
```

Python Processes: multiprocessing pipes

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import multiprocessing

def consumer(p1, p2):
        p1.close() # Close producer's end (not used)
        while True:
                try:
                        item = p2.recv()
                except EOFError:
                        break
                print "Consuming ",item # Do something with item

def producer(sequence, output_p):
        for item in sequence:
                print "Sending ",item
                output_p.send(item)



if __name__ == '__main__':

        p1, p2 = multiprocessing.Pipe()

        cons = multiprocessing.Process(target=consumer,args=(p1,p2))
        cons.start()

        p2.close() # Close the input end in the producer

        # Produce some data
        sequence = xrange(1000)
        producer(sequence, p1)

        p1.close() # Close the pipe
```

Python Processes: queues in the multiprocessing module

- The interface is the same as for the Queue module but they are not the same.
  - They have a maximum size limit (32767)
  - They are implemented using pipes
- Three types of queues, and only one has task_done() and join() methods (JoinableQueue)
- General advice: instead of one big queue, you can use more queues or at least more data accumulated before actually sending it to the queue in order to avoid that it slows down

Python Processes: multiprocessing module

- Other features of multiprocessing
  - Process Pools
    - Groups of worker processes that can be defined at once and be given different arguments
  - Shared ctypes objects and arrays
    - Shared objects using shared memory that can be inherited by child processes
  - Synchronization primitives
    - Lock, Rlock, Event, Condition, etc…
  - Connections
    - Sending and receiving of pickable objects or strings

Process pools: quite straightforward to implement, very useful for embarrassingly parallel where only input changes

```
#print 'SHERLOCK psutil.cpu_count(logical=False)',psutil.cpu_count(logical=False)
from multiprocessing import Pool

# start your parallel workers at the beginning of your script
nucpu = (psutil.cpu_count(logical=False)-1)
pool = Pool(nucpu)
print '\n\n Opening the pool with ',nucpu,' workers'

# prepare the iterable with the arguments
list_args = []
for op,tuplels in enumerate(list_ls_to_process):
    namels = tuplels[0]
    phs_in_ls = tuplels[1]
    phs_ref = tuplels[2]
    list_args.append((namels[:-3],"./CLUSTERING/",path_phstat,resolution,0,100,1,orisub,weight))

# execute a computation(s) in parallel
pool.map(al.call_phstat_print_for_clustering_parallel, list_args)

# turn off your parallel workers at the end of your script
print 'Closing the pool'
pool.close()
```

```
def call_phstat_for_clustering_in_parallel_pool(args):
    name_phstat, wd, path_phstat, resolution, seed, tolerance, n_cycles,orisub, weight = args
    # Check if the correct files are there
    ls_filename = name_phstat + ".ls"
    pda_filename = name_phstat + ".pda"
    if not (((os.path.exists(os.path.join(wd, ls_filename))) and (os.path.exists(os.path.join(wd, pda_filename))))):
        sys.exit("\nAn error has occurred. Please make sure that you have provide a .ls and a .pda file")
    # If everything is OK, we continue and run phstat with the given command line
    command_line = []
    command_line.append(path_phstat)
    # NOTE change name phstat to get it to its shortest version so that FORTRAN will accept it
    name_phstat = os.path.split(name_phstat)[1]
    path_name_phstat = os.path.join(wd, name_phstat)
    file_out = open(path_name_phstat+'.out','w')
    command_line.append(path_name_phstat)
    arguments = ["-r" + str(resolution), "-s" + str(seed), "-t" + str(tolerance), "-c" + str(n_cycles)]
    if weight=='e':
        arguments.append('-e')
    if orisub=='sxosfft':
        arguments.append('-o')
    for i in range(len(arguments)):
        command_line.append(arguments[i])
    try:
        print "Command line used in phstat", command_line
        p = subprocess.Popen(command_line, stdin=subprocess.PIPE, stdout=file_out, stderr=subprocess.PIPE)
        complete_output, errors = p.communicate()
        # Move the file to change its name and make it consistent with the convention (avoid putting _0 or the nseed)
        os.rename(path_name_phstat + '_'+str(seed)+'.phi', path_name_phstat + ".phi")
        return complete_output,errors
    except Exception:
        exctype, value = sys.exc_info()[:2]
        print "\n An error has occurred:\n" + str(exctype) + "\n" + str(value)
        print "CHECK YOUR PHSTAT PATH IN YOUR BOR FILE!!"
        return None, None
```

Multiprocessing vs Threading

Threading Pros	Multiprocessing Pros
• Low memory footprint • Shared memory (easier access) • Makes easier to do responsive user interfaces • Extensions modules in C can release the GIL of the cPython interpreter • Good for Input / Output bound code	• Not shared memory • Code is more straightforward • Takes advantage of multiple CPUs • Eliminates most needs of synchronization primitives • Child processes are killable • Good for CPU-bound code
Threading Cons	**Multiprocessing Cons**
• They are subjected to the GIL • Can't be interrupted / killable • If you don't use queues, you need to use synchronization primitives, which are tricky! • Code might be harder to understand	• Interprocess communication is a bit more complicated and with more overhead • Larger memory footprint

Useful Resources and references

- The standard Python documentation
- Tutorials at http://www.bogotobogo.com/
- Questions at Stack Overflow

Thanks to

pyBCN BARCELONA PYTHON MEETUP

pyladies

R Ladies

Open calls for PhD and postdoctoral positions at our lab

You can contact/follow me on:
E-mail: cmncri@ibmb.csic.es
Twitter: @cheshire_minima
Github: clacri

Our web: http://chango.ibmb.csic.es/