AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 2
PARALLEL PROGRAMMING USING PYTHON TOOLS

BY FRED ANNEXSTEIN

2. USING PYTHON TOOLS

In this chapter we will consider some popular software tools to support parallel computing. In particular we focus on the use of the Python programming language for expressing concurrency. Many students know python, and in this chapter we examine techniques for converting a typical python program, which uses a model of serial processing, to run on a parallel platform.

CHAPTER 2 OBJECTIVES

The goal of this chapter is to expose students to software tools to easily express parallel programs. The chapter objectives are the following:

- Understand basic Python syntax for expressing parallel programs.
- Understand the opportunities and limitation of using the Python programming language for parallel computing.
- Understand how Python can support some aspects of parallel execution.
- Apply the parallel software design process in the Python language environment.

•Understand how Python supports both multithreaded and multiprogramming approaches.

•Understand the motivation and syntax for the Python Numba just-in-time compilation tool.

•Apply these tools in the construction of a computationally intensive image application.

REVIEW OF PARALLEL PROGRAM DESIGN

To parallelize any given serial program there are questions that need to be answered. Recall from Chapter 1 that to design parallel software we need to accomplish the following:

- Task Decomposition : the existing functions and data must be decomposed into distinct multiple units.
- Task Agglomeration: sub-tasks and data shards of the decomposition must be grouped together to improve the locality and granularity for high performance.
- Task Assignment and Mapping: agglomerated sub-tasks must be mapped to processors to facilitate load balancing and minimize communication costs. For high performance we must

also assess the need for dynamic mapping and load balancing as the computation progresses.

Python code can be used to express concurrency using either multiple threads or multiple processes. With a multithreaded approach we are limited to the processing power of one single CPU core. With a multiprocessing approach it is possible to utilize the full number of CPU cores available.

PYTHON THREADS

Python programs may use multiple threads of execution. The standard implementation of Python uses kernel-level threads which apply OS-specific system calls to create new threads and join together existing threads. However, Python doesn't have full control over when threads run, nor control of which CPU cores they will be mapped and executed on. The OS can preempt the currently running thread and give control to the other one at any point in time, for instance to run a thread with a higher priority.

Python has a well known limitation on its use of threads in its standard implementation. When the Python interpreter accesses objects, these accesses are

serialized by one global lock. This is done because many of the interpreter's internal structures are not thread-safe and need to be protected for correct execution.

However, not every operation requires locking, and there are certain situations when threads may release this lock. This mechanism of the standard interpreter is known as the Global Interpreter Lock (GIL). For I/O intensive applications releasing the GIL for multi-threading can lead to high performance.

The removal of the GIL to enable parallel processing is a topic that occasionally appears on the Python-dev email lists and has been considered, but never fully implemented by the Python development team.

PYTHON MULTIPROCESSING

An alternative approach that allows you to achieve parallelism is Python’s multiprocessing library. This library enables separate Python processes that do not constrain each other, since each process has a separate GIL. This style can allow for better resource utilization and is especially important for applications running on

multicore processors that are performing CPU-intensive tasks.

The other advantage of using multiple processes over multiple threads is the fact that they do not share a memory context. Thus, it is harder to corrupt data and easier to avoid deadlocks and race conditions. Not sharing the memory context means that you need some additional effort to pass the data between separate processes, but fortunately there are many good ways to implement reliable inter-process communication. Python provides some primitives that make communication between processes almost as easy as it is between threads.

With Python's multiprocessing module, we can effectively utilize the full number of cores and CPUs, which can help us to achieve greater performance when it comes to CPU-bounded problems. We can see the number of cores on an instance running on OSC as follows.

```
>>> import multiprocessing
>>> multiprocessing.cpu_count()
28
```

Not only does multiprocessing enable us to utilize more cores on our machine, but we also avoid the limitations that the Global Interpreter Lock imposes on us.

One significant, potential disadvantage of multiple processes is that we inherently have no shared state, and lack direct communication. If and when processes need to communicate they must pass it through some form of interprocess communication mechanism, and performance can take a hit. However, this lack of shared state can make them easier to work with, as you do not have concern for race conditions due to shared variables in the code.

```
import multiprocessing

print("# of CPUs ", multiprocessing.cpu_count())

def print_func(continent="North Pole"):
    print("The name of the continent is: ", continent)

if __name__ == "__main__":
    names = ["North America", 'South America', 'Europe',
'Africa', 'Asia', 'Antartica']
    procs = []
    proc = multiprocessing.Process(target=print_func())
    procs.append(proc)
    proc.start()

    for name in names:
        print('starting', name)
        proc = multiprocessing.Process(target=print_func,
args=(name,))
        procs.append(proc)
        proc.start()

        # complete the processes
    for proc in procs:
        proc.join()
```

Here is an execution of the multiprocessing program from above.

```
bash-4.2$ ipython multi.py
# of CPUs  4
The name of the continent is:  North Pole
```

```
starting North America
starting South America
starting Europe
starting Africa
starting Asia
starting Antartica
# of CPUs  4
# of CPUs  4
The name of the continent is:  North America
# of CPUs  4
The name of the continent is:  South America
# of CPUs  4
The name of the continent is:  Europe
# of CPUs  4
The name of the continent is:  Africa
# of CPUs  4
The name of the continent is:  Antartica
# of CPUs  4
The name of the continent is:  Asia
```

When each process starts, a new version of the python interpreter is created. The conditional gate given by `if __name__ == "__main__"` will guarantee that only one process executes the main block. That main block will start a process for the print_func for each name in the list names. The main process will then join all the processes to end the program.

PYTHON MULTIPROCESSING POOL

Python multiprocessing Pool can be used for parallel execution of a function across multiple input values, distributing the input data across processes (data parallelism).

Below is a simple Python multiprocessing Pool example.

```
from multiprocessing import Pool
import time

work = (["A", 6], ["B", 4], ["C", 1], ["D", 3])

def work_log(work_data):
    print(" Process %s waiting %s seconds" %
                    (work_data[0], work_data[1]))
    time.sleep(int(work_data[1]))
    print(" Process %s Finished." % work_data[0])

def pool_handler():
    p = Pool(4)
    p.map(work_log, work)

if __name__ == '__main__':
    pool_handler()
```

Here is an execution of the multiprocessing program from above.

```
 Process C waiting 1 seconds
 Process C Finished.
 Process D waiting 3 seconds
 Process D Finished.
 Process B waiting 4 seconds
 Process B Finished.
 Process A waiting 6 seconds
 Process A Finished
.
```

NUMBA

Numba is an alternative for high-performance execution of python code. Numba is a just-in-time compiler for Python that works best on code that uses NumPy arrays and functions, and loops. The most common way to use Numba is through its

collection of decorators that can be applied to your functions to instruct Numba to compile them for fast execution. When a call is made to a Numba decorated function it is compiled to machine code “just-in-time” for execution and all or part of your code can subsequently run at native machine code speed.

The Numba library can often be used to speed up Python code that uses the popular NumPy library without changing the target code. Functions (with @numba.jit decorator) that are just-in-time (JIT) compiled into code can be significantly faster than the pure Python code, by as much as a factor of several hundred or more. Numba speedup is obtained mainly for functions that use NumPy arrays, for which Numba can automatically perform type interference and generate optimized code for the required type signatures. Numba is also designed to take advantage of multicore hardware and commodity GPUs.

Here are commands that will work for executing Numba decorated code on OSC.

```
-bash-4.2$ module load miniconda3
-bash-4.2$ conda create -n local numpy numba
-bash-4.2$ source activate local
-bash-4.2$ python myproj.py
```

Numba works well on code that looks like it has loops over large arrays. For example in the following script we have a function gofast() to compute a special trace, which is a sum reduction using the tanh function on the diagonal elements of the matrix. With the numba jit decorator the function is compiled to machine code when called the first time. The behavior of the 'nopython=True' compilation mode is to essentially compile the decorated function so that it will run entirely without the involvement of the Python interpreter. This is the recommended and best-practice way to use the Numba jit decorator as it leads to the best performance.

```
from numba import jit
import numpy as np

@jit(nopython=True)
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
       trace += np.tanh(a[i, i])
    return a + trace

>>> x = np.arange(100).reshape(10, 10)
>>> gofast(x)
[[  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.]
 [ 19.  20.  21.  22.  23.  24.  25.  26.  27.  28.]
 [ 29.  30.  31.  32.  33.  34.  35.  36.  37.  38.]
 [ 39.  40.  41.  42.  43.  44.  45.  46.  47.  48.]
 [ 49.  50.  51.  52.  53.  54.  55.  56.  57.  58.]
 [ 59.  60.  61.  62.  63.  64.  65.  66.  67.  68.]
 [ 69.  70.  71.  72.  73.  74.  75.  76.  77.  78.]
 [ 79.  80.  81.  82.  83.  84.  85.  86.  87.  88.]
 [ 89.  90.  91.  92.  93.  94.  95.  96.  97.  98.]
 [ 99. 100. 101. 102. 103. 104. 105. 106. 107. 108.]]
```

ANALYZING PERFORMANCE

Here we run an experiment by running a numba jit decorated function go_fast() twice to compare the performance. We find that the second run, that is without the compilation overhead run substantially faster.

```
from numba import jit
import numpy as np
import time

@jit(nopython=True)
def go_fast(a): # Function is compiled and runs in
machine code
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

x = np.arange(100).reshape(10, 10)
# Here COMPILATION TIME IS INCLUDED
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (with compilation) = %s" % (end -
start))

# Here Compilation time is not included as only the
# pre-compiled code is timed
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (after compilation) = %s" % (end -
start))
```

The results are that running this program (gofast.py) show speed differences both on a supercomputer and a laptop that are

significant - the precompiled version runs about 2.5x times faster. Here is a run on OSC.

```
-bash-4.2$ module load miniconda3
-bash-4.2$ source activate local
(local) -bash-4.2$ python gofast.py
Elapsed (with compilation) = 0.00016260147094726562
Elapsed (after compilation) = 6.508827209472656e-05
```

On my laptop the results also show significant speedup, with the precompiled version running about 6x faster:

```
$ python gofast.py
Elapsed (with compilation) = 0.2538139820098877
Elapsed (after compilation) = 0.04169797897338867
```

LAB PROJECT #1

1. Using your OSC accounts or any other parallel computing platform you will create an environment for running concurrent tasks using Python (or another high-level language) for parallel programming using multithreaded computing.
2. For a deliverable, you will need to upload your code, along with a short document describing your platform and the results achieved. Please include a screenshot of a running session.

3. Your goal is to use your platform to demonstrate parallel speed up on a computationally demanding application. For this lab we will use the generation of Mandelbrot set. You may use the numba library available on OSC or through most any Anaconda distribution.
4. Test your code for evidence of speedup over a sequential version. Include in your submission document details of results of your tests comparing sequential to parallel execution.
5. Starter Code:

```
#Starter Code for the fast generation of the
# Mandelbrot Set
import numpy as np
import time
from numba import jit


def mandel(x, y, max_iters):
  c = complex(x, y)
  z = 0.0j
  for i in range(max_iters):
    z = z*z + c
    if (z.real*z.real + z.imag*z.imag) >= 4:
      return i

  return max_iters

def create_fractal(min_x, max_x, min_y, max_y, image,
iters):
  height = image.shape[0]
  width = image.shape[1]

  pixel_size_x = (max_x - min_x) / width
  pixel_size_y = (max_y - min_y) / height

  for x in range(width):
    real = min_x + x * pixel_size_x
    for y in range(height):
      imag = min_y + y * pixel_size_y
```

```
      color = mandel(real, imag, iters)
      image[y, x] = color

image = np.zeros((1024, 2024), dtype = np.uint8)


start = time.time()
create_fractal(-2.0, -1.7, -0.1, 0.1, image, 20)
end = time.time()
print('Elapsed = %s' % (end - start))

#from pylab import imshow, show
#imshow(image)
#show()
```

The last lines which are commented out can be used to show the image that was produced by the create_fractal function.

OPTIONAL MATERIAL

As optional material we list several alternative approaches using python to the problem of high performance parallel implementation of python code. These tools are not need for the Lab Project above.

IPYTHON

IPython is a popular distribution designed to avoid subtle problems with standard Python's global interpreter lock (GIL). IPython's networking layers are modular and can be updated to high-performance

network protocols. IPython can be used interactively.

The IPython engine is a Python instance that takes Python commands over a network connection. The engine can also handle incoming and outgoing Python objects sent over a network connection. When multiple engines are started, parallel and distributed computing becomes possible. An important feature of an IPython engine is that it blocks while user code is being executed.

The IPython controller provides an interface for working with a set of engines. At an general level, the controller is a process to which IPython engines can connect. For each connected engine, the controller manages a queue. All actions that can be performed on the engine go through this queue. While the engines themselves block when user code is run, the controller hides that from the user to provide a fully asynchronous interface to a set of engines.

There are different ways of working with a controller. In IPython these ways correspond to different interfaces that the controller is adapted to. Currently there are two default interfaces to the controller:

- The MultiEngine interface, which provides the simplest possible way of working with engines interactively.

- The Task interface, which provides presents the engines as a load balanced task farming system.

Advanced users can easily add new custom interfaces to enable other styles of parallelism. Data movements are often a bottleneck for high-performance. There are several suggested ideas for addressing this bottleneck when parallel programming using IPythons **`MultiEngineClient`**:

1. Have the engines write data to files on the locals disks of the engines.
2. Have the engines write data to files on a file system that is shared by the engines.
3. Have the engines write data to a database that is shared by the engines.
4. Simply keep data in the persistent memory of the engines and move the computation to the data (rather than the data to the computation).
5. See if you can pass data directly between engines using MPI.

CYTHON

Cython is a different solution for speeding up Python code. Whereas Numba is a Python library that converts pure Python

code to LLVM code that is JIT-compiled into machine code, Cython is a programming language that is a superset of the Python programming language. Cython extends Python with C-like properties. Most notably, Cython allows static type declarations. The purpose of the extensions to Python introduced in Cython is to make it possible to translate the code into efficient C or C++ code, which can be compiled into a Python extension module that can be imported and used from regular Python code.

If we are using CPython the GIL (Global Interpreter Lock) makes dicts seem thread-safe, because the Python interpreter can only execute on one thread at a time (no matter how many cores we have), so individual method calls execute as atomic actions. However, this doesn't help when we need to call two or more dict methods as a single atomic action. And in any case, we should not rely on this implementation detail; after all, other Python implementations (e.g., Jython and IronPython) don't have a GIL, so their dict methods cannot be assumed to execute atomically.

If we want a genuinely thread-safe dictionary, we must use a third-party one or create one ourselves. Creating one isn't

difficult, since we can take an existing dict and provide access to it via our own thread-safe methods. In this subsection, we will review the ThreadSafeDict, a thread-safe dictionary that provides a subset of the dict interface that is sufficient to provide meter dictionaries.

AYSYNCHROUS PROGRAMMING

Asynchronous programming in python has gained a lot of traction in the last few years. Python now has some syntax features that support concepts of asynchronous execution. Asynchronous programming in Python is similar to multi-threading, but without system scheduling involved. This means that an asynchronous program can concurrently process information, but the execution context is switched internally and not by the system scheduler. There are several differing asynchronous programming solutions, and are named and implemented differently; see for example, green threads or greenlets, coroutines, and stackless Python.

In asynchronous programming, tasks are never preempted by the main event loop and must instead return control explicitly. So this

style of multitasking is called “non-preemptive”.