**Using Python, multiprocessing and NumPy/SciPy for parallel numerical computing**

© Sturla Molden, January 2009

University of Oslo

Modern computers have processors with multicore CPUs. To use this efficiently, programs must be

written to use multiple threads or multiple processes. Because of Python’s global interpreter lock

(GIL), only one thread at a time can access the interpreter. Thus, Python’s threads (from the thread

or threading modules) cannot be used to exploit the full power of a multicore CPU, with an

exception for threads that releases the GIL in extension modules. In contrast, processes do not share

GIL. Python 2.6 and 3.0 have a standard module called multiprocessing, which allows processes

to be used with approximately the same API as threading.Thread. Thus, multiprocessing

can be used to let Python code take advantage of computers with multiple virtual processors,

without depending on extension modules to release the GIL. There is a backport of multiprocessing

to Python 2.5 that can be used together with the current versions of NumPy and SciPy.

Since processes run in isolated virtual memory spaces, objects must either be serialized and

communicated using IPC (e.g. pipes or sockets) or kept in special shared memory segments. The

multiprocessing package has a special Queue object for synchronized IPC via pipes, and special

objects for allocating ctypes objects (and arrays of them) in shared memory: Value, RawValue,

Array and RawArray. Access to Value and Array is serialized by a lock. For example, a shared

memory array of 1000 double precision floating point numbers is created as follows:

import ctypes

import multiprocessing as mp

shmem = mp.RawArray(ctypes.c\_double, 1000)

Processes are created with the multiprocessing.Process object, which has almost the same

interface as threading.Thread. Shared memory objects must be used as initialization

arguments to the Process object. To use them for scientific computing with NumPy and SciPy, we

need to view the shared memory region as a numpy.ndarray object. This is a moving target as

multiprocessing’s internals keep changing. Currently, we can do it like this:

def shmem\_as\_ndarray( raw\_array ):

\_ctypes\_to\_numpy = {

ctypes.c\_char : numpy.int8,

ctypes.c\_wchar : numpy.int16,

ctypes.c\_byte : numpy.int8,

ctypes.c\_ubyte : numpy.uint8,

ctypes.c\_short : numpy.int16,

ctypes.c\_ushort : numpy.uint16,

ctypes.c\_int : numpy.int32,

ctypes.c\_uint : numpy.int32,

ctypes.c\_long : numpy.int32,

ctypes.c\_ulong : numpy.int32,

ctypes.c\_float : numpy.float32,

ctypes.c\_double : numpy.float64

}

address = raw\_array.\_wrapper.get\_address() size = raw\_array.\_wrapper.get\_size()

dtype = \_ctypes\_to\_numpy[raw\_array.\_type\_]

class Dummy(object): pass

d = Dummy()

d.\_\_array\_interface\_\_ = {

'data' : (address, False),

'typestr' : numpy.uint8.str,

'descr' : numpy.uint8.descr,

'shape' : (size,),

'strides' : None,

'version' : 3

}

return numpy.asarray(d).view( dtype=dtype )

The next thing to check is the number of processors on the computer. It seldom helps to have more

worker processes than virtual processors:

def number\_of\_processors():

''' number of virtual processors on the computer '''

# Windows

if os.name == 'nt':

return int(os.getenv('NUMBER\_OF\_PROCESSORS'))

# Linux

elif sys.platform == 'linux2':

retv = 0

with open('/proc/cpuinfo','rt') as cpuinfo:

for line in cpuinfo:

if line[:9] == 'processor': retv += 1

return retv

# Please add similar hacks for MacOSX, Solaris, Irix,

# FreeBSD, HPUX, etc.

else:

raise RuntimeError, 'unknown platform'

To serialize access to an array in shared memory, we can use a special scheduler protected by an

instance of multiprocessing.Lock. Internally, multiprocessing.Lock is implemented

using an OS semaphore. In addition to serializing memory access, the scheduler works as a “load

balancer” for the CPUs. We ideally want to keep the CPUs busy at all times, not being idle waiting for

the others, and scheduling work as seldom as possible. The choice of scheduling strategy is

important. With “static” scheduling, an array is split in equally sized sub-regions between the

processes. This is the simplest strategy, and has the smallest scheduling overhead. In “dynamic”

scheduling, an array is split into multiple small sub-arrays, many more than the number of

processors, e.g. by an order of magnitude. This will be the most efficient strategy if the computations

vary in duration, thus preventing some of the processors to idle for a long time while the others

complete their work. This strategy can have a large scheduling overhead. A “guided” scheduler is a

compromise between dynamic and static scheduling. The chunk size is adaptively reduced until a

lower limit is reached.

Here is a scheduler that is implemented as a Python iterator, and returns slice objects for

accessing reserved sub-arrays.

class Scheduler(object):

def \_\_init\_\_(self, ndata, nprocs, chunk=None,schedule='guided'):

if not schedule in ['guided','dynamic','static']:

raise ValueError, 'unknown scheduling strategy'

self.\_ndata = mp.RawValue(ctypes.c\_int,ndata)

self.\_start = mp.RawValue(ctypes.c\_int,0)

self.\_lock = mp.Lock()

self.\_schedule = schedule

self.\_nprocs = nprocs

if schedule == 'guided' or schedule == 'dynamic':

min\_chunk = ndata // (10\*nprocs)

if chunk:

min\_chunk = chunk

min\_chunk = 1 if min\_chunk < 1 else min\_chunk

self.\_chunk = min\_chunk

elif schedule == 'static'

min\_chunk = ndata // nprocs

if chunk:

min\_chunk = chunk if chunk > min\_chunk else min\_chunk

min\_chunk = 1 if min\_chunk < 1 else min\_chunk

self.\_chunk = min\_chunk

def \_\_iter\_\_(self):

return self

def next(self):

self.\_lock.acquire()

ndata = self.\_ndata.value

nprocs = self.\_nprocs

start = self.\_start.value

if self.\_schedule == 'guided':

\_chunk = ndata // nprocs

chunk = max(self.\_chunk, \_chunk)

else:

chunk = self.\_chunk

if ndata:

if chunk > ndata:

s0 = start

s1 = start + ndata

self.\_ndata.value = 0

else:

s0 = start

s1 = start + chunk

self.\_ndata.value = ndata - chunk

self.\_start.value = start + chunk

self.\_lock.release()

return slice(s0, s1)

else:

self.\_lock.release()

raise StopIteration

For illustration I will show you a subclass of scipy.spatial.cKDTree that uses

multiprocessing for parallel queries.

import numpy

import scipy

import scipy.spatial

import multiprocessing as mp

import ctypes

import os

import sys

class cKDTree\_MP( scipy.spatial.cKDTree ):

''' Multiprocessing cKDTree subclass, shared memory '''

def \_\_init\_\_(self, data, leafsize=10):

'''

Same as cKDTree.\_\_init\_\_ except that an internal copy

of data to shared memory is made.

'''

n, m = data.shape

# Allocate shared memory for data

self.shmem\_data = mp.RawArray(ctypes.c\_double, n\*m)

# View shared memory as ndarray, and copy over the data.

# The RawArray objects have information about the dtype and

# buffer size.

\_data = shmem\_as\_ndarray(self.shmem\_data).reshape((n,m))

\_data[:,:] = data

# Initialize parent, we must do this last because

# cKDTree stores a reference to the data array. We pass in

# the copy in shared memory rather than the origial data.

self.\_leafsize = leafsize

super(cKDTree\_MP, self).\_\_init\_\_(\_data, leafsize=leafsize)

def parallel\_query(self, x, k=1, eps=0, p=2,

distance\_upper\_bound=numpy.inf,

chunk=None, schedule='guided'):

'''

Same as cKDTree.query except parallelized with multiple

processes and shared memory.

Extra keyword arguments:

chunk : Minimum chunk size for the load balancer.

schedule: Strategy for balancing work load

('static', 'dynamic' or 'guided').

'''

# allocate shared memory for x and result

nx = x.shape[0]

shmem\_x = mp.RawArray(ctypes.c\_double, nx\*self.m)

shmem\_d = mp.RawArray(ctypes.c\_double, nx\*k)

shmem\_i = mp.RawArray(ctypes.c\_int, nx\*k)

# view shared memory as ndarrays

\_x = shmem\_as\_ndarray(shmem\_x).reshape((nx,self.m))

\_d = shmem\_as\_ndarray(shmem\_d).reshape((nx,k))

\_i = shmem\_as\_ndarray(shmem\_i).reshape((nx,k))

# copy x to shared memory

\_x[:] = x

# set up a scheduler to load balance the query

nprocs = number\_of\_processors()

scheduler = Scheduler(nx, nprocs, chunk=chunk, schedule=schedule)

# return status in shared memory

# access to these values are serialized automatically

ierr = mp.Value(ctypes.c\_int, 0)

err\_msg = mp.Array(ctypes.c\_char, 1024)

# query with multiple processes

query\_args = (scheduler,

self.shmem\_data, self.n, self.m, self.leafsize,

shmem\_x, nx, shmem\_d, shmem\_i,

k, eps, p, distance\_upper\_bound,

ierr)

query\_fun = \_parallel\_query

pool = [mp.Process(target=query\_fun, args=query\_args) for n in range(nprocs)]

for p in pool: p.start()

for p in pool: p.join()

if ierr.value != 0:

raise RuntimeError, \

('%d errors in worker processes. Last one reported:\n%s'

% (ierr.value, err\_msg.value))

# return results (private memory)

return \_d.copy(), \_i.copy()

# This is executed in an external process:

def \_parallel\_query(scheduler, # scheduler for load balancing

data, ndata, ndim, leafsize, # data needed to reconstruct the kd-tree

x, nx, d, i, # query data and results

k, eps, p, dub, # auxillary query parameters

ierr): # return values (0 on success)

# Notice how this almost looks like a subroutine call in Fortran77,

# as shapes and return arrays are passed in as arguments.

try:

# View shared memory as ndarrays.

\_data = shmem\_as\_ndarray(data).reshape((ndata,ndim))

\_x = shmem\_as\_ndarray(x).reshape((nx,ndim))

\_d = shmem\_as\_ndarray(d).reshape((nx,k))

\_i = shmem\_as\_ndarray(i).reshape((nx,k))

# Reconstruct the kd-tree from the data.

# This is relatively inexpensive.

kdtree = scipy.spatial.cKDTree(\_data, leafsize=leafsize)

# Query for nearest neighbours, using slice ranges,

# from the load balancer.

for s in scheduler:

\_d[s,:],\_i[s,:] = kdtree.query(\_x[s,:], k=k, eps=eps, p=p,

distance\_upper\_bound=dub)

# An error occured, increment the return value ierr.

# Access to ierr is serialized by multiprocessing.

except:

ierr.value += 1

Now we can benchmark the performance of this code. For comparison with multithreading in C, a

version of scipy.spatial.cKDTree that use OpenMP was written. Kd-tree search for 5

nearest-neighbours was tested for an 8 dimensional data set of various size. The single-threaded

cKDTree is marked with black, the multiprocessing sublass is blue, and the OpenMP version is red.

We can now see how they perform on a computer with two processors (my dualcore laptop used to

write this document):

Using multiprocessing incurs some overhead, in the order of a few secon

below 10000 points this is worse than just using

version with OpenMP was as expected the fastest. But as

Python and multiprocessing

The final thing that I will show is this tutorial is how to synchronize a pool of multiple processes.

“barrier” is a synchronization primitive often encountered in

not have a barrier primitive in multipro

multiprocessing.Event objects. It is so useful for numerical code that I will show how to

construct one. There are many ways to construct a barrier primitive. A dissemination barrier is not

the most efficient, but it is simple to program. So

anyway. This example also shows the other way

subclassing and providing a custom

import multiprocessing as mp

from math import ceil, log

class PoolProcess( mp.Process ):

def \_\_init\_\_(self, rank, events, numproc, lock):

mp.Process.\_\_init\_\_(self)

self.rank = rank

self.events = events

self.numproc = numproc

self.lock = lock

def barrier(self):

if self.numproc == 1: return

incurs some overhead, in the order of a few seconds. And for data sizes

below 10000 points this is worse than just using cKDTree with a single-thread. In comparison, the

version with OpenMP was as expected the fastest. But as the number of data increase

multiprocessing was not much worse than multithreading in C with

I will show is this tutorial is how to synchronize a pool of multiple processes.

synchronization primitive often encountered in parallel numerical code. Python does

not have a barrier primitive in multiprocessing, but we can easily create one using

objects. It is so useful for numerical code that I will show how to

There are many ways to construct a barrier primitive. A dissemination barrier is not

the most efficient, but it is simple to program. Some overhead from using Python

anyway. This example also shows the other way to use multiprocessing.Process

subclassing and providing a custom run() method.

from math import ceil, log

class PoolProcess( mp.Process ):

def \_\_init\_\_(self, rank, events, numproc, lock):

ess.\_\_init\_\_(self)

if self.numproc == 1: return

s. And for data sizes

thread. In comparison, the

of data increases, using

multithreading in C with OpenMP.

I will show is this tutorial is how to synchronize a pool of multiple processes. A

code. Python does

cessing, but we can easily create one using a number of

objects. It is so useful for numerical code that I will show how to

There are many ways to construct a barrier primitive. A dissemination barrier is not

using Python is expected

multiprocessing.Process: # loop log2(num\_threads) times, rounding up

for k in range(int(ceil(log(self.numproc)/log(2)))):

# send event to thread (rank + 2\*\*k) % numproc

receiver = (self.rank + 2\*\*k) % self.numproc

evt = self.events[ self.rank \* self.numproc + receiver ]

evt.set()

# wait for event from thread (rank - 2\*\*k) % numproc

sender = (self.rank - 2\*\*k) % self.numproc

evt = self.events[ sender \* self.numproc + self.rank ]

evt.wait()

evt.clear()

def run(self):

# print the rank of this process

# synchronize access to stdout

self.lock.acquire()

print 'Hello World, I am process %d' % self.rank

self.lock.release()

# wait for the self.numproc - 1 other processes

# to finish printing

self.barrier()

# print the rank of this process

# synchronize access to stdout

self.lock.acquire()

print 'Hello World again, I am process %d' % self.rank

self.lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

numproc = 4

lock = mp.Lock()

events = [mp.Event() for n in range(numproc\*\*2)]

pool = [PoolProcess(rank, events, numproc, lock) for rank in range(numproc)]

for p in pool: p.start()

for p in pool: p.join()

When we run this code, the output becomes:

Hello World, I am process 2

Hello World, I am process 0

Hello World, I am process 1

Hello World, I am process 3

Hello World again, I am process 3

Hello World again, I am process 1

Hello World again, I am process 0

Hello World again, I am process 2

The effect of the barrier is easy to spot.

That’s all folks. Have fun multiprocessing with NumPy!