Physics 468: Computing Project 8

January 24, 2020

Diffusion Monte-Carlo on Parallel Processors

In computing project 4 you used the idea of "diffusion monte-carlo", or DMC, to estimate the properties of a one dimensional system's ground state. In this project you'll extend the code you wrote last time so that it can operate on a system with multiple processors. We have built such a system in the department that you can use on this project to execute your final code. However, to develop your code you'll need some kind of framework that enables to factor the calculation in such a way that it can be parallelized automatically. In order to minimize the amount of computer science you need to learn to make this work, I've cooked up a python module that I call "`handympi`" that provides two ways: one "trivial" and another "easy" way to take advantage of parallel processors if they are available. However, the beauty of `handympi` is that it *works* even if there is only one processor around.

The `handympi` module

MPI [1] is a technology that allows multiple processes, even on separate computing platforms, to interoperate cooperatively to generate higher performance than would be possible with a single process. The `handympi` module leverages the simple python package `pypar` [2] to provide a novice programmer easy access to some the potential of MPI without requiring mastery of any of the intricacies of the MPI/pypar API.

The `handympi` module provides two basic functions: `foreach` and `RunMasterSlave`.

The easy way: `foreach`

The `foreach` function is the easiest to use. Just pass in a function and a list. The function is applied to each element in the list. If you set `return_` to 'True', then the return values for each element of the list are accumulated and returned. What's so amazingly cool about that? Well.. what's fun is that if there is no MPI environment, the function just iterates through the list serially. However, if there is a working MPI environment, the list is split up and sent out to the compute nodes for evaluation and the results are accumulated automatically! In other words, you can develop your code in a non-MPI environment, and the RUN your code on an MPI system with *no change*.

`foreach` interface

```

def foreach(f, l, useMPI=True, return_=False, debug=False, finalRun=True):
    """
    for each element in list 'l' apply the function 'f'.
    You can force serial operation by setting useMPI to 'False'
    The last time you're call foreach... make sure finalRun=True.
    """
```

A bit less easy, but more flexible: `RunMasterSlave`

The `foreach` function is great for simple parallel problems where each function call is completely independent of all the others. There are times however where you want the function to *remember* something from the last time. The `RunMasterSlave` function is intended to fill that gap in a simple way that avoids having to learn much about MPI, but does require the user to know a bit about object oriented programming. The idea is for the user to supply a 'master' and a 'slave' class. These classes must have default constructors that take no arguments. They also need to produce instances that are "callable". In python that means they need to supply a `__call__` method that is invoked when the instances are treated like functions. Here's the calling interface:

`RunMasterSlave` interface

```
def RunMasterSlave(masterClass, slaveClass, workParams, useMPI=True,
    finalRun=True):
    """
    This is a generic master/slave runner.
    """
```

In a non-MPI environment only one slave and one master are constructed. The `workParams` argument is like the list argument from `foreach`. Each element of the `workParams` list is fed to a slave instance (as the only argument in a 'call') and the result is fed back to the master using it's 'call' interface as well. When the master is called, it also get's the corresponding index from the `workParams` list. Here is the worlds simplest example of a working implementation of a master and a slave:

master/slave call interface

```
class Master:
    def __init__(self):
        self.results = []

    def __call__(self, index, result):
        self.results.append(result)
```

```

class Slave:
    def __call__(self, params):
        return f(params)
```

What's cool about that? Again, similar to `foreach` code built on `RunMasterSlave` can be run on a computer with *no MPI* setup and tested there. Once the code is moved to an MPI enabled system, it can take advantage of those resources.

Below are full listings of simple test programs that exercise the `foreach` and `RunMasterSlave` functions from the `handympi` module.

testMSMPI.py

```

from handympi import RunMasterSlave, MY_RANK

from pylab import *
import sys

N=10000000
if len(sys.argv)>1:
    N = int(sys.argv[1])

def f(N):
    y = arccos(1.0-2*rand(N))
    return 2.0*y.sum()/N

Ns = [N]*30

class Master:
    def __init__(self):
        self.results = []

    def __call__(self, index, result):
        self.results.append(result)

class Slave:
    def __call__(self, params):
        return f(params)

masterInstance = RunMasterSlave(Master, Slave, Ns, finalRun=False)

if MY_RANK==0:
    print masterInstance.results
    print "in between runs..."
else:
    print "different rank"

masterInstance = RunMasterSlave(Master, Slave, Ns)
print masterInstance.results
```

testHMPI.py

```

from handympi import foreach, MY_RANK
from pylab import *
import sys

N=10000000
if len(sys.argv)>1:
    N = int(sys.argv[1])

def f(N):
    y = arccos(1.0-2*rand(N))
    return 2.0*y.sum()/N

Ns = [N]*30

result = foreach(f, Ns, return_=True, finalRun=False)

if MY_RANK==0:
    print result
    print "in between runs..."
else:
    print "different rank"

print foreach(f, Ns, return_=True)
```

So... how to we parallelize our code?

There is a modified version of the `mcSteps` code you used in project 4 on the 'K' drive that should enable you to get started. For DMC it's best to use the Master/Slave class approach since we want to "reuse" the walkers after each iteration. I've given an example for the simple harmonic oscillator again. It's called `cp8_sho.py` and it is also on the 'K' drive along with the `mcSteps.py` and `handympi.py` modules.

Here's an example that uses the SHO potential `cp8_sho.py`:

Notice that the slave *saves* it's walkers from call to call so that they can be reused.

cp8_sho.py

```
#
# Computing Project 8: Diffusion MonteCaro in Parallel, SHO version
#

from handympi import RunMasterSlave, HAVE_MPI, MY_RANK
from mcSteps import doMCSteps
from numpy import *

bins = 50
xrange = 2.5
```

```python

def SHO(x,params=None):
    """
    SHO potential
    """
    v=x**2/2
    return v

class Master:

    def __init__(self,
                 histdims=1,        # number of histogram dimensions
                 numBins=bins,      # number of bins in each dimension
                 ):

        self.numBins = numBins
        self.histdims = histdims
        self.h = zeros((numBins,)*histdims)
        self.energies = []

    def __call__(self, index, result):
        """
        Master called with new result from a slave
        """
        h=result['hist']
        print "in master h:",h.shape

        self.h += h                              # accumulate histogram
        self.energies.append(result['vavg'])   # get current vref

class Slave:

    def __init__(self,
                 steps=1000,
                 walkers=1000,
                 numBins=bins,
                 ):

        self.count = 0
        self.steps=steps
        self.walkers=walkers
        self.numBins=numBins

    def __call__(self, params):

        h=None

        if self.count == 0:
            #
            # need to thermalize one time
            #
            h, binArray, vrefs, self.ws = doMCSteps(mcsteps=self.steps,
                                                    nWalkers=self.walkers,
                                                    numBins=self.numBins,
                                                    V=SHO, dims=1, minX=-xrange,
                                                         maxX=xrange)

            print "slave:", MY_RANK, "got hist (thermalizing)", h.shape

        self.count += 1
        h, binArray, vrefs, self.ws = doMCSteps(mcsteps=self.steps,
                                                nWalkers=self.walkers,
                                                numBins=self.numBins,
                                                V=SHO, oldWalkers = self.ws,
                                                minX=-xrange, maxX=xrange,
                                                )

        vavg = array(vrefs).sum()/len(vrefs)

        print "slave", MY_RANK, "got hist:", h.shape, self.count

        return {'hist':h, 'vavg':vavg}


if __name__=='__main__':

    if HAVE_MPI:
        print "We found an MPI environment", MY_RANK
    else:
        print "No MPI found...."

    theMaster = RunMasterSlave(Master, Slave, [0]*20)
    print "Finished! Now write out results"
    import csv

    if MY_RANK==0:
        csvWriter = csv.writer(open('dmc.csv', 'wb'), delimiter=',',
            quotechar='"', quoting=csv.QUOTE_MINIMAL)
        csvWriter.writerow(['Index','Value'])
        for i in range(len(theMaster.h)):
            csvWriter.writerow([`i`, `theMaster.h[i]`])
```

Finally, you can use the mcSteps function which has the following interface:

master/slave call interface

```
def doMCSteps(mcsteps = 1000,    # number of monte-carlo steps
              ds=0.1,            # step size
              nWalkers=1000,     # target number of walkers
              minX=-3.0,         # minX to start walkers
              maxX=3.0,          # maxX to start walkers
              numBins=30,        # number of histogram bins
              V=None,            # potential. Default to Simple Harmonic
                  Oscillator
              oldWalkers = None, # use these walkers.. (already thermalized)
              oldHist = None,    # use this histogram
              alpha = 1.0,       # the dn factor...
```

```
                 dims=1,             # the number of dimensions
                 params=None,        # other parameters needed by the potential
                 debug=False,        # print debugging output?
                 histdims=None,      # histogram dimensions if different
                 useInline=None,     # set to true to use inline histogram (if
                     you have scipy.weave)
                 ):
```

So, what are we supposed to do?

Rewrite your project 4 code to use the `handympi` module. We'll take some class time soon (as soon as everyone has a chance to get their code running with `handympi` on their own laptop) and get some timing numbers on the little cluster.

Questions

1) What did you find most difficult about moving from a single processor implementation of your DMC project to the `handympi` code?

2) How much faster did your code run on the baby cluster?

3) Do you have any suggestions for improving this project?

Please answer these questions at the end of your report.

[1] `http://www.open-mpi.org`. MPI stands for Message Passing Interface. See the URL!

[2] `http://code.google.com/p/pypar/`. PyPar builds easily on LittleFe. You will need to apt-get the development files for python to build it.