The `handympi` Module

Steve Spicklemire

April 2, 2012

Why another MPI 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` (inspired by the similarly named `handythread` module) 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

```
1
2 def foreach(f, l, useMPI=True, return_=False, debug=False, finalRun=True):
3     """
4     for each element in list 'l' apply the function 'f'.
5     You can force serial operation by setting useMPI to 'False'
6     The last time you're call foreach... make sure finalRun=True.
7     """
```

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

```
1
2  from handympi import RunMasterSlave, MY_RANK
3
4  from pylab import *
5  import sys
6
7  N=10000000
8  if len(sys.argv)>1:
9      N = int(sys.argv[1])
10
11 def f(N):
12     y = arccos(1.0-2*rand(N))
13     return 2.0*y.sum()/N
14
15 Ns = [N]*30
16
17 class Master:
18     def __init__(self):
19         self.results = []
20
21     def __call__(self, index, result):
22         self.results.append(result)
23
24 class Slave:
25     def __call__(self, params):
26         return f(params)
27
28 masterInstance = RunMasterSlave(Master, Slave, Ns, finalRun=False)
29
30 if MY_RANK==0:
31     print masterInstance.results
32     print "in between runs..."
33 else:
34     print "different rank"
35
36 masterInstance = RunMasterSlave(Master, Slave, Ns)
37 print masterInstance.results
```

testHMPI.py

```
1
2  from handympi import foreach, MY_RANK
3  from pylab import *
4  import sys
5
6  N=10000000
7  if len(sys.argv)>1:
8      N = int(sys.argv[1])
9
```

```
10 def f(N):
11     y = arccos(1.0-2*rand(N))
12     return 2.0*y.sum()/N
13
14 Ns = [N]*30
15
16 result = foreach(f, Ns, return_=True, finalRun=False)
17
18 if MY_RANK==0:
19     print result
20     print "in between runs..."
21 else:
22     print "different rank"
23
24 print foreach(f, Ns, return_=True)
```

[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.