Parallel computation All Versions

Examples

Using the multiprocessing module to parallelise tasks

```
import multiprocessing

def fib(n):
    """computing the Fibonacci in an inefficient way
    was chosen to slow down the CPU."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# Out: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

As the execution of each call to `fib` happens in parallel, the time of execution of the full example is **1.8× faster** than if done in a sequential way on a dual processor.

Python 2.2+

Using a C-extension to parallelize tasks

The idea here is to move the computationally intensive jobs to C (using special macros), independent of Python, and have the C code release the GIL while it's working.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Using Parent and Children scripts to execute code in parallel

child.py

```
import time

def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

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

parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

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

This is useful for parallel, independent HTTP request/response tasks or Database select/inserts. Command line arguments can be given to the **child.py** script as well. Synchronization between scripts can be achieved by all scripts regularly checking a separate server (like a Redis instance).

Using PyPar module to parallelize

PyPar is a library that uses the message passing interface (MPI) to provide parallelism in Python. A simple example in PyPar (as seen at https://github.com/daleroberts/pypar) looks like this:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
  msh = 'P0'
  pp.send(msg, destination=1)
  msg = pp.receive(source=rank-1)
  print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
else:
  source = rank-1
  destination = (rank+1) % ncpus
  msg = pp.receive(source)
  msg = msg + 'P' + str(rank)
  pypar.send(msg, destination)
pp.finalize()
```

Syntax

Parameters

Remarks

Due to the GIL (Global interpreter lock) only one instance of the python interpreter executes in a single process. So in general, using multi-threading only improves IO bound computations, not CPU-bound ones. The multiprocessing module is recommended if you wish to parallelise CPU-bound tasks.

GIL applies to CPython, the most popular implementation of Python, as well as PyPy. Other implementations such as Jython and IronPython have no GIL .