Computing Thoughts
Concurrency with Python, Twisted, and Flex
by Bruce Eckel
May 3, 2008

Summary
An example of parallel programming using all the CPUs on your computer or cluster. Also shows how to add a Flex user interface.

For the past few years I've been consulting with a group at Parsons-Brinkerhoff, led by Rick Donnelly, which specializes in traffic analysis. The numerical models for this analysis are extremely detailed and a model can require days to run. I've been helping them develop concurrent systems to solve these problems more quickly. Some of this consulting is knowledge transfer and some is software development. This article is based on some of the work we've done, which came together last week with the help of Michail Xyntarakis, one of the Parsons-Brinkerhoff engineers.

This article shows a basic framework for putting a subprocess on each CPU in your system. Each subprocess is a worker that solves part of the problem. The example works on a single machine, but it can easily be modified to work with a cluster.

The system you see here is only a framework for creating concurrent programs. There are various ways to implement your particular solution, but what I show here solves all the hard problems of wiring things up.

Dividing Your Problem into Subprocesses

This is the tricky part, from both a math and a programming standpoint. Somehow, your problem must be divisible into pieces that can run independently of each other. In the simplest case, the so-called "embarrassingly parallel" scenario, the pieces of a problem are inherently independent. Here, you just divide up the work and parcel it out to the different processes. Balancing the work across processors so that progress is optimized is relatively easy in this case.

If your problem is *not* embarrassingly parallel you'll need to guess and experiment with different tasks in your program, trying to estimate which parts of the problem can be solved independently. Very often you'll need to use multiple steps, waiting for all the parts to catch up before moving onto the next step.

The framework shown here can be used for embarrassingly parallel problems, but it also shows multiple steps and uses a "barrier" tool to automatically let all the pieces finish at each step.

Using easy_install

easy_install is a system that belongs in the standard Python library, and I'm sure it will get there once Phillip Eby finishes working on it (hint, hint). In the meantime, download it here. It's a single script that takes care of its own installation.

easy_install dramatically simplifies the process of installing Python libraries, basically down to the point where you can say **easy_install somepackage** and it will hunt through the internet, find the right library and install it. It will also check to see if you have the right dependencies, and install those -- including the correct versions -- if you don't. If necessary, it will build code by running compilers.

The dependency and version checking and fixing is especially nice for software installations for end-users.

Understanding the Twisted Library

I've written an earlier article on the basics of Twisted. And this article by the creators of Twisted gives a broad overview of the library.

It might seem strange that a networking library is the solution to concurrency, but once you divide your task into subprocesses, the next problem is to communicate with these subprocesses. Some platforms have specific solutions (like pipes in Unix), but to create a universal solution we need something that is cross platform.

Before wading into the world of Twisted, you should be forewarned. Twisted contains lots of libraries to do all kinds of things, and this abundance can be overwhelming. They use weird naming conventions, often making things up even though there are accepted names for things, so names are often nonsensical. The documentation sucks -- badly. You usually have to thrash around quite a bit to figure out what you need, and even then it requires luck (but the newsgroup tends to be helpful). There *is* an O'Reilly book on Twisted, but it's not great -- you get it anyway because that's the only book there is.

Normally, caveats like that might send you away from an open-source project. Despite the issues, Twisted is worth it because the code is very good, and it solves your problem. The project *is* very actively maintained, but it's by people who are good at writing (and testing) code, and not so good at documentation. It's definitely worth using, but you should be ready to roll up your sleeves. However, if you can fit your project within what I'm showing in this article, things should go easily.

Go to TwistedMatrix.com to install Twisted. If you're on Windows or OSX you should use the specific installers; for Linux **easy_install** should work.

What is Asynchronous Programming?

One of the many problems with threading is that it has multiple meanings. There are three basic uses for threads:

1. **Code organization.** Here, the use of threads don't increase the performance of an application but instead make the code easier to create and maintain.
2. **Long Calls.** While you're waiting on disk or network IO, for example, you might be able to do something useful. By attaching a thread to the IO call, the main execution path of the program can continue (watching for user input, for example, to keep the UI responsive). This type of

threading can easily be done with a single CPU, and it's basically a variation of #1, because you could write your code to periodically go back and check on the IO. This checking code is effectively what's being done for you when you use threads.

Both Ruby and Python can only use #1 and #2. Because of the global interpreter lock in both languages, neither can use more than one CPU.

3. **Parallelism.** Although #2 makes better use of a single CPU, to make use of more than one CPU we need true parallelism, where multiple CPUs are working on the problem at the same time. To accomplish this in Python in this article, we create multiple operating system processes, one for each solver (worker) (The master only requires occasional CPU cycles and it shares the CPUs to do this). A separate Python interpreter is started in each process, and each process has its own memory so there is no possibility of collision as there is with threads, which makes multi-process parallelism much easier and safer than thread programming.

Asynchronous programming is similar to case #2 in that it allows you to make a call without waiting for that call to complete. But instead of using threads to accomplish this, it uses callbacks. Callbacks are typically much faster than threads because there is no overhead for context switching, and you also don't have the cost of periodically visiting threads to see if something has changed. You simply set up callbacks and when something changes your functions get called.

The downside to asynchronous programming is that you must always be aware that you're doing it (with threads, you have the illusion that you can ignore the issue, although this is often a mistake). Every time you make an asynchronous call, you must hook up the "success" callback function and the "failure" callback function. Initially this can be a little distracting, but I've found that it's not that much of a bother. And in the end, when you make an asynchronous call you really don't know how long it's going to take or whether it will fail, so an asynchronous model is often more useful than a pretend-synchronous approach like XML-RPC, where if a call takes a long time you must set up your own asynchronous approach, which is usually polling.

The Reactor Pattern

In a regular program, the program makes things happen. In an event-driven system like Twisted (and Flex), the program sits and waits for things (events) to happen, and then it responds to them. Thus, you don't write a typical **main()** that drives things; instead, you hand over control to something that watches for events.

Because there's only a single thread running everything, it's essential that incoming events be guaranteed to be handled. Twisted uses the *reactor* pattern to accomplish this. The reactor is part of Twisted, and you just attach things to it and run it to achieve your goals.

Anytime an event occurs, it's the reactor that makes sure that it is routed to the right place. When an event happens, the reactor stores it on a queue, then goes back to watching for events. The events in the queue are handled in the background, while the reactor is constantly vigilant for new events.

Twisted has a single global object called **reactor** which becomes the main loop for your program, so you simply attach your event handlers to the **reactor** and start it up.

The Solver/Worker Process

Each **Solver** (a.k.a. *worker*) runs in its own OS process and works on its own portion of the divided labor:

```
"""
solver.py
Solves one portion of a problem, in a separate process on a separate CPU
"""
import sys, random, math
from twisted.spread import pb
from twisted.internet import reactor

class Solver(pb.Root):

    def __init__(self, id):
        self.id = id

    def __str__(self): # String representation
        return "Solver %s" % self.id

    def remote_initialize(self, initArg):
        return "%s initialized" % self

    def step(self, arg):
        "Simulate work and return result"
        result = 0
        for i in range(random.randint(1000000, 3000000)):
            angle = math.radians(random.randint(0, 45))
            result += math.tanh(angle)/math.cosh(angle)
        return "%s, %s, result: %.2f" % (self, str(arg), result)

    # Alias methods, for demonstration version:
    remote_step1 = step
    remote_step2 = step
    remote_step3 = step

    def remote_status(self):
        return "%s operational" % self

    def remote_terminate(self):
        reactor.callLater(0.5, reactor.stop)
        return "%s terminating..." % self

if __name__ == "__main__":
    port = int(sys.argv[1])
    reactor.listenTCP(port, pb.PBServerFactory(Solver(sys.argv[1])))
    reactor.run()
```

Solver looks like any other class except that it inherits from **pb.Root** and I've used an aliasing trick to create the "remote_step" methods from the single **step()**.

The **remote_** tells Twisted that this is a remote method, to be visible on the network. However, you do not include the **remote_** part when you're making a network call to the method.

main sets up the reactor to listen at the port provided on the command line, and runs the **reactor**, which takes control of the program. When the **reactor** stops, the program terminates.

The **step()** method is important. The calculation loop simulates the work that the **Solver** is doing. In a normal program, the **step()** method would not return until all the work of the method completes, but in Twisted, a remote call to any of the "step" functions will return right away -- but not with the result (which it's still working on), but with a **Deferred** object. You'll see when the controller runs that the

calls actually do return right away, while the **Solver step()** method itself goes on working. This magic is taken care of automatically by Twisted.

The **remote_status()** method will show that the **Solver** can still receive new messages even while **step()** is working. Even though **step()** immediately starts into an intensive calculation -- apparently *not* returning control to the **reactor** -- you can still make requests to the solver, which is what the following program, `controller.py`, does. These requests won't be answered right away, but neither will they be lost just because **step()** happens to be in the midst of its work. The requests are enqueued and will eventually be answered.

So although Twisted does include modules to support thread programming, it's typically not necessary. You get most of the same effect just by letting the reactor manage things.

Notice that **remote_terminate()** calls **reactor.callLater()**. The **callLater()** method inserts a request into the reactor's own queue, and you can tell it how long to wait before making the call. The second argument is the address of the function to call after waiting, which in this case is the reactor's own **stop()** method. You cannot just call **reactor.stop()** directly because once the reactor stops, the program is done which means that **remote_terminate()** would never properly exit, and that causes a failure in the remote call to **remote_terminate()**. So instead, you tell the **reactor** to stop a little later, then return from the method and clean things up before the **Solver** quits.

The Controller

The **Controller** object discovers the number of CPUs on your system, then starts a **Solver** process for each one, creating and storing a long-term network connection to each **Solver**. At the same time, it provides a server for the Flex user interface to communicate with:

```
"""
Controller.py
Starts and manages solvers in separate processes for parallel processing.
Provides an interface to the Flex UI.
"""
startIP = 8800
FlexControlPanelPort = 8050
import os, sys
from subprocess import Popen
from twisted.spread import pb
from twisted.internet import reactor, defer
from twisted.web import server, resource
from pyamf.remoting.gateway.twisted import TwistedGateway

class Controller(object):

    # Utilities:
    def broadcastCommand(self, remoteMethodName, arguments, nextStep, failureMessage):
        "Send a command with arguments to all solvers"
        print "broadcasting ...",
        deferreds = [solver.callRemote(remoteMethodName, arguments) for solver in
self.solvers.values()]
        print "broadcasted"
        reactor.callLater(3, self.checkStatus)
        # Use a barrier to wait for all to finish before nextStep:
        defer.DeferredList(deferreds, consumeErrors=True).addCallbacks(nextStep,
            self.failed, errbackArgs=(failureMessage))
```

```
    def checkStatus(self):
        "Show that solvers can still receive messages"
        for solver in self.solvers.values():
            solver.callRemote("status").addCallbacks(lambda r: sys.stdout.write(r + "\n"),
                self.failed, errbackArgs=("Status Check Failed"))
        print "Status calls made"

    def failed(self, results, failureMessage="Call Failed"):
        for (success, returnValue), (address, port) in zip(results, self.solvers):
            if not success:
                raise Exception("address: %s port: %d %s" % (address, port,
failureMessage))

    def __init__(self):
        cores = detectCPUs()
        print "Cores:", cores
        # Solver connections will be indexed by (ip, port):
        self.solvers = dict.fromkeys([("localhost", i) for i in range(startIP, startIP +
cores)])
        # Start a subprocess on a core for each solver:
        self.pids = [Popen(["python", "solver.py", str(port)]).pid for ip, port in
self.solvers]
        print "PIDs:", self.pids
        self.connected = False
        reactor.callLater(1, self.connect) # Give the solvers time to start

    def connect(self):
        "Begin the connection process"
        connections = []
        for address, port in self.solvers:
            factory = pb.PBClientFactory()
            reactor.connectTCP(address, port, factory)
            connections.append(factory.getRootObject())
        defer.DeferredList(connections, consumeErrors=True).addCallbacks(
            self.storeConnections, self.failed, errbackArgs=("Failed to Connect"))

    def storeConnections(self, results):
        for (success, solver), (address, port) in zip(results, self.solvers):
            self.solvers[address, port] = solver
        print "Connected; self.solvers:", self.solvers
        self.connected = True

    def start(self):
        "Begin the solving process"
        if not self.connected:
            return reactor.callLater(0.5, self.start)
        self.broadcastCommand("step1", ("step 1"), self.step2, "Failed Step 1")

    def step2(self, results):
        print "step 1 results:", results
        self.broadcastCommand("step2", ("step 2"), self.step3, "Failed Step 2")

    def step3(self, results):
        print "step 2 results:", results
        self.broadcastCommand("step3", ("step 3"), self.collectResults, "Failed Step 3")

    def collectResults(self, results):
```

```
        print "step 3 results:", results

def detectCPUs():
    """
    Detects the number of CPUs on a system. Cribbed from pp.
    """
    # Linux, Unix and MacOS:
    if hasattr(os, "sysconf"):
        if os.sysconf_names.has_key("SC_NPROCESSORS_ONLN"):
            # Linux & Unix:
            ncpus = os.sysconf("SC_NPROCESSORS_ONLN")
            if isinstance(ncpus, int) and ncpus > 0:
                return ncpus
        else: # OSX:
            return int(os.popen2("sysctl -n hw.ncpu")[1].read())
    # Windows:
    if os.environ.has_key("NUMBER_OF_PROCESSORS"):
            ncpus = int(os.environ["NUMBER_OF_PROCESSORS"]);
            if ncpus > 0:
                return ncpus
    return 1 # Default

class FlexInterface(pb.Root):
    """
    Interface to Flex control panel (Make sure you have at least PyAMF 0.3.1)
    """
    def __init__(self, controller):
        self.controller = controller

    def start(self, _):
        self.controller.start()
        return "Starting parallel jobs"

    def terminate(self, _):
        for solver in controller.solvers.values():
            solver.callRemote("terminate").addErrback(self.controller.failed, "Termination
Failed")
        reactor.callLater(1, reactor.stop)
        return "Terminating remote solvers"

if __name__ == "__main__":
    controller = Controller()
    # Place the namespace mapping into a TwistedGateway:
    gateway = TwistedGateway({ "controller": FlexInterface(controller) })
    # Publish the PyAMF gateway at the root URL:
    root = resource.Resource()
    root.putChild("", gateway)
    # Tell the twisted reactor to listen:
    reactor.listenTCP(FlexControlPanelPort, server.Site(root))
    # One reactor runs all servers and clients:
    reactor.run()
```

The **Controller** class begins with some utilities. **broadcastCommand()** performs all the operations necessary to make a complete network call and to deal with the consequences. In a list comprehension, the remote call is made to all **Solver** objects; the result of each call is a **Deferred** object which is stored in the list. Note the print statements before and after so you can see how fast these calls happen. The

callLater() sets up a call for three seconds later -- when each **Solver** is hard at work -- so that status calls are made to each solver, and you'll see that these also return immediately. Even though each **Solver** is in the midst of heavy math computations, the network calls are still captured and eventually handled.

In many parallel processing systems, you need to finish one step completely before moving to the next one. Twisted's **DeferredList** is another case of odd naming, because elsewhere in the concurrency world this object would be called a "barrier." A **DeferredList** simply waits for every **Deferred** in the **deferreds** list to finish, at which point it calls **nextStep**. If an error occurs it calls the subsequent method **failed()**, passing it **errbackArgs**.

This system automatically uses all available CPUs, which the constructor detects automatically using the **detectCPUs()** function, adapted from the **pp** project.

Each connection is described by an IP address and a port number, so this pair becomes the key into the **solvers** dictionary, whose values will eventually be the proxy objects representing the actual network connections. Note the use of the static **dict.fromkeys()** function to populate the keys in a dictionary using a list. The values will be inserted later once the connections have been made.

The subprocesses are created using **subprocess.Popen()** inside yet another list comprehension. Each process takes a bit of time to start up, so **callLater()** causes a one-second wait before going to **connect()**, which begins the connection process to each one of the **Solver** objects. Note the use of a **DeferredList** to wait until all the connections complete, at which point we move to **storeConnections()** where all the connection references are placed into the **solvers** dictionary.

Now nothing happens until **start()** is called, but this doesn't occur automatically; the event must come from outside the system, normally from the Flex user interface.

When **start()** is called, the system becomes a state machine, moving from one state to the next each time all the solvers have completed their tasks. You'll notice that the initialization process, starting with the constructor, is also a separate state machine; because initialization may not be complete when **start()** is called, the **connected** flag is checked and, if it's false, **start()** is re-called one half-second later. Once **collectResults()** is reached the state machine stops, at which point either **start()** can be called again or the system can be shut down.

PyAMF (described next) provides the **RemotingService**, which is convenient for talking to servers, and provides a useful sanity check:

```
"""
PyamfClientTest.py
Test the FlexInterface using the PyAMF client
"""
from pyamf.remoting.client import RemotingService

gw = RemotingService('http://localhost:8050/')
service = gw.getService('controller')
print service.start()
```

Adding a Flex User Interface

After a brief demo of Flex, and after the manager drew a sketch of the UI on the whiteboard which I threw together in a few minutes using Flex Builder's design mode, everyone was sold on creating the UI with Flex.

It's important to note that because we are using Twisted, we can't just paste some other server onto the Python system. For example, we can't use the built-in Python XML-RPC library to act as a server to the Flex UI, because that would interfere with the Twisted reactor (also, the **SimpleXMLRPCServer** is not designed for industrial use). Fortunately, Twisted supports XML-RPC among many other protocols, so we can just add another server into the Twisted system and interact with Flex that way. And Twisted's servers *are* industrial-strength.

Initially we tried using XML-RPC on the Flex end, as found in my earlier article. Unfortunately, there were some issues with the open-source Flex XML-RPC library, which might be fixed by the time you read this. After struggling with it a bit, we decided to use PyAMF, which I had experimented with during the Flex-TurboGears Jam (You can see the results of our experiments in this tutorial).

AMF (Action Message Format) is the Flex native format for network communication. This means you can use the **RemoteObject** class which comes with Flex (and more importantly, is supported). There are a number of open-source projects for using AMF with various languages -- there's a Ruby AMF, and for our case, there's PyAMF for Python. Quite conveniently, PyAMF directly supports Twisted, so you can tie it right in with the reactor.

To install PyAMF, just say `easy_install pyamf`. It's important that you get version 0.3.1 or greater in order for this example to work; during the development of this article I found a small bug in PyAMF which they promptly fixed (speed of bug fixes is always a good test of the vibrancy of an open-source project; PyAMF also supports the new Google App Engine).

The PyAMF Interface

The **FlexInterface** class was created to manage the methods that are exposed to the Flex client. This class is for organization; you can also expose standalone functions. Note that **FlexInterface** just forwards calls to the **Master** object.

The **terminate()** method tells all the solvers to terminate, then it sets the reactor to stop after 1 second. Just like with the solvers, this is necessary because stopping the reactor terminates the program. By delaying this, the **terminate()** method can return properly, so the Flex client doesn't receive an exception.

The code in **main** can just be considered voodoo necessary to start the PyAMF server within Twisted. Note that a single reactor handles all clients and servers within your system.

The Flex Client

The Flash and AIR players are single threaded. For responsiveness, they use asynchronous calls tied to callbacks, just like Twisted does.

What I've created here is just a simple demonstration with two buttons and a **TextArea** for output. However, this provides you with the basics necessary to set up communication with the Python client; any additional operations you add can follow the same format. Because we are using AMF for communication, we can just use the built-in Flex **RemoteObject** class for communication with the Python server:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" horizontalAlign="left">
<mx:RemoteObject id="remote" endpoint="http://localhost:8050" destination="controller"
    result="display(event)" fault="display(event)"/>
<mx:Button label="Start" click="remote.start()" />
```

```
<mx:Button label="Terminate" click="remote.terminate()" />
<mx:TextArea id="output" width="100%" height="100%"/>
<mx:Script>
<![CDATA[
private function display(result:*): void {
    output.text += result.message.body + "\n"
    // For more details:
    // output.text += result.toString() + "\n"
}
]]>
</mx:Script>
</mx:WindowedApplicat
```

Here, I've used a single function **display()** to show results and errors, and I've configured the **RemoteObject** with this method, so the asynchronous calls made in both buttons report back to this single function. Normally you'll provide separate functions for both results and errors, and also you'll probably need different result functions for each different call.

I've also cheated a bit by using AIR instead of a web application (note the **WindowedApplication** tag). A web application requires a cross-domain policy file to allow it to communicate. You can find out about this in this article.

Further Explorations

It was great fun to start up our program (which solved the traveling salesman problem) on an 8-core machine and watch it peg all 8 cores to 100% usage. Note the example shown here runs on a single machine with multiple cores, but the code was written to support machine clusters by including the IP address everywhere. It will still require additional adaptation for clusters, and you'll also need a small boot program written with Twisted running on each machine in the cluster in order to start the solvers on the remote machines.

Even though the program was "only written in Python" it produced very impressive performance. For further speed, we experimented a little with ctypes, which is a Python 2.5 library for making easy connections from Python to a shared library (DLL) written in a language like C or Fortran. This is an amazing tool, but it is limited to what you can do with library function calls. Boost.python allows you to create C++ objects that maintain their state, and this looks quite powerful. Both approaches provide excellent speed optimizations so you can do rapid development in Python, then profile your solution and easily optimize the bottlenecks in C or C++ (but this is a subject for another article).

Before going to Twisted, we tried the parallelization library **pp**, but this turned out to have more overhead than Twisted; the latter was noticeably faster (**pp** just does the creation and distribution of tasks for you, so perhaps if it was re-implemented using Twisted it would be faster).

Adobe's open-source Blaze provides AMF for Java, so you could add Twisted support to talk from Python to Java modules. However, Blaze is designed for creating web applications and requires you to host your Java on an application server, so the setup can seem to be overkill on a single machine or cluster. XML-RPC is often a better choice to talk to Java modules for this scenario, because the Apache project provides good support for Java XML-RPC, and XML-RPC is generally more universal. Twisted also has good XML-RPC support so it's easy to talk to Java from Python this way.

Because the new Google App Engine is based on Python programming, this framework might be a good way to put all those CPUs to work -- but I don't know the constraints of the App Engine so I can't be sure (for example, do they have Twisted on it or can you install it?).

Consulting

The speed of development in Python is deeply satisfying; in just a few days it's possible to get something working that would require weeks or months in other languages. This is also true when building user interfaces in Flex -- one can rapidly build the UI using Flex Builder and quickly wire it into your Python application using Twisted. The parallelism shown here, along with the availability of ctypes and Boost.python means you can easily get the performance you need without sacrificing Python's development speed.

I'd like to do more work like this; if you're interested you can contact me.

Talk Back!

Have an opinion? Readers have already posted 9 comments about this weblog entry. Why not add yours?

RSS Feed

If you'd like to be notified whenever Bruce Eckel adds a new entry to his weblog, subscribe to his RSS feed.

About the Blogger

Bruce Eckel (www.BruceEckel.com) provides development assistance in Python with user interfaces in Flex. He is the author of Thinking in Java (Prentice-Hall, 1998, 2nd Edition, 2000, 3rd Edition, 2003, 4th Edition, 2005), the Hands-On Java Seminar CD ROM (available on the Web site), Thinking in C++ (PH 1995; 2nd edition 2000, Volume 2 with Chuck Allison, 2003), C++ Inside & Out (Osborne/McGraw-Hill 1993), among others. He's given hundreds of presentations throughout the world, published over 150 articles in numerous magazines, was a founding member of the ANSI/ISO C++ committee and speaks regularly at conferences.