Exploring Parallel and Distributed Programming Models with River

Greg Benson and Alex Fedosov
October 25, 2006
River

- Parallel and distributed programming system
- Python-based
- River core API
- River extensions provide different models
- Reliable virtual resources
Why Parallel?

- Apps must get faster
- Multi-core processors
- Clusters
- Underutilized resources
- The tipping point
Why River?

- Tackle hard usability issues
- Simple programming and deployment
- Drag and drop computations
- Reliable execution
Low Hanging Fruit

How do I code that "easy" parallelism?
Need a Simple Platform

Ahhh, there it is!

Developer

River
Higher Hanging Fruit
The 30 Minute Challenge

- You have Linux, Mac, and Windows boxes
- In 30 minutes can you do the following?
  - Write a program that will distribute an array to remote machines
  - Compute partial sums on each machine in parallel
  - Retrieve and add partial sums
Array Sum in Python

```
$ cat array_sequential.py
a = range(0,1000)
s = sum(a)
print s

$ python array_sequential.py
499500
```
Array Sum in River

$ cat array_river.py
machines = connect(4)
chunks = [range(i, i+250) for i in range(0,1000,250)]
results = forkjoin(machines, sum, chunks)
s = sum(results)
print s

$ river array_river.py
discovered 4 VMs
499500
Really, is that it?

- The *small* fine print
- Python is required (not for long)
- Firewall must not block River port
- Need to start River on each machine
Drag and Drop Computing

- Parallel computation is notoriously fragile
- Want to easily manage a computation:
  - Suspend and migrate
  - Add or remove machines
- River provides a completely virtual run-time environment.
Reliable Execution

- Most parallel and distributed languages ignore fault tolerance
- The River virtual environment allows for seamless redundancy
- Replicated processes
- Application independent checkpoints
Remainder of this Talk

- Exploring programming models in River
- Some River concepts
- Consider two models:
  - Trickle
  - rMPI
- How to implement models in River
River Concepts

- River Virtual Machine (RVM)
- Python + River Core
- Virtual Resource (VR)
  - A container of data, code, a stack, and a thread
- VRs run on top of RVMs
Virtual Resources

- A VR is a Python class
- Each VR has a unique id (UUID)
- Each VR has a message queue
- VRs can send messages to each other
Trickle: Extended Python

- Some Python features:
  - Object oriented, dynamically typed
  - Rich module library (sockets, xml, etc.)
- Trickle
  - Extends Python and River to enable remote access to multiple Python VMs
  - Push and use objects, code, data on RVMs
Trickle Execution

RVM (Initiator) → inject/use → VR

RVM (VR) → VR

RVM (VR) → VR

RVM (VR) → VR
Trickle Programming

- Connecting to remote VRs
  - `vrlist = connect(count)`
- Injecting code/data
  - `inject(vr, element)`
- Invocation
  - `vr.foo(args)`
Parallel Invocation

- Fork/join semantics
- Asynchronously fork code on remote VRs
- Join synchronizes with forked code

```python
def foo(x,y):
    return x + y

vrlist = connect(2)
for r in vrlist: inject(r, foo)
results = forkjoin(vrlist, foo, [[1,2],[3,4]])
```
Alternate Interfaces

‣ forkall() and joinall()

hlist = forkall(vrlist, foo)
results = joinall(hlist)

‣ fork() and join()

hlist = [fork(vr, foo) for vr in vrlist]
results = [join(h) for h in hlist]
Example: Integration

\[ \int_a^b f(x) \, dx \]
def simpsonsrule( func, a, b, TOL=1e-10 ):
    h = b - a
    old2 = old = h * ( func( a ) + func( b ) ) / 2.0
    count = 0
    while True:
        h = h / 2.0
        x, sum = a + h, 0
        while x < b:
            sum = sum + func( x )
            x = x + 2 * h
        new = old / 2.0 + h * sum
        new2 = ( 4 * new - old ) / 3.0
        if abs( new2 - old2 ) < TOL * ( 1 + abs( old2 ) ): return new
        old = new
        old2 = new2
        count = count + 1
Trickle Version

```python
import math

def simpsonsrule(func, a, b, TOL=1e-10):
    ...
    f = math.sin

vrlist = connect(4)
for r in vrlist:
    inject(r, simpsonsrule)
chunks = [(f, 0, 10), (f, 10, 20), (f, 20, 30), (f, 30, 40)]
results = forkjoin(vrlist, simpsonsrule, chunks)

print sum(results)
print simpsonsrule(f, 0, 40)
```

Output:

```
$ trickle simpsonsrule.py
trickle: discovered 4 VMs:
1.66693806166
1.66693806166
```

Heterogeneous Machines

import math

def simpsonsrule(func, a, b, TOL=1e-10):
    ...

f = math.sin
work = [(f, i, i+2) for i in range(0, 40, 2)]
results = []; hlist = []

vrlist = connect(4)
for r in vrlist: inject(r, simpsonsrule)

while True:
    while len(work) > 0 and len(vrlist) > 0:
        w = work.pop(); v = vrlist.pop()
        hlist += [fork(v, simpsonsrule, w[0], w[1], w[2])]

    if len(hlist) > 0:
        h, rv = join(hlist)
        results.append(rv); vrlist.append(h.vr)
    else:
        break

print sum(results)
print simpsonsrule(f, 0, 40)

Output:

$ trickle simpsonsrule.py
trickle: discovered 4 VMs:
1.66693806166
1.66693806166
Other Cool Trick(le)s

- Injection
- Functions, Classes, Data
- Can pass remote object references to VRs
- Local directory export
River MPI (rMPI)

- MPI (Message Passing Interface)
- Almost complete MPI 1.2 implementation
- Built on top of River Core
- Only 650 lines of Python!
- Easy to modify
  - E.g., optimize collective communication
rMPI Example

```python
from mpi import *

class Hello( mpi):
    def main( self):
        self.MPI_Init()
        rank = mpi_Rank()
        np   = mpi_Size()
        self.MPI_Comm_rank( MPI_COMM_WORLD, rank )
        self.MPI_Comm_size( MPI_COMM_WORLD, np )
        status = MPI_Status()

        recvbuf = [ 0.0 ]
        sendbuf = [ rank.value * 100.0 ]

        print 'Hello from rank %d' % ( rank.value )

        if rank.value == 0:
            for i in xrange( 1, np.value ) :
                self.MPI_Recv( recvbuf, 1, MPI_FLOAT, i, 0, MPI_COMM_WORLD, status )
                print 'From rank %d: %f' % ( i, recvbuf[0] )
        else:
            print 'Rank %d sending %f' % ( rank.value, sendbuf[0] )
            self.MPI_Send( sendbuf, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD )

        self.MPI_Finalize()
```
River Core API

- More powerful than Trickle API
- Deployment becomes programmer’s responsibility
- Communication via message passing
Super Flexible Packets

- SFP is a set of (attribute : value) pairs
- Provides a powerful mechanism for communication
- Any type can be a value (ints, strings, lists, objects)
- Messages can be received from the queue by matching on any attribute
- No need to define packet structure!
SFP in a Nutshell

Send simply by listing attribute-value pairs as arguments:

```
send(dest=someVR, tag='inputlist', input=[1,2,3,4,5])
```

Receive by specifying which attributes you want to match: (multiple attributes can be specified)

```
recv(src=otherVR) or...
recv(tag='inputlist') or even...
recv(src=someVR, tag='inputlist')
```

and of course, just simply

```
recv()
```
SFP Matching

recv(tag = 'result')

Message Queue

Super Flexible Packet
src : ...
dest : ...
attr1 : value1
attr2 : value2

Super Flexible Packet
src : ...
dest : ...
name : value

Super Flexible Packet
src : ...
dest : ...
tag : result

Super Flexible Packet
src : ...
dest : ...
message : hello
Core API Example

```python
from vr import VirtualResource

class Hello (VirtualResource):
    def vr_init(self):
        discovered = self.discover()
        allocated = self.allocate(discovered)
        deployed = self.deploy(allocated, module=self.__module__)
        self.vrlist = [vm['uuid'] for vm in deployed]

        return True

    def main(self):
        if self.parent is None:
            for vr in self.vrlist:
                msg = self.recv()
                print '%s says hello' % msg.myname
        else:
            self.send(dest=self.parent, myname=gethostname())
```

The River Implementation

- Run-time support for the River interface
- Main elements
  - Socket communication (TCP/UDP)
  - Connection pool/cache
  - SFP queue
  - Communication thread
  - Control VR
Work in Progress

- A River native distributed file system
- River state capture for checkpointing and migration
- RJAX - A browser-based River GUI
- A River debugger
- RVM status monitoring
Contributors

- Current students
  - Brian Hardie, Tony Ngo, Jennifer Reyes, Joseph Gutierrez

- Past students
  - Jean Bovet, Yiting Wu, Sorasak Konglertviboon, Gao Lin, Chris Fraschetti, Deniz Efendioglu