River

A Foundation for the Rapid Development of Reliable Parallel Programming Systems

August 17, 2007

Greg Benson and Alex Fedosov
What is River?

- Reliable Virtual Resources
- A Python framework for parallel and distributed programming
- Prototype parallel programming systems
- Write parallel Python programs
River Overview

- River Core
  - Discovery
  - Process naming and creation
  - Message passing
  - State management
- River Extensions
  - RPC/RMI, Trickle, MPI, MapReduce

SciPy 2007
River Benefits

- Small, easy to use core interface
- Written entirely in Python
- Dynamic typing for rapid prototyping
- Python goodies
  - Heterogeneous (Use Python as a VM)
  - State capture at language VM level
- Integrated checkpointing and migration
Motivation

- Parallel programming is still hard
- The future: more cores, larger clusters
- Apps will have to utilize multiple processors
- Apps will have to tolerate failures
- The quest:
  - Find the next set of programming models
  - Incrementally improve current models
Current Practice

- Design/development cycle (long)
  - Specify (perhaps by committee or group)
  - Prototype (Use C/C++/Java)
  - Use prototype to provide feedback
- Examples
  - MPI, X10, Fortress
- Early implementation decisions hard to undo
River Goals

- Extend Python’s rapid development capabilities to parallel systems
- Facilitate short design/implementation cycles
- Open up design space
- Enable prototypes to run on real HW
- Demonstrate scalability/feasibility
Remainder of Talk

- River Core
- River Extensions
  - Remote Access and Invocation
  - Trickle (simple task farming language)
  - River MPI (rMPI)
- Related and Future Work
River Concepts

- Virtual machines (VMs)
- Python + River Core
- Virtual resources (VRs)
  - Named with UUIDs
  - Code, data, thread, and message queue
- Discover/allocate/deploy
- Flexible code execution
Executing VRs

VM (Initiator)
- vr_init
- main

VM
- VR
- Q
- main

VM
- VR
- Q
- main

VM
- VR
- Q
- main

send()
deploy()
Super Flexible Messaging

Sending

\[
\begin{align*}
&\text{send}(\text{dest}=\text{VRID}, \text{text}='\text{hello}') \\
&\text{send}(\text{dest}=\text{VRID}, \text{tag}=\text{'inputlist'}, \text{items} = [1, 2, 3, 4]) \\
&\text{stk} = \text{Stack}(); \text{stk}.\text{push}(1); \text{stk}.\text{push}(2) \\
&\text{send}(\text{dest}=\text{VRID}, \text{data}=\text{stk})
\end{align*}
\]

Receiving (selective)

\[
\begin{align*}
&m = \text{recv}() \quad \# \text{Any message} \\
&m = \text{recv}(\text{tag}=\text{'inputlist'}) \quad \# \text{Specific attr and value} \\
&\text{print } m.\text{items}
\end{align*}
\]

\[
\begin{align*}
&m = \text{recv}(\text{tag}=\text{'inputlist'}, \text{items}=(\lambda x:\text{len}(x) > 1)) \\
&m = \text{recv}(\text{src}=\text{VRID}, \text{data}=\text{ANY}) \\
&\text{print } m.\text{data}.\text{pop}()
\end{align*}
\]
Simple River Program

```python
from socket import gethostname
from river.core.vr import VR

class simple(VR):
    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:
                m = self.recv(src=vr)
                print m.mynname
        else:
            self.send(dest=self.parent, myname=gethostname())
```

SciPy 2007
State Management

- Designed from the beginning
- Encapsulate local state in VR
- Only hooks to outside UUIDs
- Per-VR queues hold in-transit messages
- Transparent migration and checkpointing
- Internal and external support
Coordinated Checkpointing

Algorithm

- Freeze all remote VRs (preemptively)
- Allow in-flight messages to settle
- Write frozen state (VR + queue)
- Unfreeze all remote VRs

Mechanism is extensible

- State exclusion, diskless, app assisted, etc.
River Implementation

- One net thread, one Control VR
- Control handles VR creation, state
- TCP-based, connection caching (scalable)
- Broadcast-based discovery
- Super Flexible Messaging
- Queue matching
- Serialization (Pickle)
State Implementation

- Keep soft VR state separate from hard VR state
- Two VR classes: VR and VRI (internal)
- VR has a reference to host VRI
- Stackless: run VR as a tasklet in a VRI thread
- Generate atomic system calls (VRI calls)
- State capture: unlink VRI reference from VR
## River Complexity

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>River Core</td>
<td>3701</td>
</tr>
<tr>
<td>RAI (remote invocation)</td>
<td>531</td>
</tr>
<tr>
<td>Trickle (task farming)</td>
<td>375</td>
</tr>
<tr>
<td>rMPI</td>
<td>882</td>
</tr>
<tr>
<td>rMPI derived datatypes</td>
<td>246</td>
</tr>
<tr>
<td>rMPI non-blocking communication</td>
<td>441</td>
</tr>
<tr>
<td>rMPI optimized collectives</td>
<td>335</td>
</tr>
<tr>
<td>MapReduce</td>
<td>511</td>
</tr>
</tbody>
</table>
Remote Invocation

- Remote Access and Invocation (RAI)
- RPC, RMI, and remote data access
- Create and access functions, objects, data on remote VRs
- Built on top of the River core
- Unrestricted mode

```
r = RemoteVR(server, self)
print r.add(1,2,3)
```
Trickle

- Simple task farming language
- Put code/data on remote VMs
- Execute sequentially or in parallel

```python
def foo(x):
    return x + 10

vmlist = connect()
inject(vmlist, foo)
results = [vm.foo(10) for vm in vmlist]
print results
```

```
$ trickle exsimple.py
[trickle: discovered 4 VMs]
[20, 20, 20, 20]
```
Parallel Invocation

- **Fork/join paradigm**

  ```python
  def foo(x):
      return x + 10
  
vmlist = connect()
inject(vmlist, foo)
hlist = fork(vmlist, foo, range(len(vmlist)))

  print join(hlist)
  ```

- **Dynamic scheduling**

  ```python
  def foo(x):
      return sum(x)
  
vmlist = connect()
inject(vmlist, foo)
results = forkwork(vmlist, foo, range(100), chunksize=10)

  print sum(results)
  ```
def wordcount(files):
    # count words in given files (13 lines)
def mergecounts(dlist):
    # merge resulting count dictionaries (8 lines)

# Command line processing (9 lines)

vmlist = connect(n)
inject(vmlist, wordcount)
rlist = forkwork(vmlist, wordcount, files, chunksize=cs)
final = mergecounts(rlist)
River MPI (rMPI)

- Partial implementation of MPI 1.2 in River
- Most p-to-p and collectives
- Easy to read and understand
- Models C MPI interface
- Experiment with different algorithms
- Use inheritance to add functionality
- Derived data types, non-blocking comm
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:
            # if not rank 0, send value to rank 0
            print 'Rank %d sending %f' % ( rank.value, sendbuf[0] )
            self.MPI_Send( sendbuf, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD )

        self.MPI_Finalize()
rMPI Conjugate Gradient

Data Size: 1024x1024 doubles

- Python/rMPI (GigE)
- C/LAM (GigE)

Penguin Cluster
AMD Opterons (Dual dual-core) 2.0GHz
4 GB RAM, GigE
One VM/Process per node
def MPI_Barrier(self, comm):
    root = mpi_Rank(0)
    sendbuf = [1]; recvbuf = [0]
    msgUid = self.getMsgUid()
    status = MPI_Status()

    i = self.rank.value
    p = comm.size()
    steps = int(math.ceil(math.log(p, 2)))
    for k in xrange(steps):
        dest = (i + 2**k) % p
        src = (i - 2**k + p) % p
        self.MPI_Send(sendbuf, 1, MPI_INT, mpi_Rank(dest), msgUid, comm)
        self.MPI_Recv(recvbuf, 1, MPI_INT, mpi_Rank(src), msgUid, comm, status)
    return MPI_SUCCESS
Experience

- **Trickle**
  - Design: about 2 days
  - First implementation: about 1 evening
  - Refinements: easy (e.g., dynamic scheduling)

- **rMPI**
  - First implementation: about 1 month
  - Used in a grad parallel computing class
Related Work

- Python
  - PyMPI, MYMPI, PYRO, Twisted, others
  - IPython (Trickle-like functionality)
- VM level checkpointing and migration
- Many Java-based implementations
Future Work

- Refine the River Core
- Further experimentation with rMPI and Trickle
- Evaluate different checkpointing schemes
- Develop new extensions: GAS-like language
- Automate the translation of a River implementation into a C/Java implementation
River Website and Release

http://www.cs.usfca.edu/river

- River papers
- River overview
- Super Flexible Messaging (SFM)
- Trickle
- Download River and Extensions