CMTS Cyberinfrastructure

Michael Wilde - wilde@mcs.anl.gov
http://swift-lang.org
Cyberinfrastructure Challenges

- Expressing multi-scale workflow protocols
- Leveraging and expressing parallelism
- Moving, processing and managing data
- Traversing networks and security
- Diverse environments: schedulers, storage...
- Handling system failures
- Tracking what was done: where, when, how, who?
Cyberinfrastructure Tasks

- Enable easy of use of multiple distributed large-scale systems
- Reduce effort of writing parallel simulation and analysis scripts
- Record and organize provenance of computations
- Annotate datasets to facilitate sharing, collaboration, and validation
- Reduce the cost of modeling, both in terms of wall clock time and computer time.
Swift Parallel Scripting Language: A Core Component of CMTS Cyberinfrastructure

- **Swift** is a parallel scripting language
  - Composes applications linked by files
  - Captures protocols and processes as a CMTS library

- Easy to write: a simple, high-level language
  - *Small Swift scripts can do large-scale work*

- Easy to run: on clusters, clouds, supercomputers and grids
  - *Sends work to XSEDE, campus, and Amazon resources*

- Automates solutions to four hard problems
  - *Implicit parallelism*
  - *Transparent execution location*
  - *Automated failure recovery*
  - *Provenance tracking*
Using *Swift* in the Larger Cyberinfrastructure Landscape

Swift’s runtime system supports and *aggregates* diverse, distributed execution environments.
Expressing CMTS Algorithms in Swift
Translating from a shell script of 1300 lines into code like this:

```swift
(...) EnergyLandscape(...) 
{
    loadPDB (pdbcode="3lzt");
    createSystem (LayerOfWater=7.5,
                   IonicStrength=0.15);
    preequilibrate (waterEquilibration_ps=30,
                    totalEquilibration_ps=1000);
    runMD (config="EL1", tinitial=0, tfinal=10,
           frame_write_interval=1000);
    postprocess(...);
    forcematching(...);
}
```

...to create a well-structured library of scripts that automate workflow-level parallelism
CMTS Cyberinfrastructure Architecture

- A challenging “breakthrough” project
- Integration & evaluation underway
- Full implementation by the end of Phase II

0: Develop script

1: Run script(EL1.trj)

2. Lookup file
   name=EL1.trj
   user=Anton
   type=trajectory

3: Transfer inputs

4: Run app

5: Transfer results

6: Update catalogs

Compute Facilities

Swift

Researchers

External collaborators

CMTS Collaboration Catalogs

Files & Metadata

Provenance

Storage locations

Researchers

External collaborators

CMTS Collaboration Catalogs
When do you need Swift?
Typical application: protein-ligand docking for drug screening

$O(10)$ proteins implicated in a disease

$O(100K)$ drug candidates

Tens of fruitful candidates for wetlab & APS

Work of M. Kubal, T.A.Binkowski, And B. Roux
Numerous many-task applications

- Simulation of super-cooled glass materials
- Protein folding using homology-free approaches
- Climate model analysis and decision making in energy policy
- Simulation of RNA-protein interaction
- Multiscale subsurface flow modeling
- Modeling of power grid for OE applications

A-E have published science results obtained using Swift
Nested parallel prediction loops in Swift

1. Sweep( )
2. {
3.     int nSim = 1000;
4.     int maxRounds = 3;
5.     Protein pSet[ ] <ext; exec="Protein.map">;
6.     float startTemp[ ] = [ 100.0, 200.0 ];
7.     float delT[ ] = [ 1.0, 1.5, 2.0, 5.0, 10.0 ];
8.     foreach p, pn in pSet {
9.         foreach t in startTemp {
10.            foreach d in delT {
11.                ItFix(p, nSim, maxRounds, t, d);
12.            }
13.        }
14.    }
15.}
16. Sweep();

10 proteins x 1000 simulations x 3 rounds x 2 temps x 5 deltas = 300K tasks
Programming model: all execution driven by parallel data flow

```c
(int r) myproc (int i) {
    j = f(i);
    k = g(i);
    r = j + k;
}
```

- f() and g() are computed in parallel
- myproc() returns r when they are done

- This parallelism is automatic
- Works recursively throughout the program’s call graph
Programming model: Swift in a nutshell

- All expressions are computed in parallel, throttled by various levels of the runtime system
- Simple data type model: scalars (boolean, int, float, string, file, external) and collections (arrays and structs)
- ALL data atoms and collections have “future” behavior
- A few simple control constructs
  - if, switch, foreach, iterate
- A growing library
  - tracef(), strcat(), regexp(), readData(), writeData()
Swift 2.0

- Motivation for 2.0
  - Scalability: 1M tasks/sec vs 500 – goal has been reached for basic tests
  - Richer programming model, broader spectrum of application
  - Extensibility and Maintainability

- Convergence issues
  - Some array closing; for loop extensions; library cleanup; mapper models
  - Data marshalling/passing for in-memory leaf functions

- Information and downloads
  - http://exm.xstack.org
Encapsulation enables distributed parallelism

Encapsulation is the key to transparent distribution, parallelization, and automatic provenance capture.
app( ) functions specify cmd line argument passing

To run:
```
psim -s 1ubq.fas -pdb p -t 100.0 -d 25.0 >log
```

In Swift code:
```
app (PDB pg, File log) predict (Protein seq, Float t, Float dt)
{
    psim "-c" "-s" @pseq.fasta "-pdb" @pg
        "-t" temp "-d" dt;
}
```

Protein p <ext; exec="Pmap", id="1ubq">;
PDB structure;
File log;

(structure, log) = predict(p, 100., 25.);
foreach sim in [1:1000] {
    (structure[sim], log[sim]) = predict(p, 100., 25.);
}
result = analyze(structure)
Dataset mapping example: fMRI datasets

On-Disk Data Layout

- **Study**
  - **Group**
    - **Subject**
      - **Volume**

- **Group**
  - **Subject**
    - **Volume**

- **Run**
  - **Volume**

Swift’s in-memory data model

- **Study**
  - **Group**
    - **Subject**
      - **Volume**

- **Group**
  - **Subject**
    - **Run**
      - **Volume**

- **Run**
  - **Volume**

- **Volume**
  - **Image**
  - **Header**

Mapping function or script
**Metadata**

```xml
<Metadata>
    <tag /d1/d2/f3> owner=asinitskiy group=cmts-chem
        create-date=2013.0415
        type=trajectory-namd
        state=published
        note="trajectory for UCG case 1"
        molecule=1ubq domain=loop7, bead=u2
    </tag>
    <tag /d1/d2/f0> owner=jdama group=cmts-chem
        create-date=2013.0412
        type=trajectory-namd
        state=validated
        note="trajectory for UCG case 2"
        molecule=clathrin, domain=loop1, bead=px
    </tag>
</Metadata>
```
## Metadata

Query ==> \((\text{owner}=\text{asinitiskiy}) \text{ and } (\text{molecule}=1\text{ubq})\)

<table>
<thead>
<tr>
<th>dataset_id</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/d1/d2/f4</td>
<td>owner</td>
<td>asinitiskiy</td>
</tr>
<tr>
<td></td>
<td>group</td>
<td>cmts-chem</td>
</tr>
<tr>
<td></td>
<td>create-date</td>
<td>2013.0416.12:20:29.456</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>trajectory-namd</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>published</td>
</tr>
<tr>
<td></td>
<td>note</td>
<td>trajectory for case 1</td>
</tr>
<tr>
<td></td>
<td>molecule</td>
<td>2ubq</td>
</tr>
<tr>
<td></td>
<td>domain</td>
<td>loop7</td>
</tr>
<tr>
<td></td>
<td>bead</td>
<td>dt</td>
</tr>
</tbody>
</table>
Swift information

- **Swift Web:**
  - http://www.ci.uchicago.edu/swift or newer: http://swift-lang.org

- **Swift User Guide:**
Running the CMTS “cyber” tutorials

# pick up latest changes if instructed:
cd /project/cmtsworkshop
cp –rp tutorial/cyber $USER/cyber

cd $USER/cyber
source ./setup.sh

cd basic_swift
cat README

cd part01
cat README
swift p1.swift # etc etc etc
Summary of the CMTS Cyberinfrastructure Approach

- Streamlining scientist / computer interaction:
  - Large datasets (e.g. MD trajectories), fast distributed storage
  - Chemically important metadata: force field, bound ions, …
  - Save time locating, accessing, moving, and sharing data
  - Scheduling, data management, authentication, site-specific dependencies
  - Automatic recovery of failed computations
  - Save time by load balancing, leveraging more parallel resources, …
  - Benefit: chemists & biologists can focus on science and less on the mechanics of computation.

- Facilitating scientific collaboration:
  - Understandable code, flexibility for future development
  - Find provenance and tag data and procedures for reuse and sharing
  - Facilitate collaboration by sharing data, libraries, protocols
  - Wide range of researchers able to use the environment.
  - Benefit: cyberinfrastructure enables collaborations not otherwise possible
Swift is a parallel scripting system for grids, clouds and clusters
  – for loosely-coupled applications - application and utility programs linked by exchanging files

Swift is easy to write: simple high-level C-like functional language
  – Small Swift scripts can do large-scale work

Swift is easy to run: contains all services for running Grid workflow - in one Java application
  – Untar and run – acts as a self-contained Grid client

Swift is fast: uses efficient, scalable and flexible “Karajan” execution engine.
  – Scaling close to 1M tasks – .5M in live science work, and growing

Swift usage is growing:
  – applications in neuroscience, proteomics, molecular dynamics, biochemistry, economics, statistics, and more.

Try Swift! www.ci.uchicago.edu/swift and www.mcs.anl.gov/exm
Swift: A language for distributed parallel scripting

Michael Wilde\textsuperscript{a,b,*}, Mihael Hategan\textsuperscript{a}, Justin M. Wozniak\textsuperscript{b}, Ben Clifford\textsuperscript{d}, Daniel S. Katz\textsuperscript{a}, Ian Foster\textsuperscript{a,b,c}

\textsuperscript{a} Computation Institute, University of Chicago and Argonne National Laboratory, United States
\textsuperscript{b} Mathematics and Computer Science Division, Argonne National Laboratory, United States
\textsuperscript{c} Department of Computer Science, University of Chicago, United States
\textsuperscript{d} Department of Astronomy and Astrophysics, University of Chicago, United States

Abstract

Scientists, engineers, and statisticians must execute domain-specific application programs many times on large collections of file-based data. This activity requires complex orchestration and data management as data is passed to, from, and among application invocations. Distributed and parallel computing resources can accelerate such processing, but their use further increases programming complexity. The Swift parallel scripting language reduces these complexities by making file system structures accessible via language constructs and by allowing ordinary application programs to be composed into powerful parallel scripts that can efficiently utilize parallel and distributed resources. We present Swift’s implicitly parallel and deterministic programming model, which applies external applications to file collections using a functional style that abstracts and simplifies distributed parallel execution.