Introduction to Swift

Parallel scripting for distributed systems

Mike Wilde
wilde@mcs.anl.gov

Computation Institute, University of Chicago
and Argonne National Laboratory

www.ci.uchicago.edu/swift

Swift is…

- A language for writing scripts that:
  - Process **large collections** of **persistent** data
  - with large and/or complex sequences of **application programs**
  - on **diverse** distributed systems
  - with a high degree of **parallelism**
  - persisting over **long periods** of time
  - surviving infrastructure failures
  - and tracking the provenance of execution

A simple Swift script

```
type imagefile;  // Declare a “file” type.

app (imagefile output) flip(imagefile input) {
{
    convert "-rotate” 180 @input @output ;
}

imagefile stars <"orion.2008.0117.jpg">;
imagefile flipped <"output.jpg">;

flipped = flip(stars);
```

Parallelism via foreach { }

```
type imagefile;  // Declare a “file” type.

(imagefile output) flip(imagefile input) {
  app {
    convert "-rotate" "180" @input @output;
  }
}

imagefile observations[ ] <simple_mapper; prefix=“orion”>;
imagefile flipped[ ]      <simple_mapper; prefix=“orion-flipped”>;
```

Name outputs based on inputs

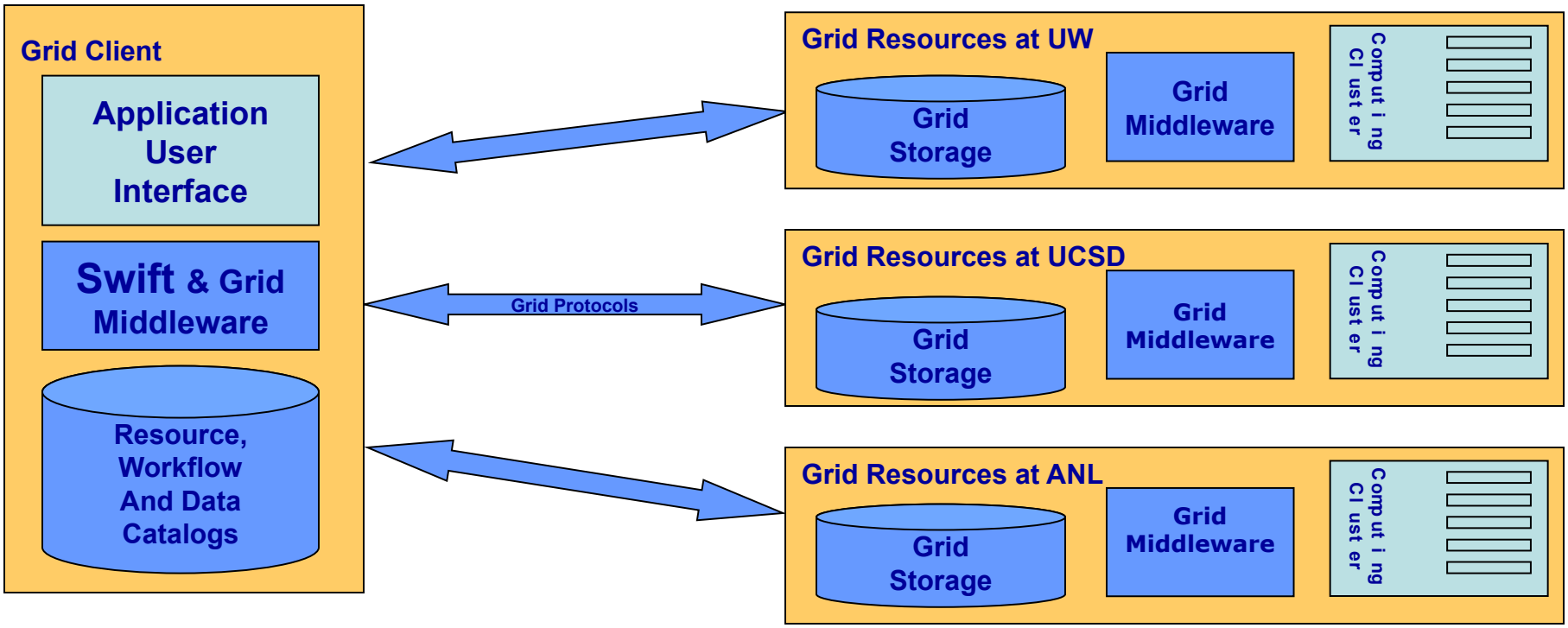
```
foreach obs,i in observations {
  flipped[i] = flip(obs);
}
```

Process all dataset members in parallel

Why script in Swift?

- Write scripts that are high-level, simpler, and location-independent: run anywhere
  - Higher level of abstraction makes a workflow script more portable than “ad-hoc” scripting
- Coordinate execution on *many* resources over *long time periods*
  - This is very complex to do manually – Swift automates it
- Enables restart of long running scripts
  - Swift tracks jobs in a parallel script completed

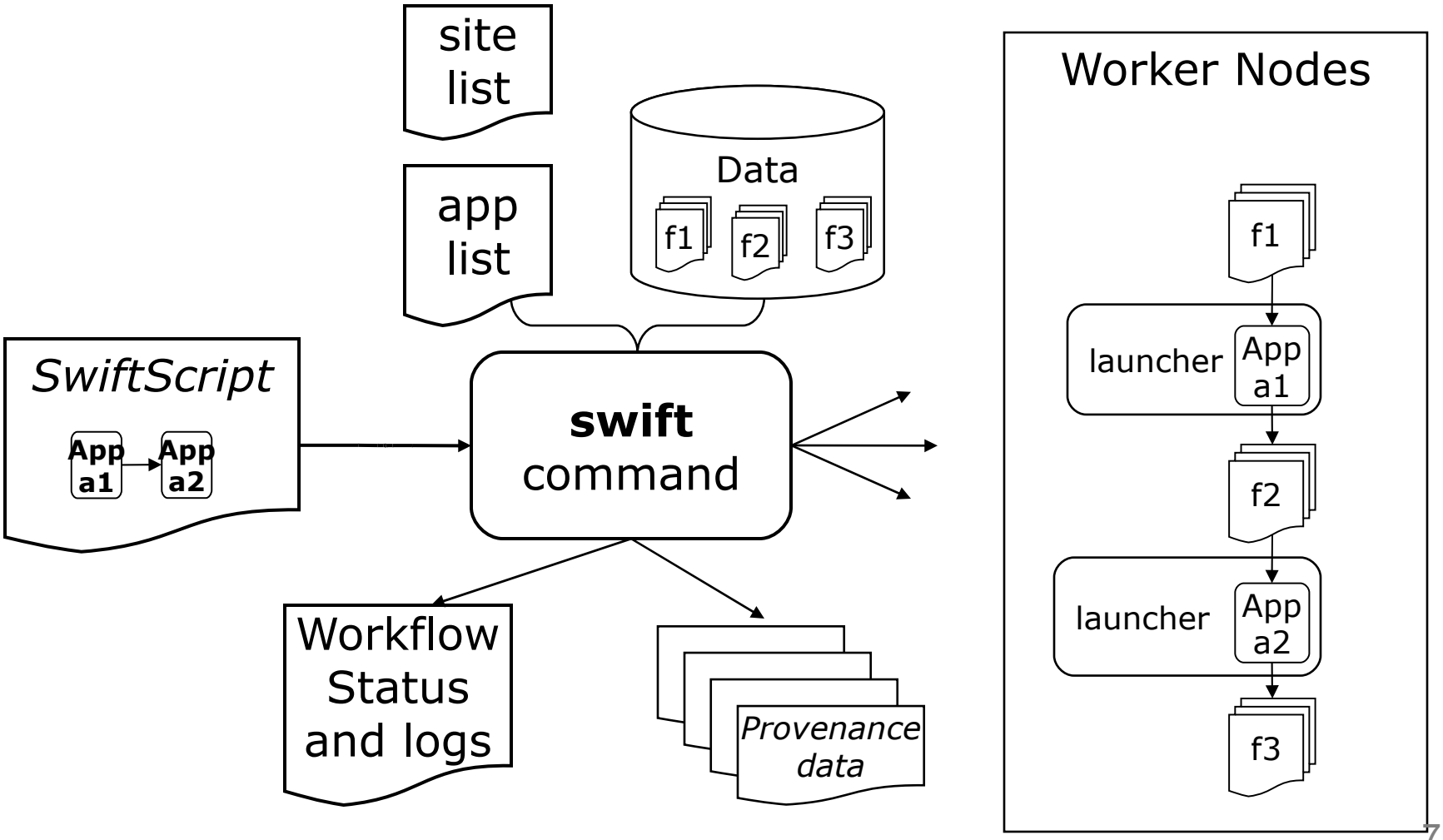
Swift runs on Cluster and Grids



Swift runs on the grid client or "submit host"

- Sends jobs to one or more grid sites using GRAM and Condor-G
- Sends files to and from grid sites using GridFTP
- Directory to locate grid sites and services: (ReSS)
- Can also run on local hosts, or directly on a local cluster
- Can overlay a faster scheduling mechanism (Coasters, Falkon)

Using Swift


site list
app list
Data
f1
f2
f3
SwiftScript
App a1
App a2
swift command
Workflow Status and logs
Provenance data
Worker Nodes
f1
launcher
App a1
f2
launcher
App a2
f3

Swift programs

- A Swift script is a set of **functions**
  - Atomic functions wrap & invoke application programs
  - Composite functions invoke other functions
- Data is **typed** as composable arrays and structures of **files** and simple scalar types (int, float, string)
- Collections of **persistent file structures** are **mapped** into this data model as arrays and structures
- Members of datasets can be processed in **parallel**
- Statements in a procedure are executed in **data-flow** dependency order and concurrency
- Variables are **single assignment**
- **Provenance** is gathered as scripts execute

Running swift

- Fully contained Java grid client
- Can test on a local multicore machine
- Can run directly on PBS or Condor clusters
- Runs on multiple clusters over Grid interfaces
  - Uses GRAM2 (or GRAM5)
  - Can use Condor-G if that's on submit host

Swift case study: Protein Folding

```
ype RamaMap;

ype RamaIndex;

ype SecSeq;

ype Fasta;


ype Protein {

   RamaMap map;

   RamaIndex index;

   SecSeq secseq;

   Fasta fasta;

   PDB native;


ype PDB;

ype OOPSLog;
```

Swift case study: Protein Folding

```
app (ProtGeo pgeo) predict (Protein pseq)
{
    PSim @pseq.fasta @pgeo;
}
```

```
(ProtGeo pg[ ]) doRound (Protein p, int n) {
  foreach sim in [0:n-1] {
    pg[sim] = predict(p);
  }
}

Protein p <ext; exec="Pmap", id="1af7">;
ProtGeo structure[ ];
int nsim = 10000;
structure = doRound(p, nsim);
```

Swift case study: Protein Folding

```
1  (ProtSim psim[ ]) doRoundCf (Protein p, int n, PSimCf cf) {
2     foreach sim in [0:n-1] {
3        psim[sim] = predictCf(p, cf.st, cf.tui, cf.coeff );
4     }
5  }
```

```
1  (boolean converged) analyze( ProtSim prediction[ ], int r, int numRounds)
2  {
3    if( r == (numRounds-1) ) {
4       converged = true;
5    }
6    else {
7       converged = false;
8    }
9  }
```

Swift case study: Protein Folding

```
1.  ItFix( Protein p, int nsim, int maxr, float temp, float dt)
2.  {
3.     ProtSim prediction[ ][ ];
4.     boolean converged[ ];
5.     PSimCf config;

1.     config.st = temp;
2.     config.tui = dt;
3.     config.coeff = 0.1;

1.     iterate r {
2.        prediction[r] =
3.           doRoundCf(p, nsim, config);
4.        converged[r] =
5.           analyze(prediction[r], r, maxr);
6.      } until ( converged[r] );
7.  }
```

Swift case study: Protein Folding

```
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, 8.0];
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. }

1.  Sweep();
```

The Variable model

- Single assignment:
  - Can only assign a value to a var once
  - This makes data flow semantics much cleaner to specify, understand and implement
- Variables are scalars or references to composite objects
- Variables are typed
- File typed variables are “mapped” to files

Data Flow Model

- This is what makes it possible to be location independent
- Computations proceed when data is ready (often not in source-code order)
- User specifies DATA dependencies, doesn’t worry about sequencing of operations
- Exposes maximal parallelism

Swift statements

- Var declarations
  - Can be mapped
- Type declarations
- Assignment statements
  - Assignments are type-checked
- Control-flow statements
  - if, switch, foreach, iterate
- Function declarations

Passing other scripts as data

- When running scripts in other languages from Swift, the target language interpreter can be the “app” executable (eg: app { R … })
- Powerful technique for running scripts in:
  - sh, bash
  - Perl, Python, Tcl
  - R, Octave
- These are often pre-installed at known places
  - No application installation needed
- Need to deal with library modules manually

Data Management

- Directories and management model
  - local dir, storage dir, work dir
  - caching within workflow
  - reuse of files on restart
- Makes unique names for: jobs, files, wf
- Can leave data on a site
  - For now, in Swift you need to track it
  - In Pegasus (and VDS) this is done automatically

Mappers and Vars

- Vars can be “file valued”
- Many useful mappers built-in, written in Java to the Mapper interface
- “Ext”ernal mapper can be easily written as an external script in any language

Mapping outputs based on input names

```
type pcapfile;
type angleout;
type anglecenter;

app (angleout ofile, anglecenter cfile) angle4 (pcapfile ifile)
{  angle4 @ifile @ofile @cfile; }

// Map inputs based on patterns
pcapfile pcapfiles[ ]<filesys_mapper; prefix="pc", suffix=".pcap">;

angleout    of[ ] <structured_regexp_mapper;
                   source=pcapfiles, match="pc(.*)\.pcap",
                   transform="_output/of/of\1.angle">;
anglecenter cf[ ] <structured_regexp_mapper;
                   source=pcapfiles,match="pc(.*)\.pcap",
                   transform="_output/cf/cf\1.center">;

foreach pf,i in pcapfiles {
  (of[i],cf[i]) = angle4(pf);
}
```

Name outputs based on input

Coding your own “external” mapper

```
awk <angle-spool-1-2 '
BEGIN {
  server="gsiftp://tp-osg.ci.uchicago.edu//disks/ci-gpfs/angle";
 }
{   printf "[%d] %s/%s\n", i++, server, $0    }’

$ cat angle-spool-1-2
spool_1/anl2-1182294000-dump.1.167.pcap.gz
spool_1/anl2-1182295800-dump.1.170.pcap.gz
spool_1/anl2-1182296400-dump.1.171.pcap.gz
spool_1/anl2-1182297600-dump.1.173.pcap.gz
…
$ ./map1 | head
[0] gsiftp://tp-osg.ci.uchicago.edu//disks/ci-gpfs/angle/spool_1/anl2-1182294000-
      dump.1.167.pcap.gz
[1] gsiftp://tp-osg.ci.uchicago.edu//disks/ci-gpfs/angle/spool_1/anl2-1182295800-
      dump.1.170.pcap.gz
[2] gsiftp://tp-osg.ci.uchicago.edu//disks/ci-gpfs/angle/spool_1/anl2-1182296400-
      dump.1.171.pcap.gz
      …
```

Site selection and throttling

- Avoid overloading target infrastructure
- Base resource choice on current conditions and real response for *you*
- Things are getting more automated.

Clustering and Provisioning

- Can cluster jobs together to reduce grid overhead for small jobs
- Can use a “provisioner” to hold processors
  - Coasters (built in) or Falkon (research tool)
  - for better performance, lower overhead, less scheduler delays
- Can use a provider to go straight to a cluster (PBS, Condor)

Testing and debugging techniques

- Debugging
  - Trace and print statements
  - Put logging into your wrapper
  - Capture stdout/error in returned files
  - Capture glimpses of runtime environment
  - Kickstart data helps understand what happened at runtime
  - Reading/filtering swift client log files
  - Check what sites are doing with local tools - condor_q, qstat
- Log reduction tools tell you how your workflow behaved

Other Workflow Style Issues

- Expose or hide parameters
- One atomic, many variants
- Expose or hide program structure
- Driving a parameter sweep with readdata() - reads a csv file into struct[].
- Swift is not a data manipulation language - use scripting tools for that

Swift: Summary

- Clean separation of logical/physical concerns
  - XDTM specification of logical data structures

+ Concise specification of parallel programs
  - SwiftScript, with iteration, etc.

+ Efficient execution (on distributed resources)
  - **Karajan+Falkon**: Grid interface, lightweight dispatch, pipelining, clustering, provisioning

+ Rigorous provenance tracking and query
  - Records provenance data of each job executed

→ **Improved usability and productivity**
  - Demonstrated in numerous applications

To learn more…

- www.ci.uchicago.edu/swift
  - Quick Start Guide:
    - http://www.ci.uchicago.edu/swift/guides/quickstartguide.php
  - User Guide:
    - http://www.ci.uchicago.edu/swift/guides/userguide.php
  - Introductory Swift Tutorials:
    - http://www.ci.uchicago.edu/swift/docs/index.php

Acknowledgments

- Swift effort is supported in part by NSF grants OCI-721939, and PHY-636265, NIH DC08638, and the UChicago/Argonne Computation Institute
- The Swift team:
  - Ben Clifford, Ian Foster, Mihael Hategan, Sarah Kenny, Mike Wilde, Zhao Zhang, Yong Zhao
- Java CoG Kit used by Swift developed by:
  - Mihael Hategan, Gregor Von Laszewski, and many collaborators
- User contributed workflows and application use
  - U. Chicago Open Protein Simulator Group (Drs. Freed & Sosnick); U.Chicago Radiology and Human Neuroscience Lab, (Dr. S. Small)