The Future of Scientific Workflow

Michael Wilde wilde@anl.gov
Argonne National Laboratory
and The University of Chicago

Collaborators in this vision of workflow

Timothy Armstrong, University of Chicago
Justin Wozniak, Argonne
Ketan Maheshwari, Argonne
Zhao Zhang, UChicago
Mihael Hategan, UChicago and UCDavis
Scott J. Krieder, Illinois Institute of Technology
David Kelly, University of Chicago
Yadu Nand Babuji, University of Chicago
Daniel S. Katz, University of Chicago
Ian T. Foster, University of Chicago and Argonne
Ioan Raicu, Illinois Institute of Technology
Michael Wilde, Argonne / University of Chicago

Supported in part by DOE-ASCR X-Stack, , NSF SI2, Argonne LDRD, NIH, ALCF, and Blue Waters/GLCPC

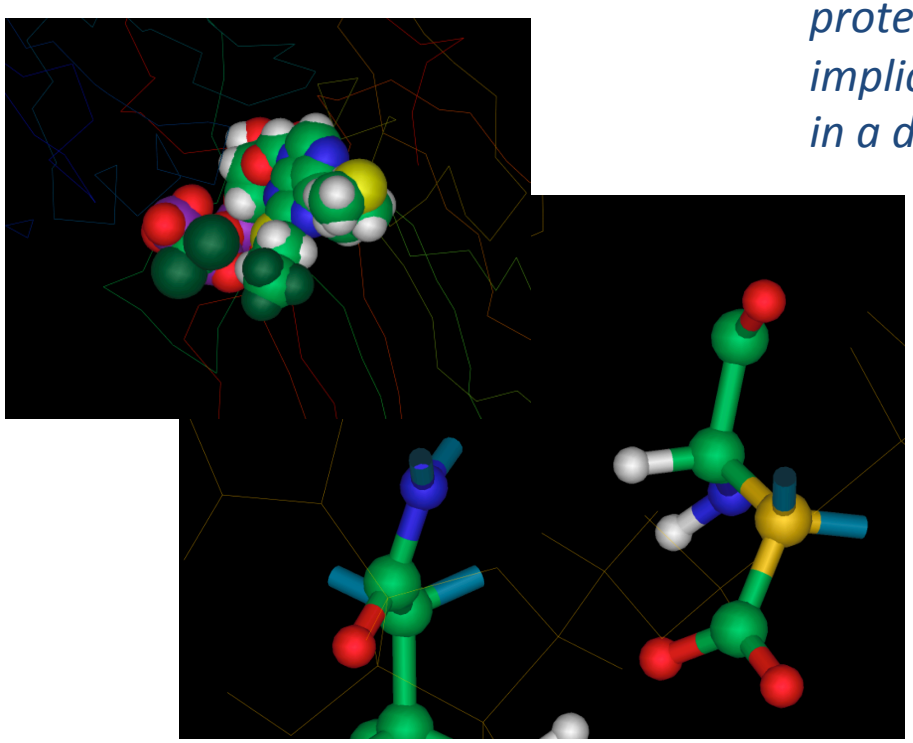
When do you need workflow ?

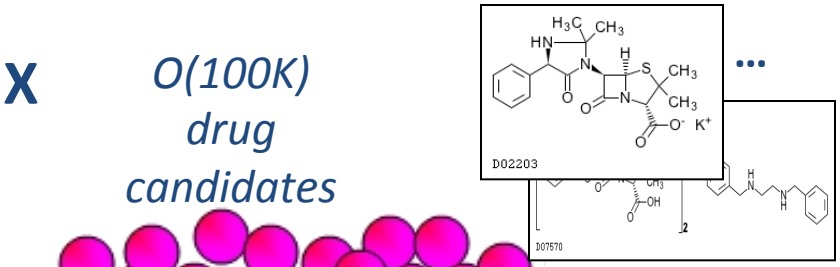
Sample application: protein-ligand docking for drug screening

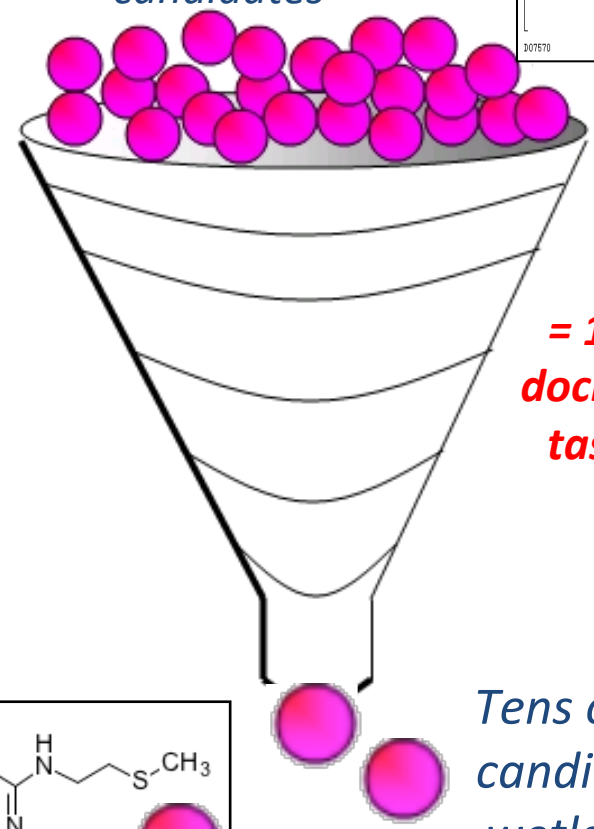
O(10) proteins implicated in a disease

X

O(100K) drug candidates

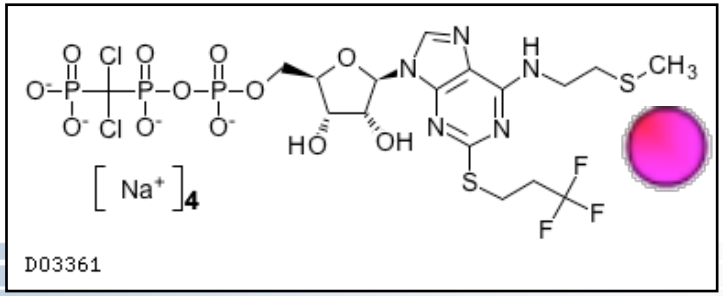






= 1M docking tasks

Tens of fruitful candidates for wetlab & APS



Work of M. Kubal, T.A.Binkowski, and B. Roux

How to code this?
Compact, portable scripting

Swift code excerpt:

```
foreach p, i in proteins {
   foreach c, j in ligands {
      (structure[i,j], log[i,j]) =
       dock(p, c, minRad, maxRad);
}
scatter_plot = analyze(structure)
```

To run:

```
swift –site tukey,blues dock.swift
```

Swift programming model

- Data types

```
int    i = 4;
int    A[];
string s = "hello world";
```

- Mapped data types

```
file image<"snapshot.jpg">;
```

- Structured data

```
image  A[]<array_mapper…>;
type protein {
   file pdb;
   file docking_pocket;
}
protein p<ext; exec=protein.map>;
```

- Conventional expressions

```
if (x == 3) {
    y = x+2;
    s = @strcat("y: ", y);
}
```

- Parallel loops

```
foreach f,i in A {
    B[i] = convert(A[i]);
}
```

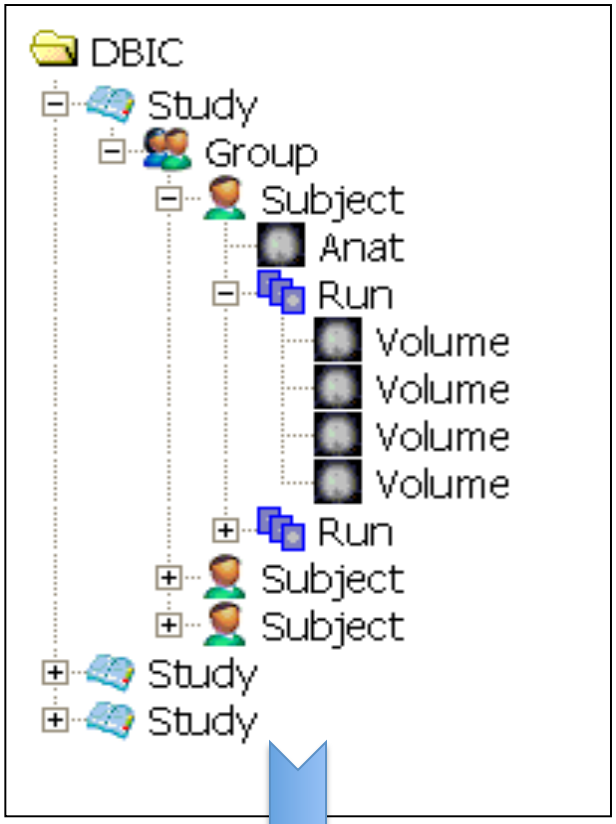
- Data flow

```
analyze(B[0], B[1]);
analyze(B[2], B[3]);
```

Swift: A language for distributed parallel scripting, J. Parallel Computing, 2011

Dataset mapping of structured image directory tree

On-Disk Data Layout



Mapping function or script

```
type Study {
        Group g[ ];
}
type Group {
        Subject s[ ];
}
type Subject {
        Volume anat;
        Run run[ ];
}
type Run {
        Volume v[ ];
}
type Volume {
        Image img;
        Header hdr;
}
```

In-memory data model

Benefits of a functional dataflow model

Makes parallelism more transparent

Implicitly parallel functional dataflow programming

Makes computing location more transparent

Runs your script on multiple distributed sites and diverse computing resources (desktop to petascale)

Makes basic failure recovery transparent

Retries/relocates failing tasks

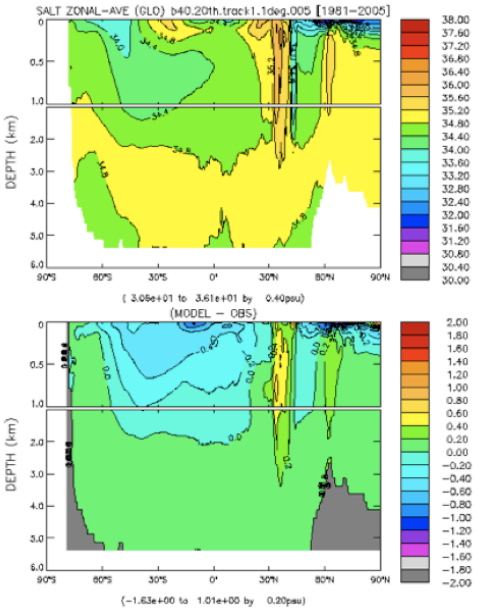
Can restart failing runs from point of failure

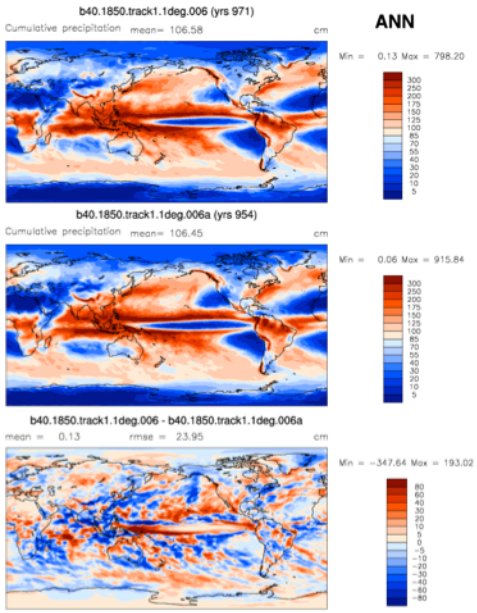
Functional model amenable to provenance capture

Tasks have recordable inputs and outputs

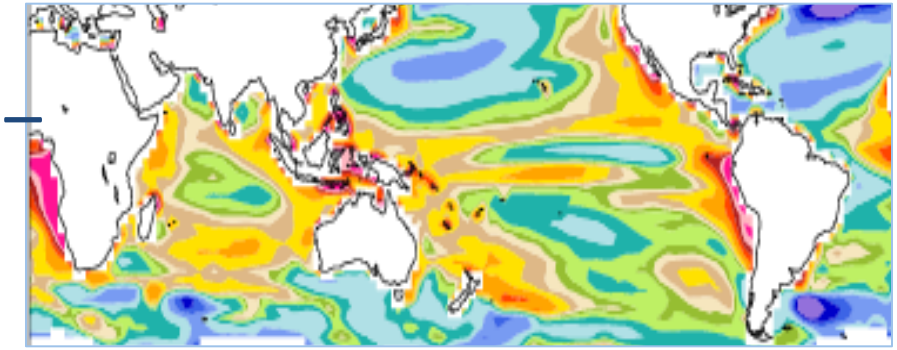
Analysis & visualization of high-resolution climate models

parVis





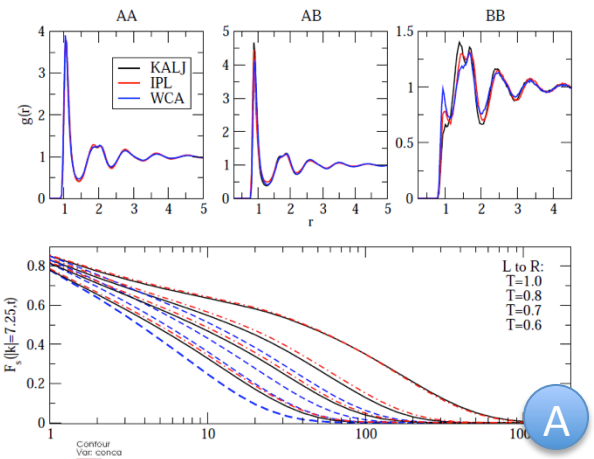
- Climate model diagnostics yield thousands of images and summary datasets
- Originally done by serial shells scripts
- Converted to swift dataflow – script logic is automatically parallelized

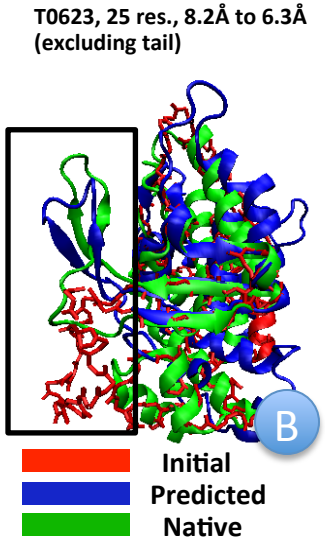


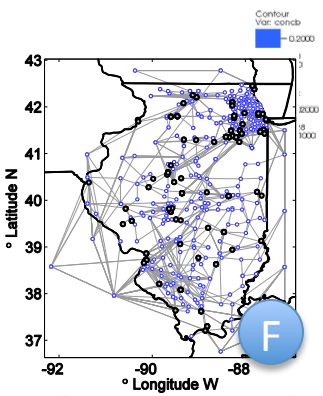
Work of: S Mickelson, R Jacob, J Dennis, M Woitasek

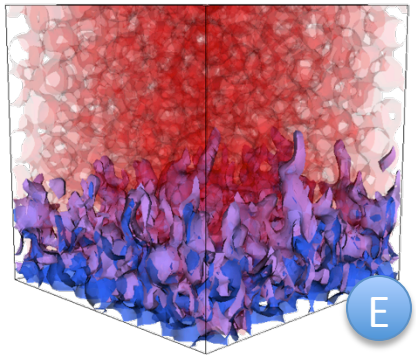
Large-scale many-task applications using Swift

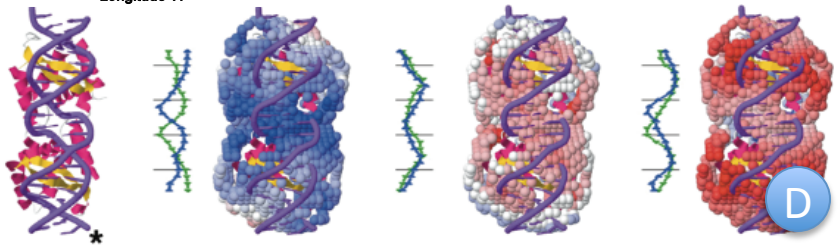
- **A** Simulation of super-cooled glass materials
- **B** Protein folding using homology-free approaches
- **C** Climate model analysis and decision making in energy policy
- **D** Simulation of RNA-protein interaction
- **E** Multiscale subsurface flow modeling
- **F** Modeling of power grid for OE applications
- **>** All have published science results obtained using Swift

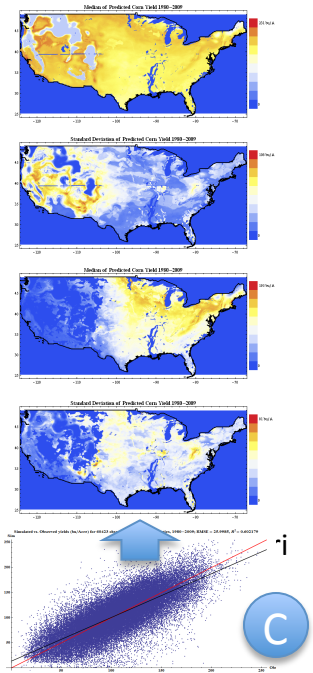




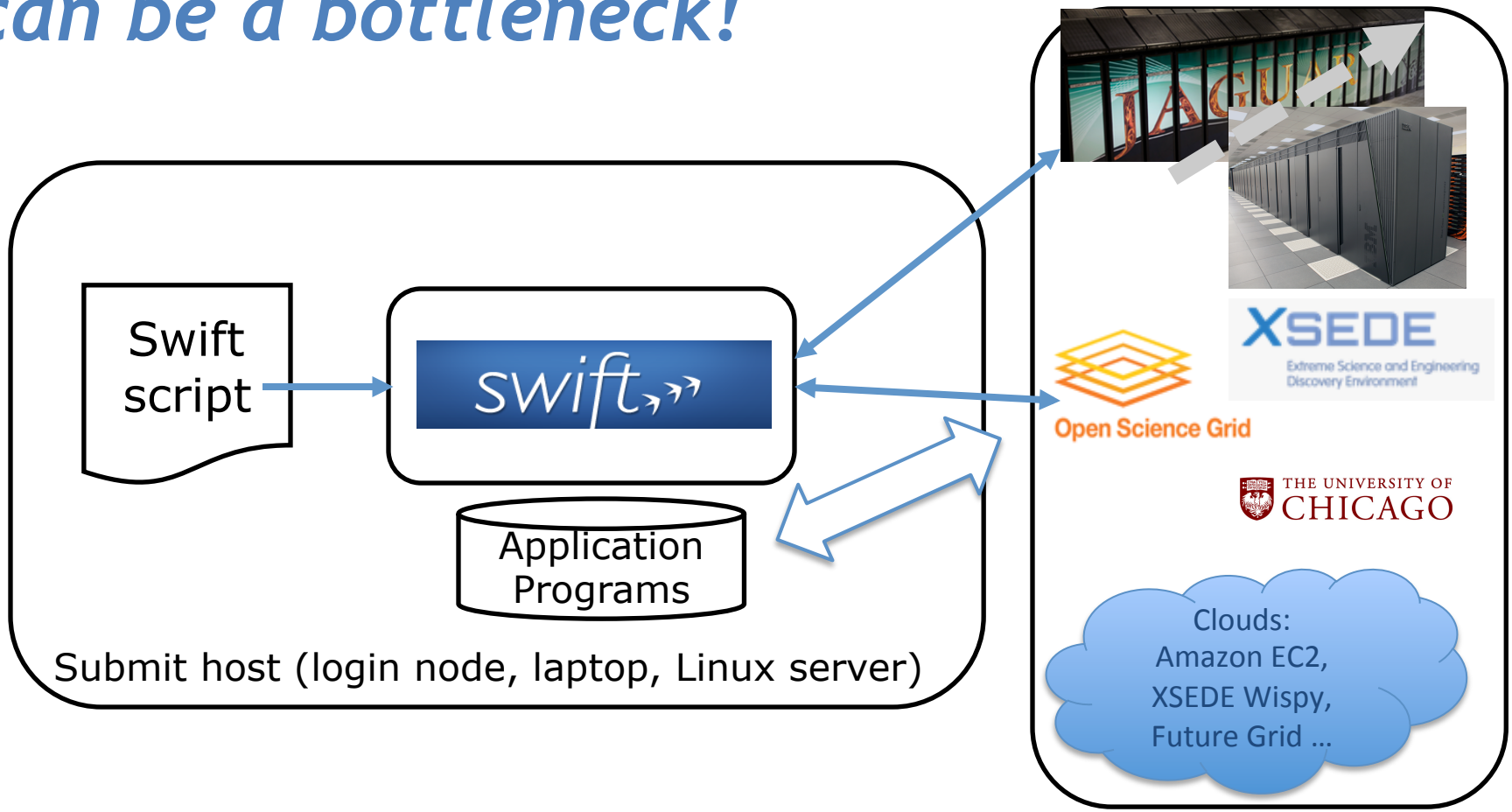








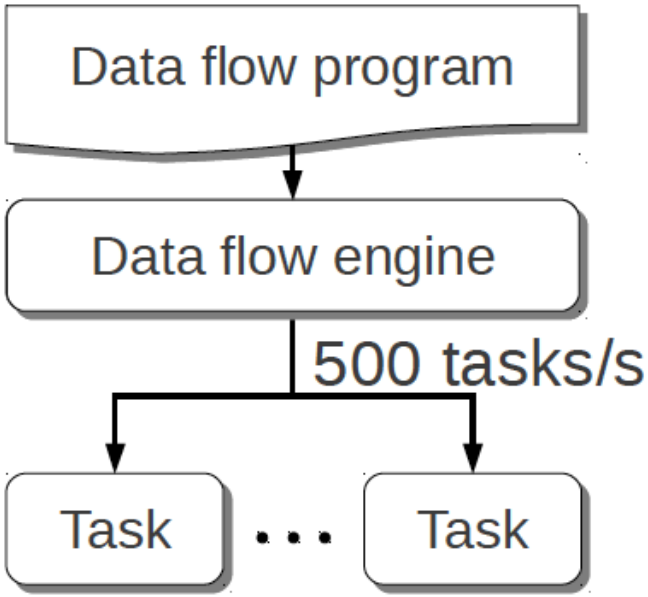
But, a problem: *Centralized evaluation can be a bottleneck!*



500 tasks/sec is good for traditional workflow systems, but can't utilize a large supercomputer if tasks are short.

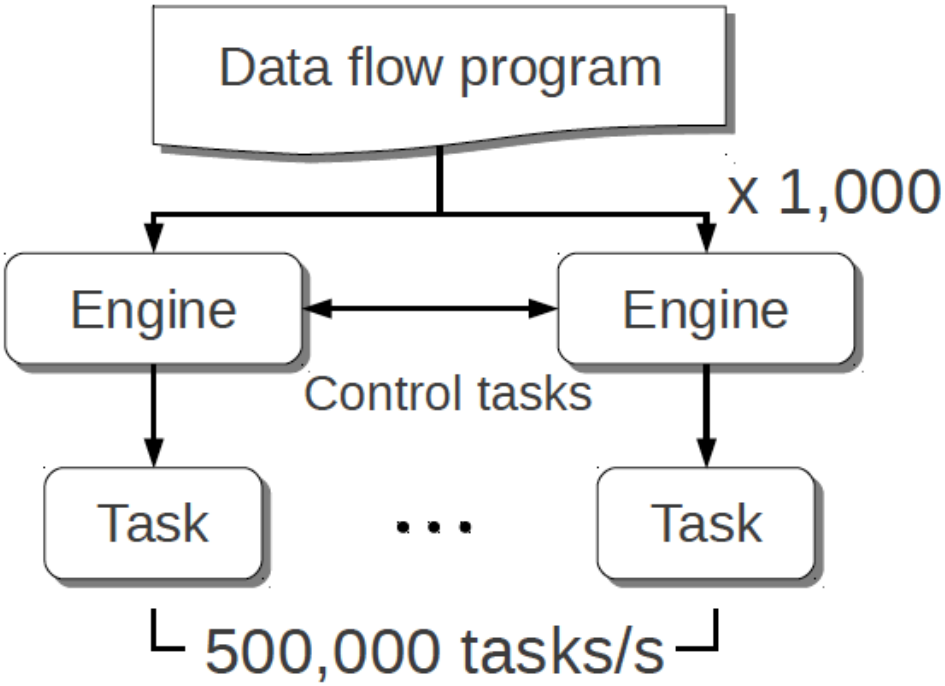
Centralized evaluation can be a bottleneck at extreme scales

Had this (Swift/K):



Centralized evaluation

For extreme scale, we need this (Swift/T):



Distributed evaluation

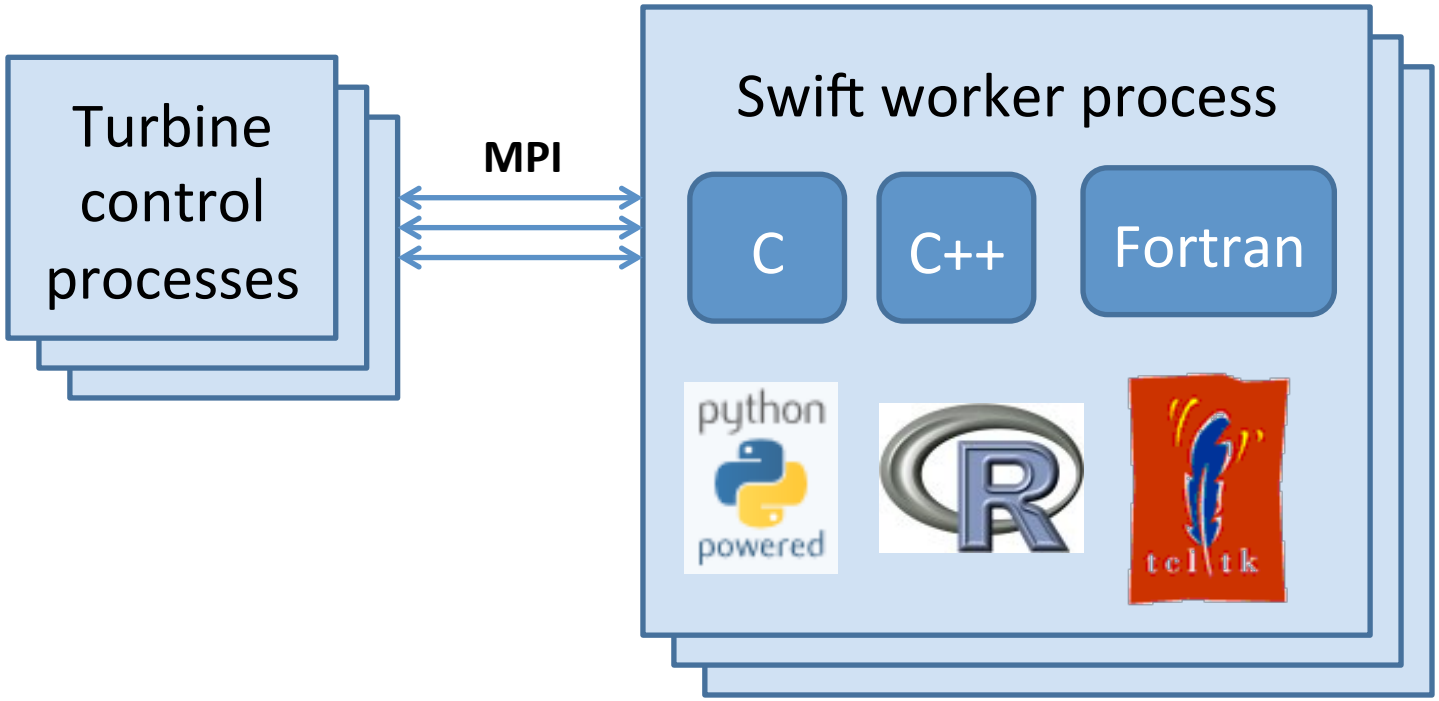
ExM - Extreme scale Many-task computing

Compiler Techniques for Massively Scalable Implicit Task Parallelism

Timothy G. Armstrong,* Justin M. Wozniak,†‡ Michael Wilde,†‡ Ian T. Foster*†‡
*Dept. of Computer Science, University of Chicago, Chicago, IL, USA
†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
‡Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

http://people.cs.uchicago.edu/~tga/pubs/stc-preprint-apr14.pdf

Swift/T: High-level model with Turbine runtime

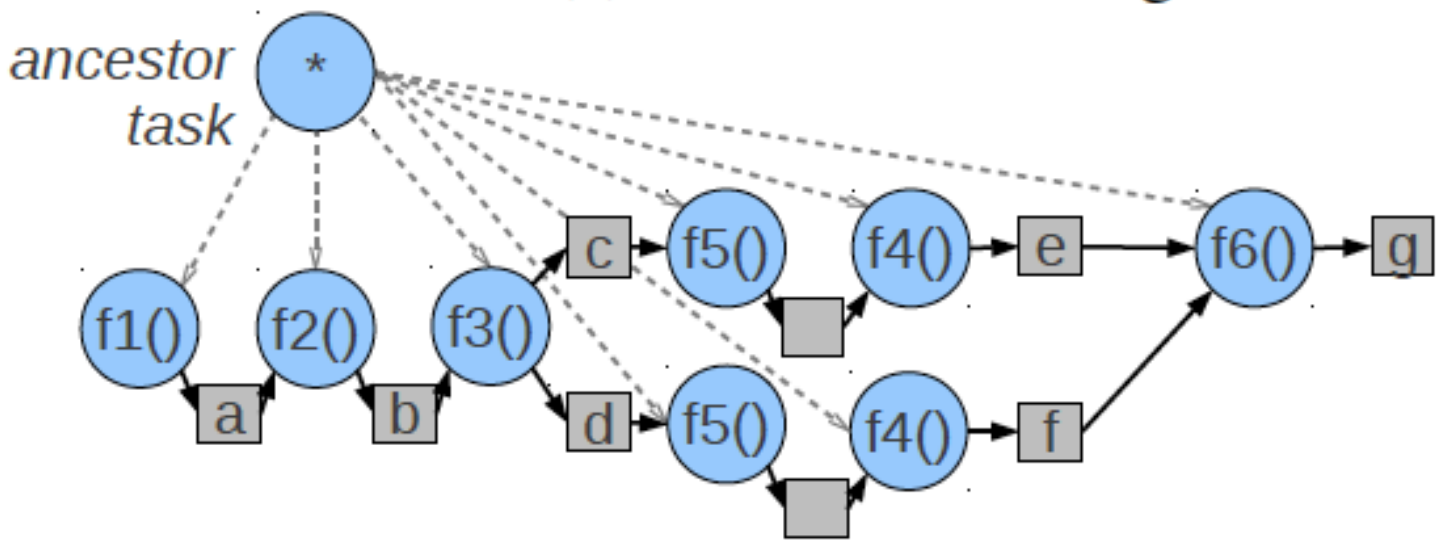


- Script-like global-view programming with "leaf" tasks
  - function calls in C, C++, Fortran, Python, R, or Tcl
- Leaf tasks can be MPI programs, etc. Can be separate processes if OS permits.
- Distributed, scalable runtime manages tasks, load balancing, data movement
- User function calls to external code run on thousands of worker nodes
- Like master-worker but with expressive Swift language to express

Optimization challenge: distributed data

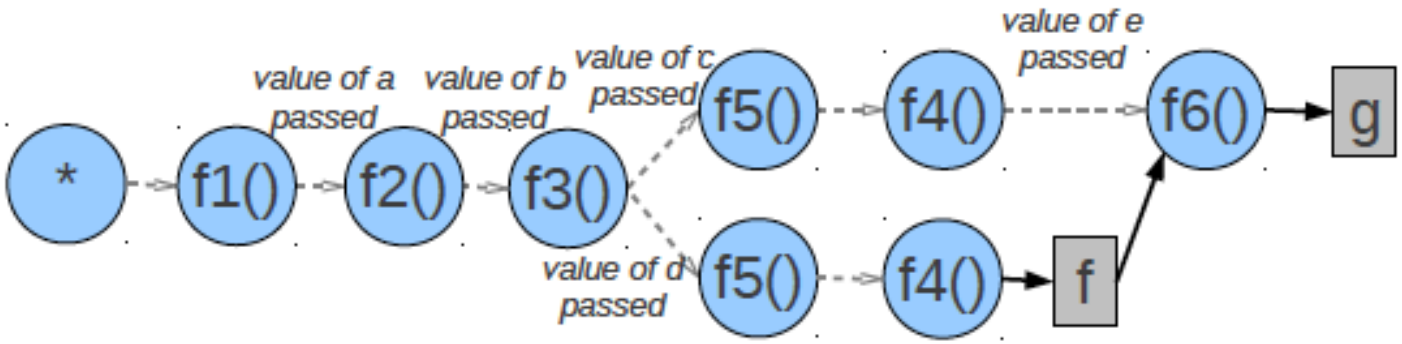
```
1 | a = f1();            b = f2(a);
2 | c, d = f3(a, b);     e = f4(f5(c);
3 | f = f4(f5(d);        g = f6(e, f);
```

(a) Swift/T code fragment

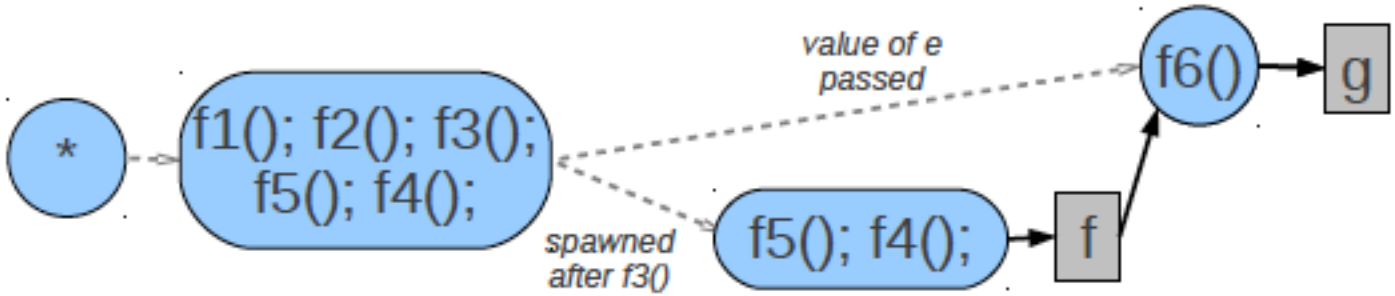


(b) Unoptimized version, passing data as shared data and perform synchronization

Swift/T optimizations improve data locality



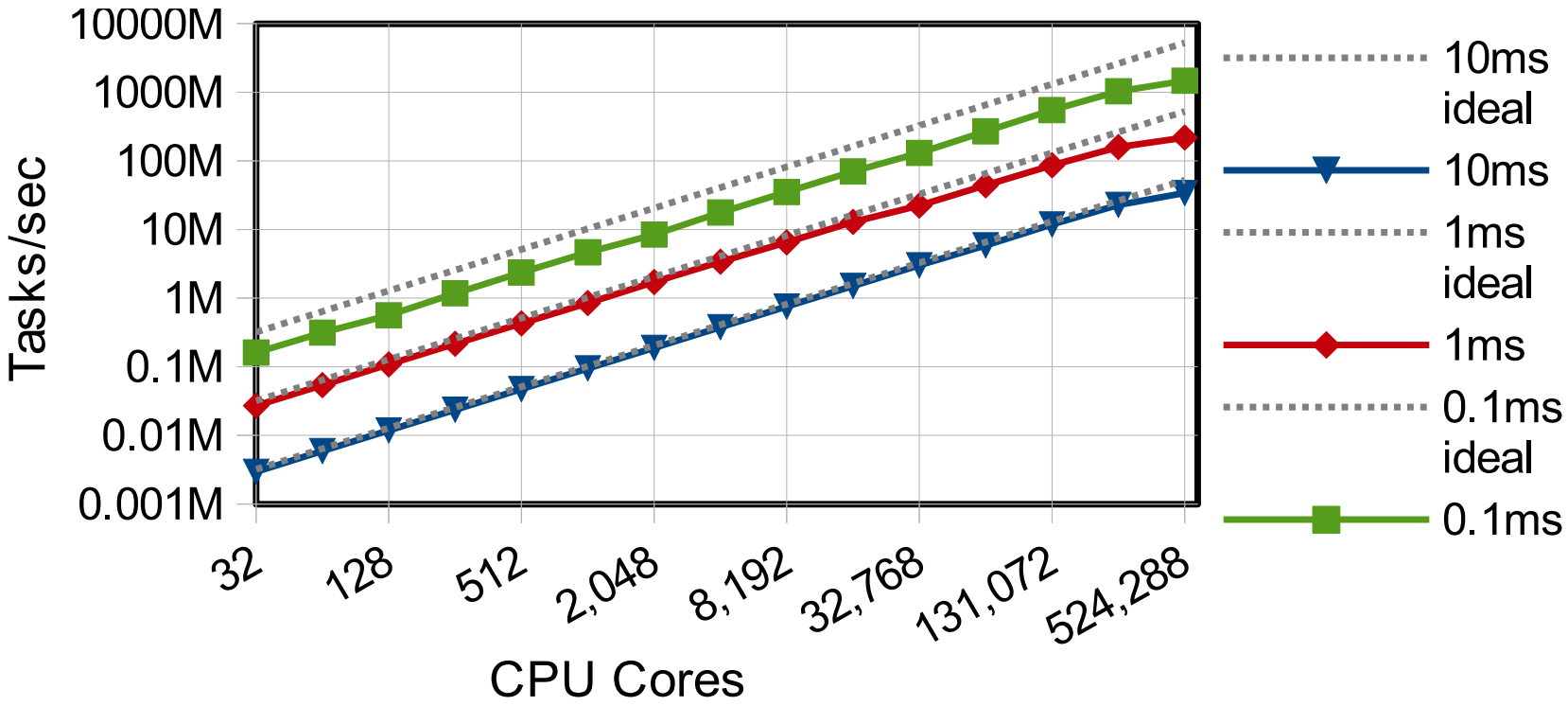
(c) After wait pushdown and elimination of shared data in favor of parent-to-child data passing



(d) After pipeline fusion merges tasks

Scaling of trivial foreach { } loop in Swift/T

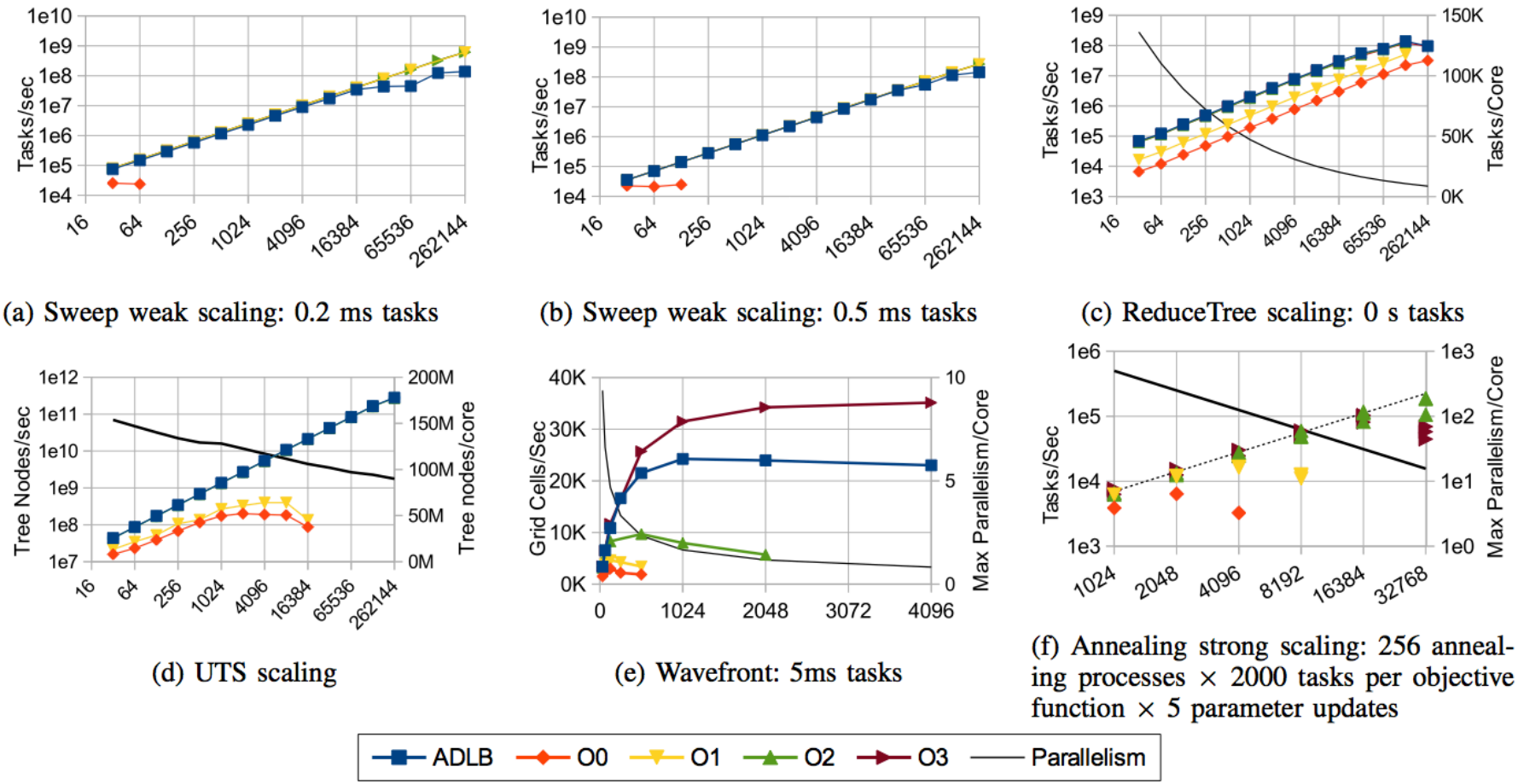
100 microsecond to 10 millisecond tasks on up to 512K integer cores of Blue Waters



Advances in compiler and runtime optimization enable Swift to be applied in new in-memory programming models.

http://people.cs.uchicago.edu/~tga/pubs/stc-preprint-apr14.pdf

Swift/T application benchmarks on Blue Waters



(a) Sweep weak scaling: 0.2 ms tasks

(b) Sweep weak scaling: 0.5 ms tasks

(c) ReduceTree scaling: 0 s tasks

(d) UTS scaling

(e) Wavefront: 5ms tasks

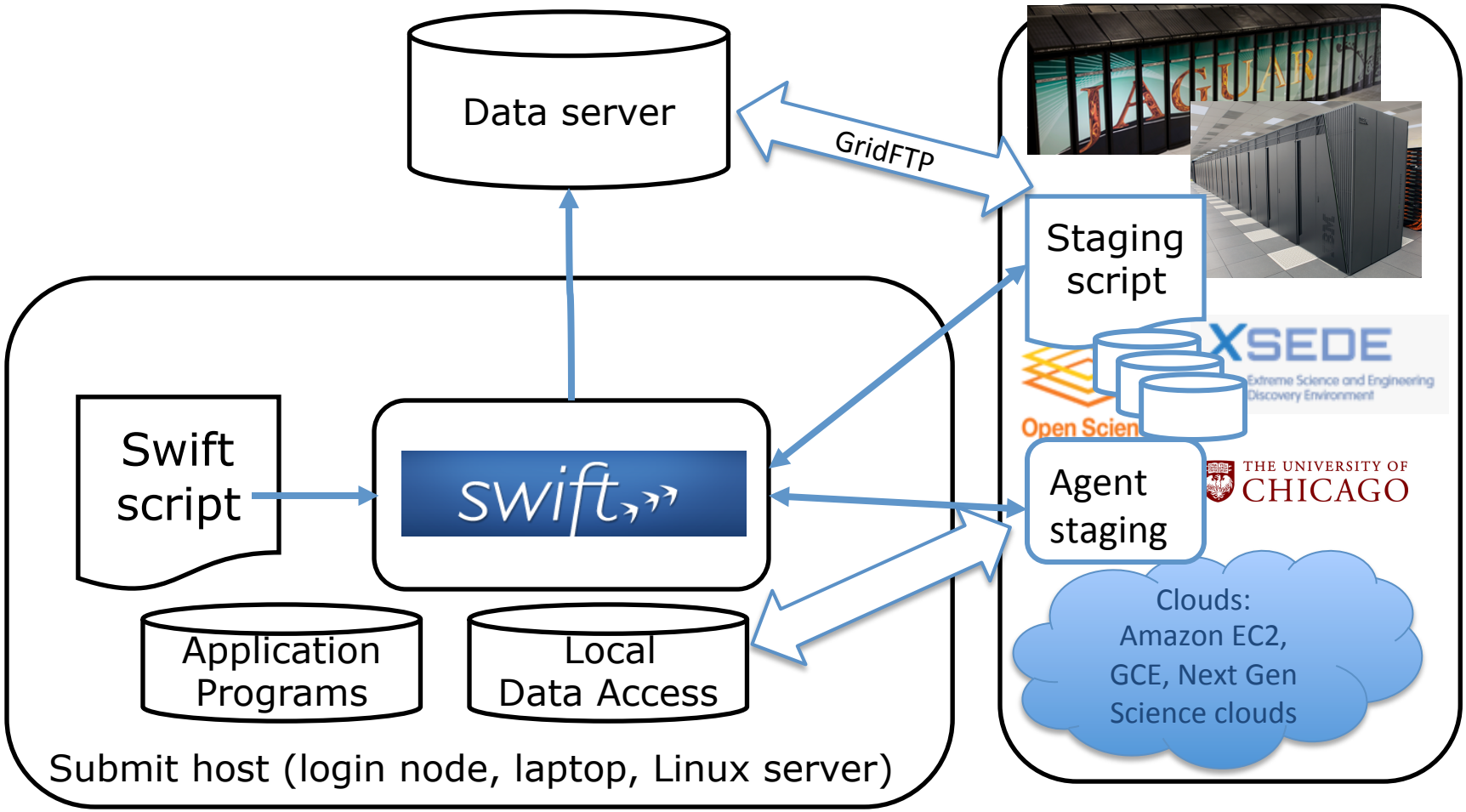
(f) Annealing strong scaling: 256 annealing processes × 2000 tasks per objective function × 5 parameter updates

Fig. 10: Application speedup and scalability at different optimization levels. X axes show scale in cores. Primary Y axes show application throughput in application-dependent terms. Secondary Y axes show problem size or degree of parallelism where applicable.

Where can this take us?

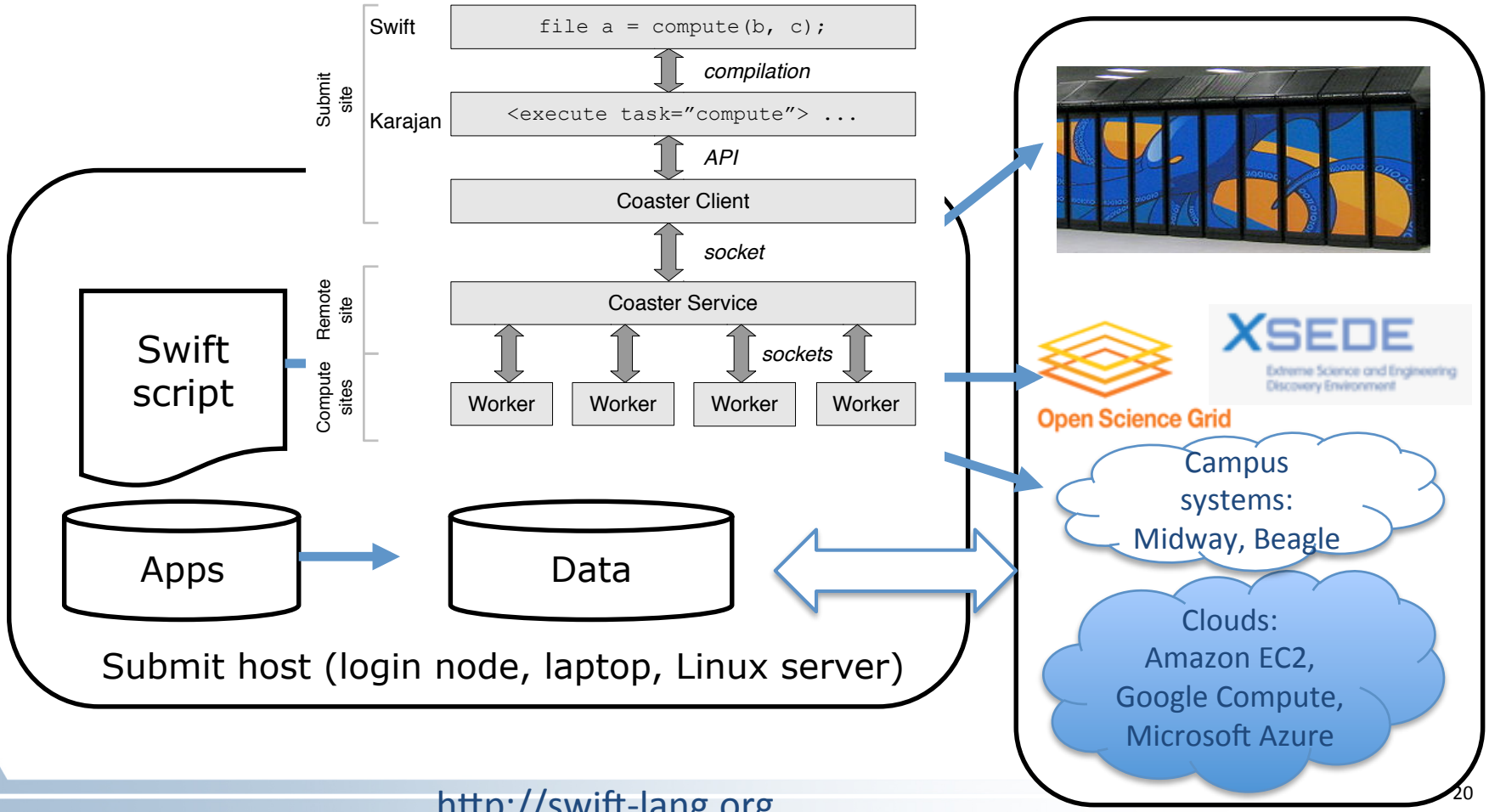
- Data management strategies are key to a workflow's utility: performance and system overhead
- Usability improvement opportunities as we scale dataflow up
- Dealing with multi-language and multi-model programming challenges
- Integration of visual and textual workflow specification

Workflows demand flexible data management models

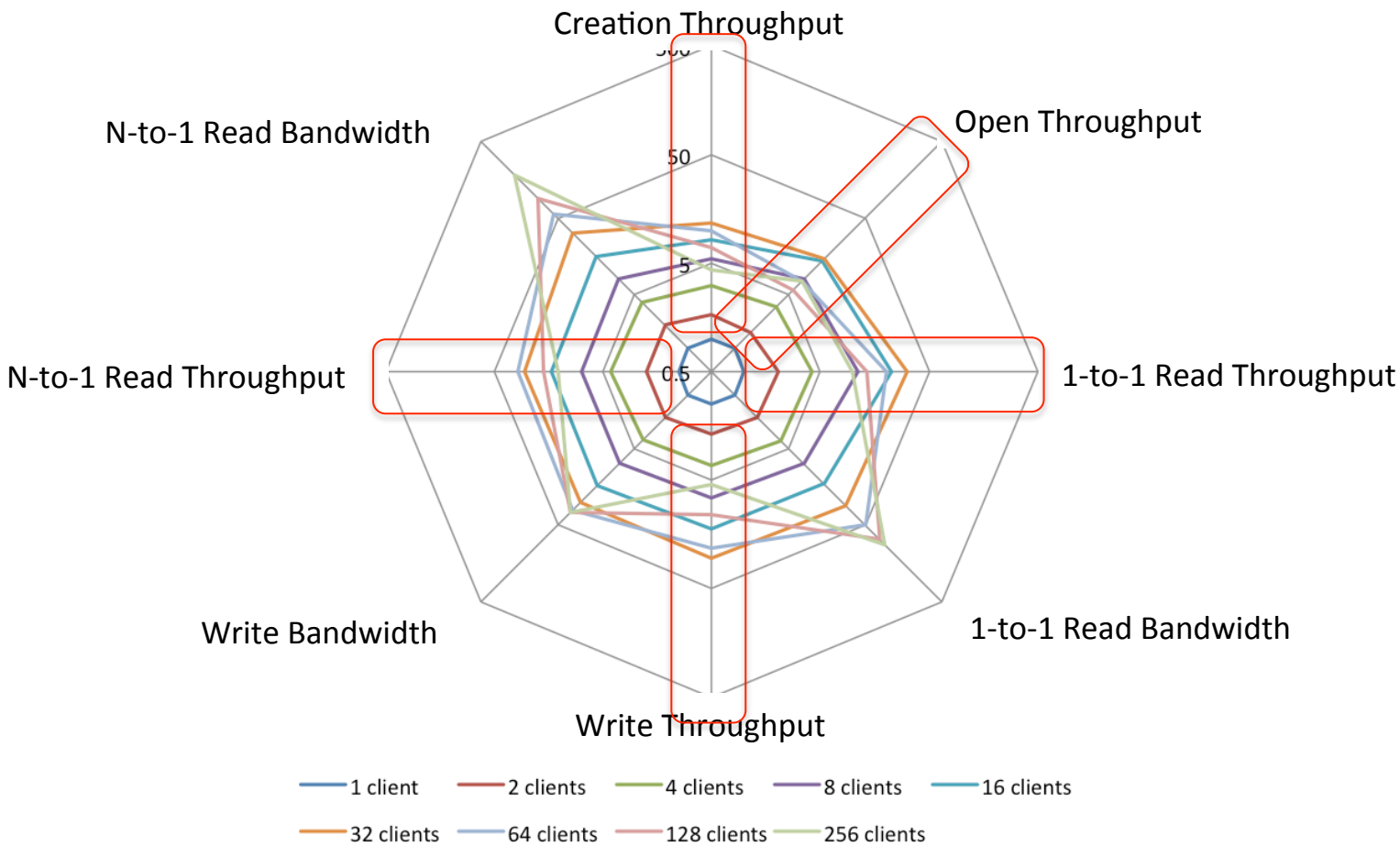


Multisystem and multijob workflow

- Dynamic provisioning with node-resident workers
  - Multi-job provisioning; Multi-system access;



MTC Envelope vs. Scale



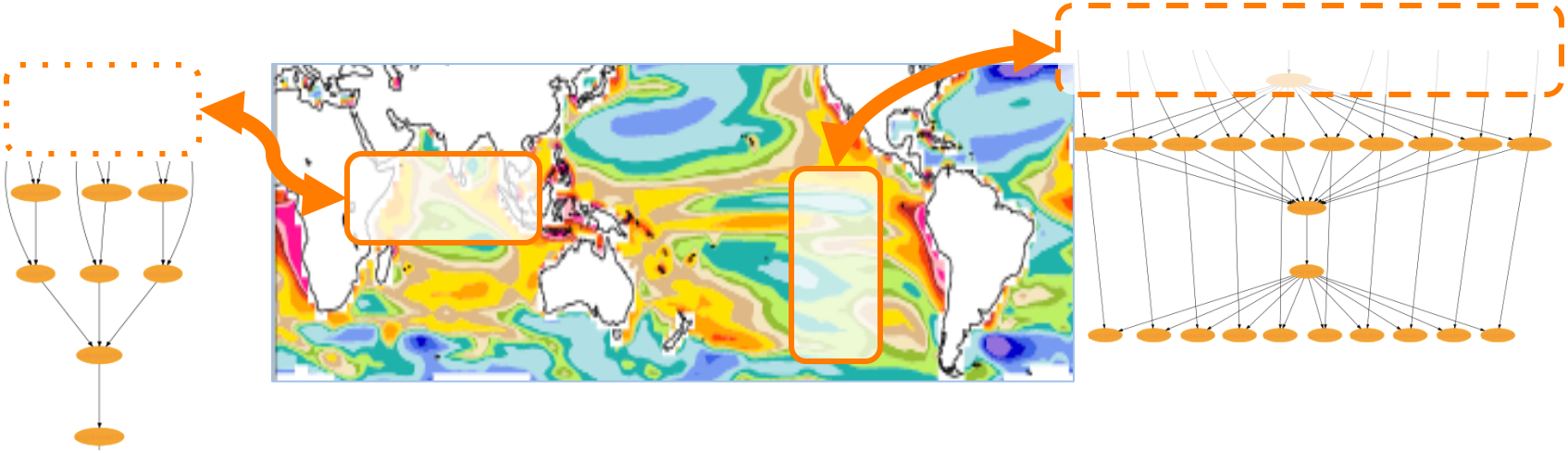
Zhao Zhang et al., MTC Envelope, HPDC 2013

Integrating the map-reduce programming model

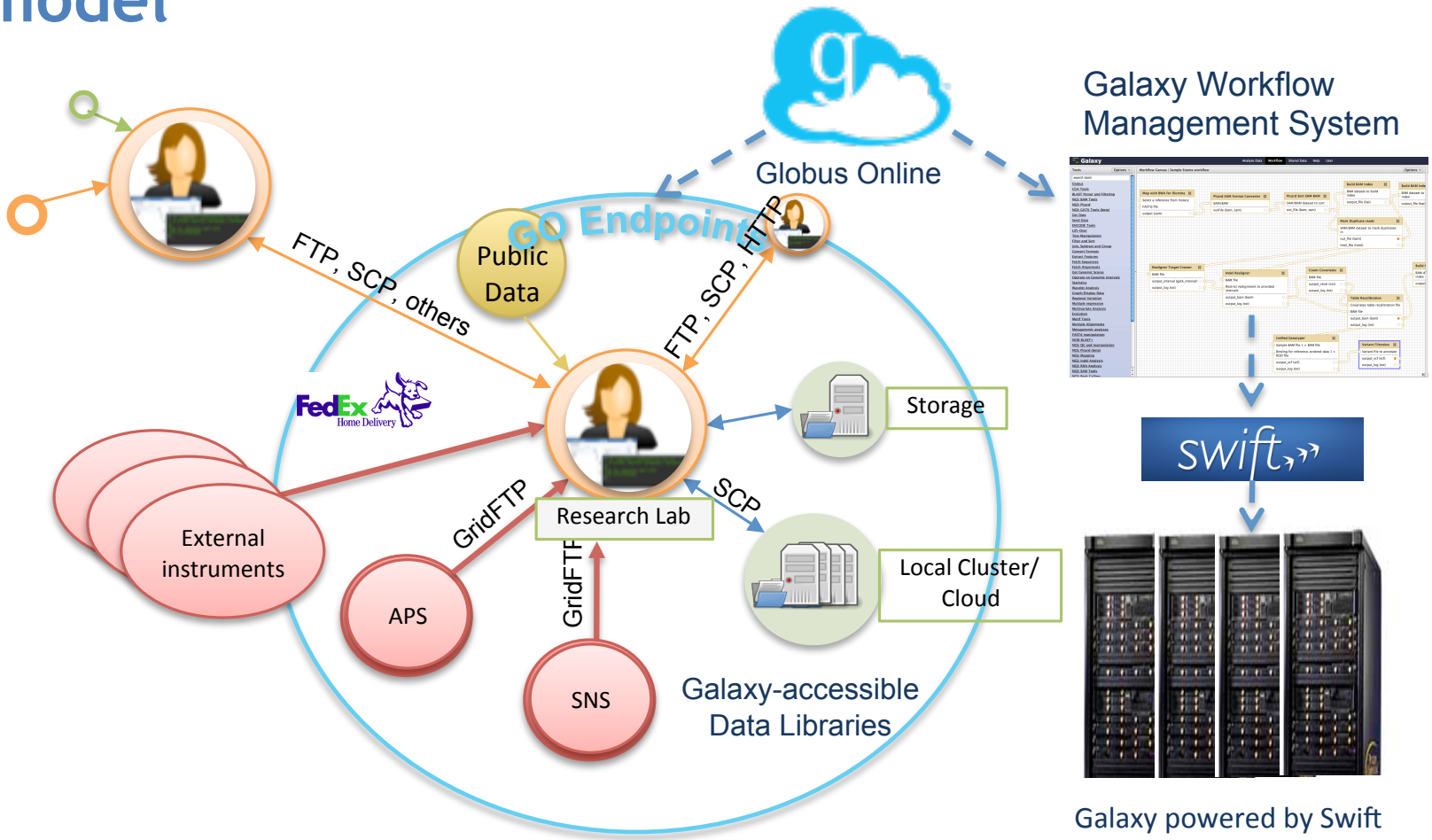
- Two main strategies
  - Operate on data in-place
  - Tree-structured data reduction
- Opportunities for pure functional dataflow
  - Locality constructs
  - Pop-up temporary filesystems and stores
  - Reduce load on shared filesystems
  - Collective data management
  - Reduction of small file usage by auto batching
  - Temp workspaces with data (semi-persistent)

Deeply in-situ processing

- Potential new model for analytics
- Multiple workflows (and even users) could share resources to analyze a running model
- Leverage lockless data structure sync
- Reduce data movement and distance to a minimum
- Work in progress by ADIOS, Dataspaces, GLEAN, DIY



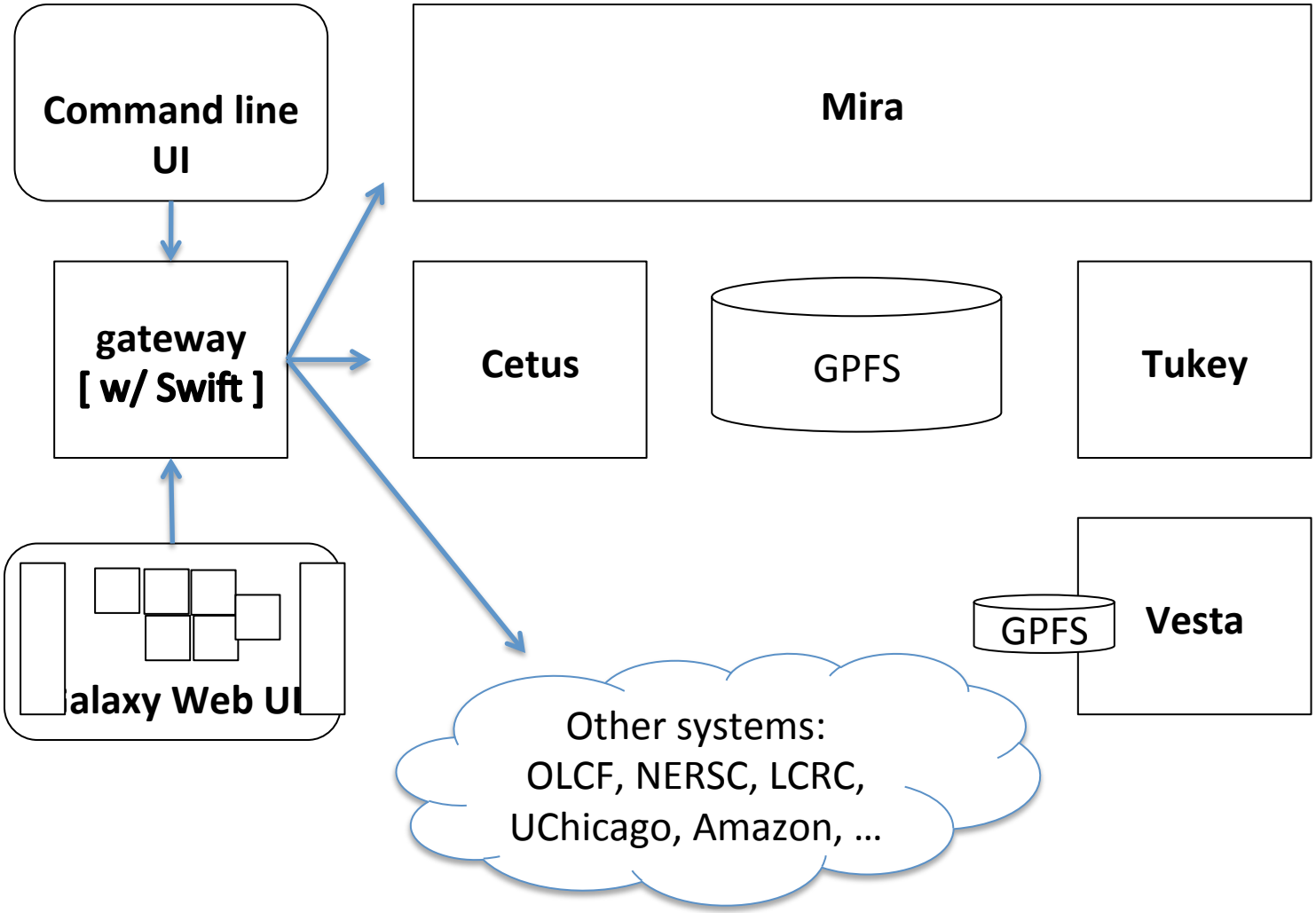
Galaxy workflow portal and the Swift execution model



Data Management

Data Analysis

Proposed architecture for ALCF Workflow Gateway prototype



Additional research directions for dataflow-based workflow model

- Exploit dataflow for energy management
- Exploit dataflow for fault recovery
- Programmability and productivity
  - More flexible dataflow constructs - foreach => formost
  - Shells Read-evaluate-print loops
  - Debugging on the compute nodes
  - Languages in dataflow => dataflow in languages

Conclusion: Implicitly parallel functional dataflow has a valuable role

- Expressive
- Portable
- Usable
- Fast
- Applicable over a broad application space
- Greatest value is for "programming in the large"
  - High productivity, lower bar to scaling
  - Works well with MPI, OpenMP, …
- Many research directions identified to extend it

Parallel Computing 37 (2011) 633–652

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Swift: A language for distributed parallel scripting

Michael Wilde [a,b,*], Mihael Hategan [a], Justin M. Wozniak [b], Ben Clifford [d], Daniel S. Katz [a], Ian Foster [a,b,c]

[a] Computation Institute, University of Chicago and Argonne National Laboratory, United States
[b] Mathematics and Computer Science Division, Argonne National Laboratory, United States
[c] Department of Computer Science, University of Chicago, United States
[d] Department of Astronomy and Astrophysics, University of Chicago, United States

ARTICLE INFO

Article history:
Available online 12 July 2011

Keywords:
Swift
Parallel programming
Scripting
Dataflow

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.

Acknowledgments

- Swift is supported in part by NSF grants OCI-1148443 and PHY-636265, and the UChicago SCI Program
- Extreme scaling research on Swift (ExM project) is supported by the DOE Office of Science, ASCR Division, X-Stack program.
- Application support from NIH (fMRI, OpenMX)

- The Swift team:
  - Tim Armstrong, Ian Foster, Mihael Hategan, Dan Katz, David Kelly, Ketan Maheshwari, Yadu Nand, Mike Wilde, Justin Wozniak, Zhao Zhang
- Special thanks to our collaborators and the Swift user community