

# Swift Workflows for Simulations and Data Analytics

August 2015

Presenter and contact:

Michael Wilde    [wilde@anl.gov](mailto:wilde@anl.gov)

<http://swift-lang.org>



*Swift* gratefully acknowledges support from:



U.S. DEPARTMENT OF  
**ENERGY**



THE UNIVERSITY OF  
**CHICAGO**

**Argonne**

NATIONAL LABORATORY



<http://swift-lang.org>

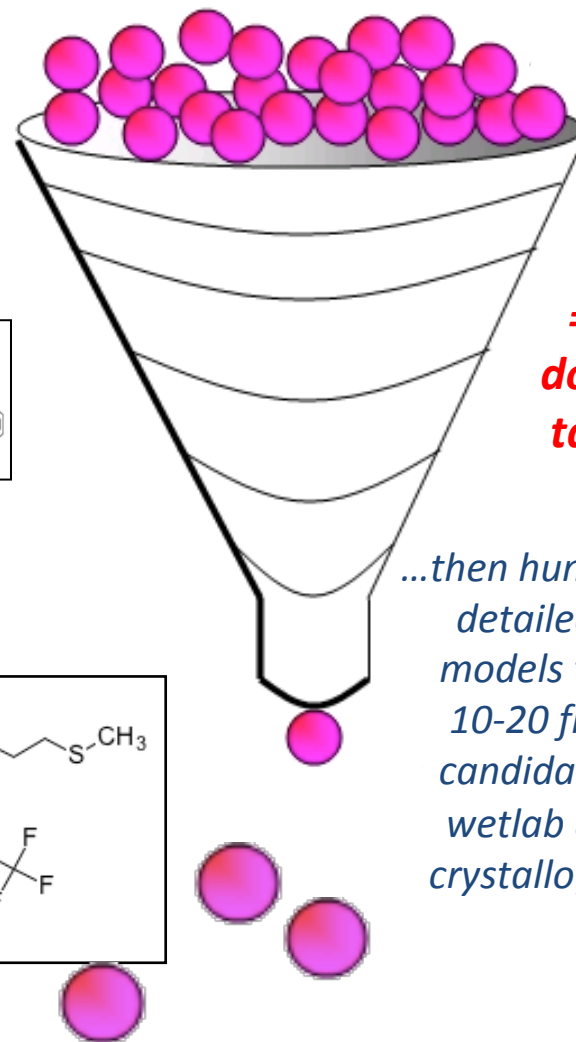
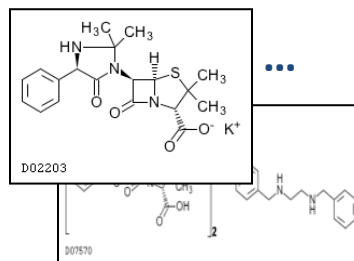
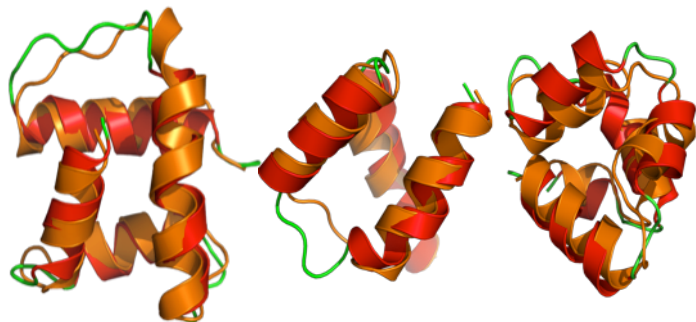
# When do you need HPC workflow?

Example application: protein-ligand docking for drug screening

$O(10)$  proteins  
implicated in a disease

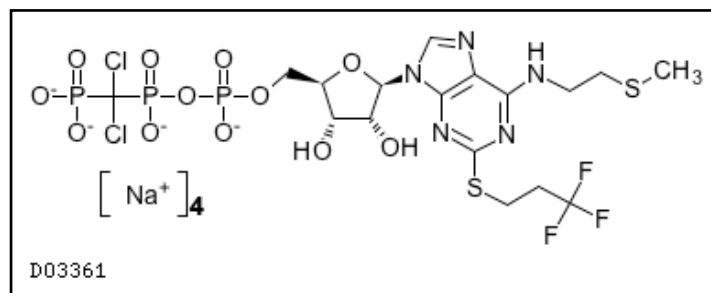
X

$O(100K)$   
drug  
candidates



= 1M  
docking  
tasks...

...then hundreds of  
detailed MD  
models to find  
10-20 fruitful  
candidates for  
wetlab & APS  
crystallography



# Expressing this many task workflow in Swift

*For protein docking workflow:*

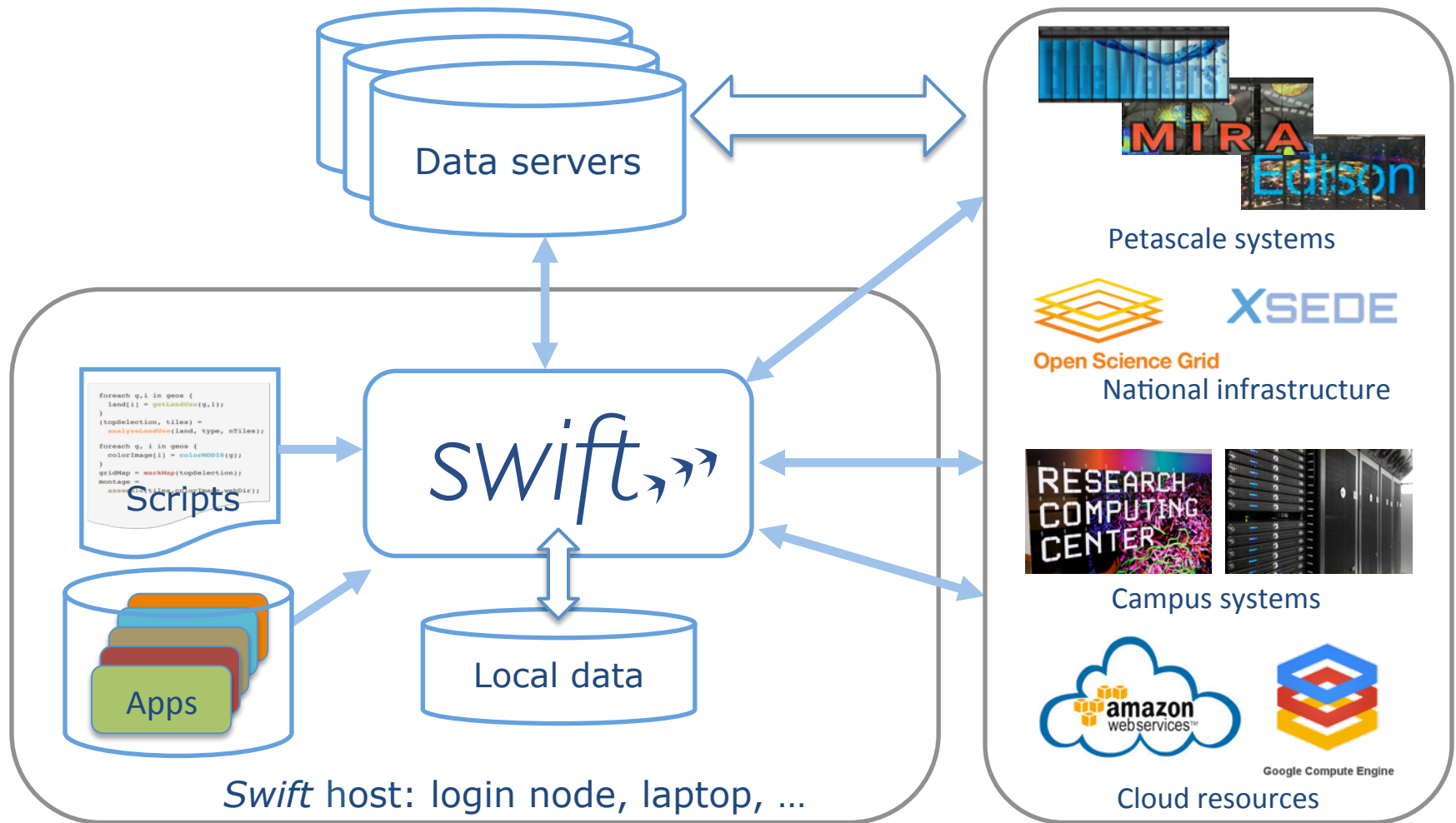
```
foreach p, i in proteins {  
    foreach c, j in ligands {  
        (structure[i,j], log[i,j]) =  
            dock(p, c, minRad, maxRad);  
    }  
scatter_plot = analyze(structure)
```

*To run:*

```
swift -site blues dock.swift
```



# Swift enables execution of simulation campaigns across multiple HPC and cloud resources



The Swift runtime system has drivers and algorithms to efficiently support and aggregate diverse runtime environments

# What Swift does for you

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*



# Swift in a nutshell

- Data types

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

- Mapped data types

```
type image;  
image file1<"snapshot.jpg">;
```

- Mapped functions

```
app (file o) myapp(file f, int i)  
{ mysim "-s" i @f @o; }
```

- Conventional expressions

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

- Structured data

```
image A[]<array_mapper...>;
```

- Loops

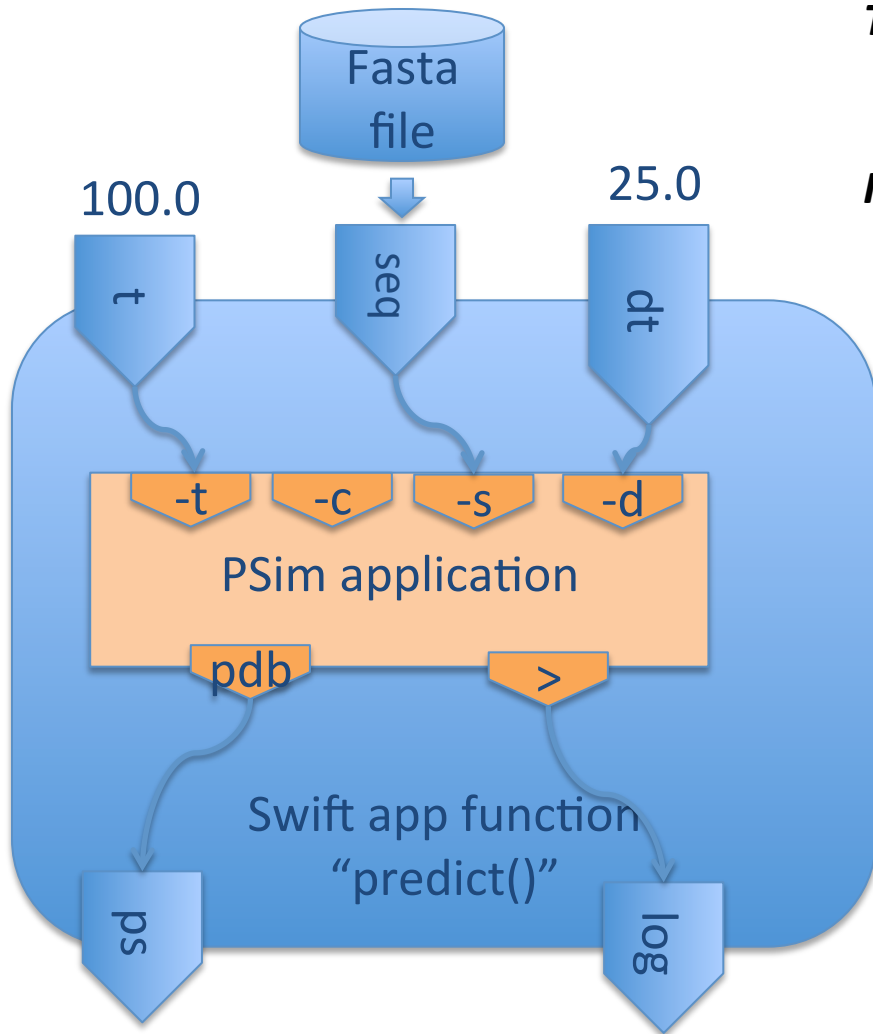
```
foreach f,i in A {  
    B[i] = convert(A[i]);  
}
```

- Data flow

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



# app( ) functions encapsulate scientific application tools



## To run:

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

## In Swift code:

```
app (PDB ps, Text log) predict (Protein seq,  
                                Float t, Float dt)  
{  
  psim "-s" @pseq.fasta "-pdb" @ps  
        "-t" temp      "-d" dt  
        stdout = log;  
}
```

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

```
(structure, log) = predict(p, 100., 25.);
```





# Pervasively parallel

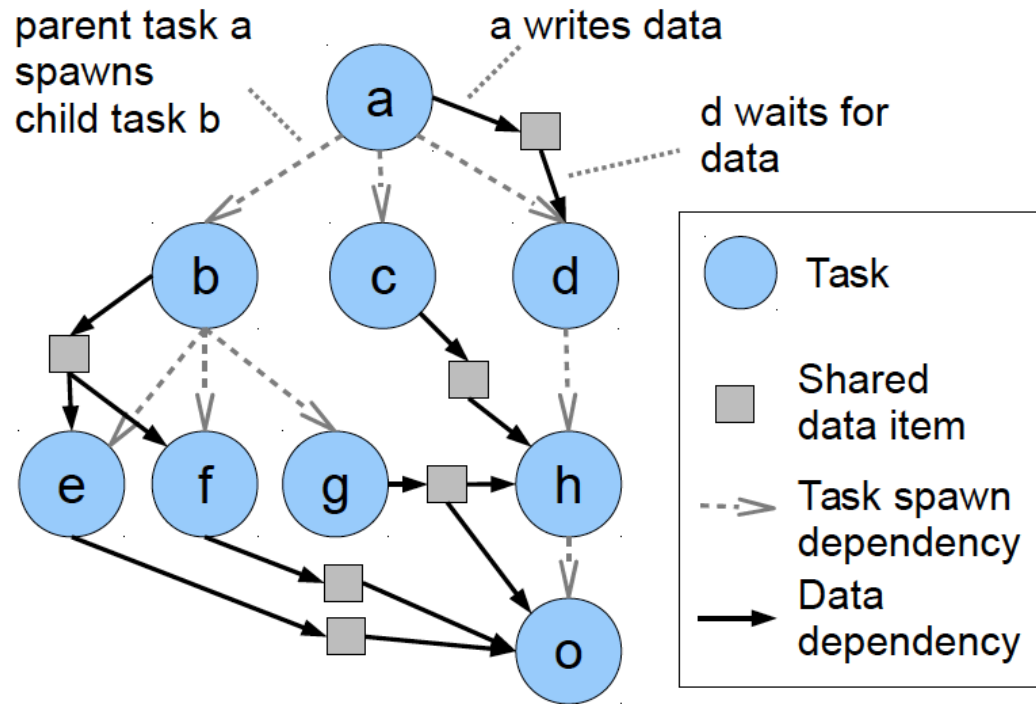
- Swift is a parallel scripting system for grids, clouds and clusters

```
(int r) myproc (int i)
{
    int f = F(i);
    int g = G(i);
    r = f + g;
}
```

- F() and G() are computed in parallel
  - Can be Swift functions, or leaf tasks (executables or scripts in shell, python, R, Octave, MATLAB, ...)
- r computed when they are done
- This parallelism is *automatic*
- Works recursively throughout the program's call graph



# Pervasive parallel data flow



# All data atoms in Swift are “futures”

$a = f(b)$

Name: a	Type: float	Value: unset	Waiting evals
---------	-------------	--------------	---------------

$x = \underline{a} + f(v)$

$y = f(\underline{a})$

$z = \underline{a} + b$



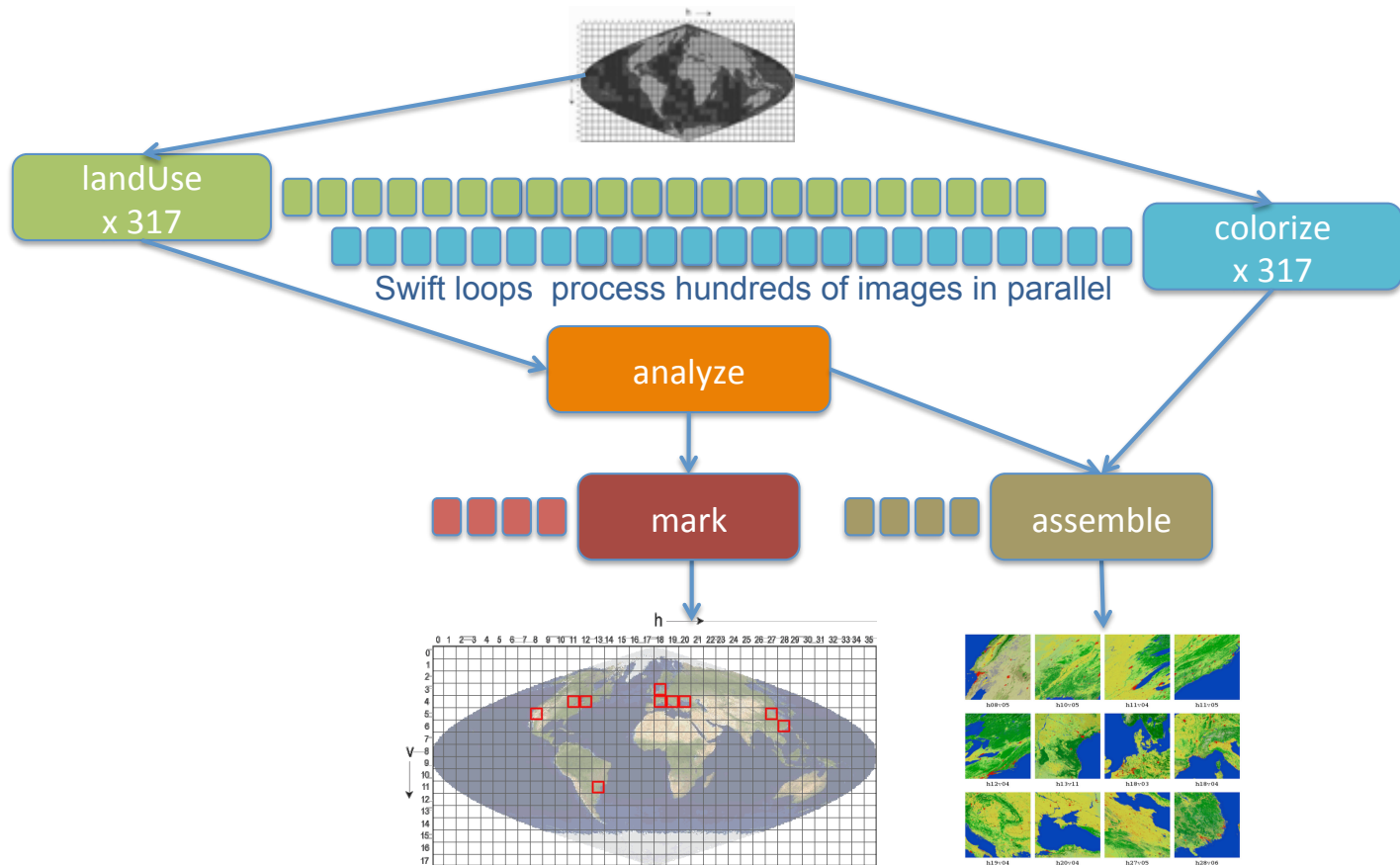
# Functional composition of a parameter sweep in *Swift*

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

10 proteins x 1000 simulations x  
3 rounds x 2 temps x 5 deltas  
= 300K tasks



# Data-intensive example: Processing MODIS land-use data



*Image processing pipeline for land-use data from the MODIS satellite instrument...*

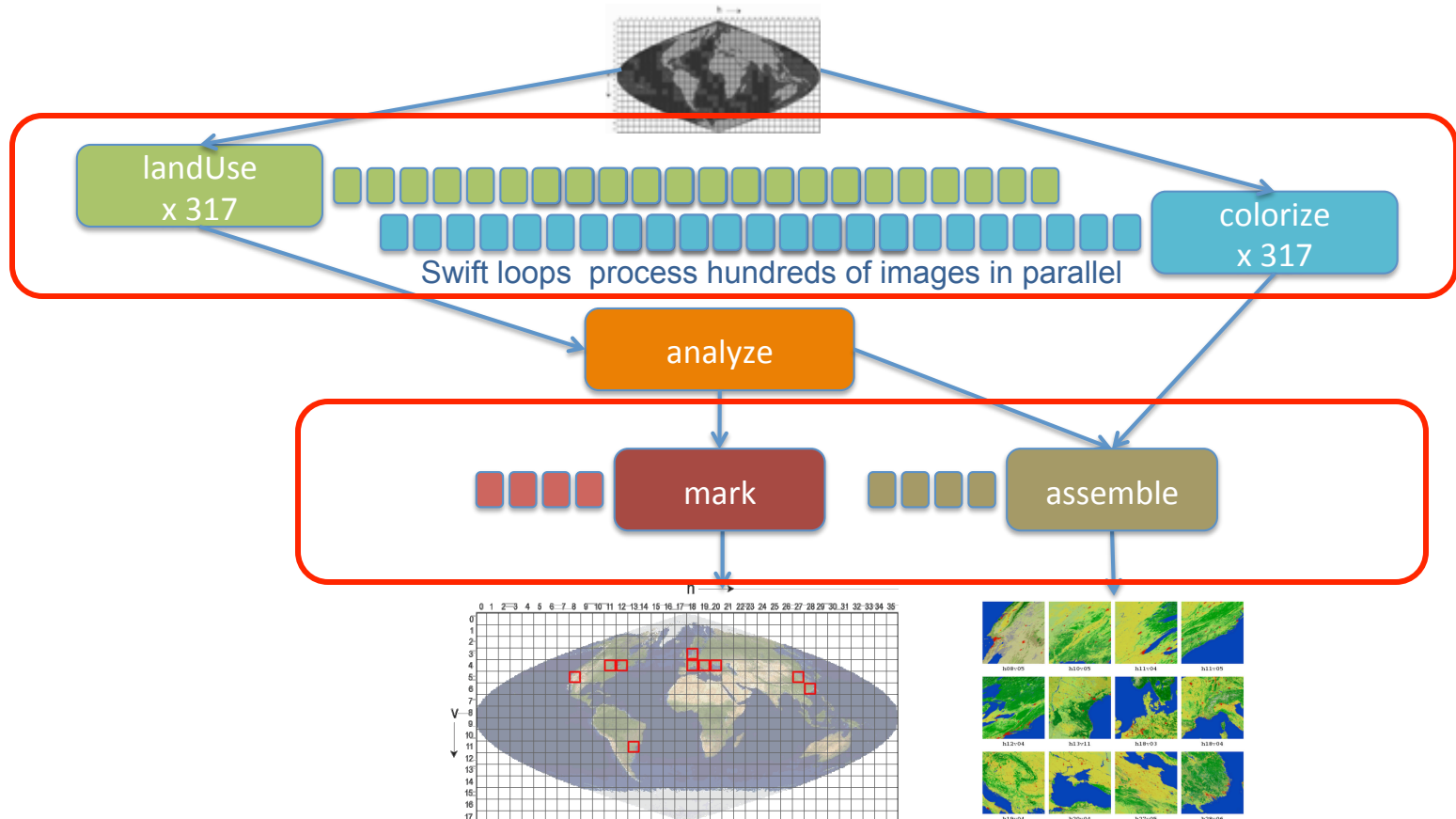


## Example of Swift's implicit parallelism: Processing MODIS land-use data

```
foreach raw,i in rawFiles {  
    land[i] = landUse(raw,1);  
    colorFiles[i] = colorize(raw);  
}  
  
(topTiles, topFiles, topColors) =  
    analyze(land, landType, nSelect);  
  
gridMap = mark(topTiles);  
montage =  
    assemble(topFiles,colorFiles,webDir);
```



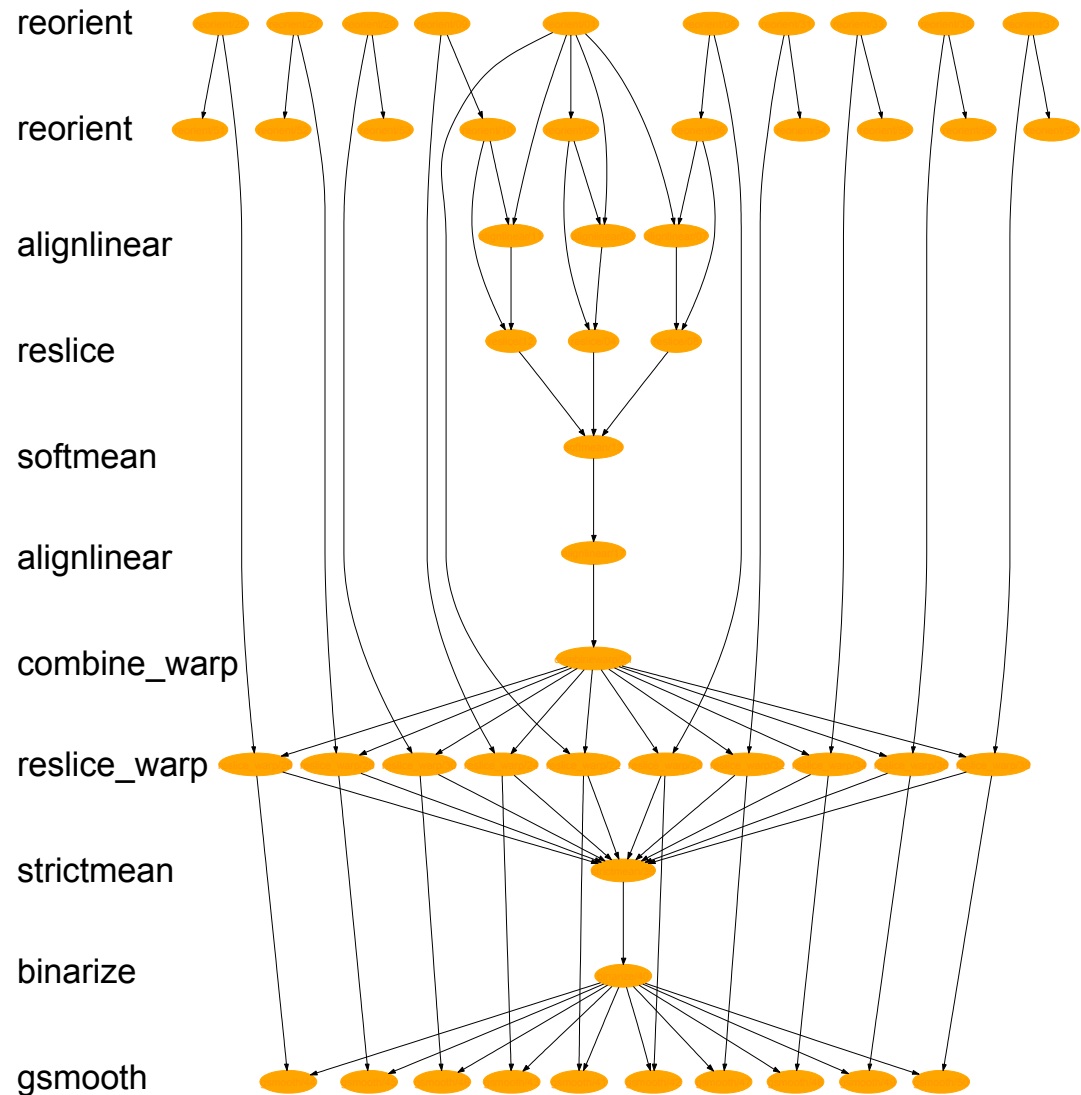
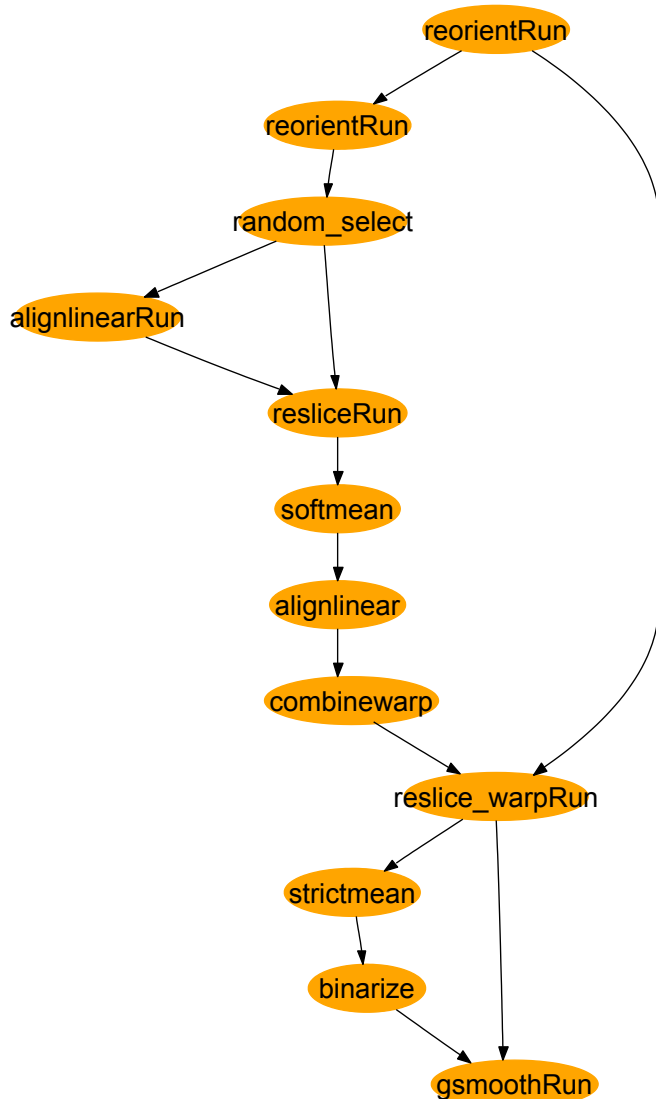
# Example of Swift's implicit parallelism: Processing MODIS land-use data



*Image processing pipeline for land-use data from the MODIS satellite instrument...*



# Spatial normalization of functional MRI runs



Dataset-level workflow

Expanded (10 volume) workflow





# Complex scripts can be well-structured

*programming in the large: fMRI spatial normalization script example*

```
(Run snr) functional ( Run r, NormAnat a,  
                      Air shrink )
```

```
{  Run yroRun = reorientRun( r , "y" );  
   Run roRun = reorientRun( yroRun , "x" );
```

```
   Volume std = roRun[0];
```

```
   Run rndr = random_select( roRun, 0.1 );
```

```
   AirVector rndAirVec = align_linearRun( rndr, std, 12, 1000, 1000, "81 3 3" );
```

```
   Run reslicedRndr = resliceRun( rndr, rndAirVec, "o", "k" );
```

```
   Volume meanRand = softmean( reslicedRndr, "y", "null" );
```

```
   Air mnQAAir = alignlinear( a.nHires, meanRand, 6, 1000, 4, "81 3 3" );
```

```
   Warp boldNormWarp = combinewarp( shrink, a.aWarp, mnQAAir );
```

```
   Run nr = reslice_warp_run( boldNormWarp, roRun );
```

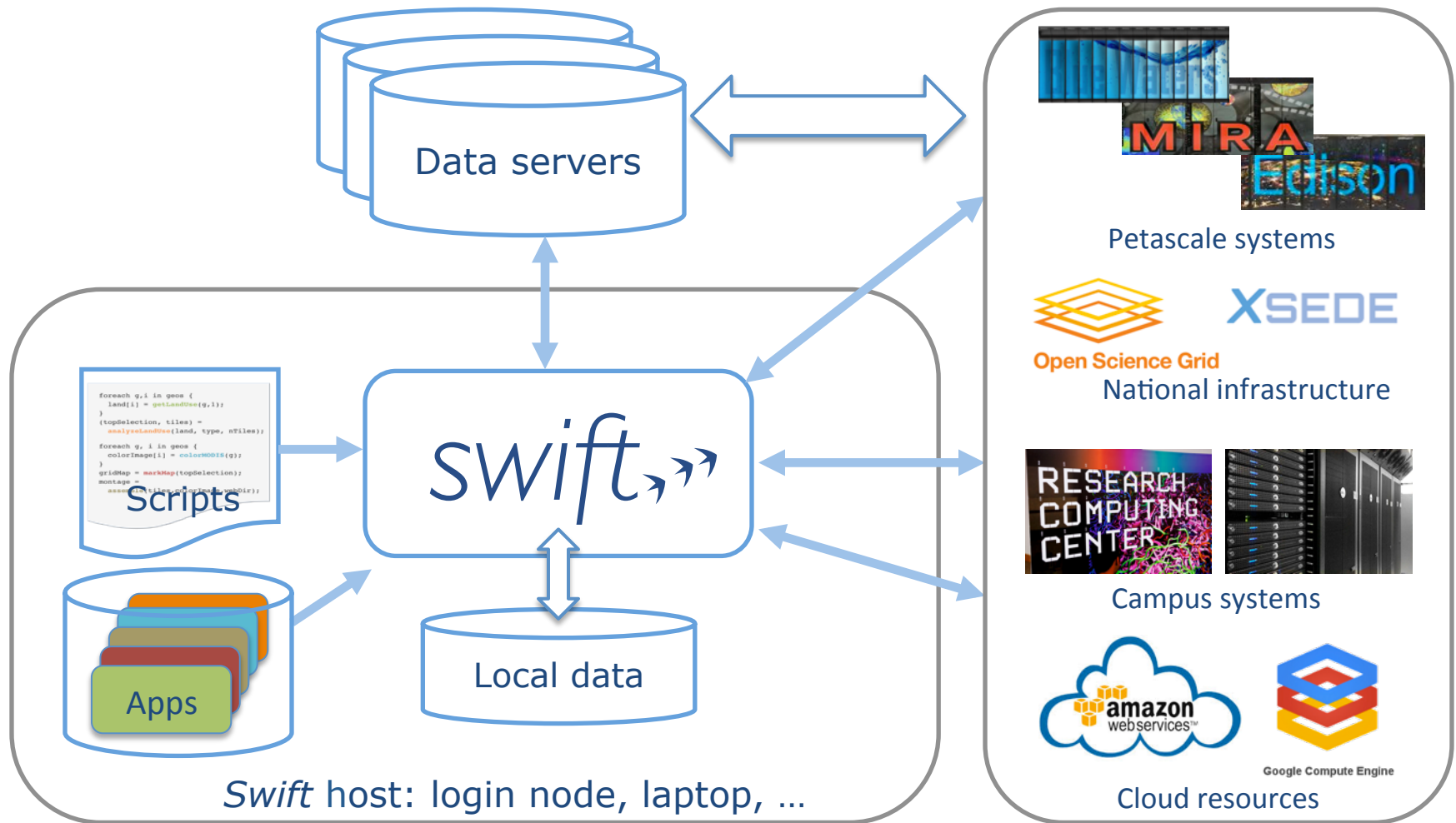
```
   Volume meanAll = strictmean( nr, "y", "null" )
```

```
   Volume boldMask = binarize( meanAll, "y" );
```

```
   snr = gsmoothRun( nr, boldMask, "6 6 6" );
```

```
(Run or) reorientRun ( Run ir, string direction )  
{  
    foreach Volume iv, i in ir.v {  
        or.v[i] = reorient(iv, direction);  
    }  
}
```

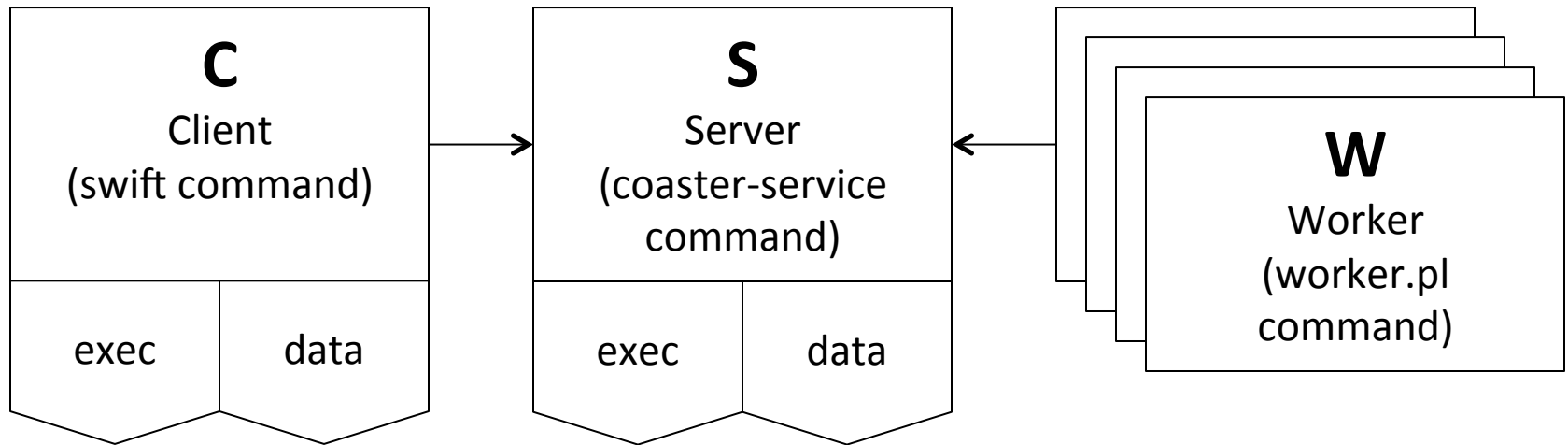
# How Swift enables workflow execution across multiple HPC and cloud resources



The Swift runtime system has drivers and algorithms to efficiently support and aggregate diverse runtime environments

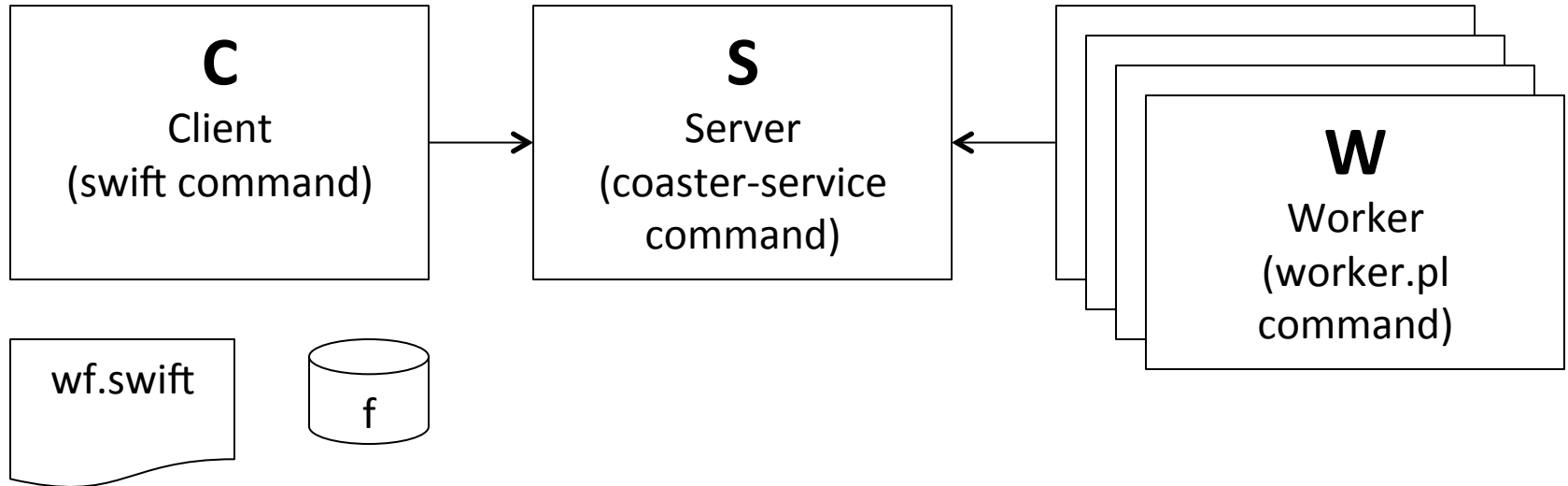
# Provider Architecture

Runs app tasks and moves files



# Service Architecture

## Building blocks of the Swift distributed workflow service



swift command runs one swift script

Lives on login runs, workflow service host, laptop, etc

Server directs tasks to workers

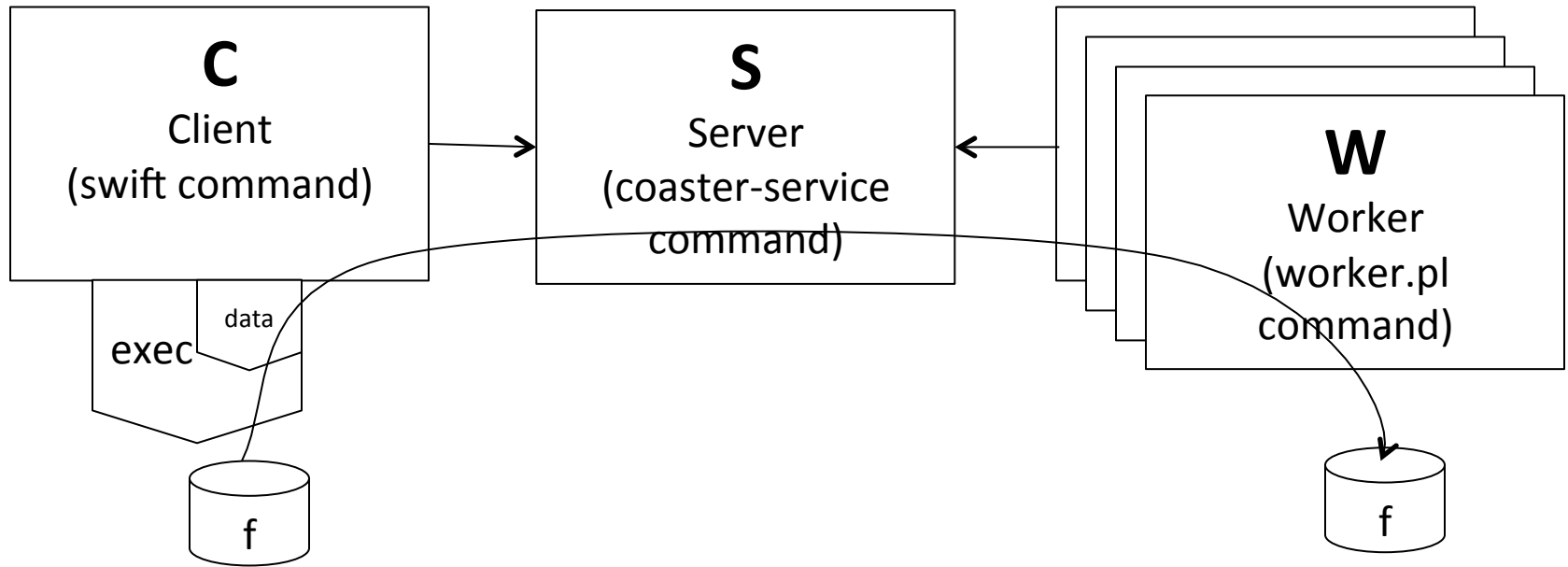
Lives inside client JVM, or standalone service on client host, or at compute site service node,

Workers run tasks on compute nodes

Runs on compute node, 1 per node or one per CPU or CPU group

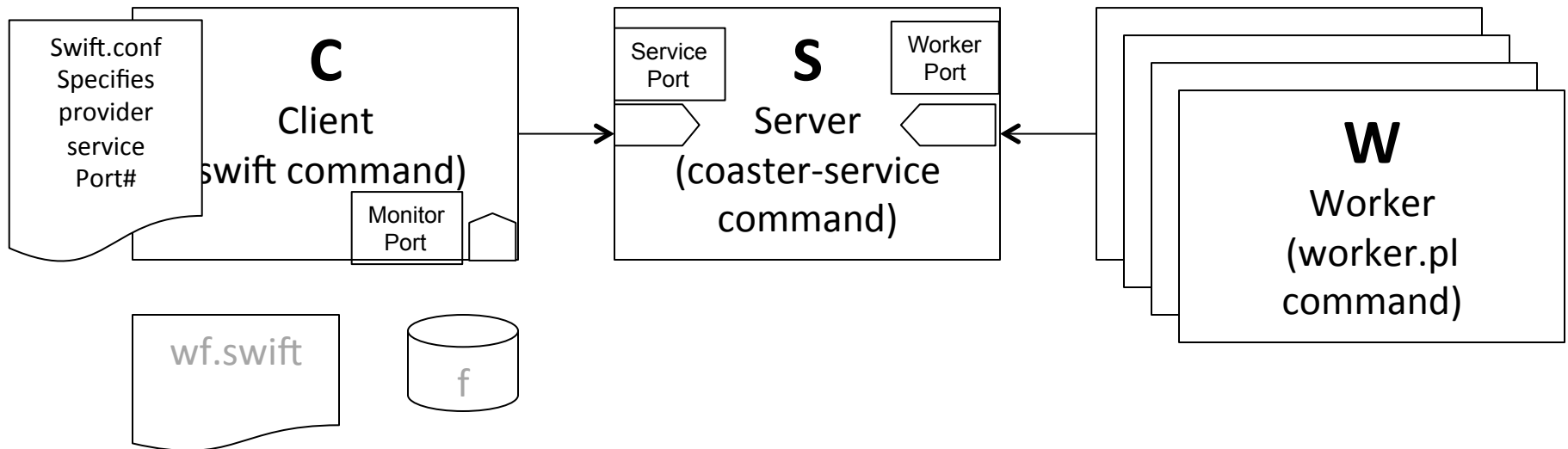
# Execution Provider Data Staging

Some execution providers can also move data (Globus, Condor and Swift's "coaster")



# Service Architecture

## Ports used to connect Swift services



swift command runs one swift script

Lives on login runs, workflow service host, laptop, etc

Server directs tasks to workers

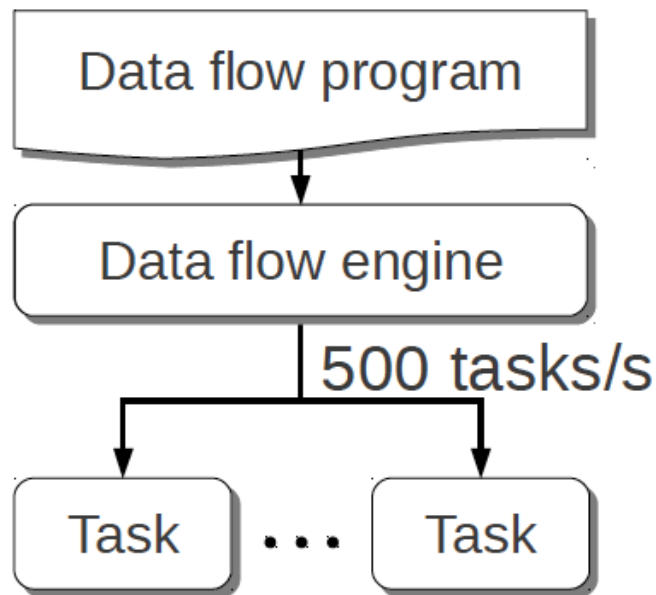
Lives inside client JVM, or standalone service on client host, or at compute site service node,

Workers run tasks on compute nodes

Runs on compute node, 1 per node or one per CPU or CPU group

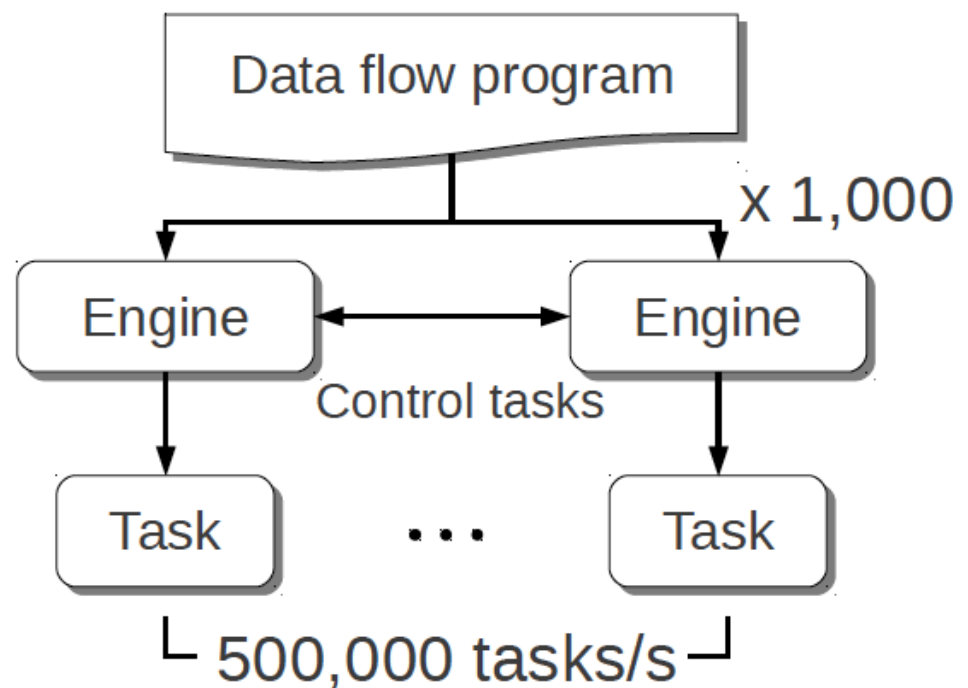
# Centralized evaluation can be a bottleneck at extreme scales

Had this (Swift/K):



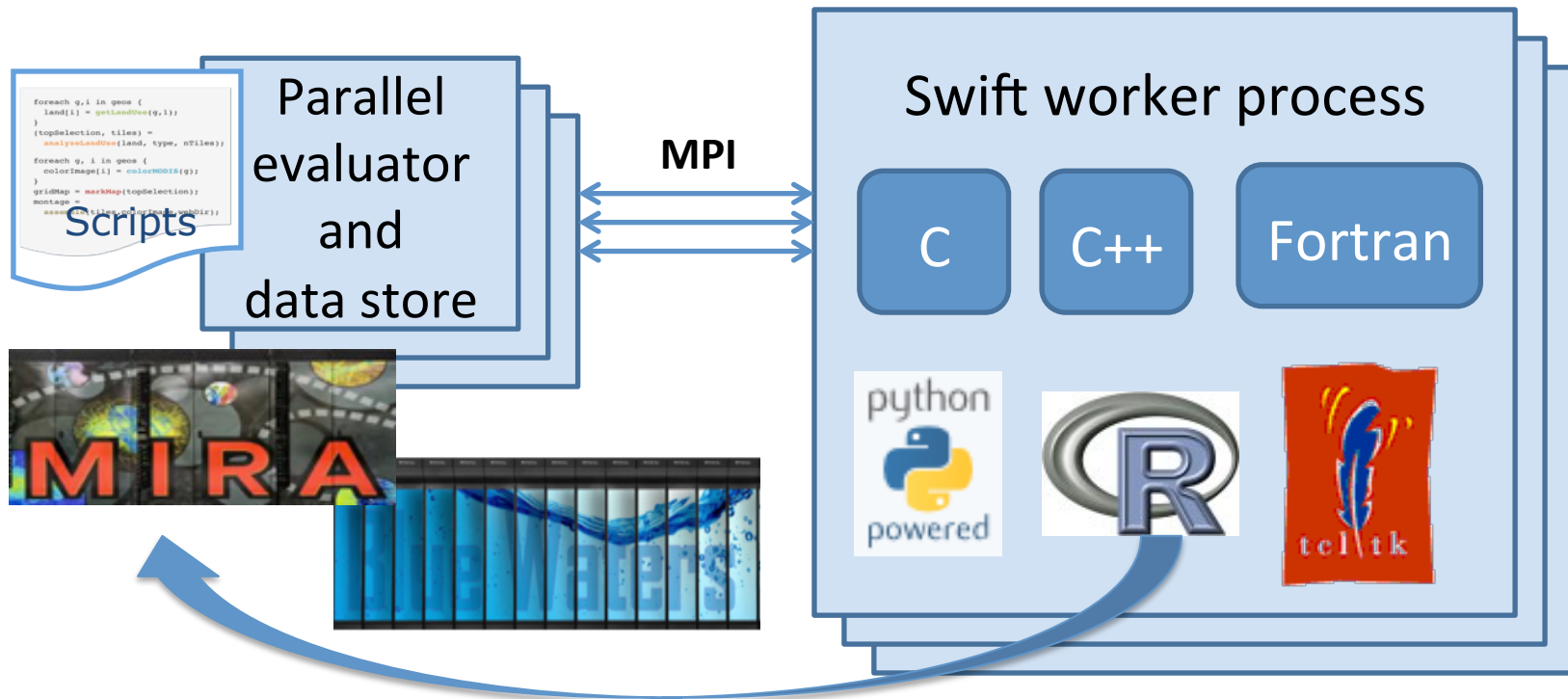
Centralized evaluation

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



Distributed evaluation

# Swift/T: productive extreme-scale scripting

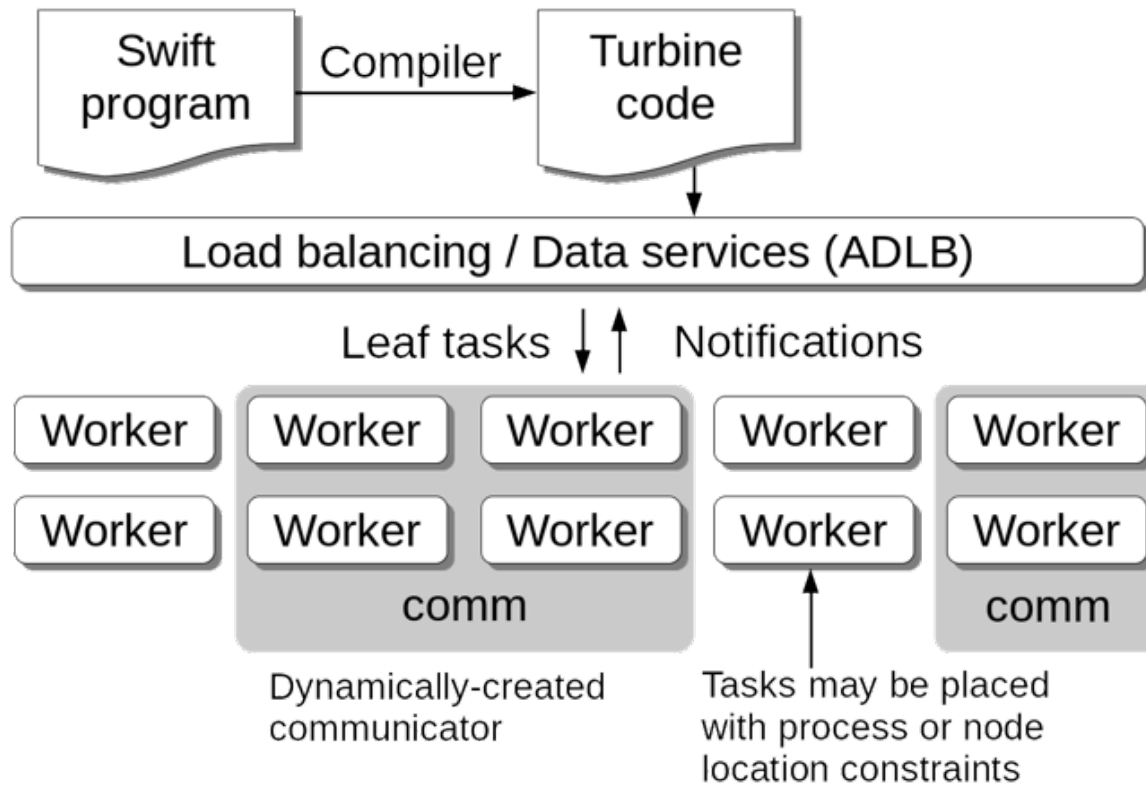


- Script-like programming with “leaf” tasks
  - In-memory function calls in C++, Fortran, Python, R, ... passing in-memory objects
  - More expressive than master-worker for “programming in the large”
  - Leaf tasks can be MPI programs, etc. Can be separate processes if OS permits.
- Distributed, *scalable* runtime manages tasks, load balancing, data movement
- User function calls to external code run on thousands of worker nodes





# Parallel tasks in Swift/T



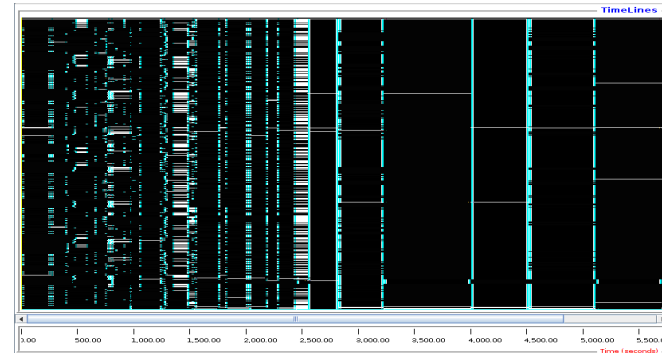
- Swift expression:  $z = @par=32 f(x, y);$
- ADLB server finds 8 available workers
  - Workers receive ranks from ADLB server
  - Performs `comm = MPI_Comm_create_group()`
- Workers perform  $f(x, y)$  communicating on `comm`



# LAMMPS parallel tasks

```
foreach i in [0:20] {  
  t = 300+i;  
  sed_command = sprintf("s/_TEMPERATURE_/%i/g", t);  
  lammps_file_name = sprintf("input-%i.inp", t);  
  lammps_args = "-i " + lammps_file_name;  
  file lammps_input<lammps_file_name> =  
    sed(filter, sed_command) =>  
    @par=8 lammps(lammps_args);  
}
```

- LAMMPS provides a convenient C++ API
- Easily used by Swift/T parallel tasks



Tasks with varying sizes packed into big MPI run  
**Black:** Compute **Blue:** Message **White:** Idle



# Swift/T-specific features

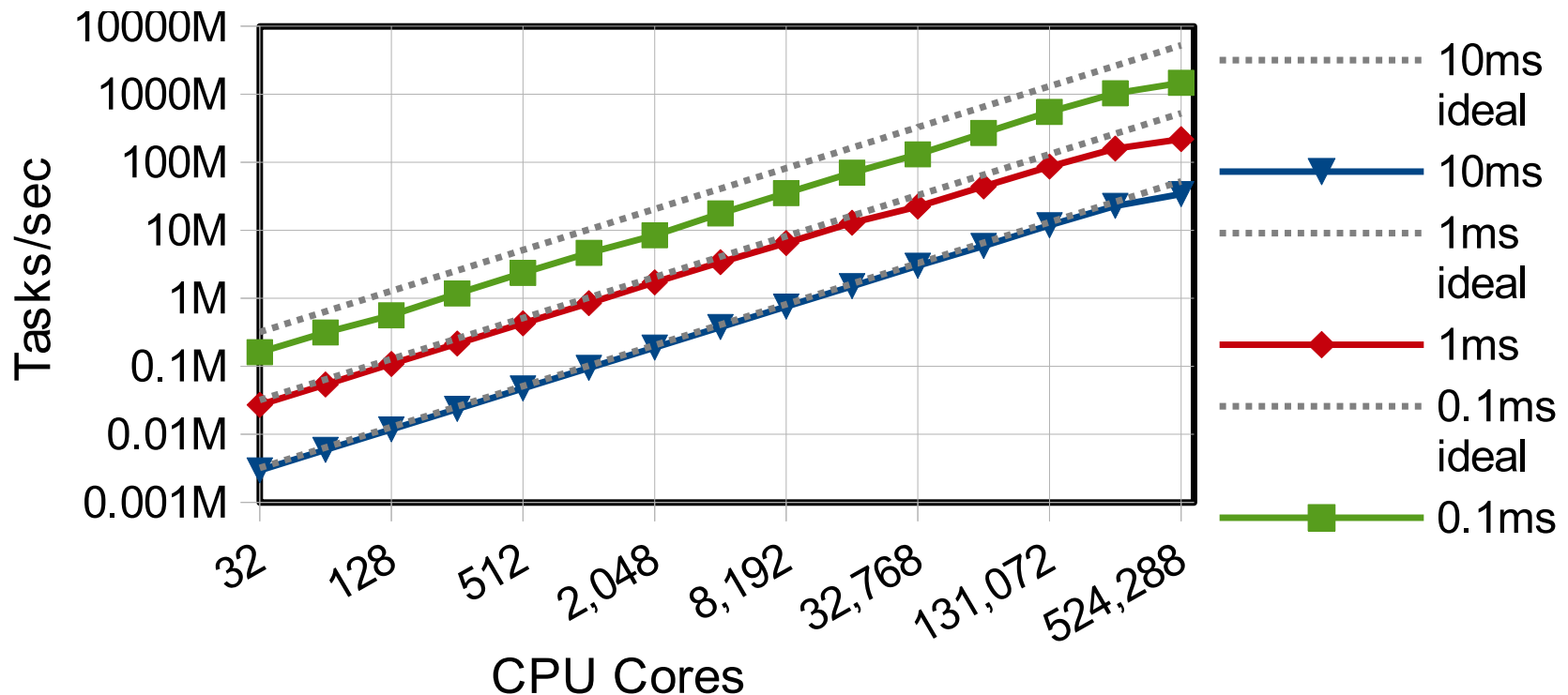
- Task locality: Ability to send a task to a process
  - Allows for big data –type applications
  - Allows for stateful objects to remain resident in the workflow
  - `location L = find_data(D);`  
`int y = @location=L f(D, x);`
- Data broadcast
- Task priorities: Ability to set task priority
  - Useful for tweaking load balancing
- Updateable variables
  - Allow data to be modified after its initial write
  - Consumer tasks may receive original or updated values when they emerge from the work queue

Wozniak et al. Language features for scalable distributed-memory dataflow computing. Proc. Dataflow Execution Models at PACT, 2014.



# Swift/T: scaling of trivial foreach { } loop

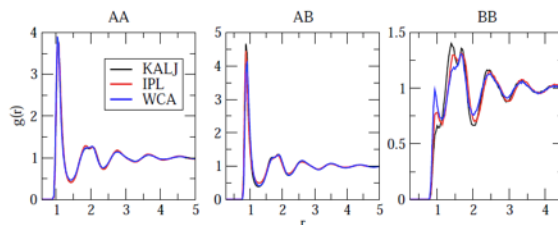
100 microsecond to 10 millisecond tasks  
on up to 512K integer cores of Blue Waters



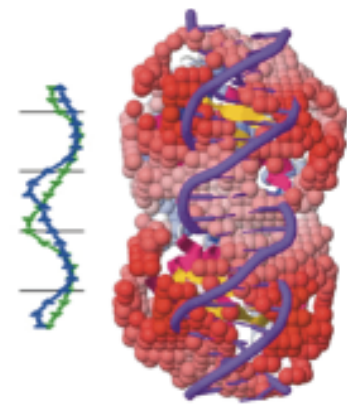
# Large-scale applications using Swift

- A** Simulation of super-cooled glass materials
- B** Protein and biomolecule structure and interaction
- C** Climate model analysis and decision making for global food production & supply
- D** Materials science at the Advanced Photon Source
- E** Multiscale subsurface flow modeling
- F** Modeling of power grid for OE applications

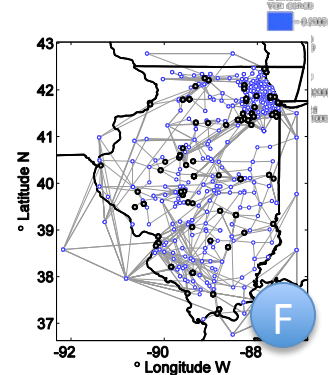
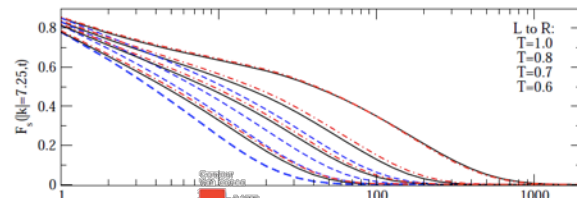
All have published science results obtained using Swift



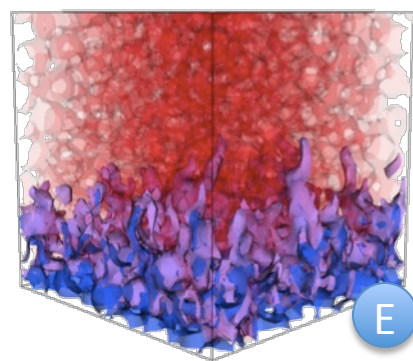
**A**



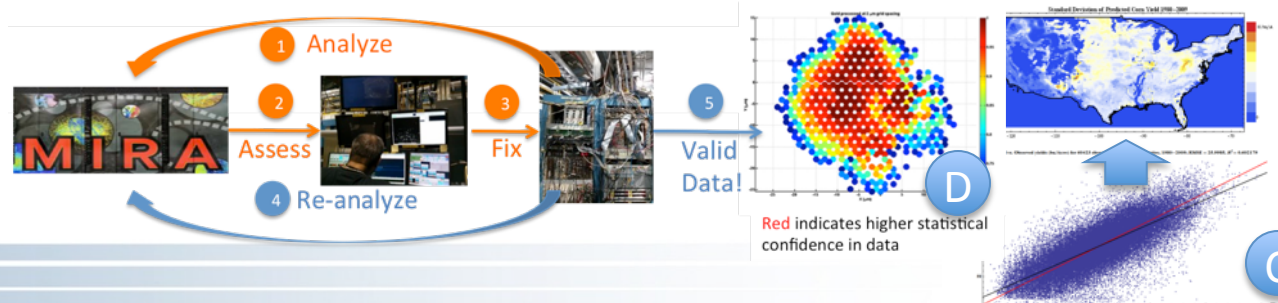
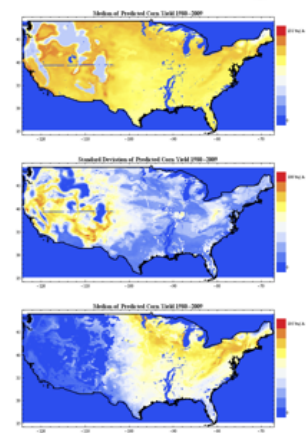
**B**



**F**



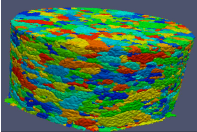
**E**



# Boosting Light Source Productivity with *Swift* ALCF Data Analysis

H Sharma, J Almer (APS); J Wozniak, M Wilde, I Foster (MCS)

## Impact and Approach

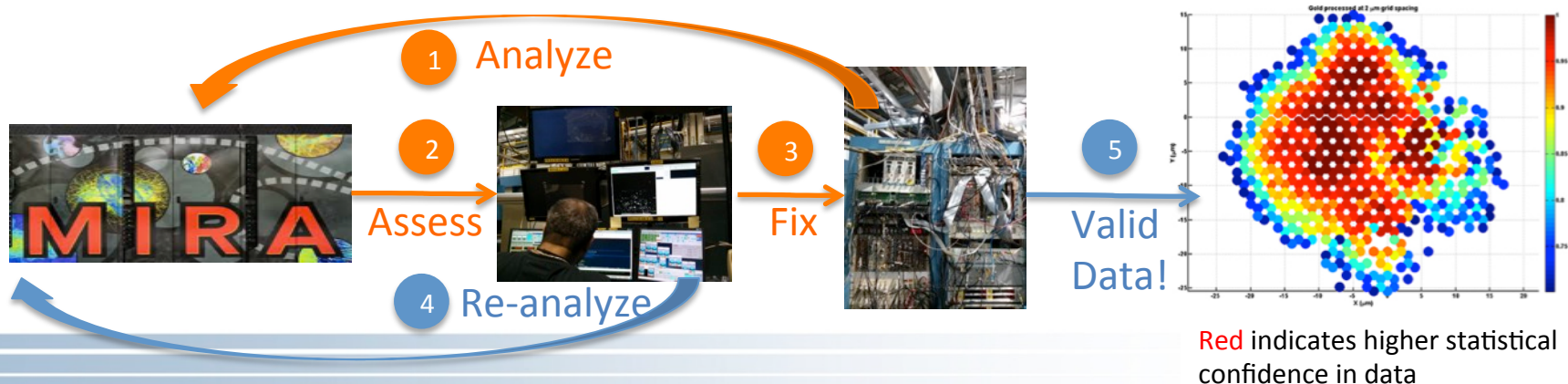
- HEDM imaging and analysis shows granular material structure, non-destructively 
- APS Sector 1 scientists use Mira to process data from live HEDM experiments, providing real-time feedback to correct or improve in-progress experiments
- Scientists working with *Discovery Engines* LDRD developed new *Swift* analysis workflows to process APS data from Sectors 1, 6, and 11

## Accomplishments

- Mira analyzes experiment in 10 mins vs. 5.2 hours on APS cluster: > 30X improvement
- Scaling up to ~ 128K cores (driven by data features)
- **Cable flaw was found and fixed at start of experiment**, saving an entire multi-day experiment and valuable user time and APS beam time.
- **In press:** *High-Energy Synchrotron X-ray Techniques for Studying Irradiated Materials*, J-S Park et al, J. Mat. Res.
- *Big data staging with MPI-IO for interactive X-ray science*, J Wozniak et al, Big Data Conference, Dec 2014

## ALCF Contributions

- Design, develop, support, and trial user engagement to make *Swift* workflow solution on ALCF systems a reliable, secure and supported production service
- Creation and support of the Petrel data server
- Reserved resources on Mira for APS HEDM experiment at Sector 1-ID beamline (8/10/2014 and future sessions in APS 2015 Run 1)



## Conclusion: parallel workflow scripting is practical, productive, *and necessary*, at a broad range of scales

- Swift programming model demonstrated feasible and scalable on XSEDE, Blue Waters, OSG, DOE systems
- Applied to numerous MTC and HPC application domains
  - attractive for data-intensive applications
  - and several hybrid programming models
- Proven productivity enhancement in materials, genomics, biochem, earth systems science, ...
- Deep integration of workflow in progress at XSEDE, ALCF

*Workflow through implicitly parallel dataflow is productive for applications and systems at many scales, including on highest-end system*



# What's next?

- Programmability
  - New patterns ala Van Der Aalst et al ([workflowpatterns.org](http://workflowpatterns.org))
- Fine grained dataflow – programming in the smaller?
  - Run leaf tasks on accelerators (CUDA GPUs, Intel Phi)
  - How low/fast can we drive this model?
- PowerFlow
  - Applies dataflow semantics to manage and reduce energy usage
- Extreme-scale reliability
- Embed Swift semantics in Python, R, Java, shell, make
  - Can we make Swift “invisible”? Should we?
- Swift-Reduce
  - Learning from map-reduce
  - Integration with map-reduce





# GeMTC: GPU-enabled Many-Task Computing

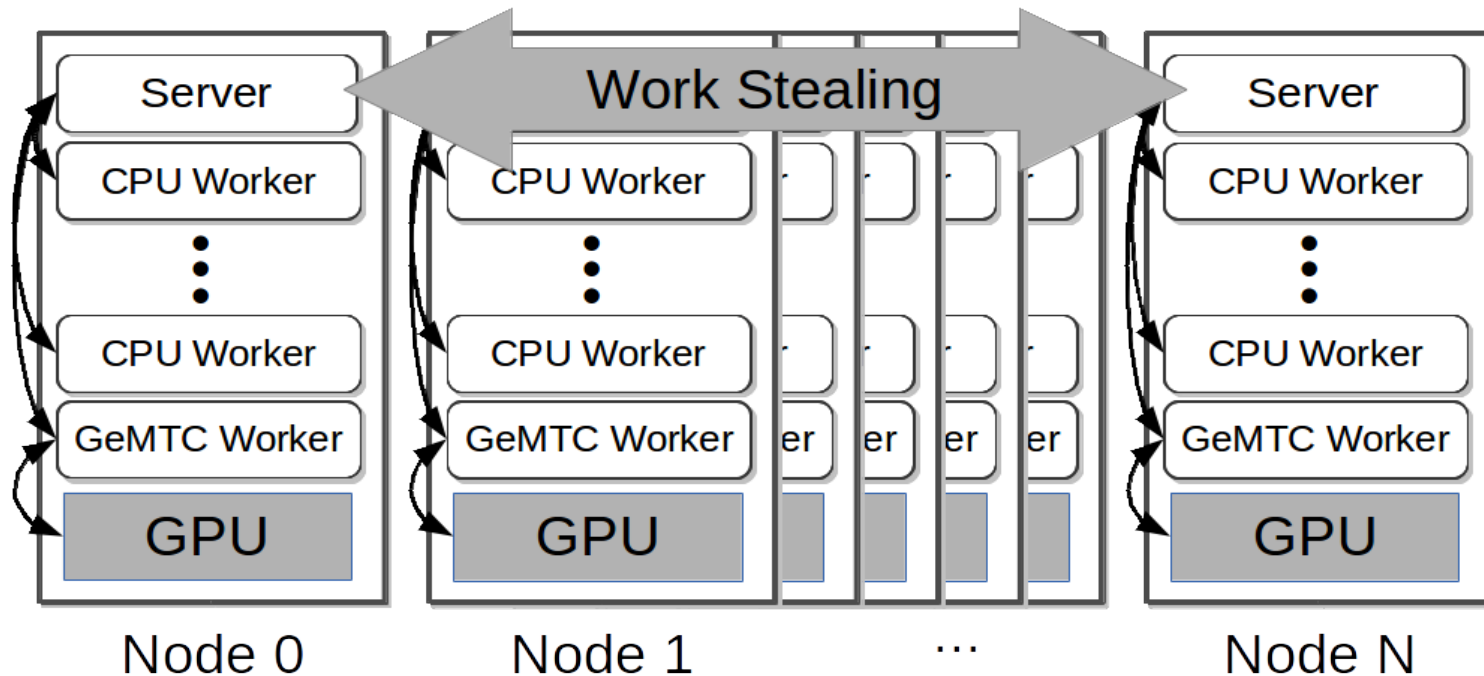
**Motivation:** Support for MTC on all accelerators!

## Goals:

- 1) MTC support
- 2) Programmability
- 3) Efficiency
- 4) MPMD on SIMD
- 5) Increase concurrency to warp level

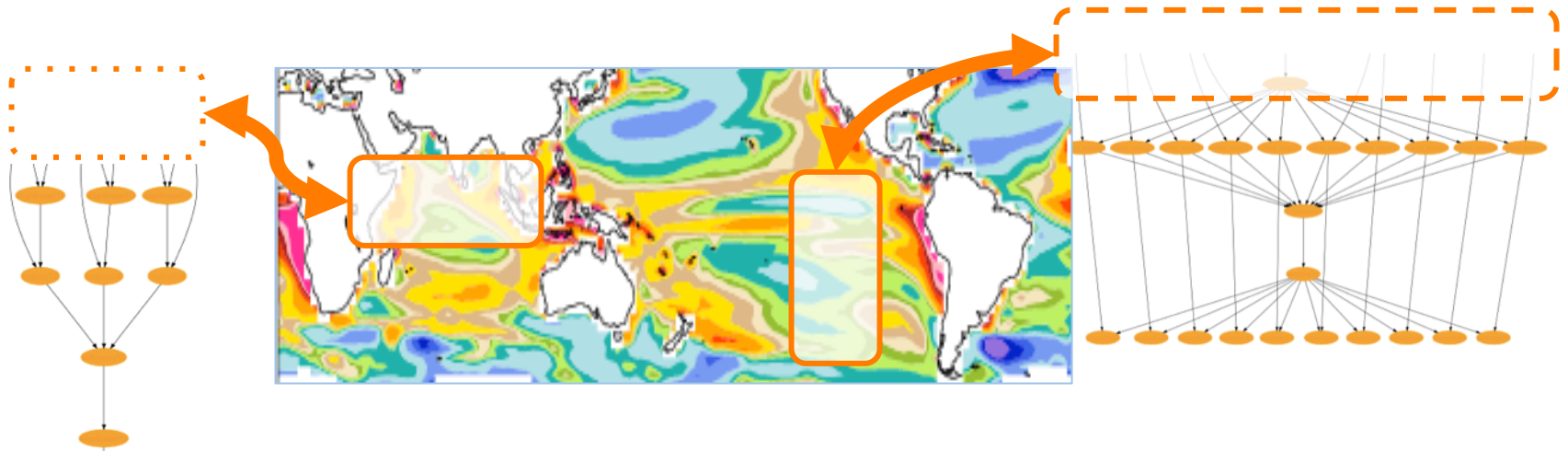
## Approach:

- Design & implement GeMTC middleware:
- 1) Manages GPU
  - 2) Spread host/device
  - 3) Workflow system integration (Swift/T)



# Further research directions

- Deeply in-situ processing for extreme-scale analytics
- Shell-like Read-Evaluate-Print Loop ala iPython
- Debugging of extreme-scale workflows



Deeply in-situ analytics of a climate simulation



*Swift* gratefully acknowledges support from:



U.S. DEPARTMENT OF  
**ENERGY**



THE UNIVERSITY OF  
**CHICAGO**

Argonne

NATIONAL LABORATORY



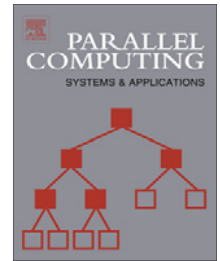
<http://swift-lang.org>

# Supplementary Material



Contents lists available at ScienceDirect

# Parallel Computing

journal homepage: [www.elsevier.com/locate/parco](http://www.elsevier.com/locate/parco)

## Swift: A language for distributed parallel scripting

Michael Wilde<sup>a,b,\*</sup>, Mihael Hategan<sup>a</sup>, Justin M. Wozniak<sup>b</sup>, Ben Clifford<sup>d</sup>, Daniel S. Katz<sup>a</sup>, Ian Foster<sup>a,b,c</sup>

<sup>a</sup> *Computation Institute, University of Chicago and Argonne National Laboratory, United States*

<sup>b</sup> *Mathematics and Computer Science Division, Argonne National Laboratory, United States*

<sup>c</sup> *Department of Computer Science, University of Chicago, United States*

<sup>d</sup> *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.

# Benefit of implicit pervasive parallelism: Analysis & visualization of high-resolution climate models

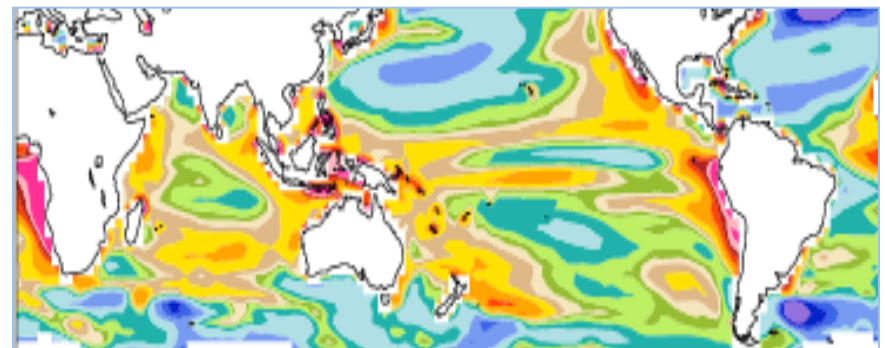
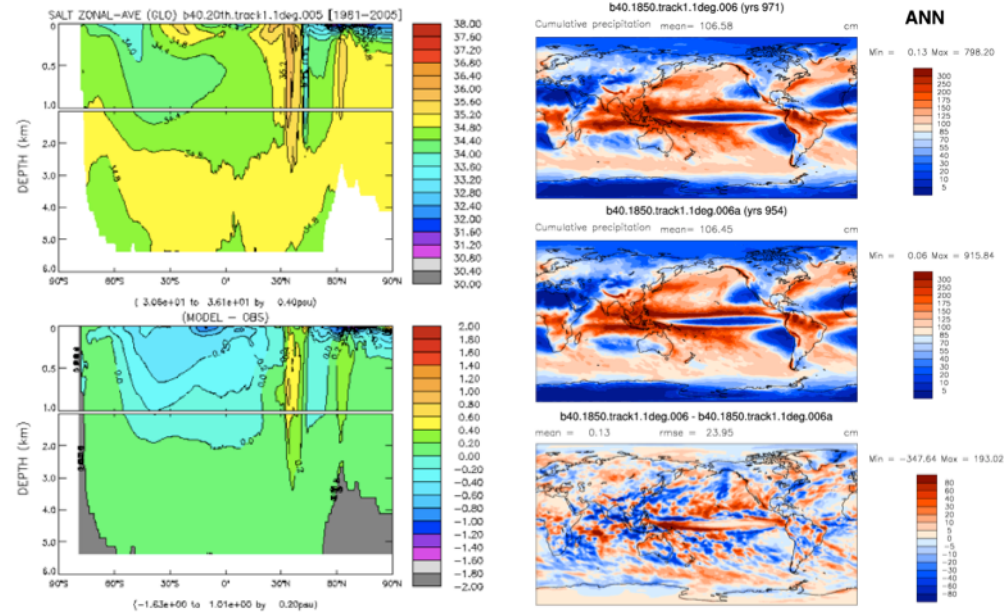
powered by *Swift*



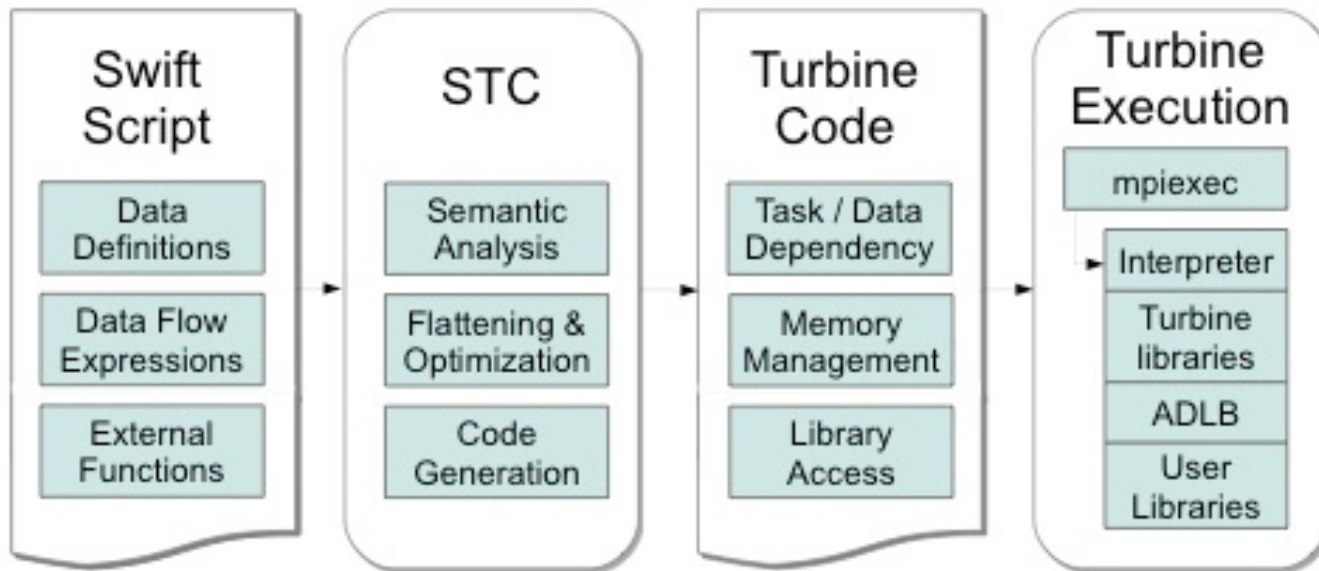
# parvis

- Diagnostic scripts for each climate model (ocean, atmosphere, land, ice) were expressed in complex shell scripts
- Recoded in Swift, the CESM community has benefited from significant speedups and more modular scripts

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



# Swift/T Compiler and Runtime



- STC translates high-level Swift expressions into low-level Turbine operations:
  - Create/Store/Retrieve typed data
  - Manage arrays
  - Manage data-dependent tasks
- Wozniak et al. Large-scale application composition via distributed-memory data flow processing. Proc. CCGrid 2013.
- Armstrong et al. Compiler techniques for massively scalable implicit task parallelism. Proc. SC 2014.

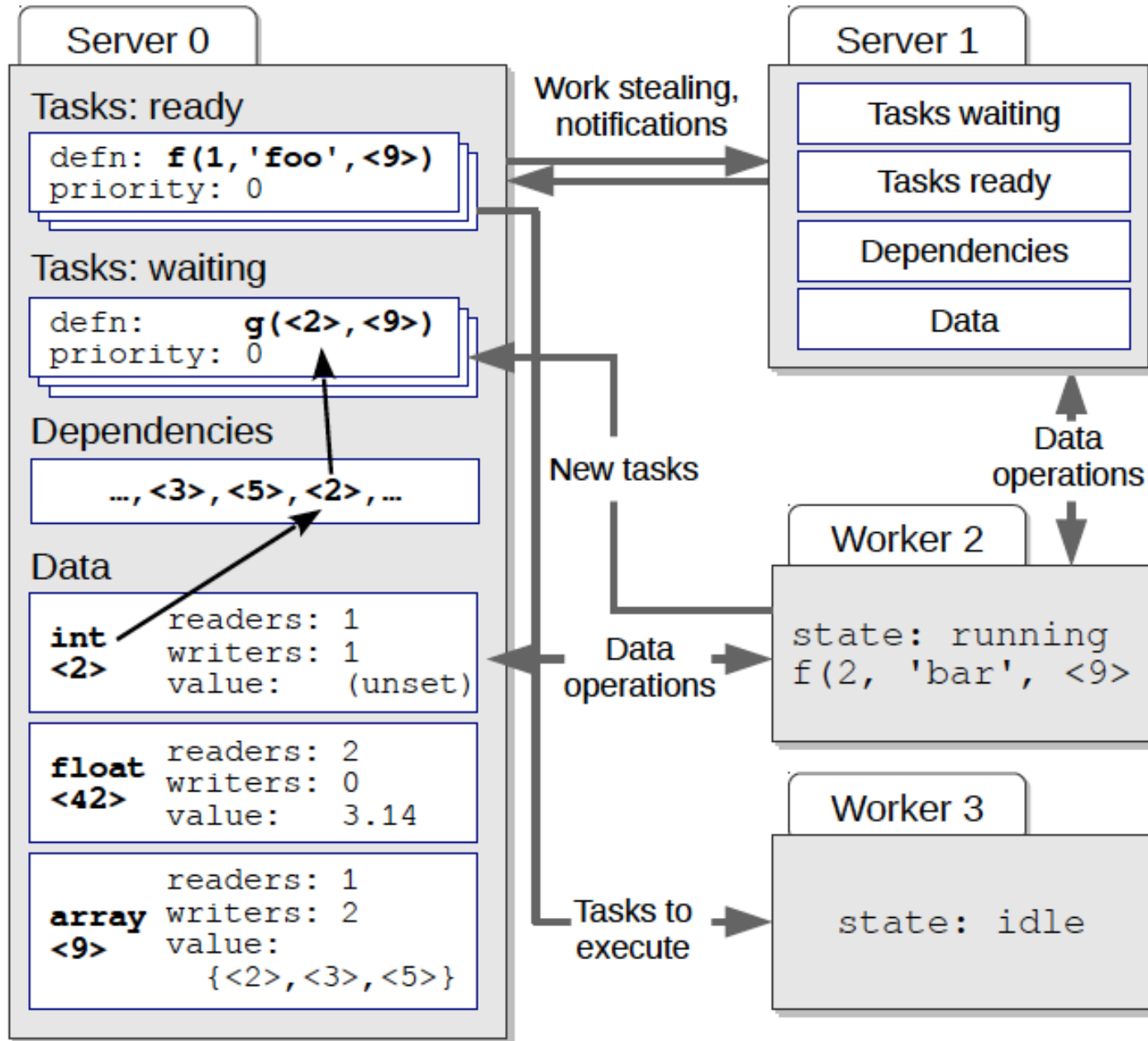
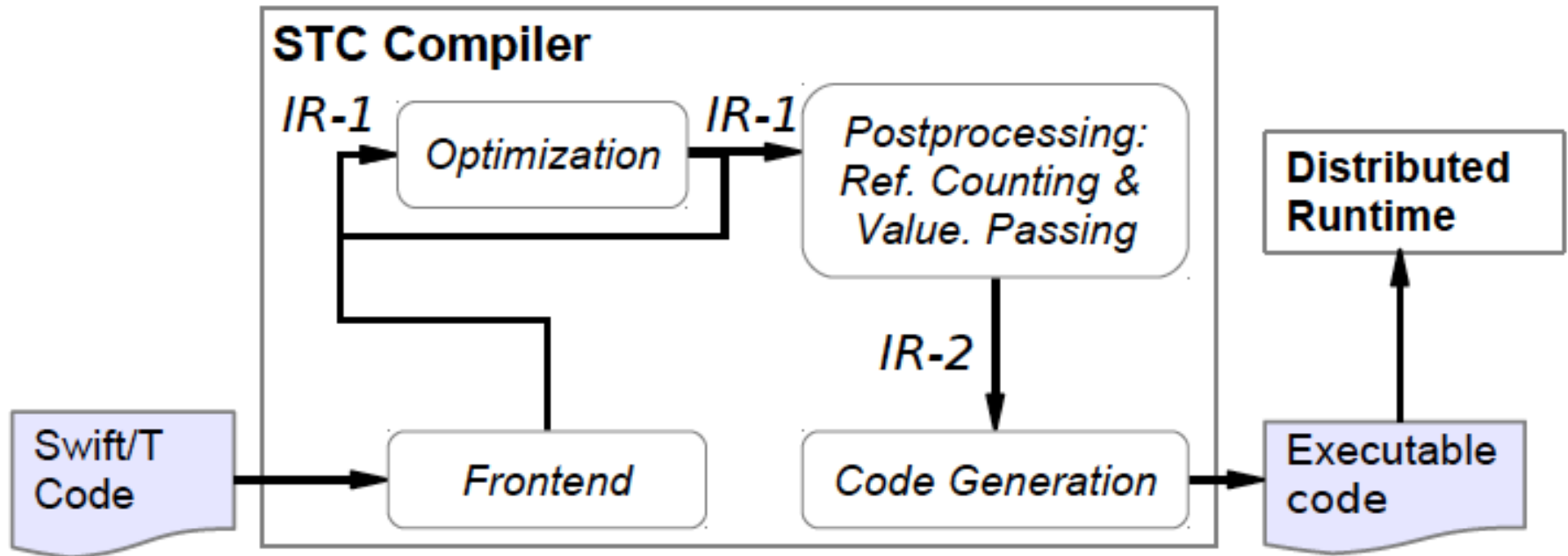


Fig. 4: Runtime architecture showing distributed worker processes coordinating through task and data operations.



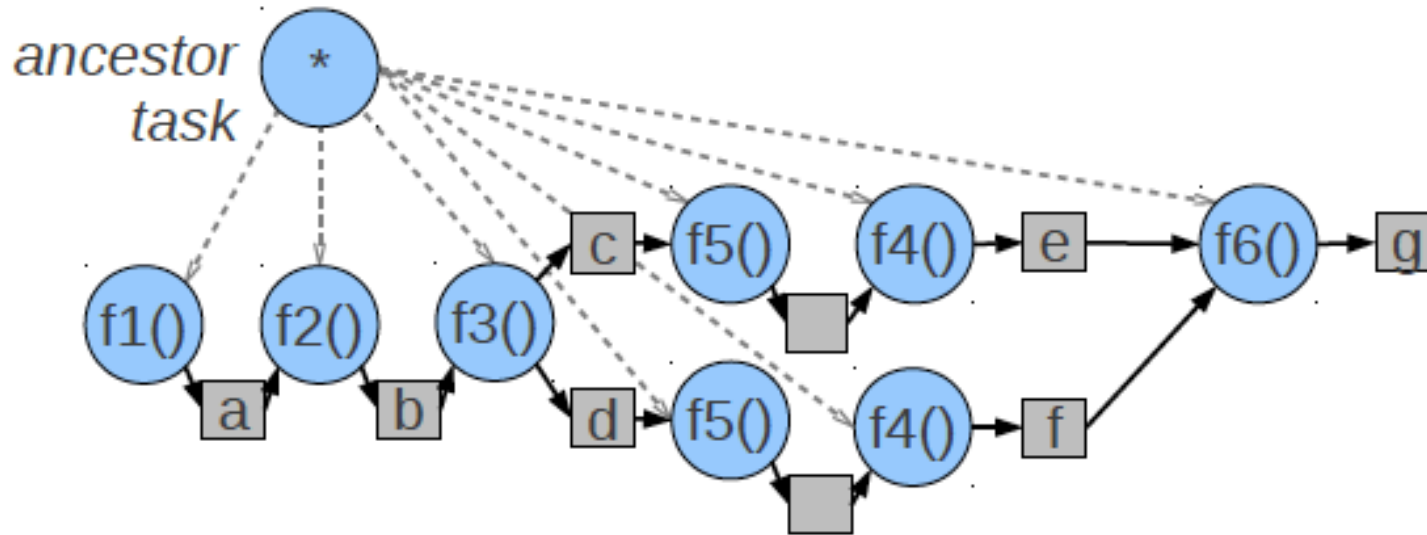
# Swift/T optimizing compiler and IR's



# Swift/T optimization challenge: distributed vars

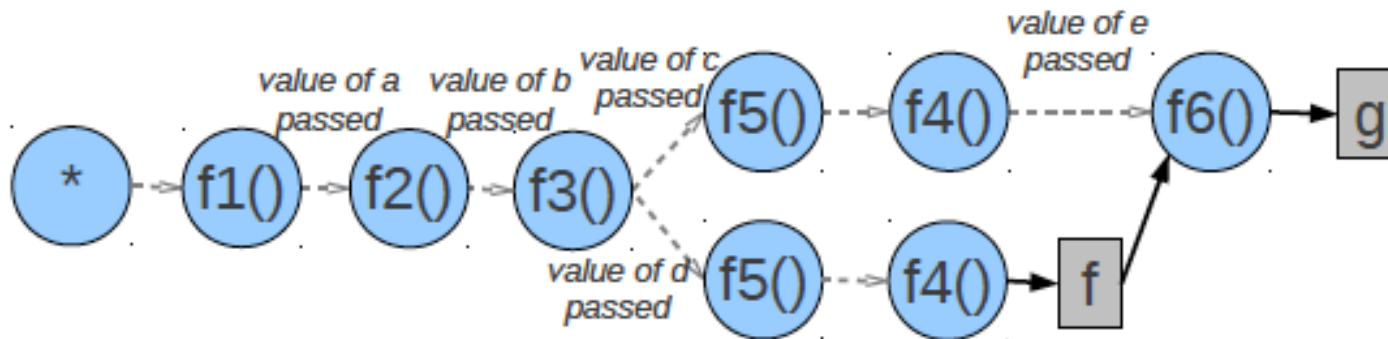
```
1 | a = f1 ();           b = f2 (a);  
2 | c, d = f3 (a, b);   e = f4 (f5 (c));  
3 | f = f4 (f5 (d));    g = f6 (e, f);
```

(a) Swift/T code fragment

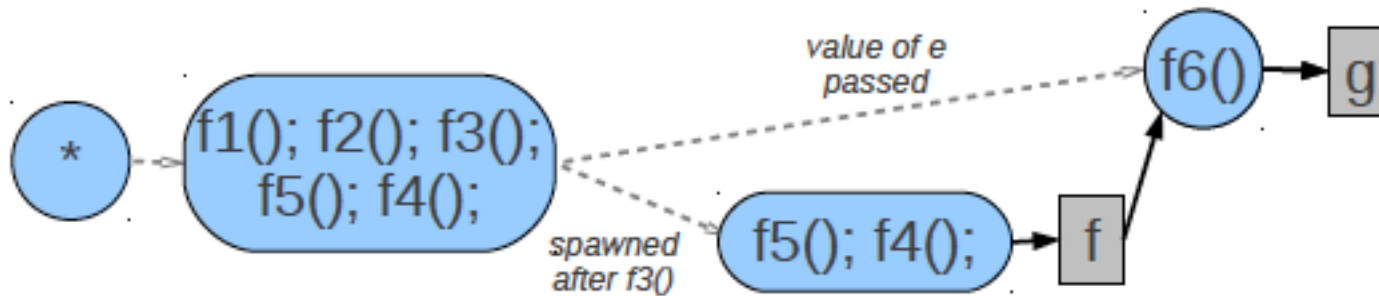


(b) Unoptimized version, passing data as shared data and perform synchronization

# 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



# Swift/T application benchmarks on Blue Waters

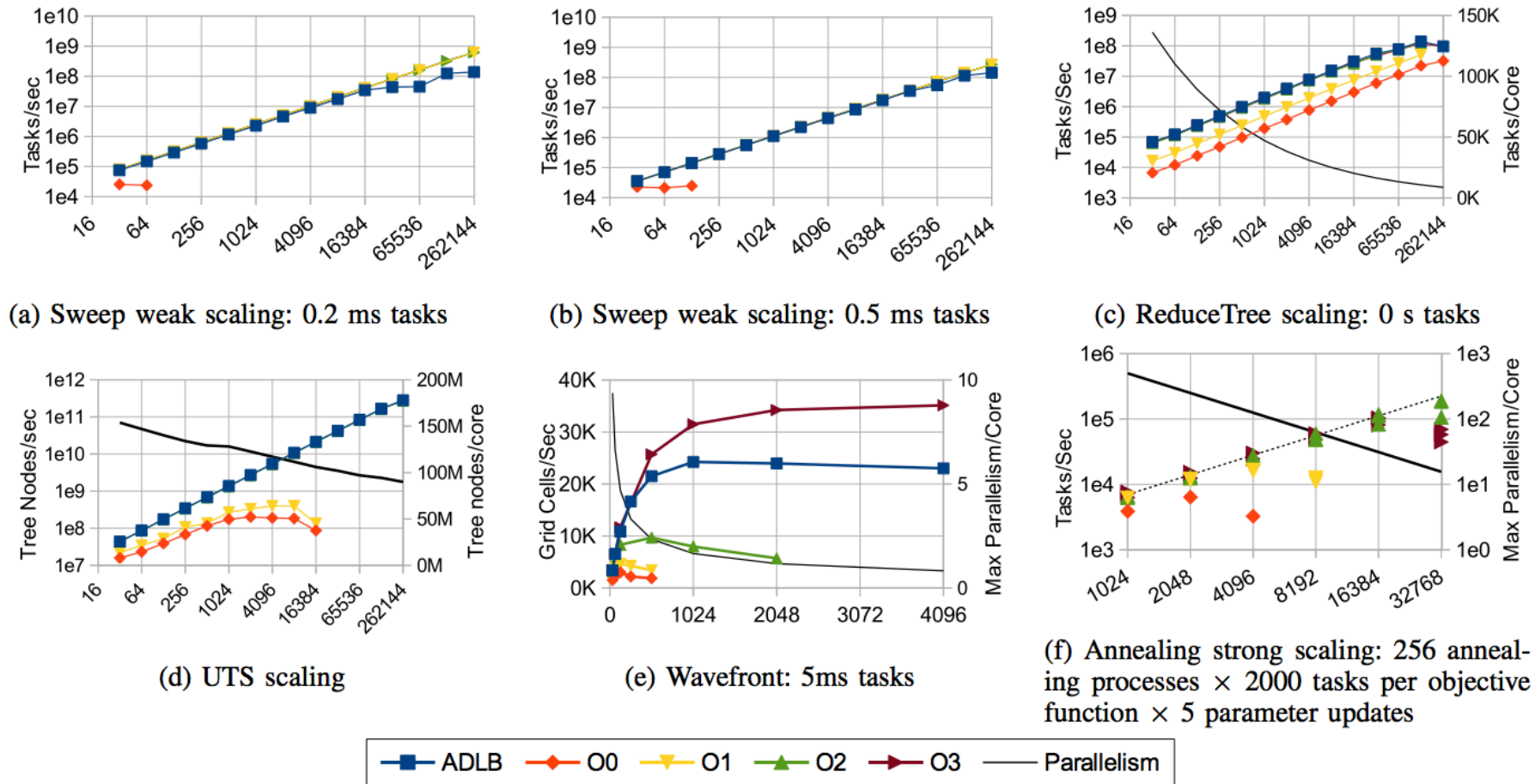


Fig. 10: Application speedup and scalability at different optimization levels. X axes show scale in cores. Primary Y axes show application throughput in application-dependent terms. Secondary Y axes show problem size or degree of parallelism where applicable.