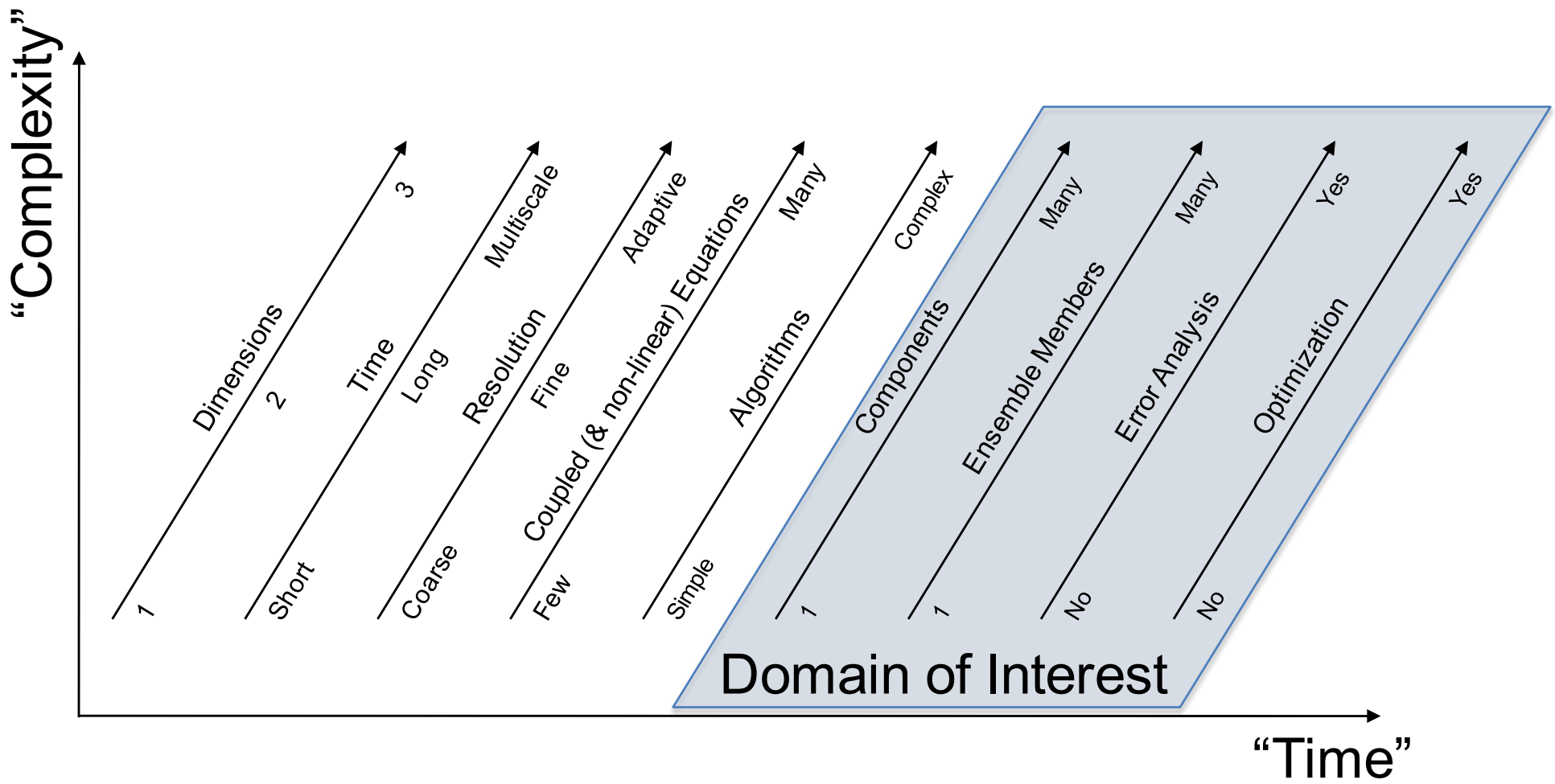
The Swift Parallel Scripting Language for extreme-scale workflow applications

ATPESC 2015 - August 11, 2015

Michael Wilde wilde@anl.gov
Justin Wozniak wozniak@anl.gov

http://swift-lang.org

Increasing capabilities in computational science



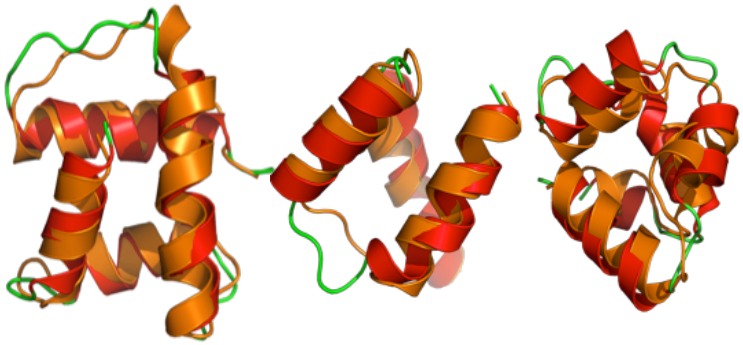
Workflow needs

- Application Drivers
  - Applications that are many-task in nature: parameters sweeps, UQ, inverse modeling, *and data-driven applications*
  - Analysis of capability application outputs
  - Analysis of stored or collected data
  - Increase productivity at major research instrumentation
  - Urgent computing
  - These applications are all *many-task* in nature
- Requirements
  - Usability and ease of workflow expression
  - Ability to leverage complex architecture of HPC and HTC systems (fabric, scheduler, hybrid node and programming models), individually and collectively
  - Ability to integrate high-performance data services and volumes
  - Make use of the system task rate capabilities from clusters to extreme-scale systems
- Approach
  - A programming model for *programming in the large*

When do you need HPC workflow?

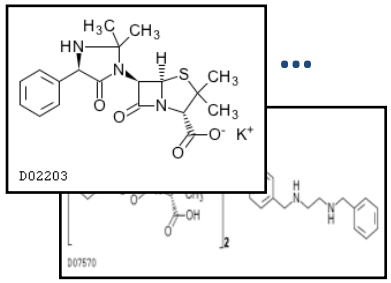
Example application: protein-ligand docking for drug screening

O(10) proteins implicated in a disease

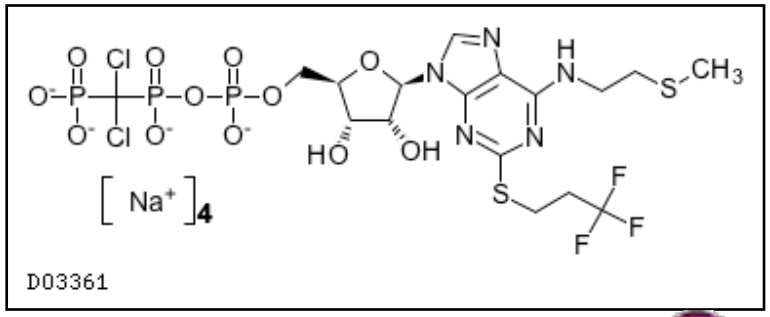


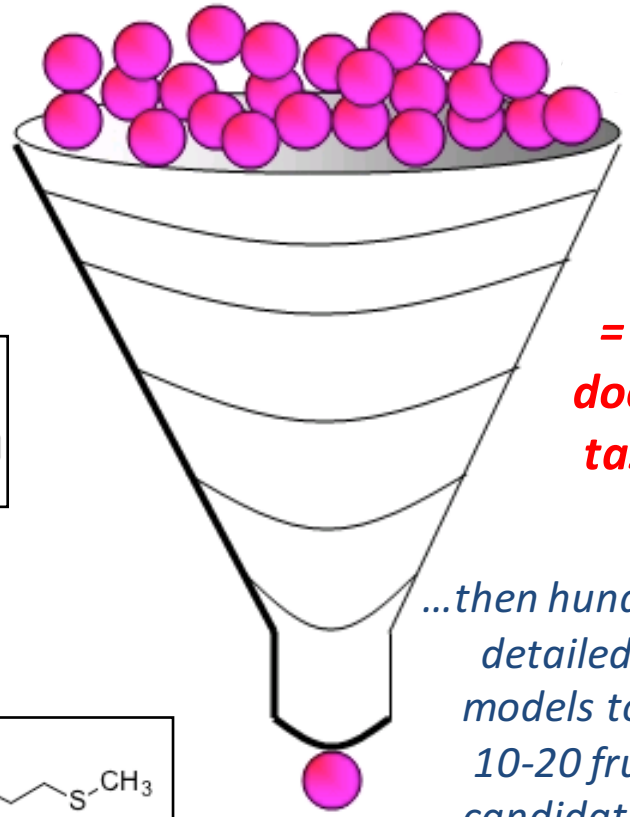
X

O(100K) drug candidates



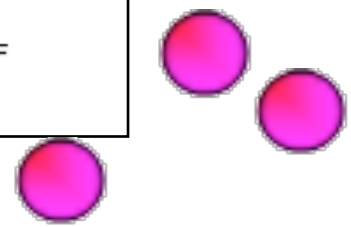
...





= 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

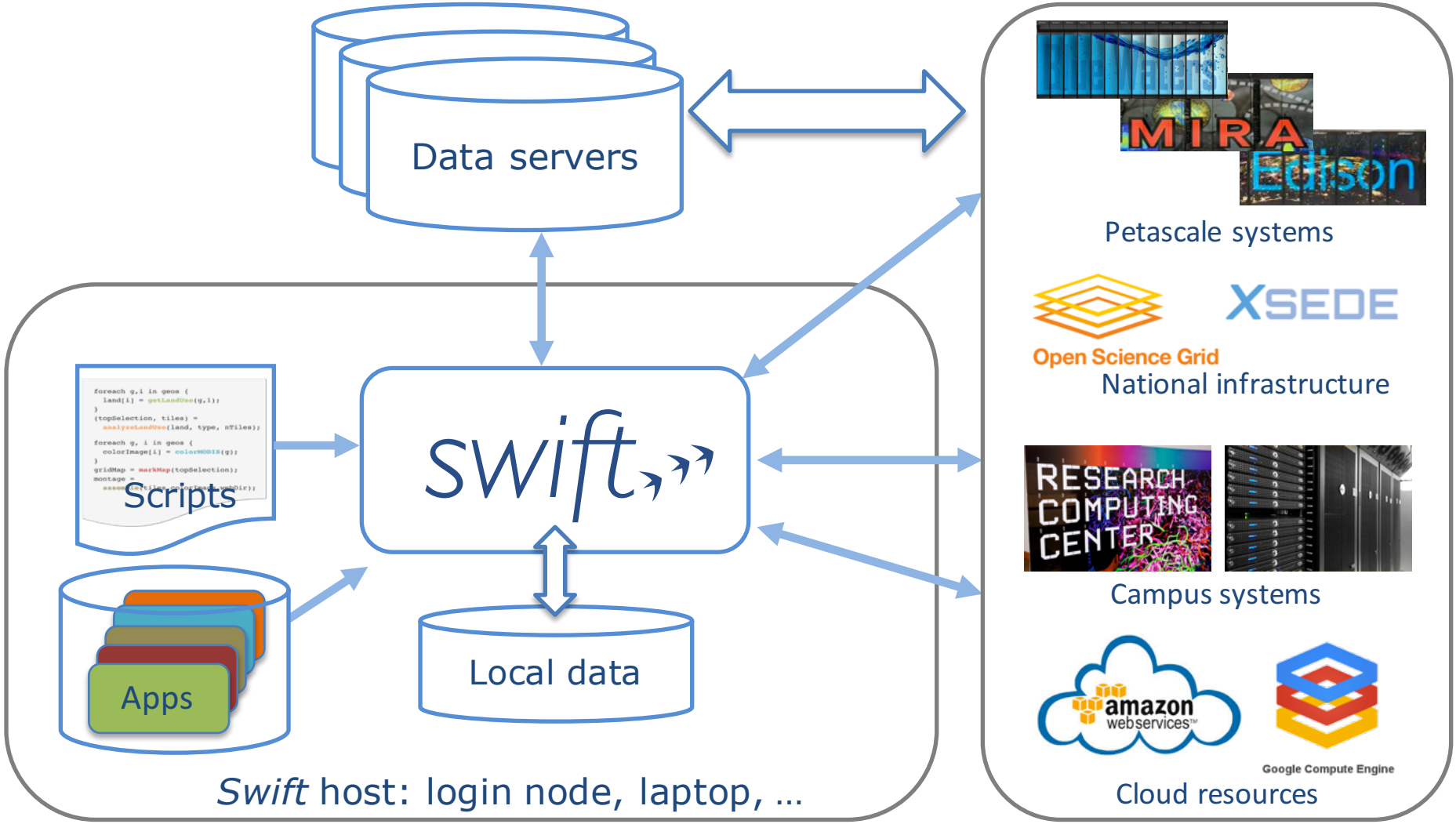
For protein docking workflow:

```
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 fusion,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

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]);
```

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

Swift provides 4 important benefits:

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

Enables provenance capture

Tasks have recordable inputs and outputs

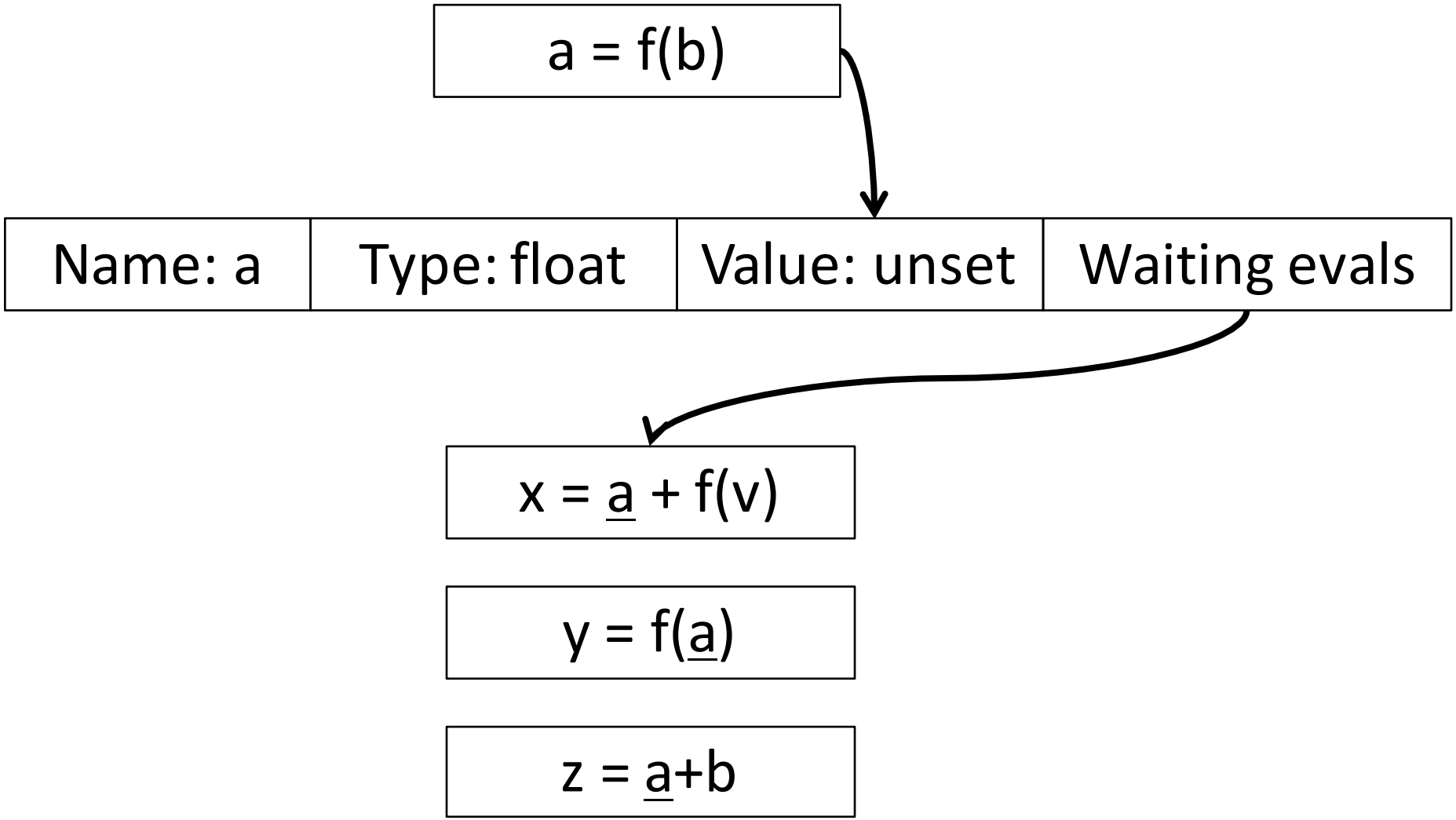
Pervasively parallel

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

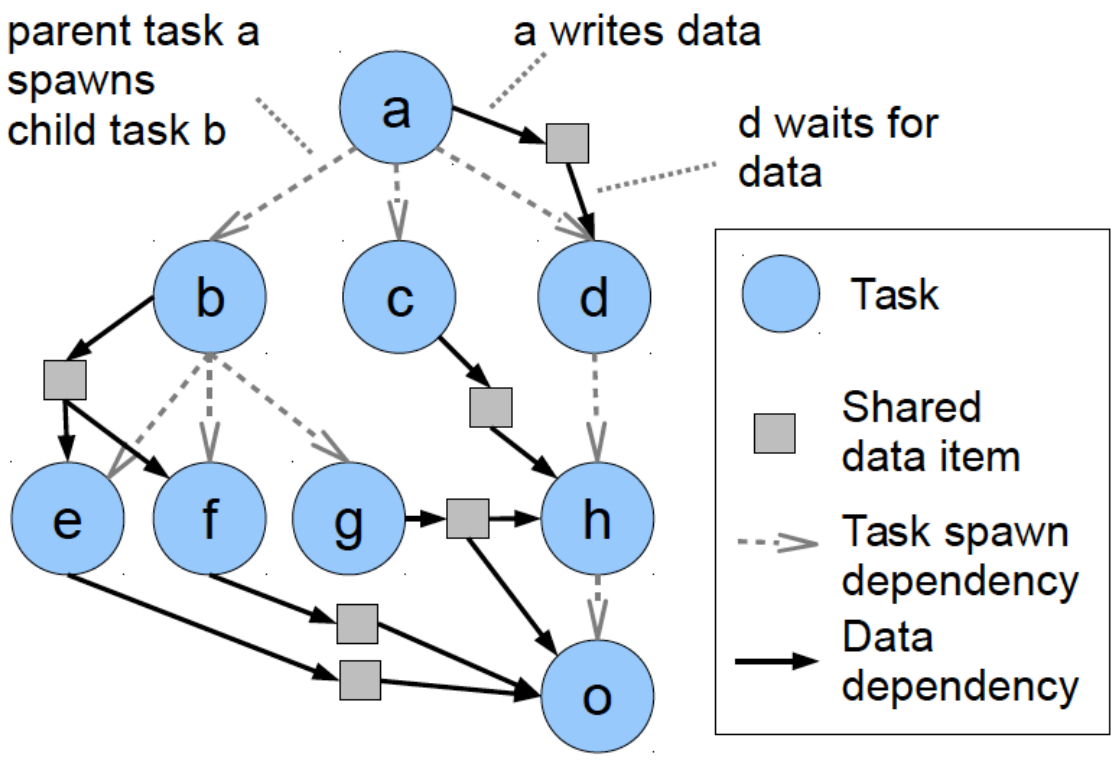
```
(int r) myproc (int i)
{
    int f = F(i);
    int g = G(i);
    r = f + g;
}
```

- 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

All data atoms in Swift are "futures"



Pervasive parallel data flow



Data-intensive example: Processing MODIS land-use data

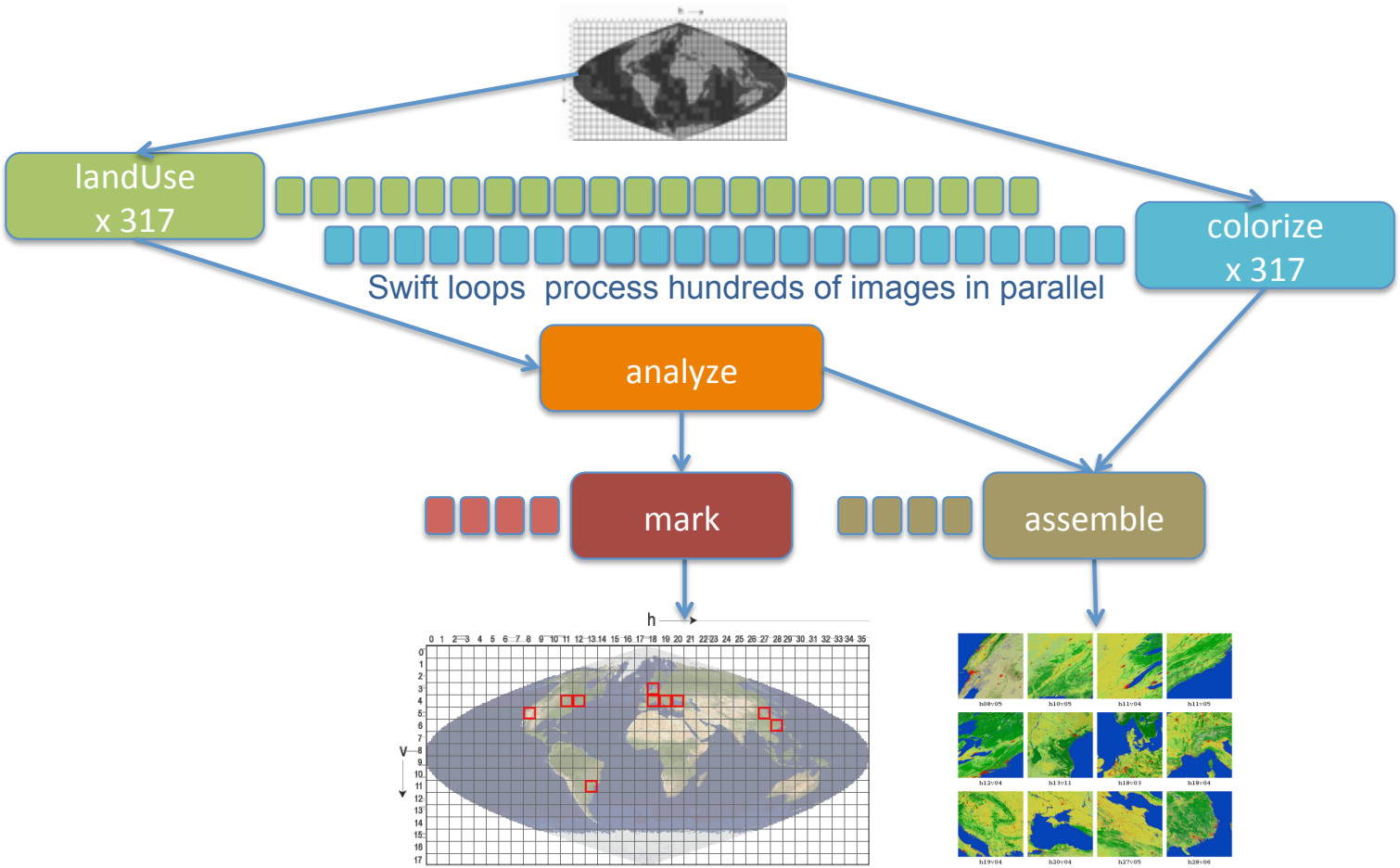


Image processing pipeline for land-use data from the MODIS satellite instrument…

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

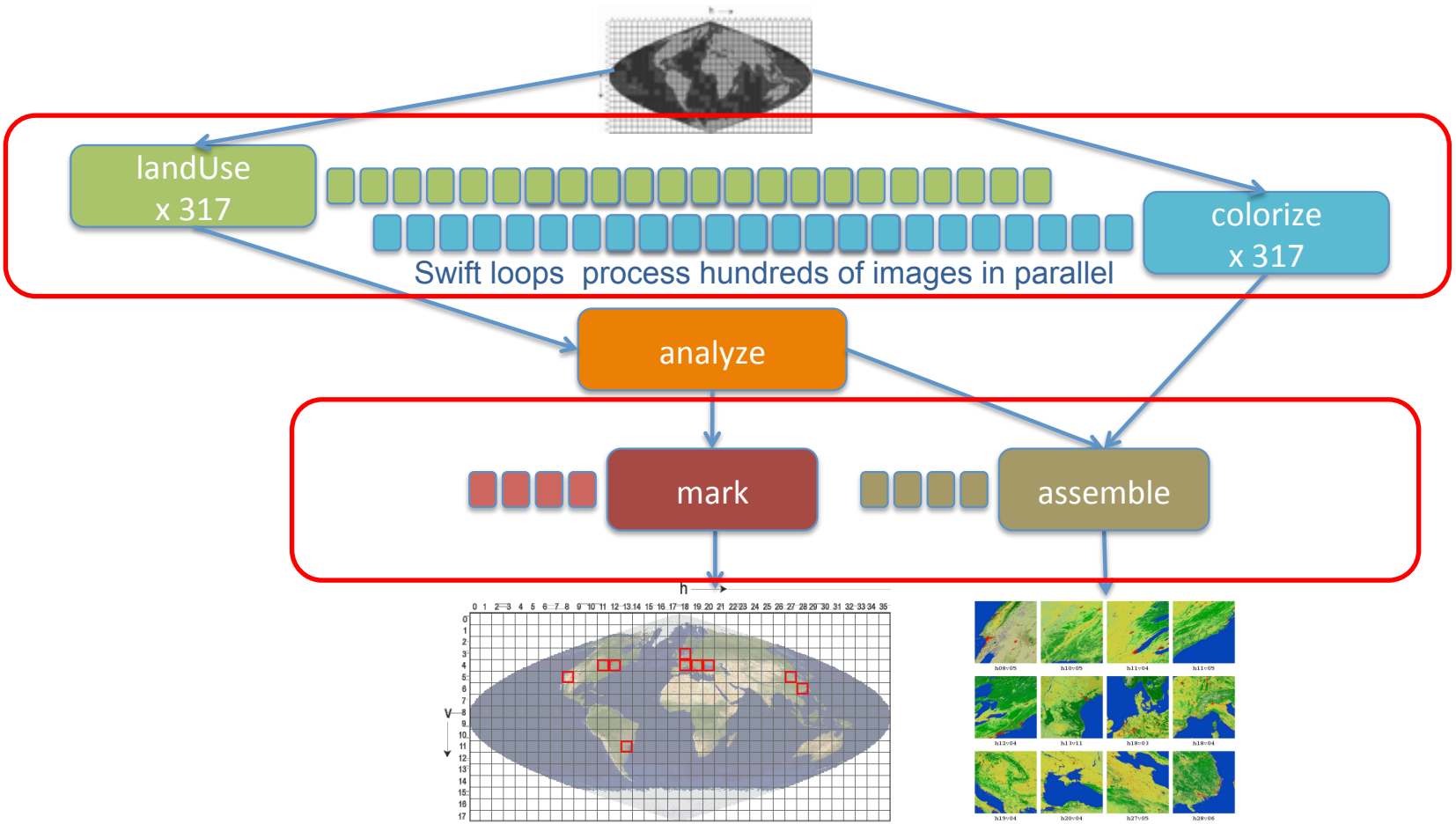
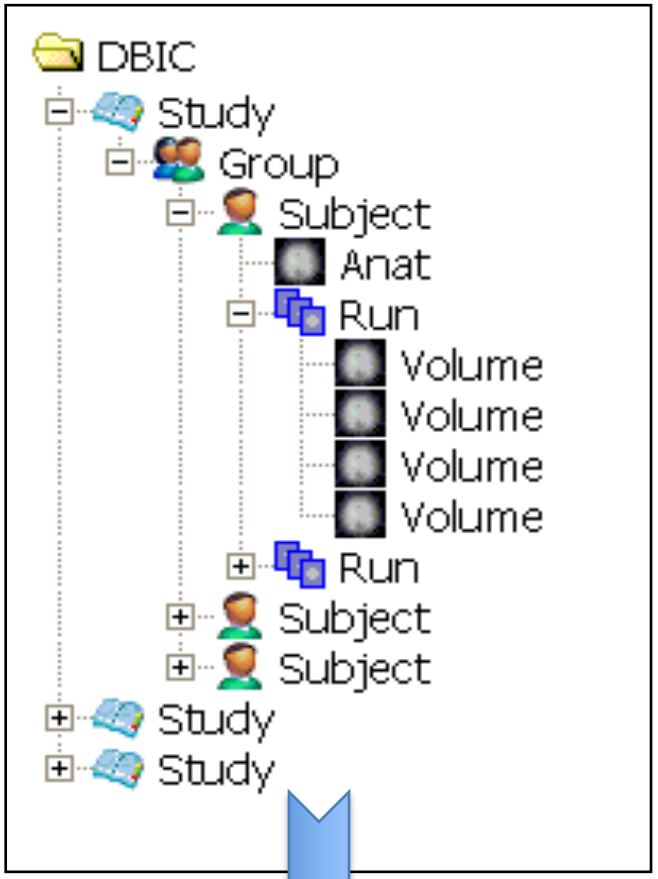


Image processing pipeline for land-use data from the MODIS satellite instrument…

Dataset mapping example: deep fMRI directory tree

On-Disk Data Layout

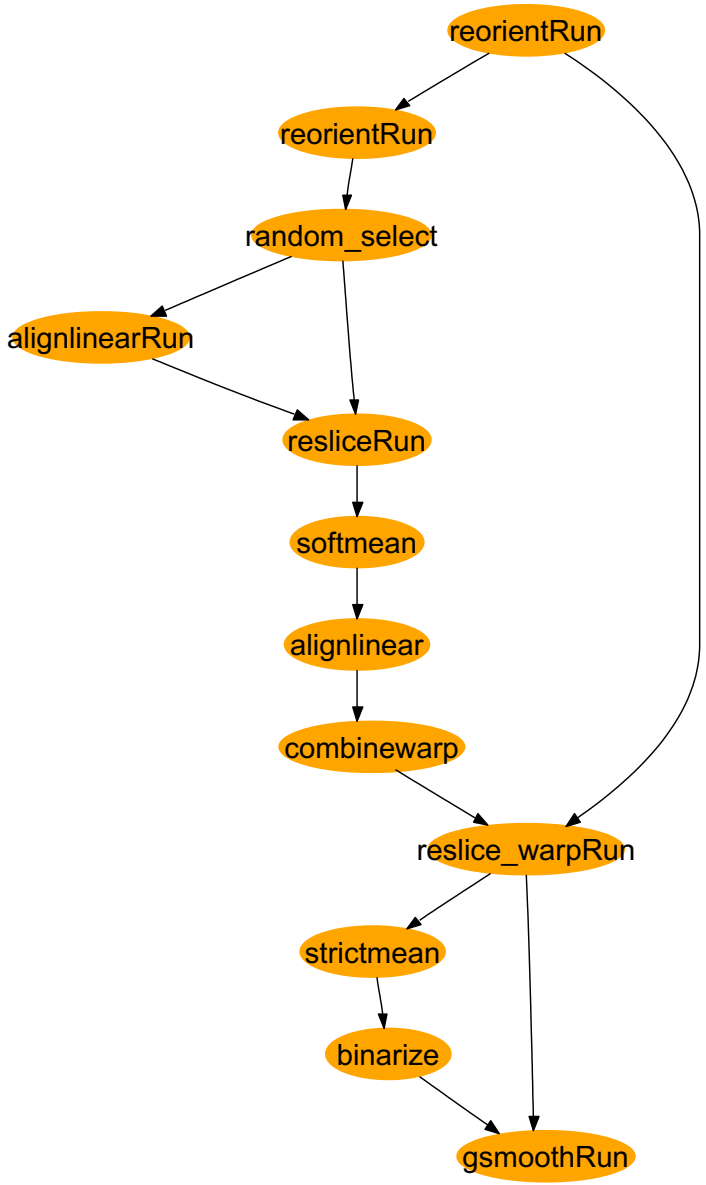


Mapping function or script

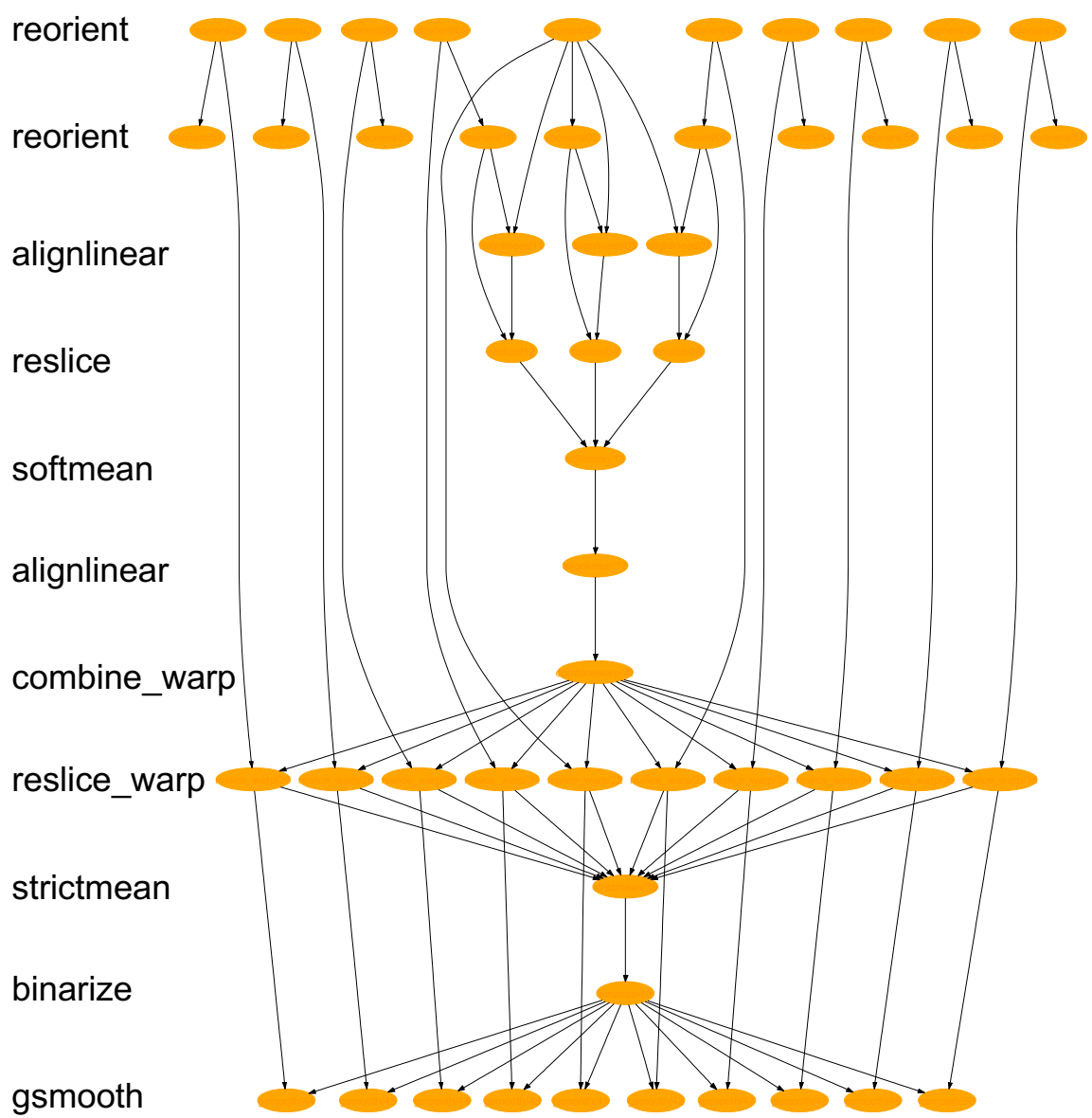
```
type Study {
Group g[ ];
}
type Group {
Subject s[ ];
}
type Subject {
Volume anat;
Run run[ ];
}
type Run {
Volume v[ ];
}
type Volume {
Image img;
Header hdr;
}
```

Swift's in-memory data model

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

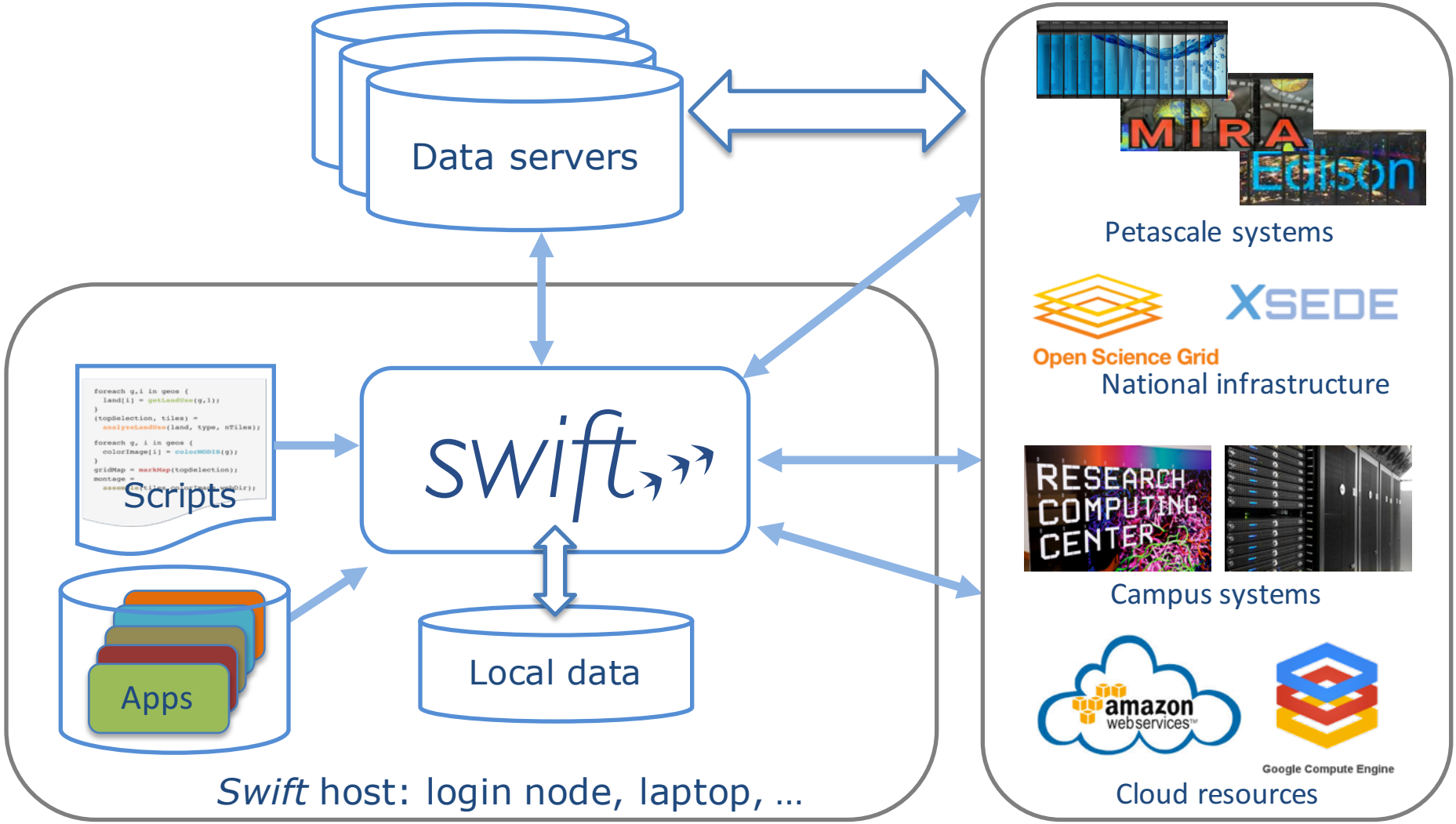
```
(Run snr) functional ( Run r, NormAnat a,
                       Air shrink )
{    Run yroRun = reorientRun( r , "y" );
     Run roRun = reorientRun( yroRun , "x" );
     Volume std = roRun[0];
     Run rndr = random_select( roRun, 0.1 );
     AirVector rndAirVec = align_linearRun( rndr, std, 12, 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);
        }
}
```

Swift flavors

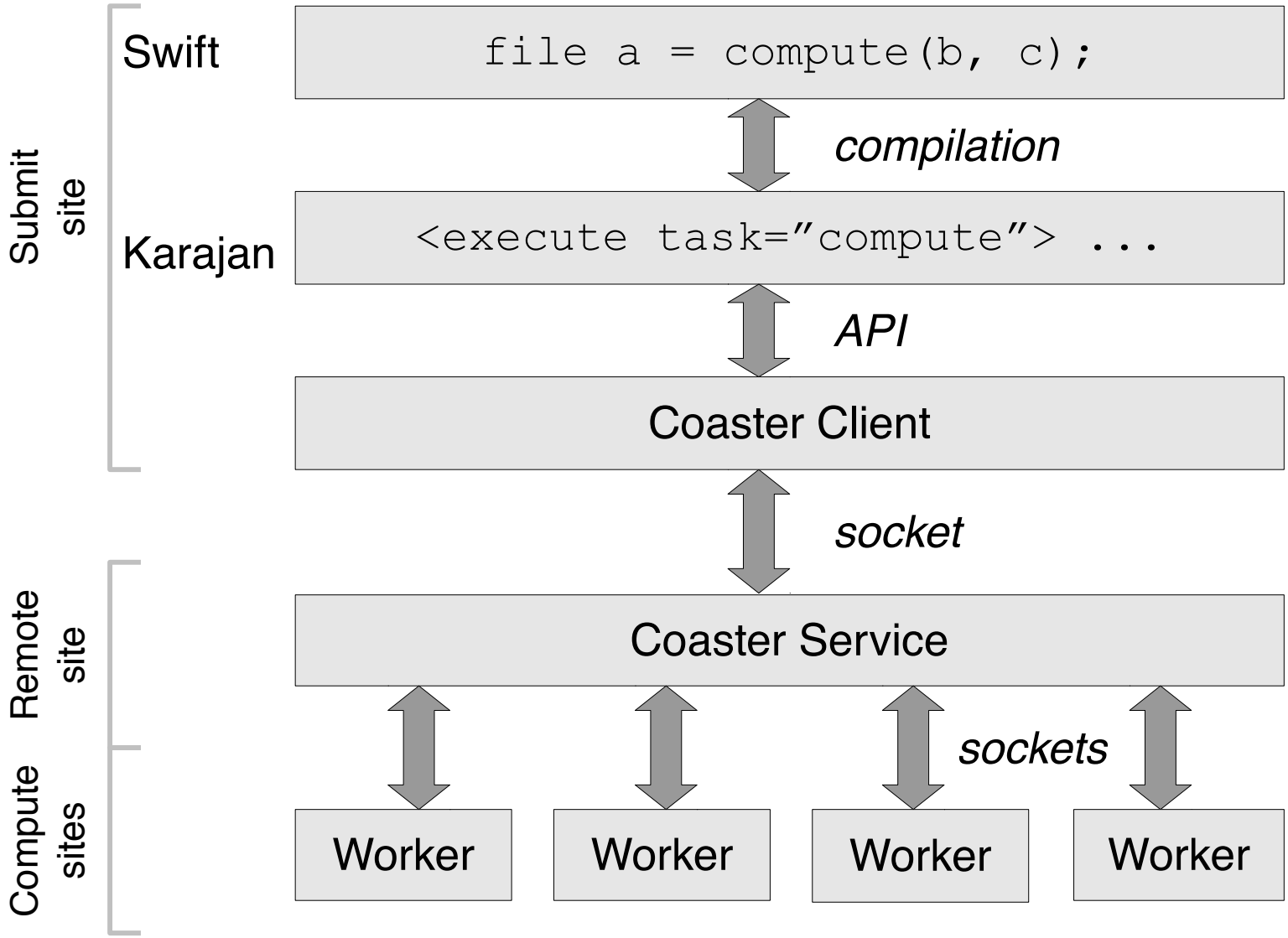
- Swift/K
  - “Classic” Swift based on Java and Karajan
  - Currently used to drive BG/Q subjob workflows
- Swift/T
  - The “HPC” Swift, based on ADLB, MPI, and Turbine
  - Used for high task counts and rates within a single MPI job
  - Developed under ASCR X-Stack program, for extreme-scale
- These will converge into a unified system

Swift's distributed architecture is based on a client/worker mechanism (internally named "coasters")



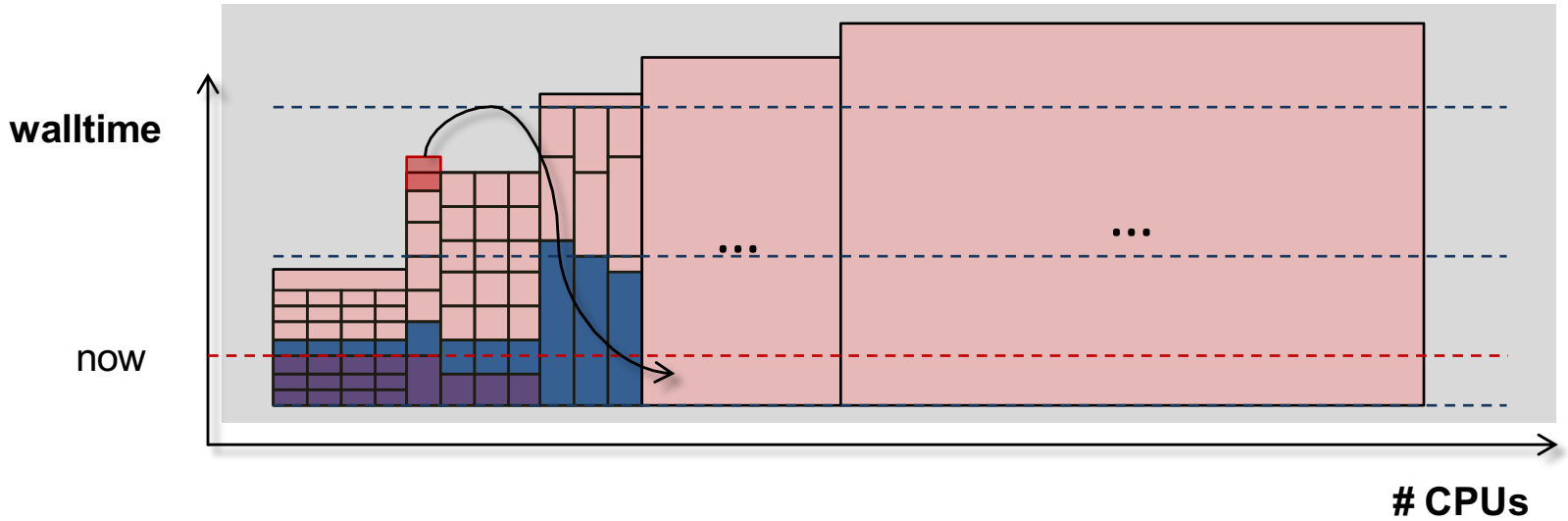
The Swift runtime system has drivers and algorithms to efficiently support and aggregate diverse runtime environments

Worker architecture handles diverse environments



Implementation: The job packing problem (II) (also not to scale)

- Commit jobs to blocks and adjust as necessary based on actual walltime

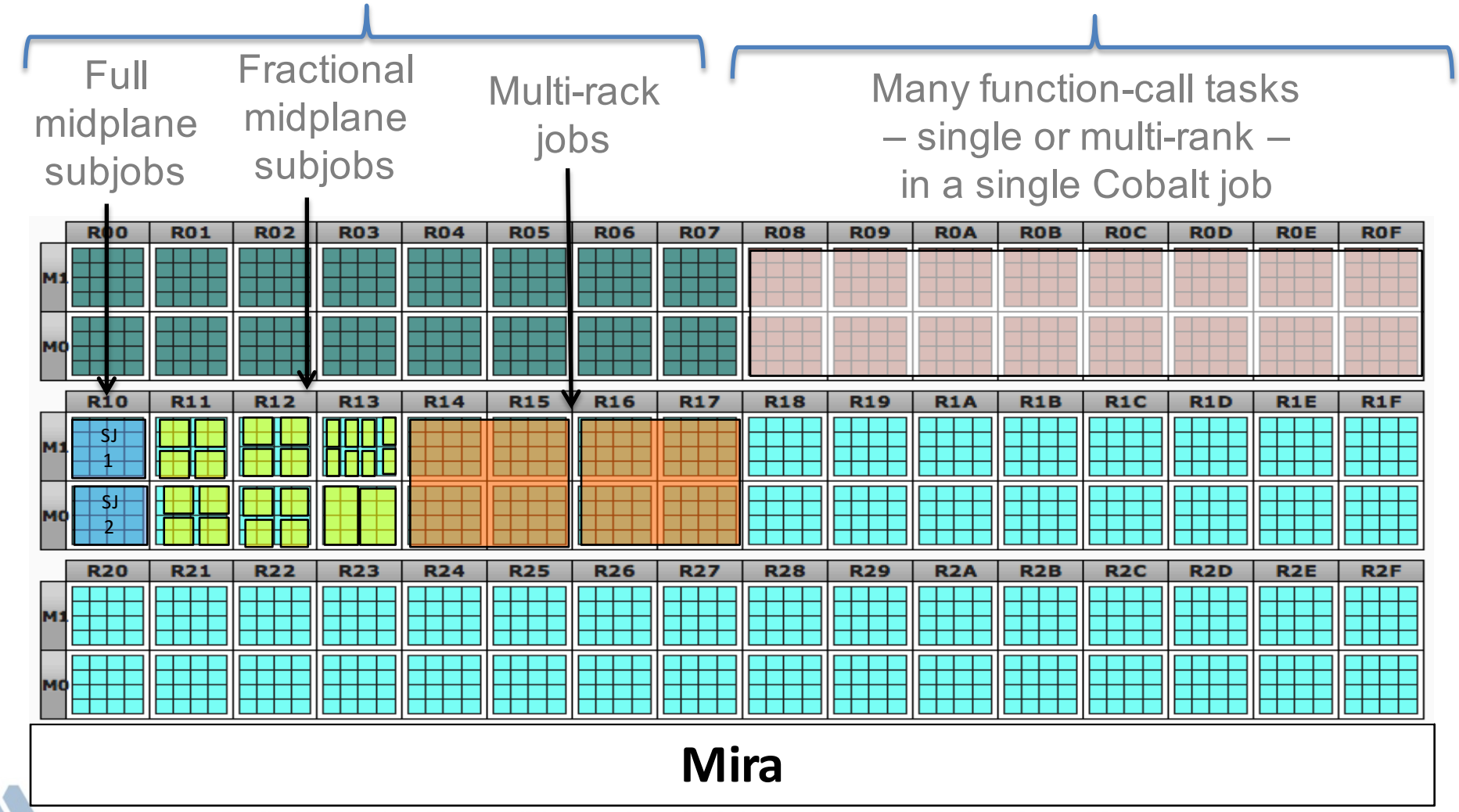


- The actual packing problem is NP-complete
- Solved using a greedy algorithm: always pick the largest job that will fit in a block first

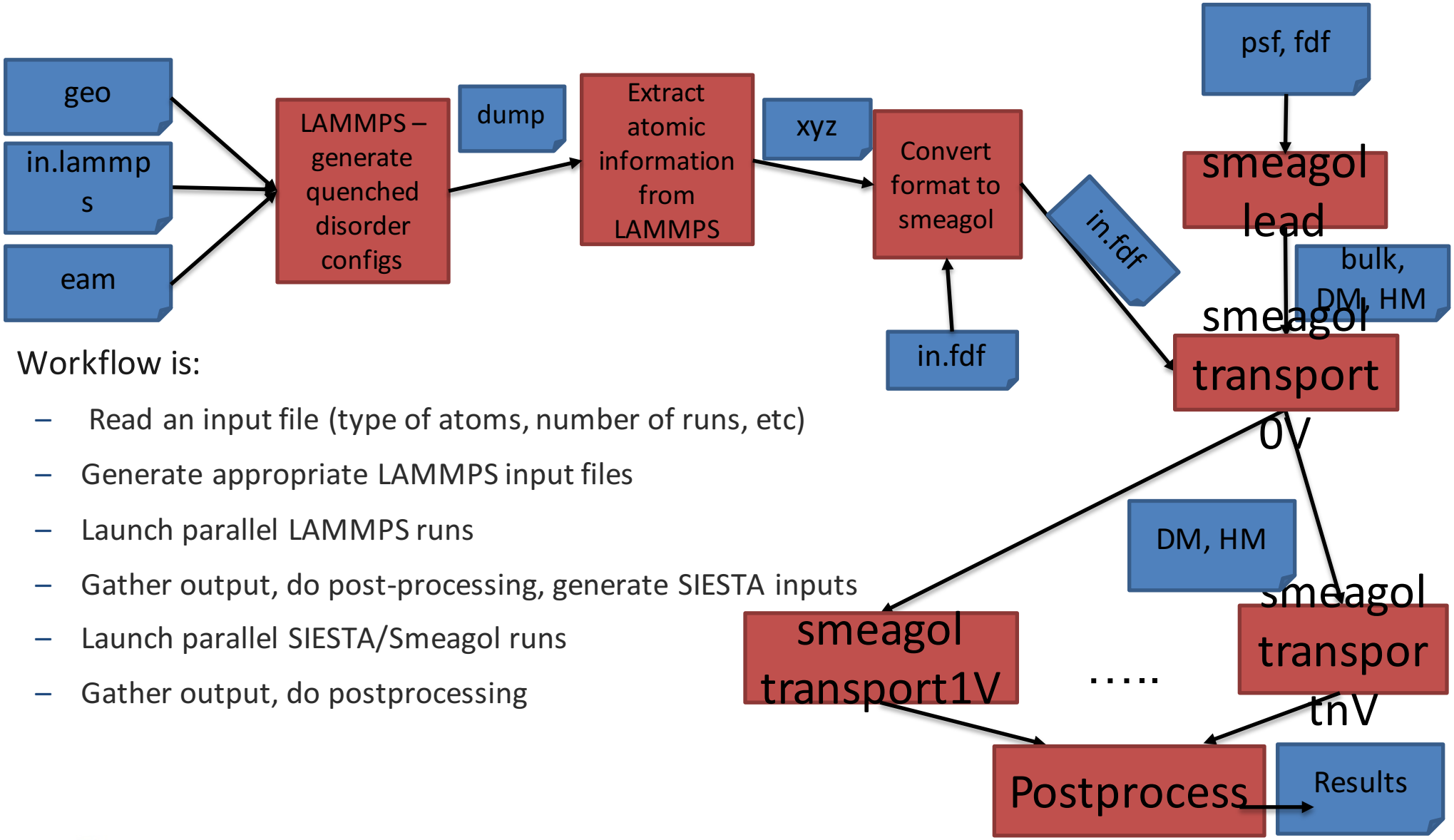
Two modes of workflow on ALCF systems

Swift/K workflows driving Cobalt Script-mode jobs:

Swift/T Workflow:



Multi-scale Materials Science Workflow



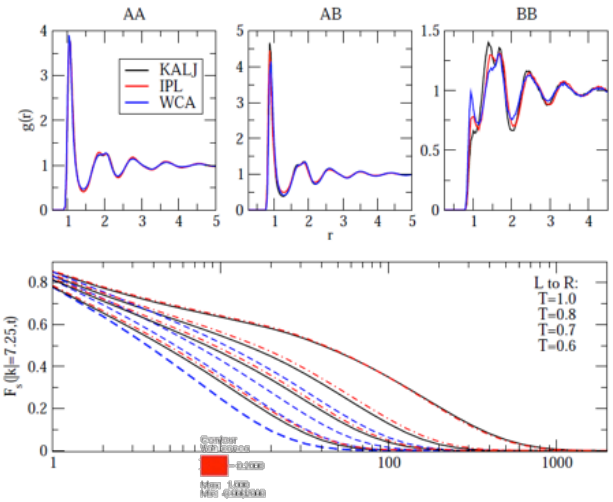
Workflow is:

- Read an input file (type of atoms, number of runs, etc)
- Generate appropriate LAMMPS input files
- Launch parallel LAMMPS runs
- Gather output, do post-processing, generate SIESTA inputs
- Launch parallel SIESTA/Smeagol runs
- Gather output, do postprocessing

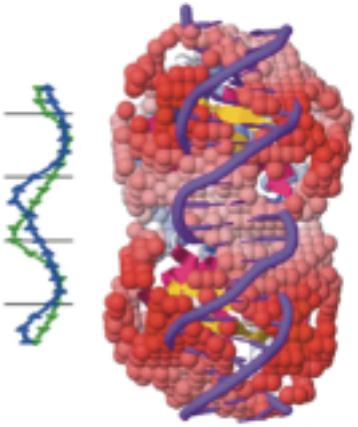
Large-scale applications using Swift

- **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

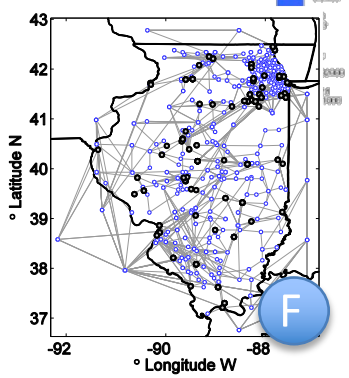
All have published science results obtained using Swift



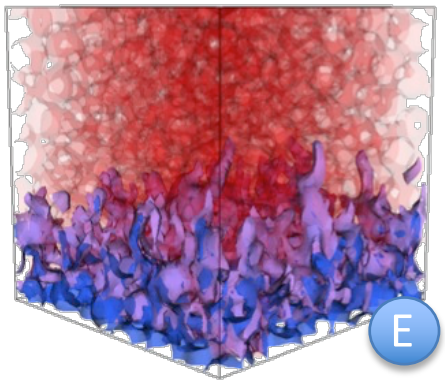
A



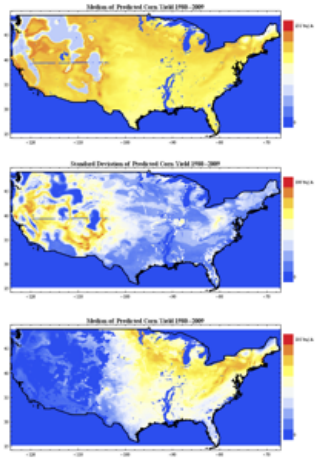
B

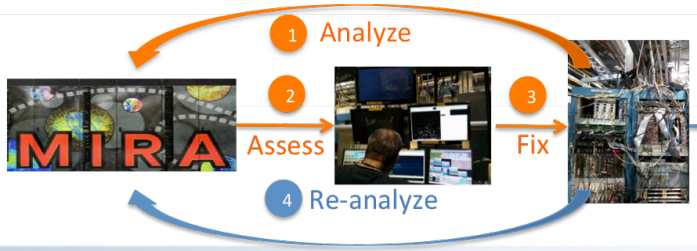


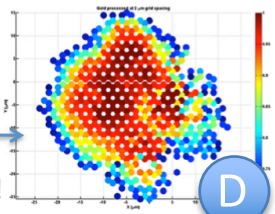
F



E

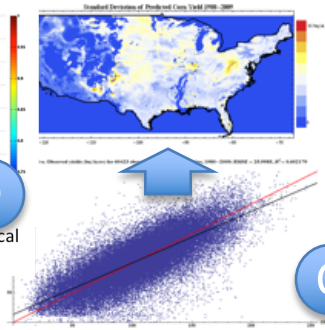






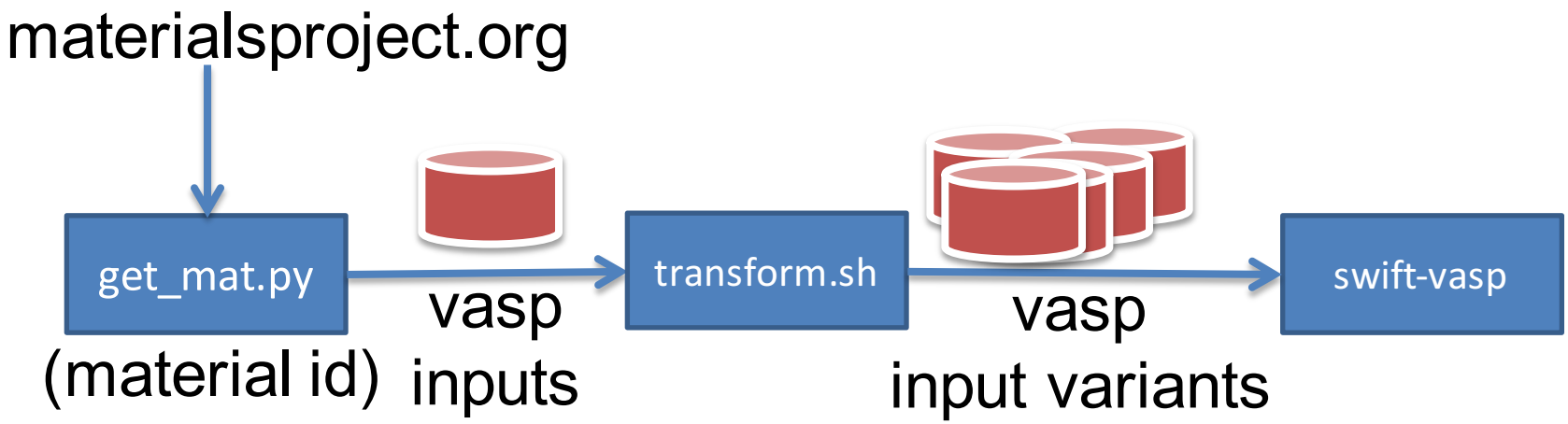
D

Red indicates higher statistical confidence in data



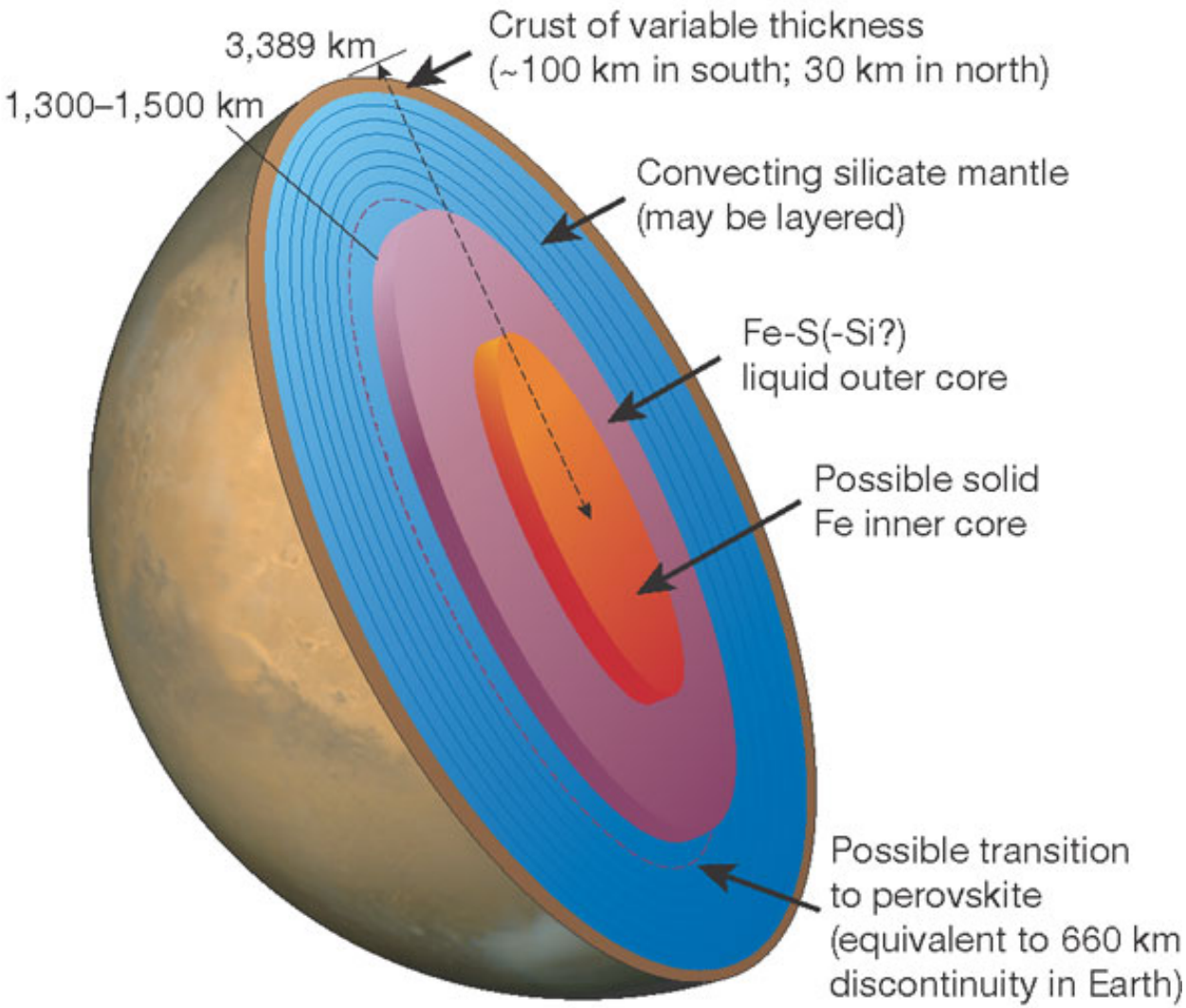
C

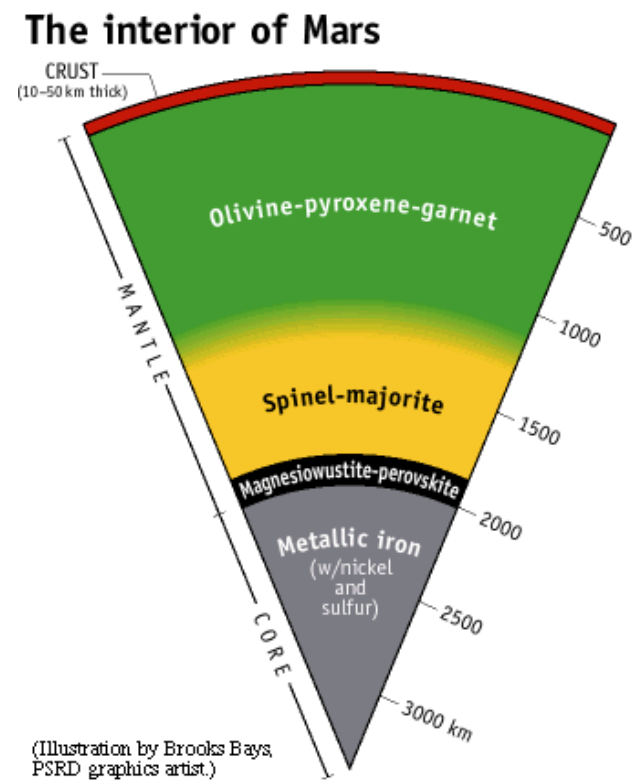
Swift-Material Prototype Workflow



- A python API fetches materials data using id
- A transform script perturbs the data to generate variants
- Swift-Vasp runs VASP under Swift over ALCF resources
- Turnkey solution for material scientists

High Pressure Behavior of Fe3S using VASP (with Mainak Mookherjee, Cornell Geoscience)



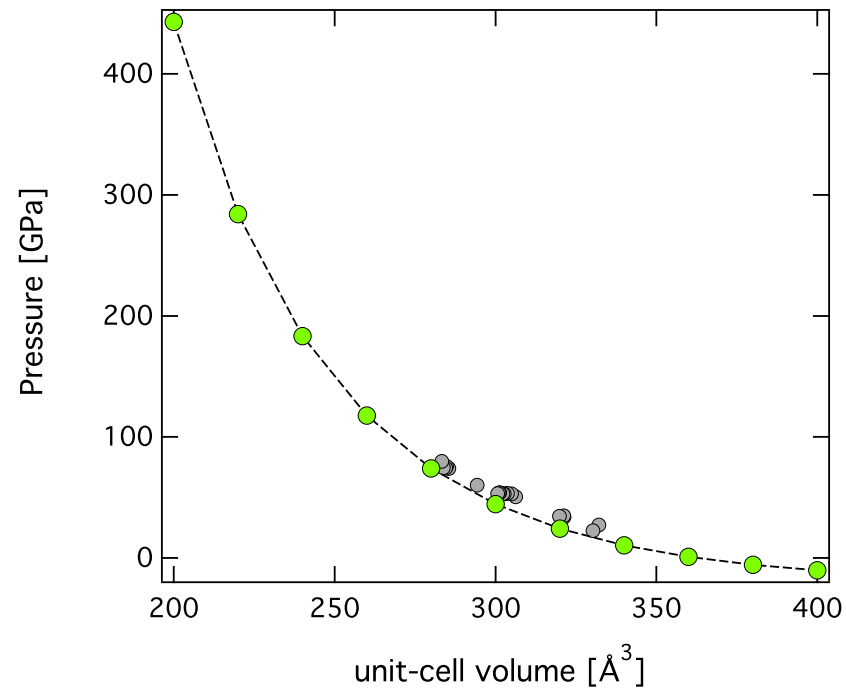


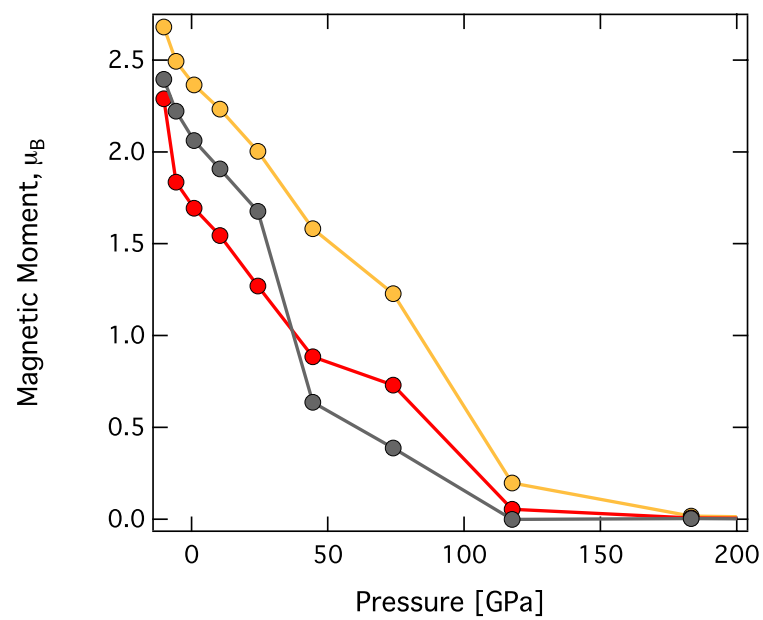
Stevenson, D. J. (2001), Mars' core and magnetism, *Nature*, 412, 214-219.

http://www.psrd.hawaii.edu/WebImg/MarsGuts.gif

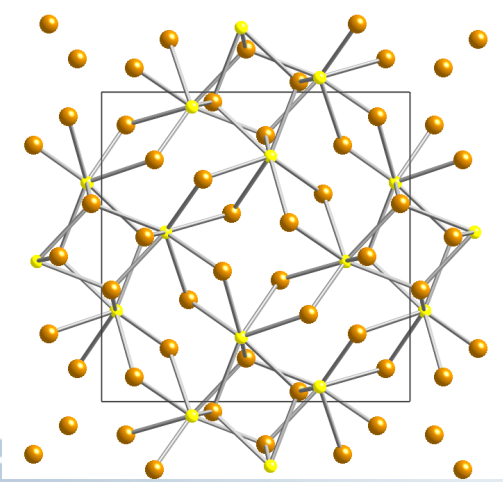
Cornell Geosciences: First Results from BG/Q

- VASP app running on Mira
  - One-rack initial results completed
  - Further runs after analysis
- Abstract in the upcoming American Geophysical Union Conference
- Planning a proposal for Director's Discretionary Allocation
- Scientist successfully enabled to perform VASP runs with Swift



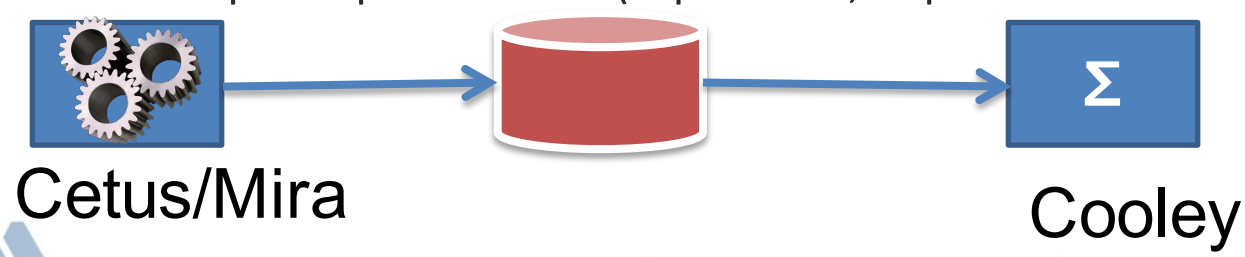


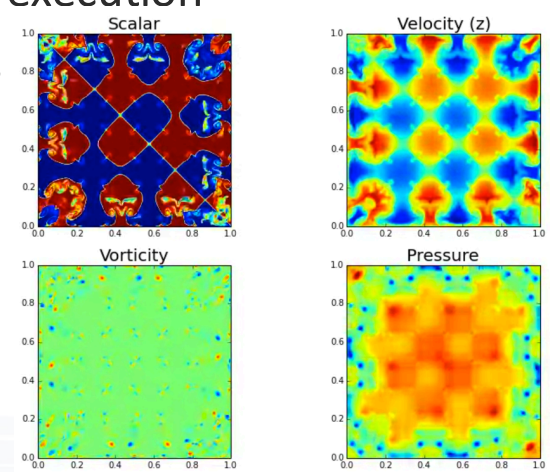
Fe_3S- crystal structure
Fe- brown spheres
S- yellow spheres



Numerical Simulations of the Rayleigh-Taylor instability (in progress)

- Nek5000 code to conduct spectrally converged direct numerical simulations of the Rayleigh-Taylor instability.
  - Max Hutchinson (student of Bob Rosner at UChicago)
  - Applications to Reactors, Fusion, Ocean Modeling
- Application spans Mira and Tukey:
  - Three stages: Preprocess -> Simulate -> Postprocess
  - Preprocess: dynamically build source with input parameters (on Cooley)
  - Simulate: Run NEK5000 framework (on Mira/Cetus)
  - Postprocess: Python analysis scripts (On Cooley)
- Swift solutions:
  - Provide seamless access from one point of workflow execution
  - dynamically sized jobs to optimize BG/Q queue waits
  - capture provenance (repeatable, reproducible runs)





ALCC Submission on Uncertainty Analysis for engine design modeling with HPC

Case	No. of cores	No. simulations	Total core hours
Engine	~ 512	300	~ 38 million
Spray	~ 4000	10	~ 17 million

60 million core hours for 1 year

Port and Optimize on MIRA

Improve scalability and establish best practices on MIRA

Quantify uncertainties in simulations

Identify the uncertainty range and set-up simulations

Advancing engine simulations with sensitivity analysis and HPC resources

Identify important variables influencing particular engine targets

PI: Sibendu Som

Co-PIs:

Marta Garcia (ALCF)

Al Wagner (CSE)

P. K. Senecal (Convergent Science Inc.)

Janardhan Kodavasal (ES)

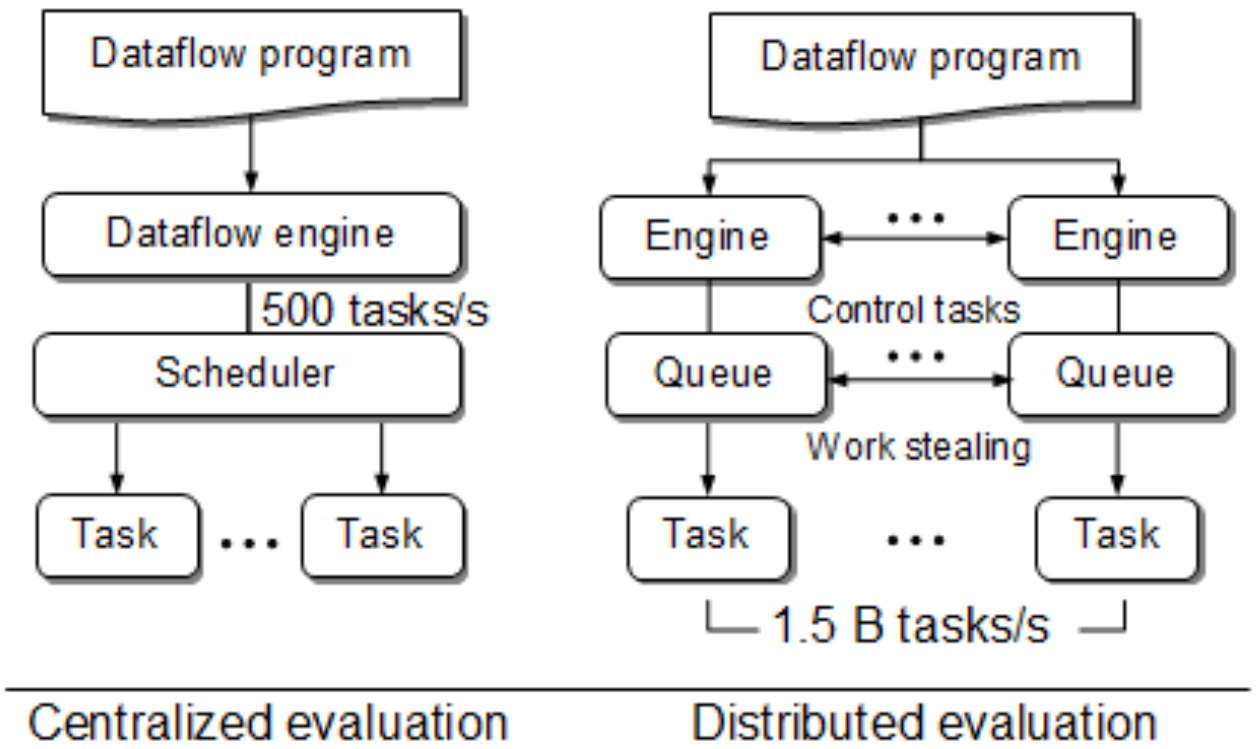
Yuanjiang Pei (ES)

Plan to run ~10K simulations per day

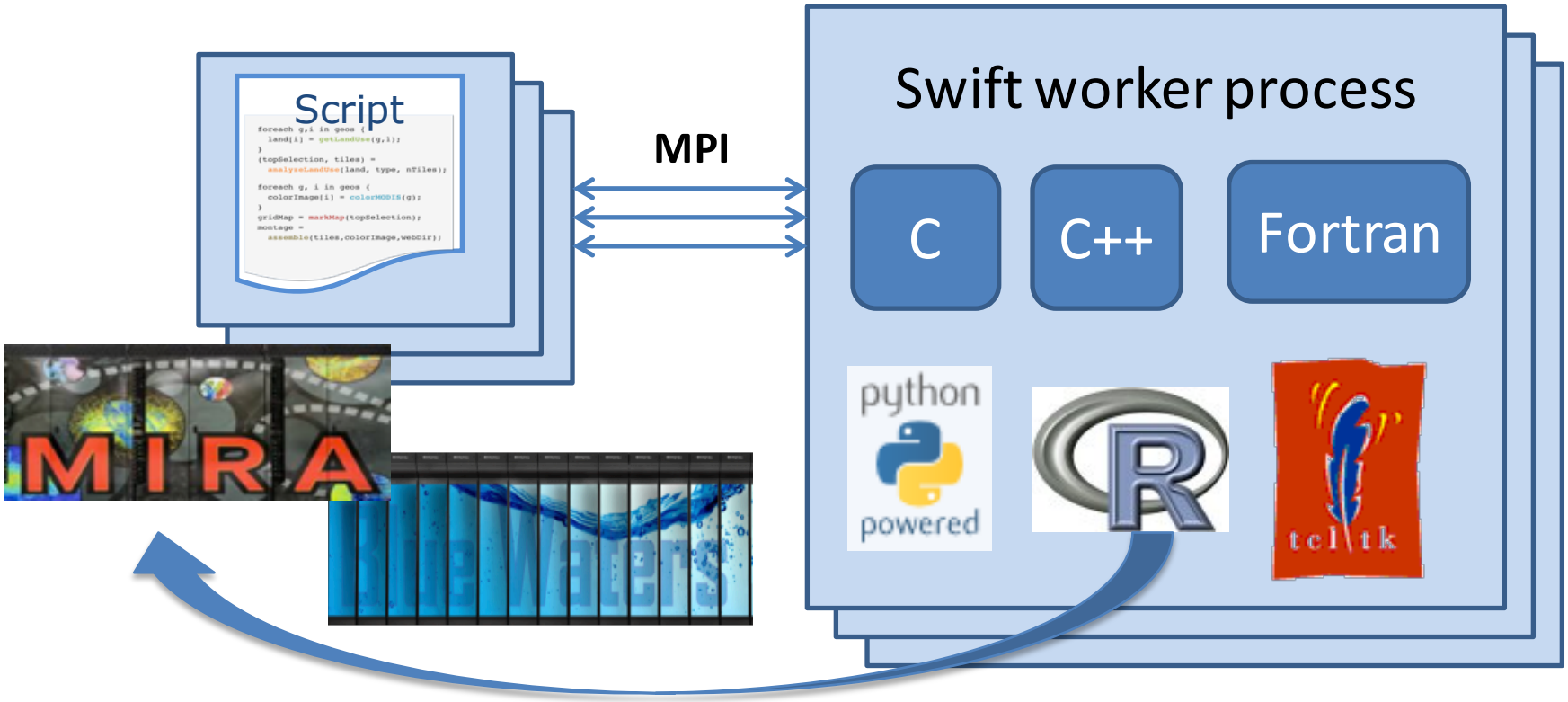
Centralized evaluation can be a bottleneck at extreme scales

Had this (Swift/K):

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



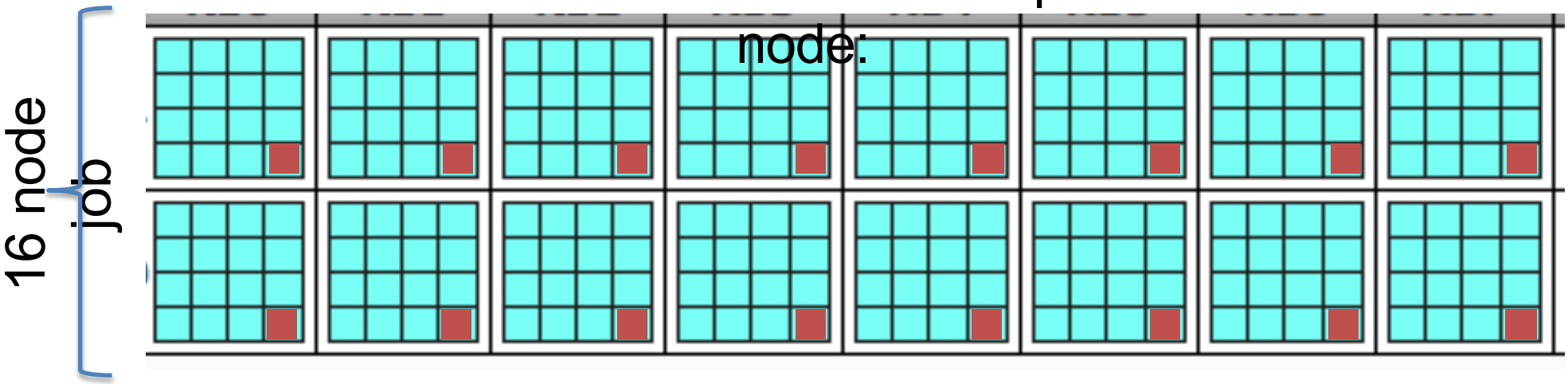
Swift/T: productive extreme-scale scripting



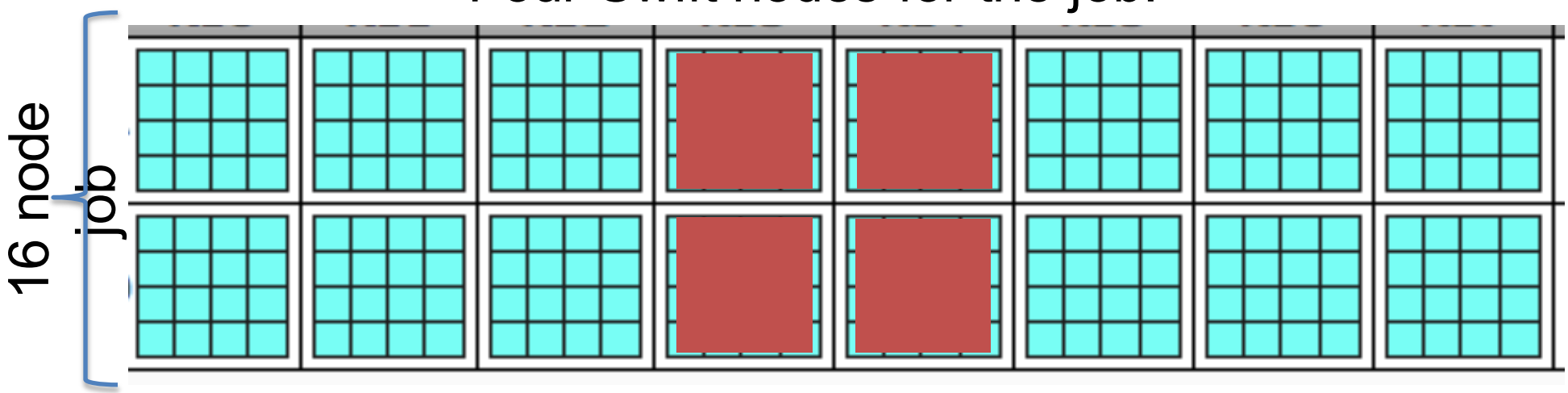
- Script-like programming with "leaf" tasks
  - In-memory function calls in C++, Fortran, Python, R, … passing in-memory objects
  - More expressive than master-worker for "programming in the large"
  - Leaf tasks can be MPI libraries, etc. Can be separate processes if OS permits.
- Distributed, *scalable*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

Flexible placement of server ranks in a Swift/T job

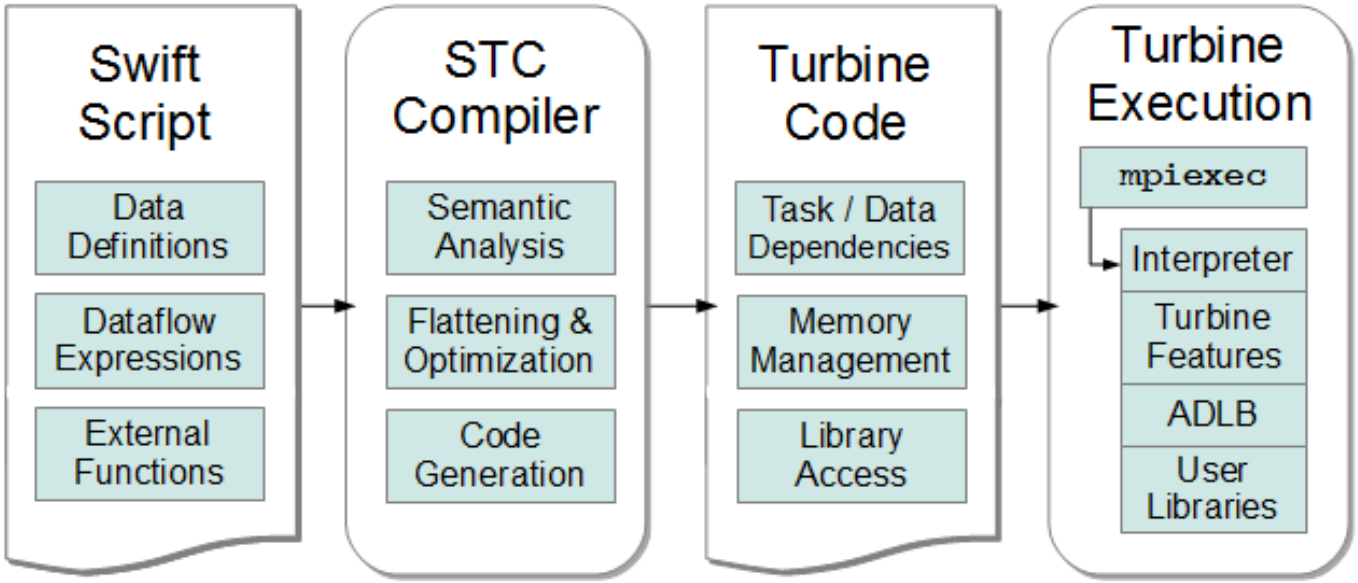
One Swift server core per node:



Four Swift nodes for the job:



Swift/T Compiler and Runtime



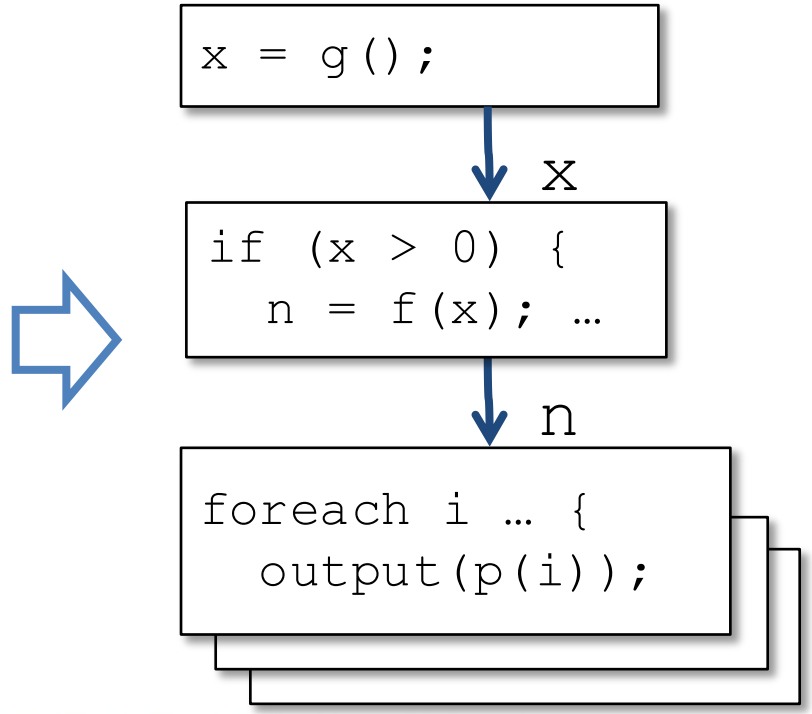
- STC translates high-level Swift expressions into low-level Turbine operations:
  - Create/Store/Retrieve typed data
  - Manage arrays
  - Manage data-dependent tasks

- 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.

Swift code in dataflow

- Dataflow definitions create nodes in the dataflow graph
- Dataflow assignments create edges
- In typical (DAG) workflow languages, this forms a static graph
- In Swift, the graph can grow dynamically – code fragments are evaluated (conditionally) as a result of dataflow
- In its early implementation, these fragments were just tasks

```
x = g();
if (x > 0) {
  n = f(x);
  foreach i in [0:n-1] {
    output(p(i));
  }}
```



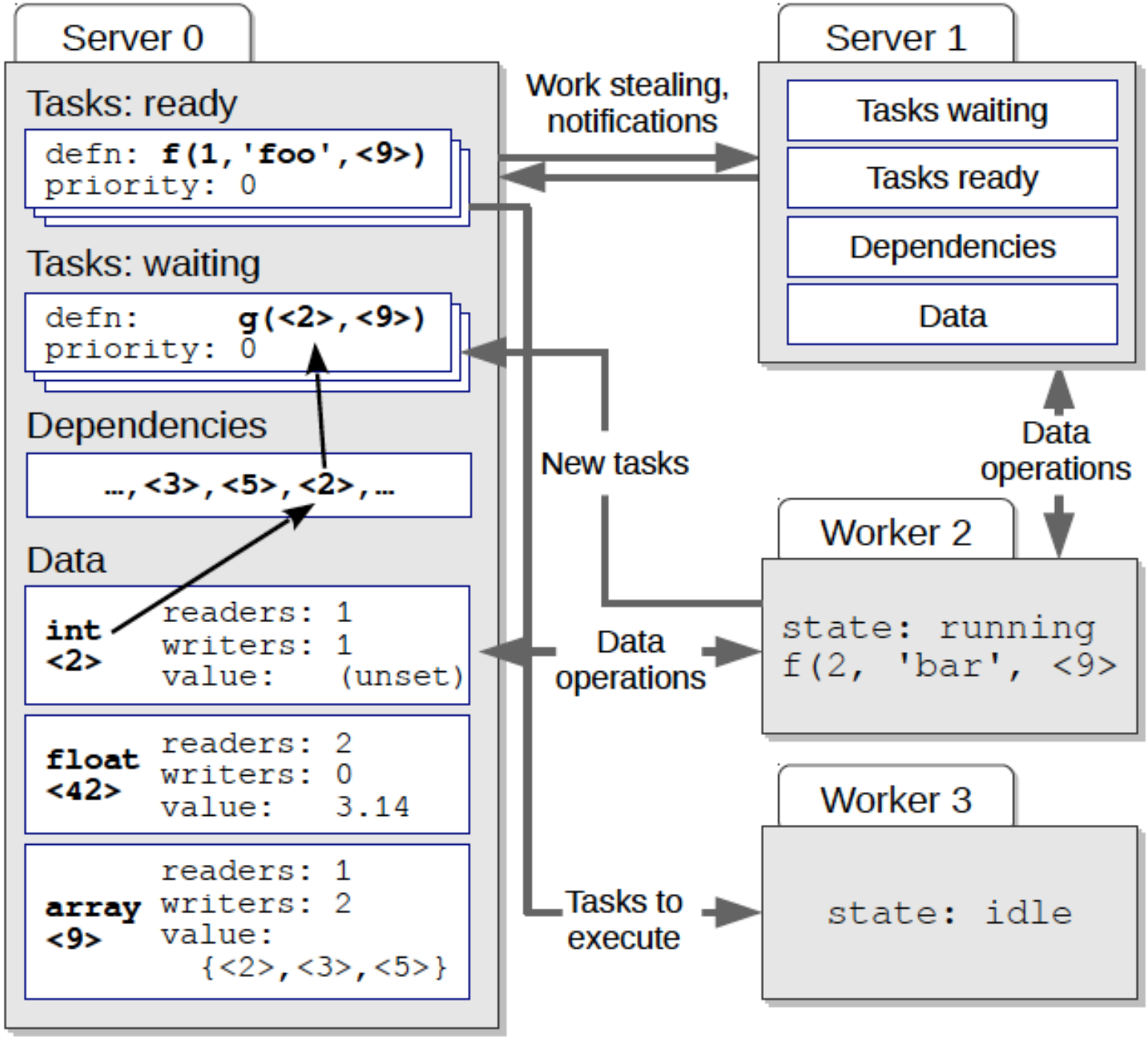
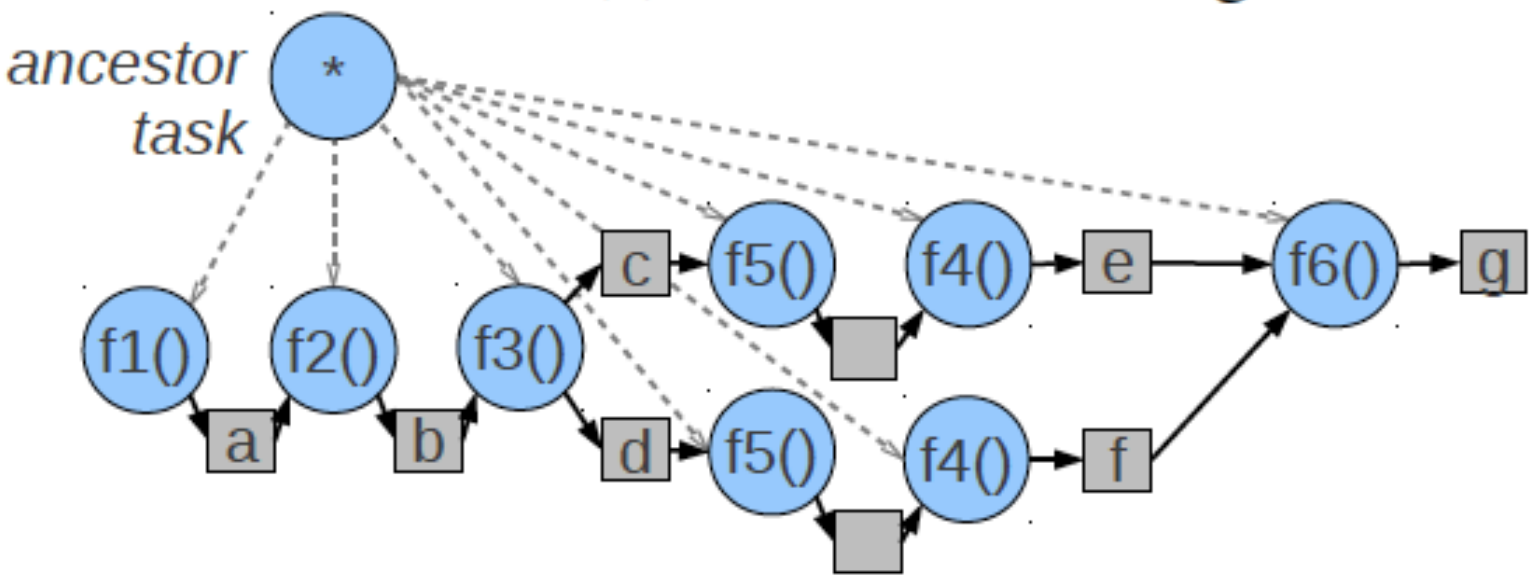


Fig. 4: Runtime architecture showing distributed worker processes coordinating through task and data operations.

Swift/T optimization challenge: distributed vars

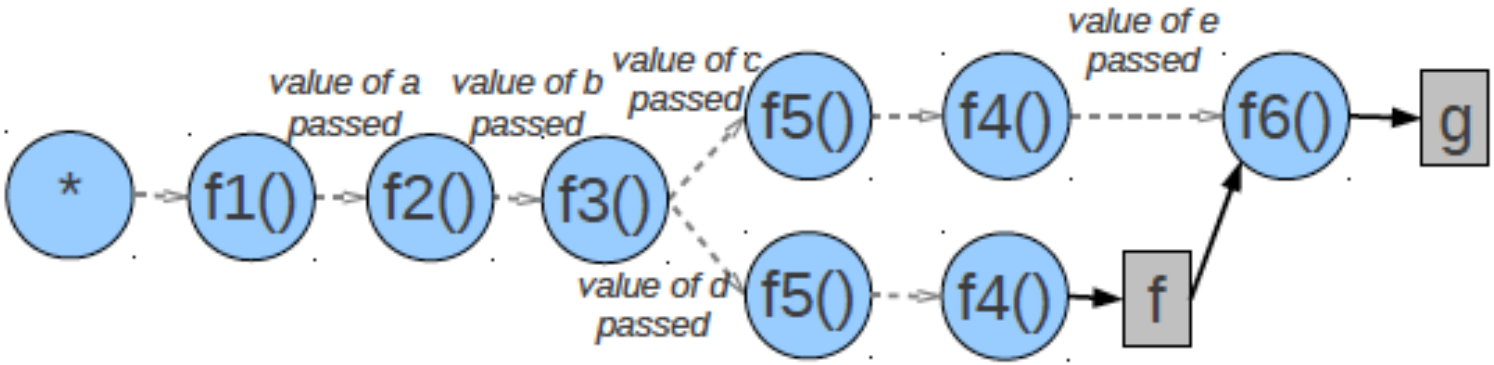
```
1 | a = f1();            b = f2(a);
2 | c, d = f3(a, b);     e = f4(f5(c);
3 | f = f4(f5(d);        g = f6(e, f);
```

(a) Swift/T code fragment

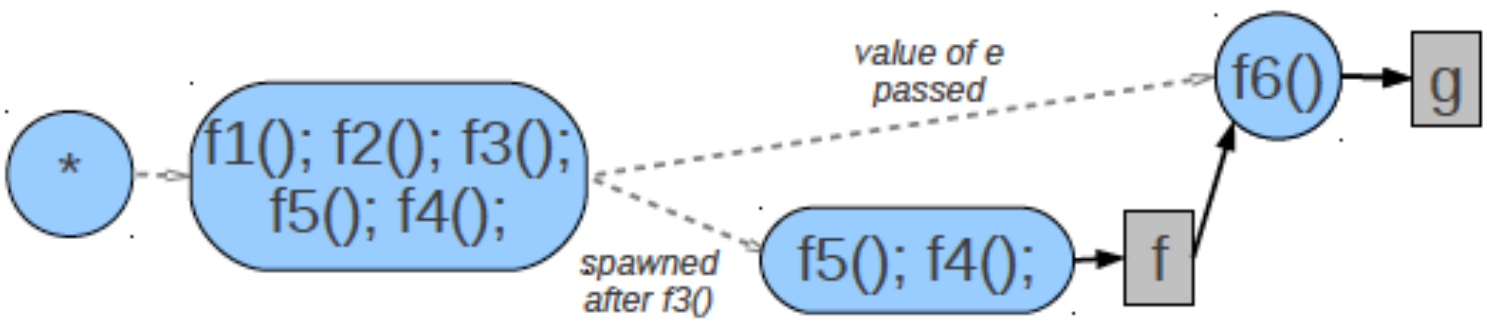


(b) Unoptimized version, passing data as shared data and perform synchronization

Swift/T optimizations improve data locality

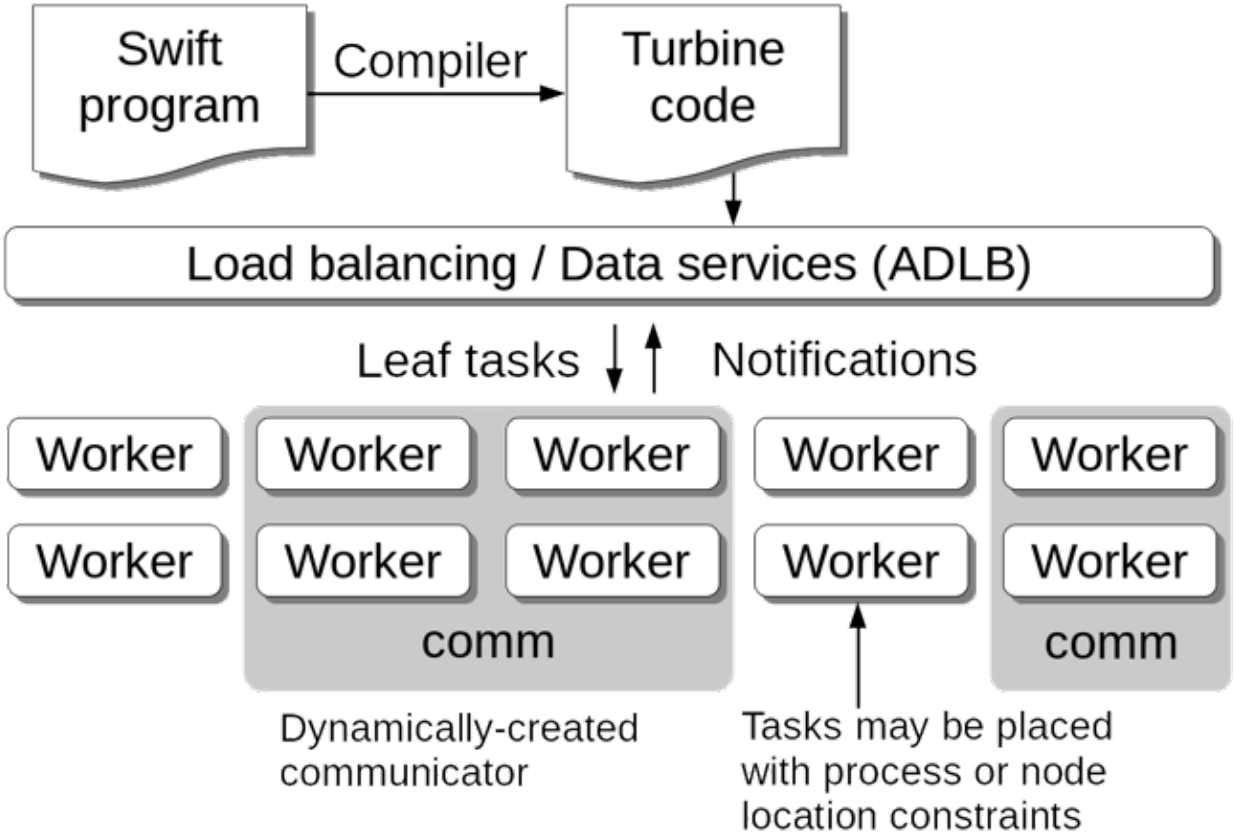


(c) After wait pushdown and elimination of shared data in favor of parent-to-child data passing



(d) After pipeline fusion merges tasks

Parallel tasks in Swift/T

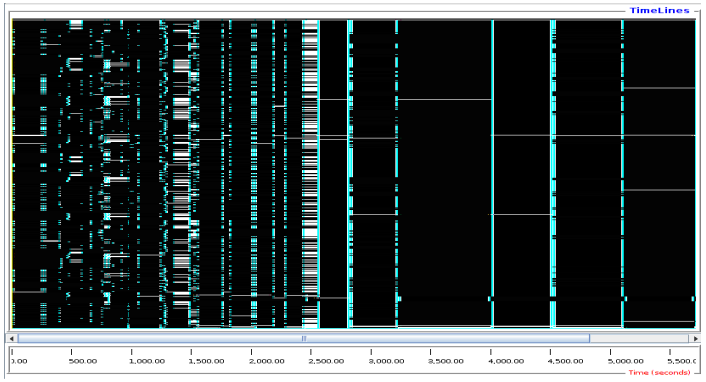


- Swift expression: `z = @par=32 f(x,y);`
- ADLB server finds 8 available workers
  - Workers receive ranks from ADLB server
  - Performs `comm = MPI_Comm_create_group()`
- Workers perform `f(x,y)` communicating on `comm`

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

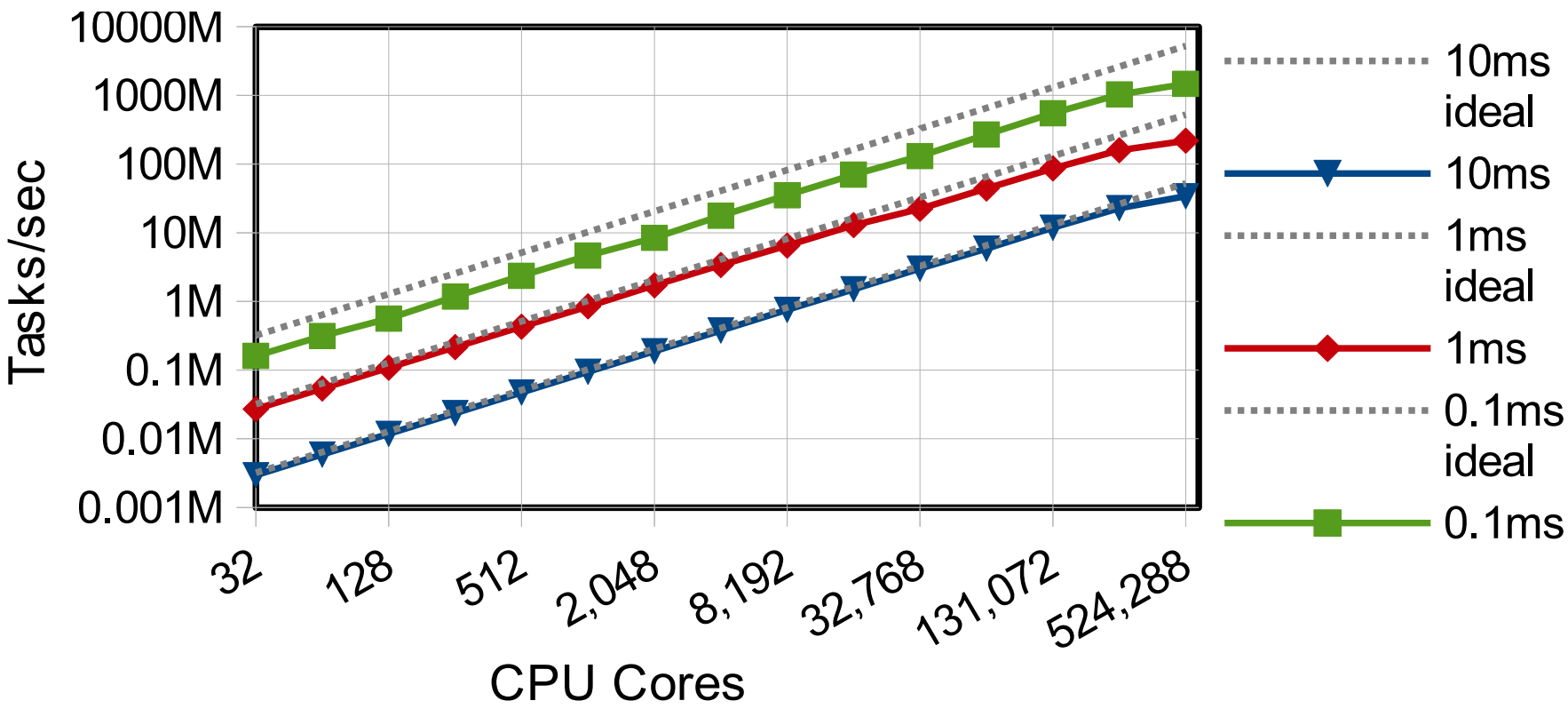
Swift/T-specific features

- 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

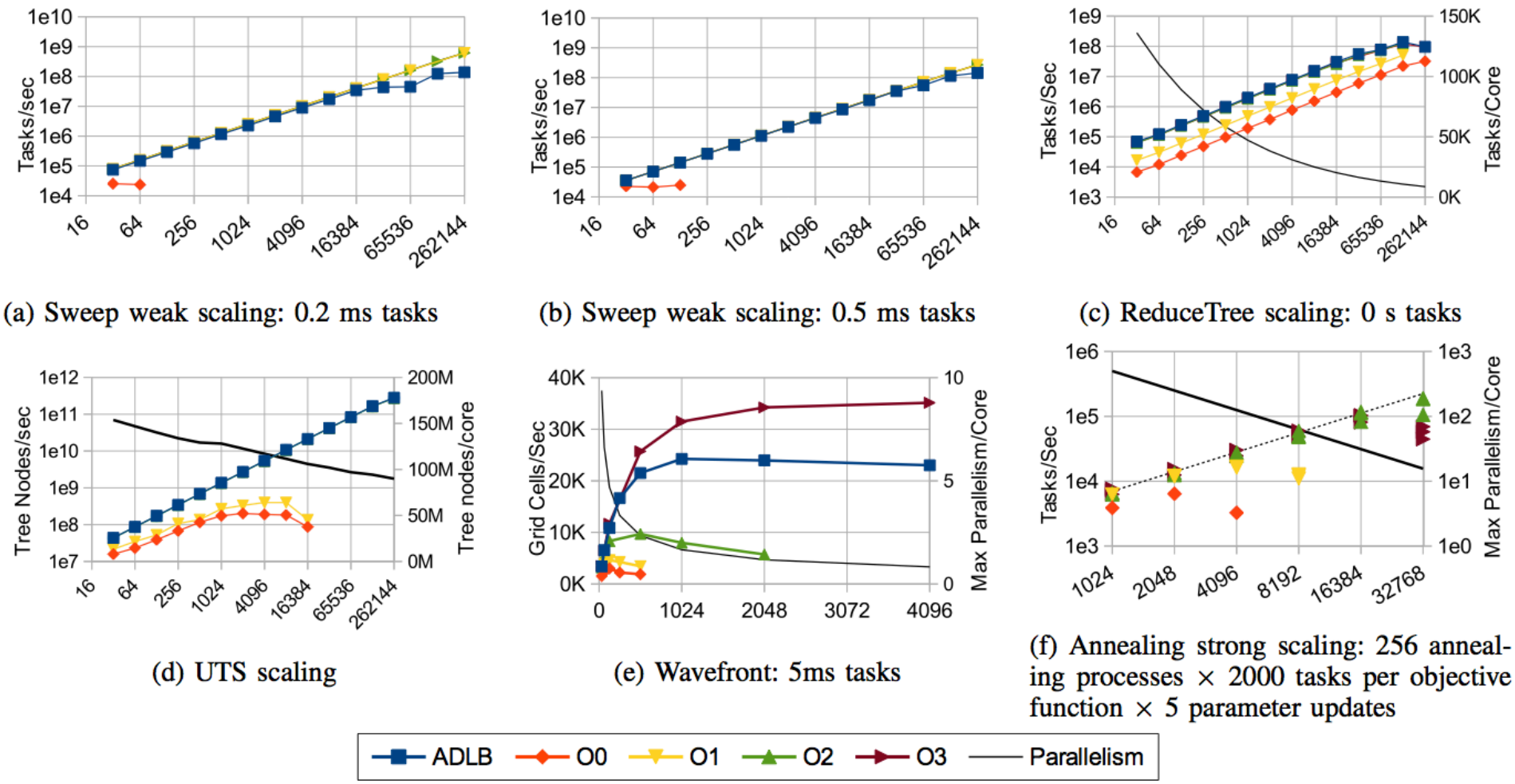
Wozniak et al. Language features for scalable distributed-memory dataflow computing. Proc. Dataflow Execution Models at PACT, 2014.

Swift/T: scaling of trivial foreach { } loop

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



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.

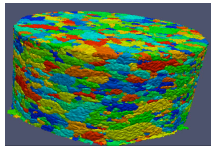
Boosting Light Source Productivity with *Swift* ALCF Data Analysis

DD 2014

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



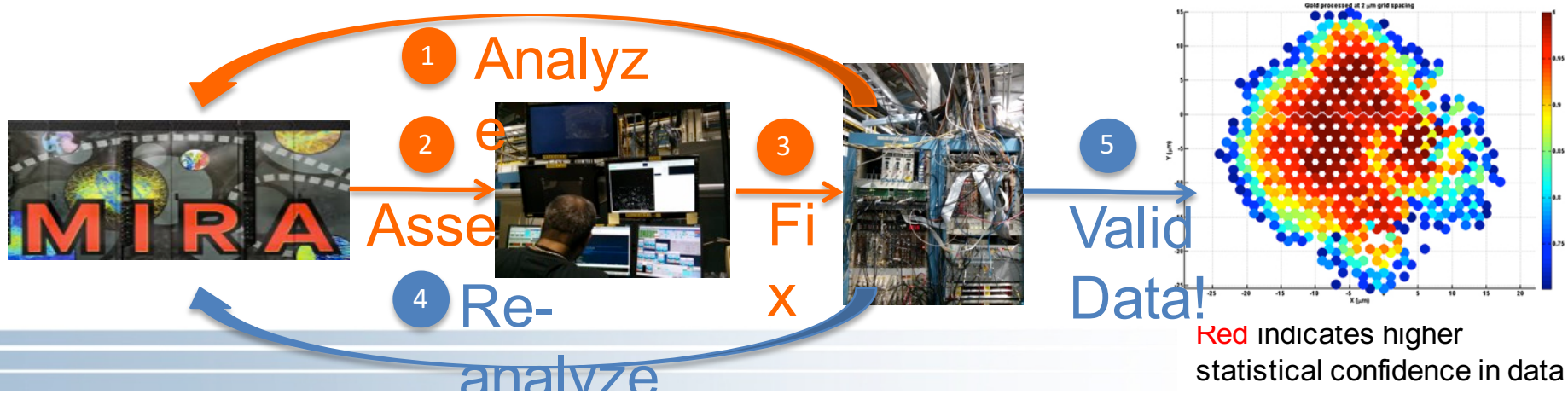
- APS Sector 1 scientists use Mira to process data from live HEDM experiments, providing real-time feedback to correct or improve in-progress experiments
- Scientists working with *Discovery Engines* LDRD developed new *Swift* analysis workflows to process APS data from Sectors 1, 6, and 11

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

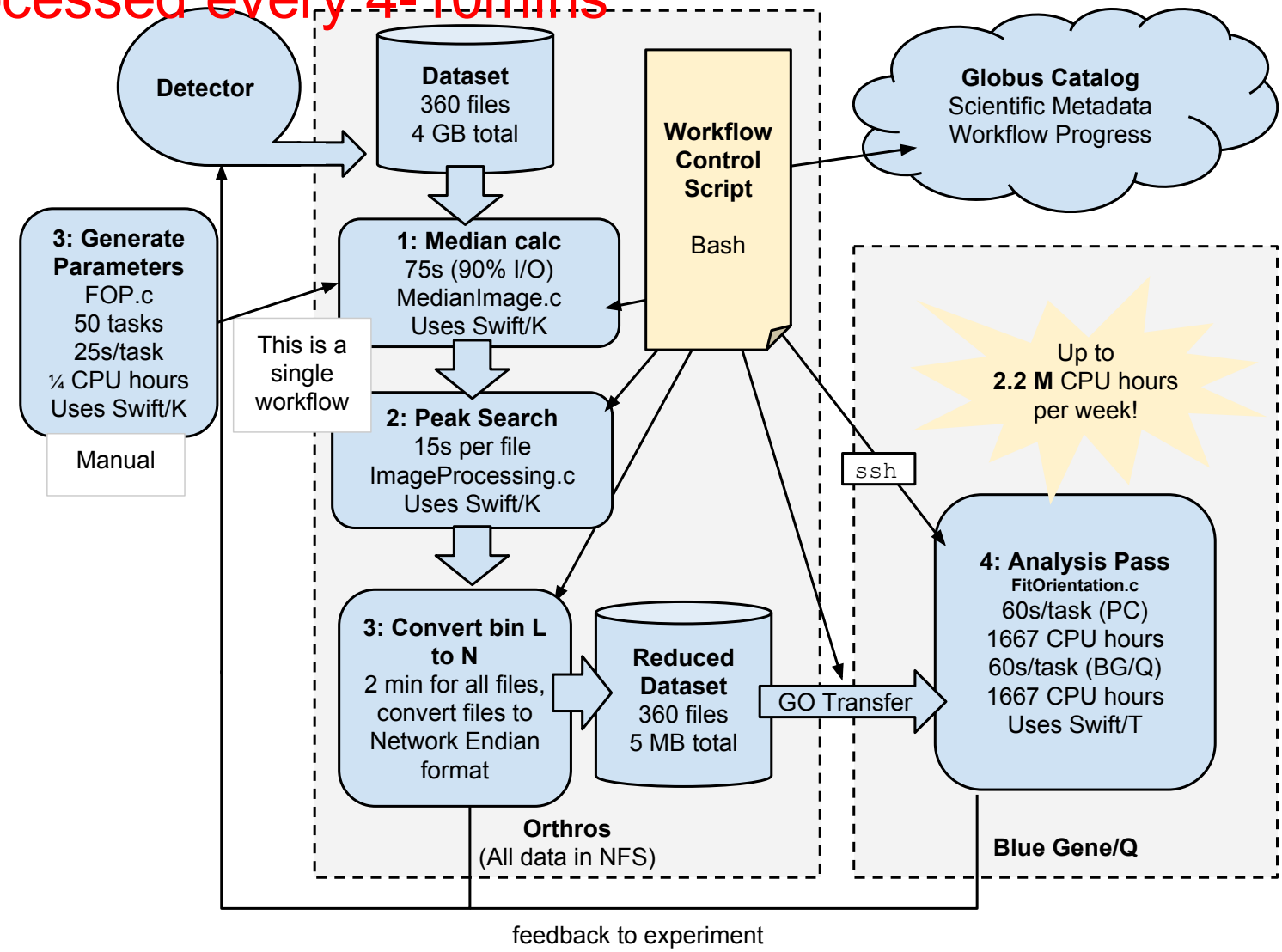
ALCF Contributions

- 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)



Near Field-HEDM Using Mira via Swift

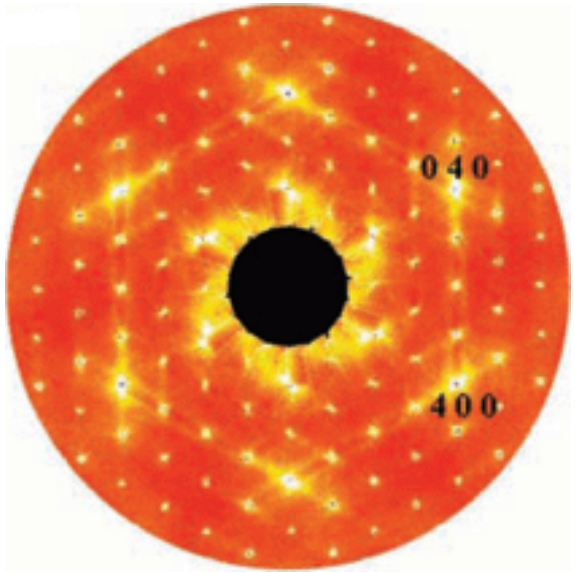
Single integrated cross-system script – 4GB processed every 4-10mins



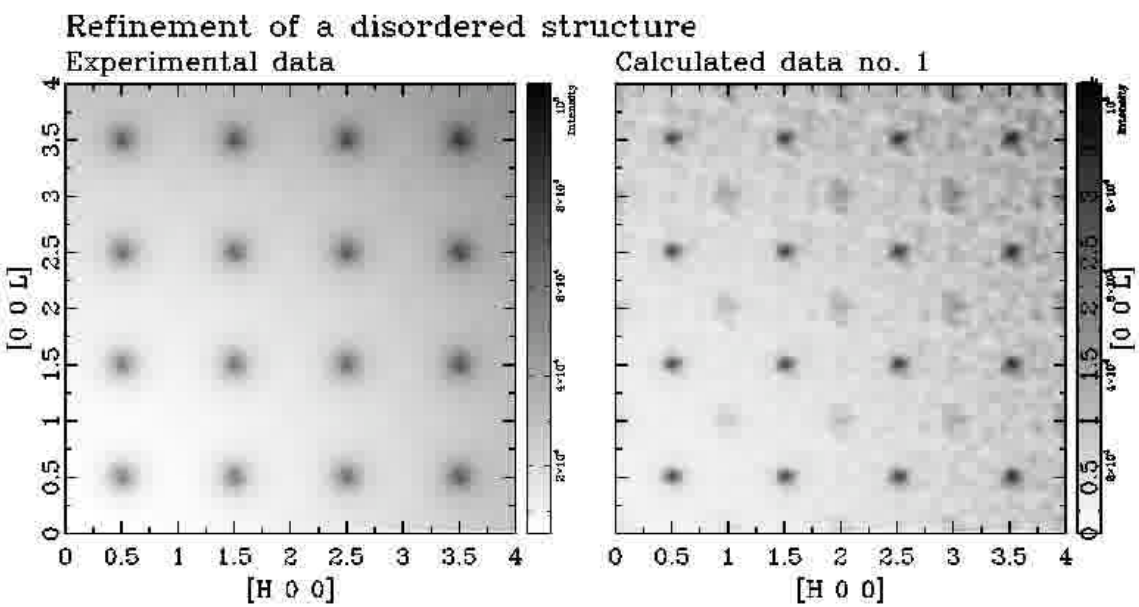
Diffuse scattering and crystal analysis

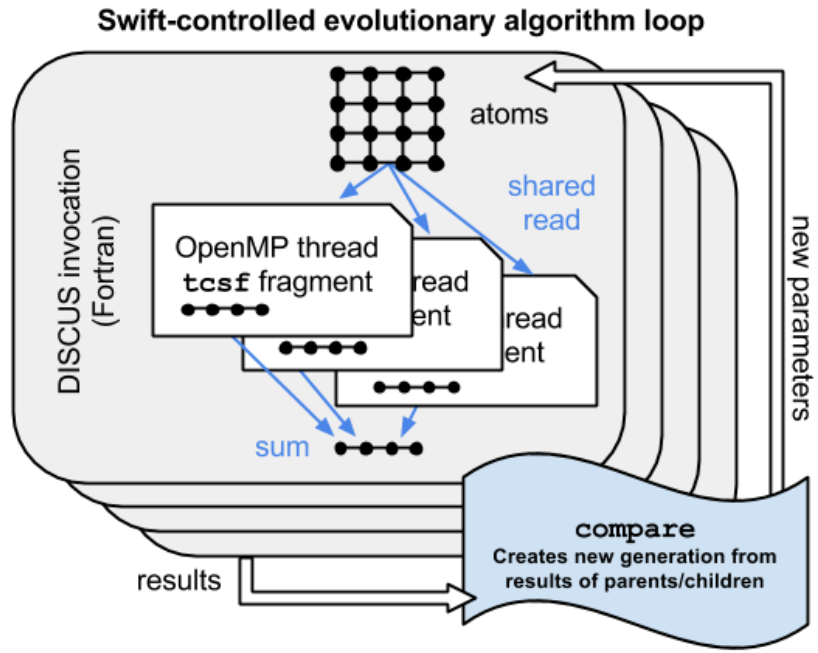
- DISCUS is a general program to generate disordered atomic structures and compute the corresponding experimental data such as single crystal diffuse scattering (http://discus.sourceforge.net)
- Given experimental data, can we fit a modeled crystal to the measurement?

- Experimental image: (Billinge, 2006)



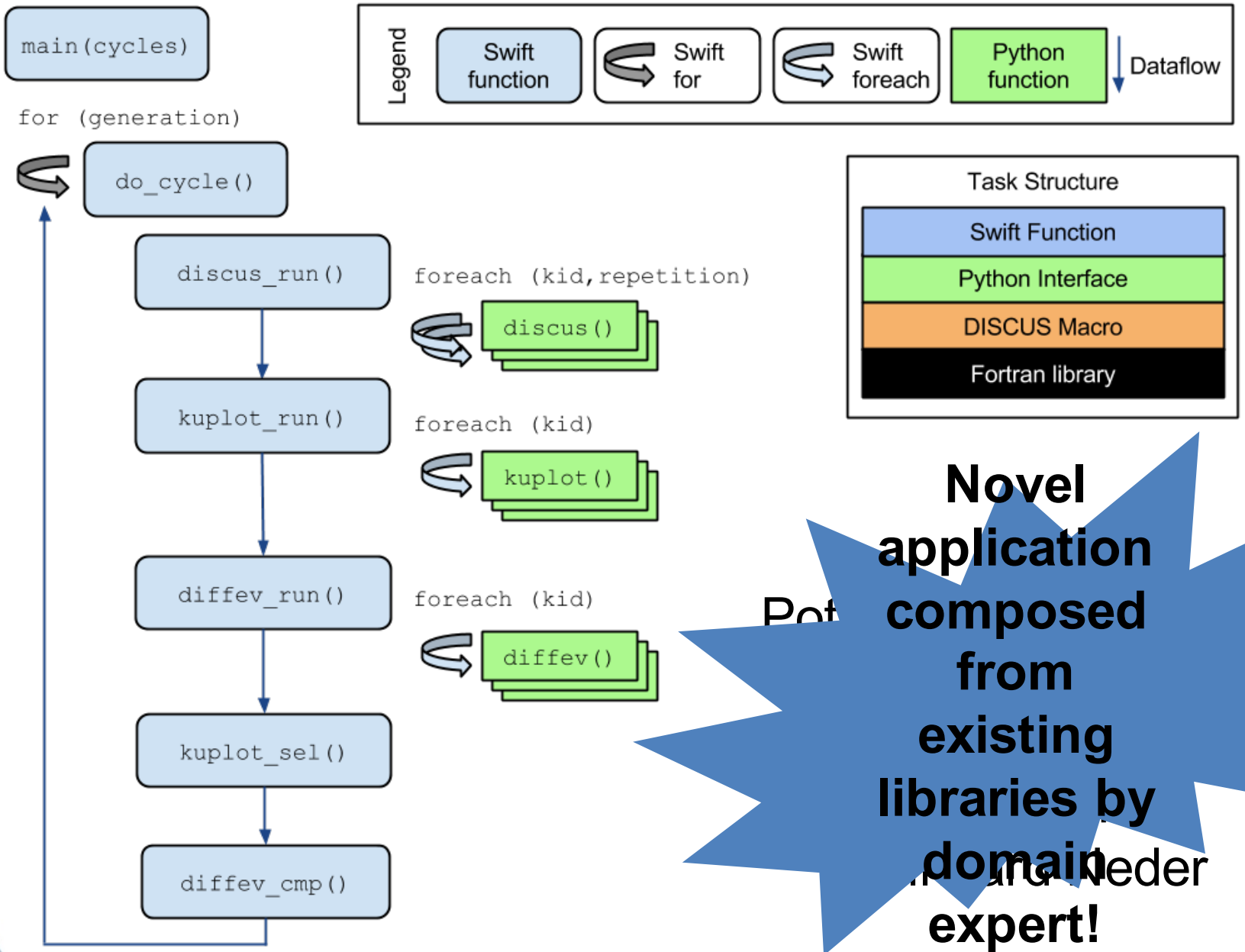
DIFFEV: Scaling crystal diffraction simulation





- Determines crystal configuration that produced given scattering image through simulation and evolutionary algorithm
- Swift/T calls DISCUS via Python interfaces

DIFFEV: Genetic algorithm via dataflow



Novel application composed from existing libraries by domain expert!

Swift integration into NAMD and VMD

www.ks.uiuc.edu/Research/swift

www.ks.uiuc.edu/Research/swift/

Apps | git-svn | RTM | BigBoard | Maps | Wikipedia | Swift, Inc | ProxyIt! | Other Bookmarks

NIH CENTER FOR MACROMOLECULAR MODELING & BIOINFORMATICS | UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Type Keywords SEARCH

THEORETICAL *and* COMPUTATIONAL BIOPHYSICS GROUP

Home | Research | Publications | Software | Instruction | News | Galleries | Facilities | About Us

- Home
- Overview
- Publications
- Research
  - Membrane Biology
  - Mechanobiology
  - Nanoengineering
  - Bioenergetics
  - SMD/IMD
  - Quantum Biology
  - Neurobiology
  - X-ray/Cryo-EM Modeling with MDFF
  - Driving Biomedical Projects
  - Collaborations
  - Other Topics
- Software
- Outreach

Integrating NAMD and VMD with Swift/T

NAMD and VMD have recently been successfully coupled to the **Swift/T high performance parallel scripting language** developed as part of the **ExM project**, a collaboration led by Argonne National Laboratory with University of Chicago and University of British Columbia, as a part of the **Department of Energy ASCR X-Stack** program. Swift/T is now supported as part of the **Swift project** under the **NSF SI2 program**. Standard NAMD 2.10 and VMD 1.9.2 binaries can be launched across the nodes of a parallel computer and efficiently execute Swift/T dataflow programs with functions implemented in the embedded Tcl scripting language. The NAMD and VMD user communities are already familiar with Tcl, and Tcl allows access to the two programs' complete functionality. The NAMD integration with Swift/T has been used to demonstrate n:m multiplexing of n replicas across a smaller arbitrary number m of NAMD processes, a very complex capability to implement with normal NAMD scripting that can be expressed naturally in under 100 lines of Swift/T code.

All example files: **directory**, **tar archive**

VMD Swift/T Hello World

VMD and Turbine must be built with compatible Tcl libraries so that VMD can dynamically load libtclturbine.so.

- Example command: `mpiexec -n 8 vmdwrapper -e vmdswift.tcl`
- Wrapper script to run standard VMD under MPI: **vmdwrapper**
- Tcl package and Swift startup for VMD: **vmdswift.tcl**
- Swift program source code: **hello.swift**
- Swift compiler Tcl output: **hello.tcl**

NAMD Swift/T Replica Exchange

NAMD and Turbine must be built with compatible Tcl libraries so that NAMD can dynamically load libtclturbine.so.

- Example command: `mpiexec -n 8 namdwrapper namdswift.tcl apoa1.namd --run 0 --source $cwd/replica.tcl < /dev/null &`
- Wrapper script to run multicore NAMD under MPI: **namdwrapper**
- Tcl package and Swift startup for NAMD: **namdswift.tcl**
- Swift program source code: **replica.swift**
- Swift compiler Tcl output: **replica.tcl**

NAMD Swift/T MPI Tight Binding

Charm++ and NAMD must be built from source code. An MPI-based Charm++ must be used. Apply the patches below to Charm++ and NAMD, respectively, to allow Turbine to access the Charm++ inter-partition communicator. Charm++, NAMD, and Turbine must be built with compatible Tcl and MPI libraries so that NAMD can dynamically load libtclturbine.so.

- Example command: `mpiexec -n 32 Linux-x86_64-g++.mpi/namd2 namdswift.tcl apoa1.namd --run 0 --source $cwd/replica.tcl +replicas 8 +stdout /var/tmp/stdout.%d.log < /dev/null &`
- Patch for Charm++ source code: **charmswift.patch**
- Patch for NAMD source code: **namdswift.patch**

Funded by a grant from the National Institute of General Medical Sciences of the National Institutes of Health

Center for the Physics of Living Cells

Beckman Institute for Advanced Science and Technology // National Institutes of Health // National Science Foundation // Physics, Computer Science, and Biophysics at University of Illinois at Urbana-Champaign

ILLINOIS UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

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)
- 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

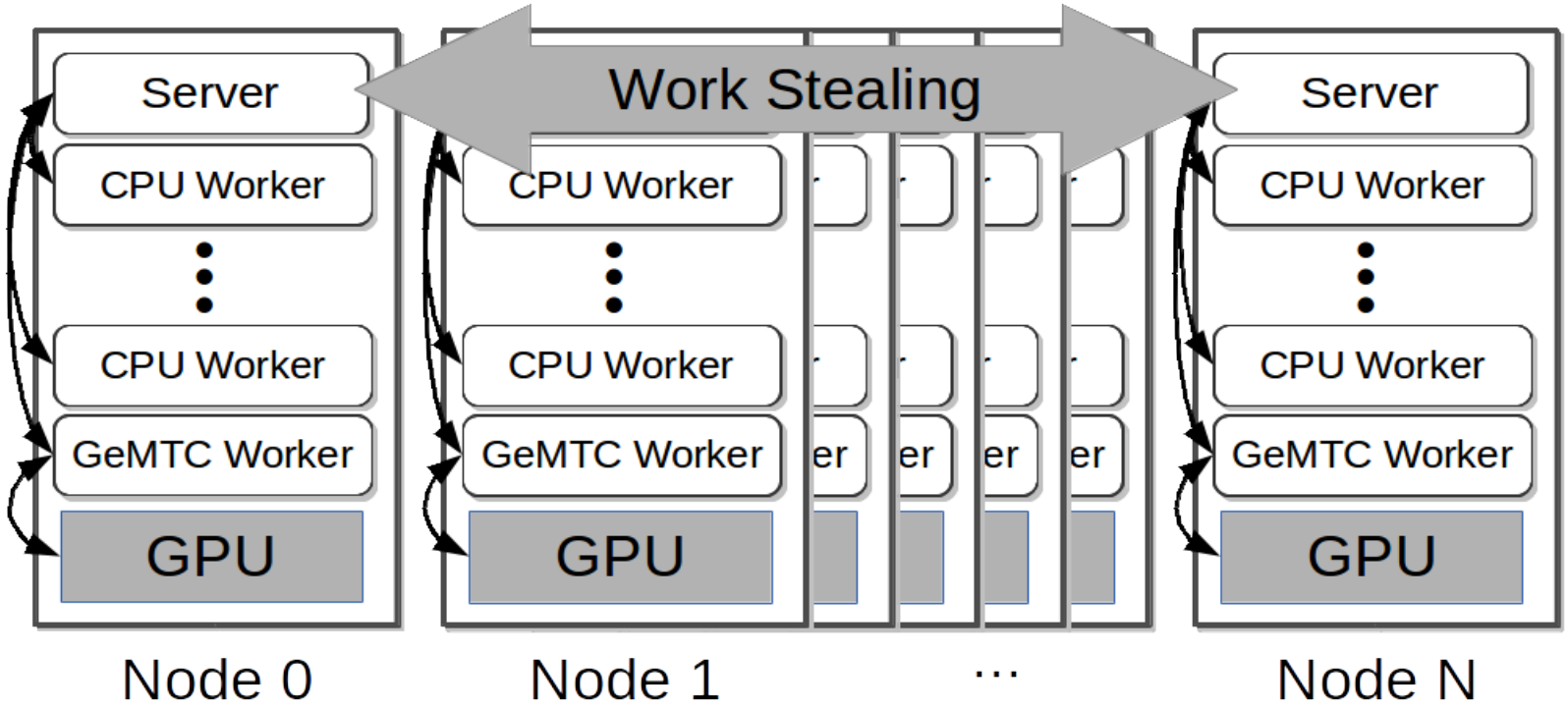
5) Increase concurrency to warp level

Approach:

Design & implement GeMTC middleware:

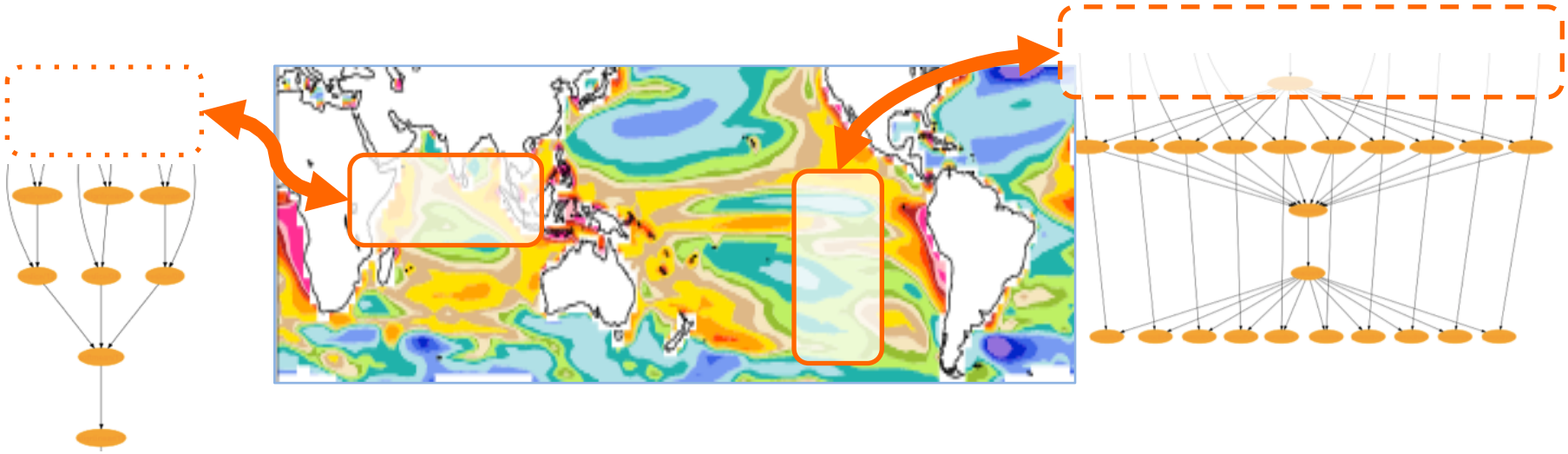
1) Manages GPU 2) Spread host/device

3) Workflow system integration (Swift/T)



Further research directions

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



Deeply in-situ analytics of a climate simulation

Swift gratefully acknowledges support from:

NSF

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

THE UNIVERSITY OF CHICAGO

ALCF

http://swift-lang.org