Workflow for extreme-scale systems

Presenter: Michael Wilde wilde@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory
University of Chicago/Argonne Computation Institute

Outline

- Overview – context of workflow for science and engineering
- Workflow environments
- Expressing workflows – tools and programming models
- Workflow issues for extreme scale
- IO performance envelopes for workflow
- Expressing workflows in Swift
- Hands-on workflow examples and exercises using Swift
  - Language basics
  - Running on a cluster
  - Running on a cloud
  - Running on a supercomputer
  - Multi-machine workflows
  - Scaling up with Swift/Turbine: in memory and inter-language
  - More advanced (self-paced) workflows exercises and experiments

Definitions

- Workflow: the execution of a set of application programs
  - Often for a diverse set of application programs
  - Often with logical and physical dependencies
    - Logical: data dependencies
    - Physical: resource dependencies (space, processor, solution priorities)
  - Scripting is one way to implement workflows (Ad-hoc, Parallel libraries, Swift)
  - Generation of engine-specific input is another (DAGMan, Pegasus, Galaxy, Kepler)
- Scripting: higher-level dynamic programming
  - J. Ousterhout: "Scripting: Higher level programming for the 21st century"
- High throughput computing (HTC)
- Many-task computing (MTC)
- Dataflow
- Data parallel vs. task parallel
  - Workflow is almost always task-parallel at its outer levels
  - SPMD: typified by MPI
  - MPMD: multiple programs, multiple data – more typical of workflow

Definition of MTC Applications

- Many-task Computing applications assemble existing parallel or sequential programs
- Those programs read and write data to a filesystem
- Applications often have multiple stages
- Task dependencies between stages are in the form of file production and consumption
- Can have very high rates (eg hundreds per second) of very short tasks (minutes seconds)

Slide courtesy of Zhao Zhang

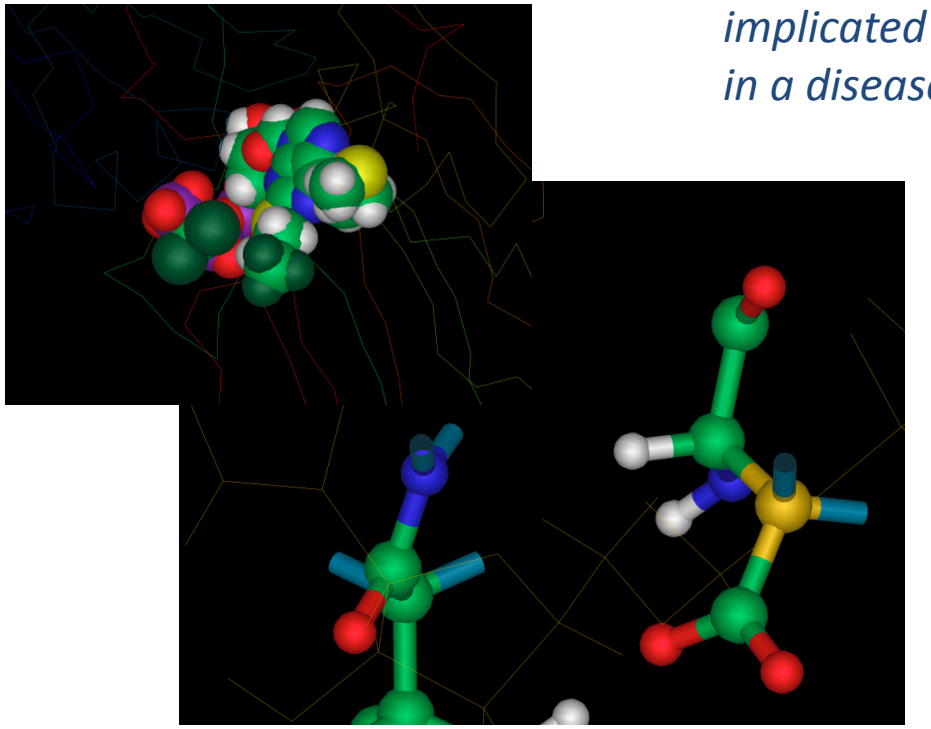
When do you need workflow?

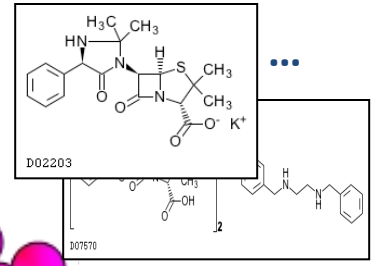
Typical application: protein-ligand docking for drug screening

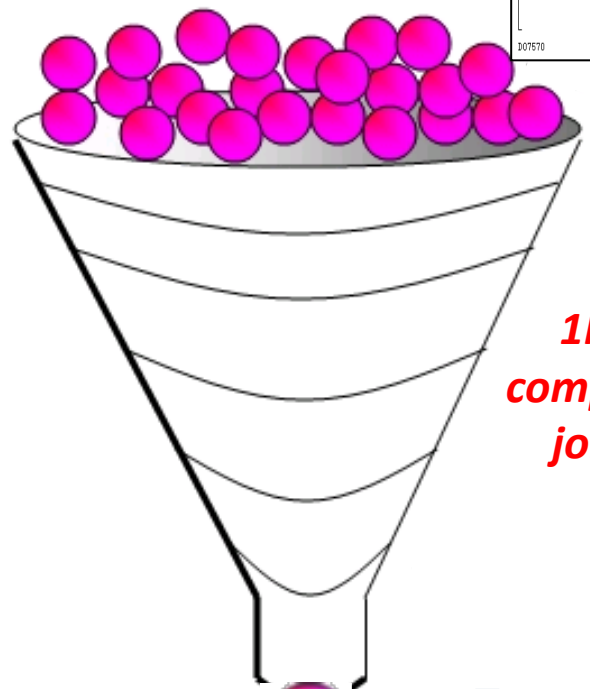
O(10) proteins implicated in a disease

X

O(100K) drug candidates

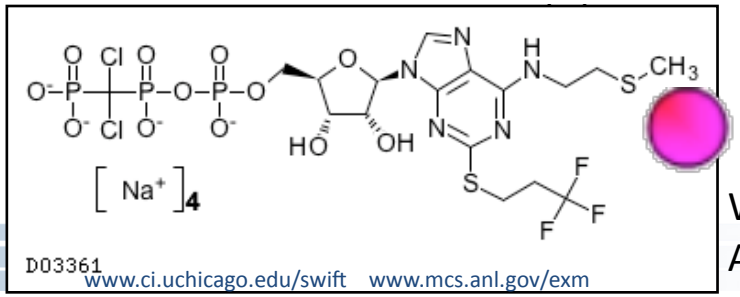






1M compute jobs

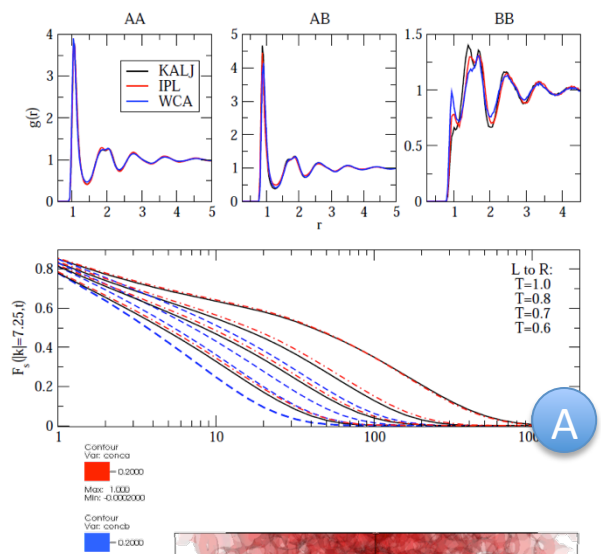
Tens of fruitful candidates for wetlab & APS

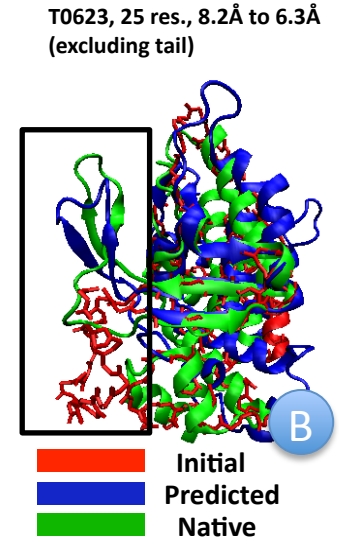


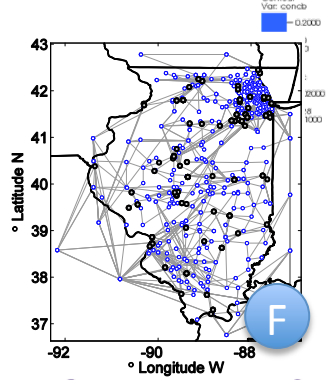
Work of M. Kubal, T.A.Binkowski, And B. Roux

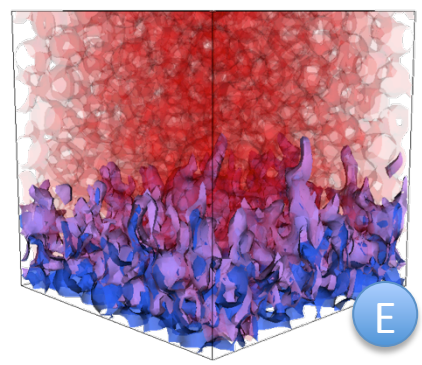
Numerous many-task workflow applications

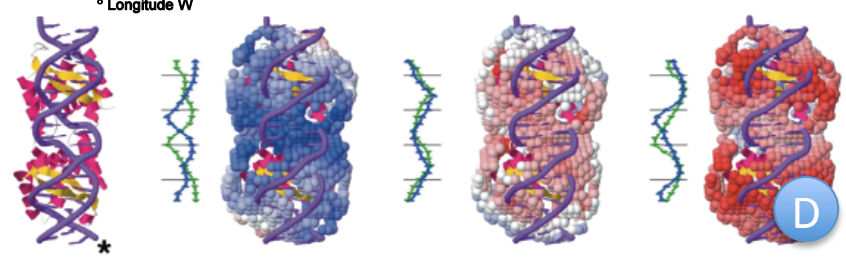
- A Simulation of super-cooled glass materials
- B Protein folding using homology-free approaches
- C Climate model analysis and decision making in energy policy
- D Simulation of RNA-protein interaction
- E Multiscale subsurface flow modeling
- F Modeling of power grid for OE applications

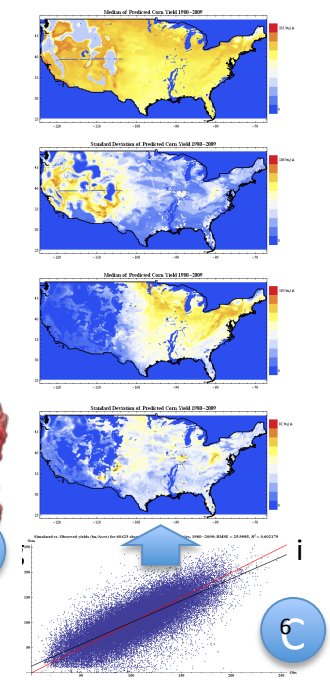








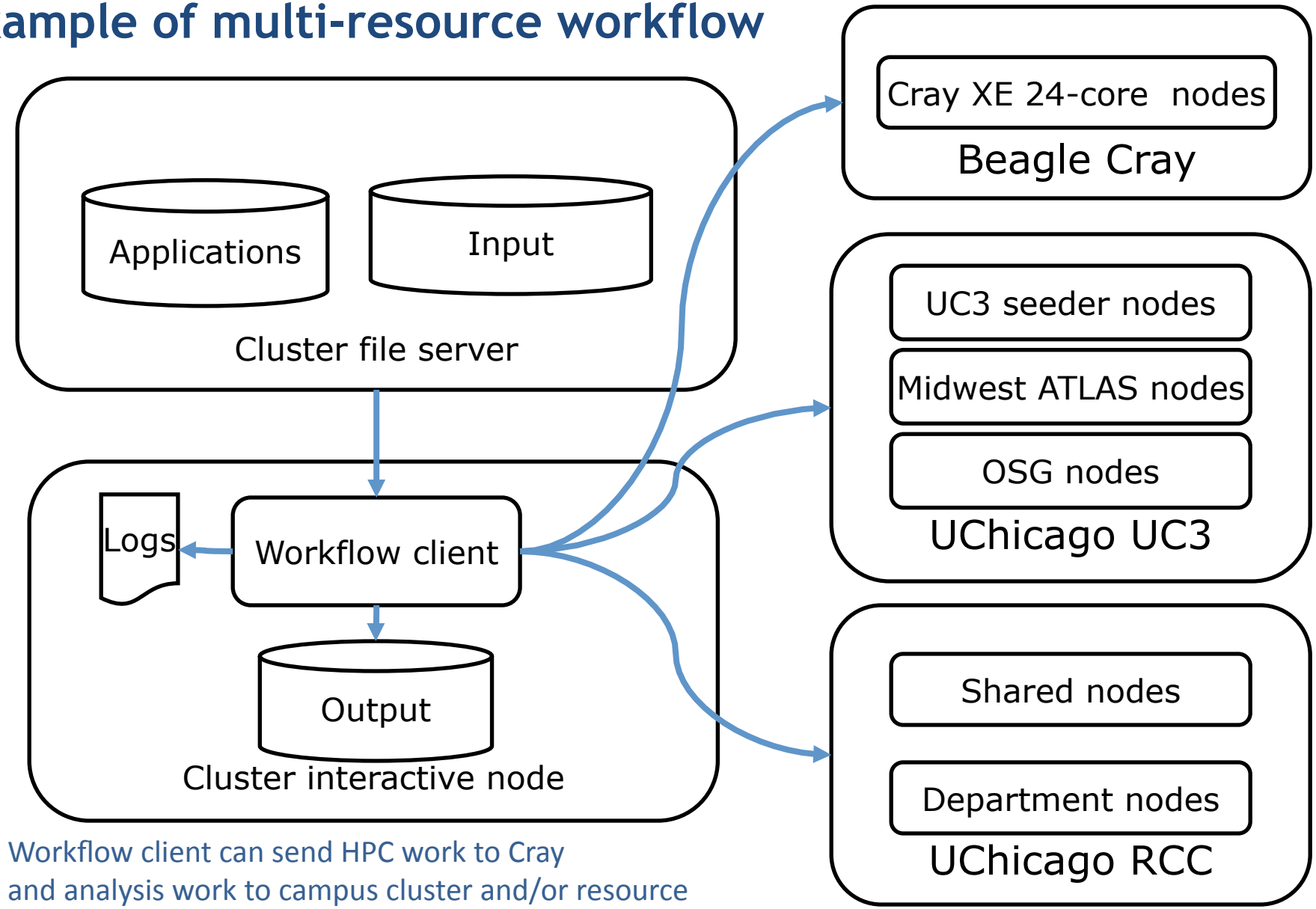




TARGET RESOURCES

- System types
  - Clouds
  - Clusters (campus, department)
  - Petascale HPC systems
  - Grids (OSG, LCG, …)
  - Multi/many-cores – 256 core nodes!
- Patterns
  - A single big HPC machine
  - HPC Machine with attached resources
  - Extend campus cluster with cloud
  - Many HPC machines
  - Many combinations of above

Example of multi-resource workflow



Workflow client can send HPC work to Cray and analysis work to campus cluster and/or resource aggregator

Workflow patterns and issues

- Parameter sweeps
- Ensembles
- Data analysis
- Scaling studies
- Specialized patterns: uncertainty quantification, branch and bound
- Programming an application from libraries of applications
- Dataflow vs control flow
  - Ultimately, workflow is essentially dataflow
  - The difference is who writes and thinks about the dataflow
- Pipelining and concurrency (and how dataflow is good at this)
- Workflow manager drives application (outer workflow, inner scripts)
- Workflow manager embedded in application (outer scripts, inner workflow)

PROGRAMMING MODELS

- MPI, OpenMP, Hybrid
- Map reduce
- Record processing (with functions) vs file processing (with apps)
- Generating workflows for other engines
- Dynamically interpret the workflow
- Script mode (for Blue Gene, Cray systems)
- Dependent job processing

A *partial* sampler of workflow tools

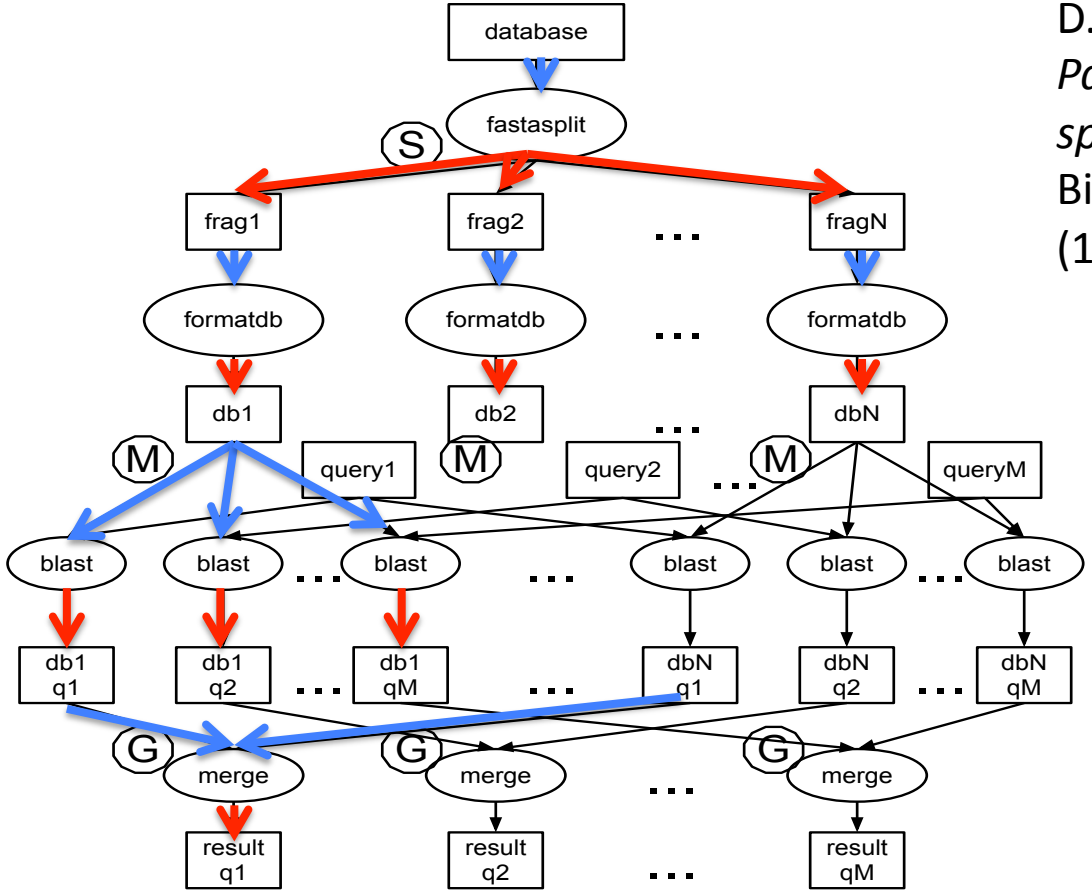
- High throughput tools
  - Condor
  - Cluster schedulers / local resource managers (PBS, SGE, Cobalt, LSF, LL, SLURM,..)
- Workflow task dependency managers
  - DAGMan
  - Schedulers with job dependencies
- Integrated dependency and data management
  - Pegasus
- Dataflow languages
  - Dryad, Ciel, Swift
- Big data solutions
  - Hadoop, Spark, Zookeeper, Uzi
- Multicore tools
  - GNU Parallel
- Languages with parallel support
  - Py_nnn, Java_nnn, Haskell, R, MATLAB => PSOM, Parallel BASH (Walker)

A *sampler* of workflow tools - part II

- Interactive workflow frameworks
  - Galaxy
  - Taverna
  - Kepler
  - LONI Pipeline (neuroscience)
  - Microsoft Workflow manager
  - Airivata
- Science gateways

Parallel BLAST as a workflow

Original script by D. R. Mathog, *Parallel BLAST on split databases.* Bioinformatics, 19 (14), 2003.

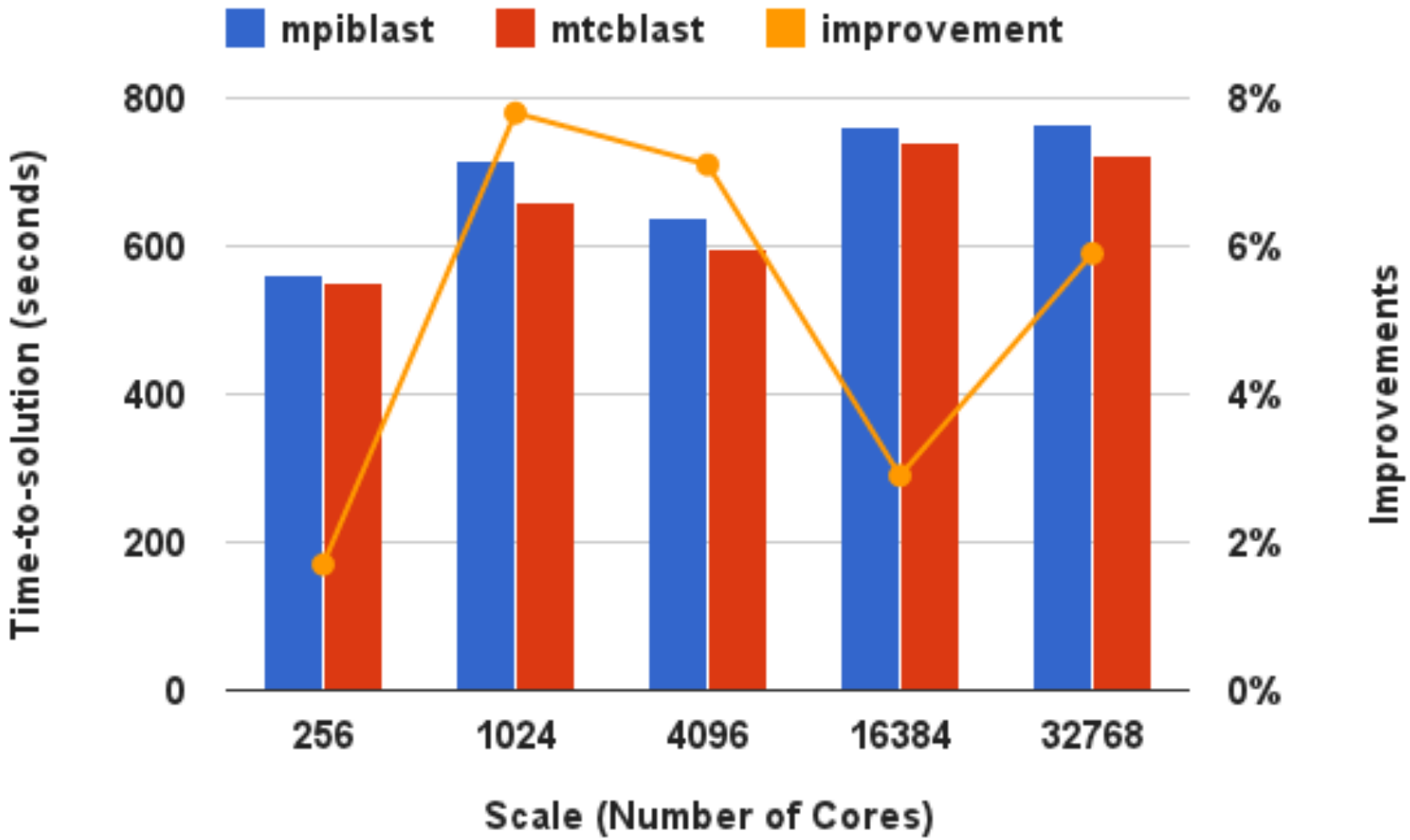


BLAST STAGE TASKS, INPUTS, OUTPUTS, AND INPUT AND OUTPUT SIZE

Stage	# Tasks	# In	# Out	In (MB)	Out (MB)
fastasplitn	1	1	N	4039	4039
formatdb	N	N	3N	4039	4400
blastp	N*M	N+M	N*M	73*N*M	2.4*N*M
merge	M	N*M	M	2.4*N*M	4.8*M

Based on script of D. Matthog by Z.Zhang, L. Gahelha

Can workflow scale?
BLAST workflow lags MPI BLAST by ~ 5%



Z. Zhang, D. S. Katz, J. Wozniak, A. Espinosa, I. Foster. "Design and Analysis of Data Management in Scalable Parallel Scripting", Supercomputing 2012.

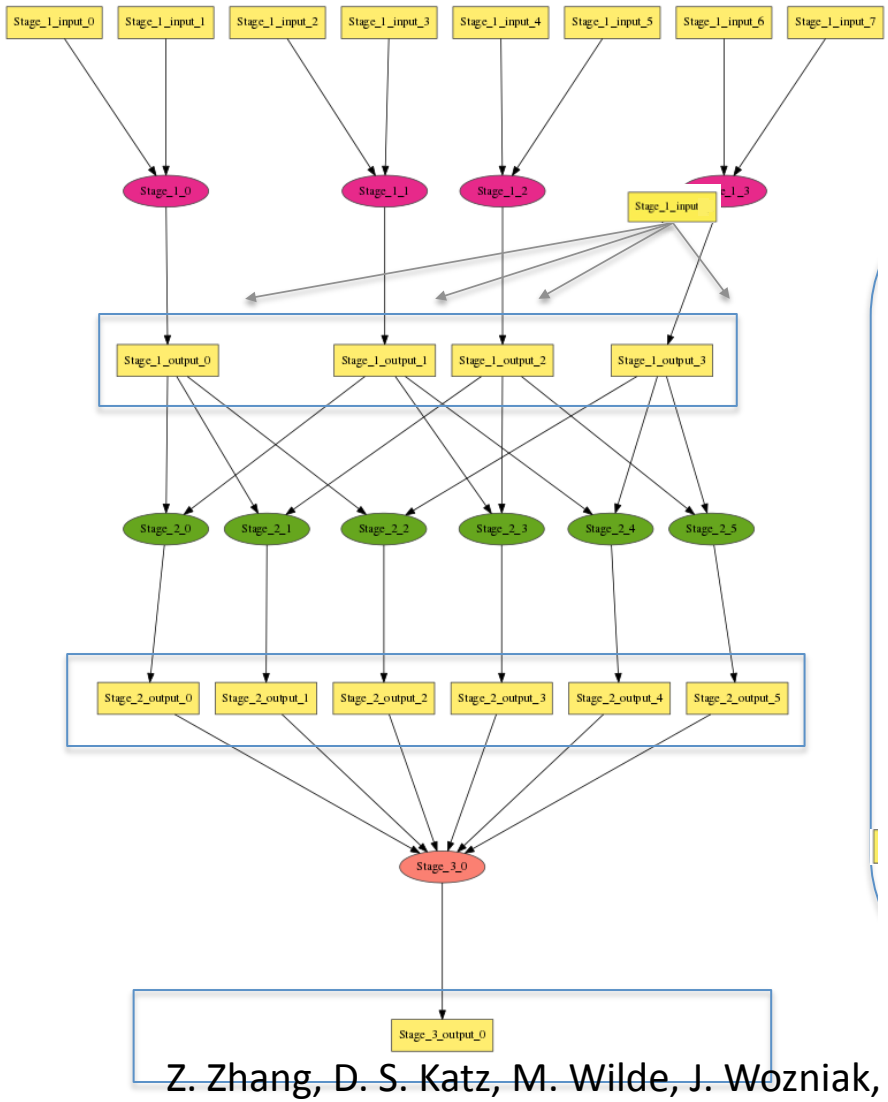
Two fundamental problems in scaling workflow

- Task rate
  - 60,000 cores / 60 sec/task = 1,000 tasks per second!
- Data management
  - 1K tasks / sec may generate 5GB/sec – not so bad if blocked efficiently
  - 1K tasks / sec may generate 2,000 files / sec – not so easy

Multi-level scheduling: pilot jobs can improve task rate performance

- Pilot jobs are long-running meta-jobs
  - allocate compute resources and run many smaller jobs
- PANDA
  - Widely used on OSG and LCG by the ATLAS physics collaboration
- GWMS using Condor Glide-Ins
  - A generalized solution widely deployed on OSG
- SAGA and Bigjob
  - Obtaining good results on XSEDE resources
- Java CoG Coasters
  - Allocates/frees resources based on demand
  - Peaks at 600 tasks per second
- Falkon
  - Research system reached 3,000 tasks per second and 1B tasks

Workflow patterns and data exchange



Filesystem Access Patterns:

- File Creation
- File Open
- 1-to-1 Read
- N-to-1 Read
- Few-to-1 Read
- 1-to-1 Write

Z. Zhang, D. S. Katz, M. Wilde, J. Wozniak, I. Foster. MTC Envelope: Defining the Capability of Large Scale Computers in the Context of Parallel Scripting Applications, HPDC 2013.

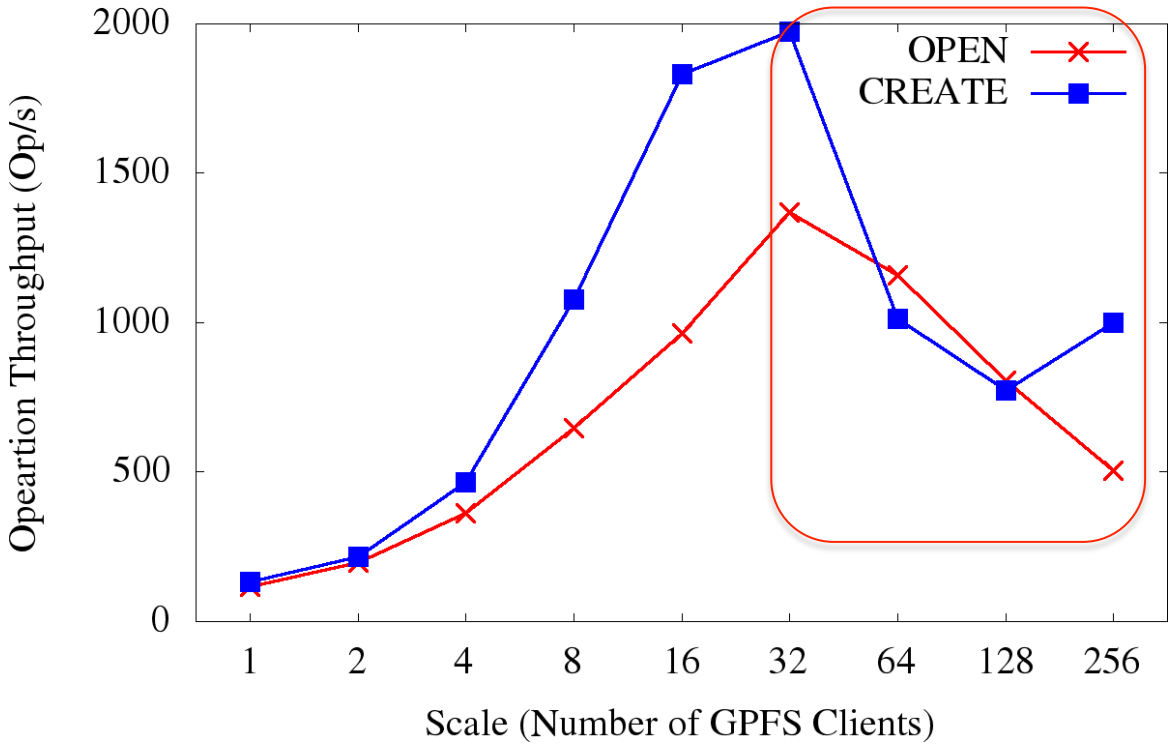
Measuring MTC Envelope:

- Target platform
  - GPFS deployed with ANL Intrepid BlueGene/P
  - Several metadata servers, but only one for each directory
  - 128 IBM x3545 file servers, each with two 2.6-GHx Dual Core CPUs and 8 GB RAM
  - We use I/O nodes as GPFS clients. (Ratio between I/O nodes and CPU cores – 1:256)
- Experimental setup
  - Metadata operation: {create, open} x {1, 2, 4, 8, 16, 32, 64, 128, 256} clients
  - I/O: {read, write} x {1 KB, 128 KB, 1 MB, 16 MB} x {1, 2, 4, 8, 16, 32, 64, 128, 256} clients
  - Total number of operations fixed at 8192 at each scale
  - All files are in one directory

Performance envelope research and slides by Zhao Zhang

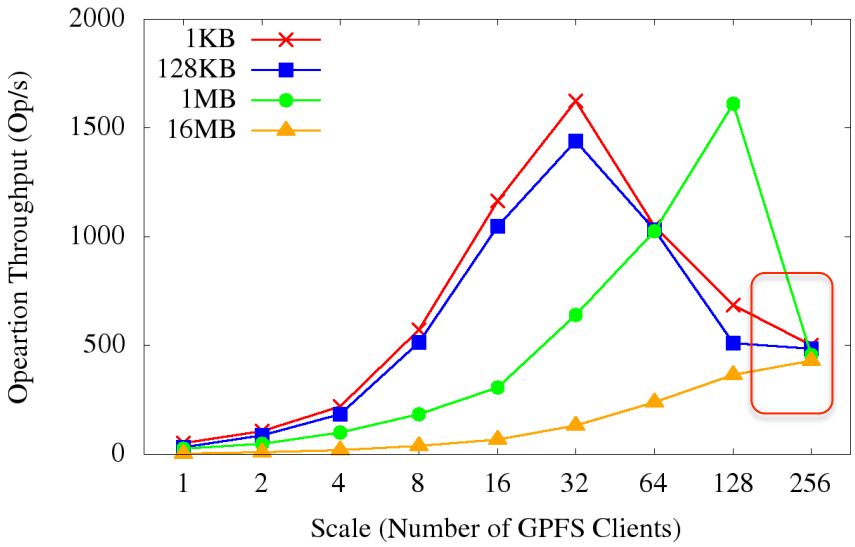
Measuring MTC Envelope

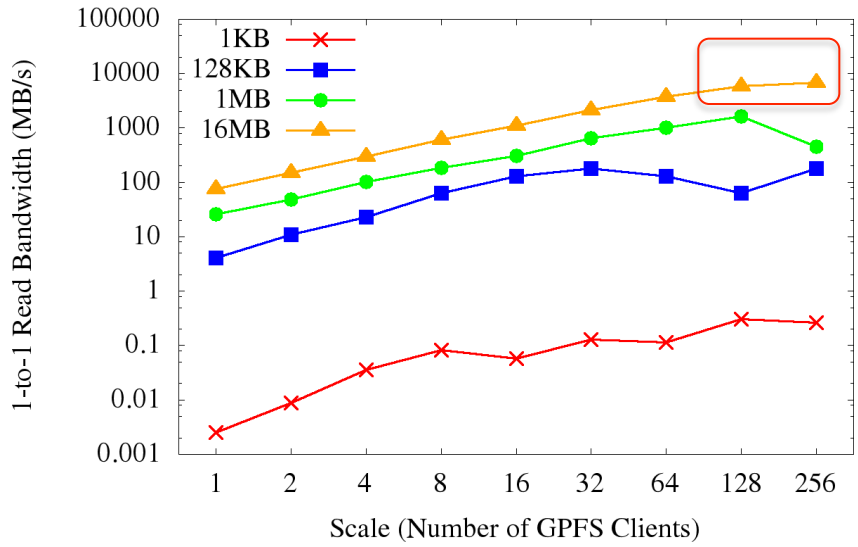
- Metadata Operation Throughput



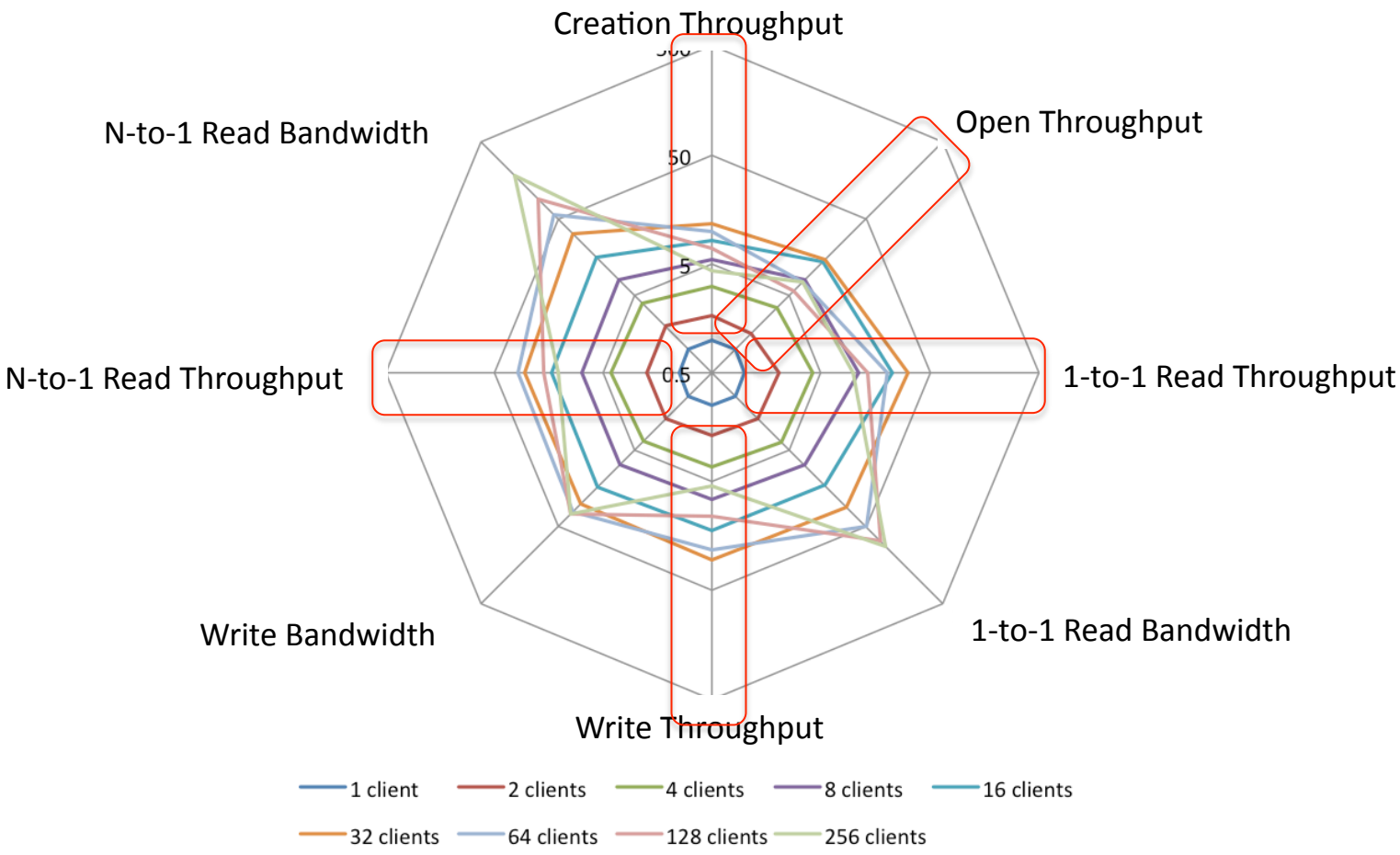
Measuring MTC Envelope

- 1-to-1 Read



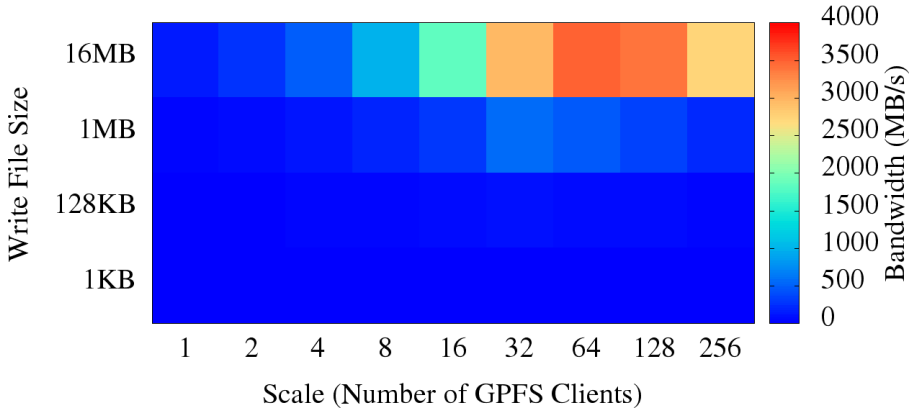


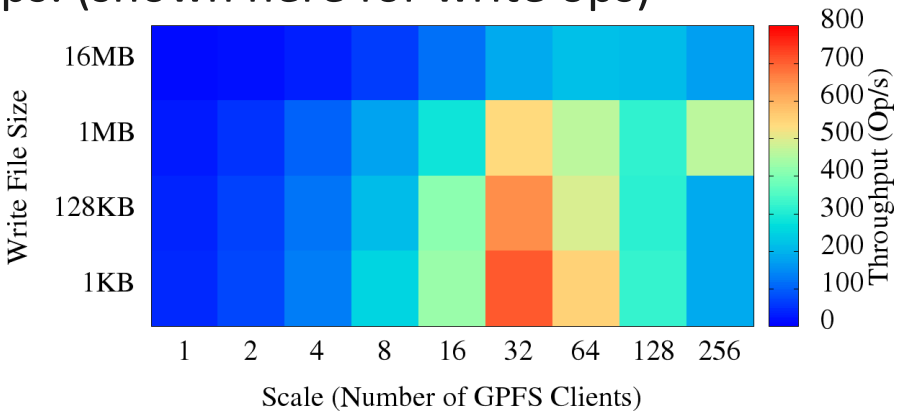
MTC Envelope vs. Scale



Performance guide for workflows

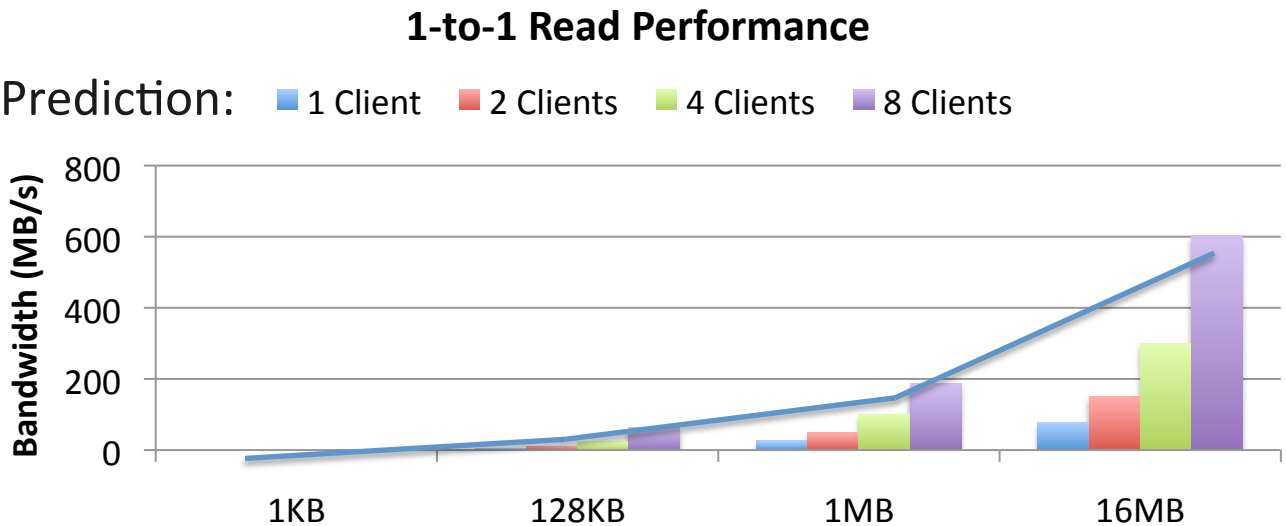
- MTC Envelope expressed as heat maps: (shown here for write ops)





- Use throughput heat map when files are small, and use bandwidth heat map when files are large.

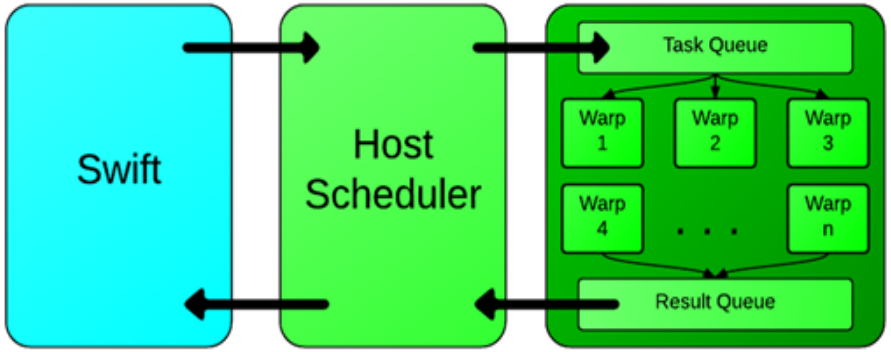
- I/O Performance Prediction:



Some engineering problems and research challenges for extreme workflow

- Engineering
  - Diversity of interfaces, hard to tame and test, hard to abstract
  - Inter-language bindings and data interchange – challenge to usability
  - Integration with extreme-scale networks, runtimes and language stacks
- Research
  - Economics and policy-based scheduling
  - Retry/recovery of large distributed task and data graphs
  - Power management
  - Load balancing
  - Programming models: integration of dataflow and big-data techniques and tools

GEMTC: GPU Enabled Many-Task Computing



Motivation: Support Many-Task Computing on Accelerators

Goals:

1) MTC support
2) Improved programmability
3) MTC efficiency
4) MIMD on SIMD
5) Increase concurrency 12X (16 -> 192 (12x)

Approach:

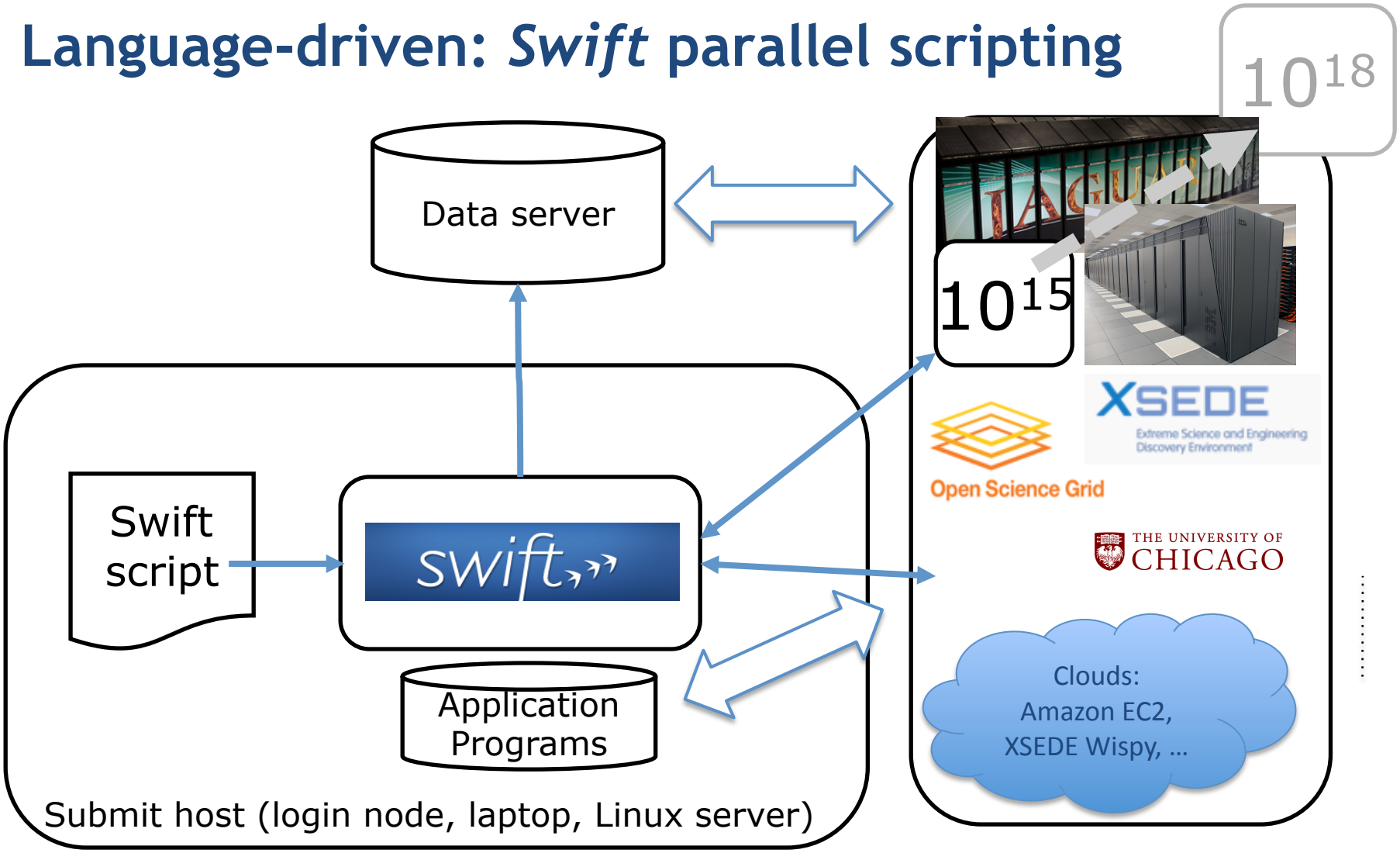
Design, implement middleware:

1) manages GPU
2) spread host/device
3) Workflow system support (Swift/T)

B Grimmer, S Krieder, I Raicu. Enabling Dynamic Memory Management Support for MTC on NVIDIA GPUs, EuroSys 2013 poster.

- Parallel scripting language for clusters, clouds & grids
  - For writing loosely-coupled scripts of application programs and utilities linked by exchanging files
  - Can call scripts in shell, python, R, Octave, MATLAB, …
- Swift does 3 important things for you:
  - Makes parallelism transparent – with functional dataflow
  - Makes basic failure recovery transparent
  - *Makes computing location transparent* – can run your script on multiple distributed sites and diverse computing resources (from desktop to petascale)
    → *this is what we'll show today*

Language-driven: *Swift* parallel scripting



Swift runs parallel scripts on a broad range of parallel computing resources.

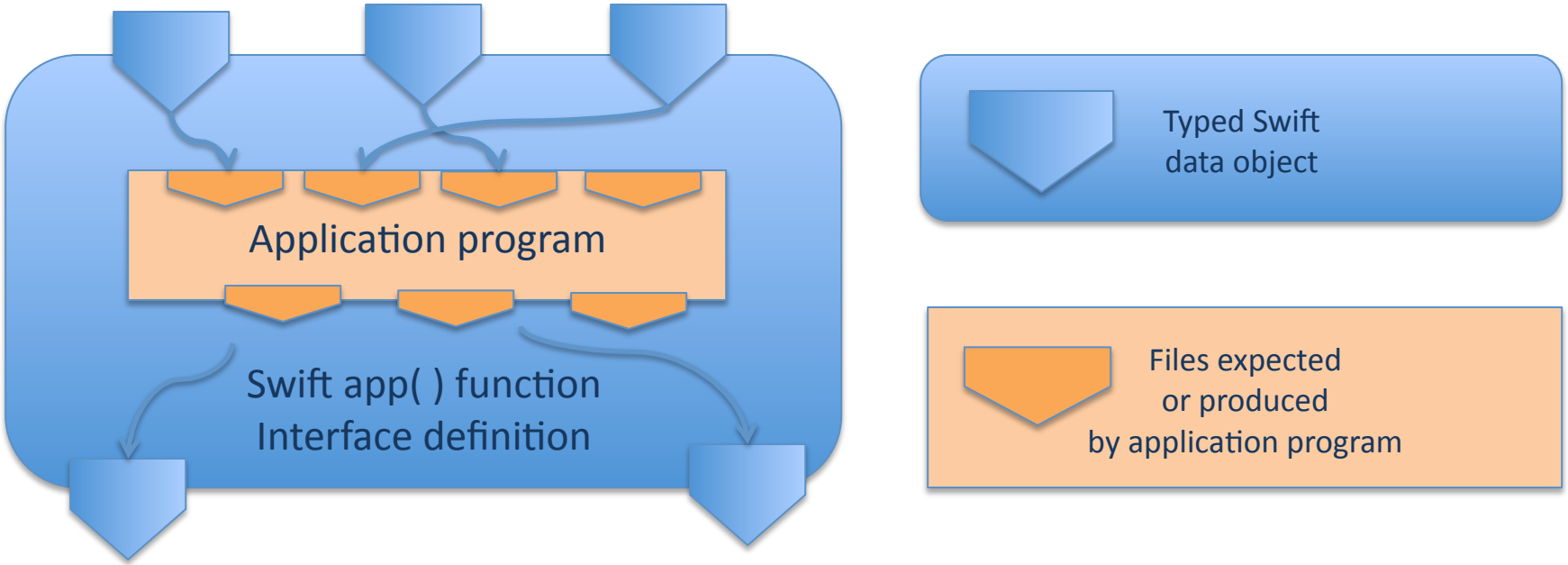
Programming model: all execution driven by parallel data flow

```
(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

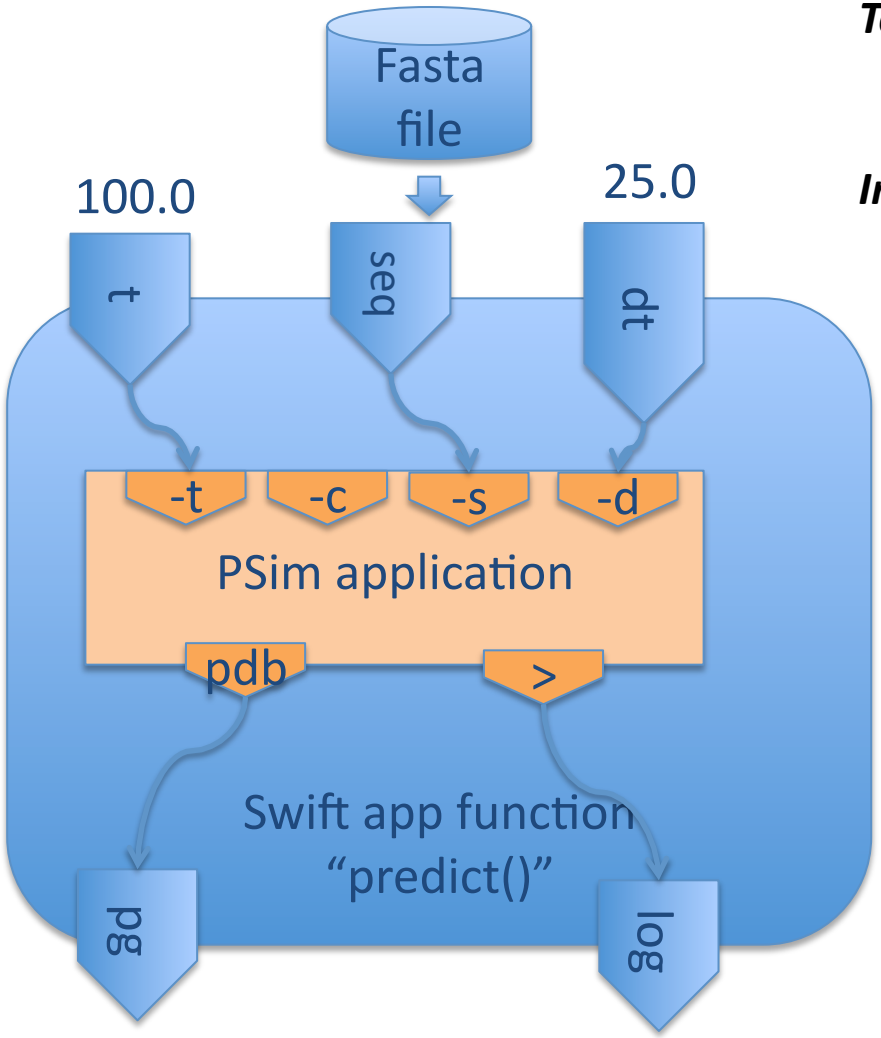
- This parallelism is *automatic*
- Works recursively throughout the program's call graph

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:

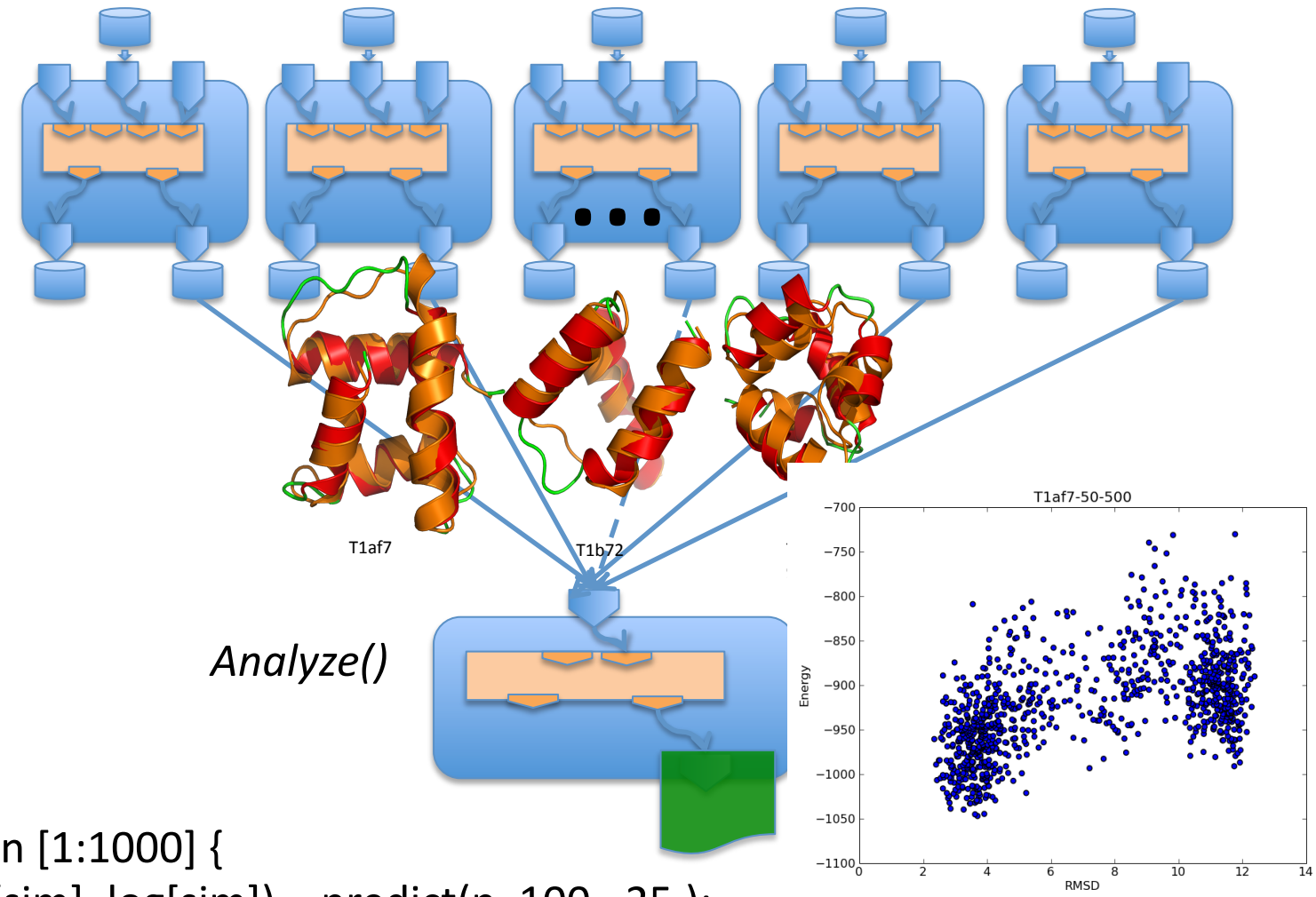
psim -s 1ubq.fas -pdb p -t 100.0 -d 25.0 >log

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.);
```

Large scale parallelization with simple loops

1000 Runs of the "predict" application



Analyze()

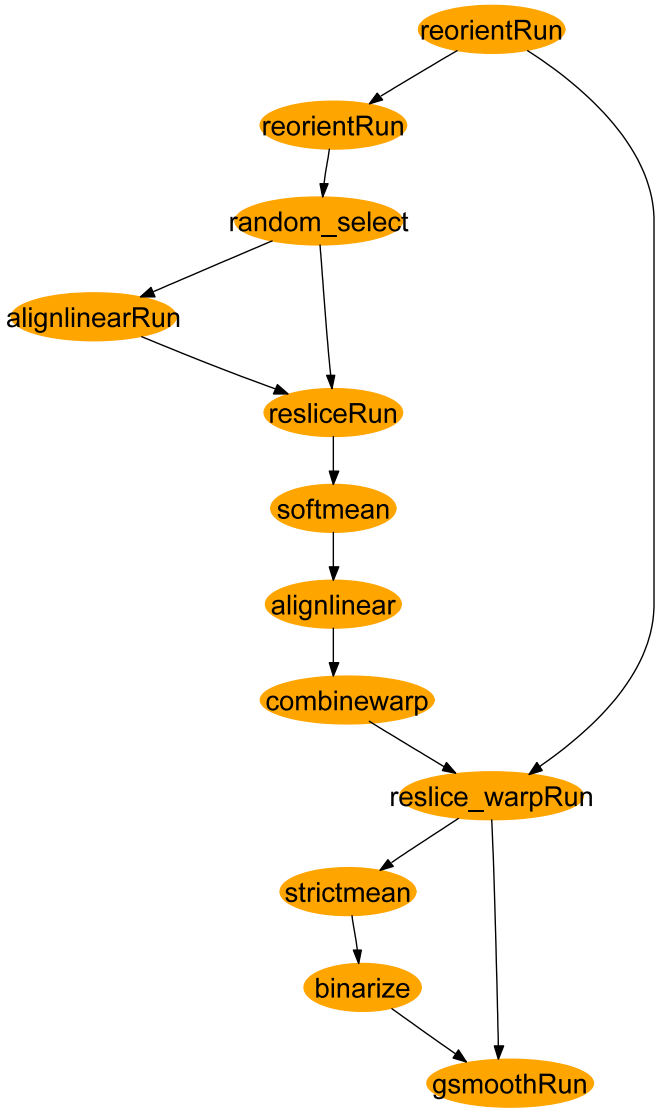
```
foreach sim in [1:1000] {
   (structure[sim], log[sim]) = predict(p, 100., 25.);
}
result = analyze(structure)
```

Nested parallel prediction loops in *Swift*

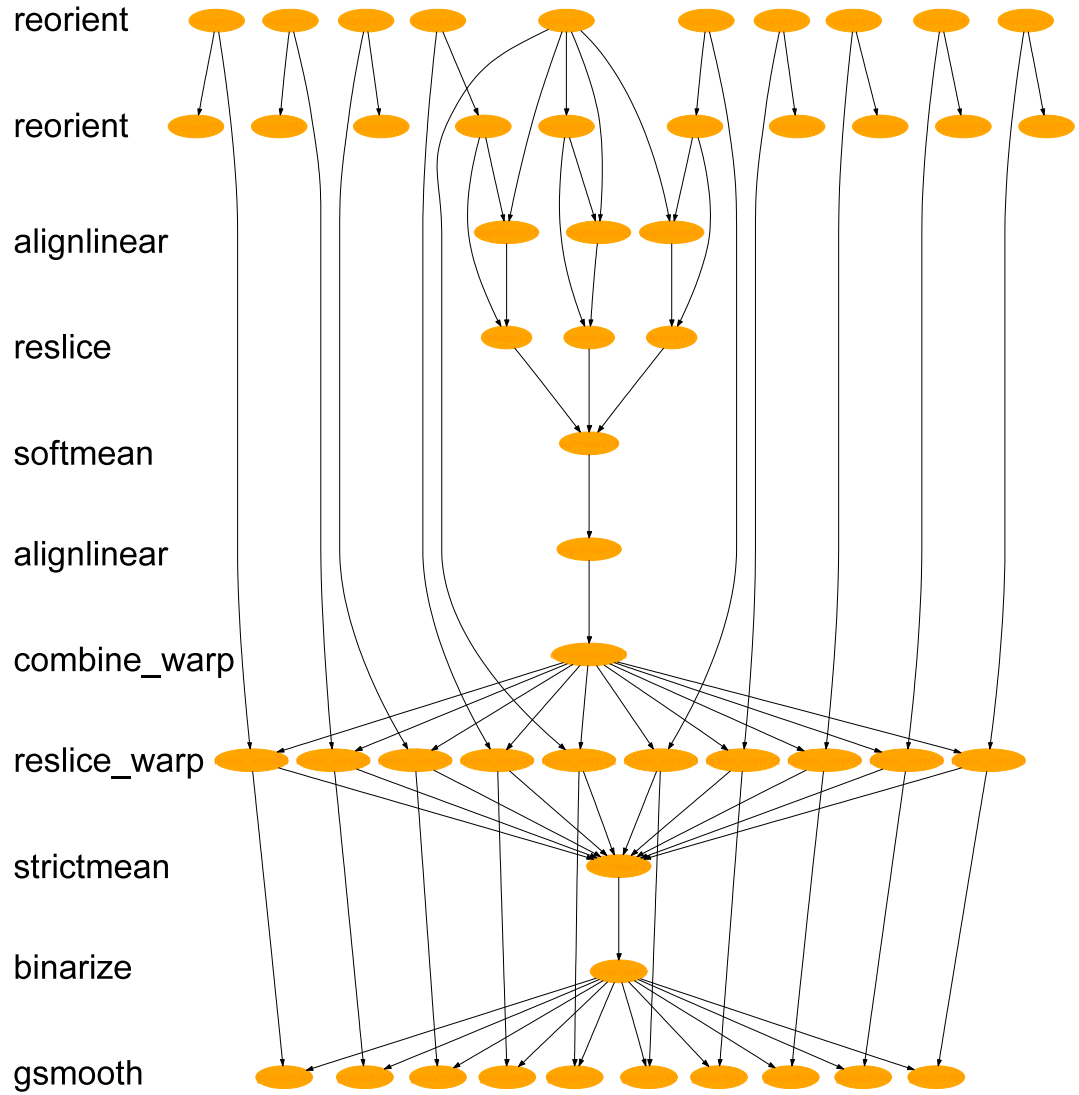
```
Sweep( )
{
  int nSim = 1000;
  int maxRounds = 3;
  Protein pSet[ ] <ext; exec="Protein.map">;
  float startTemp[ ] = [ 100.0, 200.0 ];
  float delT[ ] = [ 1.0, 1.5, 2.0, 5.0, 10.0 ];
  foreach p, pn in pSet {
    foreach t in startTemp {
      foreach d in delT {
        ItFix(p, nSim, maxRounds, t, d);
      }
    }
  }
}
Sweep();
```

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

Spatial normalization of functional run



Dataset-level workflow



Expanded (10 volume) workflow

Complex scripts can be well-structured

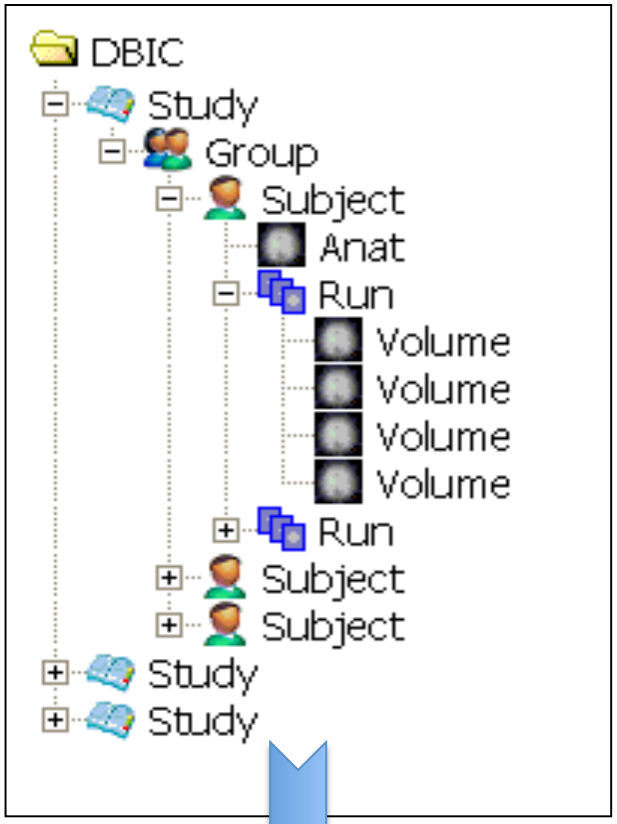
programming in the large: fMRI spatial normalization script example

```
(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);
        }
}
```

Dataset mapping example: fMRI datasets

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

Nested loops can generate massive parallelism

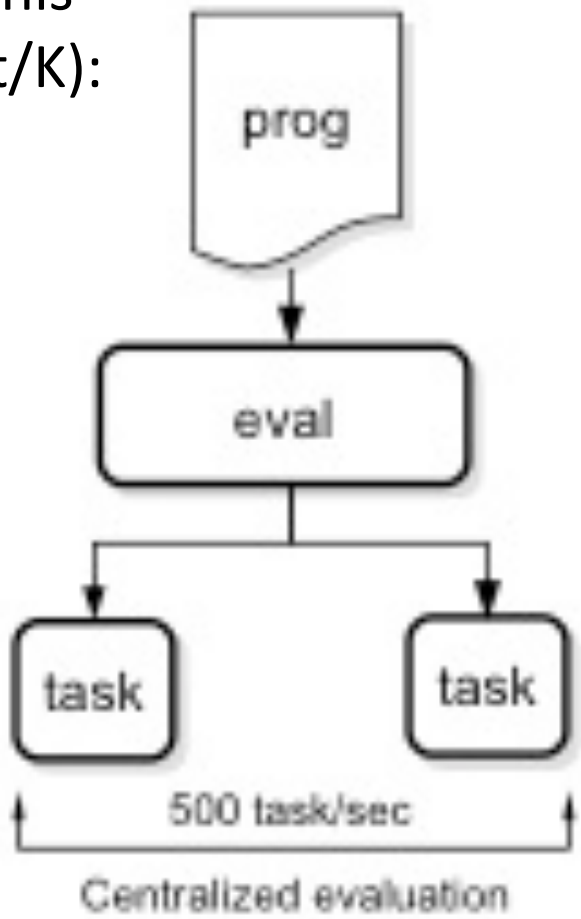
Protein folding example:

```
Sweep( )
{
   int nSim = 1000;
   int maxRounds = 3;
   Protein pSet[ ] <ext; exec="Protein.map">;
   float startTemp[ ] = [ 100.0, 200.0 ];
   float delT[ ] = [ 1.0, 1.5, 2.0, 5.0, 10.0 ];
   foreach p, pn in pSet {
      foreach t in startTemp {
         foreach d in delT {
            ItFix(p, nSim, maxRounds, t, d);
         }
      }
   }
}
Sweep();
```

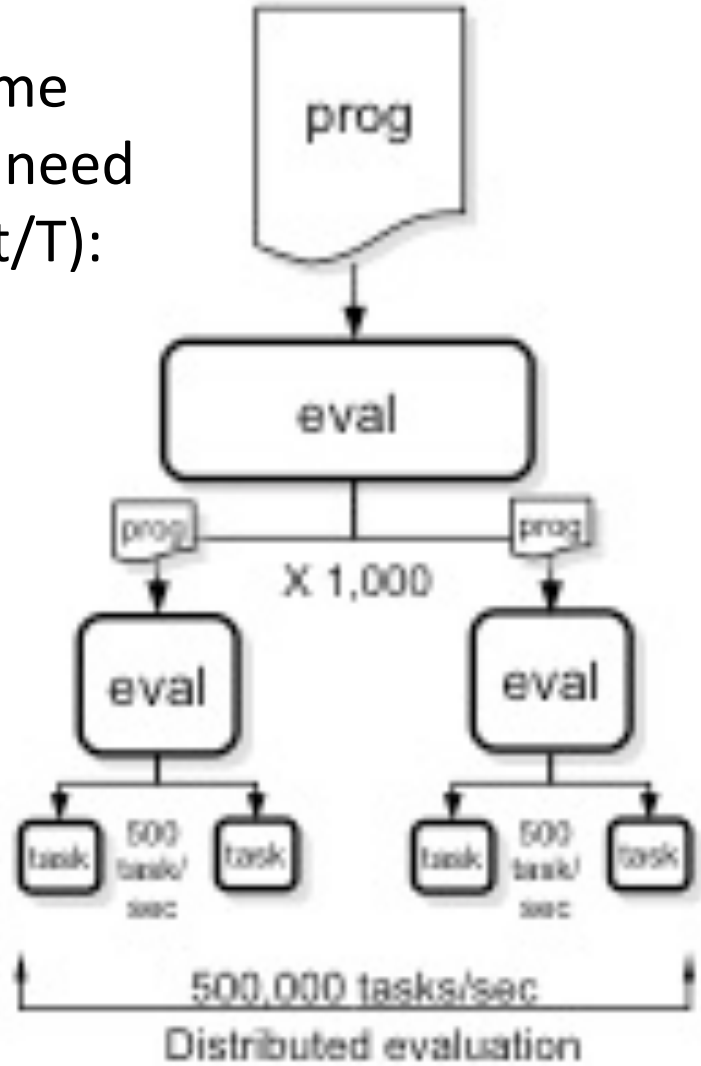
10 proteins x 1000 simulations x
3 rounds x 2 temps x 5 deltas
= 300K tasks

Centralized evaluation can be a bottleneck

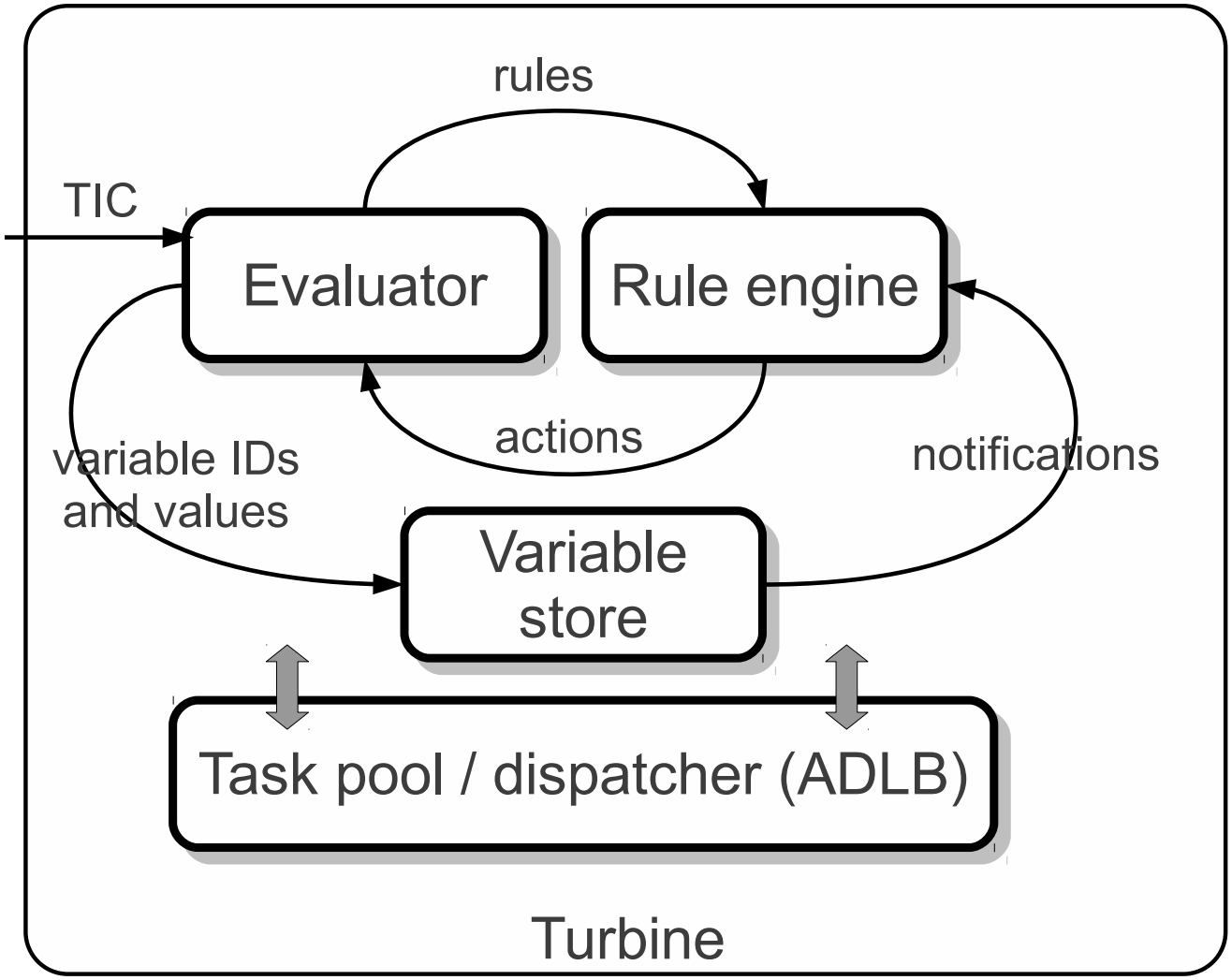
Had this (Swift/K):



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

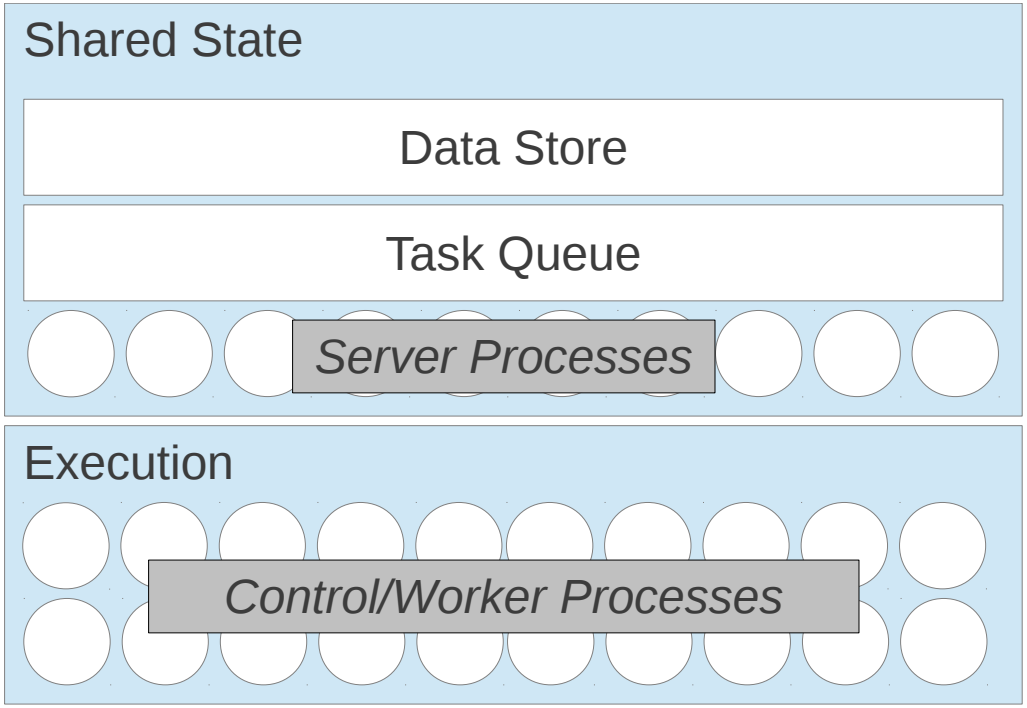


Parallel evaluation of Swift/T in ExM



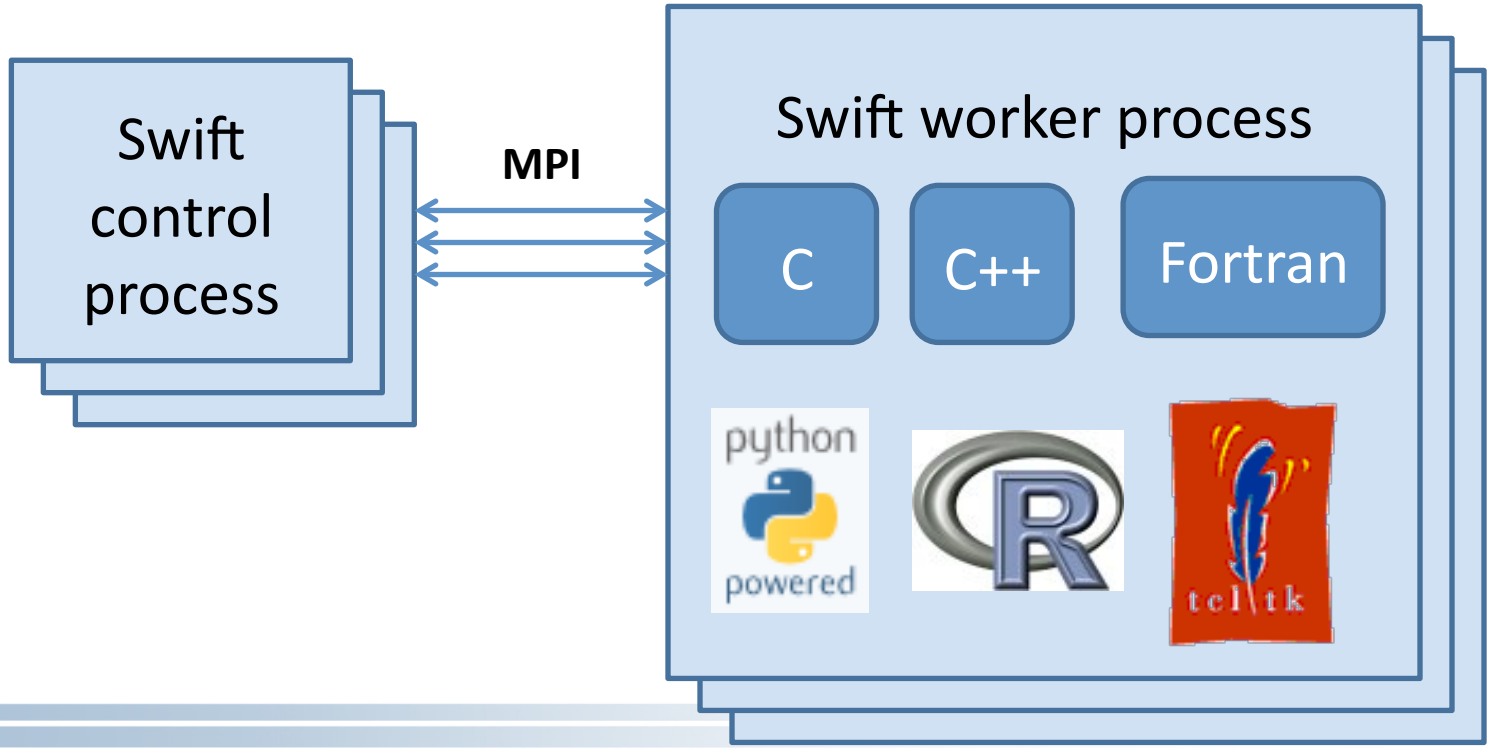
Swift/T: Large-scale application composition via distributed-memory data flow processing
J.M. Wozniak, T.G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, I. Foster Proc. CCGrid 2013.

Swift/T programs run as an SPMD MPI program using ADLB

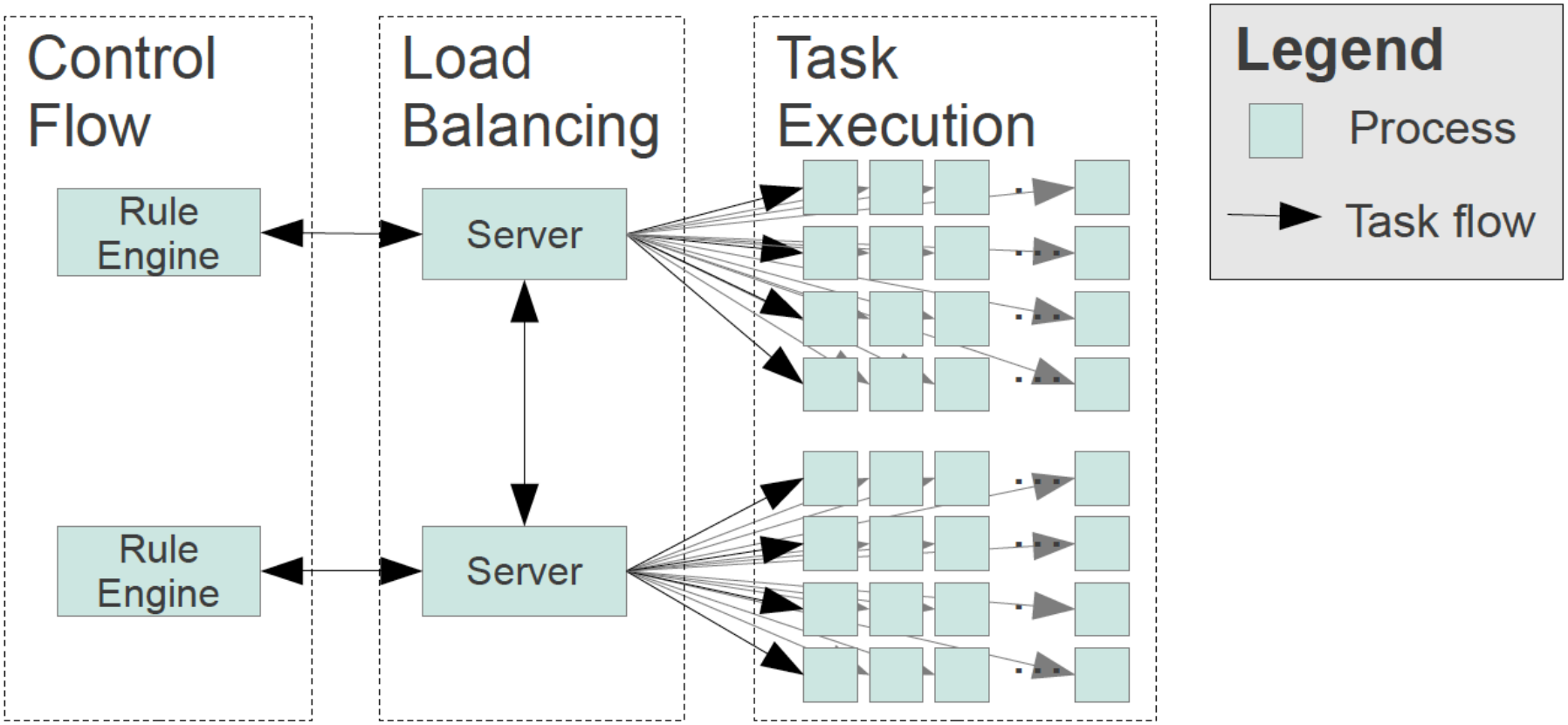


Swift/T: High-level model

- 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.
- Distributed, scalable runtime manages tasks, load balancing, data movement
- User function calls to external code run on 1000’s of workers
- Like master-worker but with the expressive Swift language to control progress

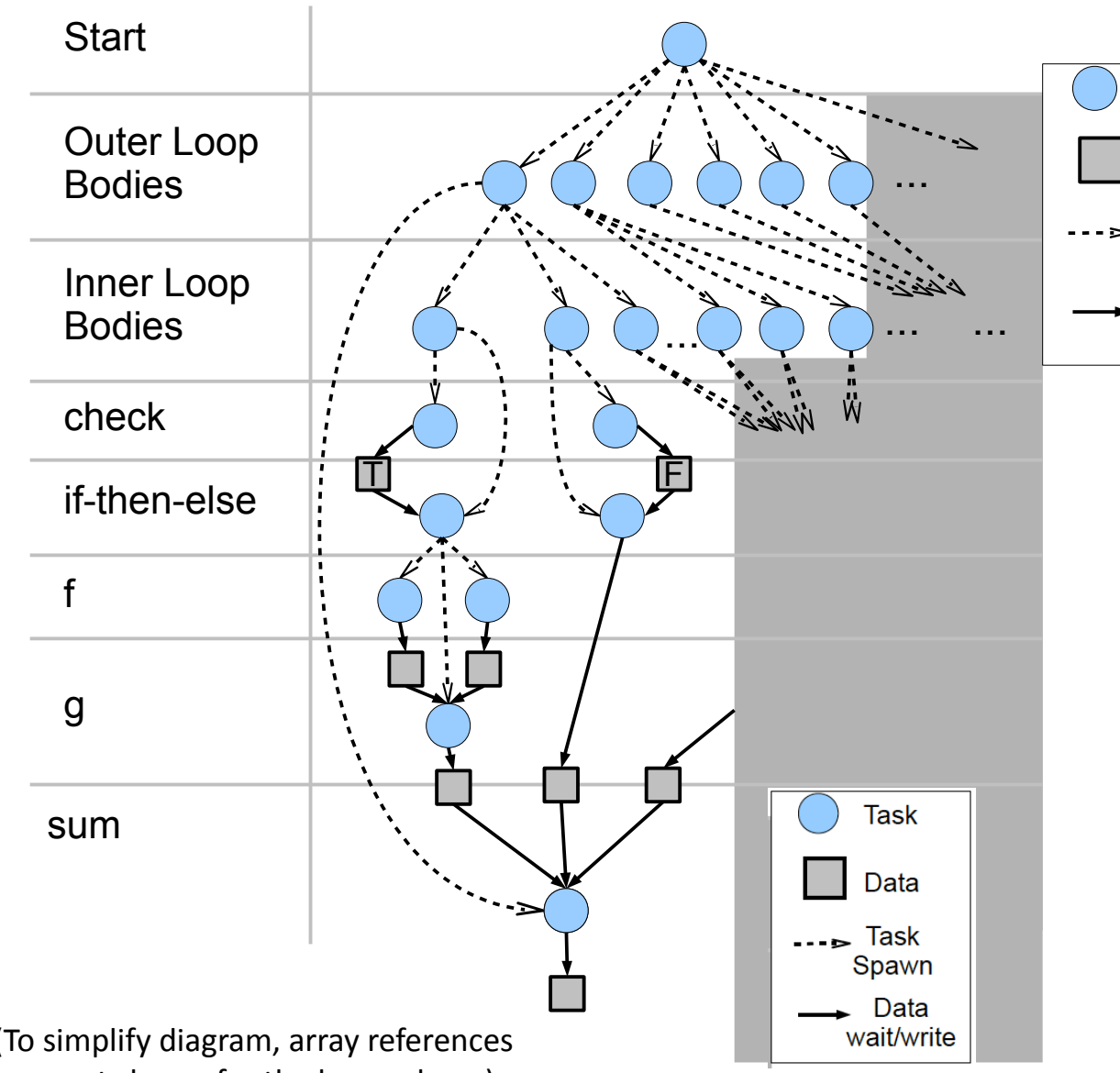


MPI process architecture for parallel evaluation in Swift/T



Parallel evaluation in action

```
int X = 100, Y = 100;
int A[][];
int B[];
foreach x in [0:X-1] {
  foreach y in [0:Y-1] {
    if (check(x, y)) {
      A[x][y] = g(f(x), f(y));
    } else {
      A[x][y] = 0;
    }
  }
  B[x] = sum(A[x]);
}
```

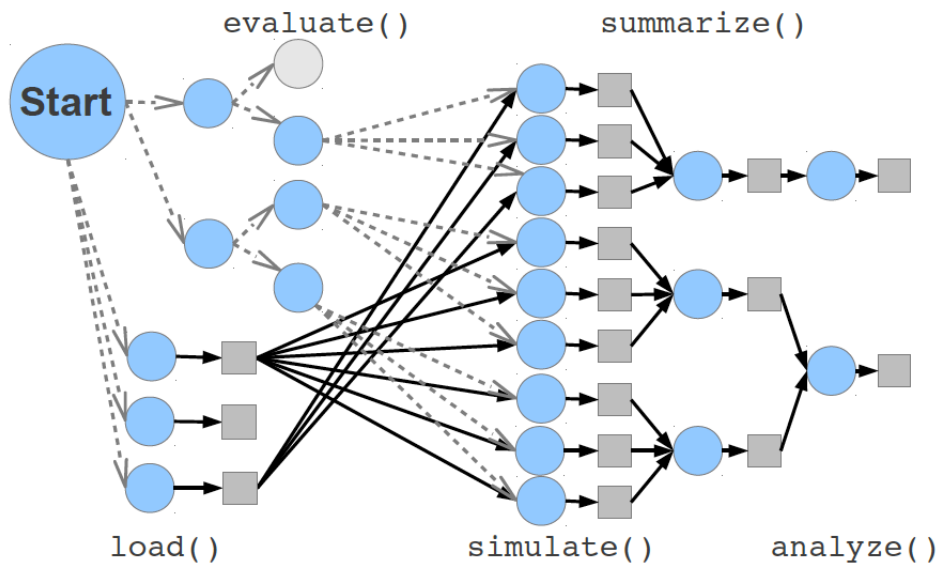


(To simplify diagram, array references are not shown for the loops above)

Pervasive implicit parallelism with automatic pipelining

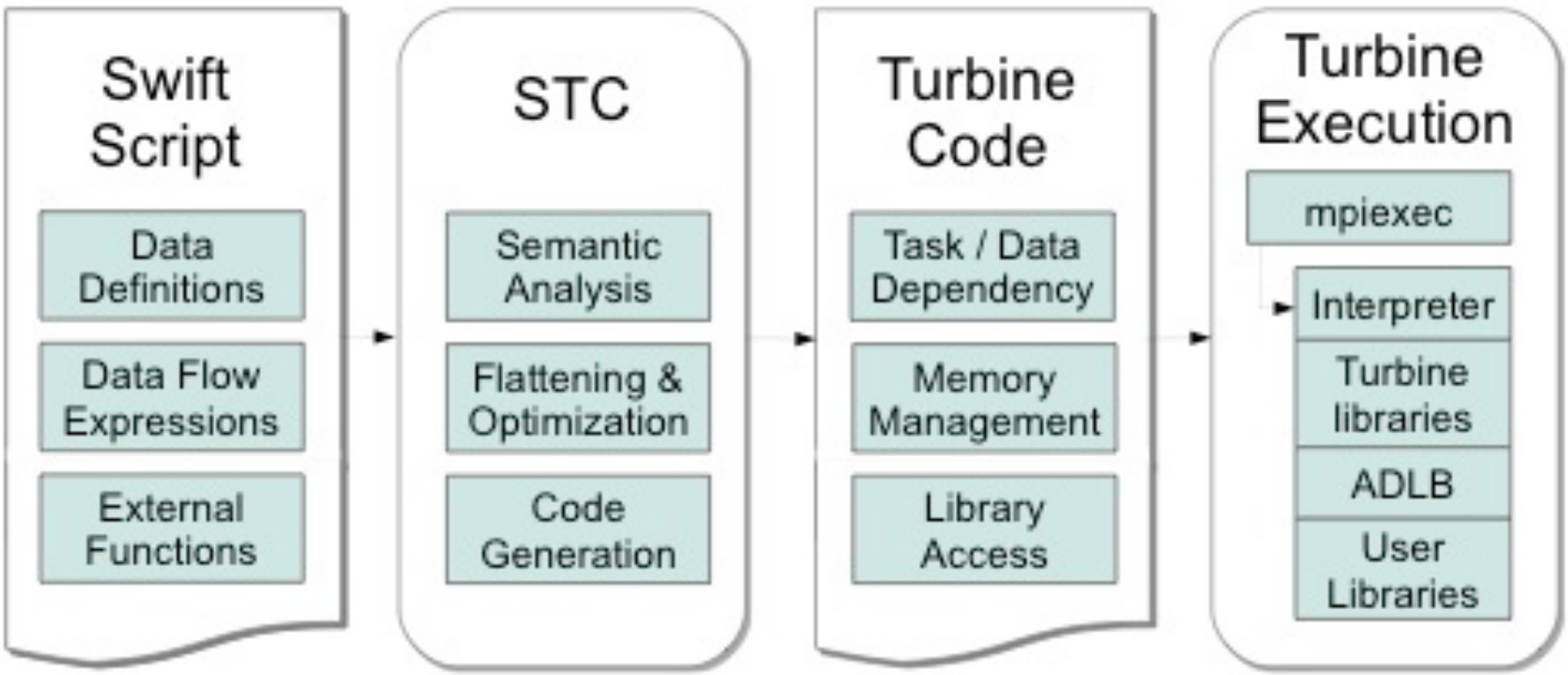
```
blob models[], res[][];
foreach m in [1:N_models] {
  models[m] = load(sprintf("model%i.data", m));
}

foreach i in [1:M] {
  foreach j in [1:N] {
    // initial quick evaluation of parameters
    p, m = evaluate(i, j);
    if (p > 0) {
      // run ensemble of simulations
      blob res2[];
      foreach k in [1:S] {
        res2[k] = simulate(models[m], i, j);
      }
      res[i][j] = summarize(res2);
    }
  }
}

// Summarize results to file
foreach i in [1:M] {
  file out<sprintf("output%i.txt", i)>;
  out = analyze(res[i]);
}
```

(a) Declarative Swift/T code

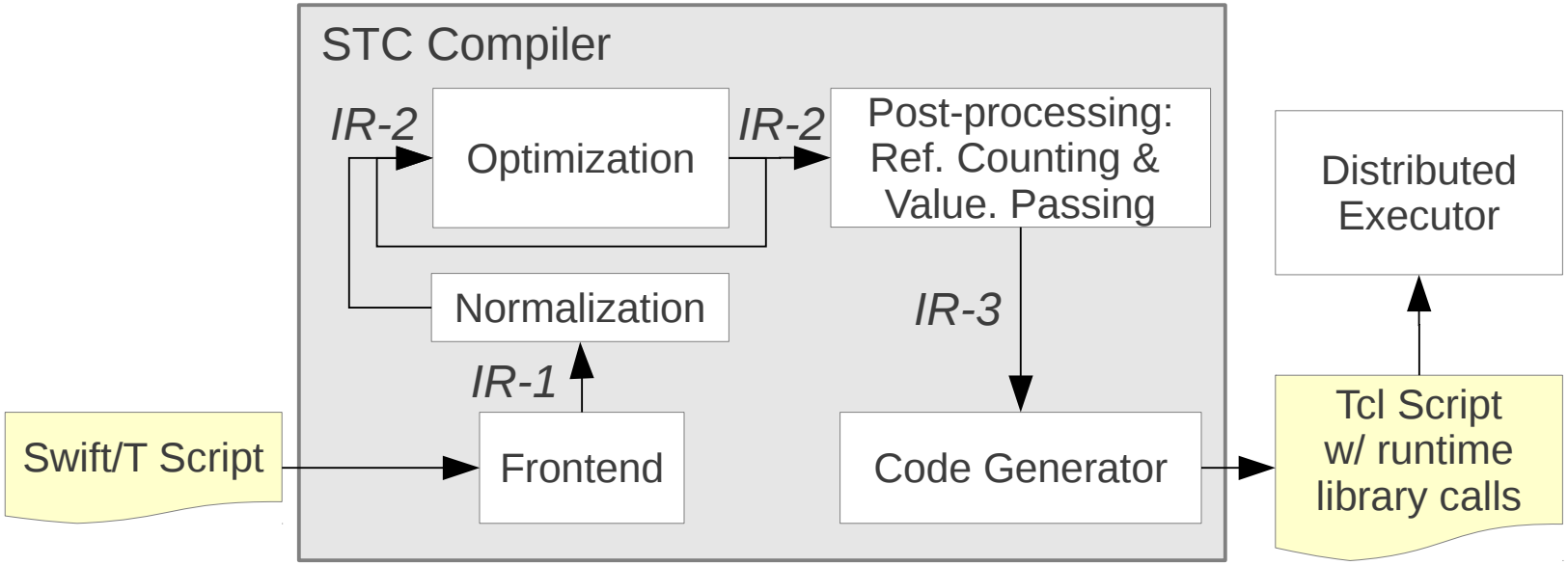


(b) Visualization of parallel execution for $M = 2$ $N = 2$ $S = 3$

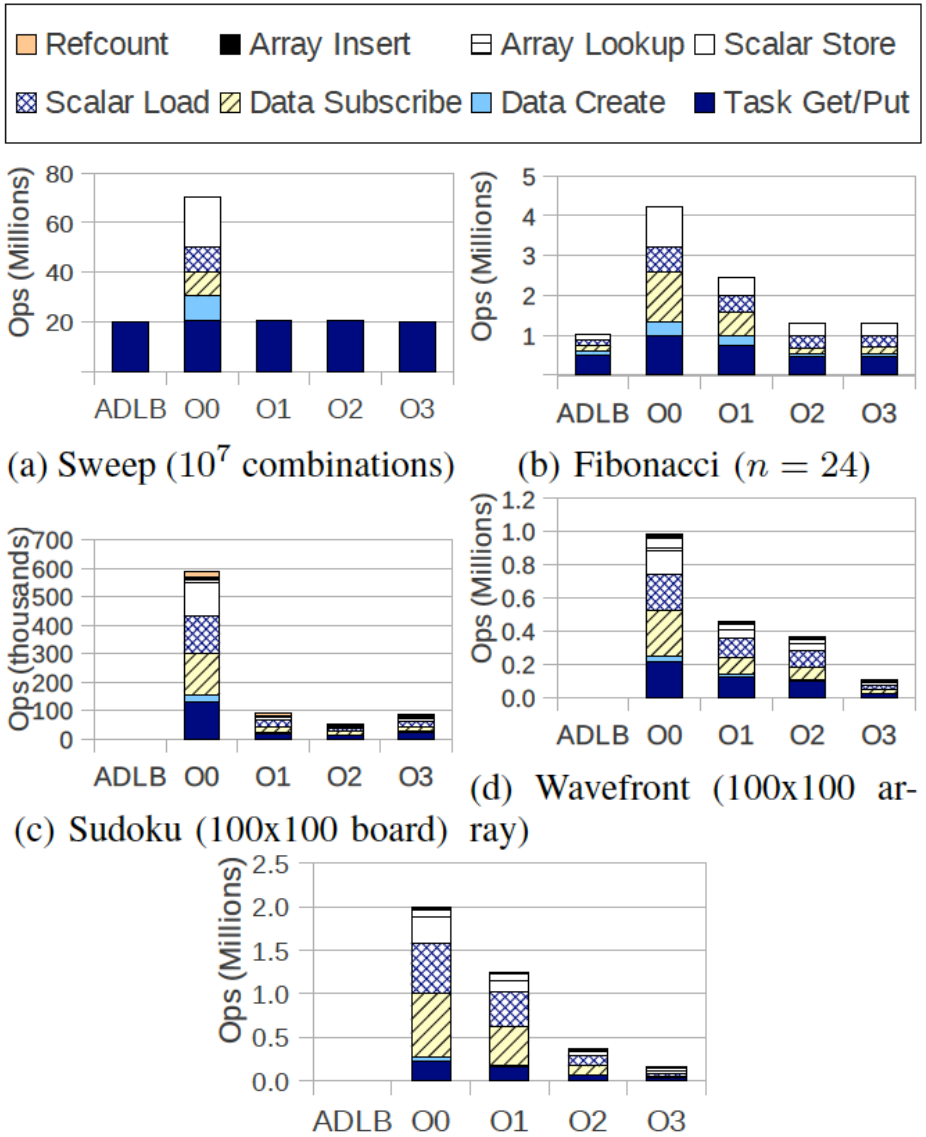
Swift/T toolchain and runtime environment



Inside the Swift/T "stc" compiler



Operation reduction optimizations by stc



(a) Sweep (10^7 combinations)

(b) Fibonacci ($n = 24$)

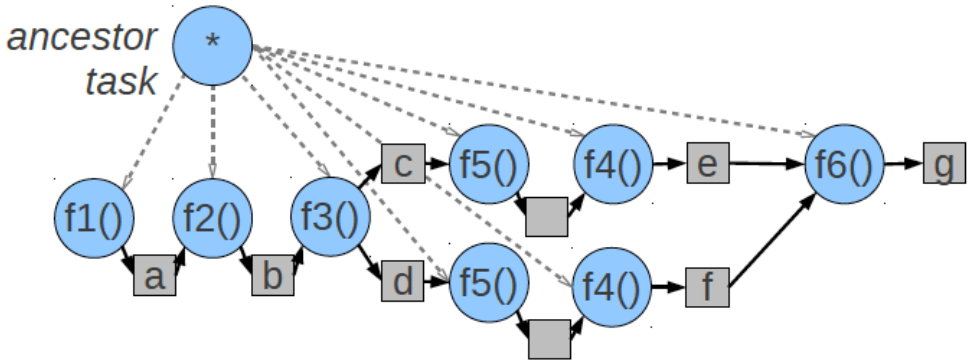
(c) Sudoku (100x100 board)

(d) Wavefront (100x100 array)

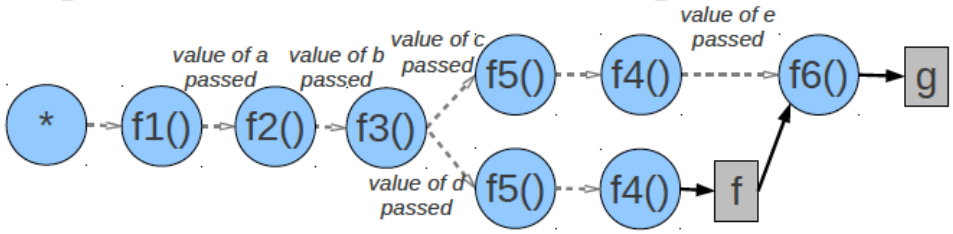
(e) Simulated Annealing (125 iterations, 100-way objective function parallelism)

```
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);
```

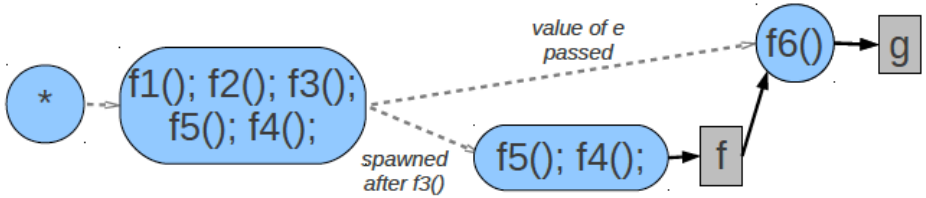
(a) Swift code fragment



(b) Unoptimized version, relying on shared data flow variables to pass data and runtime data dependency tracking



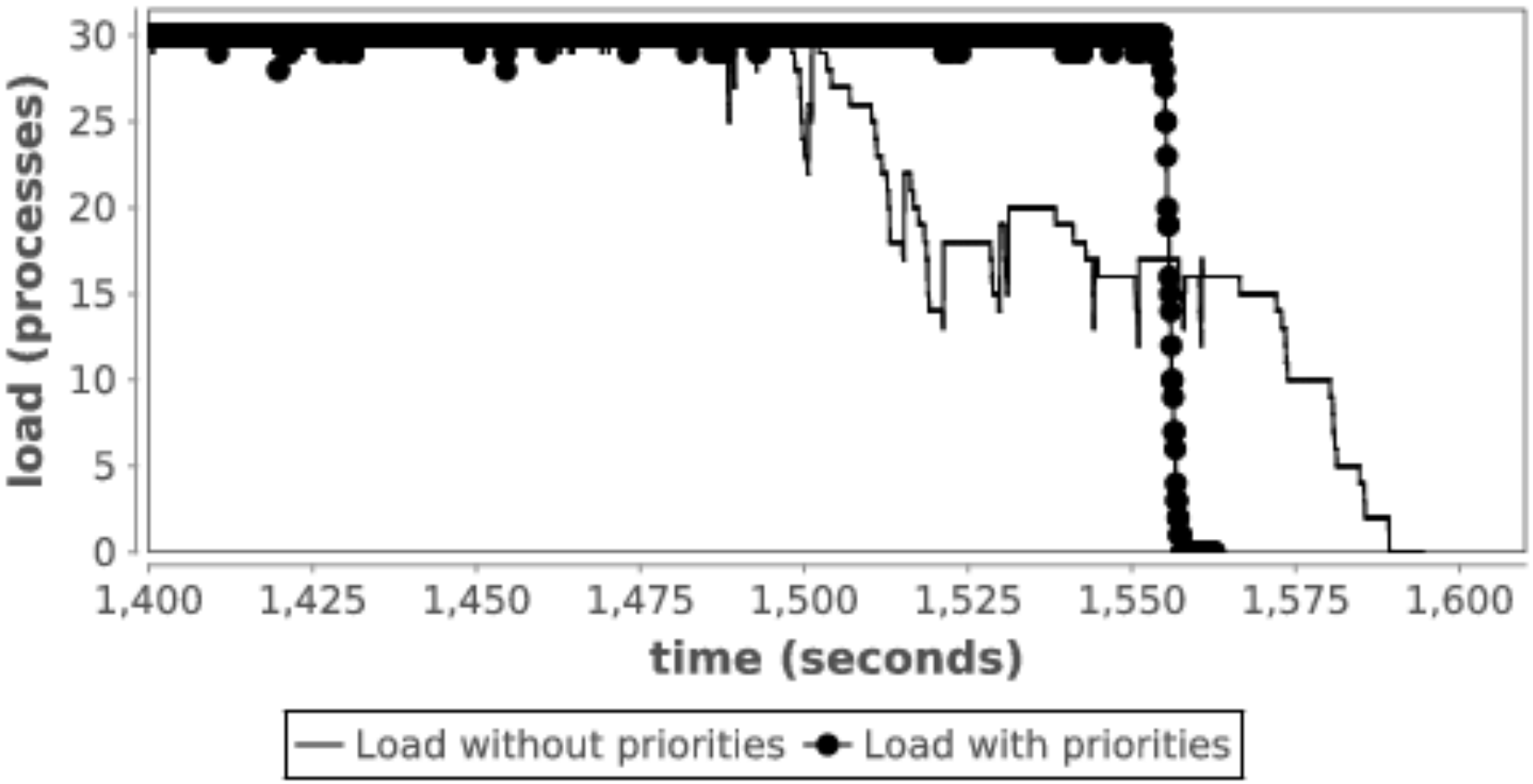
(c) After wait pushdown and elimination of shared variables in favor of parent-to-child data passing



(d) After pipeline fusion merges tasks

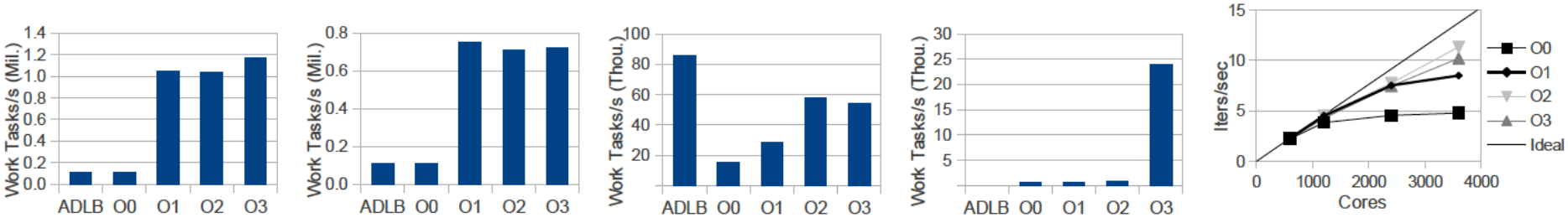
Task priority can be specified to reduce tail effects

- Variable-sized tasks produce trailing tasks:
  addressed by exposing task priorities at language level



Performance results:

10 Cray XE MC-12 24-Core nodes,
2 control nodes, 8 worker nodes (240 cores total)



(a) Sweep: $10^7 \times$ 0s tasks

(b) Sweep: $10^7 \times$ 0.2ms tasks

(c) Fibonacci: $n = 34$, 0.2ms tasks

(d) Wavefront: 100x100, 0.2ms tasks

(e) Simulated Annealing: 10 parameters $\times$ 25 iters $\times$ 1000 $\approx$ 0.25s tasks

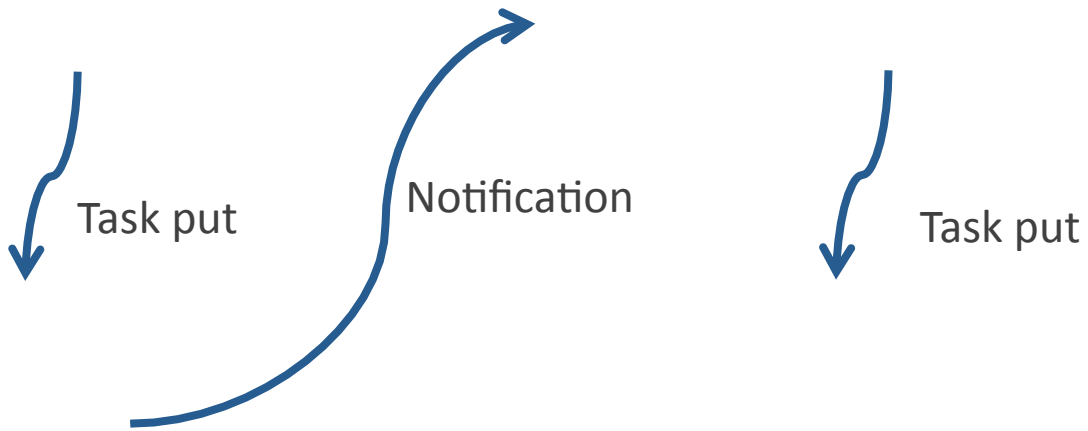
Fig. 11: Throughput at different optimization levels measured in application terms: tasks/sec, or annealing iterations/sec.

Example execution

- Code

- Engines: evaluate dataflow operations

- Workers: execute tasks

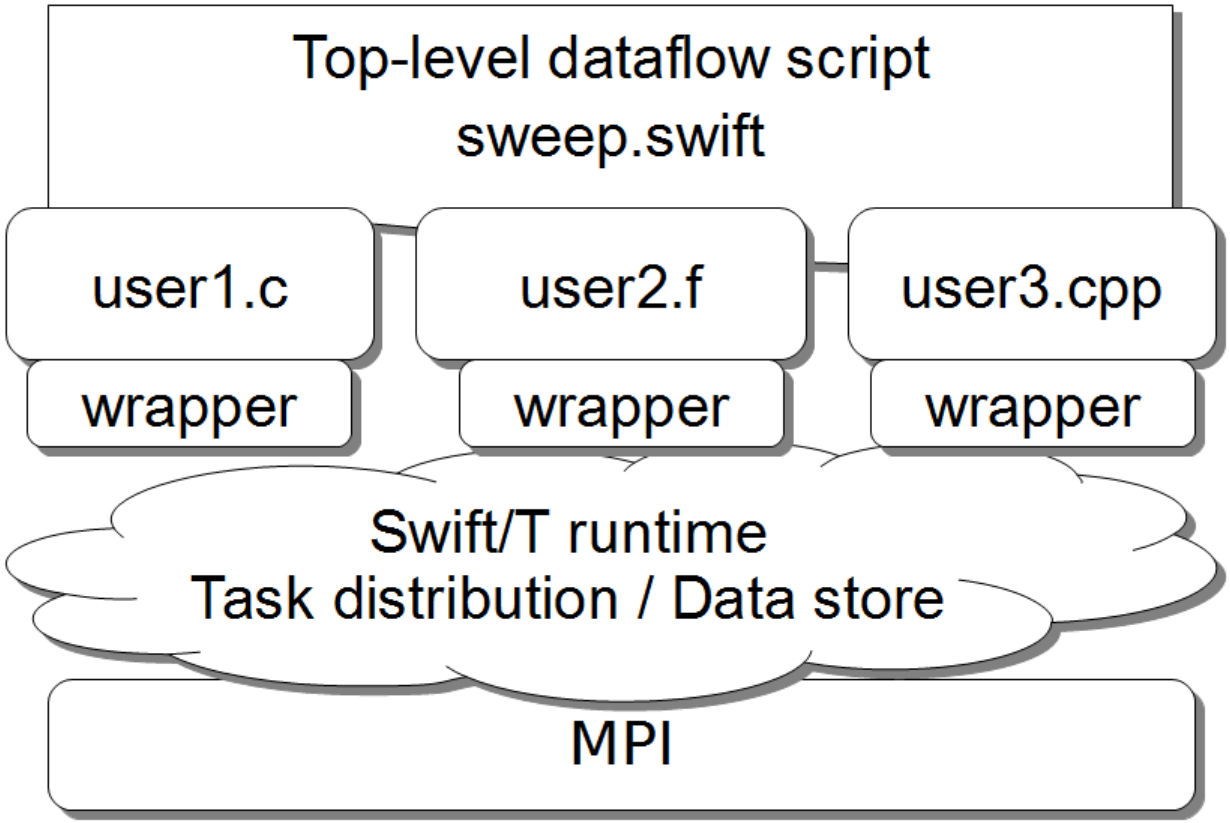


ADLB: Asynchronous Dynamic Load Balancer

- Developed previously by Lusk and Butler
- Pure MPI task distributor
- Uses client-server model with multiple servers for scalability
- Servers can share work
- Originally, supported just Put() and Get() on tasks
- We added Store(), Retrieve(), Subscribe(), etc. on data for data-dependent processing

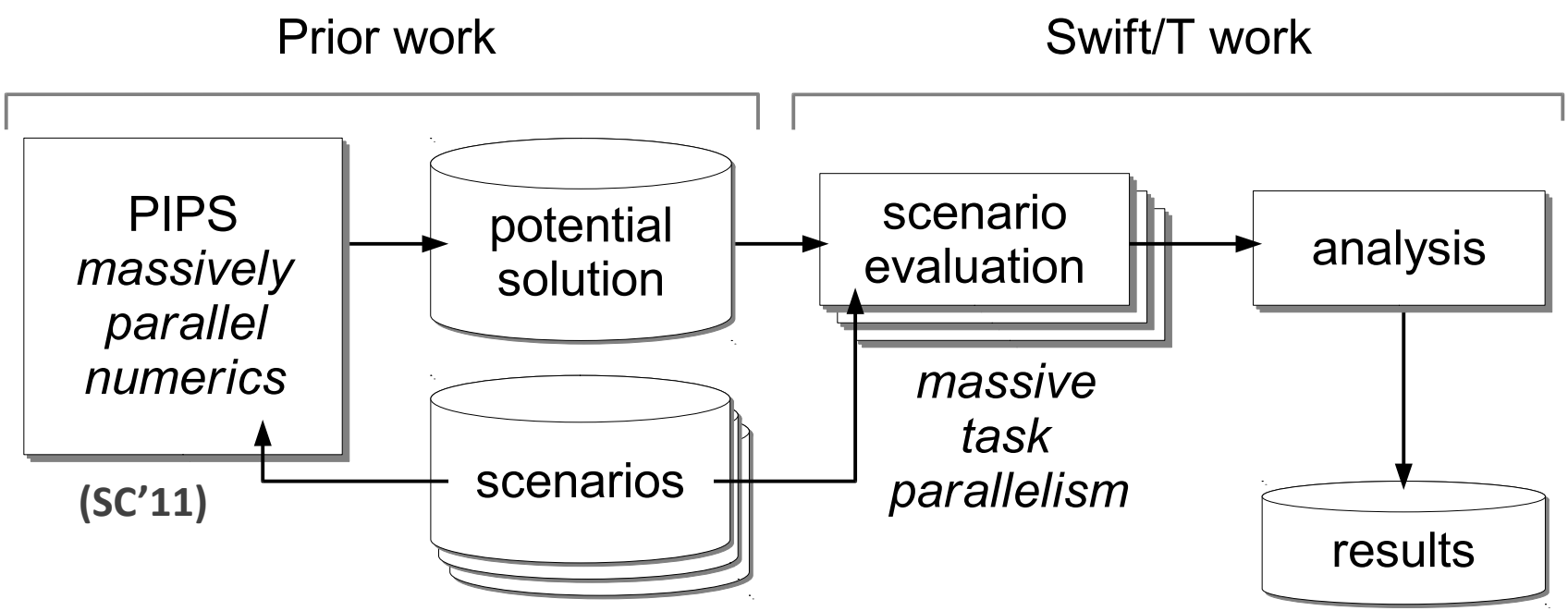
- Lusk, Pieper, and Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. SciDAC Review 17, 2010.

Supports calls to native libraries



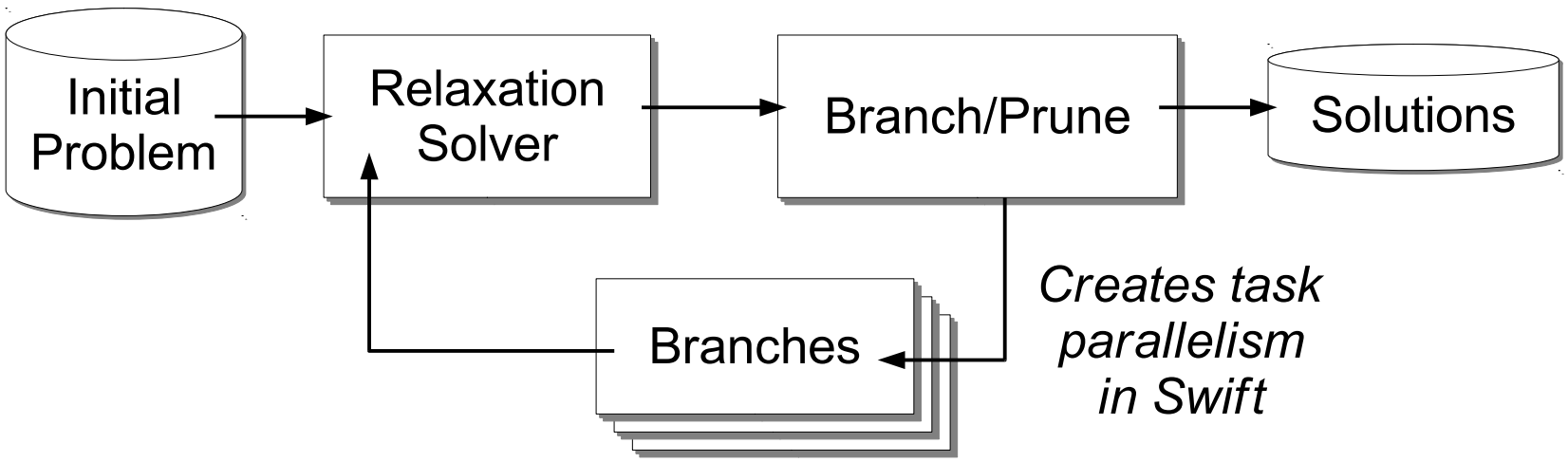
- Including MPI libraries

Application: Power Grid Modeling (PIPS)



Swift/T (and the many-task, dataflow model) *complements* existing application workflows

Application: Branch-and-Bound (Minotaur)

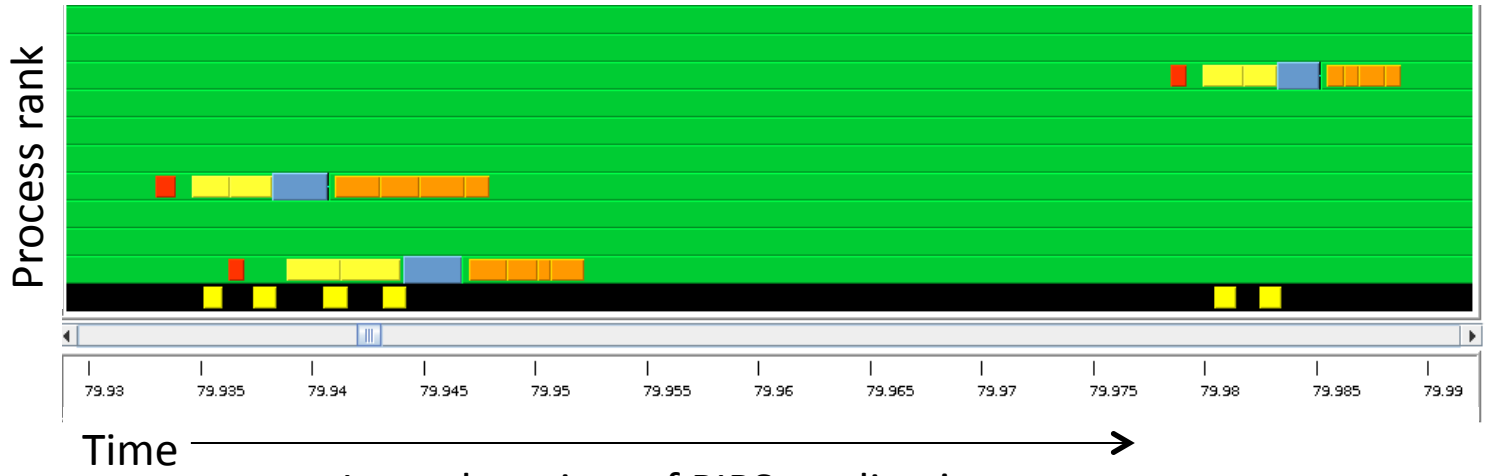


Minimize some function via recursive search, allow only for integer solutions

Builds a new, scalable application from pre-existing components

Visualization of Swift/T execution

- User writes and runs Swift script
- Notices that native application code is called with nonsensical inputs
- Turns on MPE logging – visualizes with MPE



Jumpshot view of PIPS application run

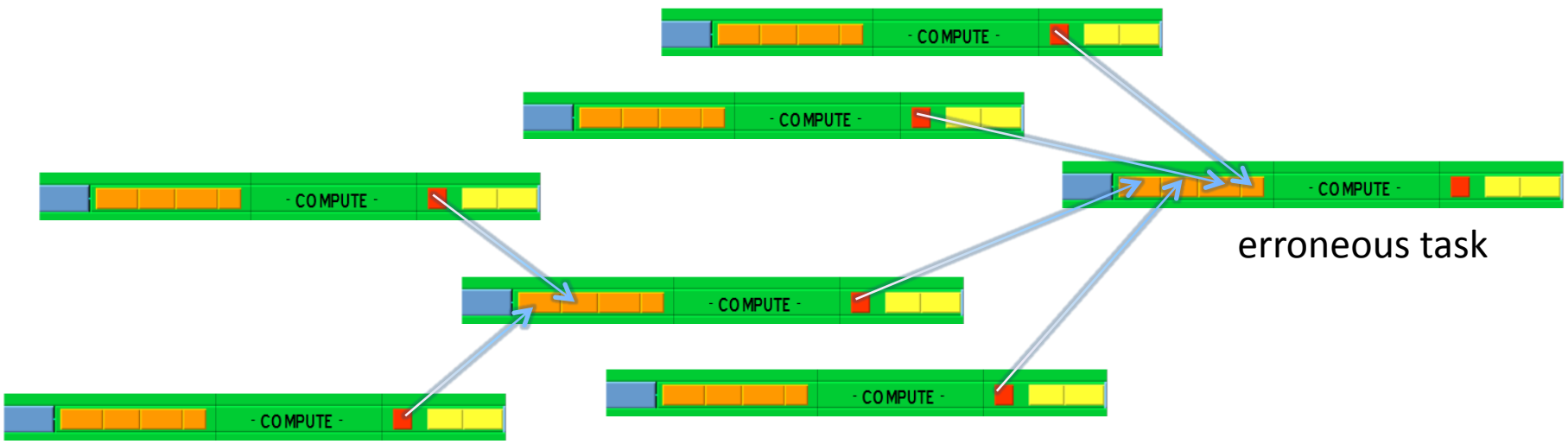
  - **PIPS task computation** **Store variable** **Notification (via control task)**
    Blue: Get next task **Retrieve variable**
    Server process (handling of control task is highlighted in yellow)

- Color cluster is task transition: 

- Simpler than visualizing messaging pattern (which is not the user's code!)
- Represents Von Neumann computing model – load, compute, store

Debugging Swift/T execution

- Starting from GUI, user can identify erroneous task
  - Uses time and rank coordinates from task metadata
- Can identify variables used as task inputs
- Can trace provenance of those variables back in reverse dataflow



Aha! Found script defect. ← ← ← (searching backwards)

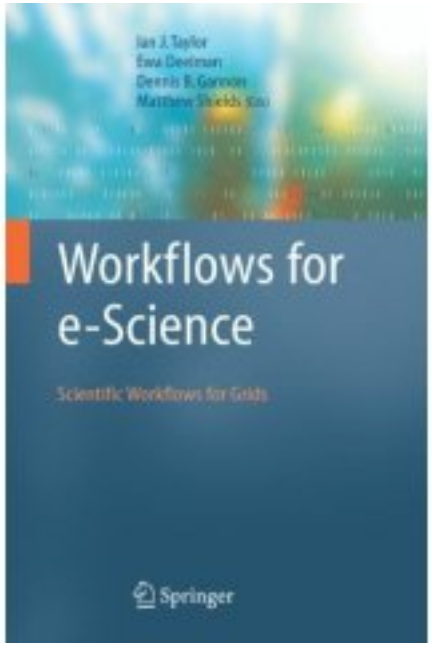
- Swift is a parallel scripting system for grids, clouds and clusters
  - for loosely-coupled applications - application and utility programs linked by exchanging files
- Swift is easy to write: simple high-level C-like functional language
  - Small Swift scripts can do large-scale work
- Swift is easy 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! http://swift-lang.org (Swift/K) and www.mcs.anl.gov/exm (Swift/T)

Summary: Challenges of workflow at extreme scale

- Inter-resource coordination
- Hybrid programming tools
- The challenges of data motion
  - Data management strategies and system envelopes
- The challenges of task scheduling and dispatch
  - Task rates and task distribution
  - Resource utilization vs. time to solution
- Workflow expression and separation of concerns
- Provenance: tracking what was done

Workflow references

- Workflows for e-Science Book
- VderA patterns
- Pegasus patterns
- Paper on characterization (Lavana, Gannon et al)
- Bibliography … ???



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.

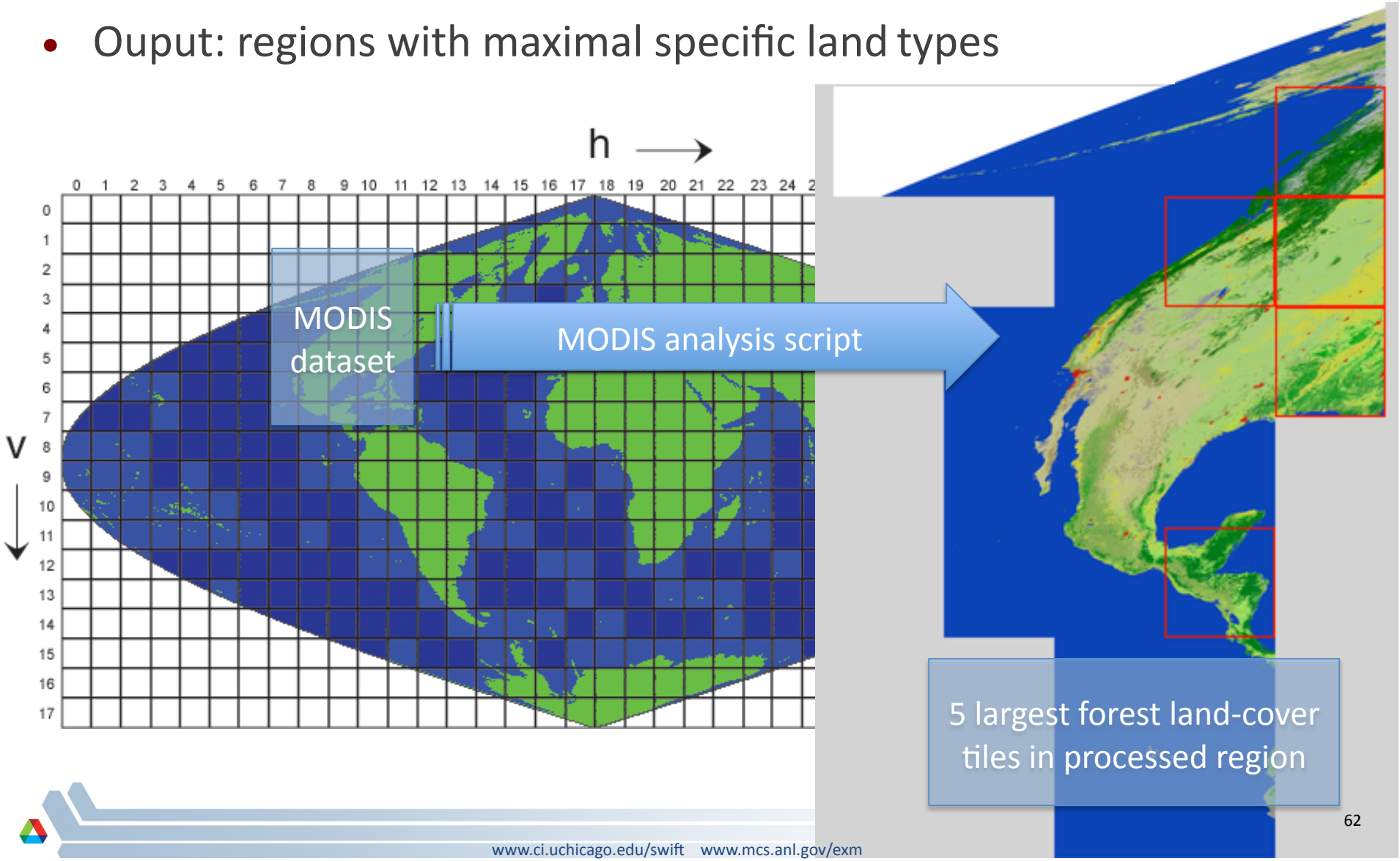
Acknowledgments

- Swift is supported in part by NSF grants OCI-1148443 and PHY-636265, NIH DC08638, and the UChicago SCI Program
- ExM is supported by the DOE Office of Science, ASCR Division
- Structure prediction supported in part by NIH
- The Swift team:
  - Tim Armstrong, Ian Foster, Mihael Hategan, Dan Katz, David Kelly, Ketan Maheshwari, Justin Wozniak, Mike Wilde, Justin Wozniak, Zhao Zhang
- Swift/T:
  - Justin Wozniak and Tim Armstrong, with Yadu Nand and Scott Krieder
- GeMTC (IIT):
  - Ioan Raicu, Scott Krieder, Ben Grimmer
- ExM:
  - Tim Armstrong, Ian Foster, Rusty Lusk, Ketan Maheshwari, Todd Munson, Matei Ripeanu, Sameer Al-Kiswani, Hao, Mike Wilde
- Java CoG Kit used by Swift developed by:
  - Mihael Hategan, Gregor Von Laszewski, and many collaborators
- Scientific application collaborators and usage described in this talk:
  - U. Chicago Open Protein Simulator Group (Karl Freed, Tobin Sosnick, Glen Hocky, Joe Debartolo, Aashish Adhikari, Mark Parisien)
  - U.Chicago Radiology and Human Neuroscience Lab, (Dr. S. Small, Sarah Kenny, Uri Hasson)
  - RDCEP / CIM-EARTH: Joshua Elliott, David Kelly
  - ParVis and FOAM: Rob Jacob, Sheri Mickelson (Argonne); John Dennis, Matthew Woitaszek (NCAR)
  - UColumbia Chemistry, David Reichman, Glen Hocky
  - Argonne Power Grid Simulator, V. Zavala, K. Maheshwari, M. Hereld

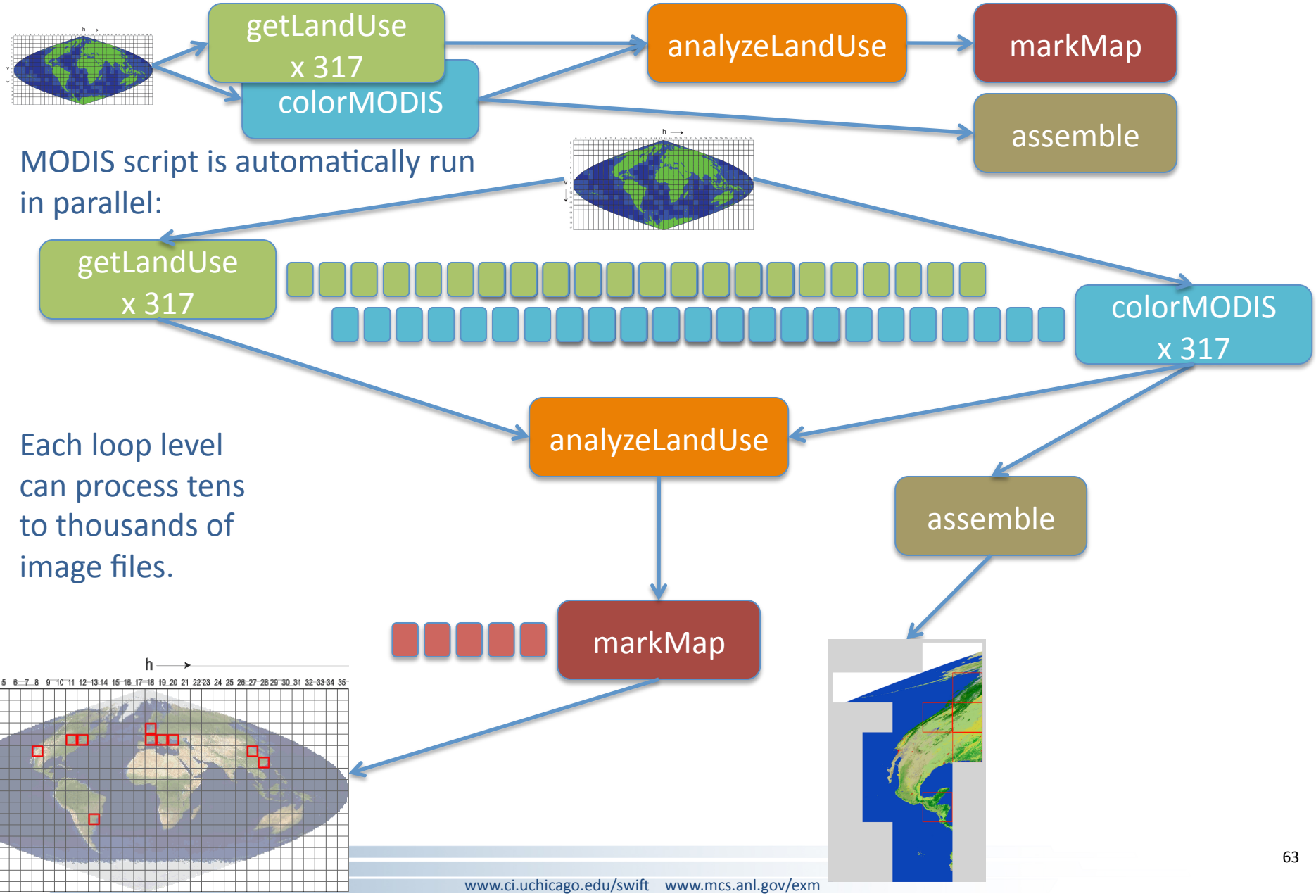
Exercise views and supplemental slides

Exercise - MODIS satellite image processing

- Input: tiles of earth land cover (forest, ice, water, urban, etc)
  - Ouput: regions with maximal specific land types



Goal: Run MODIS processing pipeline in cloud

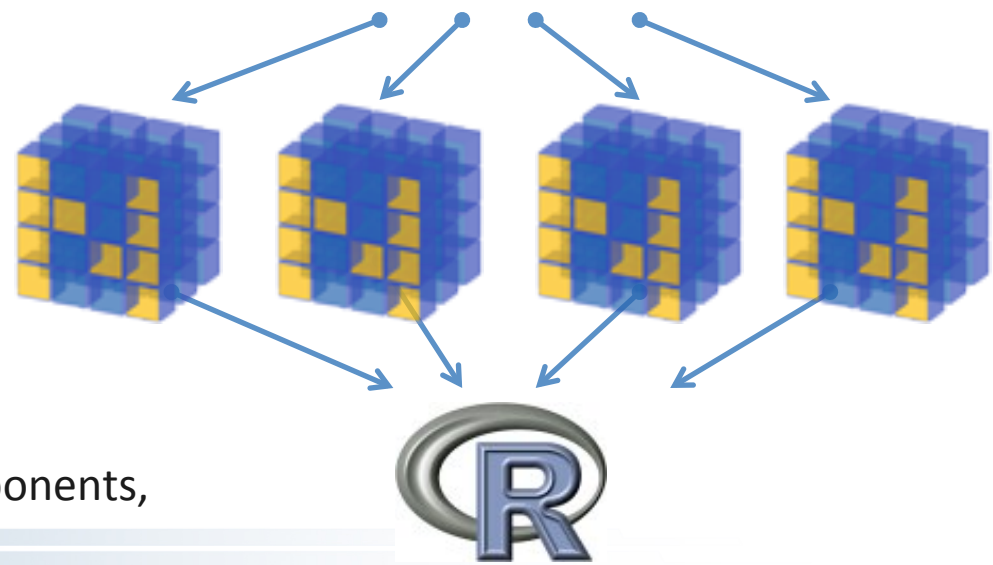


Swift/T example: Part 11

- Overview: Find biggest parallelepiped volume via Python and R
  - Construct several matrices according to simple arithmetic
  - Compute determinants in parallel in Python (via Numpy)
  - Fix maximal determinant in R (reduction step)
  - Python reference code is included (dets.py == dets.swift)

- Construct matrices (in Swift arithmetic)
  - Matrices stored in distributed global store
- Determinant (Numpy/Python)
  - Cf. numpy.swift

- Find maximum (in R)

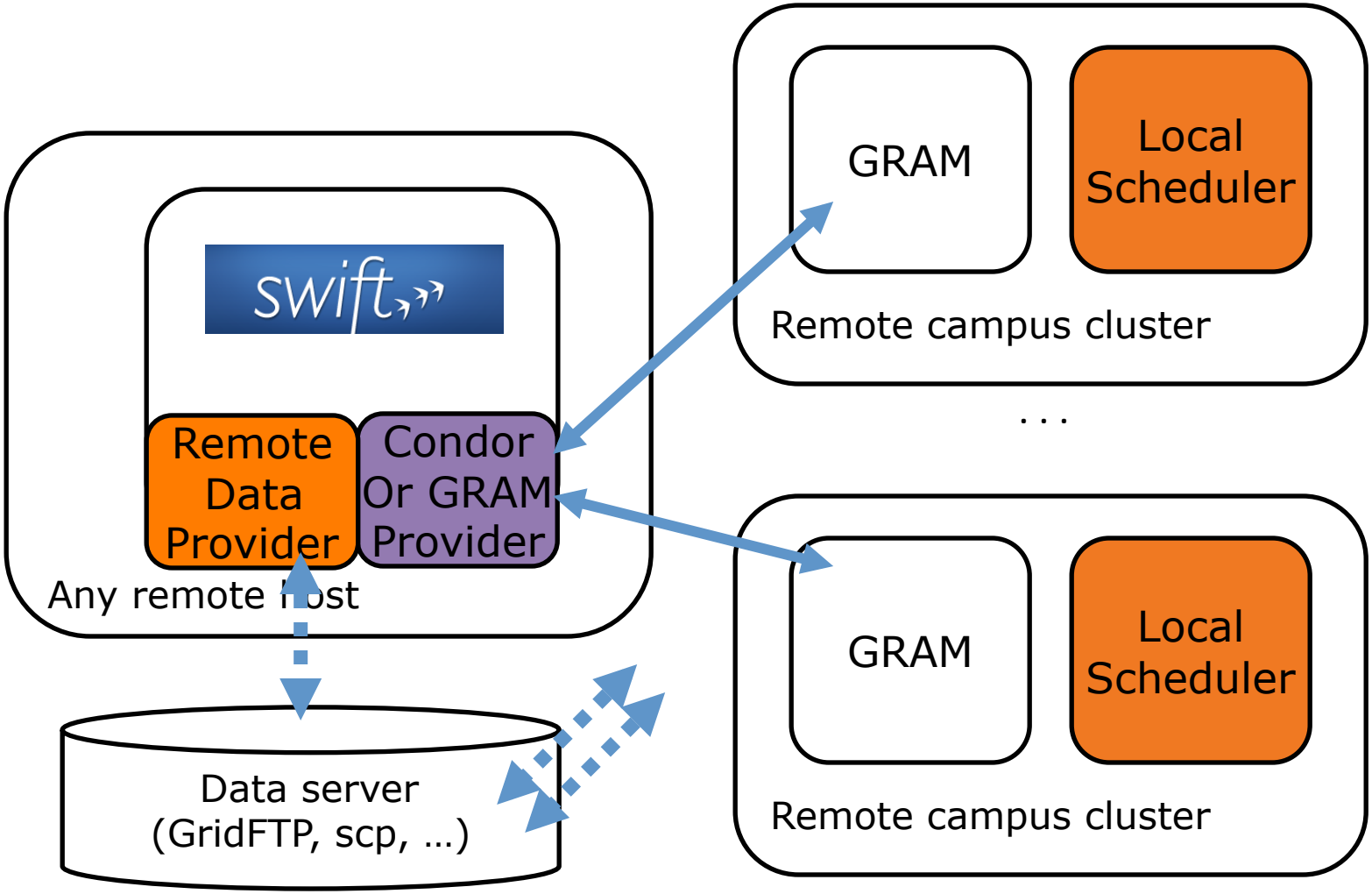


- Could call to C, C++, Fortran, instead
- Normally would call to application components, not numerical libraries

MODIS script in Swift: main data flow

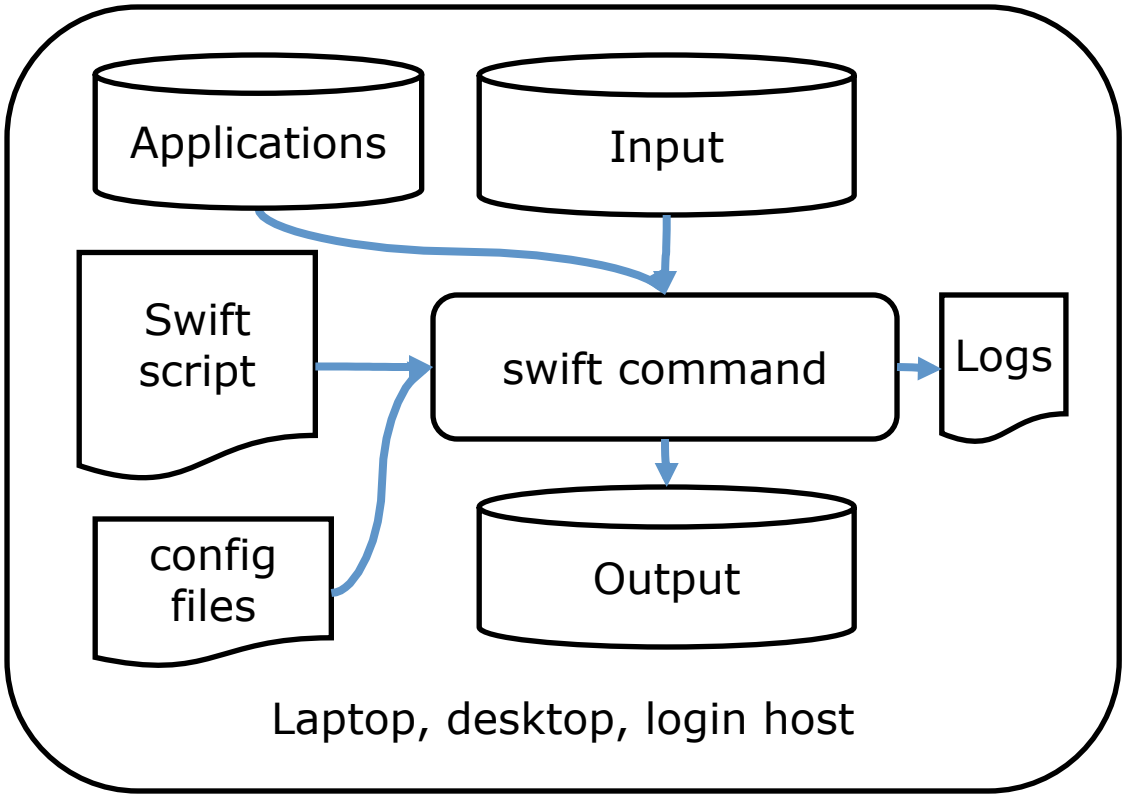
```
foreach g,i in geos {
    land[i] = getLandUse(g,1);
}
(topSelected, selectedTiles) =
    analyzeLandUse(land, landType, nSelect);

foreach g, i in geos {
  colorImage[i] = colorMODIS(g);
}
gridMap = markMap(topSelected);
montage =
   assemble(selectedTiles,colorImage,webDir);
```

Swift can send work to multiple sites



Simple campus Swift usage: running locally on a cluster login host

User first tests a new script on a local login host



Swift script is location-independent – debug locally then run distributed

MODIS script excerpt used in this demo

```
$ cat modis.swift

type imagefile;
type landuse;
type perlscript;

perlscript getlanduse_pl <"getlanduse.pl">;

app (landuse output) getLandUse (imagefile input, perlscript ps)
{
  perl @ps @filename(input) stdout=@filename(output);
}

# Constants and command line arguments
string MODISdir  = @arg("modisdir", "../data/modis/2002");

# Input Dataset
imagefile geos[] <filesys_mapper; location=MODISdir, suffix=".rgb">;

# Compute the land use summary of each MODIS tile
landuse land[] <structured_regexp_mapper; source=geos, match="(h..v..)",
          transform=@strcat("landuse/\\1.landuse.byfreq")>;

foreach g,i in geos {
   land[i] = getLandUse(g, getlanduse_pl);
}
```

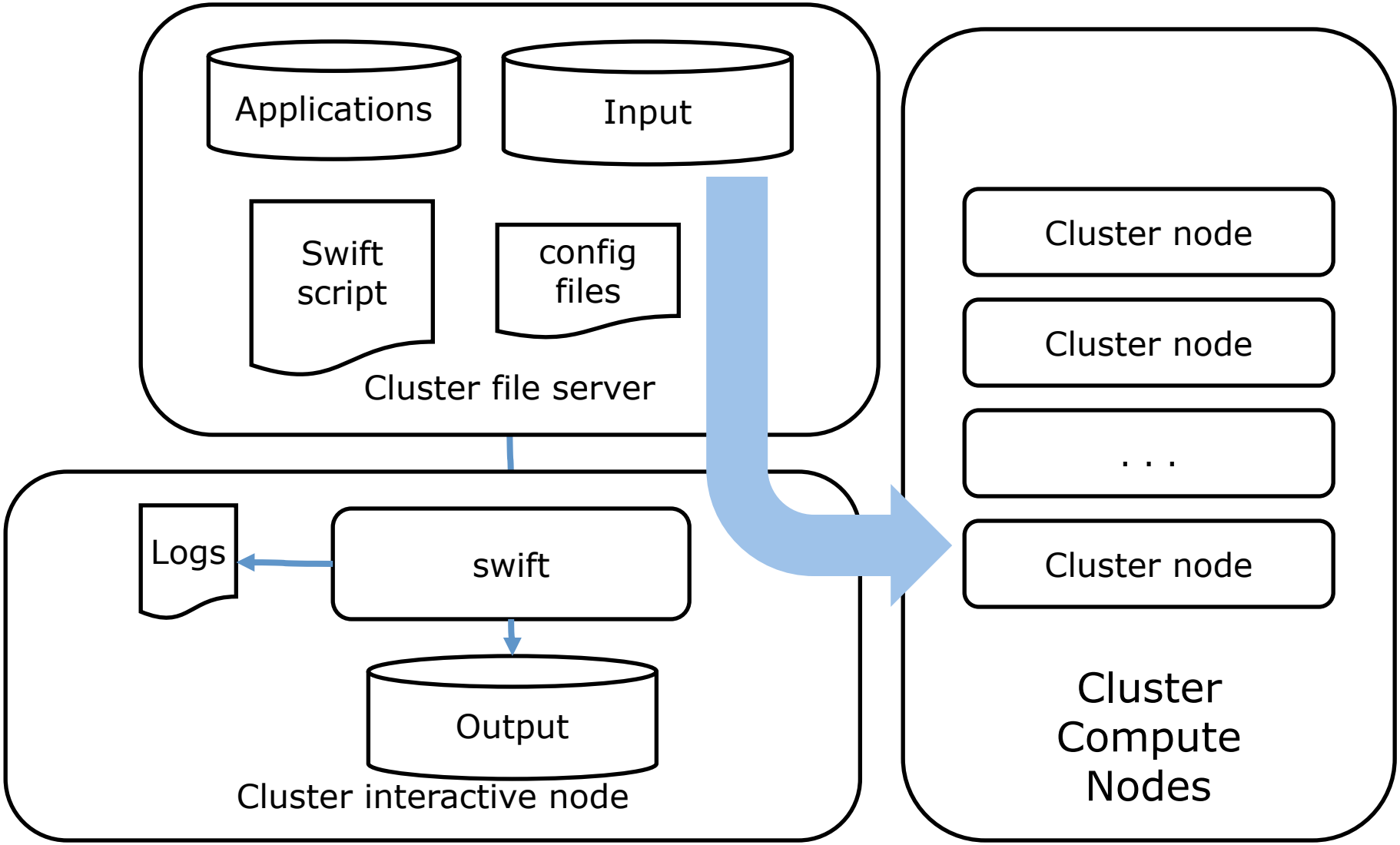
User's Perl app is passed as data

Input dataset is a script parameter

Output filenames are based on inputs

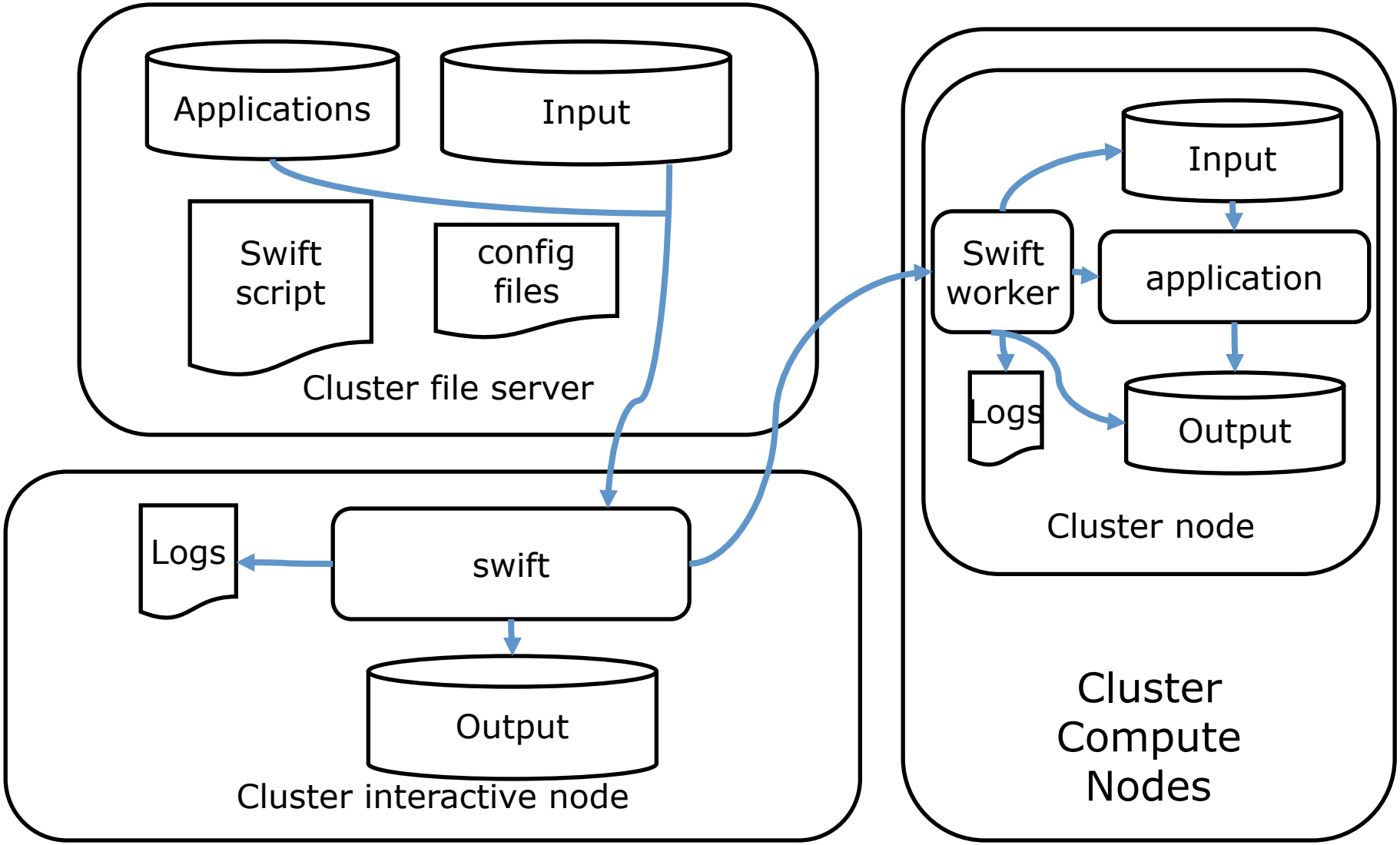
Iteration over the dataset is implicitly parallel

User runs on a campus or department cluster



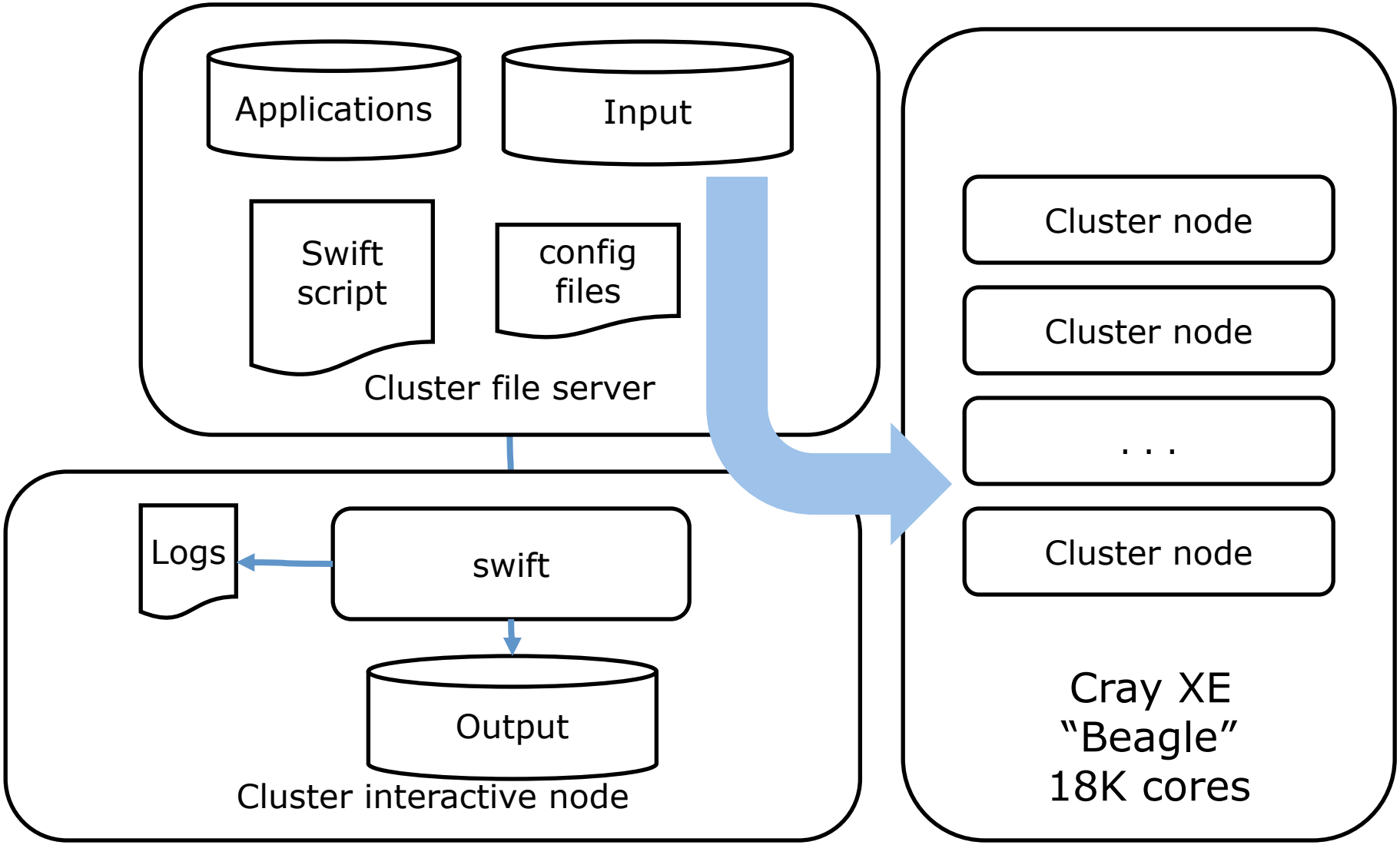
Single-node script scales easily to local cluster

User runs on a campus cluster: what's inside



Multiple data streams of data moved from client to worker local FS

Campus or XSEDE supercomputer access is same



Same script runs unchanged between campus research cluster and Cray XE systems

example

```
$ swift -sites.file sites.xml -tc.file tc.data -config beagle-ssh.cf modis02.swift \
        -modisdir=/home/wilde/osgdemo/modis/svn/data/modis/2002/

Swift trunk swift-r6362 cog-r3637 (cog modified locally)

RunID: 20130311-0159-qayhuq86
Progress:  time: Mon, 11 Mar 2013 01:59:55 +0000
Progress:  time: Mon, 11 Mar 2013 02:00:07 +0000  Selecting site:269  Submitting:47  Submitted:1
Progress:  time: Mon, 11 Mar 2013 02:00:14 +0000  Selecting site:269  Stage in:1  Submitted:47
Progress:  time: Mon, 11 Mar 2013 02:00:15 +0000  Selecting site:269  Stage in:25  Submitted:23
Progress:  time: Mon, 11 Mar 2013 02:00:18 +0000  Selecting site:269  Stage in:47  Active:1
Progress:  time: Mon, 11 Mar 2013 02:00:19 +0000  Selecting site:269  Stage in:29  Active:15  Stage out:4
Progress:  time: Mon, 11 Mar 2013 02:00:20 +0000  Selecting site:239  Stage in:24  Submitting:6  Stage out:17  Finished successfully:31
Progress:  time: Mon, 11 Mar 2013 02:00:21 +0000  Selecting site:221  Stage in:36  Submitting:11  Stage out:1  Finished successfully:48
Progress:  time: Mon, 11 Mar 2013 02:00:22 +0000  Selecting site:221  Stage in:44  Active:1  Stage out:2  Finished successfully:49
Progress:  time: Mon, 11 Mar 2013 02:00:23 +0000  Selecting site:218  Stage in:43  Submitting:3  Stage out:2  Finished successfully:51
…
Progress:  time: Mon, 11 Mar 2013 02:00:43 +0000  Selecting site:30  Stage in:41  Submitted:3  Active:3  Finished successfully:240
Progress:  time: Mon, 11 Mar 2013 02:00:44 +0000  Selecting site:29  Stage in:36  Submitting:1  Active:4  Stage out:7  Finished successfully:240
Progress:  time: Mon, 11 Mar 2013 02:00:45 +0000  Selecting site:23  Stage in:28  Submitting:6  Active:1  Stage out:12  Finished successfully:247
Progress:  time: Mon, 11 Mar 2013 02:00:46 +0000  Selecting site:9  Stage in:39  Submitting:7  Active:1  Stage out:1  Finished successfully:260
Progress:  time: Mon, 11 Mar 2013 02:00:47 +0000  Selecting site:7  Stage in:21  Submitting:2  Active:5  Stage out:20  Finished successfully:262
Progress:  time: Mon, 11 Mar 2013 02:00:48 +0000  Stage in:28  Submitted:1  Stage out:1  Finished successfully:287
Progress:  time: Mon, 11 Mar 2013 02:00:49 +0000  Stage in:15  Active:4  Stage out:7  Finished successfully:291
Final status: Mon, 11 Mar 2013 02:00:50 +0000  Finished successfully:317

real        0m57.478s
user        0m32.923s
sys         0m1.248s
$
```

Simple script runs 300+ apps in under a minute

example

```
$ cat beagle-ssh.cf

wrapperlog.always.transfer=true
sitedir.keep=true
execution.retries=0
status.mode=provider
use.provider.staging=true
provider.staging.pin.swiftfiles=false

#site beagle-ssh WALLTIME=00:55:00
#app perl=/usr/bin/perl
midway001$
$

$ cat sites.xml
<config>
 <pool handle="beagle">
  <execution provider="coaster" jobmanager="ssh-cl:pbs" url="login4.beagle.ci.uchicago.ed
  <profile namespace="globus" key="jobsPerNode">24</profile>
  <profile namespace="globus" key="lowOverAllocation">100</profile>
  <profile namespace="globus" key="highOverAllocation">100</profile>
  <profile namespace="globus" key="providerAttributes">pbs.aprun;pbs.mpp;depth=24</profile>
  <profile namespace="globus" key="maxtime">3600</profile>
  <profile namespace="globus" key="maxWalltime">00:55:00</profile>
  <profile namespace="globus" key="userHomeOverride">/lustre/beagle/{env.USER}/swiftwork</profile>
  <profile namespace="globus" key="slots">2</profile>
  <profile namespace="globus" key="maxnodes">1</profile>
  <profile namespace="globus" key="nodeGranularity">1</profile>
  <profile namespace="karajan" key="jobThrottle">.47</profile>
  <profile namespace="karajan" key="initialScore">10000</profile>
  <filesystem provider="local"/>
  <workdirectory>/tmp/{env.USER}/swiftwork</workdirectory>
 </pool>
</config>
$
```

Swift moves user's dataset from campus server direct to Cray compute nodes

...and passes Cray-specific PBS parameters

...but most of the site spec is the same as for the campus cluster

example

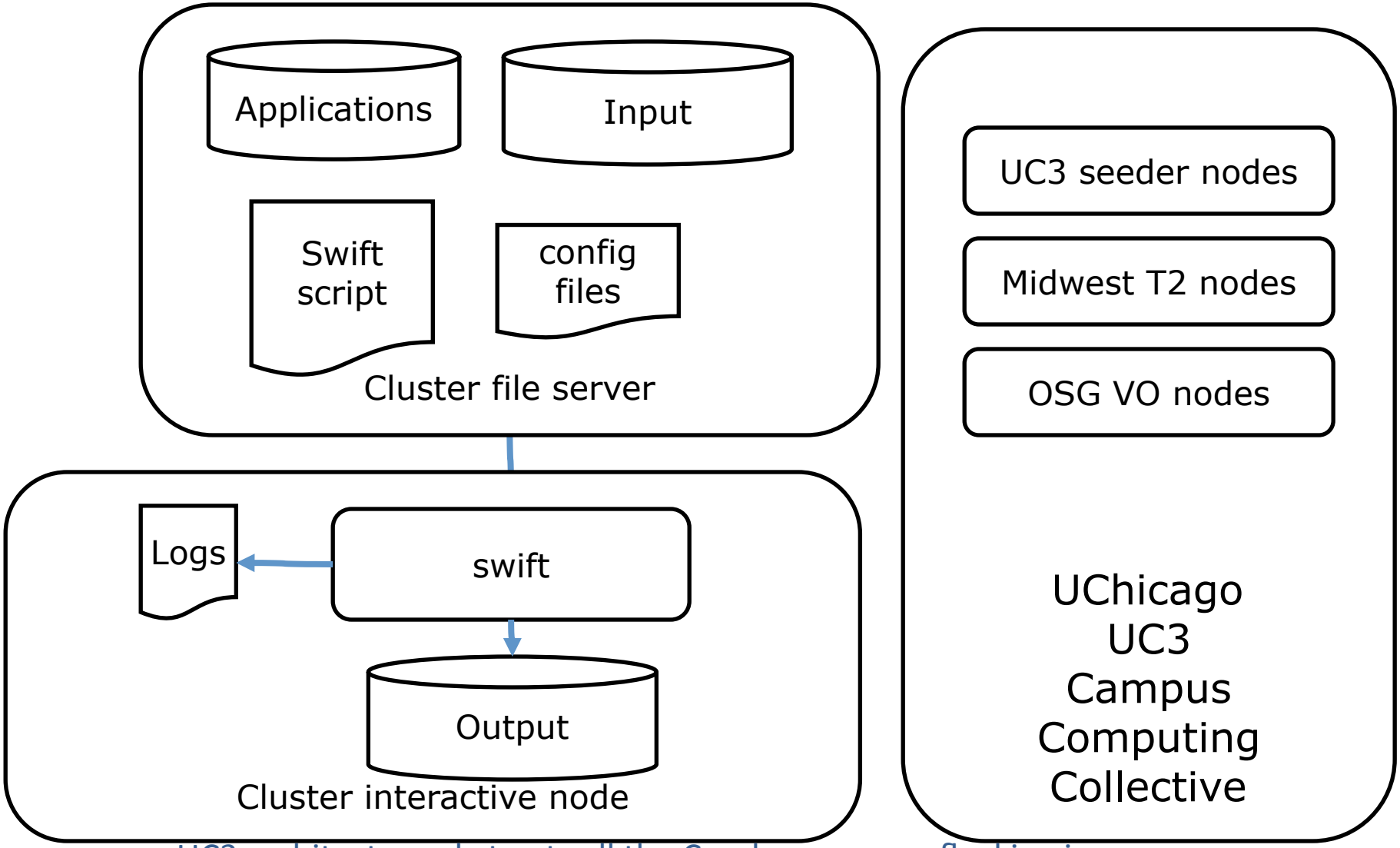
```
midway001$ ls landuse
h00v08.landuse.byfreq  h11v10.landuse.byfreq  h17v06.landuse.byfreq  h21v10.landuse.byfreq  h27v10.landuse.byfreq
h00v09.landuse.byfreq  h11v11.landuse.byfreq  h17v07.landuse.byfreq  h21v11.landuse.byfreq  h27v11.landuse.byfreq
…
h11v06.landuse.byfreq  h17v02.landuse.byfreq  h21v06.landuse.byfreq  h27v06.landuse.byfreq  h35v10.landuse.byfreq
h11v07.landuse.byfreq  h17v03.landuse.byfreq  h21v07.landuse.byfreq  h27v07.landuse.byfreq
h11v08.landuse.byfreq  h17v04.landuse.byfreq  h21v08.landuse.byfreq  h27v08.landuse.byfreq
h11v09.landuse.byfreq  h17v05.landuse.byfreq  h21v09.landuse.byfreq  h27v09.landuse.byfreq
midway001$
```

Input is MODIS satellite raster image dataset

```
midway001$ cat landuse/h03v07.landuse.byfreq
211094 0 00
5348 1 01
4376 2 02
3236 3 03
3196 4 04
1242 5 05
731 6 06
405 7 07
292 8 08
225 9 09
83 10 0a
61 11 0b
43 12 0c
39 13 0d
25 14 0e
4 15 0f
midway001$
```

Output is histogram of land use codes

UChicago campus "collective" adds OSG resources



UC3 architecture abstracts all the Condor resource flocking issues;
Swift accesses local, MWT2, and OSG as a unified Condor facility using campus user identity

example

```
midway001$ pwd
/home/wilde/osgdemo/modis/svn/run051
midway001$ cat sites.xml
<config>
 <pool handle="uc3">
  <execution provider="coaster" url="uc3-sub.uchicago.edu" jobmanager="ssh-cl:condor"/>
  <profile namespace="karajan" key="jobThrottle">3.99</profile>
  <profile namespace="karajan" key="initialScore">10000</profile>
  <profile namespace="globus"  key="jobsPerNode">1</profile>
  <profile namespace="globus"  key="maxWalltime">3600</profile>
  <profile namespace="globus"  key="highOverAllocation">100</profile>
  <profile namespace="globus"  key="lowOverAllocation">100</profile>
  <profile namespace="globus"  key="slots">400</profile>
  <profile namespace="globus"  key="maxNodes">1</profile>
  <profile namespace="globus"  key="nodeGranularity">1</profile>
  <profile namespace="globus"  key="condor.+AccountingGroup">"group_friends.{env.USER}"</profile>
  <profile namespace="globus"  key="jobType">nonshared</profile>
  <filesystem provider="local" url="none" />
  <workdirectory>.</workdirectory>
 </pool>

</config>
midway001$
```

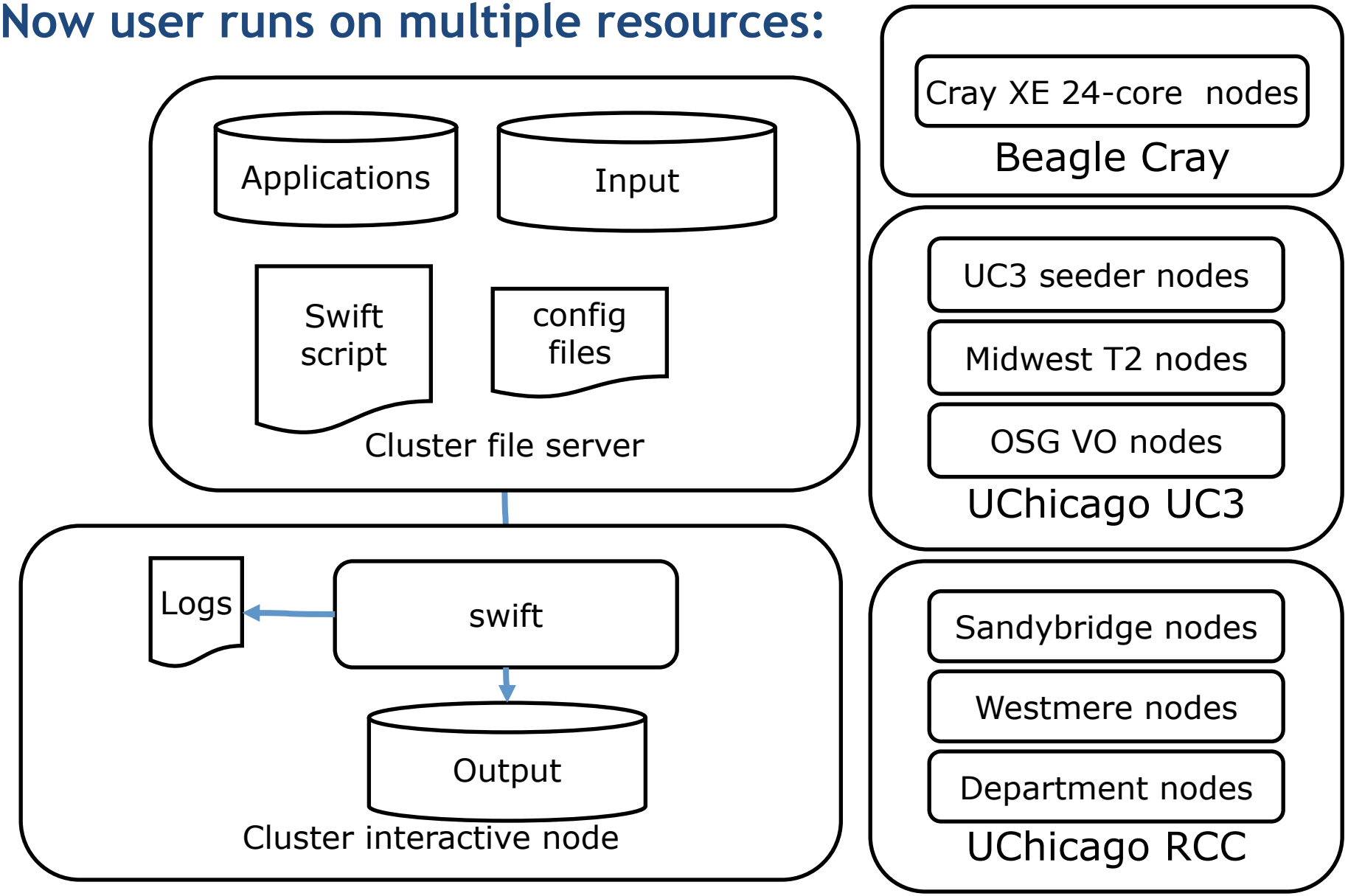
Swift forwards Condor parameters

 # Example of running 1,000 MODIS jobs on just the UC3 collective: local UC3 resources full but work routed to Midwest Tier 2 and OSG

```
$ showsites

midway   0
beagle   0
uc3      0
mwt2     256
OSG      744
Total    1000
```

When local UC3 “seeder” resource full, UC3 flocks to other resources

Now user runs on multiple resources:



Same script runs on broad range of resources; separate throttles can be set for each site.

example

```
<config>

 <pool handle="uc3">
  <execution provider="coaster" url="uc3-sub.uchicago.edu" jobmanager="ssh-cl:condor"/>
  <profile namespace="karajan" key="jobThrottle">10.00</profile>
  <profile namespace="karajan" key="initialScore">10000</profile>
  <profile namespace="globus"  key="jobsPerNode">1</profile>
  …
  <profile namespace="globus"  key="jobType">nonshared</profile>
  <!-- <profile namespace="globus"  key="condor.+Requirements">isUndefined(GLIDECLIENT_Name) == FALSE</profile> -->
  <workdirectory>.</workdirectory>
 </pool>

 <pool handle="beagle">
  <execution provider="coaster" jobmanager="ssh-cl:pbs" url="login4.beagle.ci.uchicago.edu"/>
  <profile namespace="globus" key="jobsPerNode">24</profile>
  <profile namespace="globus" key="lowOverAllocation">100</profile>
  <profile namespace="globus" key="highOverAllocation">100</profile>
  <profile namespace="globus" key="providerAttributes">pbs.aprun;pbs.mpp;depth=24;pbs.resource_list=advres=wilde.1768</profile>
  …
   <workdirectory>/tmp/{env.USER}/swiftwork</workdirectory>
 </pool>

 <pool handle="sandyb">
  <execution provider="coaster" jobmanager="local:slurm"/>
…
  <workdirectory>/tmp/{env.USER}</workdirectory>
 </pool>

 <pool handle="westmere">
  <execution provider="coaster" jobmanager="local:slurm"/>
  …
  <workdirectory>/tmp/{env.USER}</workdirectory>
 </pool>

</config>
```

Multiple site definitions, managed by support staff

User can specify custom parameters

App list selects where app() run

```
$ cat tc
uc3        perl     /usr/bin/perl null null null
beagle     perl /usr/bin/perl null null null
#sandyb    perl /usr/bin/perl null null null
westmere perl /usr/bin/perl null null null
```

Swift's location-independent scripting lets the user focus on science

- Example of running 3,000 jobs to 3 hosts including the UC3 campus collective:

```
$ ./showsites

midway    289
beagle    1070
uc3       1011
mwt2      295
OSG       335
Total     3000
```

- The user started on a basic login host processing 10 files and moved up to a 3,000 file dataset, changing only the dataset name and a site-specification list to get to the resources above
- Expanded the scope of their computations from one node to hundreds or thousands of cores
- **User didn't need to look at what sites were busy, or adjust arcane scripts, to get to these resources.**