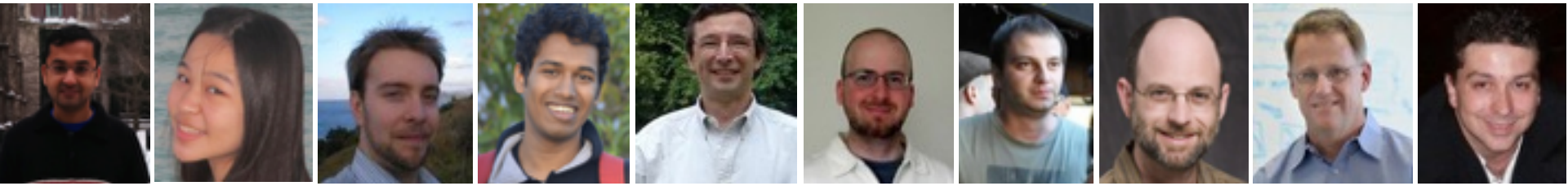
Swift/T: Dataflow Composition of Tcl Scripts for Petascale Computing

Justin M Wozniak

Argonne National Laboratory and University of Chicago

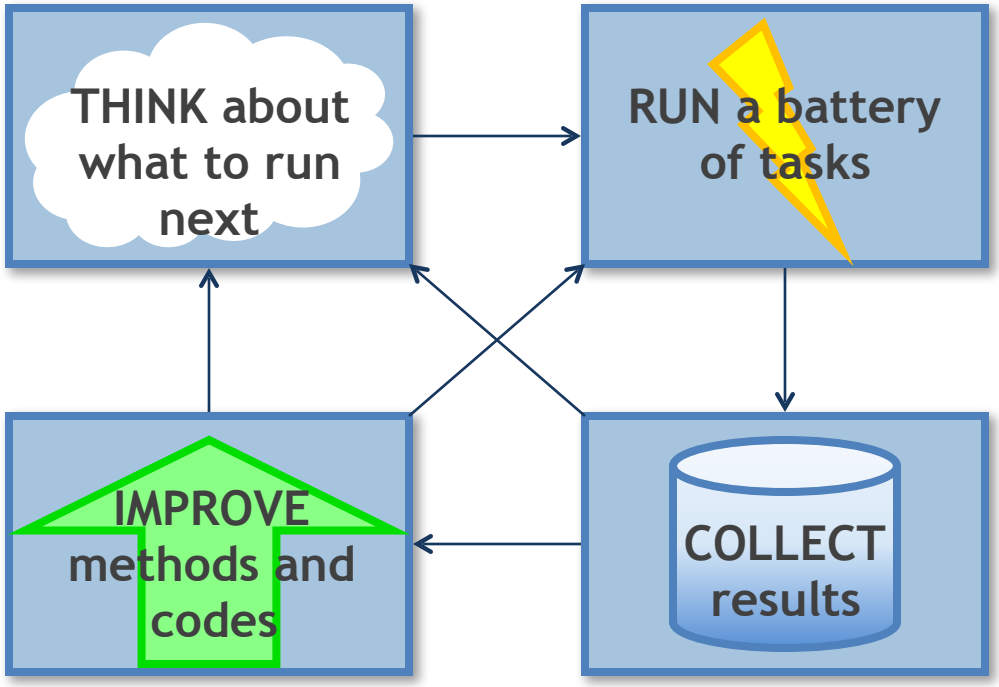
http://swift-lang.org/Swift-T wozniak@mcs.anl.gov



Big picture: solutions for scientific scripting

SCIENTIFIC WORKFLOWS

The Scientific Computing Campaign

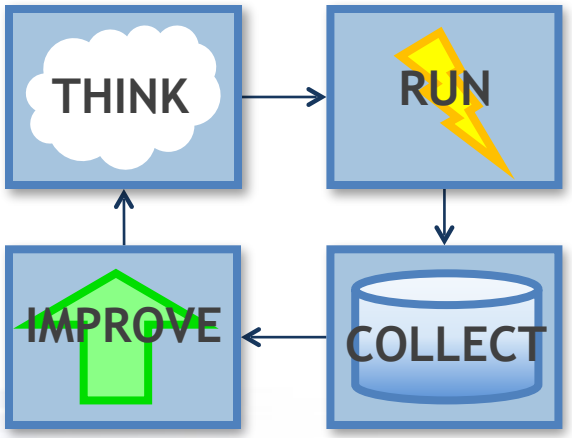


- The Swift system addresses most of these components
- Primarily a language, with a supporting runtime and toolkit

Goals of the Swift language

Swift was designed to handle many aspects of the computing campaign

- Ability to integrate many application components into a new workflow application
- Data structures for complex data organization
- Portability- separate site-specific configuration from application logic
- Logging, provenance, and plotting features



Goal: Programmability for large scale computing

- **Approach: Many-task computing**: Higher-level applications composed of many run-to-completion tasks:

  input→compute→output

- **Programmability**
  - Large number of applications have this natural structure at upper levels: Parameter studies, ensembles, Monte Carlo, branch-and-bound, stochastic programming, UQ
  - Easy way to exploit hardware concurrency

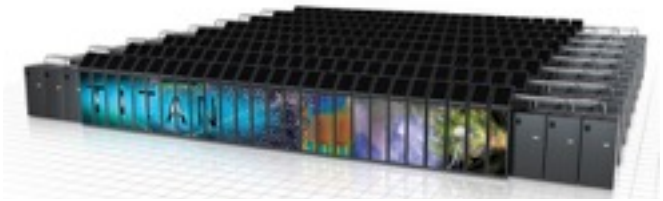
- **Experiment management**
  - Address workflow-scale issues: data transfer, application invocation

The Race to Exascale

- The exaflop computer: a quintillion (10^{18}) floating point operations per second
- Expected to have massive (billion-way) concurrency
- Significant issues must be overcome
  - Fault-tolerance
  - I/O
  - Heat and power efficiency
  - Programmability!
- Can scripting systems like Tcl help?
  - I think so!

TOP500 leaderboard

#1 *Tianhe-2*: 33 PF, 18 MW (China)



#2 *Titan*: 20 PF, 8 MW (Oak Ridge)

#5 *Mira*: 8.5 PF, 4 MW (Argonne)

= 2.5 MW

Outline

- Introduction to Swift/T
  - Introduction to MPI
  - Introduction to ADLB
  - Introduction to Turbine, the Swift/T runtime
- Use of Tcl in Swift/T
- Interesting Swift/T features
- Applications
- Performance

High-performance dataflow for compositional programming

SWIFT/T OVERVIEW

Swift programming model: all progress driven by concurrent dataflow

```
(int r) myproc (int i, int j)
{
    int x = A(i);
    int y = B(j);
    r = x + y;
}
```

- `A()` and `B()` implemented in native code
- `A()` and `B()` run in concurrently in different processes
- `r` is computed when they are both done

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

Swift programming model

- Data types

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

- Mapped data types

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

- Structured data

```
image  A[]<array_mapper…>;
type protein {
   file pdb;
   file docking_pocket;
}
bag<blob>[] B;
```

- Conventional expressions

```
if (x == 3) {
    y = x+2;
    s = sprintf("y: %i", y);
}
```

- Parallel loops

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

- Implicit data flow

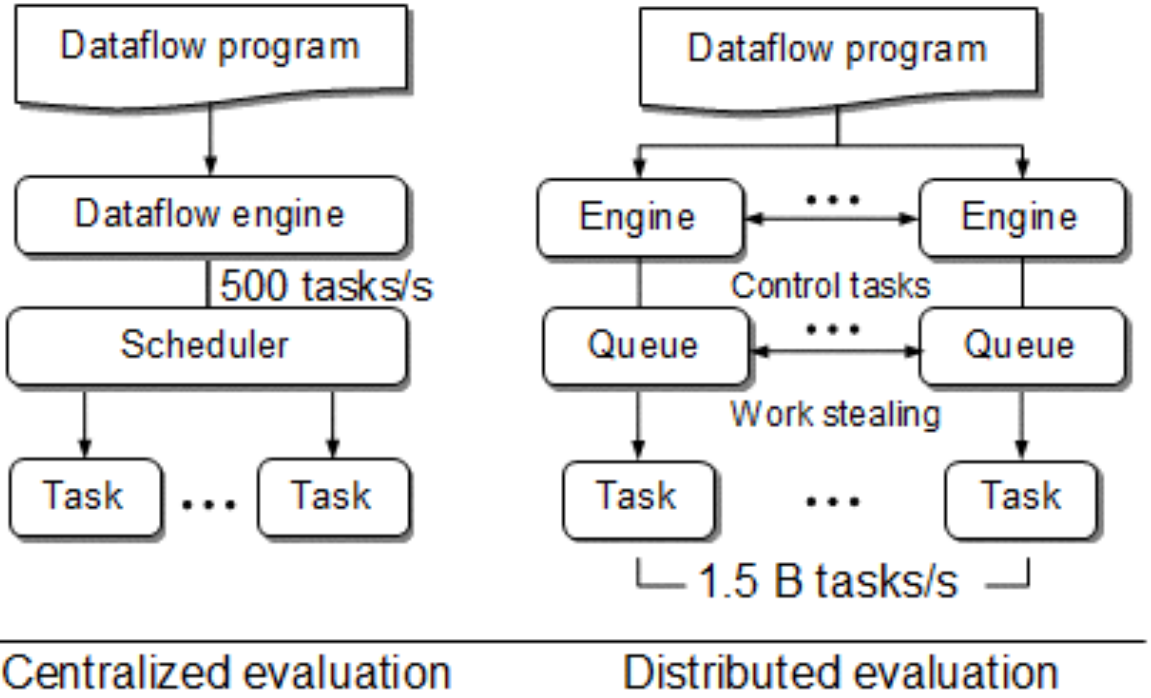
```
merge(analyze(B[0], B[1]),
      analyze(B[2], B[3]));
```

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

Swift/T: Swift for high-performance computing

Had this:
(Swift/K)

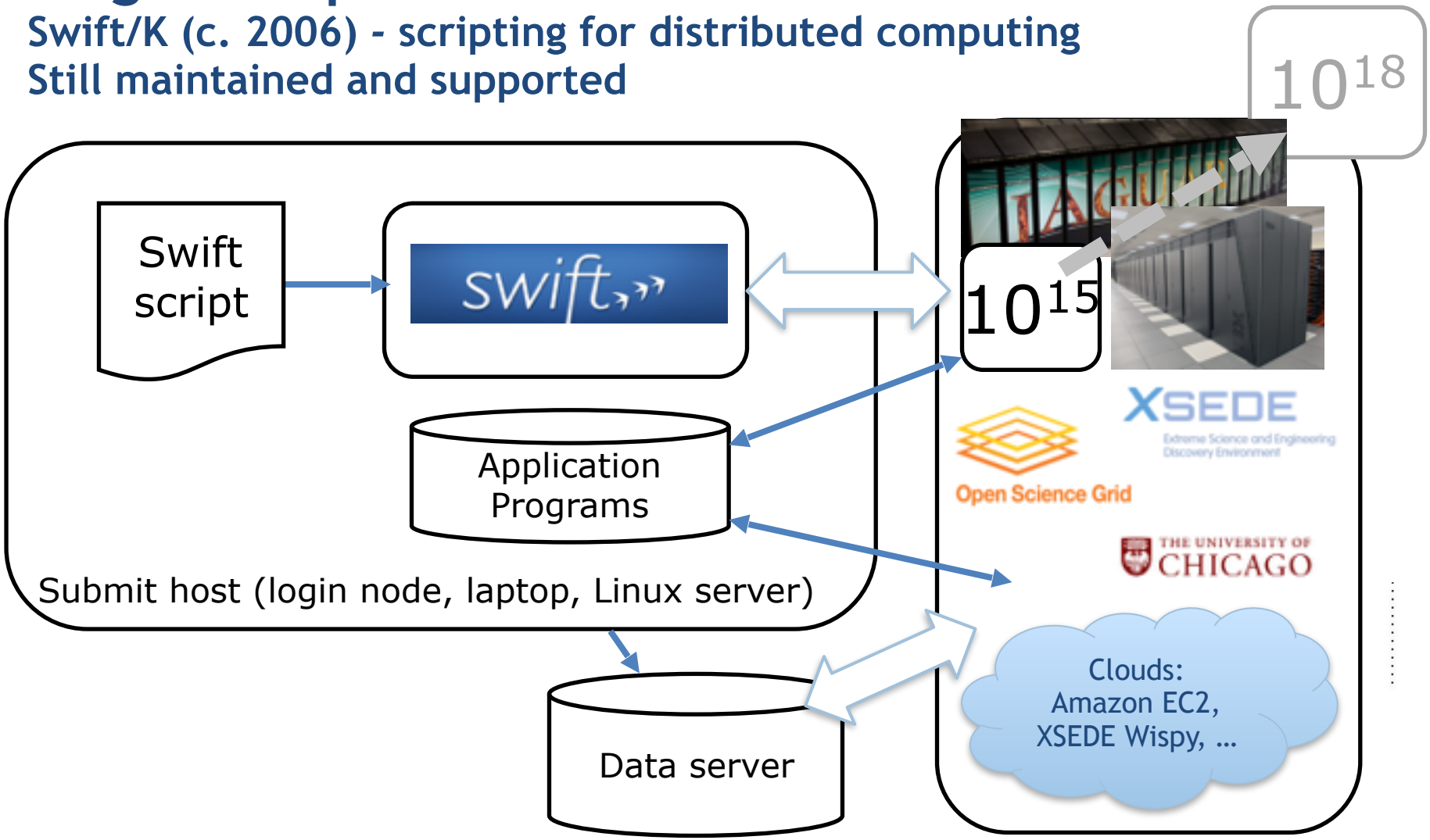
For extreme scale,
we need this:
(Swift/T)



- Wozniak et al. Swift/T: Scalable data flow programming for distributed-memory task-parallel applications . Proc. CCGrid, 2013.

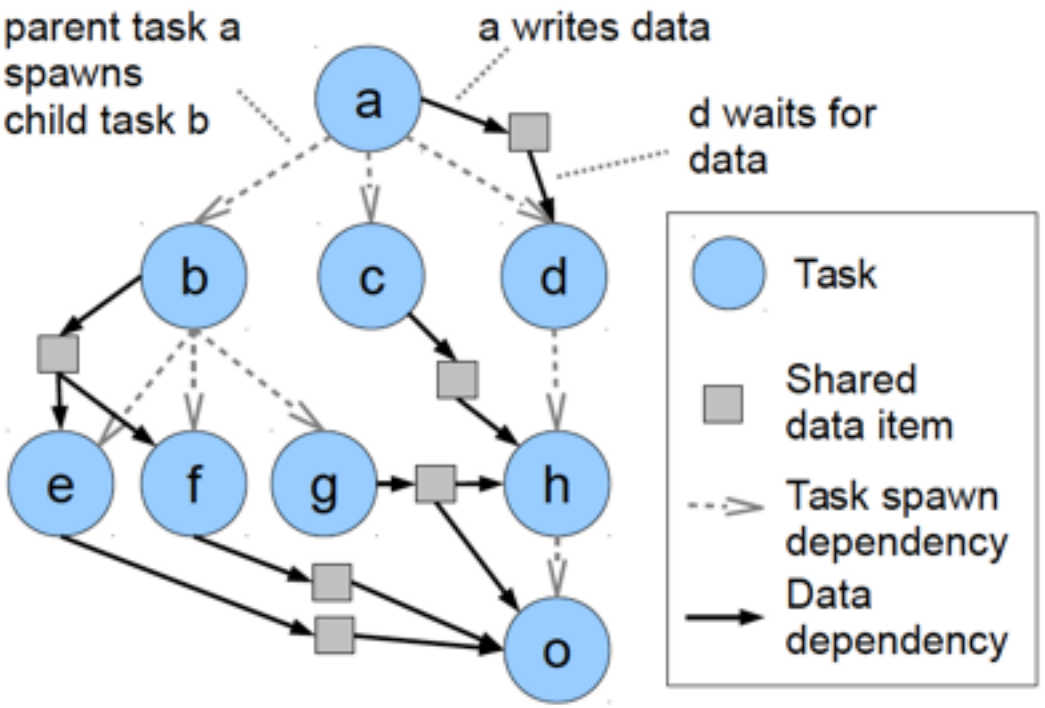
Original implementation:

Swift/K (c. 2006) - scripting for distributed computing
Still maintained and supported



Swift/K runs parallel scripts on a broad range of parallel computing resources

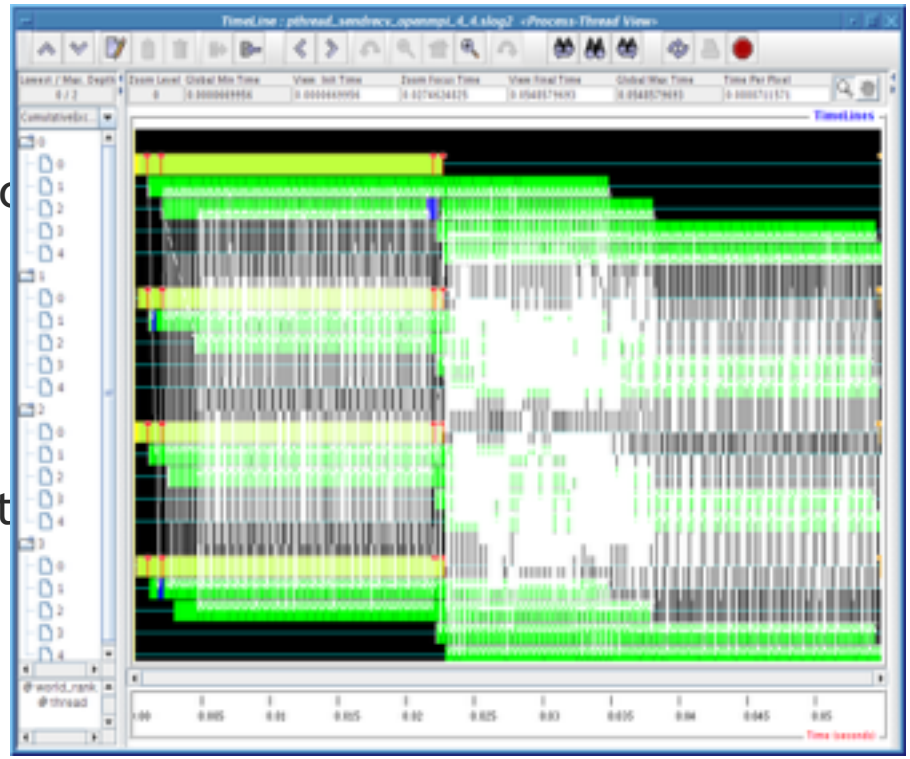
Pervasive parallel data flow



- Simple dataflow DAG on scalars
- Does not capture generality of scientific computing and analysis ensembles:
  - Optimization-directed iterations
  - Conditional execution
  - Reductions

MPI: The Message Passing Interface

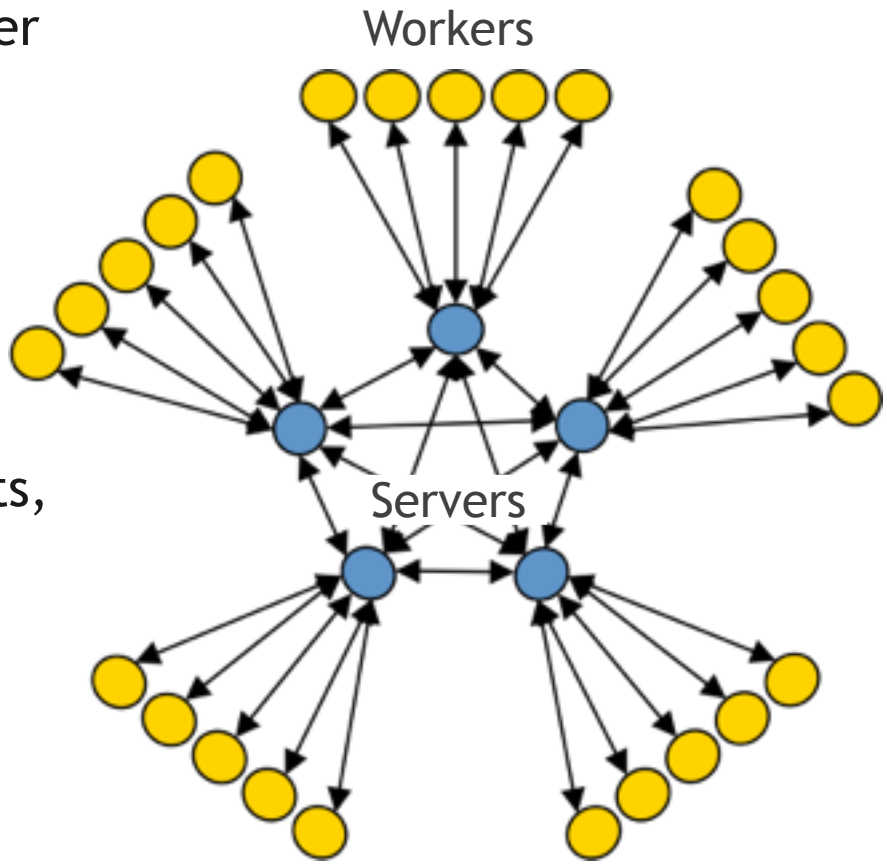
- Programming model used on large supercomputers
- Can run on many networks, including sockets, or shared memory
- Standard API for C and Fortran, other languages have working implementations
- Contains communication calls for
  - Point-to-point (send/recv)
  - Collectives (broadcast, reduce, etc
- Interesting concepts
  - Communicators: collections of communicating processing and a context
  - Data types: Language-independent data marshaling scheme



ADLB: Asynchronous Dynamic Load Balancer

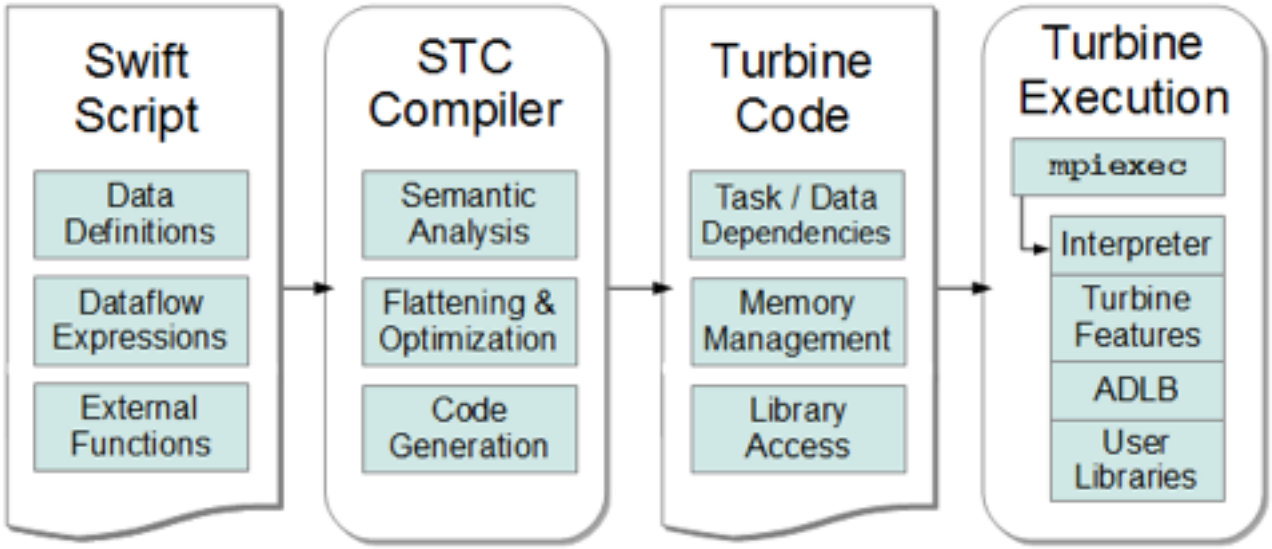
- An MPI library for master-worker workloads in C
- Uses a variable-size, scalable network of servers
- Servers implement work-stealing
- The work unit is a byte array
- Optional work priorities, targets, types

- For Swift/T, we added:
  - Server-stored data
  - Data-dependent execution
  - Tcl bindings!



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

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.

Turbine Code is Tcl

- Why Tcl?
  - Needed a simple, textual compiler target for STC
  - Needed to be able to post **code** into ADLB
  - Needed to be able to easily call C (ADLB and user code)
- Turbine
  - Includes the Tcl bindings for ADLB
  - Builtins to implement Swift primitives in Tcl (arithmetic, string operations, etc.)
- Swift/T Compiler (STC)
  - A Java program based on ANTLR
  - Generates Tcl (contains a Tcl abstract syntax tree API in Java)
  - Performs variable usage analysis and optimization

Distributed Data-dependent Execution

- STC can generate arbitrary Tcl but Swift requires dataflow processing
- Implemented this requirement in the Turbine **rule** statement
- Rule syntax:

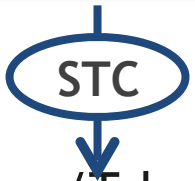
  `rule [ list` *`inputs`* `] "action string"` *`options`*`…`

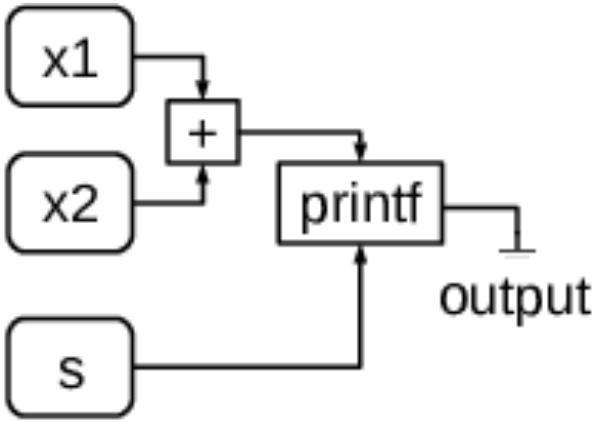
- All Swift data is registered with the ADLB distributed data store
- Rules post data-dependent tasks in ADLB
- When all inputs are stored, the action string is released
- The action string is a Tcl fragment

Translation from Swift to Turbine

- Swift:

```
x1 = 3;
s  = "value: ";
x2 = 2;
int x3;
printf("%s%i", s, x3);
x3 = x1+x2;
```





- Turbine/Tcl:

```
literal  x1 integer 3
literal  s  string  "value: "
literal  x2 integer 2
allocate x3 integer
rule [ list $x3 ] "puts \[retrieve $s\]\[retrieve $x3\]"
rule [ list $x1 $x2 ] \
 "store_integer $x3 \[expr \[retrieve $x1\]+\[retrieve $x2\]\]"
```

Tcl variables contain TDs (addresses)

Interacting with the Tcl Layer

- Can easily specify a fragment of Tcl to access:

```
(int c) add(int a, int b) "turbine" "0.0" [
  "set <<c>> [ expr <<a>> + <<b>> ]"
];
```

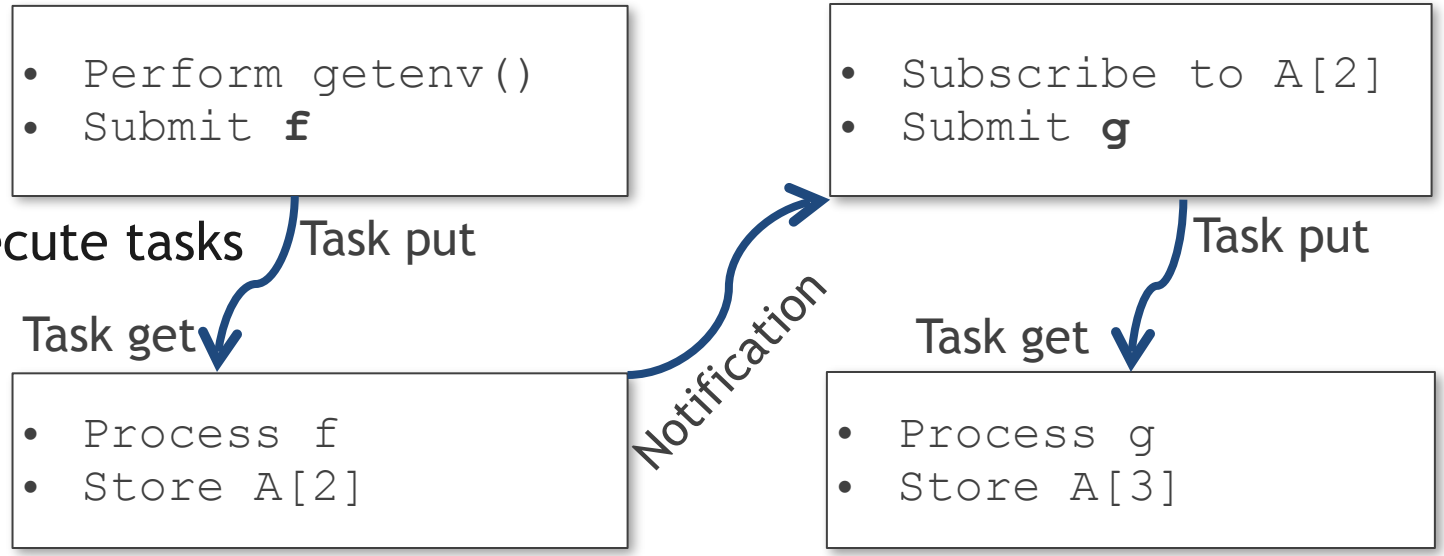
- Automatically loads the given Tcl package/version (`turbine 0.0`)
- STC substitutes Tcl variables with the `<<·>>` syntax
- Typically want to simply reference some greater Tcl or native code library

Example distributed execution

- Code

```
A[2] = f(getenv("N"));
A[3] = g(A[2]);
```

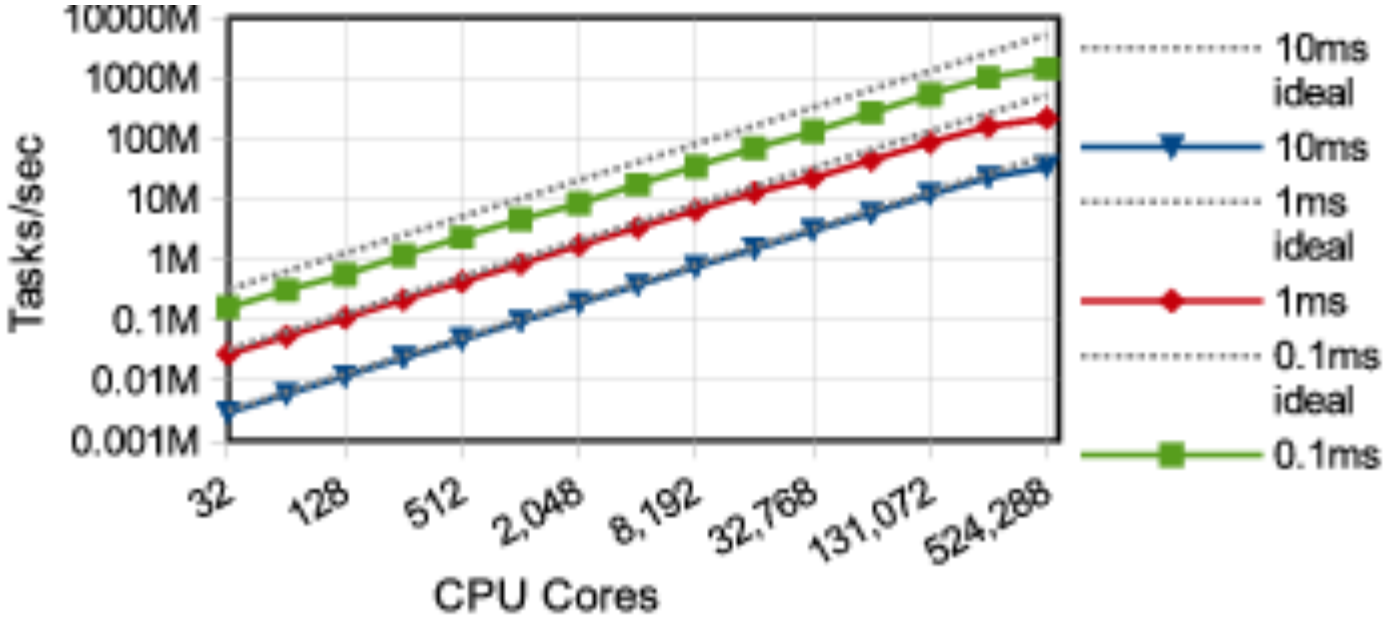
- Evaluate dataflow operations
- Workers: execute tasks



- Wozniak et al. Turbine: A distributed-memory dataflow engine for high performance many-task applications. Fundamenta Informaticae 128(3), 2013

Examples!

Extreme scalability for small tasks



- 1.5 billion tasks/s on 512K cores of Blue Waters, so far
- Armstrong et al. Compiler techniques for massively scalable implicit task parallelism. Proc. SC 2014.

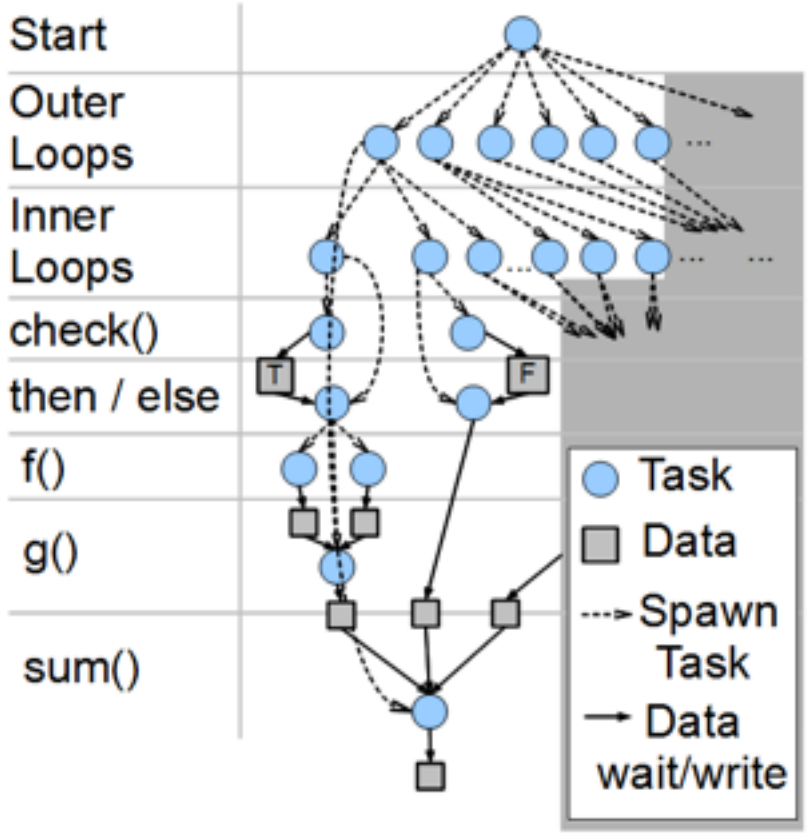
Characteristics of very large Swift programs

```
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]);
}
```

- The goal is to support billion-way concurrency: O(10^9)
- Swift script logic will control trillions of variables and data dependent tasks
- Need to distribute Swift logic processing over the HPC compute system

Swift/T: Fully parallel evaluation of complex scripts

```
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]);
}
```

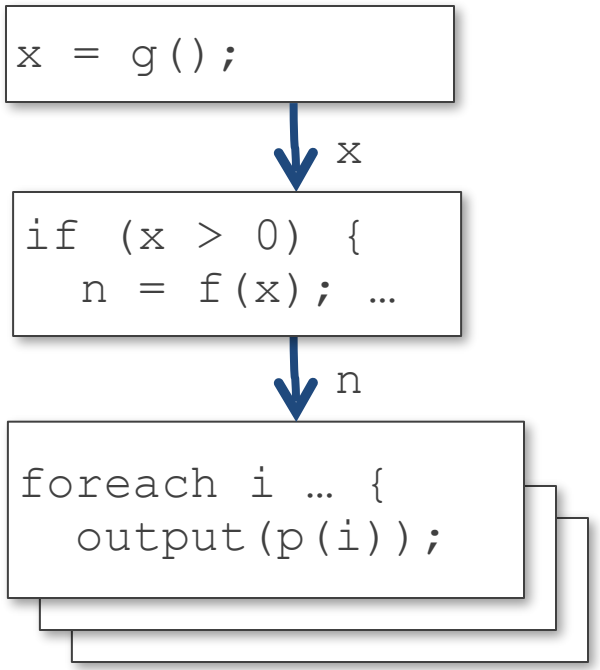


- Wozniak et al. Large-scale application composition via distributed-memory data flow processing. Proc. CCGrid 2013.

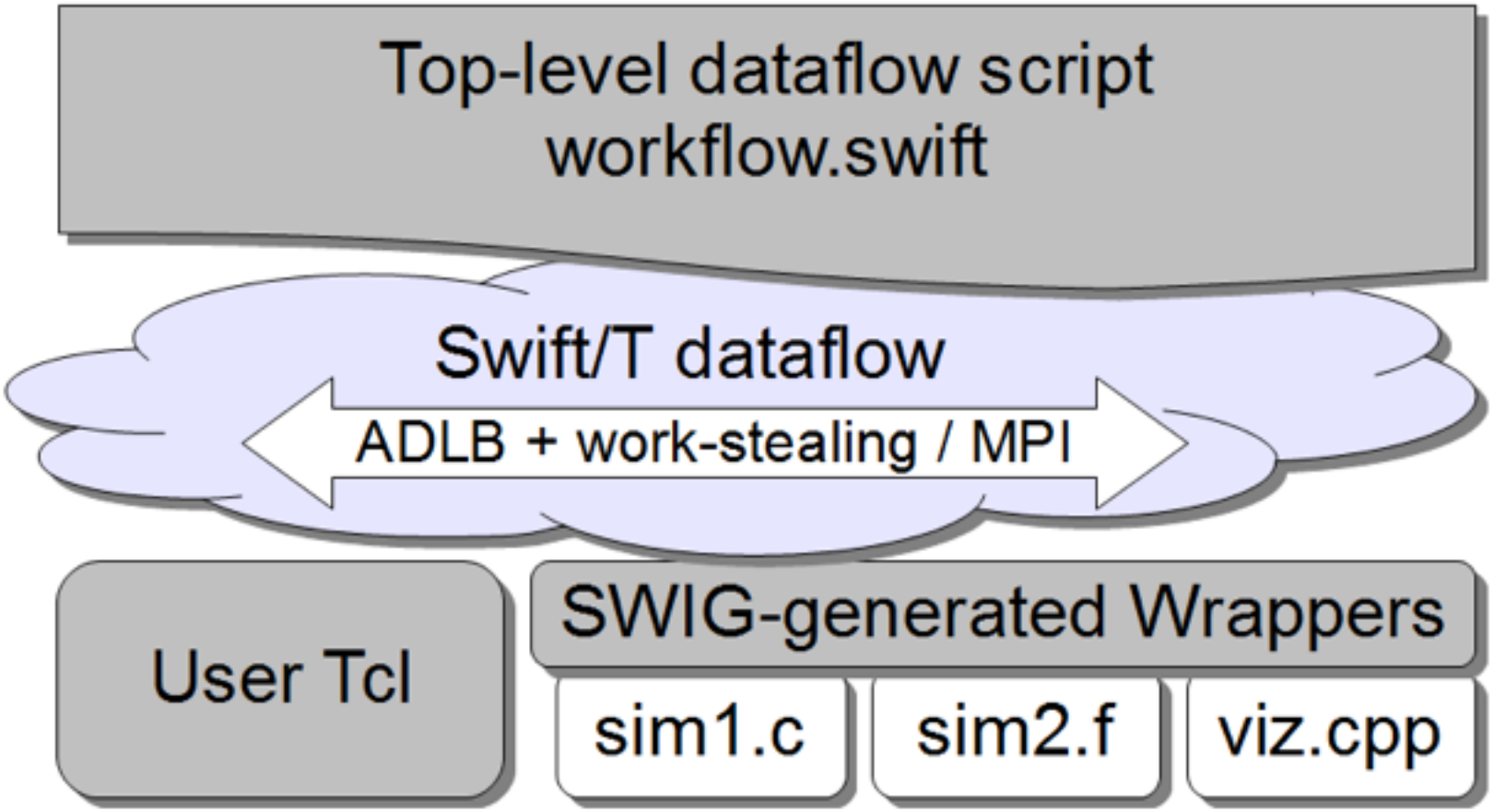
Swift code in dataflow

- 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
- Data dependent-tasks are managed by ADLB

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

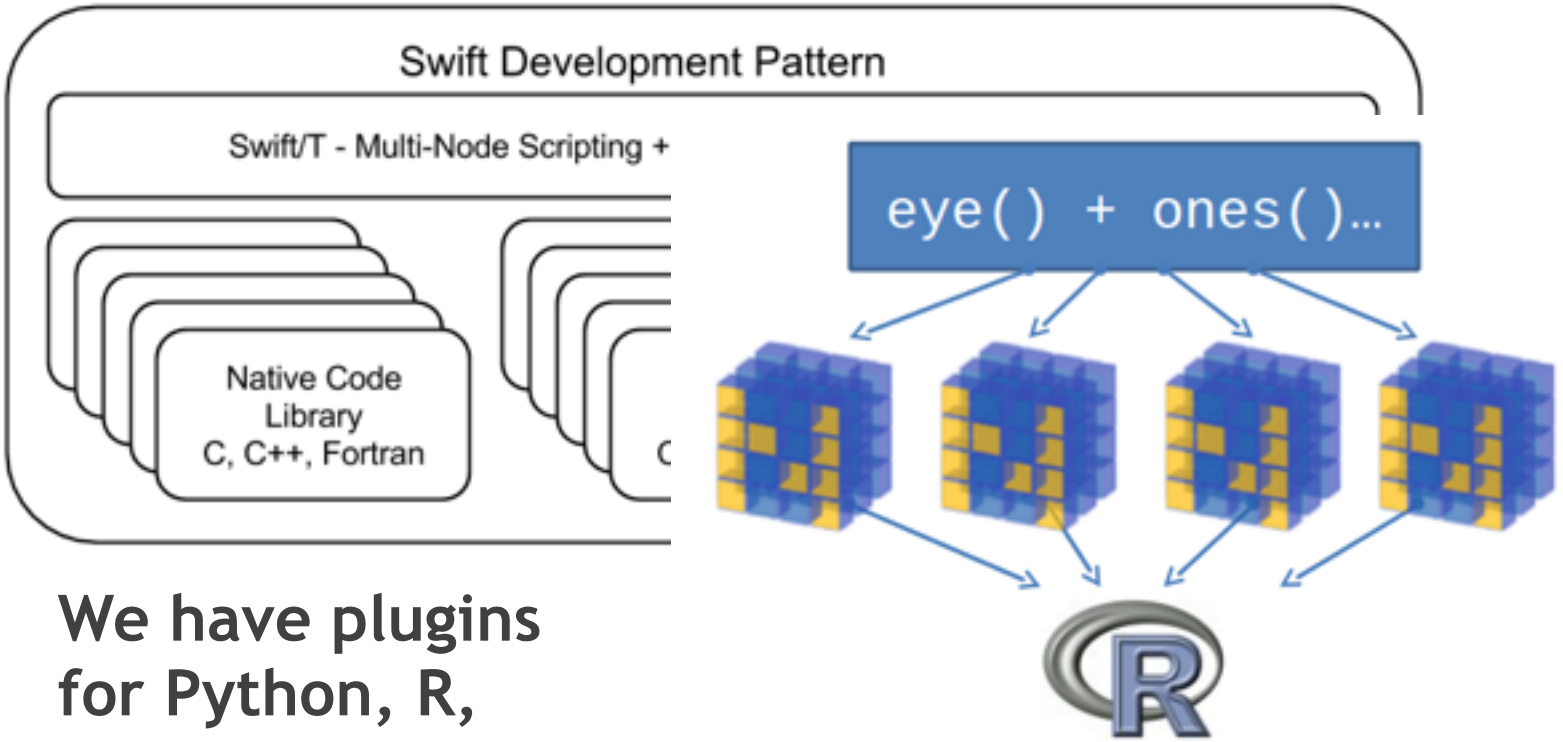


Hierarchical programming model



- Including MPI libraries

Support calls to embedded interpreters

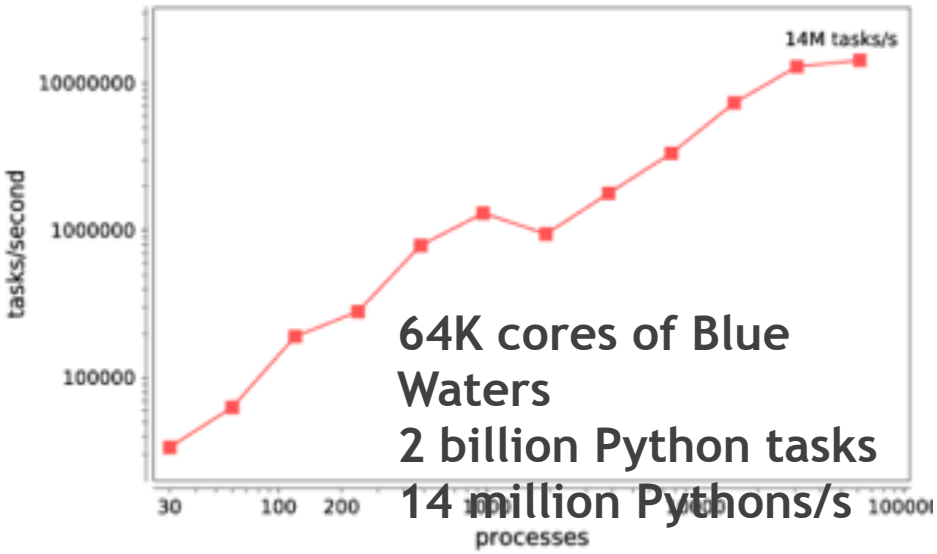


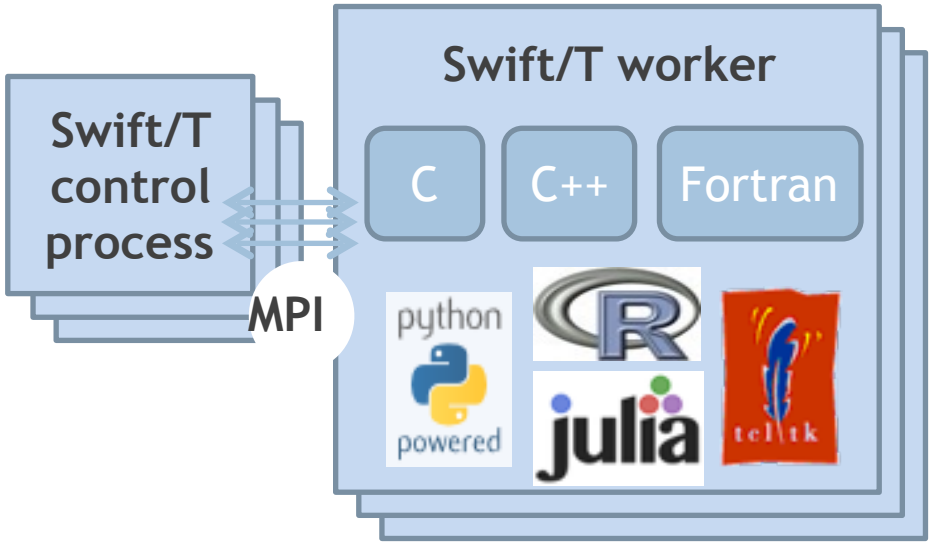
We have plugins for Python, R, Tcl, Julia, and QtScript

- Wozniak et al. Toward computational experiment management via multi-language applications. Proc. ASCR SWP4XS, 2014.
- Wozniak et al. Interlanguage parallel scripting for distributed-memory scientific computing. Proc. CLUSTER 2015.

Swift/T: Enabling high-performance scripting

- Write site-independent scripts in **Swift** language
- Execute on scalable runtime: **Turbine**
- Automatic **parallelization** and data movement
- Run **native code or script fragments** as application tasks
- Rapidly subdivide large partitions for MPI libraries using **MPI 3**





Swift/T features for task control

NOVEL FEATURES: RUNTIME

Task priorities

- User-written annotation on function call

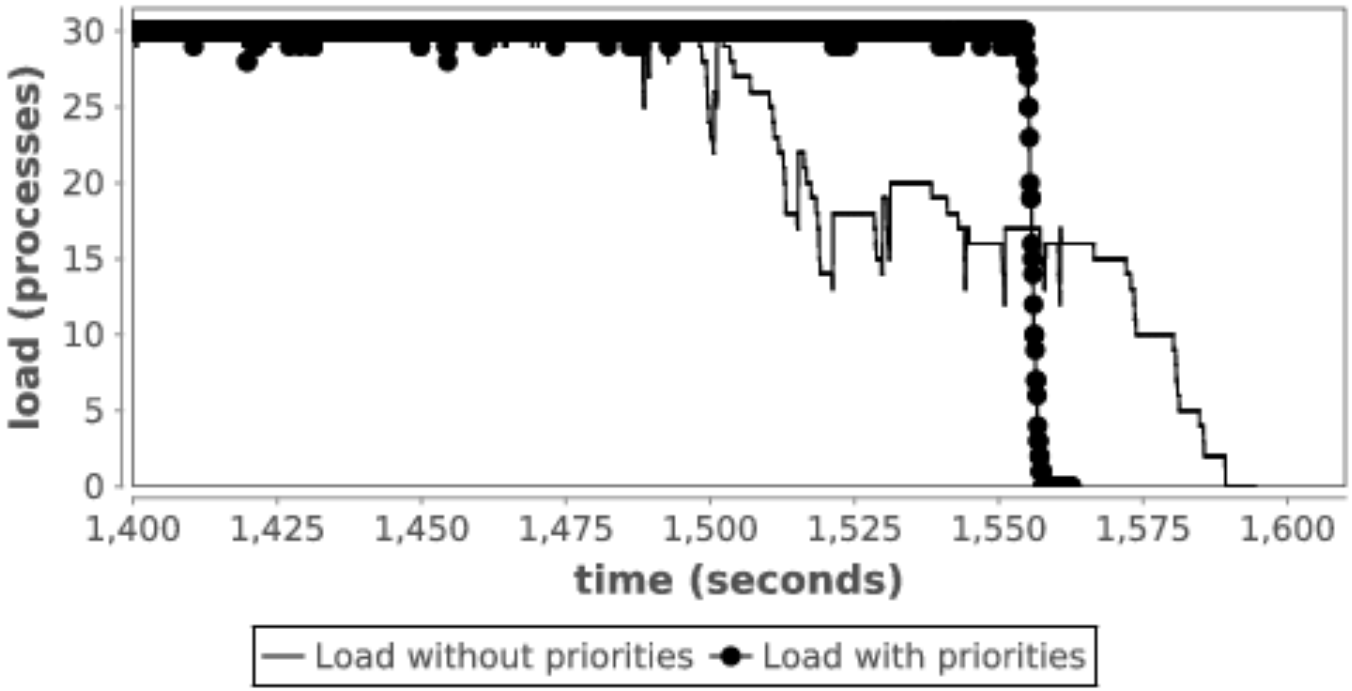
```
foreach i in 0:N-1 {
  @prio=i f(i);
}
```

- Priorities are best-effort and are relative to tasks on a given ADLB server

- Could be used to:
  - Promote tasks that release lots of other dependent work
  - Compute more important work early (before allocation expires!)
  - Deal with trailing tasks (next slide)

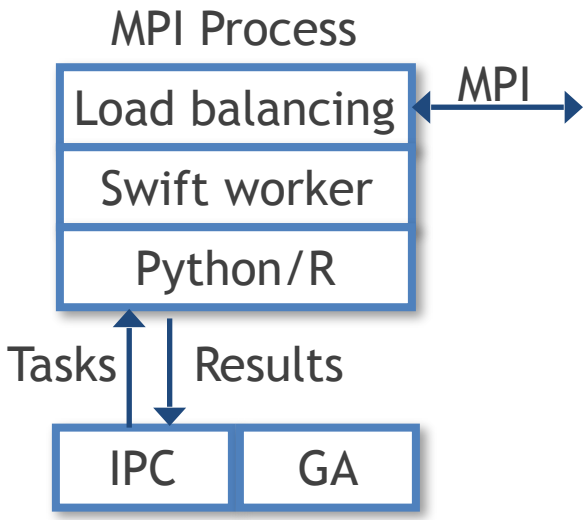
Prioritize long-running tasks

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

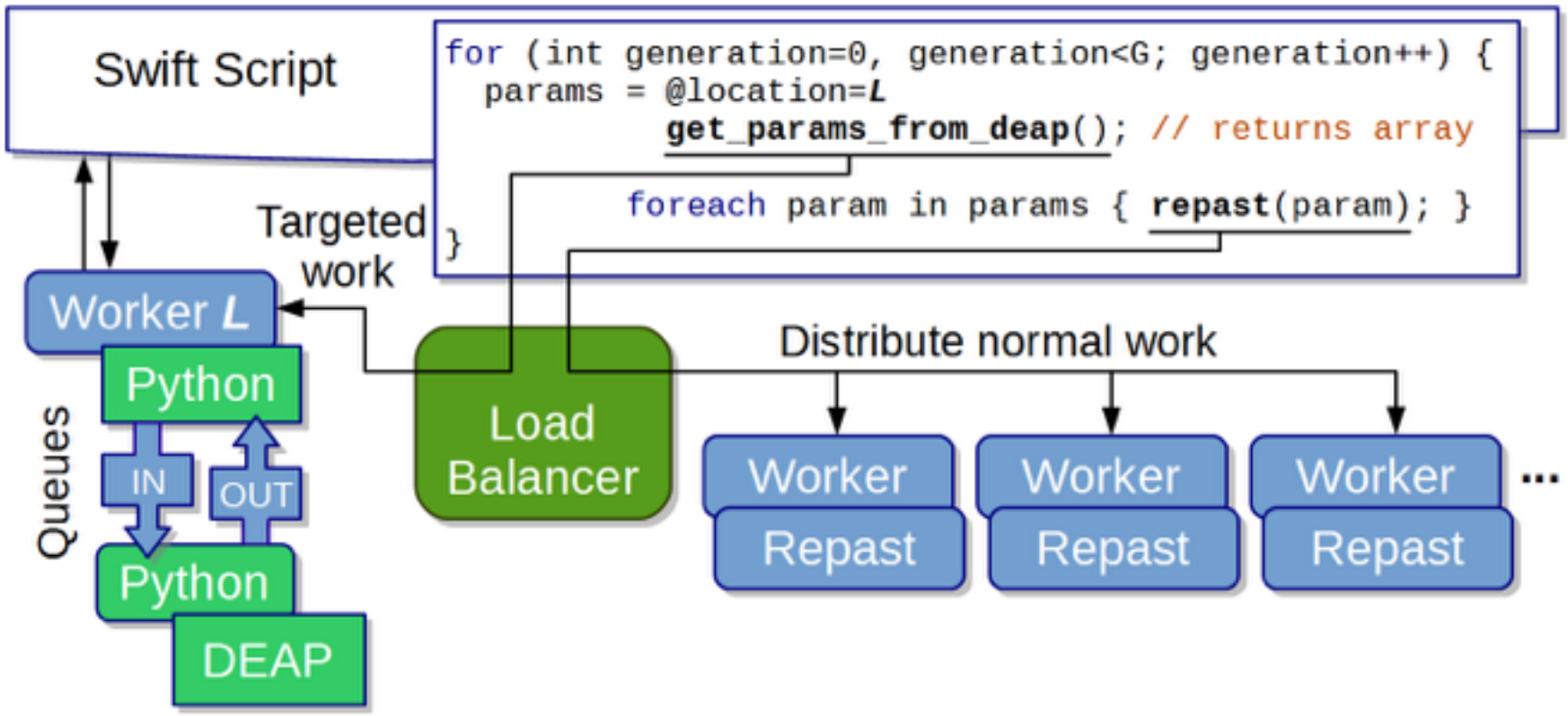


Stateful external interpreters

- Desire to use high-level, 3rd party algorithms in Python, R to orchestrate Swift workflows, e.g.:
  - Python DEAP for evolutionary algorithms
  - R language GA package
- Typical control pattern:
  - GA minimizes the cost function
  - You pass the cost function to the library and wait
- We want Swift to obtain the parameters from the library
  - We launch a stateful interpreter on a thread
  - The "cost function" is a dummy that returns the parameters to Swift over IPC
  - Swift passes the real cost function results back to the library over IPC
- Achieve **high productivity** and **high scalability**
  - Library is not modified – unaware of framework!
  - Application logic extensions in high-level script



Unnecessary details: Epidemics ensembles

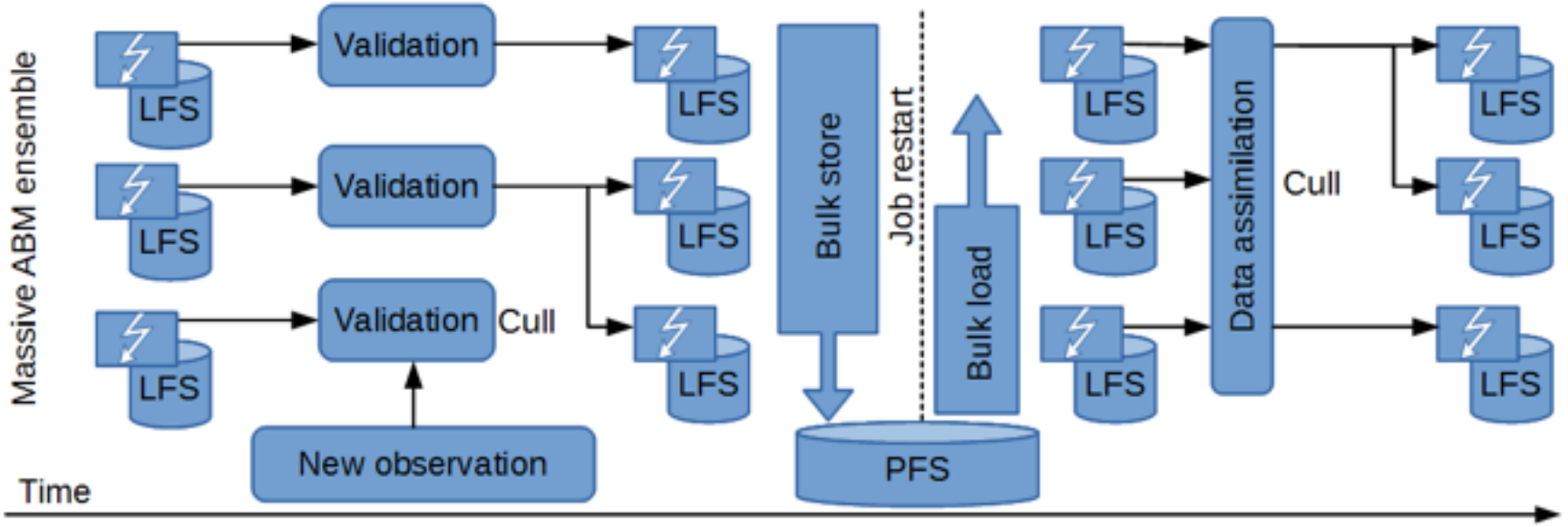


Epidemic simulators

- Wozniak et al. Many Resident Task Computing in Support of Dynamic Ensemble Computations. Proc. MTAGS 2015.

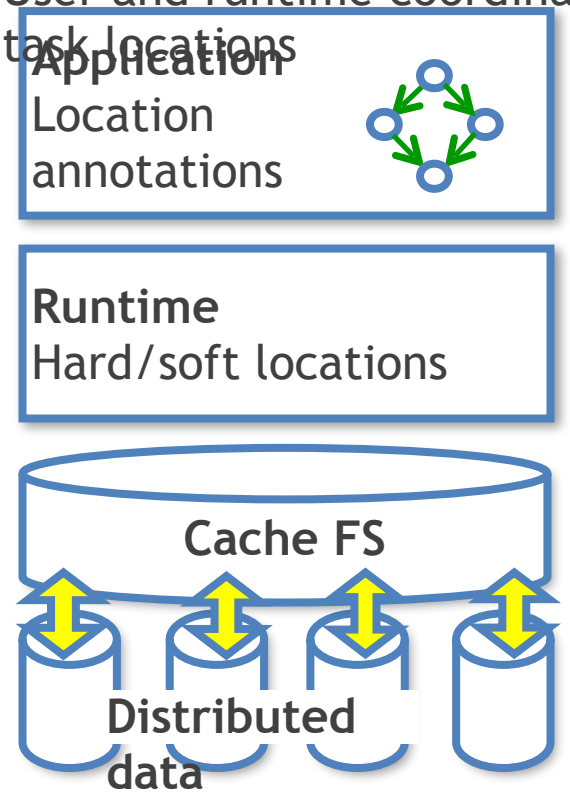
Ebola spread modeling

- Epidemic analysis- combining **agent-based models** with observation
- Received emergency funding late last year
- Combines Python-based evolutionary algorithm with high-performance **agent-based** epidemic modeling code
- Want to compare simulations with observations in real-time *as disease spreads through a population*



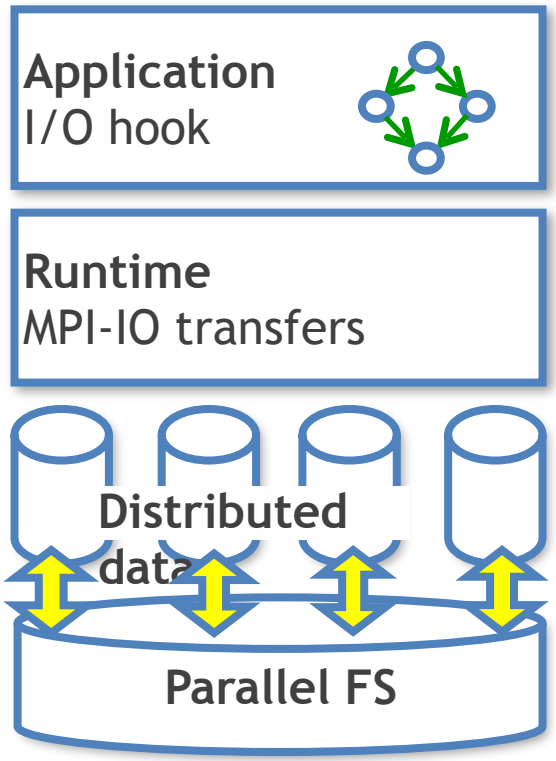
Features for Big Data analysis

- **Location-aware scheduling**
  User and runtime coordinate data/ task locations



- F. Duro et al. Exploiting data locality in Swift/T workflows using Hercules . Proc. NESUS Workshop, 2014.

- **Collective I/O**
  User and runtime coordinate data/ task locations



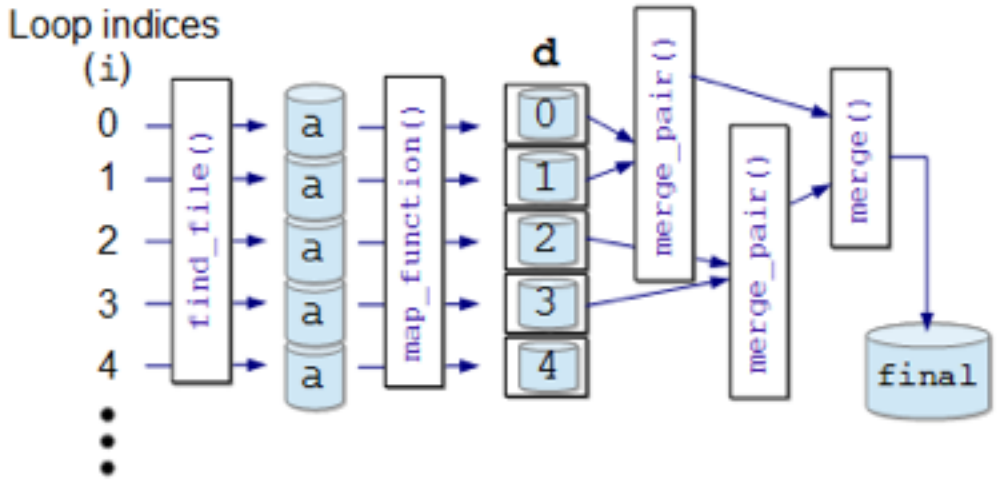
- Wozniak et al. Big data staging with MPI-IO for interactive X-ray science. Proc. Big Data Computing, 2014.

Abstract, extensible MapReduce in Swift

```
main {
  file d[];
  int N = string2int(argv("N"));
  // Map phase
  foreach i in [0:N-1] {
    file a = find_file(i);
    d[i] = map_function(a);
  }
  // Reduce phase
  file final <"final.data"> = merge(d, 0, tasks-1);
}

(file o) merge(file d[], int start, int stop) {
  if (stop-start == 1) {
    // Base case: merge pair
    o = merge_pair(d[start], d[stop]);
  } else {
    // Merge pair of recursive calls
    n = stop-start;
    s = n % 2;
    o = merge_pair(merge(d, start,     start+s),
                   merge(d, start+s+1, stop));
  }}
```

- User needs to implement map_function() and merge()
- These may be implemented in native code, Python, etc.
- Could add **annotations**
- Could add additional custom **application logic**



Hercules

- Want to run arbitrary workflows over distributed filesystems that expose data locations: **Hercules** is based on **Memcached**
  - Data analytics, post-processing
  - Exceed generality MapReduce: without losing data optimizations
- Can *optionally* send a Swift task to a particular location with simple syntax:

```
foreach i in 0:N-1 {
  location L = locationFromRank(i);
  @location=L f(i);
}
```

- Can obtain ranks from hostnames:

```
int rank = hostmapOneWorkerRank("my.host.edu");
```

- Can now specify location constraints:

```
location L = location(rank, HARD|SOFT, RANK|NODE);
```

- Much more to be done here!

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

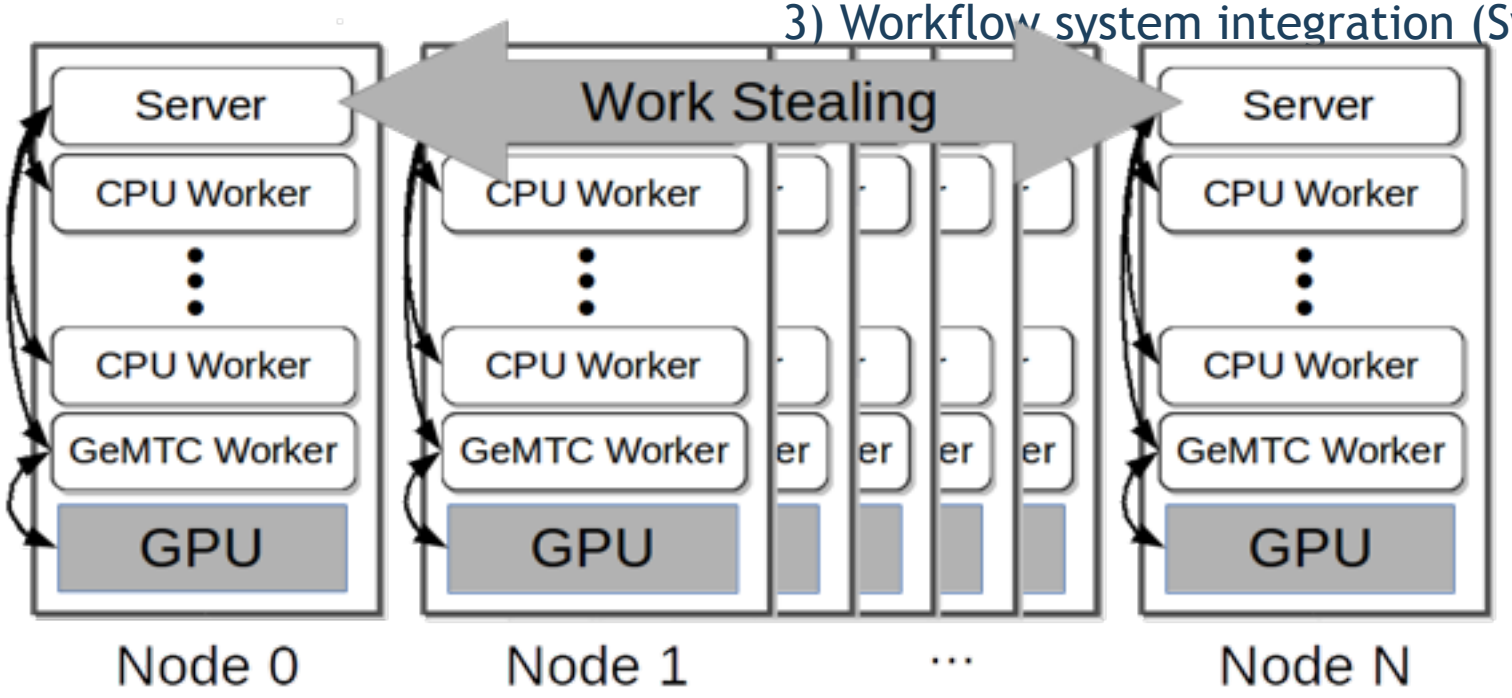
5) Increase concurrency to warp level

Approach:

Design & implement GeMTC middleware:

1) Manages GPU 2) Spread host/ device

3) Workflow system integration (Swift/



What just happened?

LOGGING AND DEBUGGING

Logging and debugging in Swift

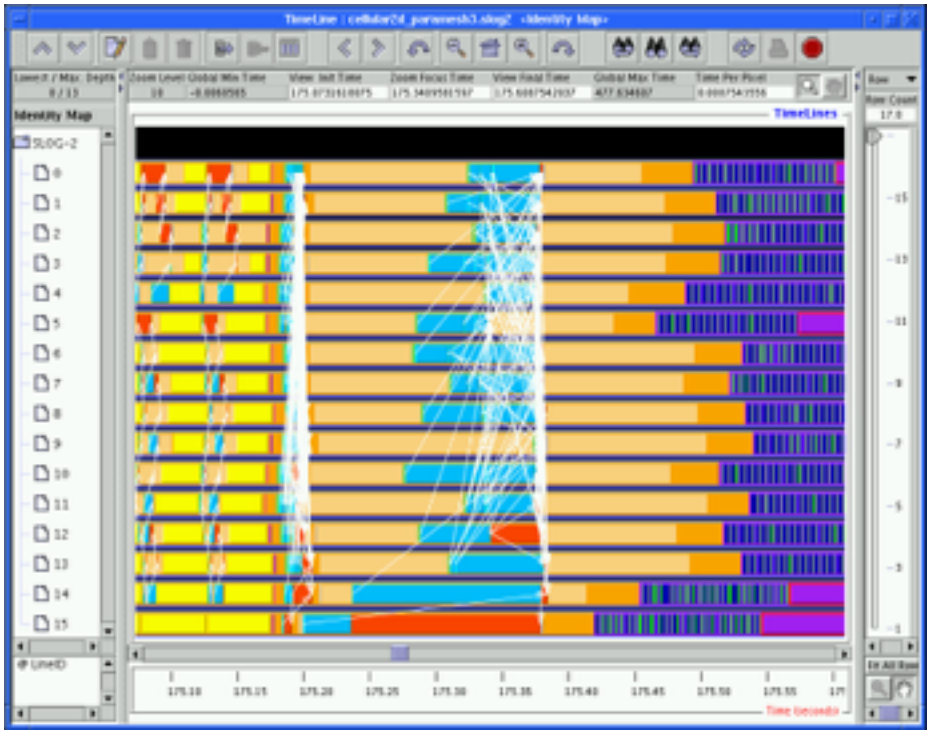
- Traditionally, Swift programs are debugged through the log or the TUI (text user interface)
- Logs were produced using normal methods, containing:
  - Variable names and values as set with respect to thread
  - Calls to Swift functions
  - Calls to application code
- A restart log could be produced to restart a large Swift run after certain fault conditions
- Methods require single Swift site: do not scale to larger runs

Logging in MPI

- The Message Passing Environment (MPE)
- Common approach to logging MPI programs
- Can log MPI calls or application events – can store arbitrary data
- Can visualize log with Jumpshot

- Partial logs are stored at the site of each process
  - Written as necessary to shared file system
    - in large blocks
    - in parallel
  - Results are merged into a big log file (CLOG, SLOG)

- Work has been done optimize the file format for various queries



Logging in Swift & MPI

- Now, combine it together
- Allows user to track down erroneous Swift program logic

- Use MPE to log data, task operations, calls to native code
- Use MPE metadata to annotate events for later queries

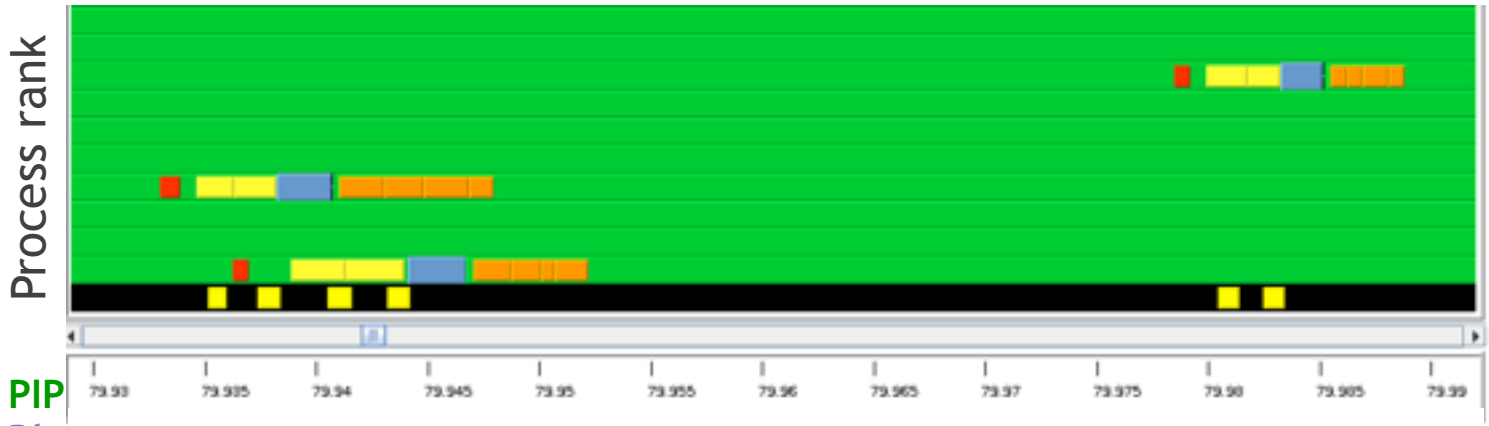
- MPE cannot be used to debug native MPI programs that abort
  - On program abort, the MPE log is not flushed from the process-local cache
  - Cannot reconstruct final fatal events

- MPE can be used to debug Swift application programs that abort
  - We finalize MPE before aborting Swift
  - (Does not help much when developing Swift itself)
  - But primary use case is non-fatal arithmetic/logic errors

- Wozniak et al. A model for tracing and debugging large-scale task-parallel programs with MPE. Proc LASH-C, 2013.

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



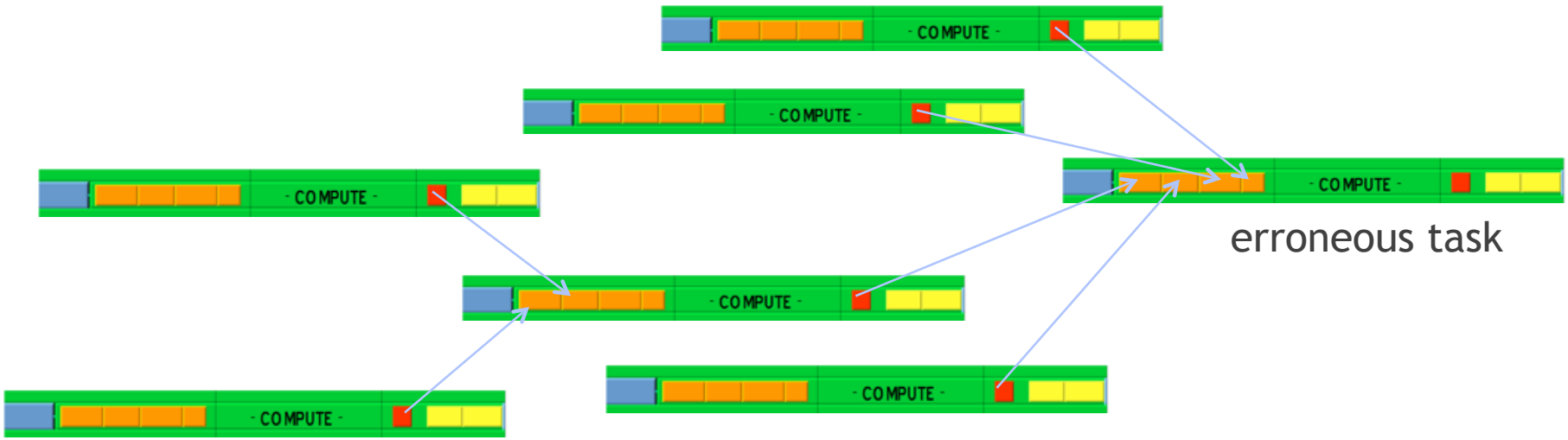
- PIP
Blu
Server process (main task) of PIPS application run

Jumpshot view of PIPS application run

- 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)

Molecular dynamics simulation, X-ray science data processing

APPLICATIONS

Can we build a Makefile in Swift?

- User wants to test a variety of compiler optimizations
- Compile set of codes under wide range of possible configurations
- Run each compiled code to obtain performance numbers
- Run this at large scale on a supercomputer (Cray XE6)

- **In Make you say:**

```
CFLAGS = ...
f.o : f.c
    gcc $(CFLAGS) f.c -o f.o
```

In Swift you say:

```
string cflags[] = ...;
f_o = gcc(f_c, cflags);
```

CHEW example code

Apps

```
app (object_file o) gcc(c_file c, string cflags[]) {
// Example:
//  gcc   -c   -O2    -o  f.o f.c
   "gcc" "-c" cflags "-o" o   c;
}

app (x_file x) ld(object_file o[], string ldflags[]) {
// Example:
//  gcc          -o  f.x f1.o f2.o ...
   "gcc" ldflags "-o" x   o;
}

app (output_file o) run(x_file x) {
  "sh" "-c" x @stdout=o;
}

app (timing_file t) extract(output_file o) {
  "tail" "-1" o "|" "cut" "-f" "2" "-d" " " @stdout=t;
}
```

Swift code

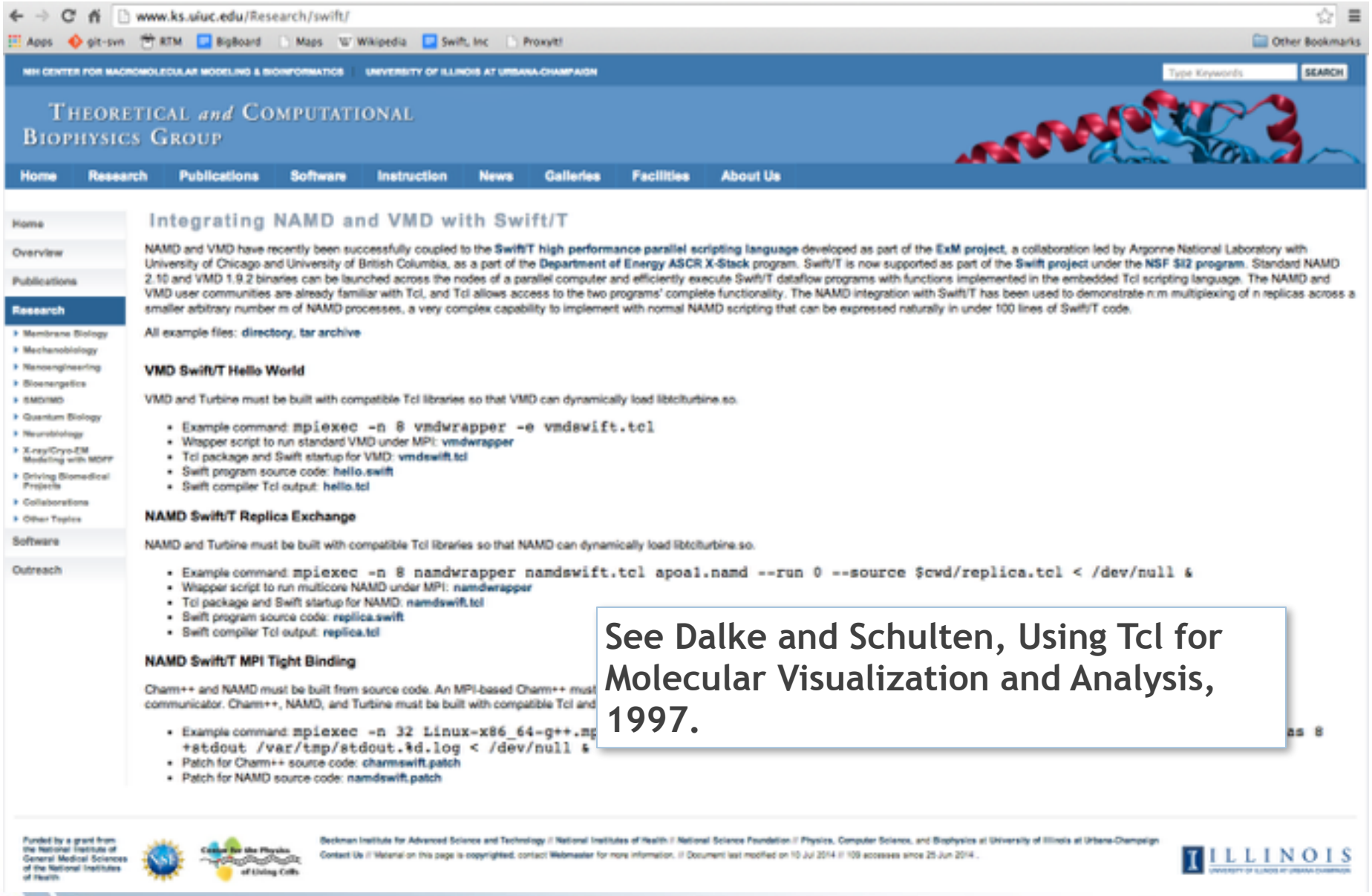
```
string program_name = "programs/program1.c";
c_file c = input(program_name);

// For each
foreach O_level in [0:3]  {
  make file names…
  // Construct compiler flags
  string O_flag = sprintf("-O%i", O_level);
  string cflags[] = [ "-fPIC", O_flag ];

  object_file o<my_object> = gcc(c, cflags);
  object_file objects[] = [ o ];
  string ldflags[] = [];
  // Link the program
  x_file x<my_executable> = ld(objects, ldflags);
  // Run the program
  output_file out<my_output> = run(x);
  // Extract the run time from the program output
  timing_file t<my_time> = extract(out);
```

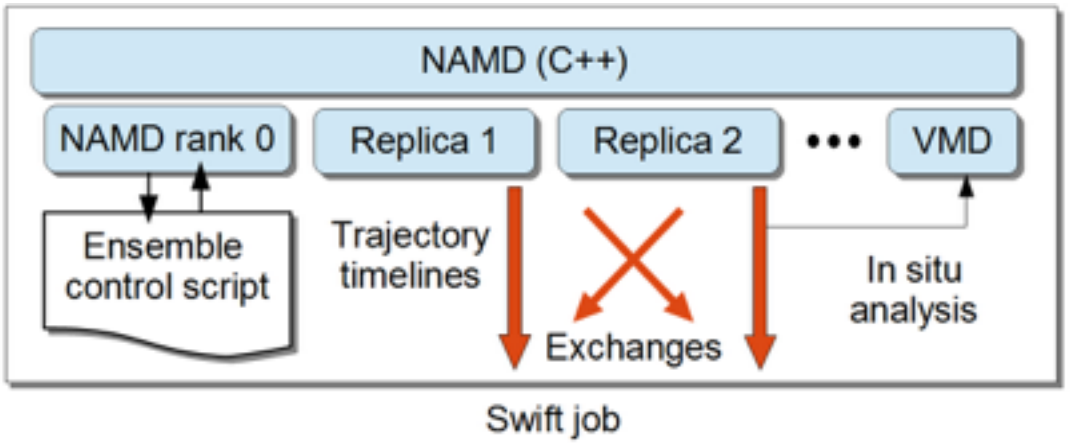
Swift integration into NAMD and VMD

www.ks.uiuc.edu/Research/swift



NAMD Replica Exchange Limitations

- One-to-one replicas to Charm++ partitions:
  - Available hardware must match science.
  - Batch job size must match science.
  - Replica count fixed at job startup.
  - No hiding of inter-replica communication latency.
  - No hiding of replica performance divergence.
- Can a different programming model help?



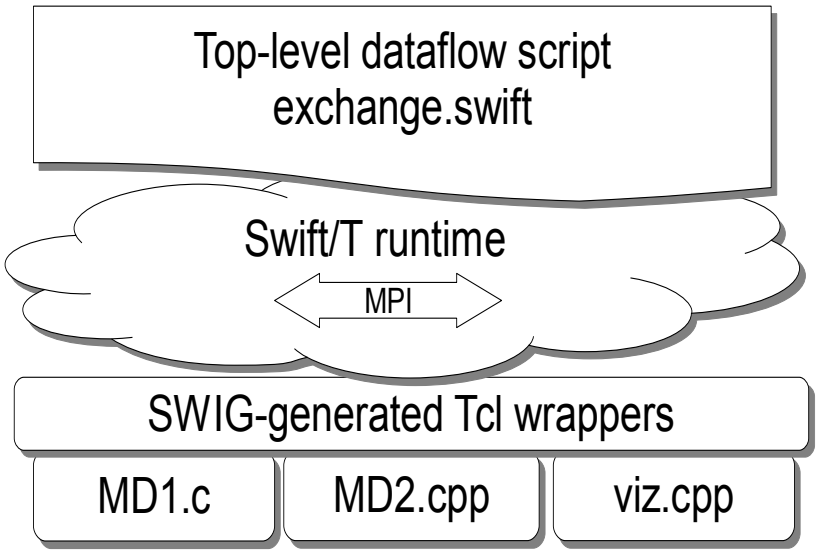
Benefits of using Swift within NAMD / VMD

Work by Jim Phillips and John Stone of UIUC NAMD Group (Schulten Lab) :

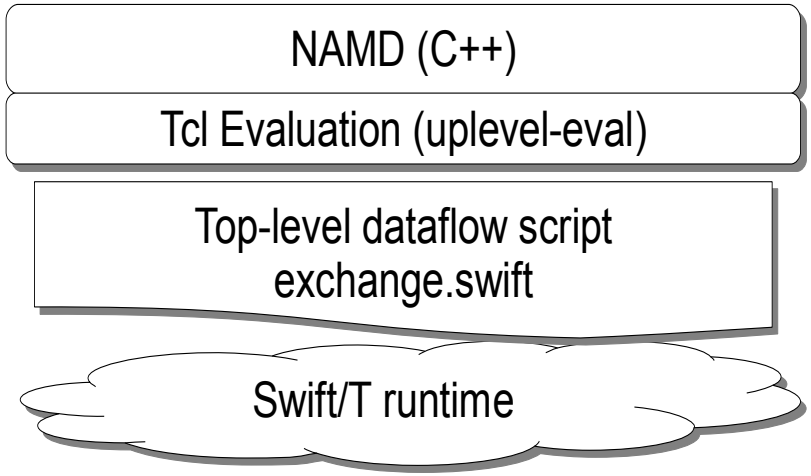
- NAMD 2.10 and VMD 1.9.2 can run Swift dataflow programs using functions from their embedded Tcl scripting language.
- NAMD and VMD users are already familiar with Tcl, and Tcl allows access to the two apps' complete functionality.
- Swift has been used to demonstrate n:m multiplexing of n replicas across a smaller arbitrary number m of NAMD processes
- This is very complex to do with normal NAMD scripting that can be expressed naturally in under 100 lines of Swift/T code.

NAMD/VMD and Swift/T

Typical Swift/T Structure

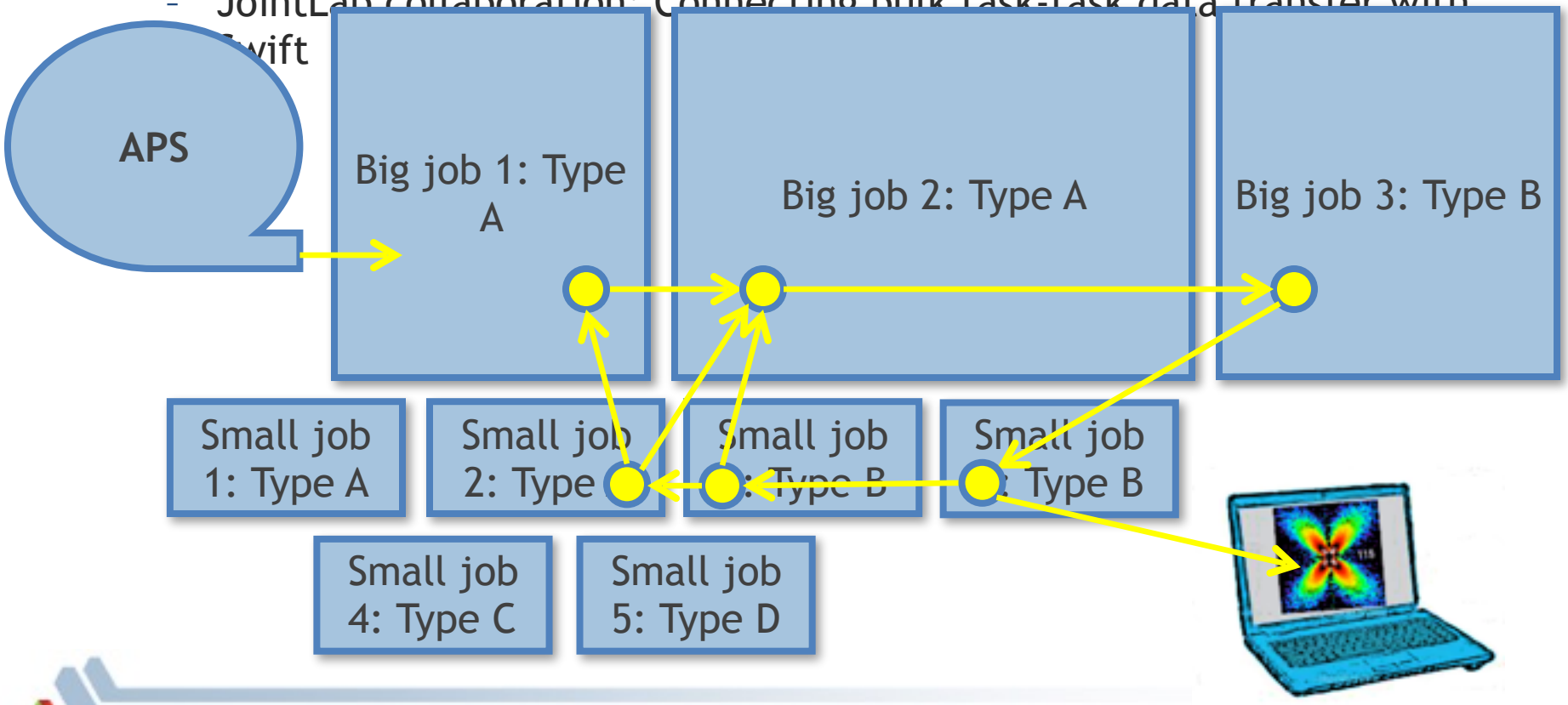


NAMD/VMD Structure



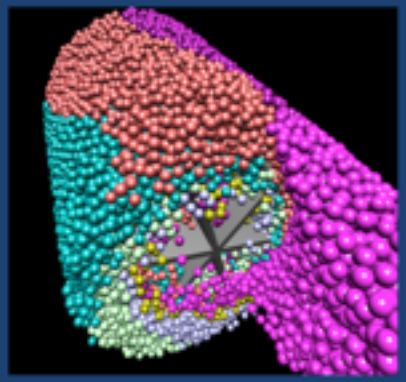
Future work: Extreme scale ensembles

- Enhance Swift for exascale experiment/simulate/analyze ensembles
  - Deploy stateful, varying sized jobs
  - Outermost, experiment-level coordination via dataflow
  - Plug in experiments and human-in-the-loop models (dataflow filters)
  - JointLab collaboration: Connecting bulk task-task data transfer with Swift

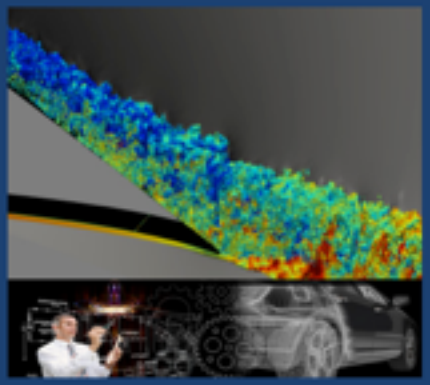


Technology transfer - *Parallel.Works*

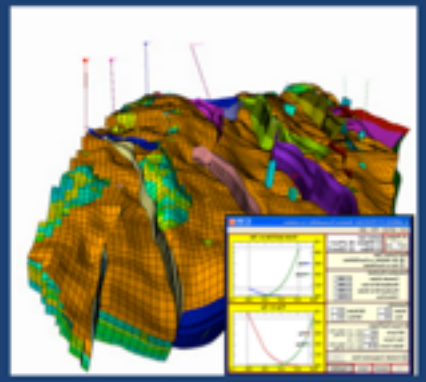
Companies increasingly depend on computer modeling and analysis for product designs and critical business decisions



Pharma, Materials, Batteries



Vehicles, Planes, Wind Turbines



Oil and Gas Reservoirs

Buildings, Infrastructure, Urban Planning

Companies need to *compute to compete*:

- more design simulations at higher fidelity to optimize products and services.
- more big-data analyses, faster, to ask more questions.

The end of "Moore's Law" requires *parallel computing* to meet these needs.

parallel.works

An incubation venture of the University of Chicago's CIE: Chicago Innovation Exchange
http://cie.uchicago.edu

Technology transfer - *Parallel.Works*

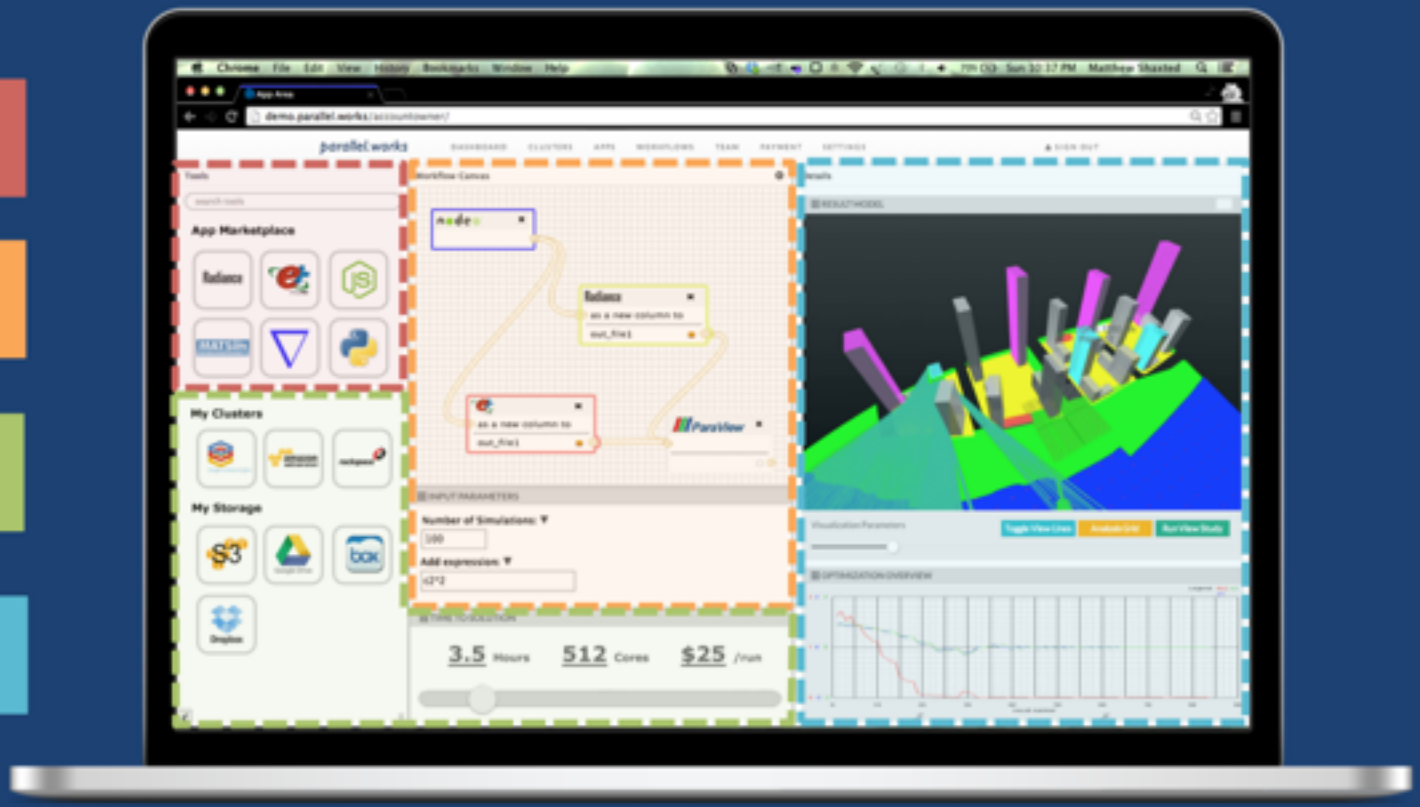
The *parallel.works* solution

Select third-party and open-source applications

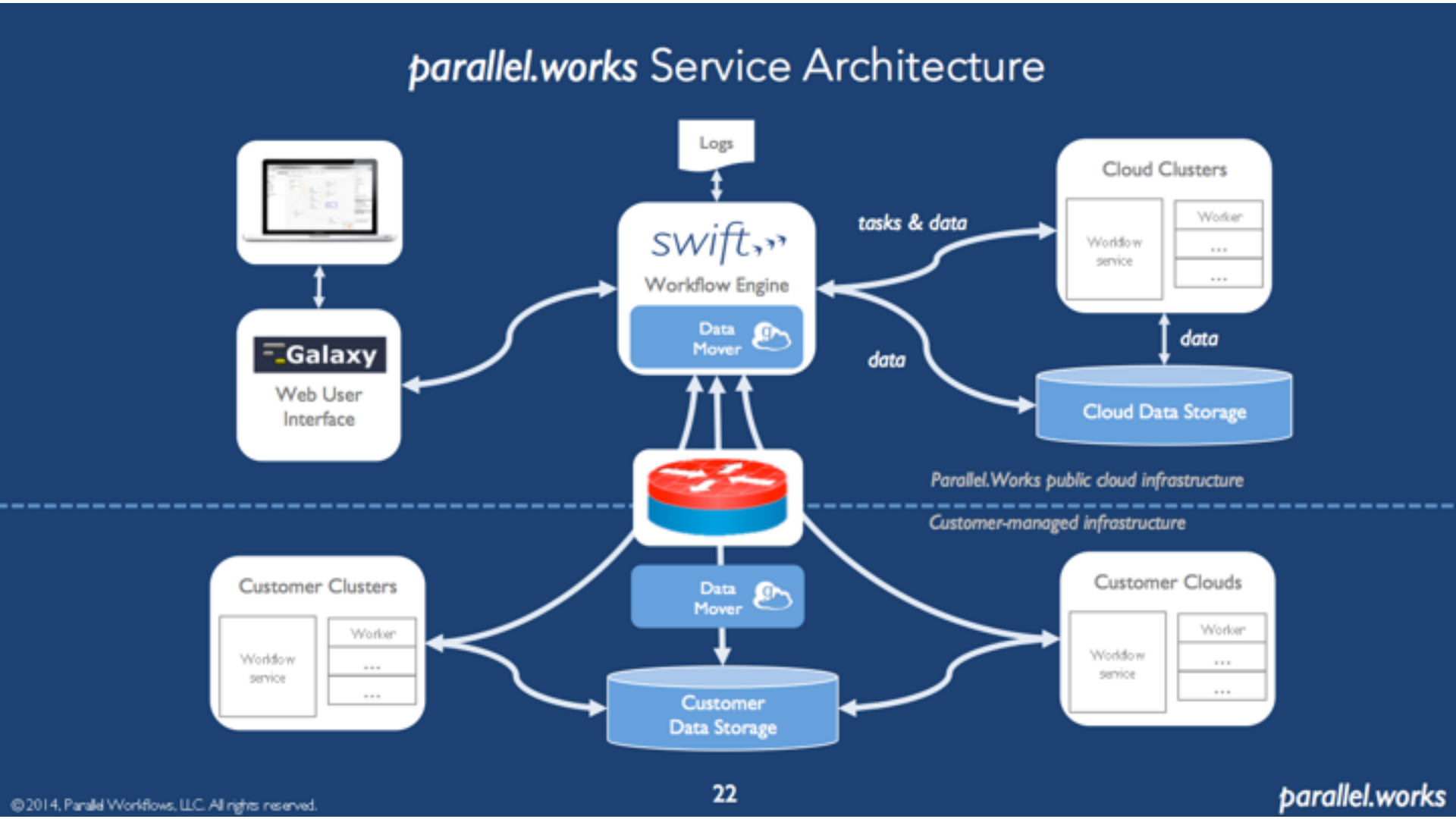
Link apps together to create a workflow

Select compute and storage resources and desired time to solution

Run simulation workflow and visualize results



Technology transfer - *Parallel.Works*



Summary

- **Swift:** High-level scripting for outermost programming constructs
- Heavily based on **Tcl**!
- Described novel features for **task control** and **big data computing** on clusters and supercomputers
- **Thanks** to the Swift team: Mike Wilde, Ketan Maheshwari, Tim Armstrong, David Kelly, Yadu Nand, Mihael Hategan, Scott Krieder, Ioan Raicu,
  Dan Katz, Ian Foster
- **Thanks** to the Tcl organizers

- **Questions?**