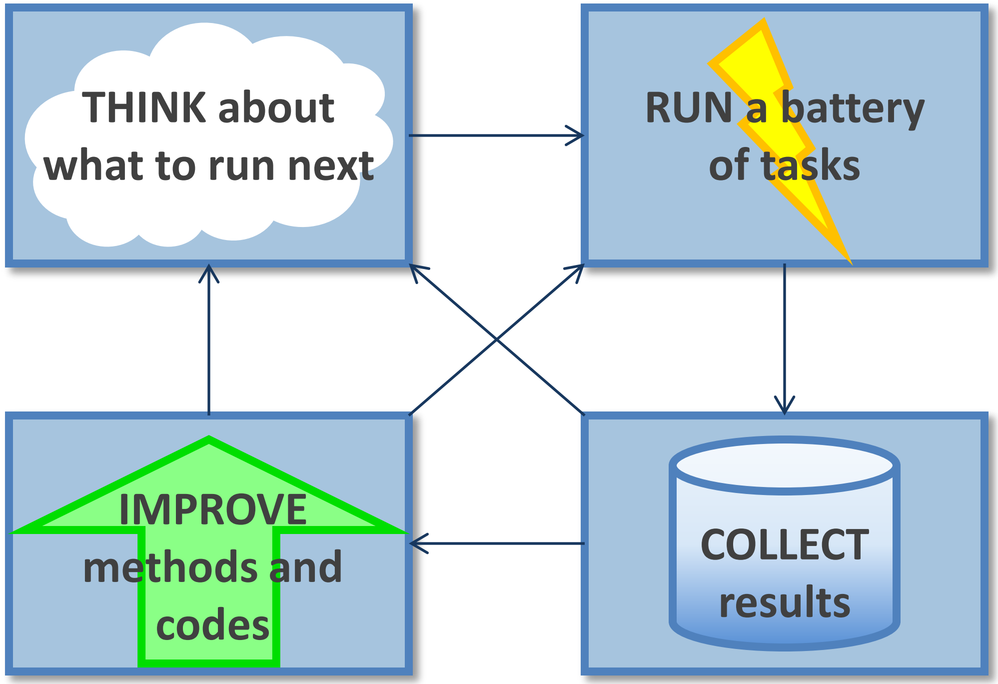
The Assembly and Management of Scalable Computational Experiments

Justin M Wozniak

wozniak@mcs.anl.gov

The Scientific Computing Campaign



- This talk will address most of these components

Software for the Computing Campaign

- Assembling the compute tasks
  - Code coupling
  - Task communication
- Running large numbers of tasks
  - Expressing complex workflows
  - Deploying large workloads
- Managing experimental data
  - Performing I/O on big machines
  - Data organization and provenance
- Improving experimental runs
  - Debugging and performance analysis for workflows
  - Plotting and visualization

Goal: Programmability for large scale analysis

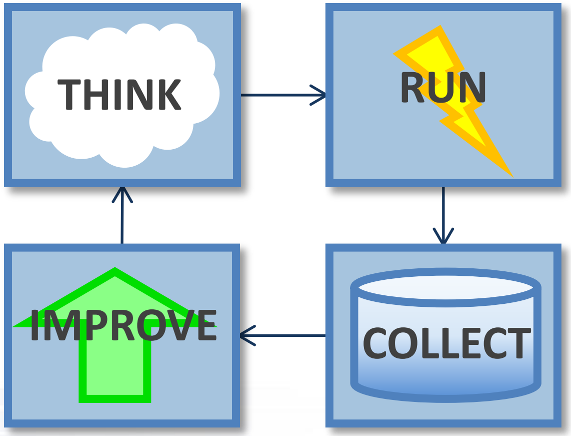
- Our solution is "many-task" computing: higher-level applications composed of many run-to-completion tasks: **input→compute→output**
  Message passing is handled by our implementation details

- Programmability
  - Large number of applications have this natural structure at upper levels: Parameter studies, ensembles, Monte Carlo, branch-and-bound, stochastic programming, UQ
  - Coupling extreme-scale applications to preprocessing, analysis, and visualization
- Data-driven computing
  - Dataflow-based execution models
  - Data organization tools in the programming languages
- Challenges
  - Load balancing, data movement, expressibility

Practical context: 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



SWIFT/K OVERVIEW

Swift programming model: all progress driven by concurrent dataflow

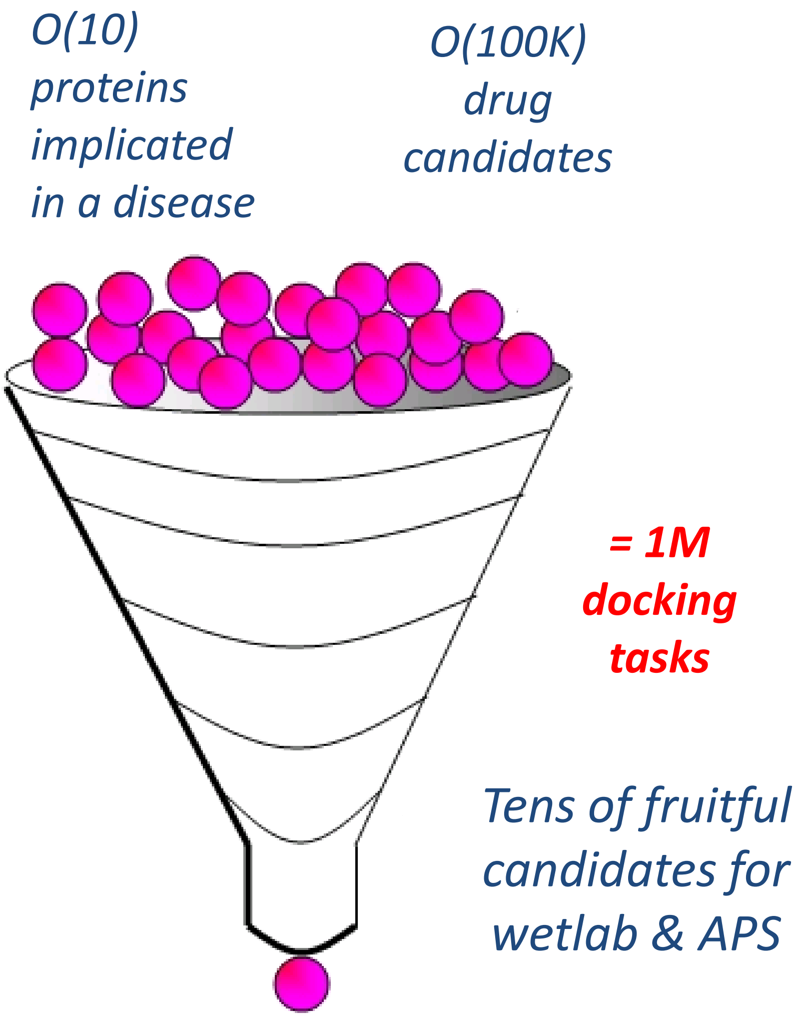
```
(int r) myproc (int i, int j)
{
    int f = F(i);
    int g = G(j);
    r = f + g;
}
```

- `F()` and `G()` implemented in native code or external programs
- `F()` and `G()` run in concurrently in different processes
- `r` is computed when they are both done

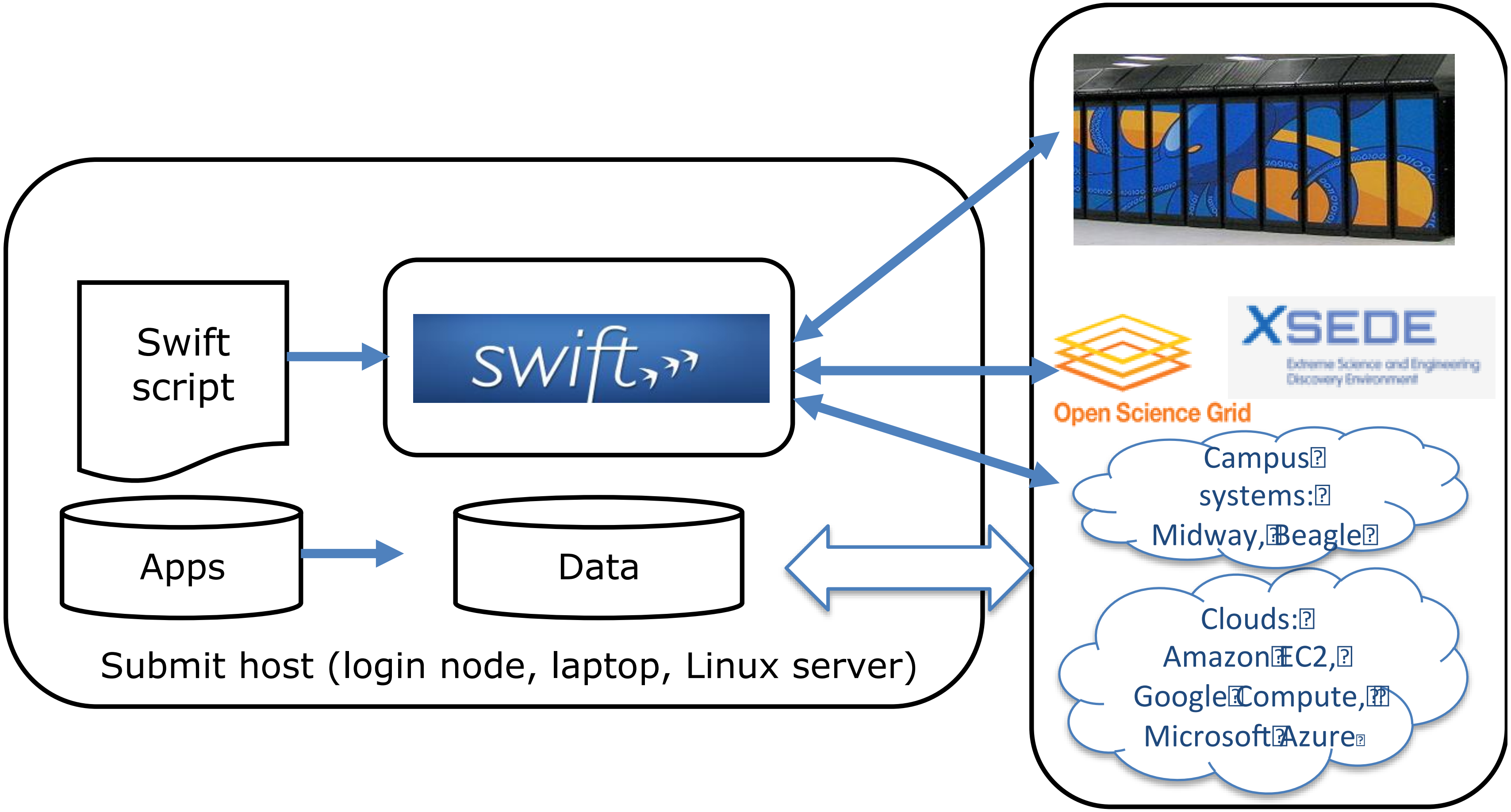
- This parallelism is *automatic*
- Works recursively throughout the program's call graph

More concurrency: Loops and arrays

```
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)
```



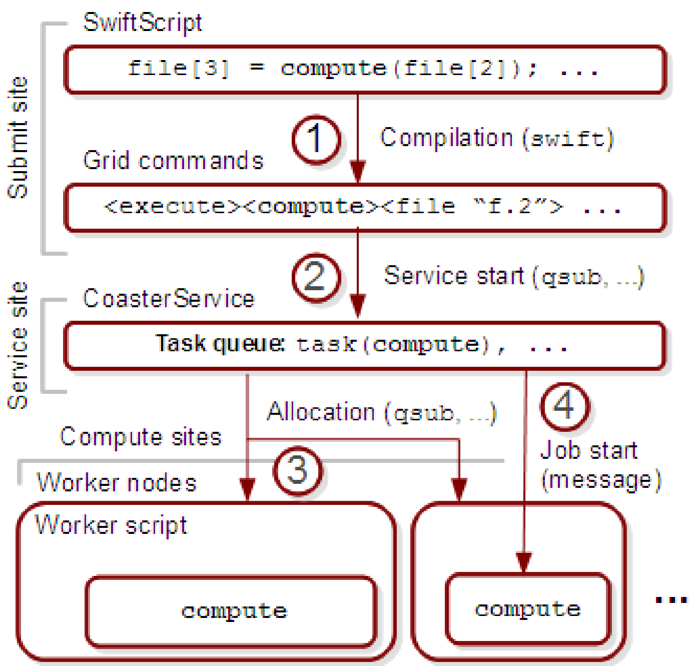
Swift/K: Swift for clusters, clouds, and grids



- Wilde et al. Swift: A language for distributed parallel scripting. Parallel Computing 37(9), 2011.

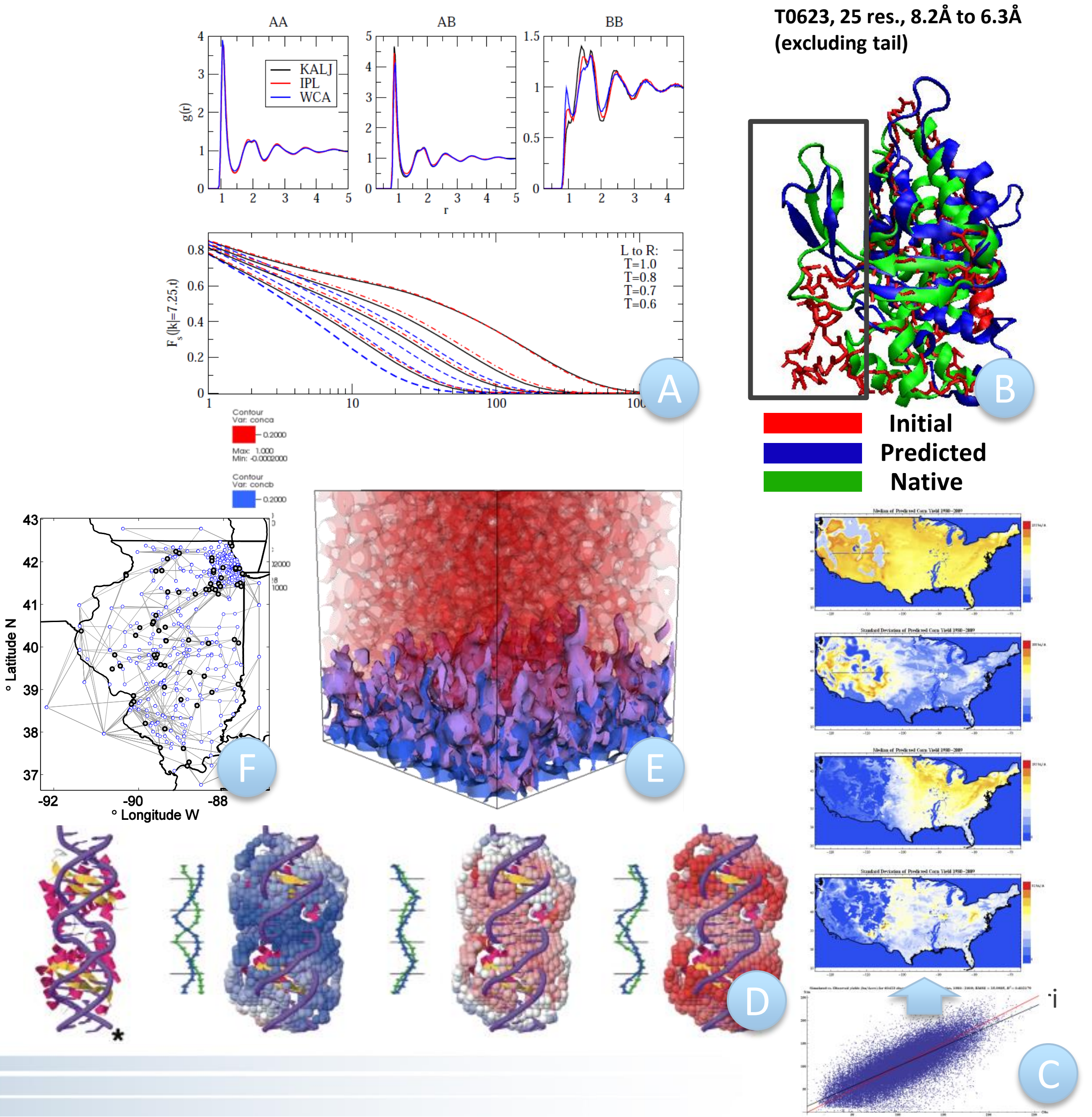
Execution infrastructure - Coasters

- Coasters: a high task rate execution provider
  (Previously developed for the Swift system)
  - Automatically deploys worker agents to resources with respect to user task queues and available resources
  - Implements the Java CoG provider interfaces for compatibility with Swift and other software
  - Currently runs on clusters, grids, and HPC systems
  - Can move data along with task submission
  - Contains a “block” abstraction to manage allocations containing large numbers of CPUs
  - **Originally only supported sequential tasks**



Large-scale many-task applications using Swift

- Simulation of metals under stress
- Molecular dynamics: NAMD
- Molecular dynamics: LAMMPS
- X-ray scattering data aggregation
- X-ray imaging analysis
- Multiscale subsurface flow modeling
- Modeling of the power grid
- Climate data extraction
- … and many more

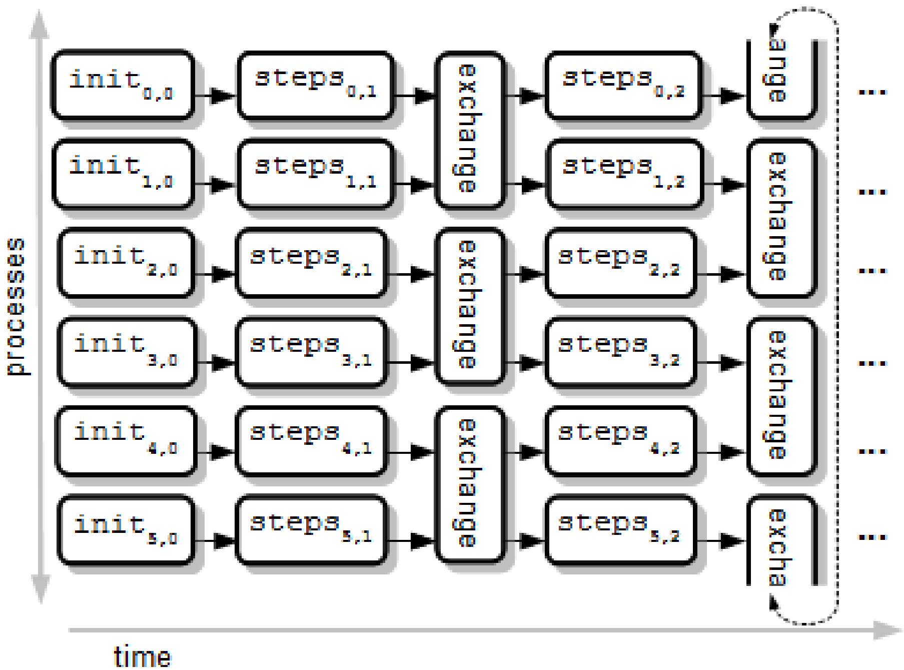


SWIFT/K: MPI TASKS

NAMD - Replica Exchange Method

- Original JETS use case- sizeable batch of short parallel jobs with data exchange

- Method extracts information about a complex molecular system through an ***ensemble*** of concurrent, parallel simulation tasks

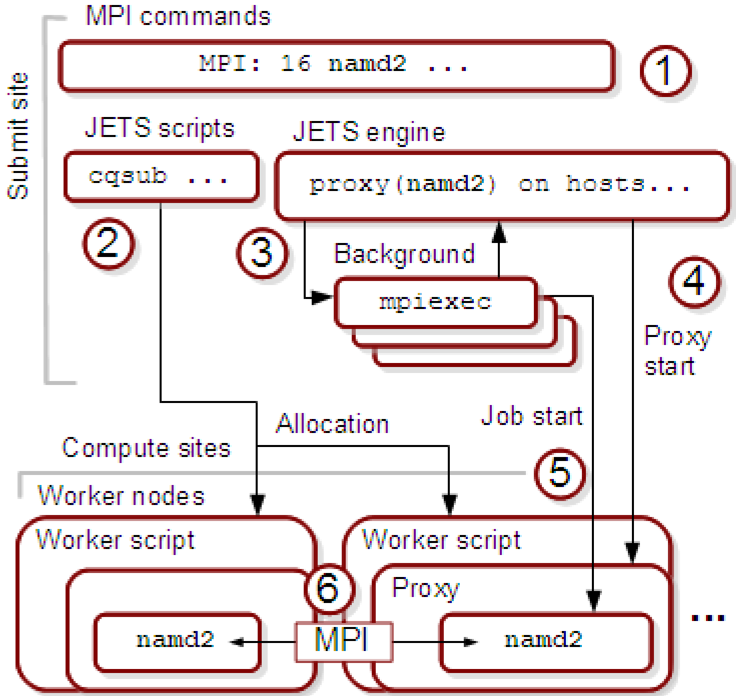


Application parameters (approx.):

- 64 concurrent jobs
  x 256 cores per job =
  16,384 cores
- 10-100 time steps per job =
  10-60 seconds wall time
- Requires 6.4 MPI executions/sec. →
  1,638 processes/sec. over
  a 12-hour period =
  70 million process starts

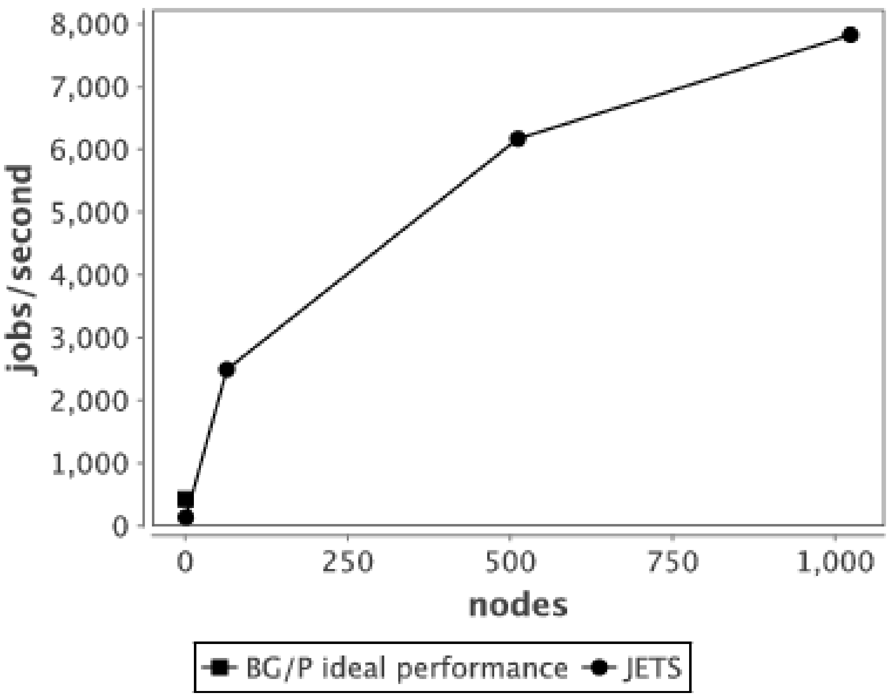
Execution infrastructure - JETS

- Stand-alone JETS: a high task rate parallel-task launcher
  - User deploys worker agents via customizable, provided submit scripts
  - Currently runs on clusters, grids, and HPC systems
    - Great over SSH
    - Ran on the BG/P through ZeptoOS sockets- great for debugging, performance studies, ensembles
  - Faster than Coasters but provides fewer features
    - Input must be a flat list of command lines
    - Limited data access features

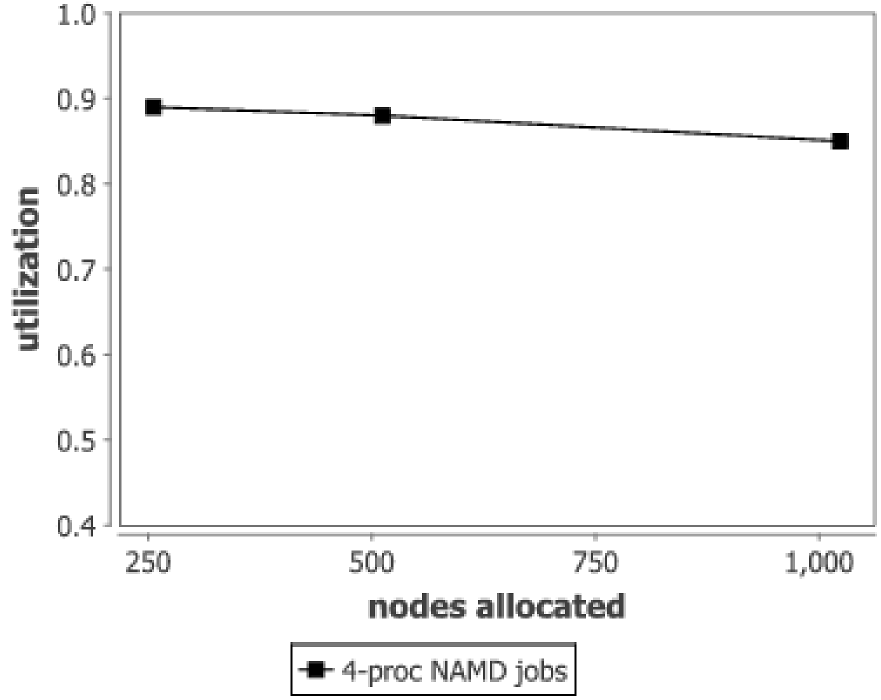


JETS - Task rates and utilization

- Calibration: Sequential performance on synthetic jobs:



- Utilization for REM-like case: not quite 90%



NAMD REM in Swift

- Constructed SwiftScript to implement REM in NAMD
  - Whole script ~ 100 lines
  - Intended to substitute for multi-thousand line Python script (that was incompatible with the BG/P)
- Script core structures shown to the right
- Represents REM data flow from previous slide as Swift data items, statements, and loops

```
app (positions p_out, velocities v_out,
     energies e_out)
namd(positions p_in, velocities v_in)
{
  namd @p_out @v_out @p_in @v_in stdout=@e_out;
}

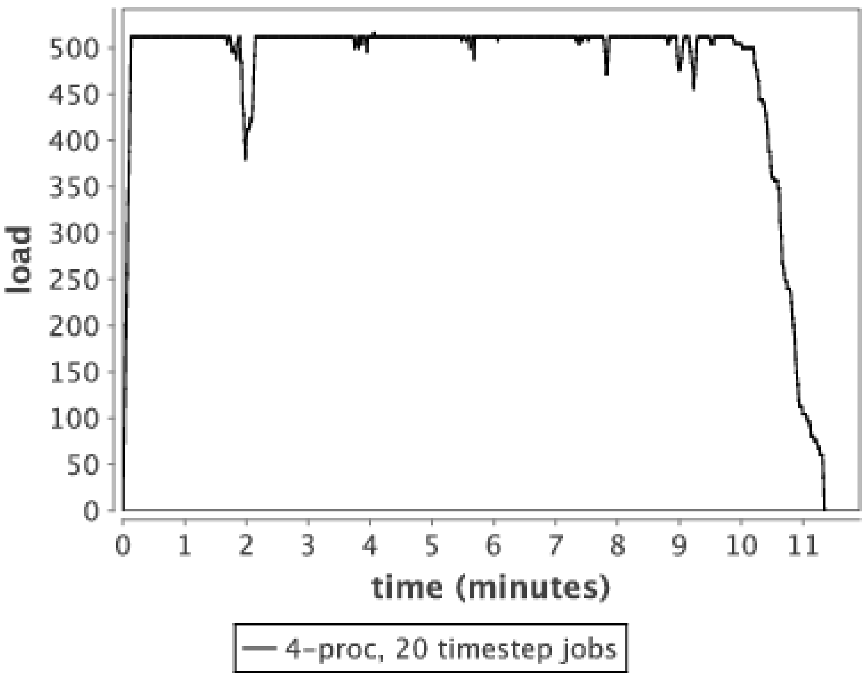
positions  p[]<array_mapper;files=p_strings>;
velocities v[]<array_mapper;files=v_strings>;
energies   e[]<array_mapper;files=e_strings>;

// Initialize first segment in each replica
foreach i in [0:replicas-1] {
  int index = i*exchanges;
  p[i] = initial_positions();
  v[i] = initial_velocities();
}

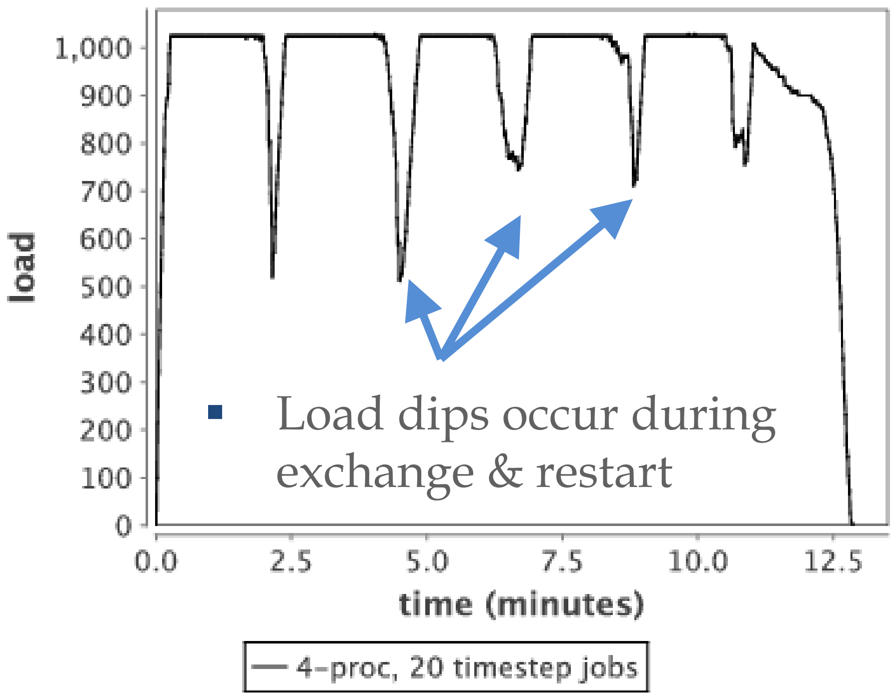
// Launch data-dependent NAMDs…
iterate j {
  foreach i in [0:replicas-1] {
    int current  = i*exchanges + j+1;
    int previous = i*exchanges + j;
    (p[current], v[current], e[current]) =
      namd(p[previous], v[previous]);
  }
} until (j == exchanges);
```

NAMD/JETS load levels

- Allocation size: 512 nodes



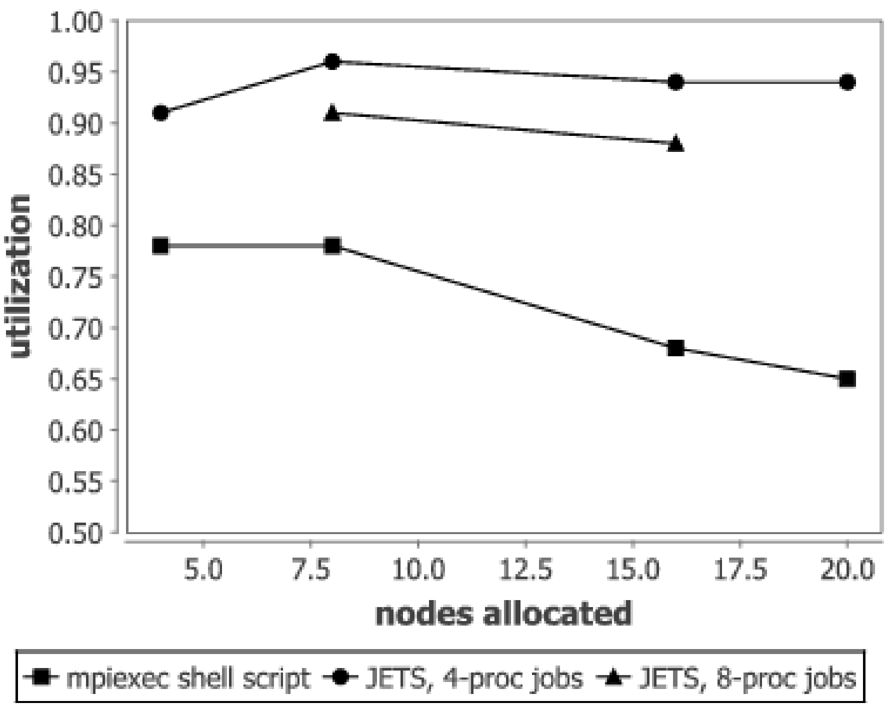
- Allocation size: 1024 nodes



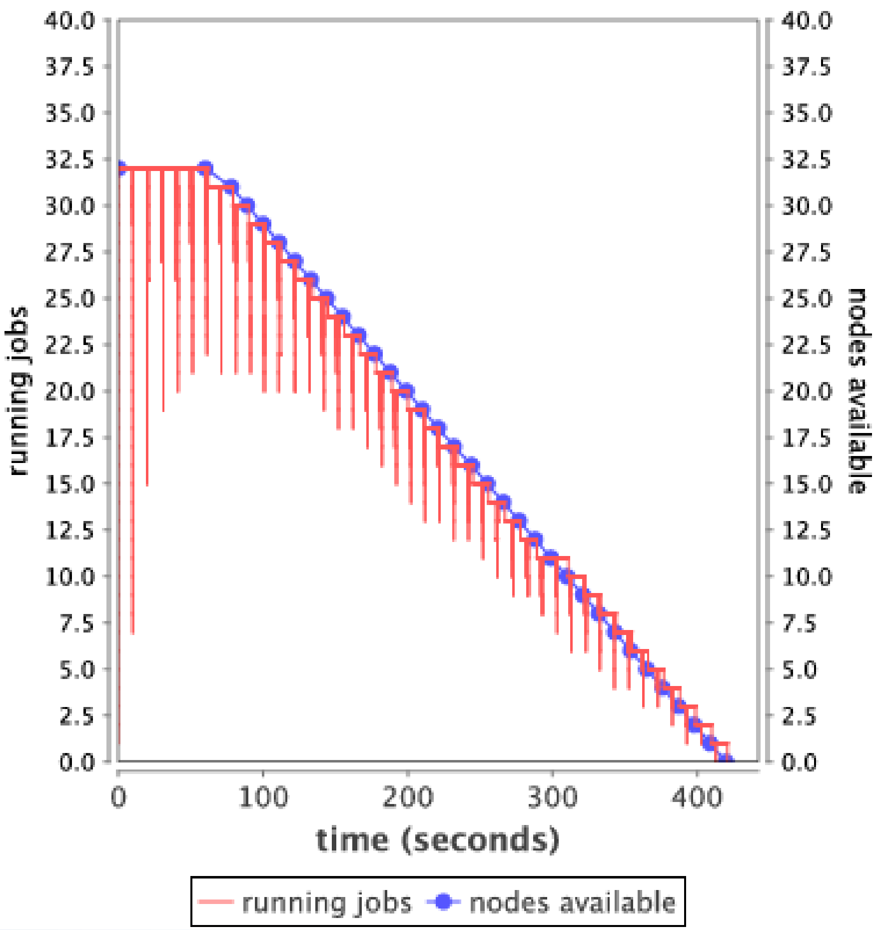
- Wozniak et al. JETS: Language and system support for many-parallel-task workflows. J. Grid Computing 11(3), 2013.

JETS - Misc. results

- Effective for short MPI jobs on clusters
- Single-second duration jobs on Breadboard cluster



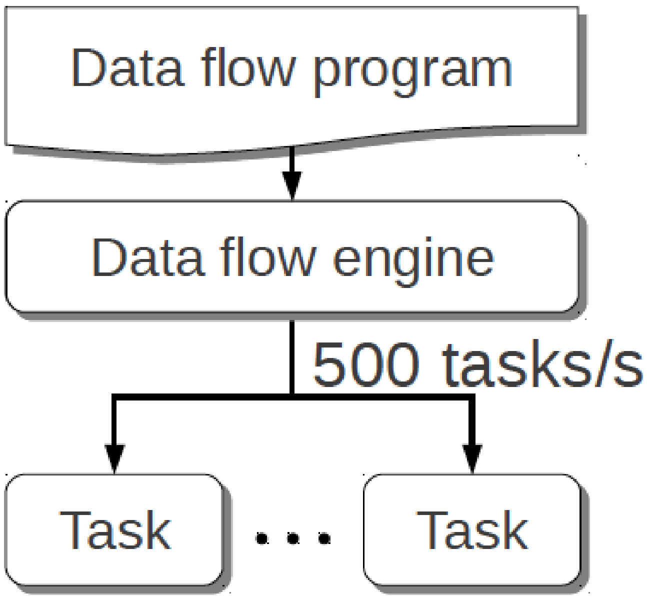
- JETS can survive the loss of worker agents (BG/P)



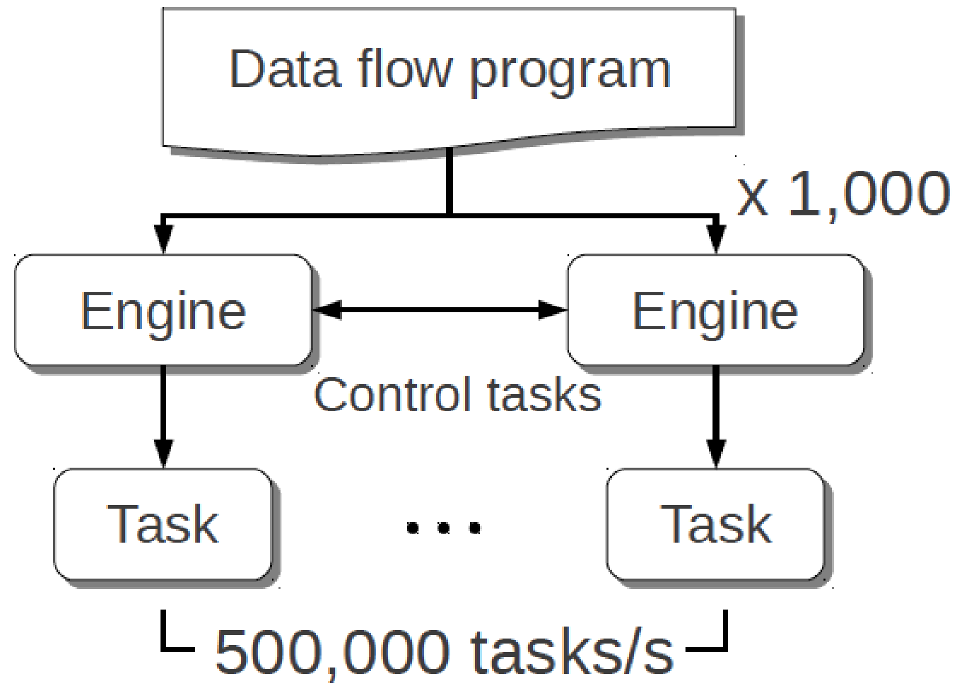
SWIFT/T OVERVIEW

Swift/T: Swift for high-performance computing

Had this:
(Swift/K)



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

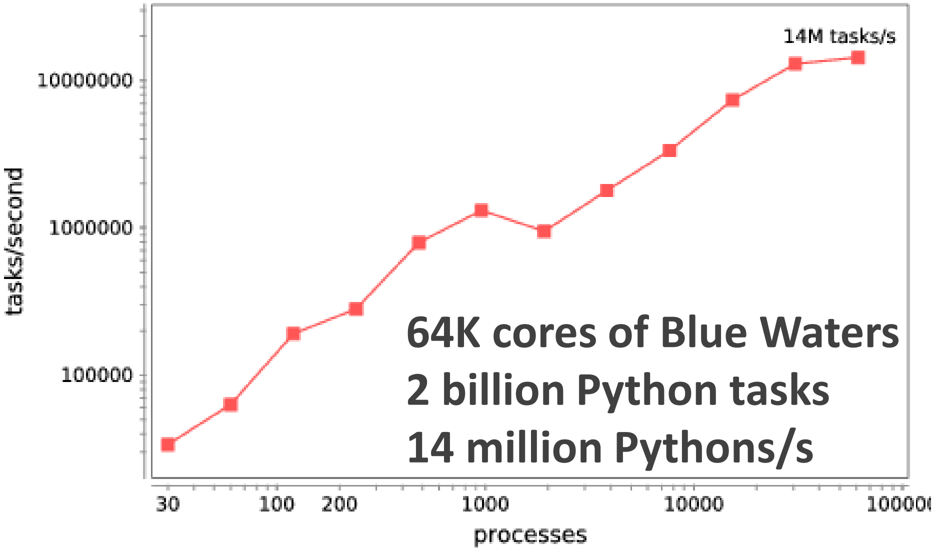


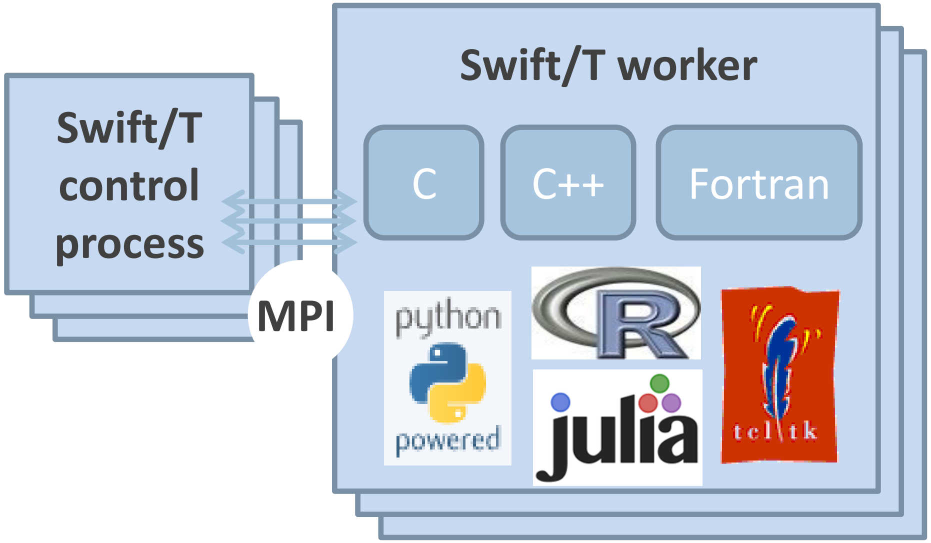
Centralized evaluation

Distributed evaluation

Swift/T: Enabling high-performance workflows

- Write site-independent scripts
- Automatic parallelization and data movement
- Run native code, script fragments as applications
- Rapidly subdivide large partitions for MPI jobs
- **Move work to data locations**



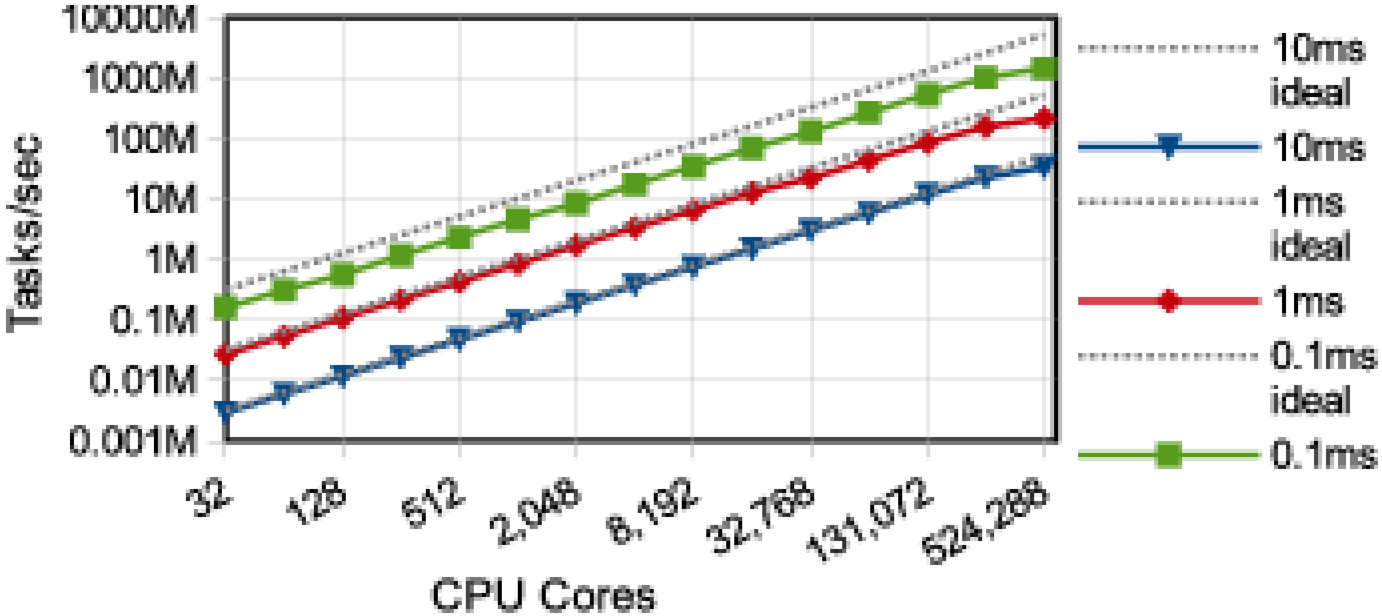


Characteristics of very large Swift programs

```
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

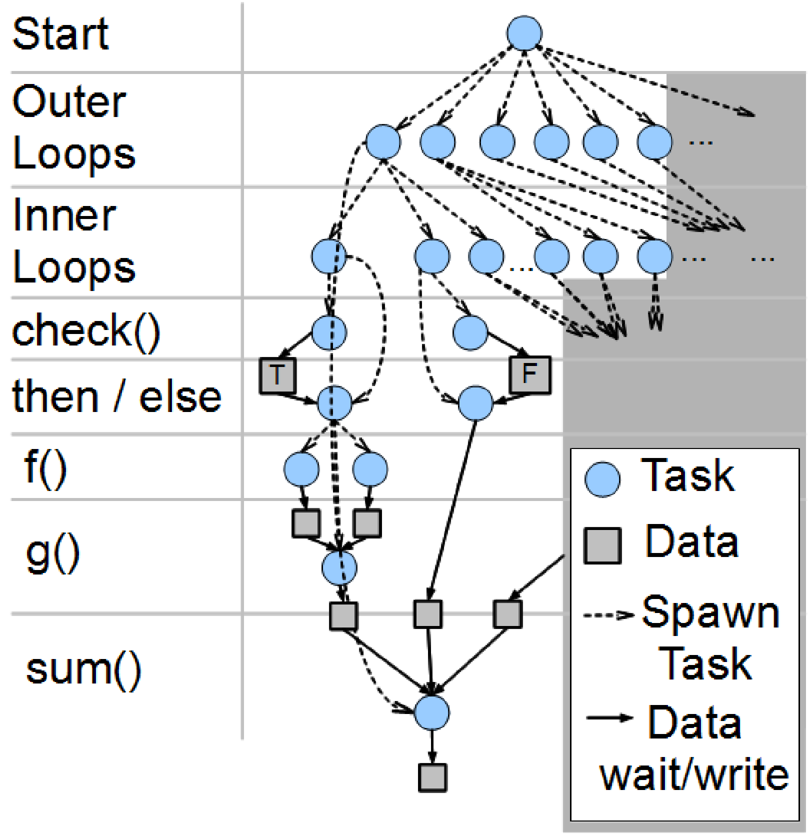
Basic scalability



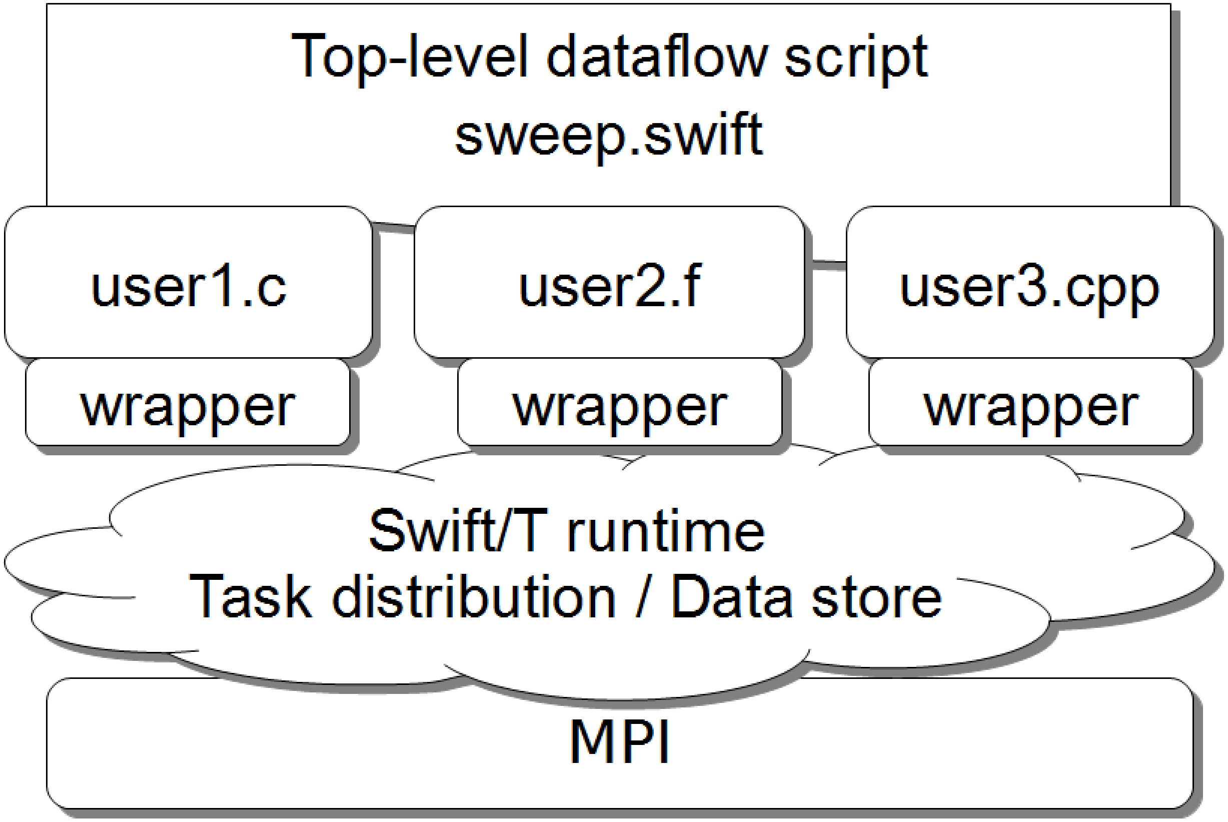
- 1.5 billion tasks/s on 512K cores of Blue Waters, so far

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]);
}
```



Support calls to native libraries



- Including MPI libraries

Example execution

- Code

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

- Engines: evaluate dataflow operations

```
• Perform getenv()
• Submit f
```

```
• Subscribe to A[2]
• Submit g
```

- Workers: execute tasks

Task put

Notification

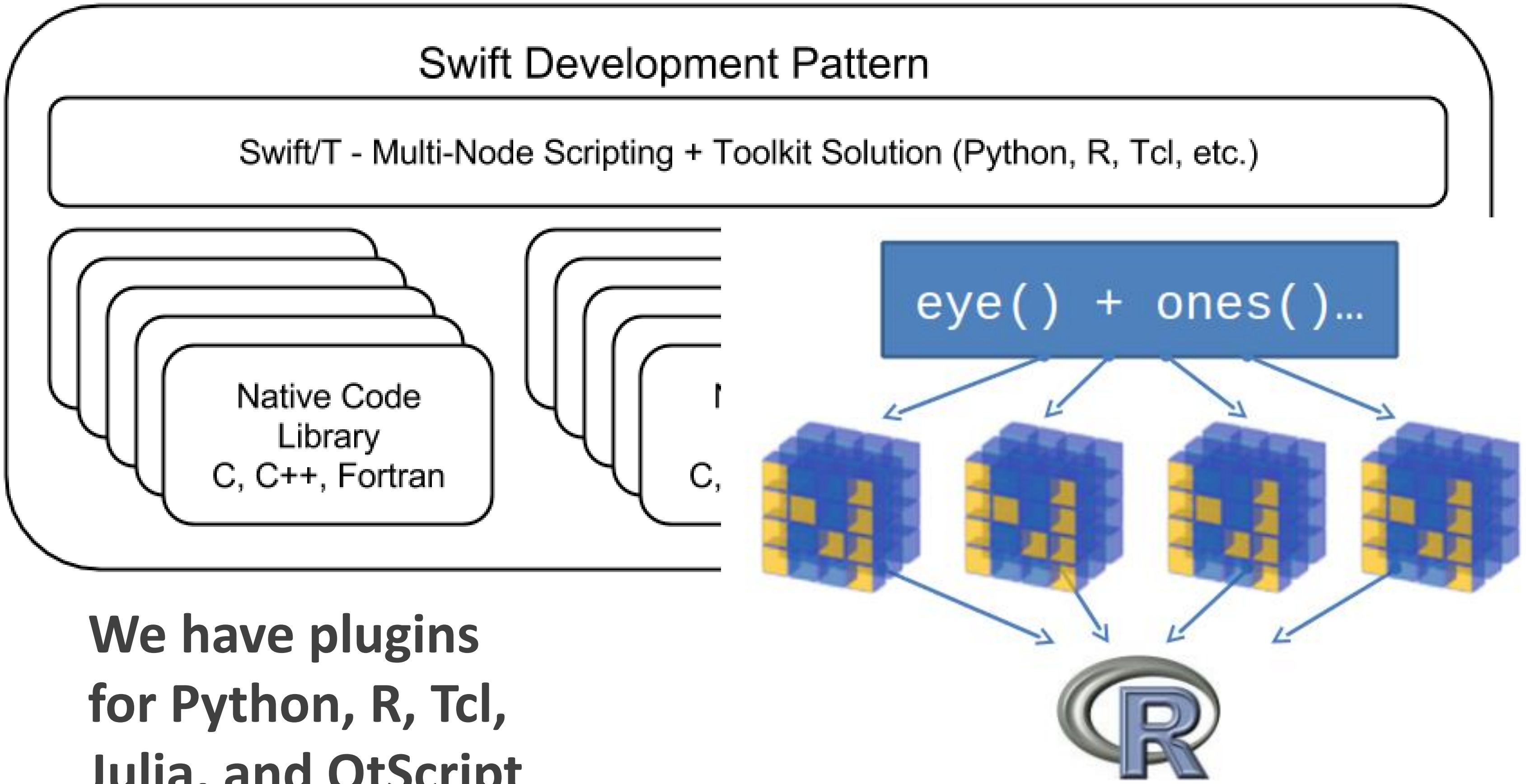
Task put

```
• Process f
• Store A[2]
```

```
• Process g
• Store A[3]
```

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

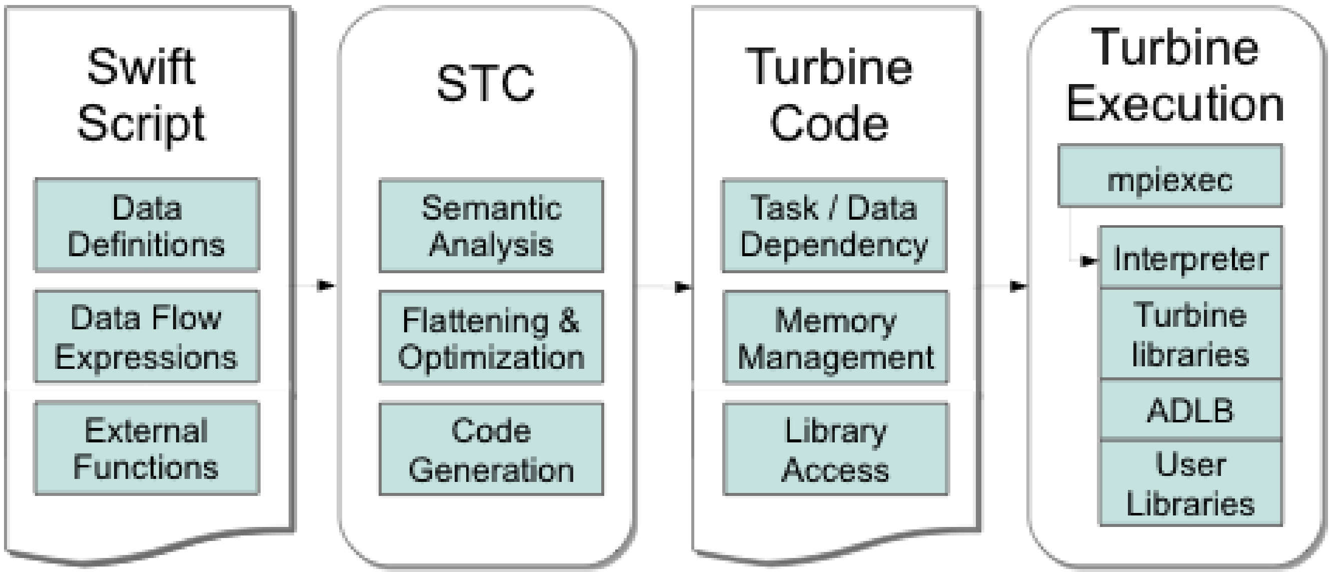
Support calls to embedded interpreters



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.

STC: The Swift-Turbine Compiler



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

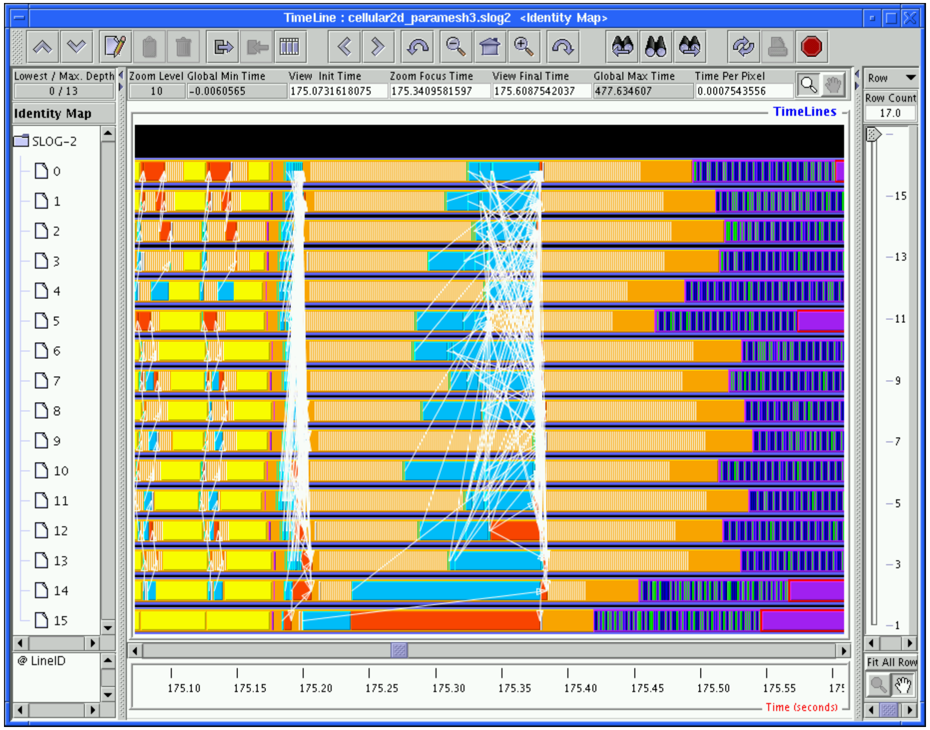
- Wozniak et al. Large-scale application composition via distributed-memory data flow processing. Proc. CCGrid 2013.
- Armstrong et al. Compiler techniques for massively scalable implicit task parallelism. Proc. SC 2014.

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

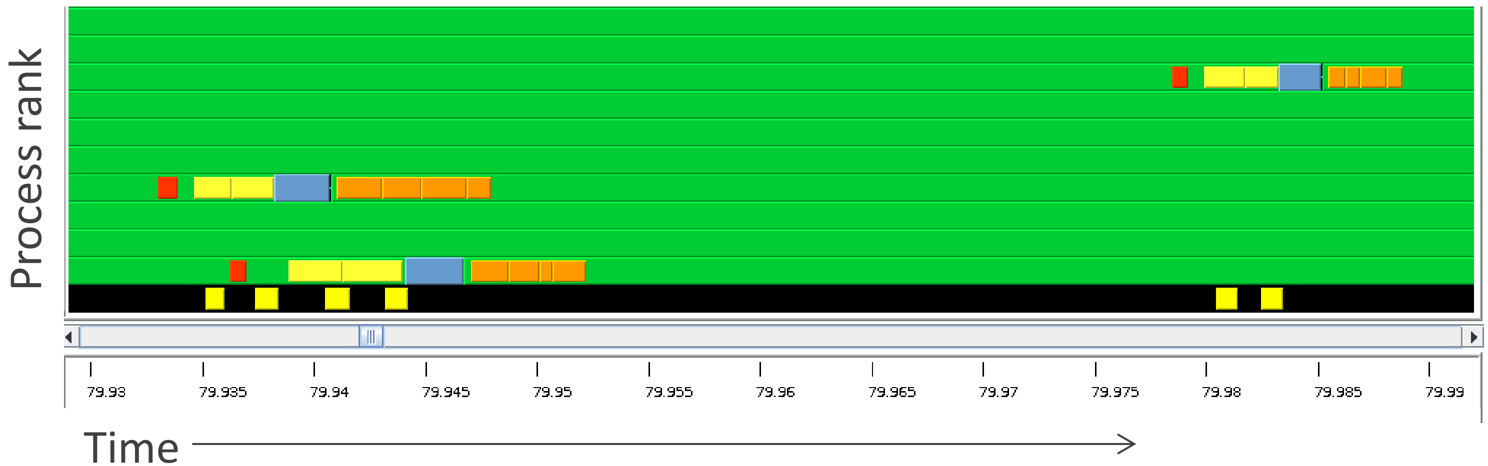
- Use MPE to log data, task operations, calls to native code
- Use MPE metadata to annotate events for later queries

- 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

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

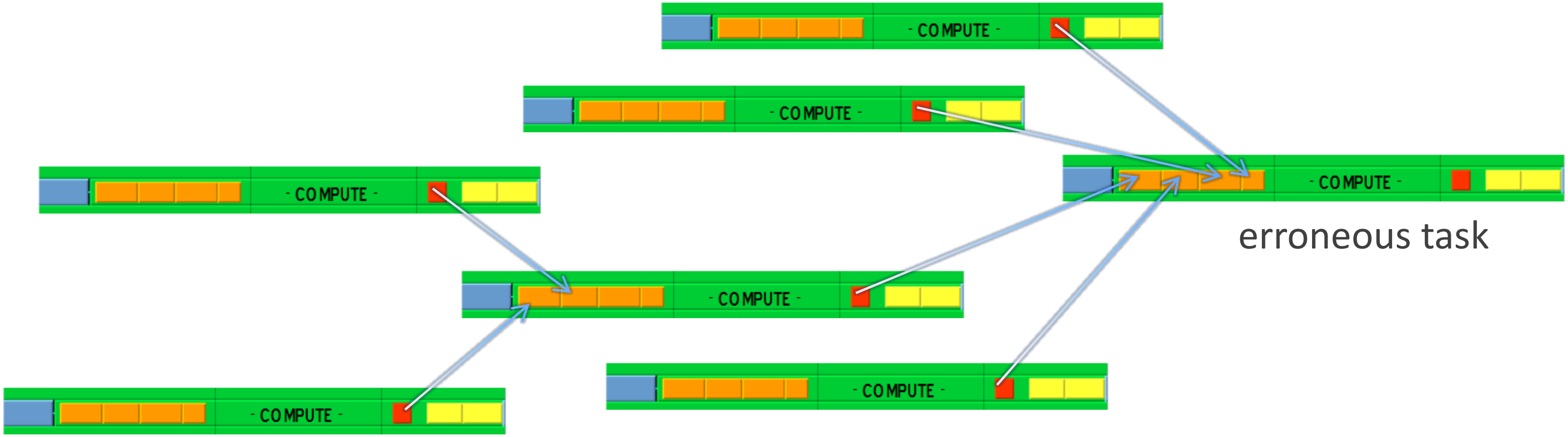
  – **PIPS task computation** **Store variable** **Notification (via control task)**
    Blue: Get next task **Retrieve variable**
    Server process (handling of control task is highlighted in yellow)

- 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



erroneous task

Aha! Found script defect.

← ← ← (searching backwards)

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

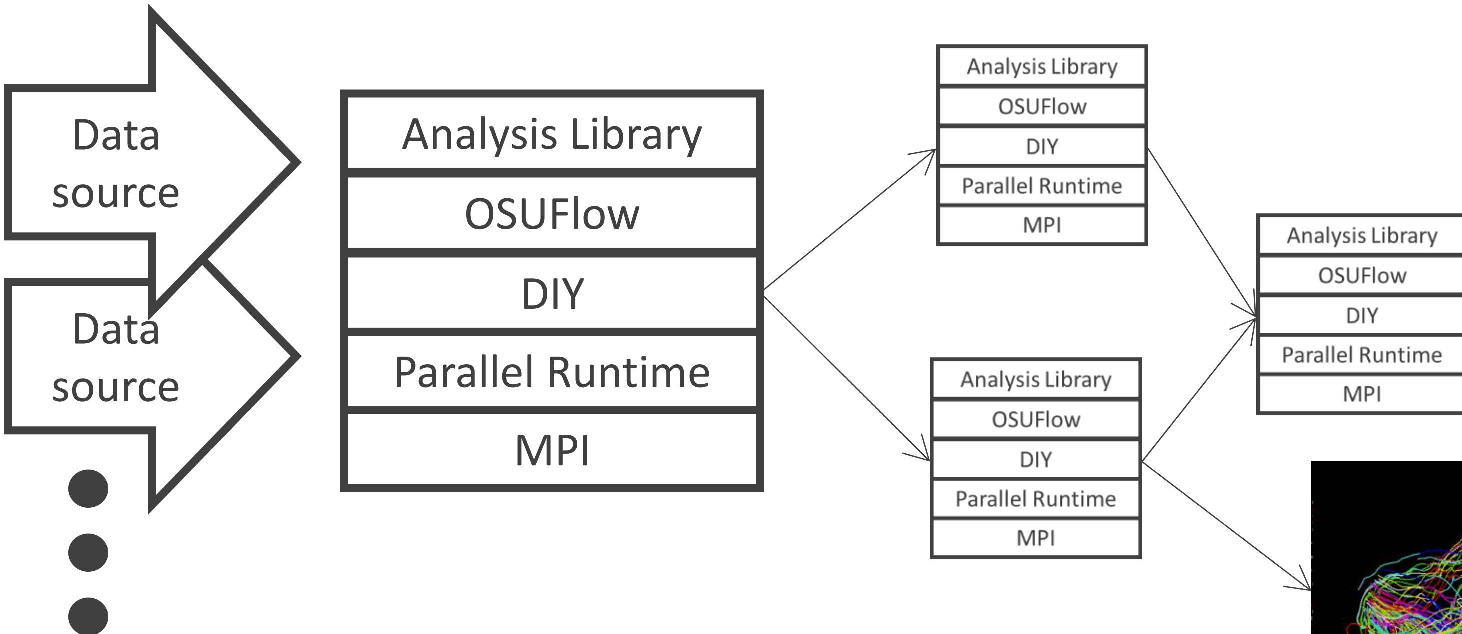
Other Swift/T features

- 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);
    ```
- 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

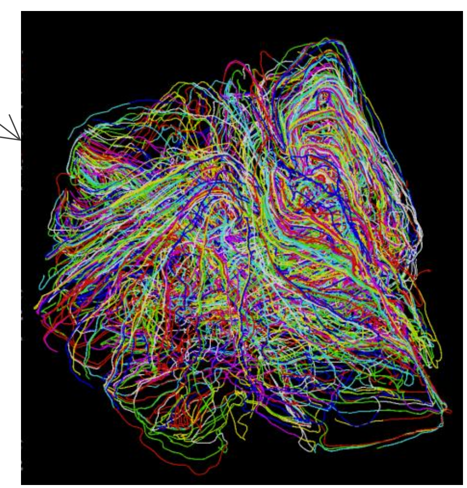
- Wozniak et al. Language features for scalable distributed-memory dataflow computing. Proc. Dataflow Execution Models at PACT, 2014.
    ```

SWIFT/T: MPI TASKS

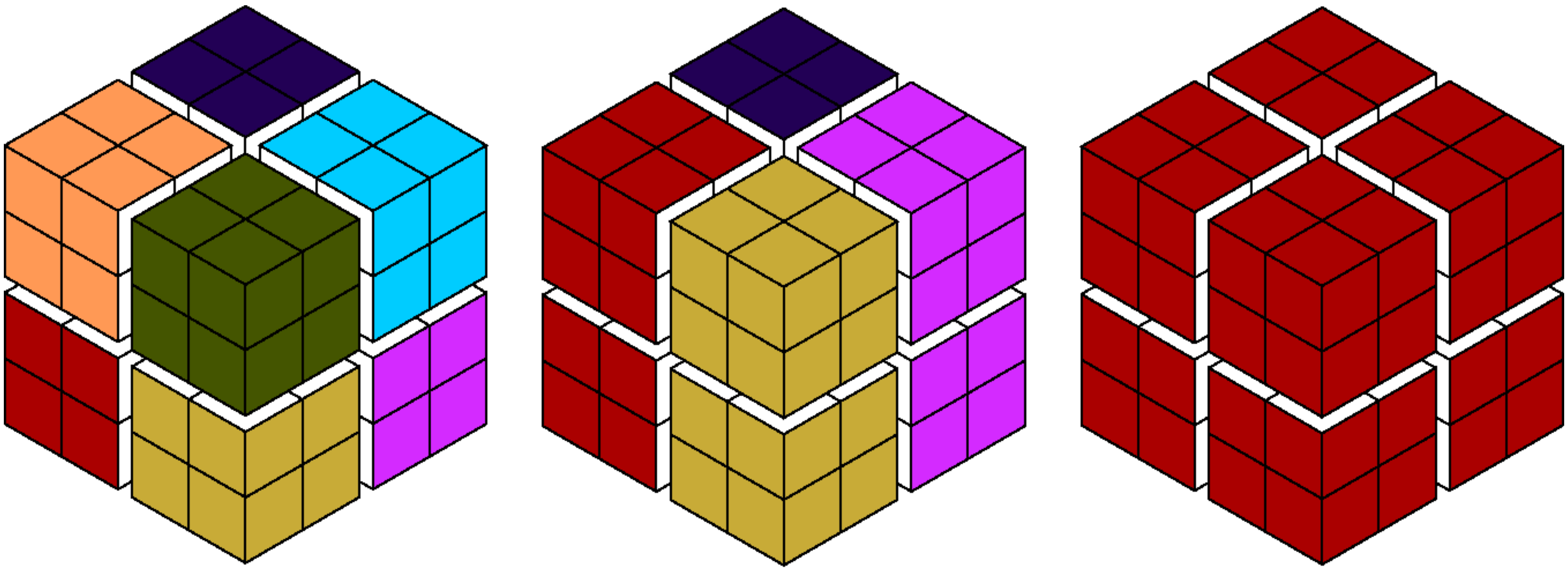
Dataflow+data-parallel analysis/visualization



Dataflow-structured analysis framework based on OSUFlow/DIY



Parameter optimization for data-parallel analysis: *Block factor*



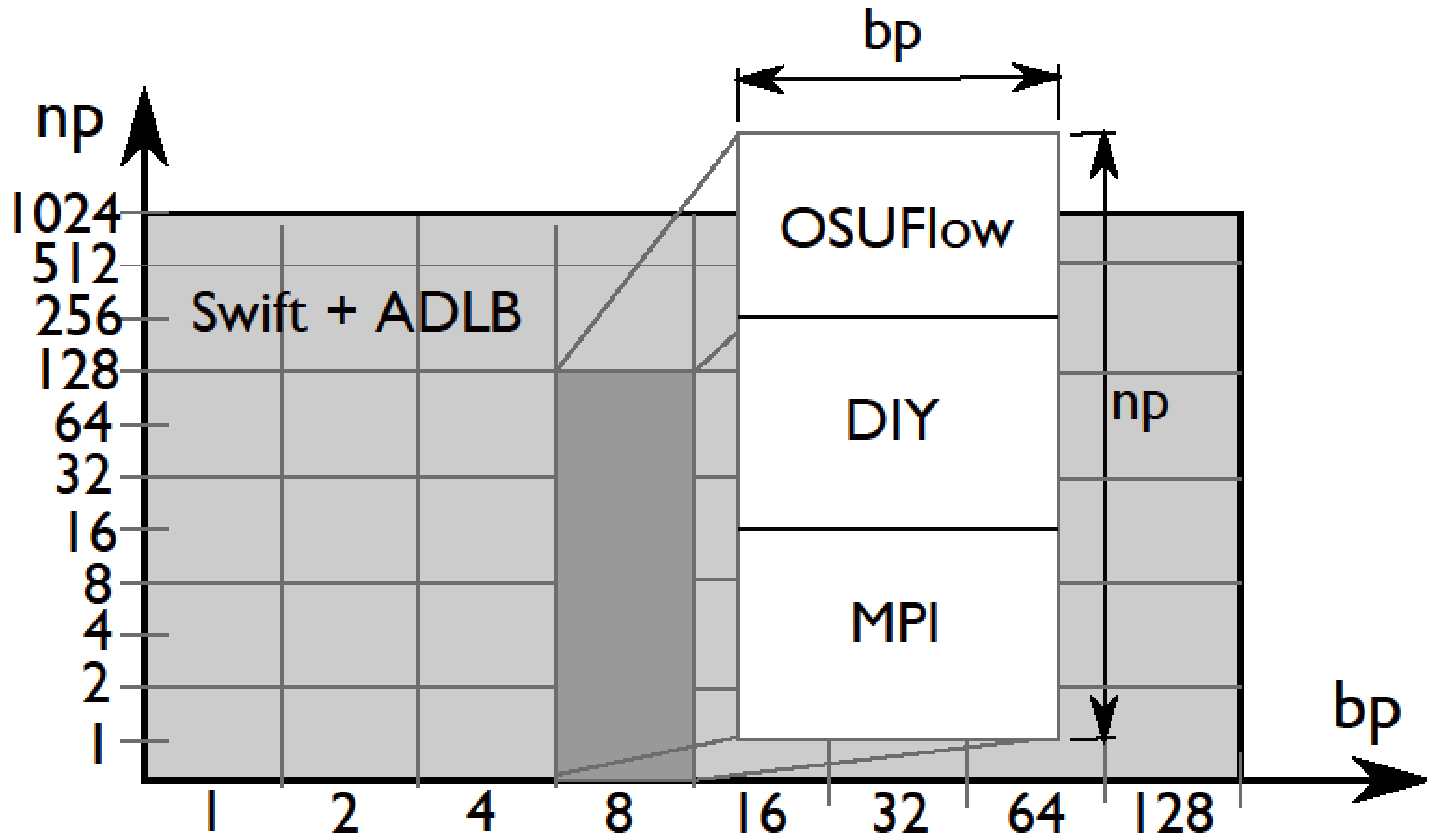
8 processes
1 block per process

4 processes
2 blocks per process

1 process
8 blocks per process

Can map blocks to processes in varying ways

Parameter optimization for data-parallel analysis: *Process configurations*



- **Try all configurations to find best performance**
- **Goal: Rapidly develop and execute sweep of MPI executions**

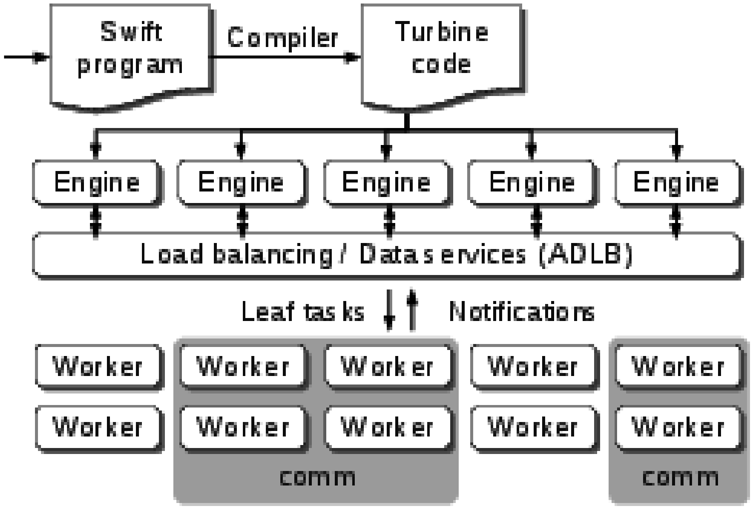
Refresher: MPI_Comm_create_group()

- In MPI 2, creating a subcommunicator was collective over the parent communicator
  - Required global coordination
  - Scalability concern
  - (Could use intercommunicator merges- somewhat slow)
- In MPI 3, the new `MPI_Comm_create_group()` allows the implementation to assemble the new communicator quickly from a group – only group members must participate
  - In ADLB, servers just pass rank list for new group to workers

- Motivating investigation by Dinan et al. identified fault tolerance and dynamic load balancing as key use cases – both relevant to Swift (Dinan et al., EuroMPI 2011.)

Parallel tasks in Swift/T

- Swift expression: `z = @par=8 f(x,y);`
- When x, y are stored, Turbine releases task `f` with `parallelism=8`
- Performs `ADLB_Put(f, parallelism=8)`
- Each worker performs `ADLB_Get(&task, &comm)`
- ADLB server finds 8 available workers
- Workers receive ranks from server
  - **Perform `MPI_Comm_create_group`**
- `ADLB_Get()` returns: `task=f, size(comm)=8`
- Workers perform user task
  - communicate on `comm`
- `comm` is released by Turbine

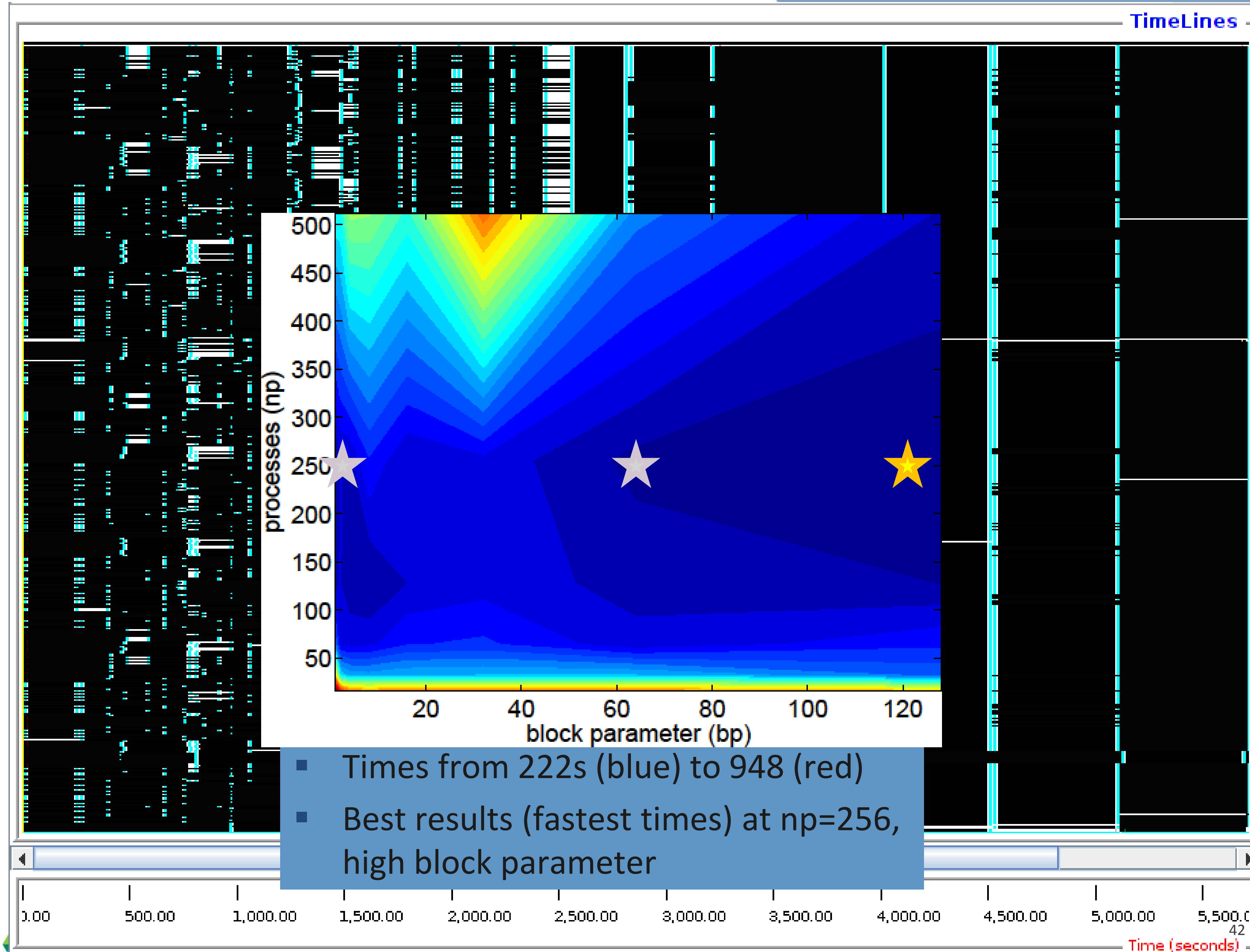


- Wozniak et al. Dataflow coordination of data-parallel tasks via MPI 3.0. Proc EuroMPI, 2013.

OSUFlow application

```
// Define call to OSUFlow feature MpiDraw
@par (float t) mpidraw(int bf) "mpidraw";

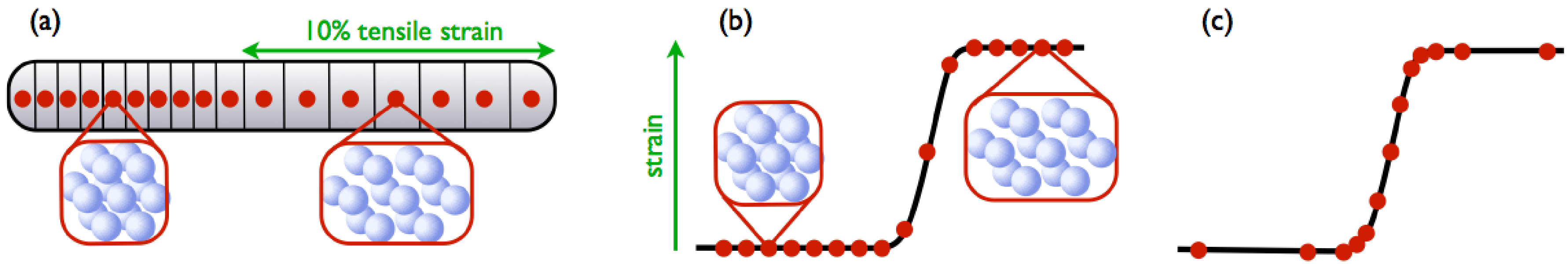
main {
  foreach b in [0:7] {
    // Block factor: 1-128
    bf = round(2**b);
    foreach n in [4:9] {
      // Number of processes/task: 16-512
      np = round(2**n);
      t = @par=np mpidraw(bf);
      printf("RESULT: bf=%i np=%i -> time=%0.3f",
                      bf,   np,      t);
    }}}
```



- Times from 222s (blue) to 948 (red)
- Best results (fastest times) at np=256, high block parameter

SWIFT/T APPLICATIONS

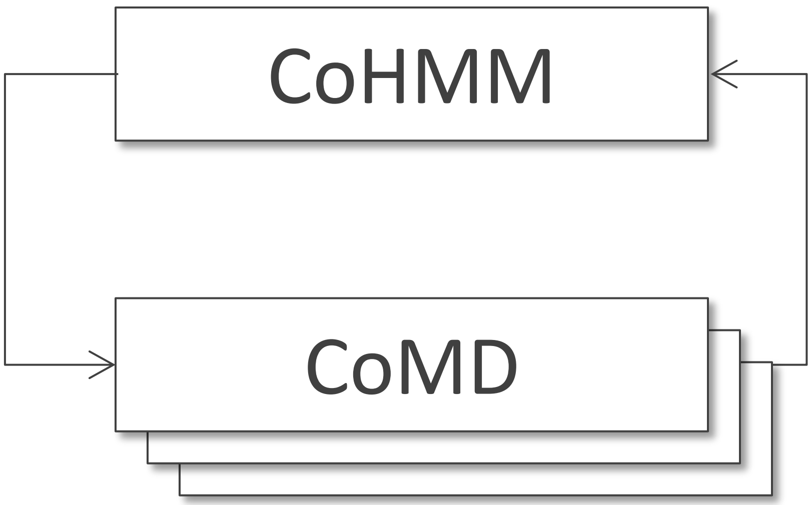
ExMatEx: Co-design for materials research



- CoHMM: Heterogeneous Multiscale Method
- CoMD: Molecular Dynamics
- Coarse-grain strain evolution using basic conservation laws
- Fine-grain molecular dynamics as necessary for physical coefficients

From http://www.exmatex.org

CoHMM/Swift



- Concurrency gained primarily by calls to CoMD

- 300 lines of sequential C
- Coordinates multiple sequential calls to CoMD
- We rewrote this in Swift

- 1000's lines of sequential C
- Simplified MD simulator
- Typically called as standalone program
- We exposed CoMD as a Swift function – no exec()

CoMD: Library access from Swift

- **CoMD binding: (example-1)**

```
string s = "-f data/8k.inp.gz";
int N = 3;
foreach i in [0:N-1] {
    float virial_stress = COMDSWIFT_runSim(s);
    printf("Swift: virial_stress: %e",
           virial_stress);
}
```

CoMD: Library access from CoHMM

C

```
#define ZERO_TEMP_COMD "../../CoMD/CoMD -x 6 -y 6 -z 6"
#ifdef ZERO_TEMP_COMD
// open pipe to CoMD
FILE *fPipe = popen(ZERO_TEMP_COMD,"r");
if (fPipe == NULL) {
       …
```

Swift

```
#define ZERO_TEMP_COMD "../../CoMD/CoMD -x 6 -y 6 -z 6"
#ifdef ZERO_TEMP_COMD
    string command = ZERO_TEMP_COMD;
    stressXX = COMDSWIFT_runSim(command);
#else
    // Just the derivative of the zero temp energy wrt A
    stressXX = rho0*c*c*(A-1);
#endif
```

CoHMM: Translation from C to Swift: main()

C

```
int main(int argc, char **argv) {
  initializedConservedFields();
  for (i = 0; i < 100; i++) {
    for (j = 0; j < 1; j++)
      fullStep();
```

Swift

```
main {
  (A[0], p[0], e[0]) = initializedConservedFields();
  for (int t = 0; t < 5; t = t+1) {
    (A[t+1], p[t+1], e[t+1]) =
                  fullStep(A[t], p[t], e[t]);
```

CoHMM: Translation from C to Swift: call CoMD

C

```
void fluxes(double *A, double *p, double *e,
            double *f_A, double *f_p, double *f_e) {
    for (int i = 0; i < L; i++) {
        double stress = stressFn(A[i], e[i]);
        double v = p[i] / rho0;
        f_A[i] = -v;
        f_p[i] = -stress;
        f_e[i] = -stress*v;
```

Swift

```
(float f_A[], float f_p[], float f_e[])
fluxes(float A[], float p[], float e[]) {
  foreach i in [0:L-1] {
        float stress = stressFn(A[i], e[i]);
        float v = p[i] / rho0;
        f_A[i] = -v;
        f_p[i] = -stress;
        f_e[i] = -stress*v;
```

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 ];

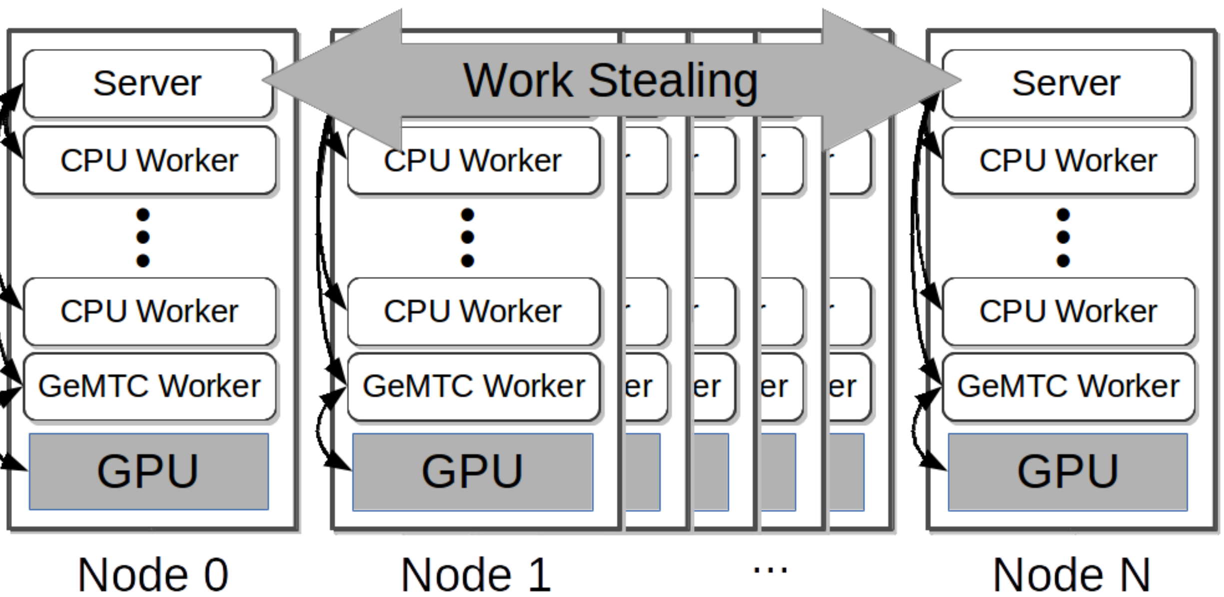
  object_file o<my_object> = gcc(c, cflags);
  object_file objects[] = [ o ];
  string ldflags[] = [];
  // Link the program
  x_file x<my_executable> = ld(objects, ldflags);
  // Run the program
  output_file out<my_output> = run(x);
  // Extract the run time from the program output
  timing_file t<my_time>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

Swift Use of GPUs

GeMTC: GPU-enabled Many-Task Computing

Approach:

1) Deploy kernel to manage GPU warps
2) Manage memory
3) Integrate with workflow system (Swift/T)



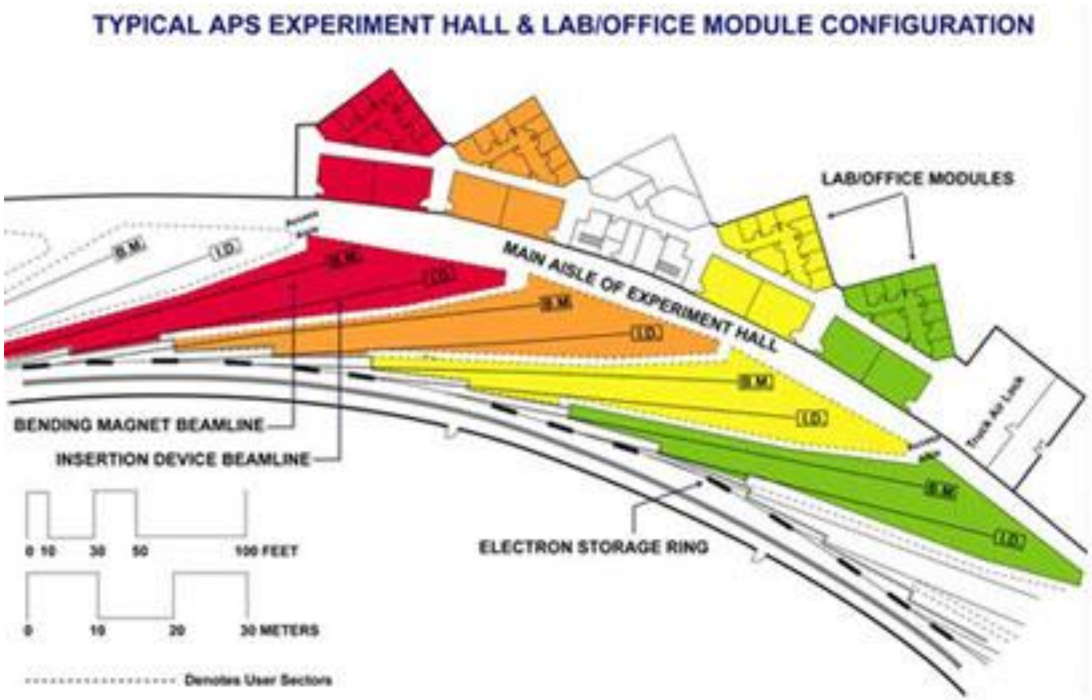
- Krieder et al. Evaluation of Many-Task Computing on Accelerators for High-End Systems. Proc. HPDC 2014.

DISCOVERY ENGINES LDRD: WORKFLOWS


Advanced Photon Source (APS)

Advanced Photon Source (APS)

- Moves electrons at electrons at >99.999999% of the speed of light.
- Magnets bend electron trajectories, producing x-rays, highly focused onto a small area
- X-rays strike targets in 35 different laboratories – each a lead-lined, radiation-proof experiment station

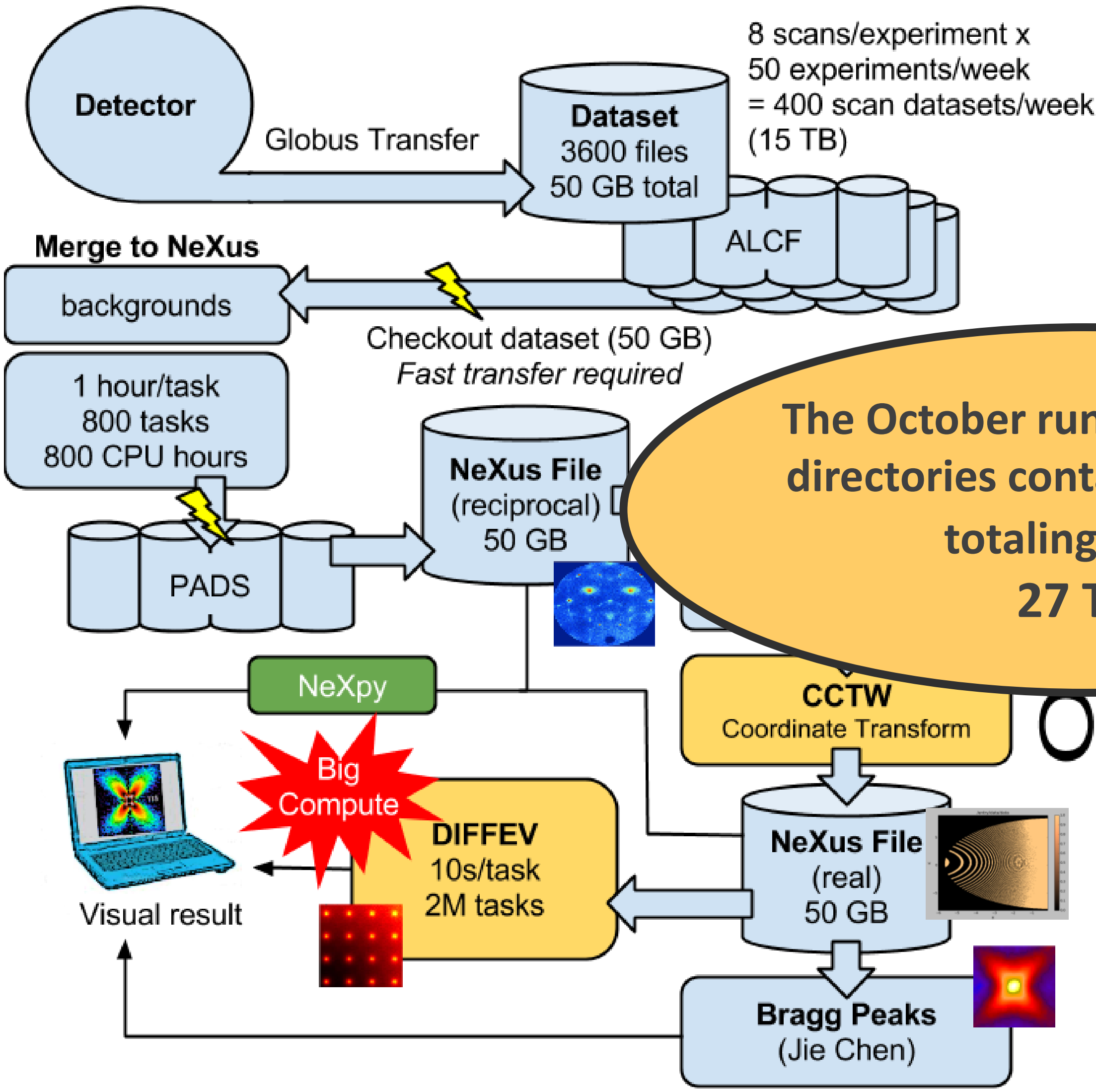


Data management for the energy sciences

- "Despite the central role of **digital data** in Dept. of Energy (DOE) research, the methods used to manage these data and to support the information and **collaboration processes** that underpin DOE research are often **surprisingly primitive**…"
  - *DOE Workshop Report on Scientific Collaborations (2011)*

- Our goals:
  - Modify the operating systems of APS stations to allow real-time streaming to a novel data storage/analysis platform.
  - Converting data from the standard detector formats (usually TIFF) to HDF5 and adding metadata and provenance, based on the NeXus data format.
  - Rewrite analysis operations to work in a massively parallel environment.
  - Scale up simulation codes that complement analysis.

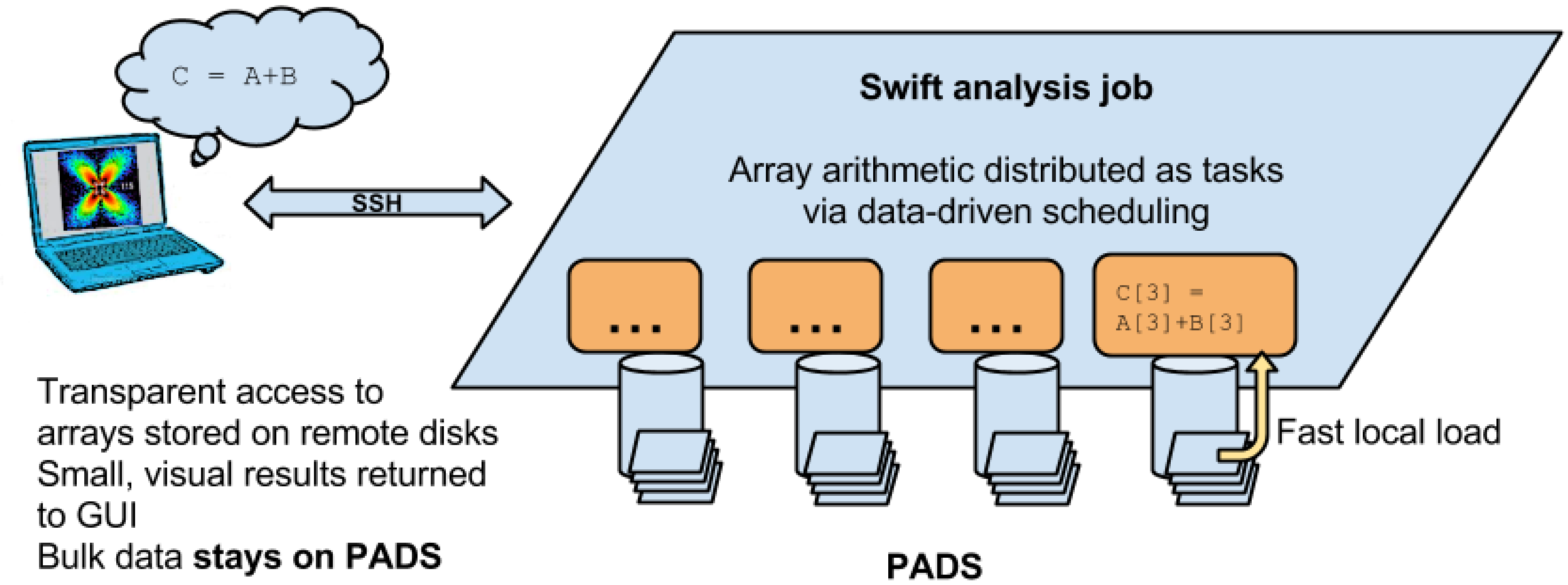
Data ingest/analysis/archive



PADS: Petascale Active Data Store

- 23 higher-end nodes for data-intensive computing, repurposed for this work (installed in 2009)
  - Each node has 12-way RAID for very fast local disk operations
- Previously, difficult to use as “Active Data Store”
  - Difficult to access specific nodes through PBS scheduler
  - No catalog (where is my data?)
  - *No way to organize/access Data Store!*
- Solution: Swift/T
  - Organizes distributed data using Swift data structures and mappers
  - Leaves data on nodes for later access
  - Allows for targeted tasks (can send work to node with data chunk)
  - Integrates with Globus Catalog for metadata, provenance, archive...
  - Combining unscheduled resource access with high performance data rates will allow for **real-time beamline data analysis, accelerating progress for materials science efforts**

Interactive analysis powered by scalable storage

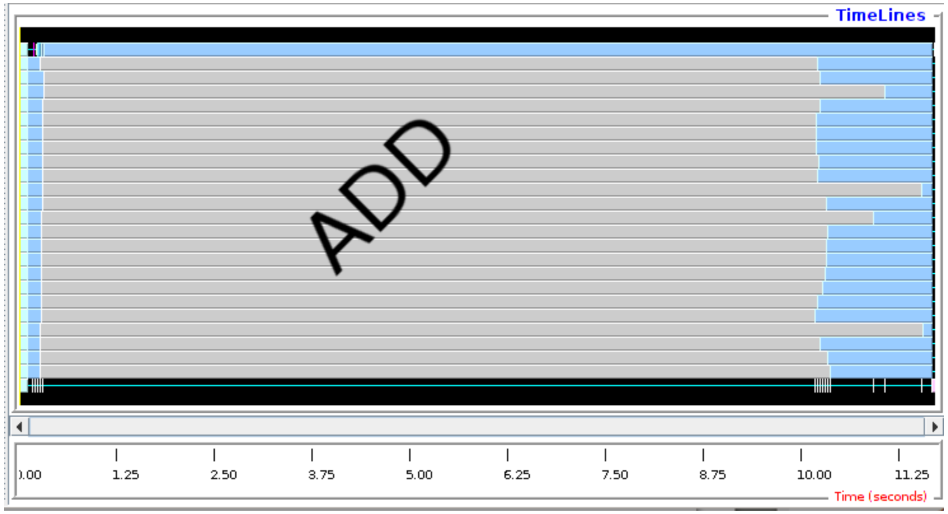


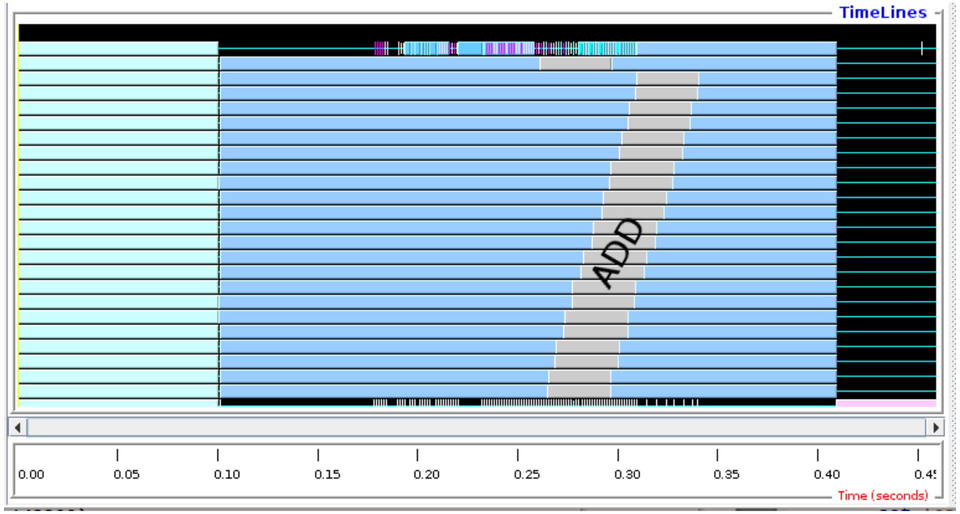
- Transparent access to arrays stored on remote disks
- Small, visual results returned to GUI
- Bulk data **stays on PADS**

- Replace GUI analysis internals with operations on remote data

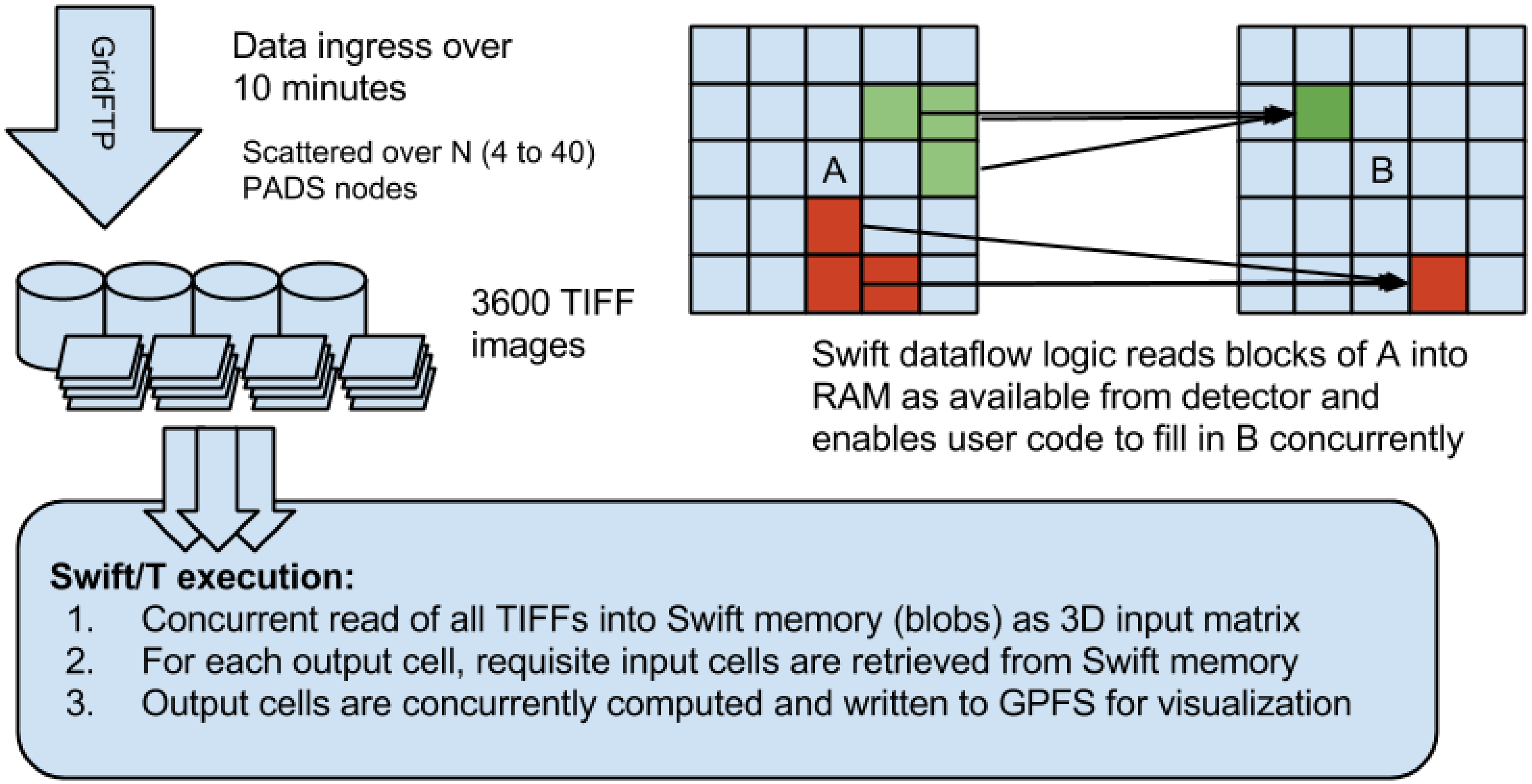
Remote matrix arithmetic: Initial results

- Initial run shows performance issue: addition took too long
- Swift profiling isolated issue: convert addition routine from script to C function: obtained 10,000 X speedup
- Swift/T integrates with MPE/Jumpshot and other MPI-based performance analysis techniques





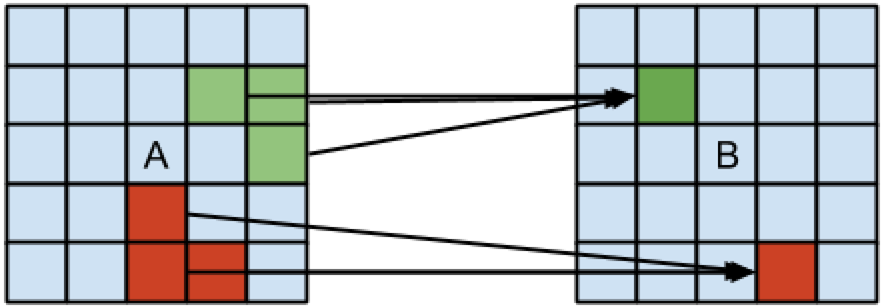
Crystal Coordinate Transformation Workflow



MapReduce-like pattern expressed elegantly in Swift

CCTW: Swift/T application (C++)

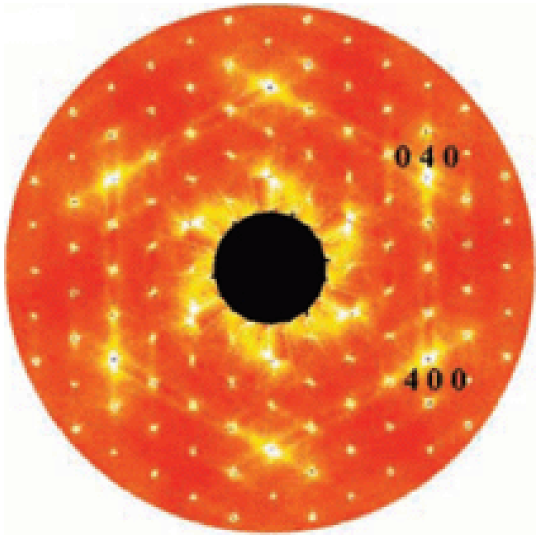
```
bag<blob> M[];
foreach i in [1:n] {
    blob b1= cctw_input("pznpt.nxs");
    blob b2[];
    int outputId[];
    (outputId, b2) = cctw_transform(i, b1);
    foreach b, j in b2 {
        int slot = outputId[j];
        M[slot] += b;
    }}
    foreach g in M {
        blob b = cctw_merge(g);
        cctw_write(b);
    }}
```



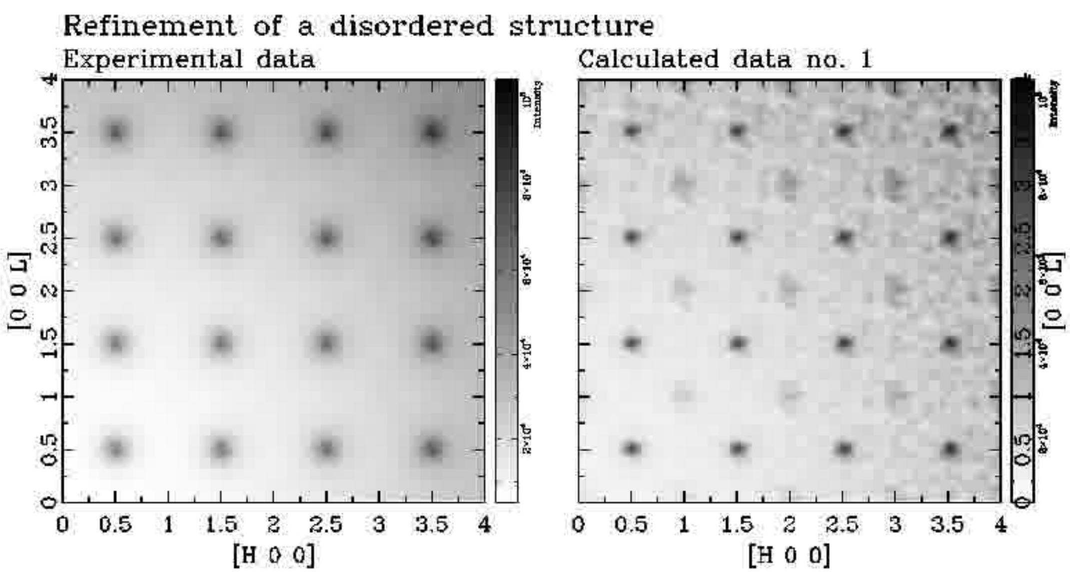
Diffuse scattering and crystal analysis

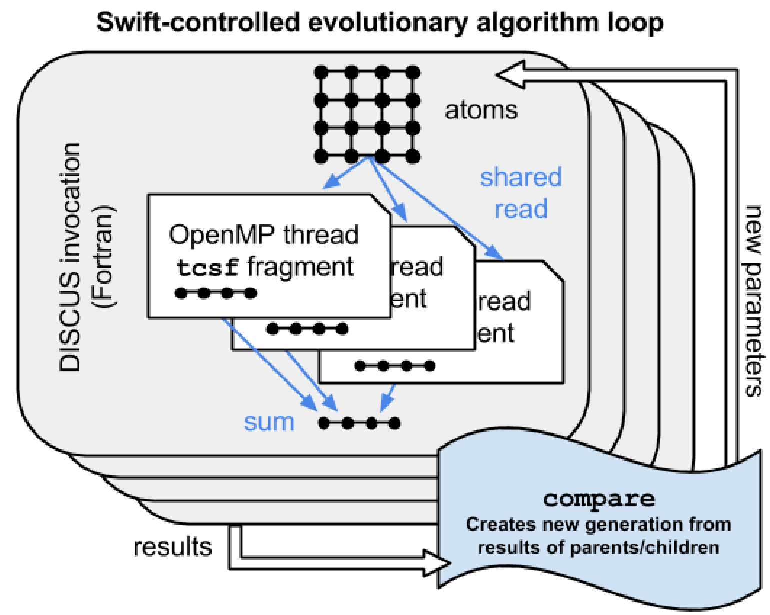
- DISCUS is a general program to generate disordered atomic structures and compute the corresponding experimental data such as single crystal diffuse scattering (http://discus.sourceforge.net)
- Given experimental data, can we fit a modeled crystal to the measurement?

- Experimental image: (Billinge, 2006)



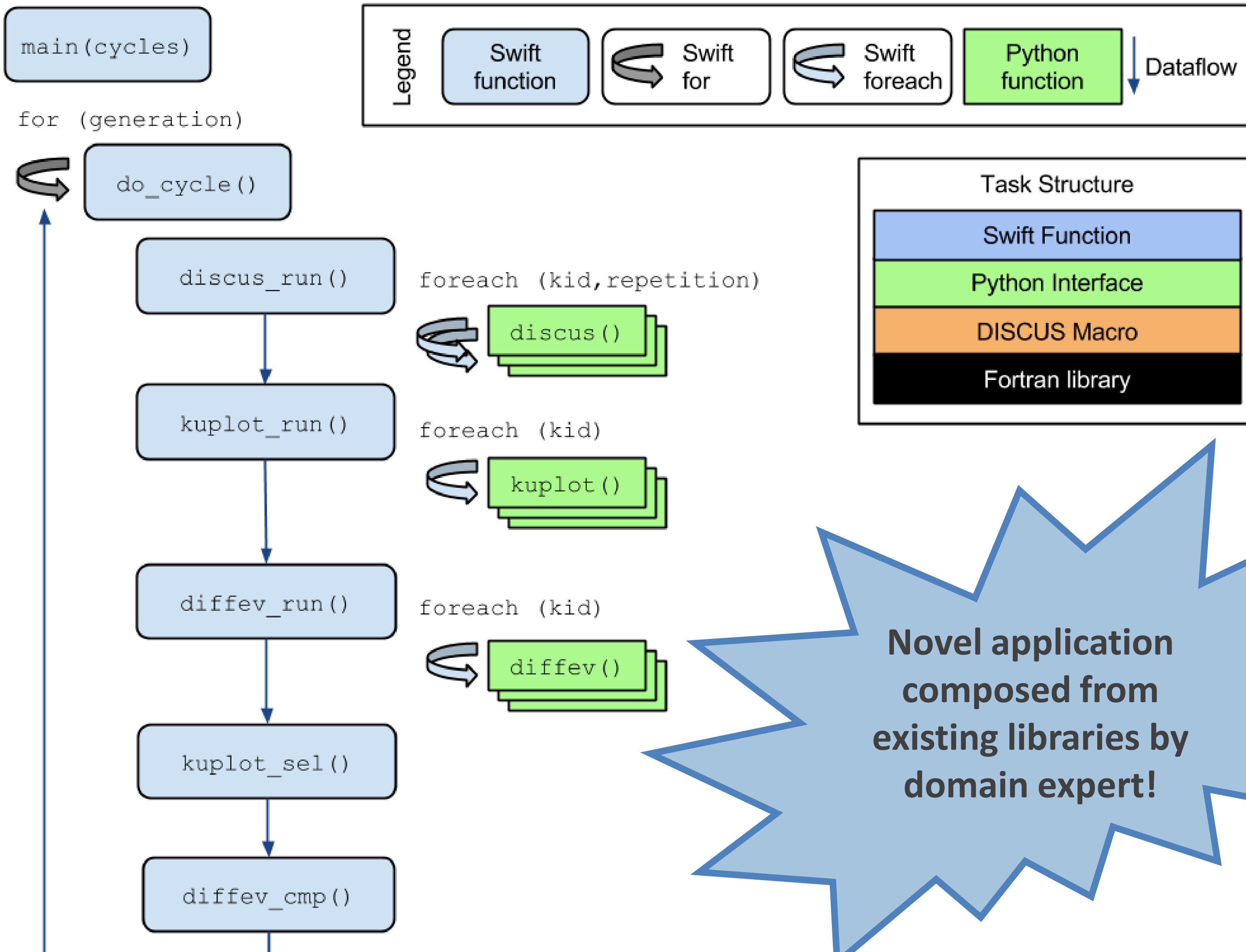
DIFFEV: Scaling crystal diffraction simulation





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

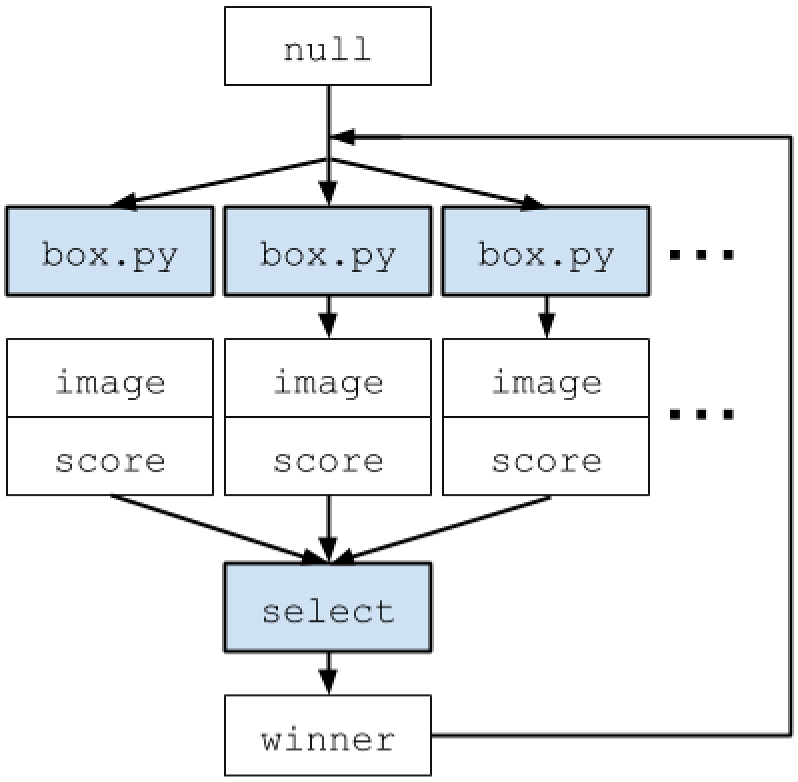
DIFFEV: Genetic algorithm via dataflow



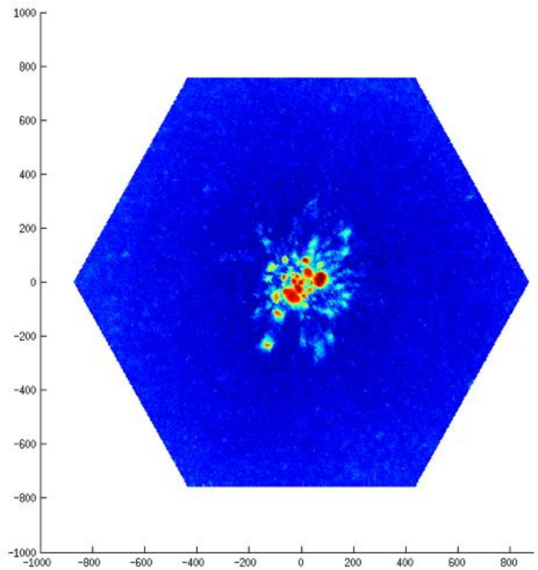
Novel application composed from existing libraries by domain expert!

R. Harder workflow: Genetic algorithm

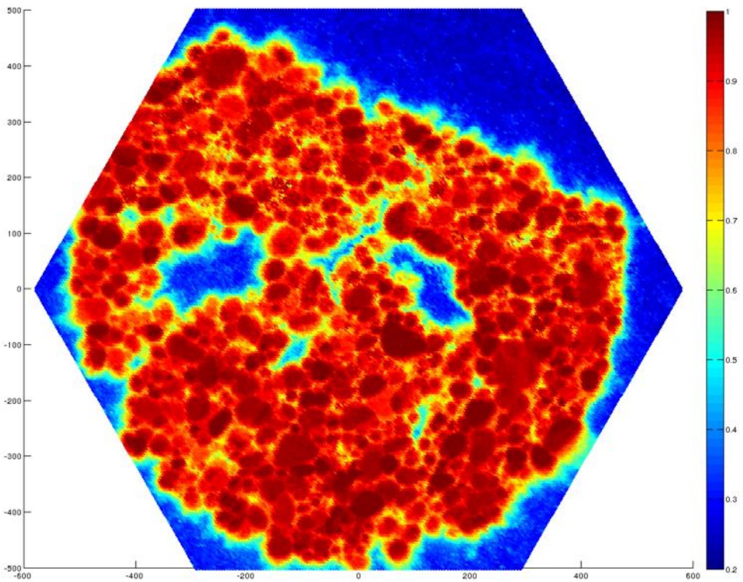
```
individuals = toint(argv("individuals"));
ngenerations = toint(argv("ngenerations"));
file winners[];
winners[0] = input("null.winner");
for (int generation = 1; generation < ngenerations;
                                        generation = generation+1) {
   file population[];
   foreach box_index in [0:individuals-1] {
     file d<sprintf("d-%i-%i.out",generation,box_index)>;
     file s<sprintf("d-%i-%i.score",generation,box_index)>;
     (d,s) = box(box_index, generation, winners[generation-1]);
     population[box_index] = d;
   }
   file winner_file<sprintf("d-%i.winner", generation)> =
                  select(generation, population);
   winners[generation] = winner_file;
 }}
```



High-Energy Diffraction Microscopy



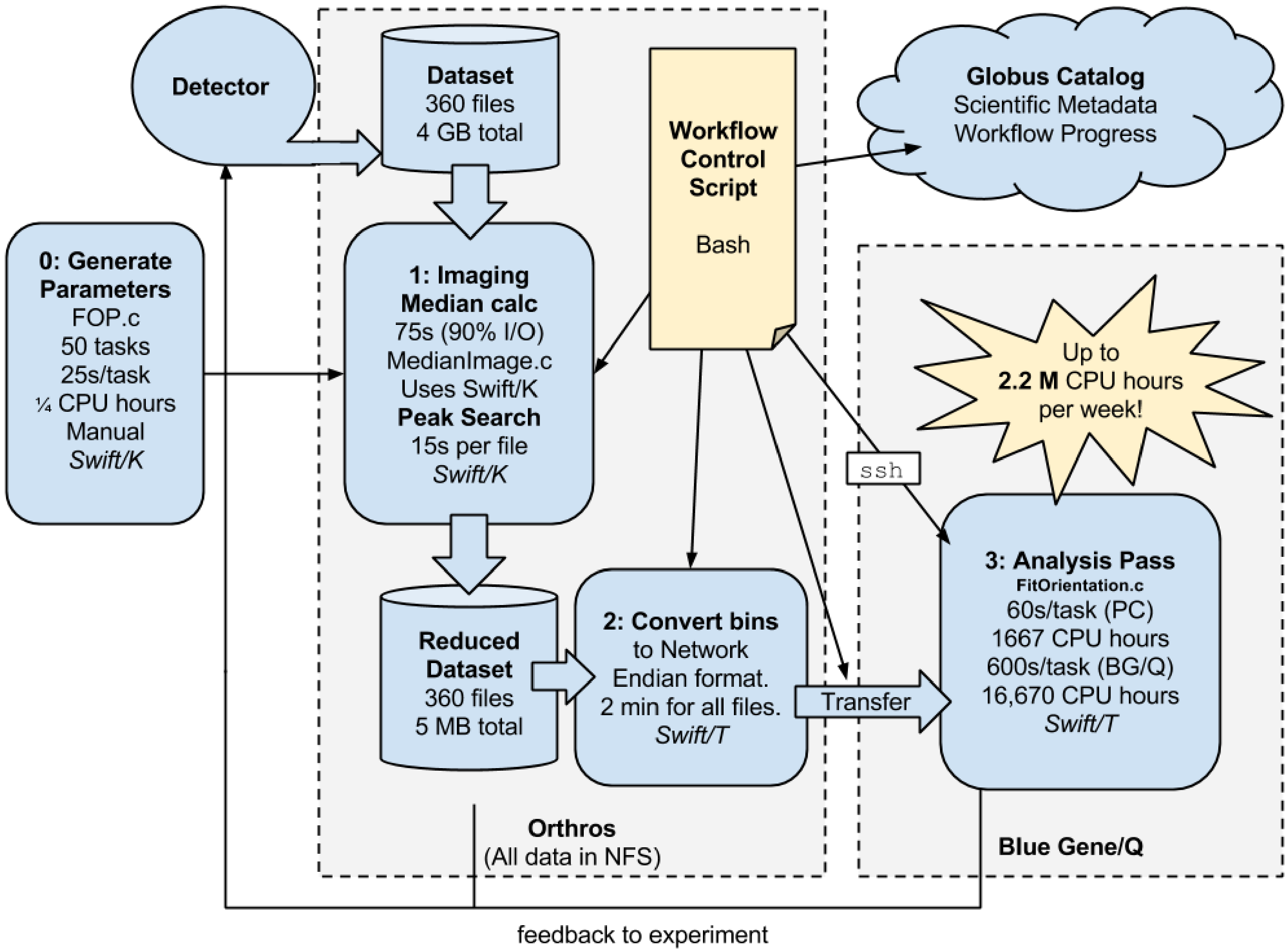
October 2013: Without Swift



April 2014: With Swift

- Near-field high-energy diffraction microscopy discovers metal grain shapes and structures
- The experimental results are greatly improved with the application of Swift-based cluster computing (**RED** indicates higher confidence in results)

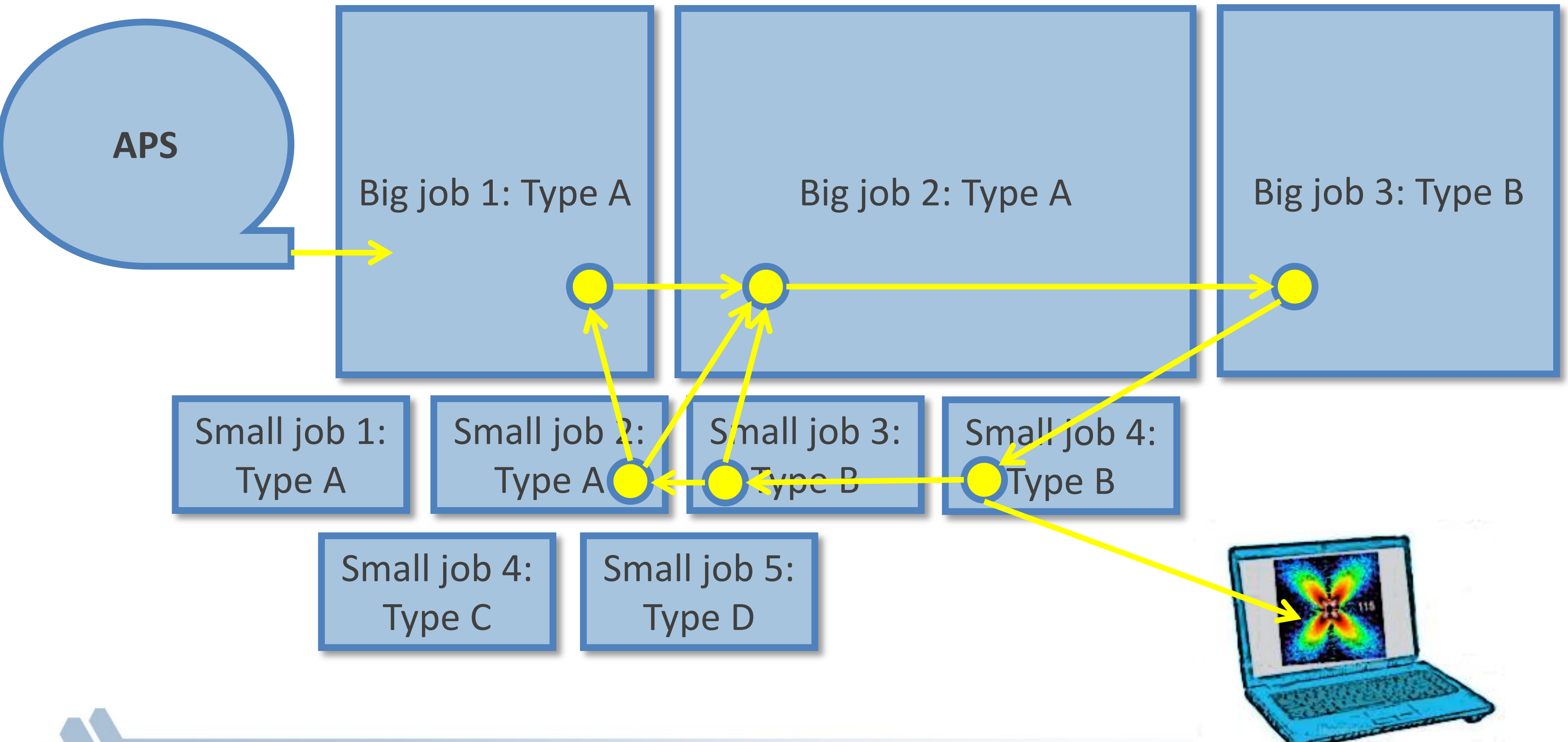
NF-HEDM: Cross-lab workflow



FUTURE WORK

Extreme scale application ensembles

- Develop Swift for exascale experiment ensembles
  - Deploy stateful, varying sized jobs
  - Outermost, experiment-level coordination via dataflow
  - Plug in experiments and human-in-the-loop models (dataflow filters)



Future Work

- Develop Swift for exascale
  - Continue scaling work: Study distributed dataflow for realistic patterns
  - Ease integration with native code
- Application collaborations
  - Materials science: APS (Osborn, Sharma)
  - Molecular dynamics: NAMD (Phillips), LAMMPS (Whitmer)
- Connect with novel systems elsewhere in MCS, ALCF:
  - Memcached (Isaila et al.)
  - Tess (Peterka et al.)
  - Filesystems (Ross et al.)
- Connect with new applications at the CI and elsewhere!

Summary

- Swift: High-level scripting for outermost programming constructs
  - Handles many aspects of the scientific computing experience
  - Described how dataflow logic is distributed
  - New features for parallel tasks
- 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 project collaborators: Tom Peterka, Jim Dinan, Ray Osborn, Reinhard Neder, Guy Jennings, Hemant Sharma, Rachana Ananthakrishnan, Ben Blaiszik, Kyle Chard, Tim Germann, and others
- **Questions?**