Case Studies in Dataflow Composition of Scalable High Performance Applications

Justin M Wozniak, Timothy Armstrong, Daniel Katz, Michael Wilde, Ian Foster

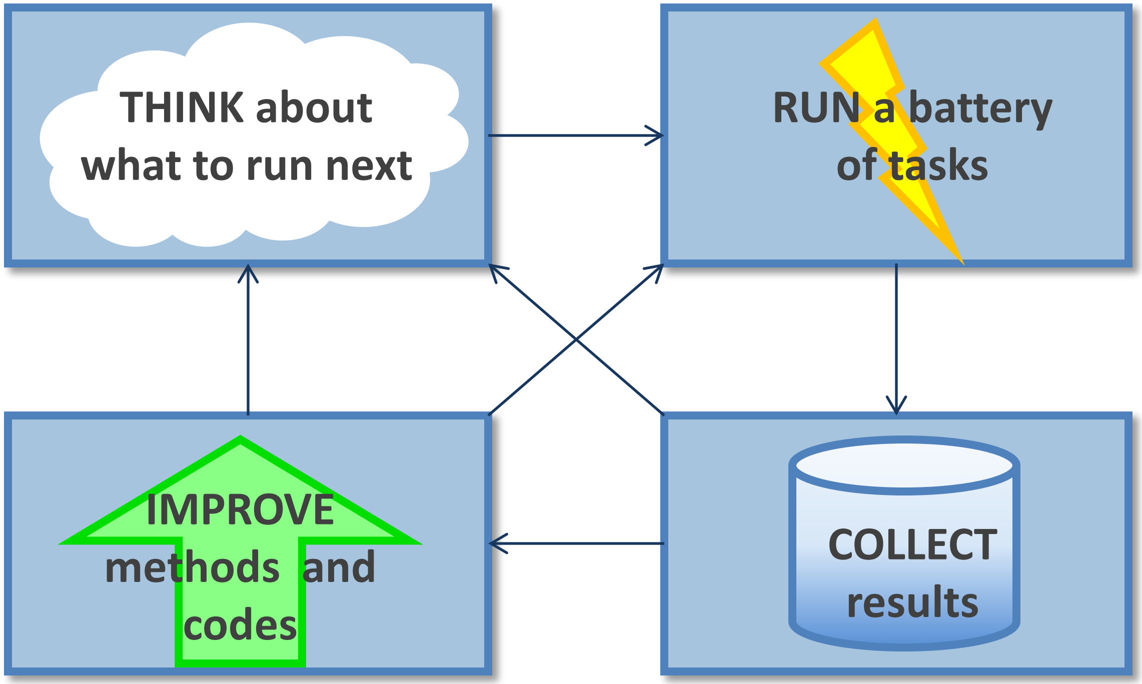
wozniak@mcs.anl.gov

http://swift-lang.org

Workshop on Extreme Scale Programming Tools at SC

November 17, 2014 – New Orleans

The Scientific Computing Campaign



- Swift addresses most of these components

Software for the Computing Campaign

- Swift: Composing the computational experiment
  - Code coupling
  - Task communication
  - Expressing complex workflows
  - Deploying large workloads
- Performance visualization
  - Debugging and performance analysis for workflows
  - Plotting and visualization
- Case studies
  - Streamline visualization – parallel tasks
  - X-ray science – remote I/O and analysis

Case Studies in Dataflow Composition of Scalable High Performance Applications

SWIFT OVERVIEW

Goal: Programmability for large scale analysis

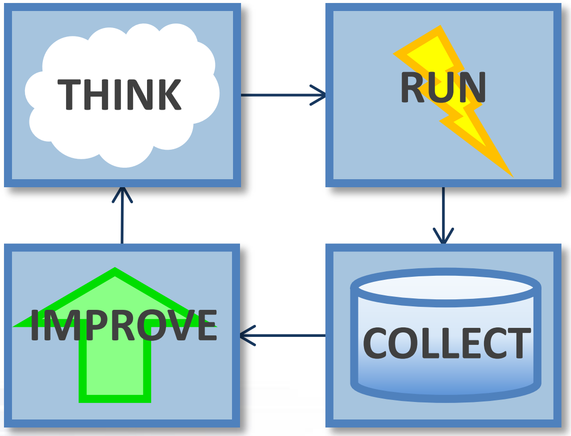
- Our solution is "many-task" computing: higher-level applications composed of many run-to-completion tasks: **input→compute→output** Message passing is handled by our implementation details

- 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 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;
}
```

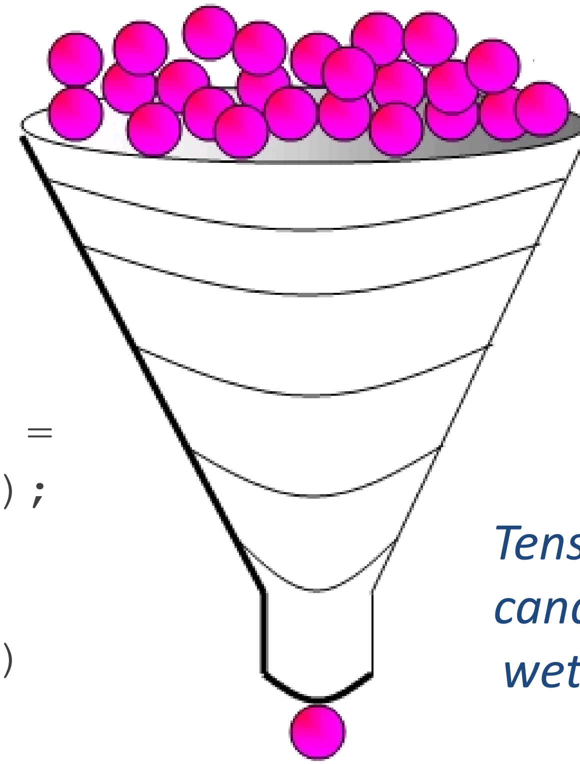
- `F()` and `G()` implemented in native code or external programs
- `F()` and `G()` run in concurrently in different processes
- `r` is computed when they are both done

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

More concurrency: Loops and arrays

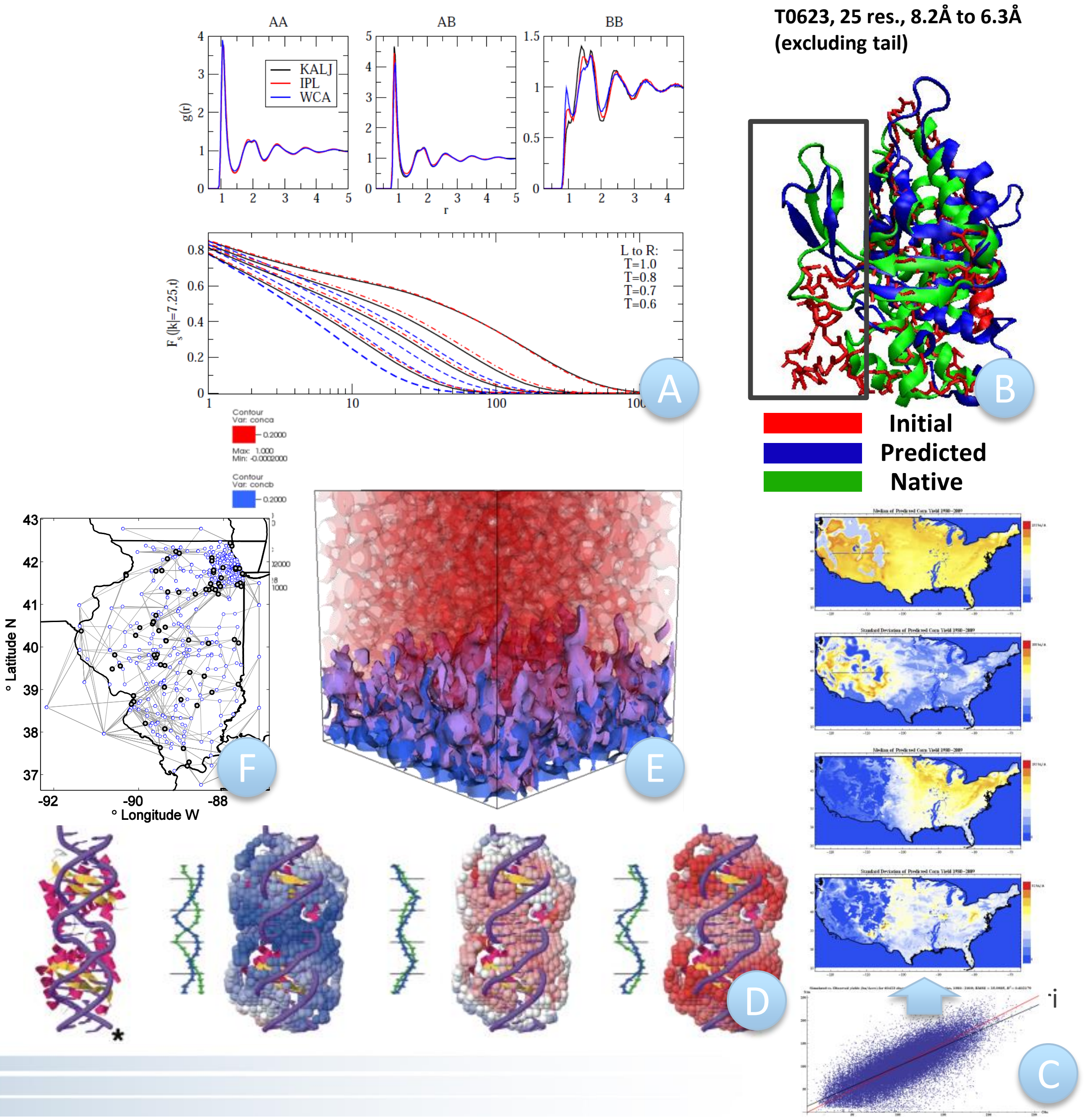
- Protein docking: attempt to dock various drugs against to a handful of protein targets
- Each task is a simulator invocation
- Generates millions of tasks

```
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)
```



Large-scale many-task applications using Swift

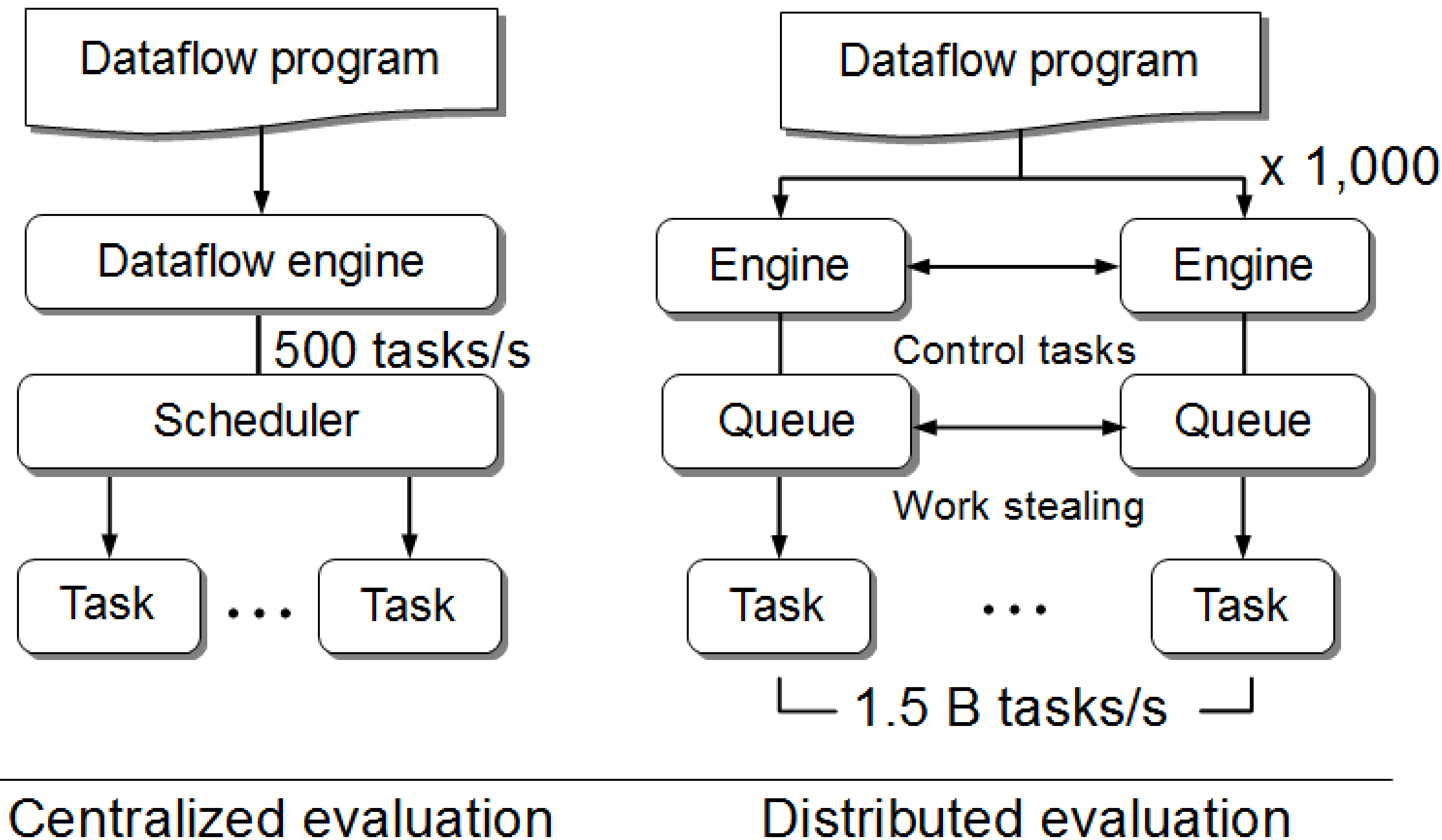
- Simulation of metals under stress
- Molecular dynamics: NAMD
- Molecular dynamics: LAMMPS
- X-ray scattering data aggregation
- X-ray imaging analysis
- Multiscale subsurface flow modeling
- Modeling of the power grid
- Climate data extraction
- … and many more



Swift/T: Swift for high-performance computing

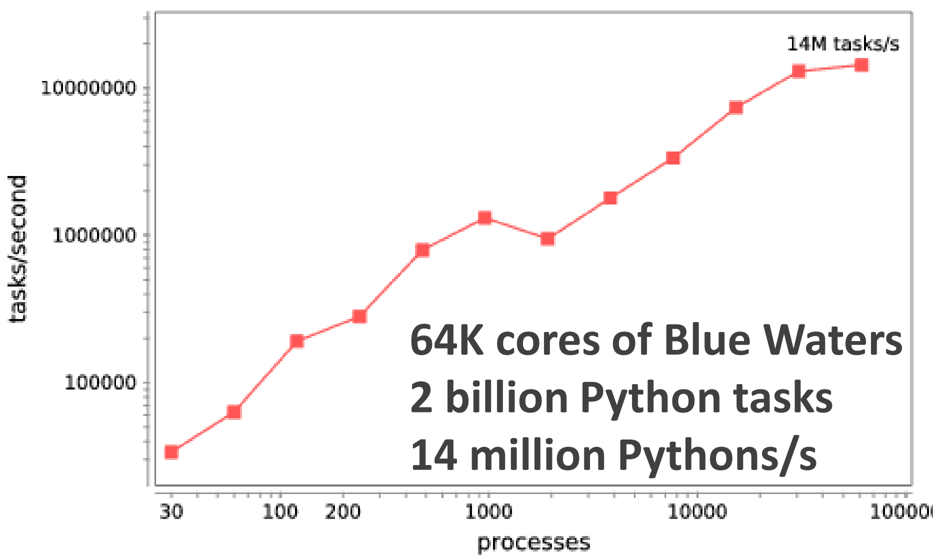
Had this:
(Swift/K)

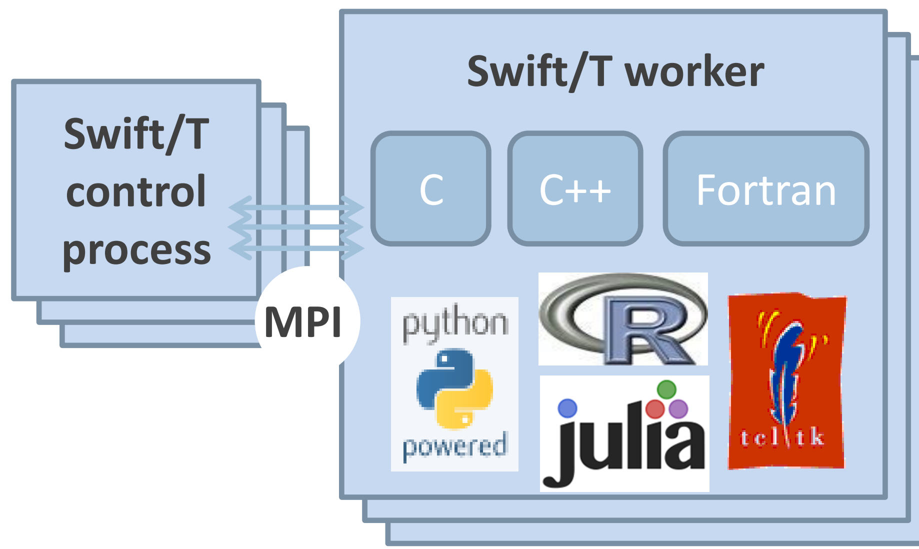
For extreme scale, we need this:
(Swift/T)



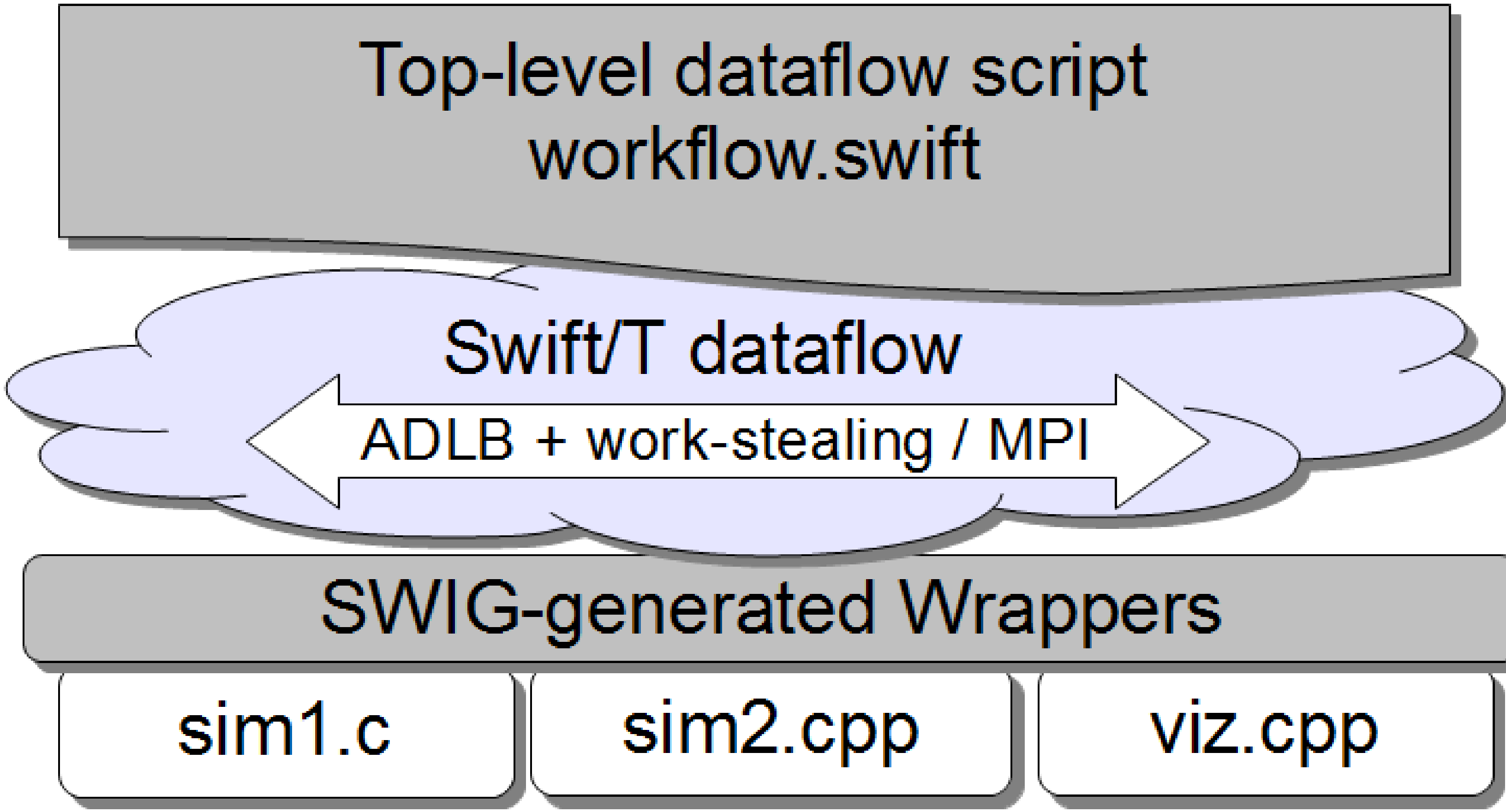
Swift/T: Enabling high-performance workflows

- 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





Dataflow script produces work for work queue



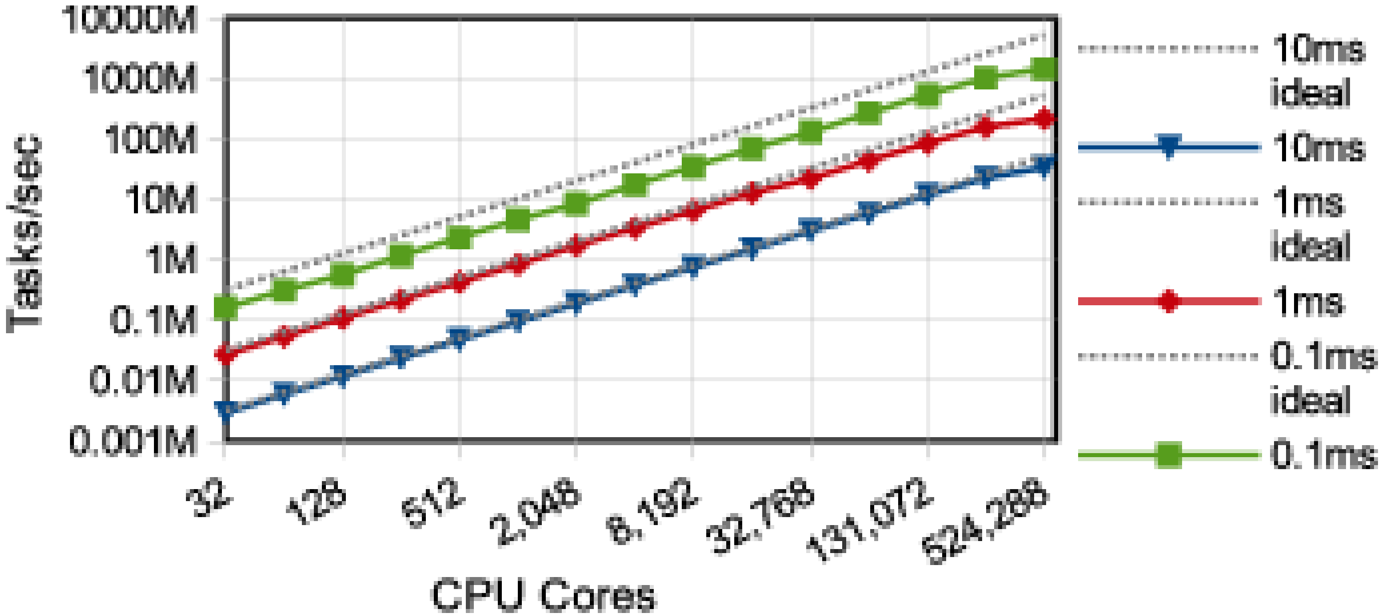
- Including MPI libraries
- We use a Scioto-like algortithm for hierarchical work-stealing among ADLB servers (ADLB/X, our fork of ADLB)

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

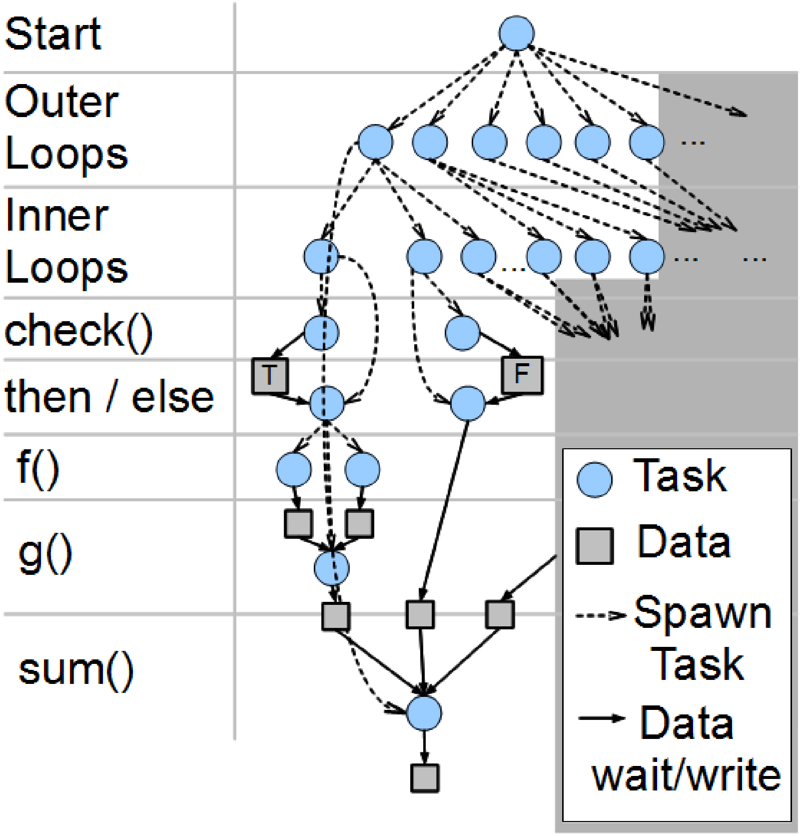
Basic scalability



- 1.5 billion tasks/s on 512K cores of Blue Waters, so far
- See our SC 2014 paper for comprehensive performance analysis

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]);
}
```



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]
```

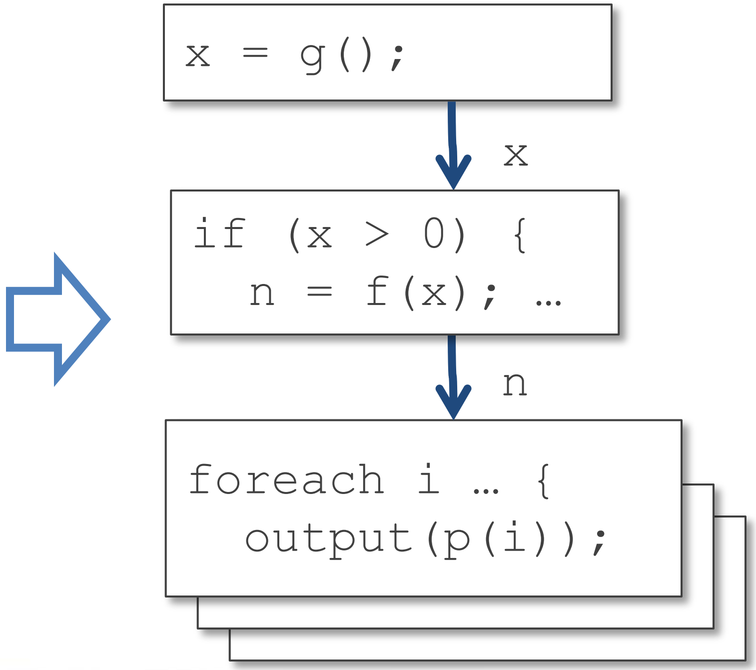
```
• Process g
• Store A[3]
```

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

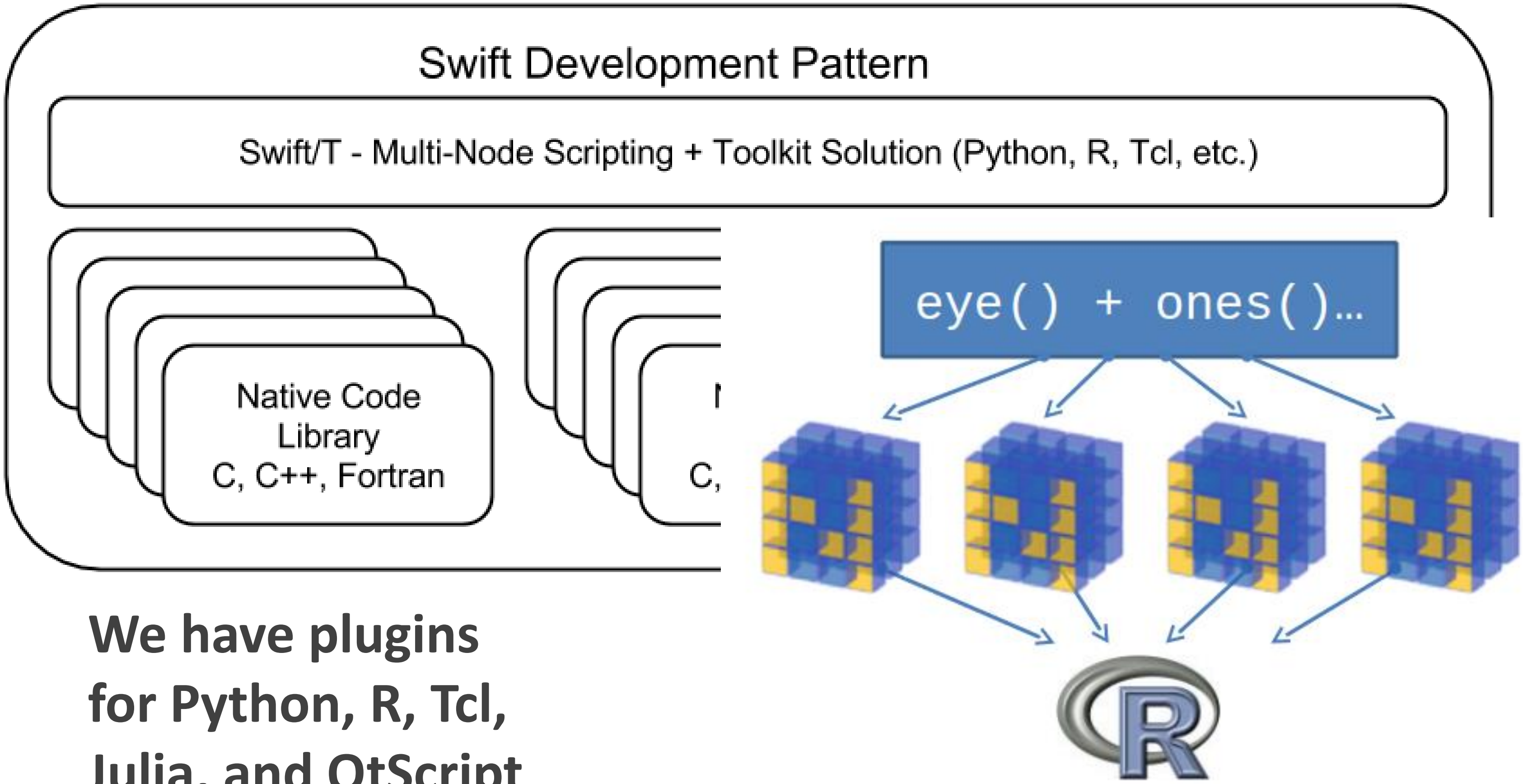
Swift code in dataflow

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

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



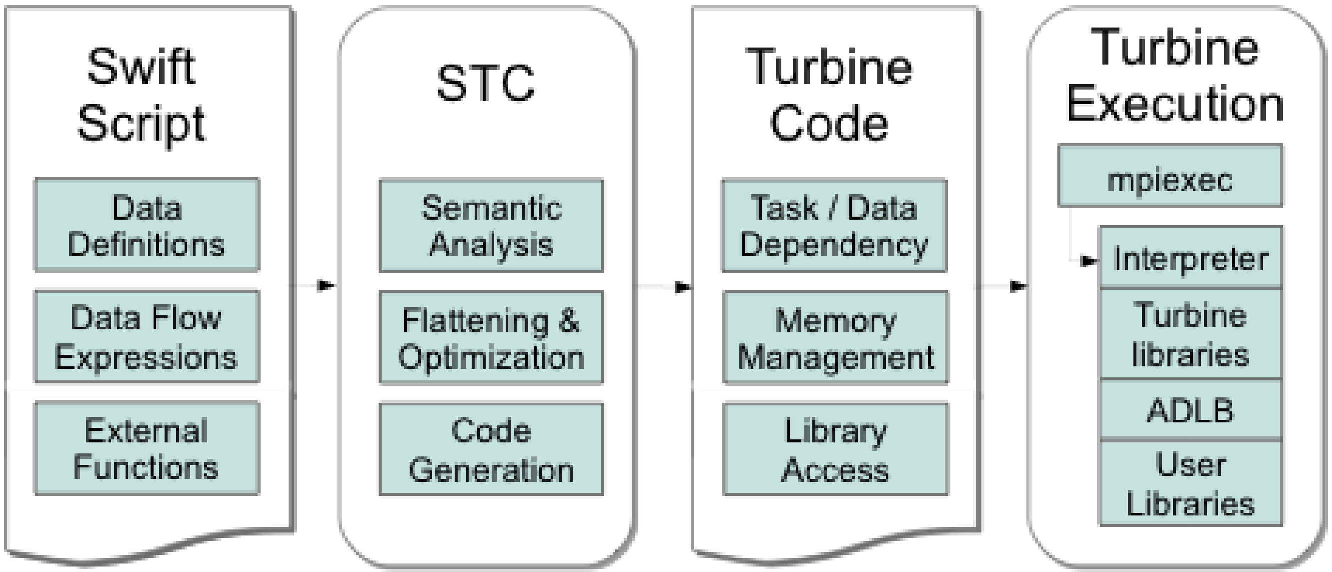
Support calls to embedded interpreters



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 (based on ANTLR) translates 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.

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);
```

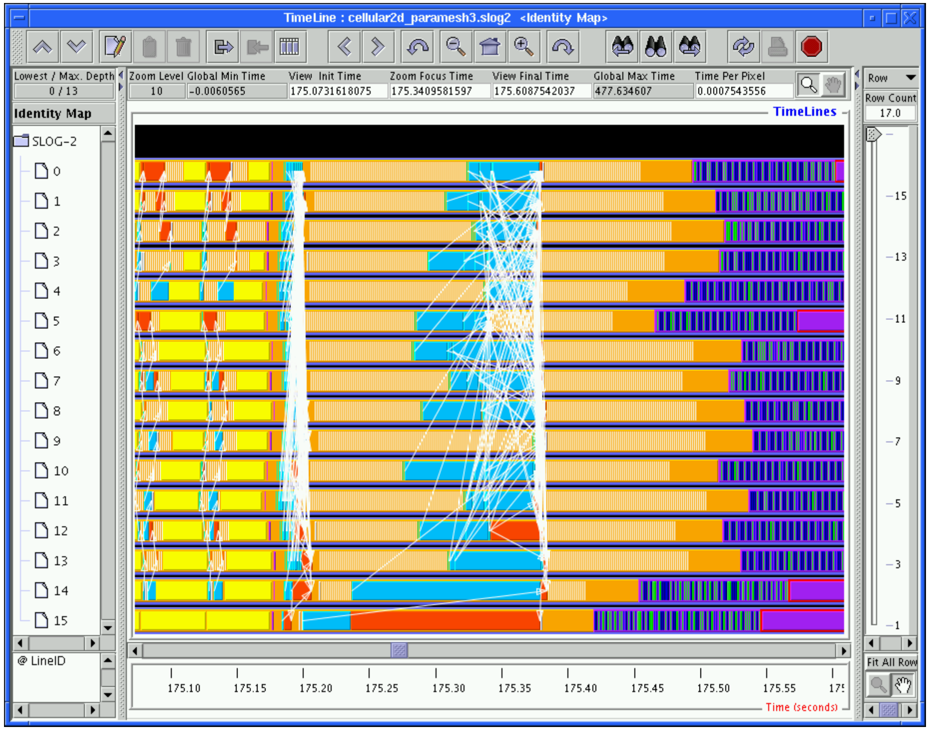
PERFORMANCE TOOLS

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

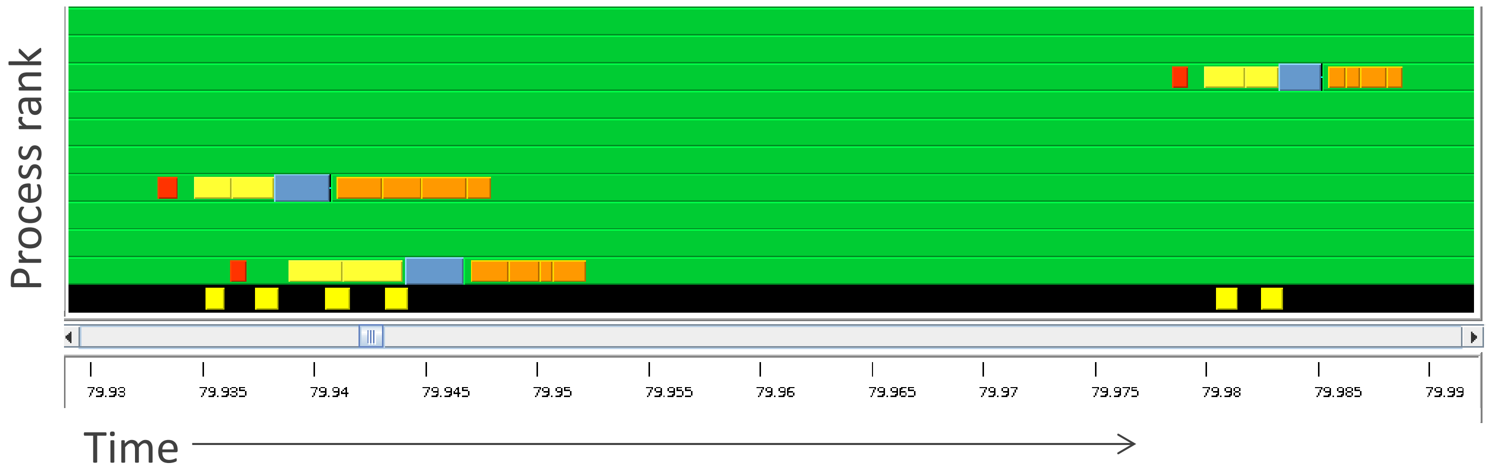
- Use MPE to log data, task operations, calls to native code
- Use MPE metadata to annotate events for later queries

- 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

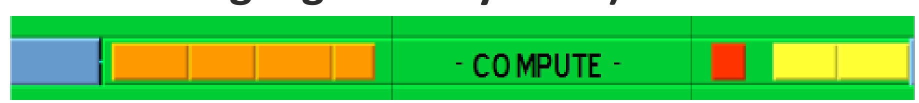
Visualization of Swift/T execution

- 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

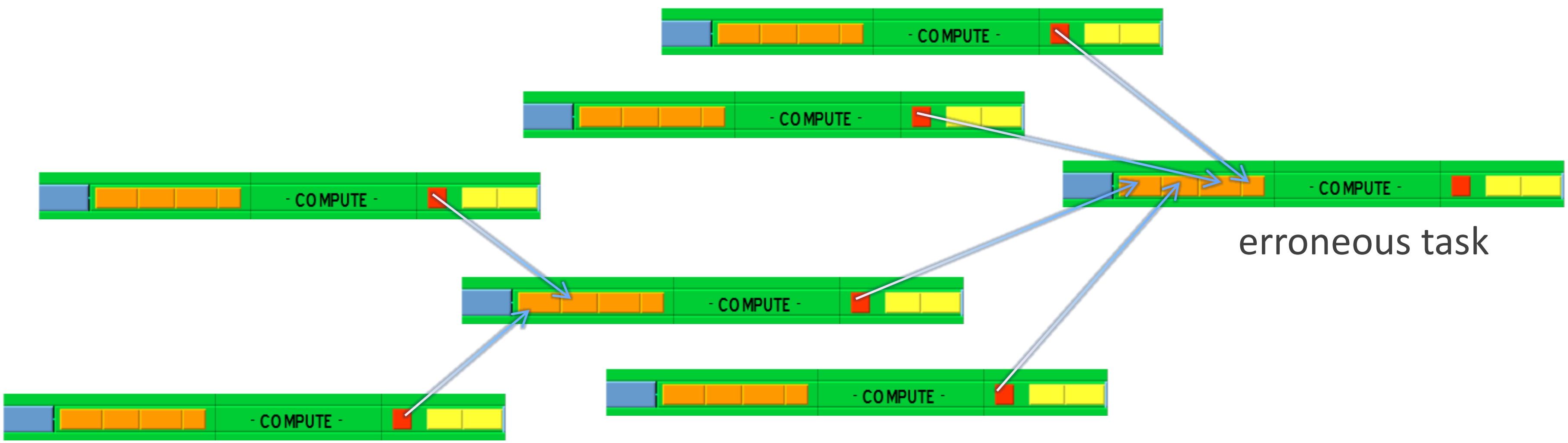
  – **PIPS task computation** **Store variable** **Notification (via control task)**
    Blue: Get next task **Retrieve variable**
    Server process (handling of control task is highlighted in yellow)

- 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.

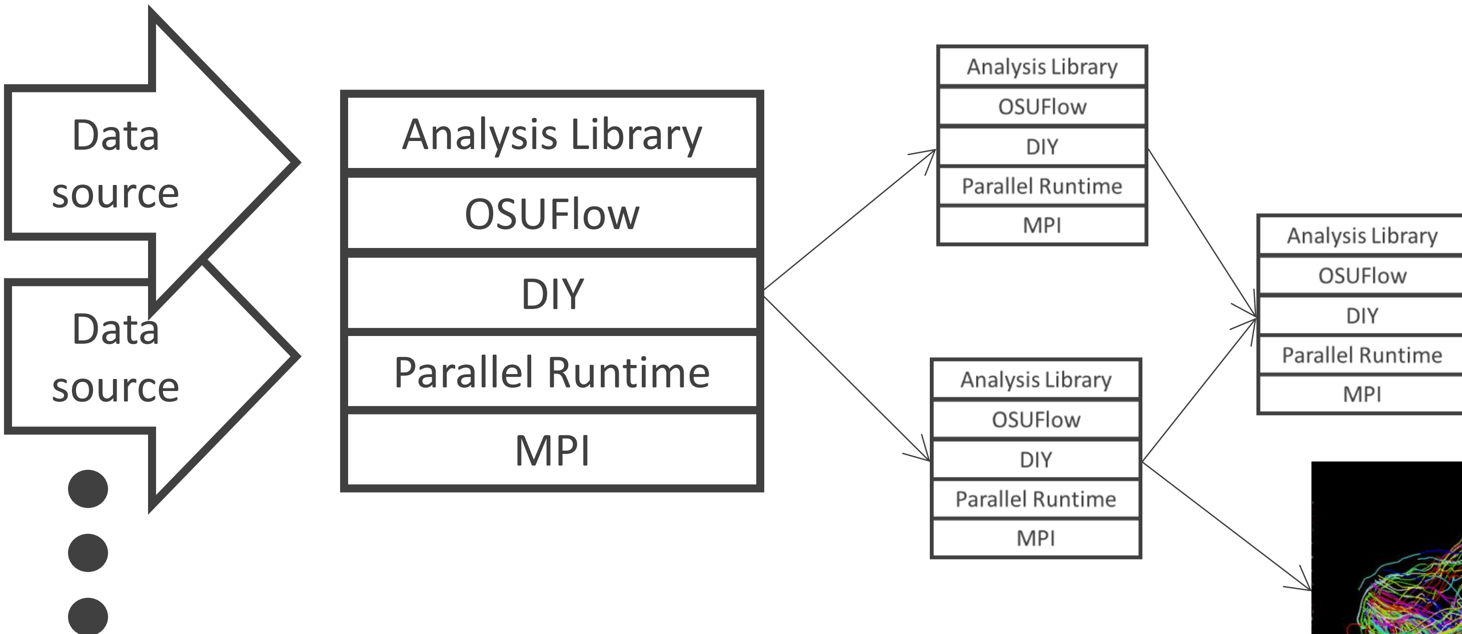
← ← ← (searching backwards)

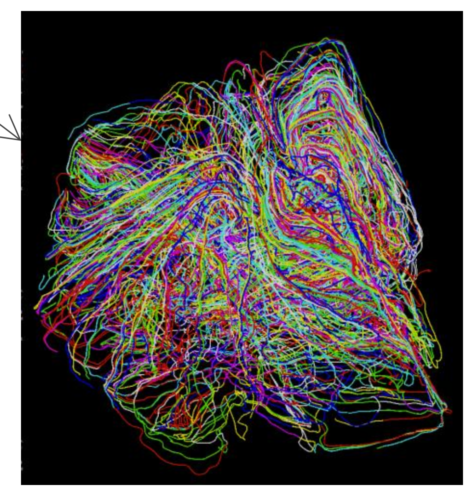
- Wozniak et al. A model for tracing and debugging large-scale task-parallel programs with MPE. Proc. LASH-C at PPoPP, 2013.

Case Studies in Dataflow Composition of Scalable High Performance Applications

CASE STUDIES

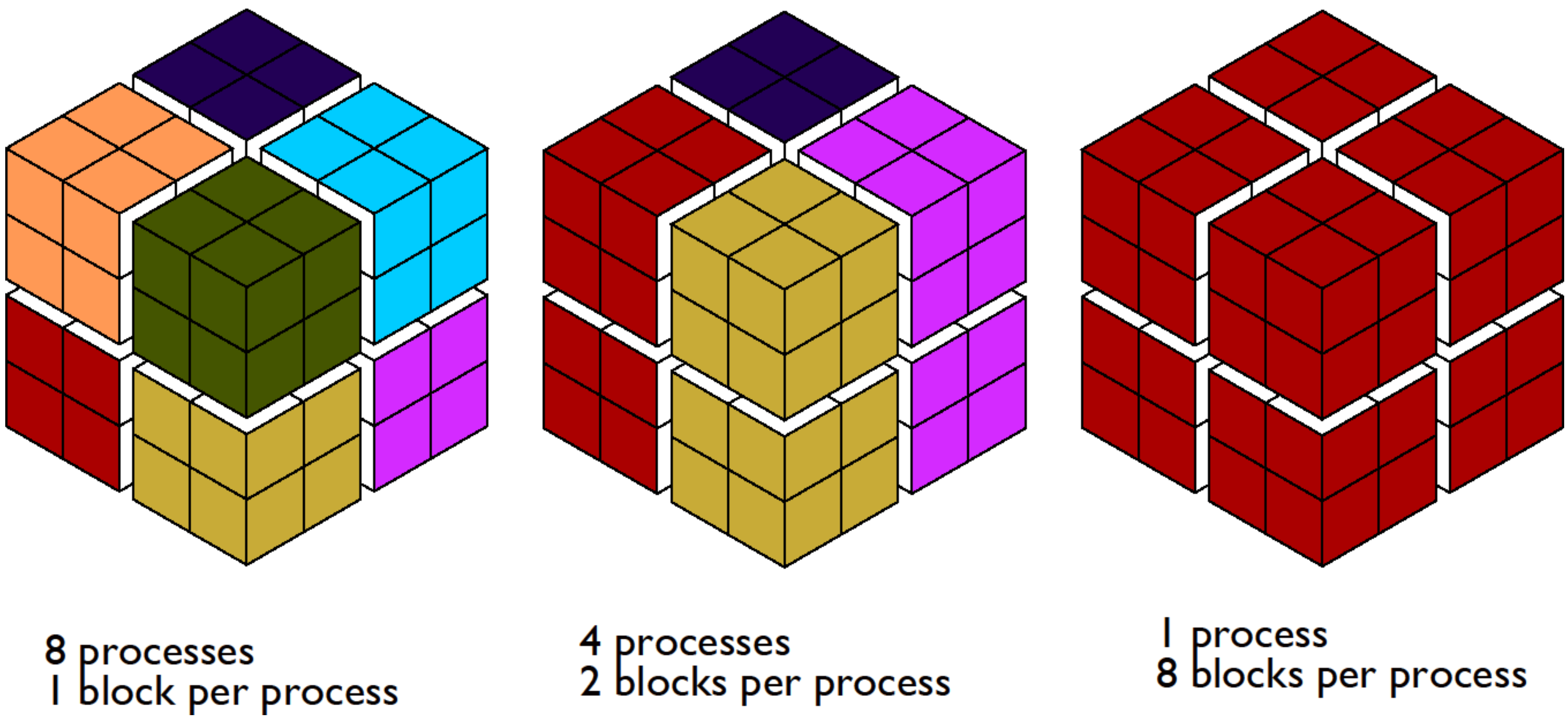
Dataflow+data-parallel analysis/visualization





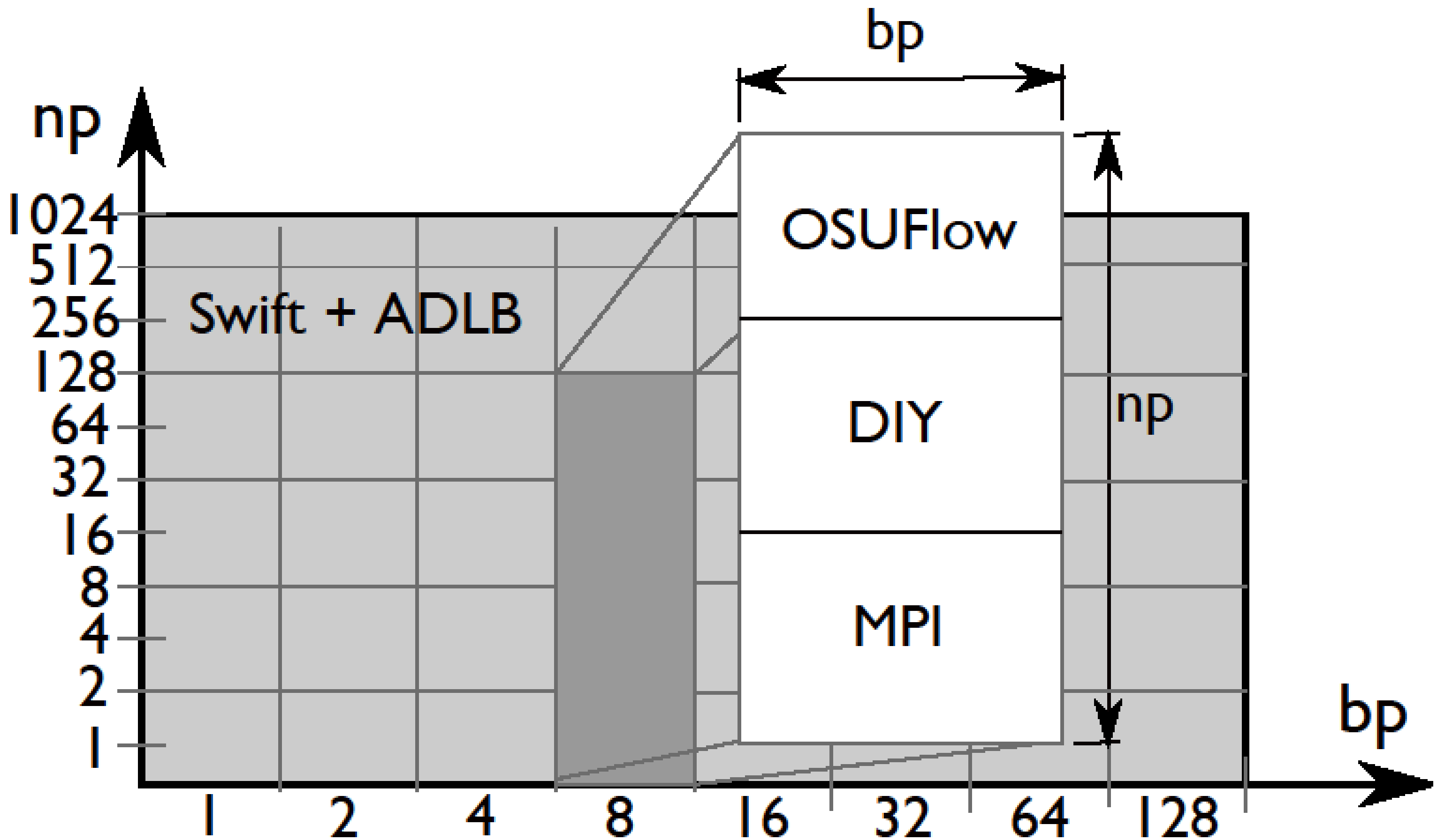
Dataflow-structured analysis framework based on OSUFlow/DIY

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



Can map blocks to processes in varying ways

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



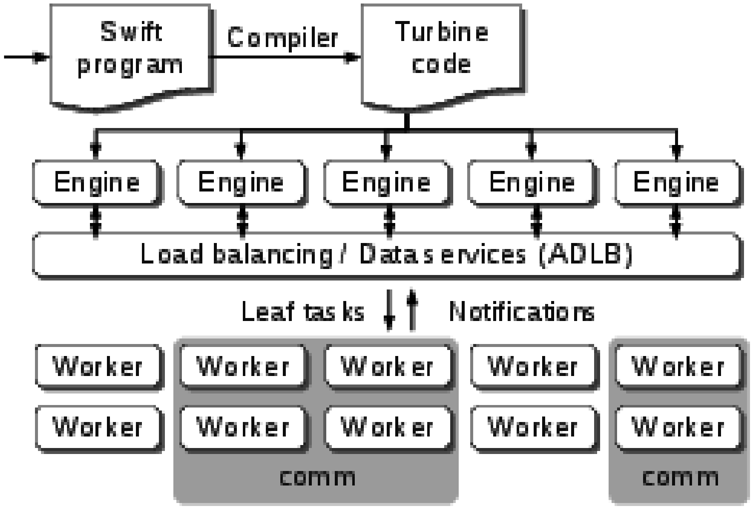
- Try all configurations to find best performance
- Goal: Rapidly develop and execute sweep of MPI executions

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

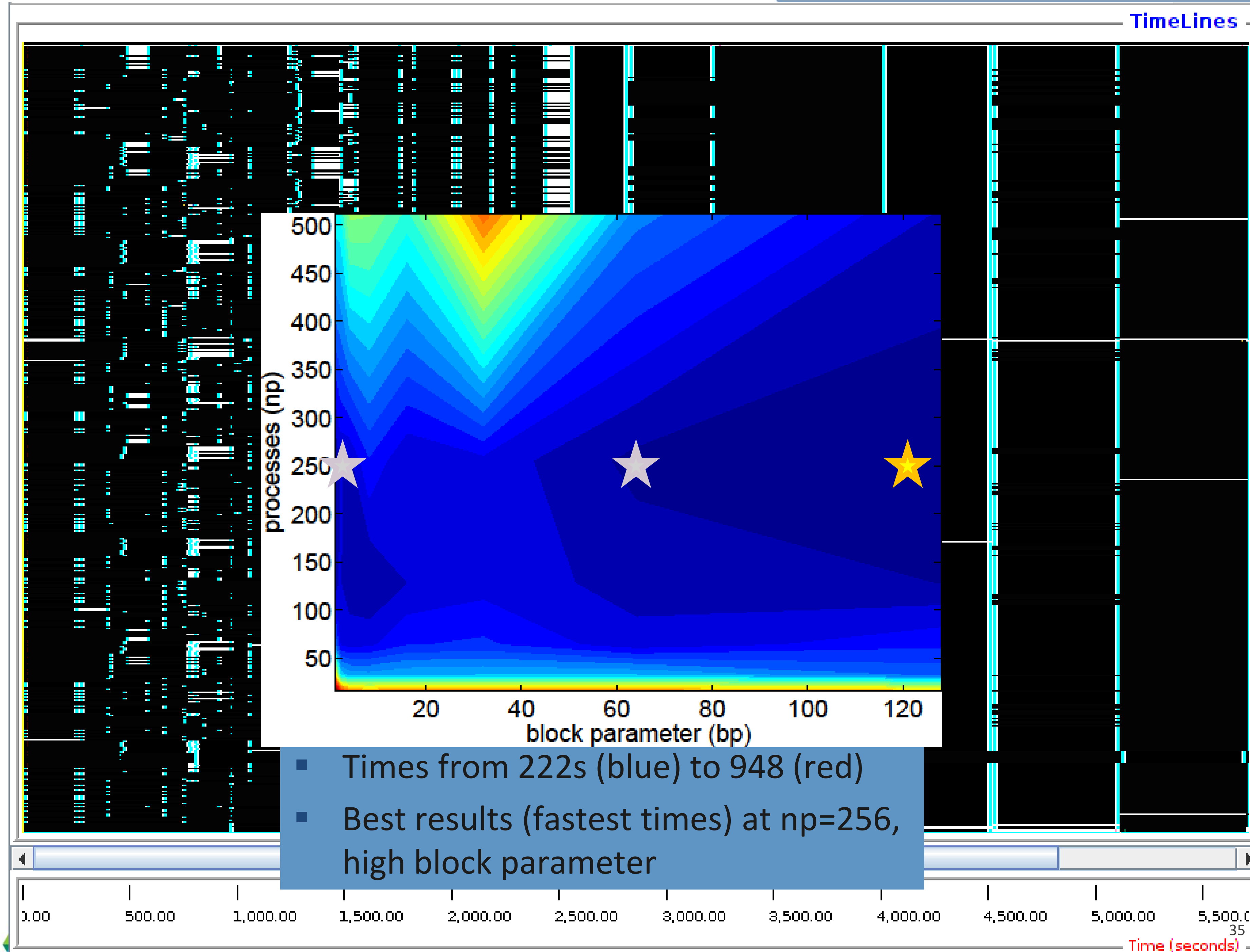


- Wozniak et al. Dataflow coordination of data-parallel tasks via MPI 3.0. Proc EuroMPI, 2013.

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);
    }}}
```

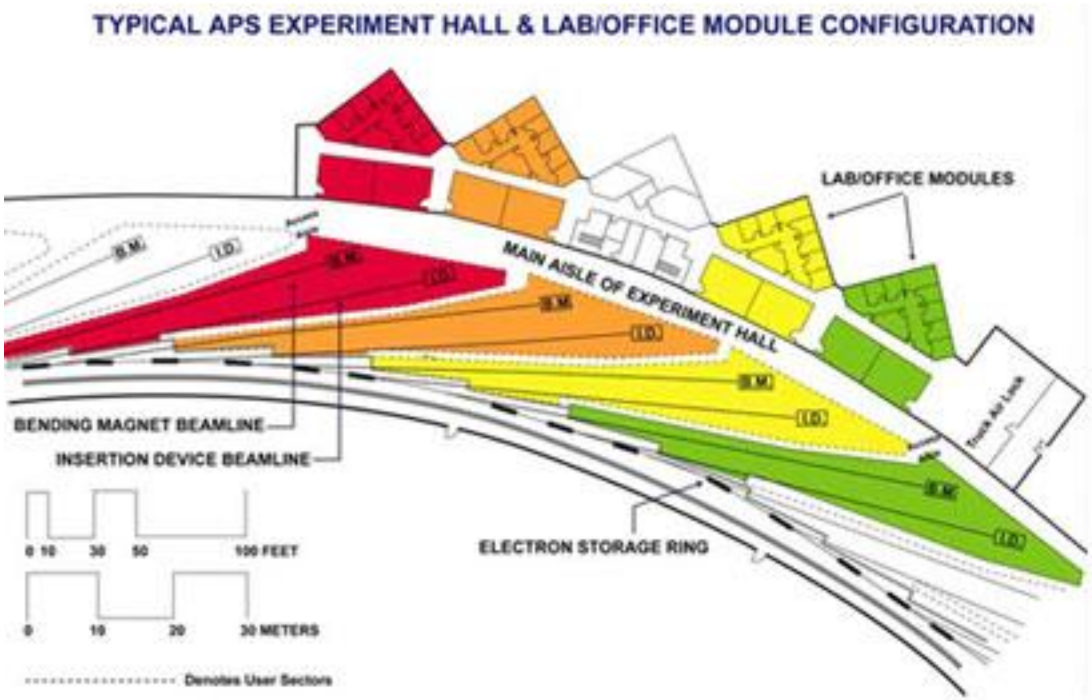


- Times from 222s (blue) to 948 (red)
- Best results (fastest times) at np=256, high block parameter


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



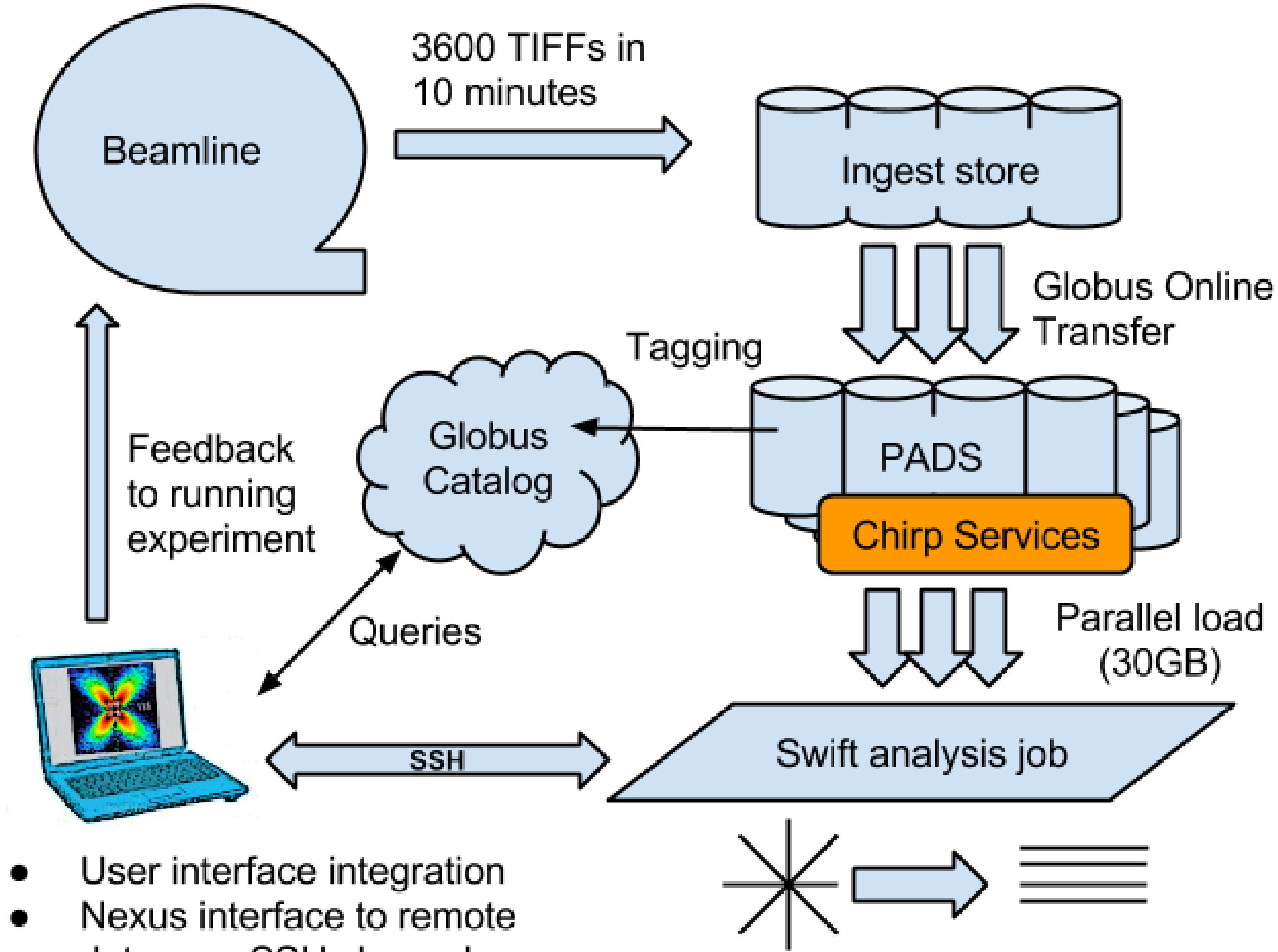
Data management for the energy sciences

- "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)*

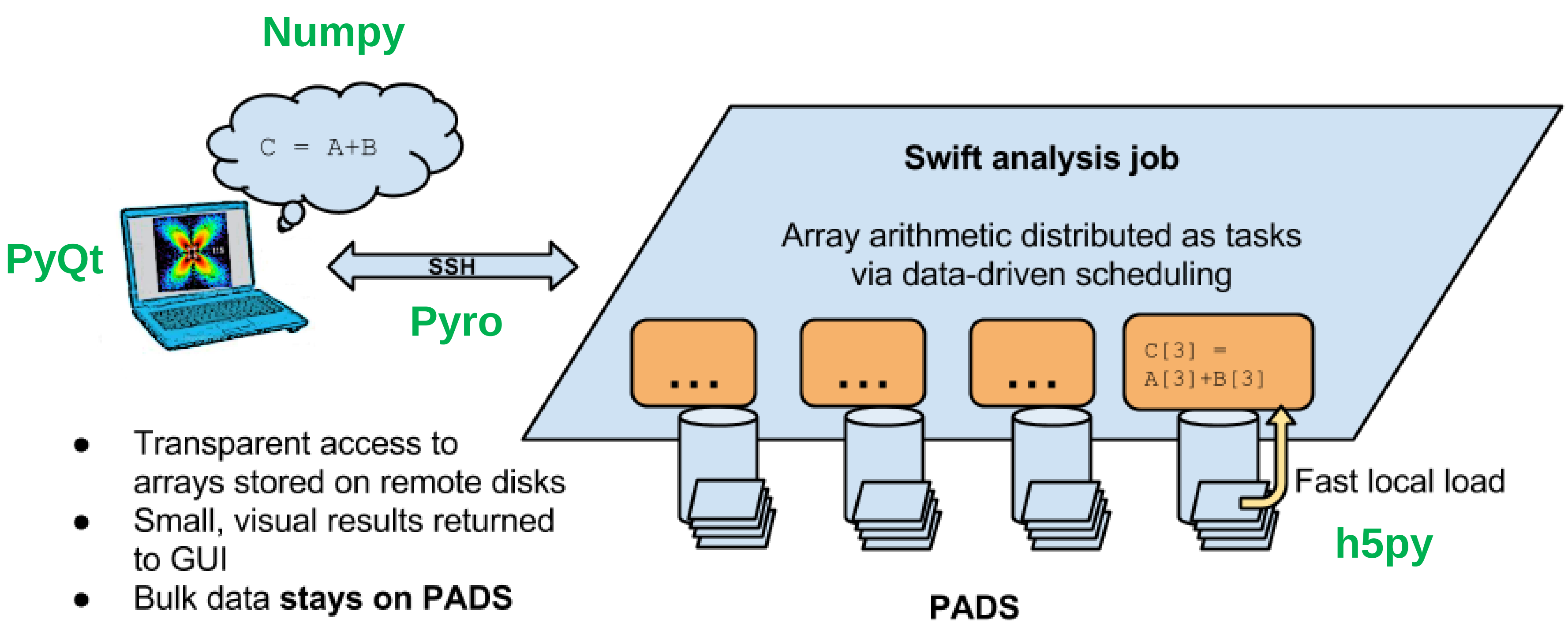
- Our goals:
  - Modify the operating systems of APS stations to allow real-time streaming to a novel data storage/analysis platform.
  - Converting data from the standard detector formats (usually TIFF) to HDF5 and adding metadata and provenance, based on the NeXus data format.
  - Rewrite analysis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

Data ingest/analysis/archive



- User interface integration
- Nexus interface to remote data over SSH channel

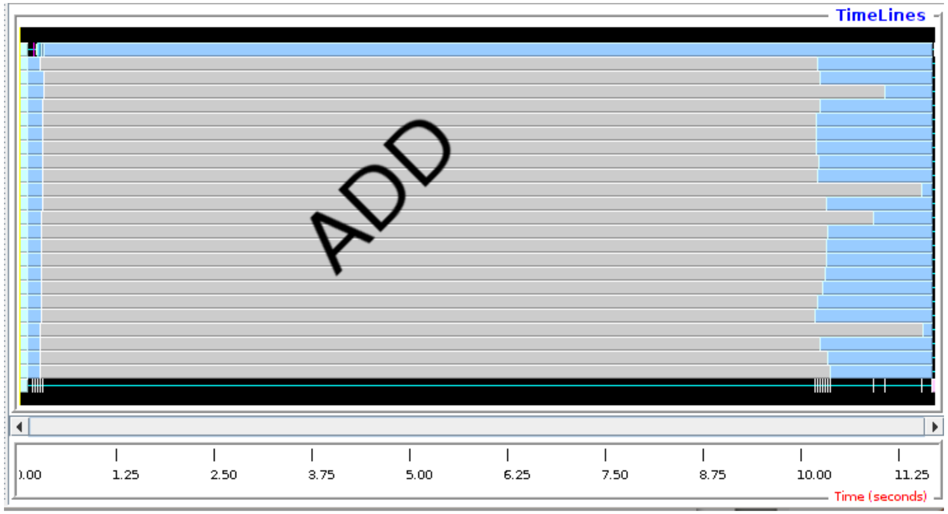
Interactive analysis powered by scalable storage

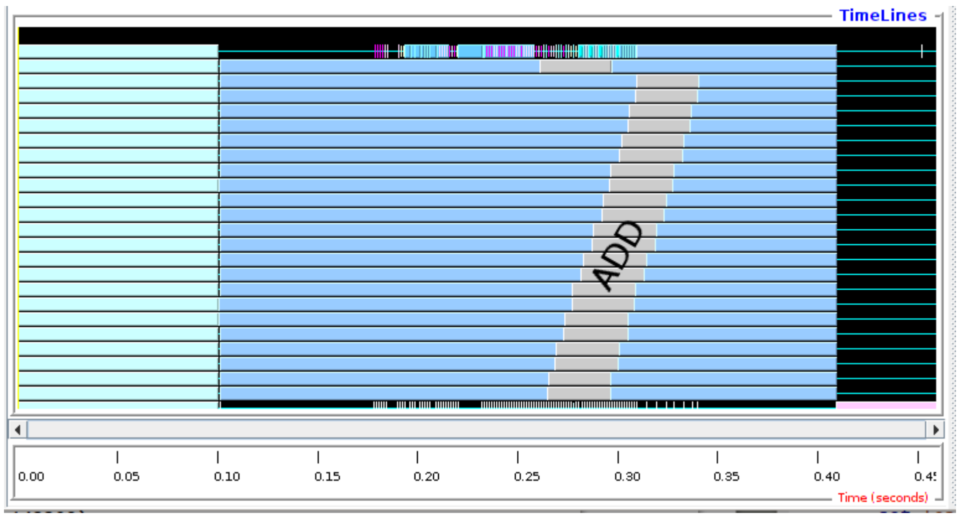


- Replace GUI analysis internals with operations on remote data

Remote matrix arithmetic: Initial results

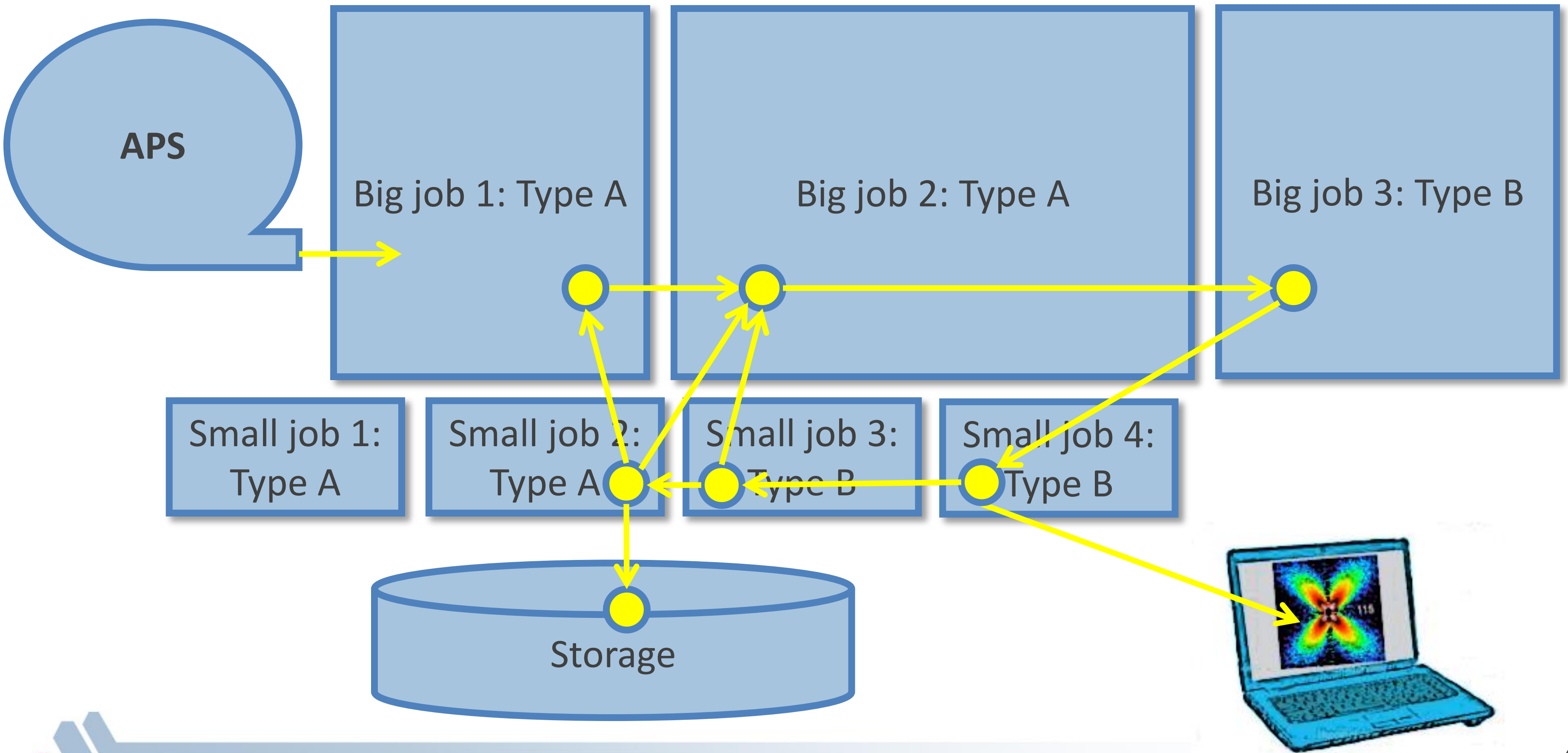
- Initial run shows performance issue: addition took too long
- Swift profiling isolated issue: convert addition routine from script to C function: obtained 10,000 X speedup
- Swift/T integrates with MPE/Jumpshot and other MPI-based performance analysis techniques





Future work: Extreme scale 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)



Summary

- Swift: High-level scripting for outermost programming constructs
  - Handles many aspects of the scientific computing experience
  - Described how logs enable performance visualization
  - Showed use cases in streamline visualization and X-ray science
- 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, and others
- Thanks to the organizers!
- **Questions?**