Swift – A parallel scripting language for petascale many-task applications

Presented by: Duncan Roweth, Cray

Swift is developed and supported by:

Computation Institute, University of Chicago
and Argonne National Laboratory

Contact: Michael Wilde, wilde@mcs.anl.gov

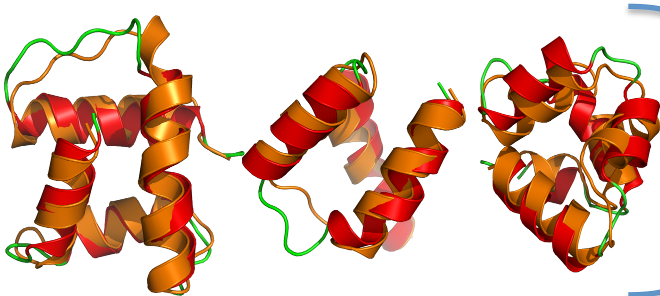
Revised 2012.0502

www.ci.uchicago.edu/swift

swift

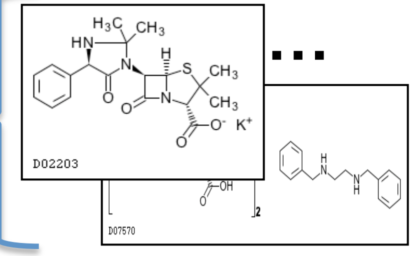
- **Swift is a parallel scripting language** for multicores, clusters, grids, clouds, and **supercomputers**
  - *for loosely-coupled "many-task" applications – programs and tools linked by exchanging files*
  - *debug on a laptop, then run on a Cray system*
- **Swift is easy to write**
  - *a simple high-level functional language with C-like syntax*
  - *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 – Swift acts as a self-contained grid or cloud client*
  - ***Swift automatically runs scripts in parallel*** *– typically without user declarations*
- **Swift is fast**: based on a powerful, efficient, scalable and flexible Java execution engine
  - *scales readily to millions of tasks*
- Swift **usage** is growing:
  - *applications in neuroscience, proteomics, molecular dynamics, biochemistry, economics, statistics, earth systems science, and beyond.*

When do you need Swift?

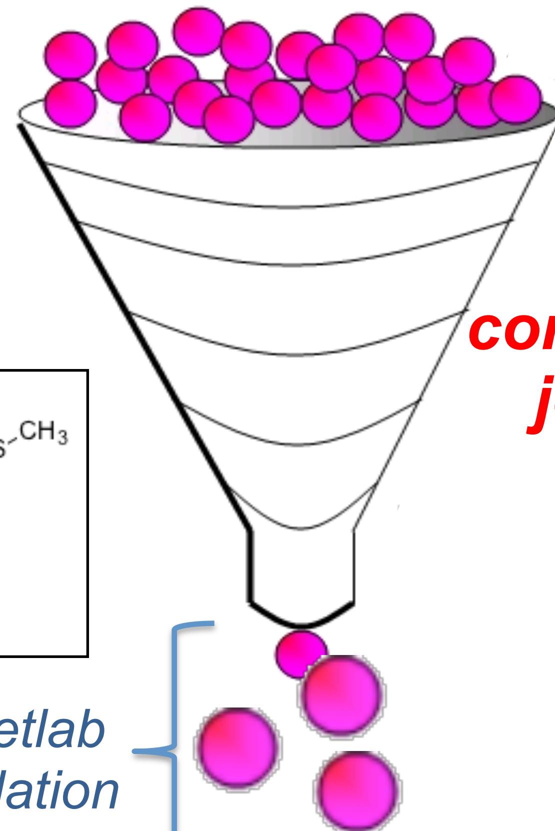


T1af7 T1b72 T1r69

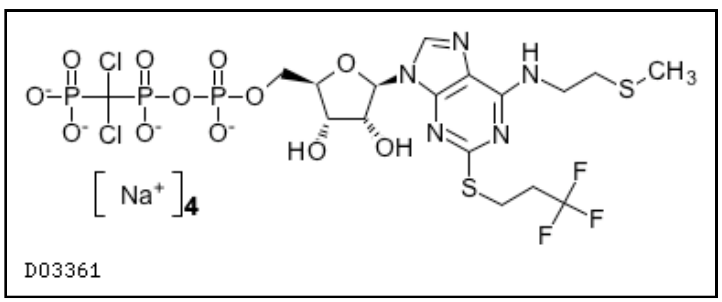
O(10) proteins implicated in a disease **x** *O(100K) Drug candidates (ligands)*



Typical application: protein-ligand docking for drug screening is a **many-task** process



1M compute jobs



Tens of fruitful candidates for wetlab and X-Ray crystallography validation

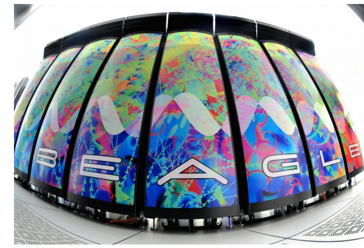
Computation Institute

www.ci.uchicago.edu/swift

Argonne NATIONAL LABORATORY

Challenge: Complexity of parallel computing

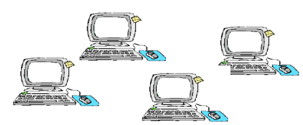
- Many problems call for a "many task" approach
- …but parallel programming is an obstacle for scientists
- And from now on, all systems will be parallel!
- Swift harnesses diverse parallel systems with simple scripts that run ordinary applications



Open Science Grid

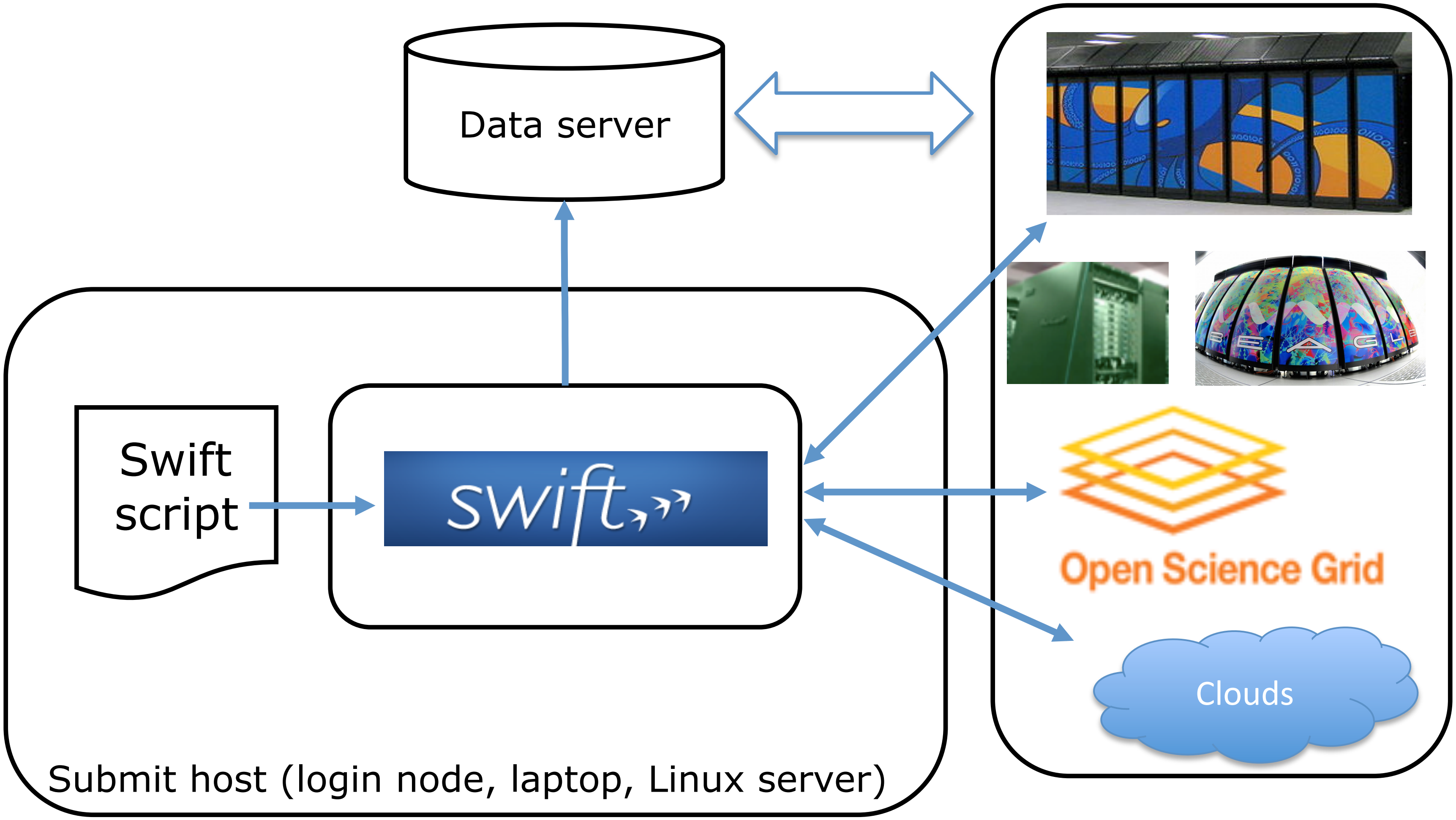
XSEDE
Extreme Science and Engineering Discovery Environment

Clouds:
Amazon EC2,
NSF FutureGrid,
BioNimbus



Multicore workstations and ad-hoc clusters

Solution: parallel scripting for high level parallelism



Swift runs parallel scripts on clusters, grids, clouds, and supercomputers.

All Swift execution is parallel, driven by data flow

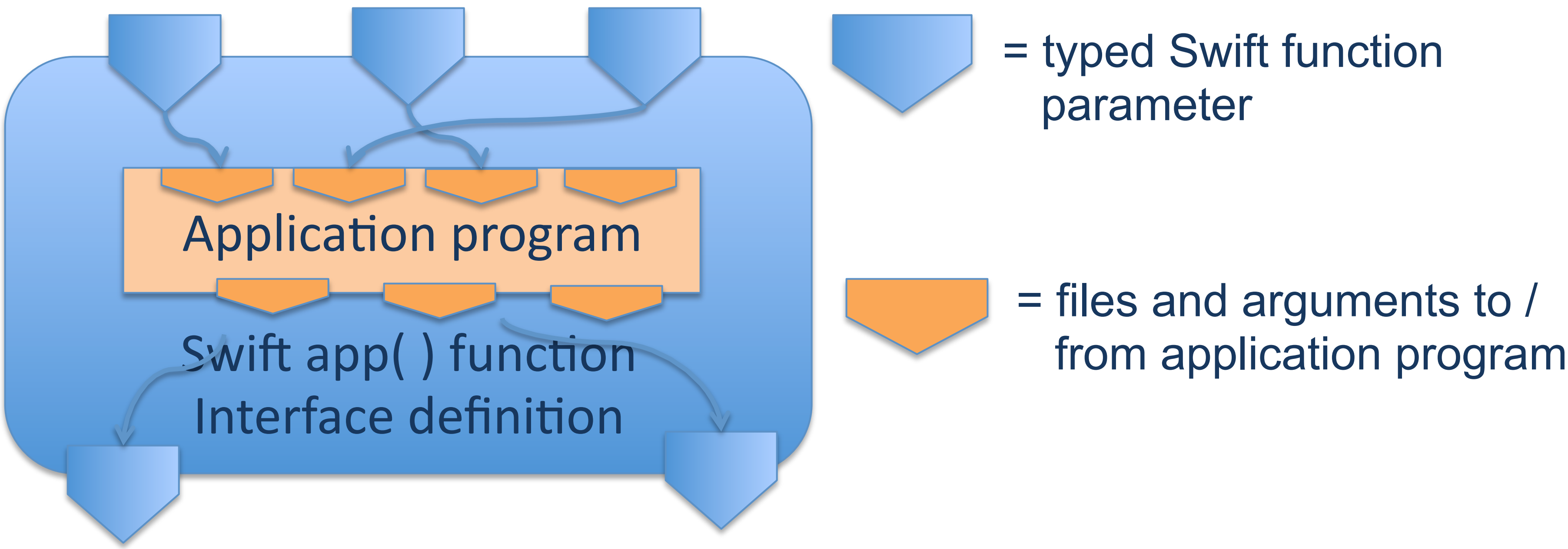
```
1  j = f(i);    // f() and g() are
2  k = g(i);    // computed in parallel
3  r = j + k;   // r is set when they are done
```

4 This parallelism is automatic and pervasive in Swift.

```
5  foreach obs,i in observations {
     inv[i] = invert(obs);
   }
```

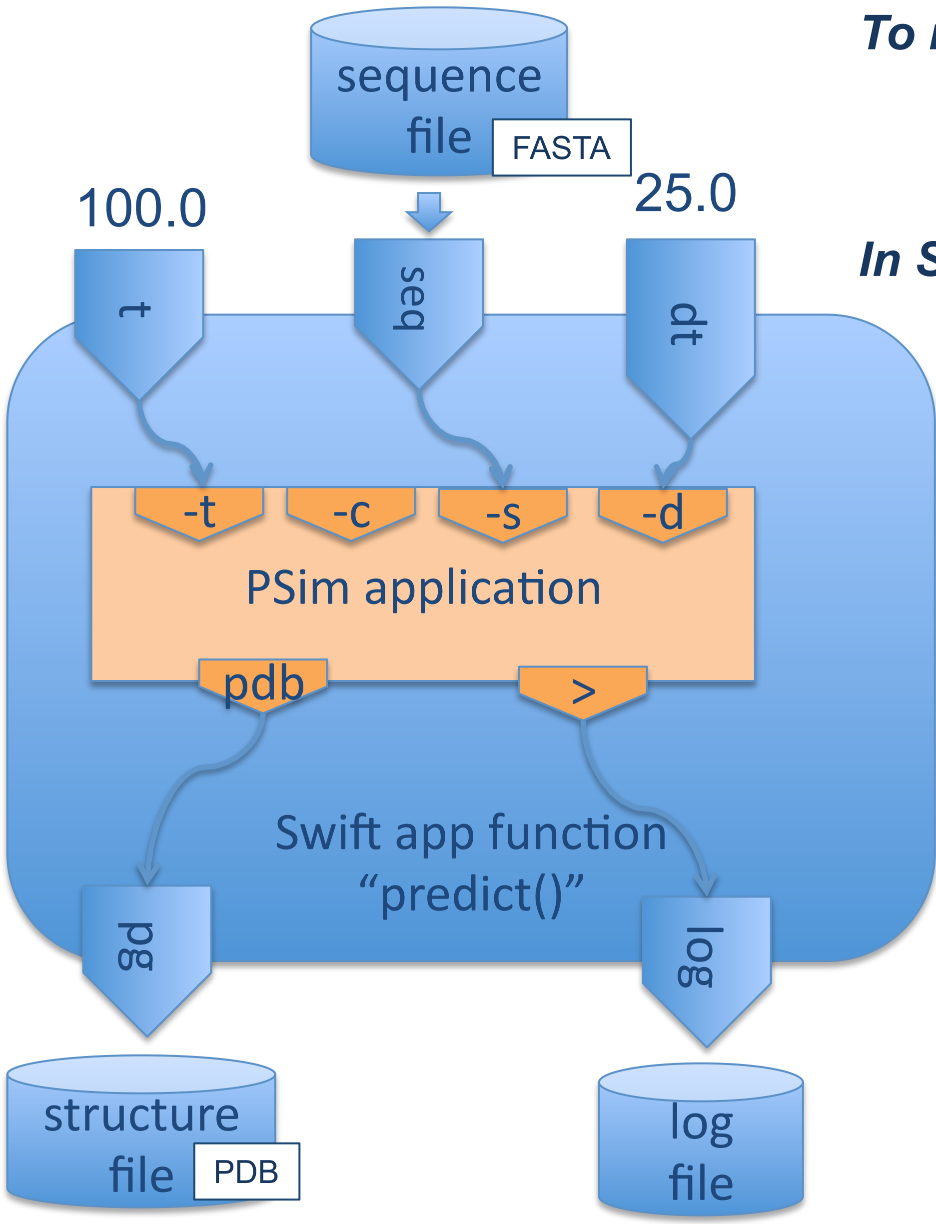
6 All members of this loop are computed in parallel.

app() functions encapsulate application programs



Wrapping applications as Swift functions facilitates data flow, enabling 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:

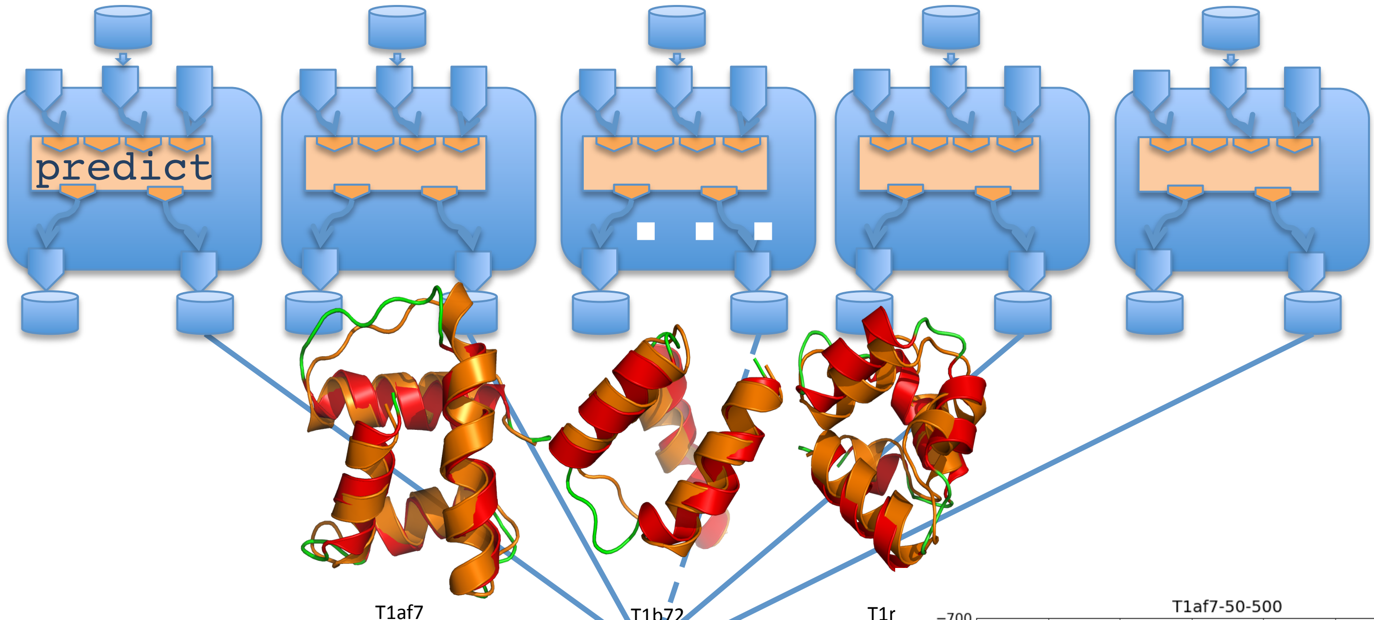
```
app (PDB pg, File log) predict
    (Protein ps, Fload t,
                 Float dt)
{ psim "-t" temp "-c"
       "-s" @ps.fasta "-d" dt
       "-pdb" @pg stdout=@log;
}
Protein p <ext; exec="Pmap",
                id="1ubq">;
PDB structure;
File log;

(structure, log) =
   predict (p, 100., 25.);
```

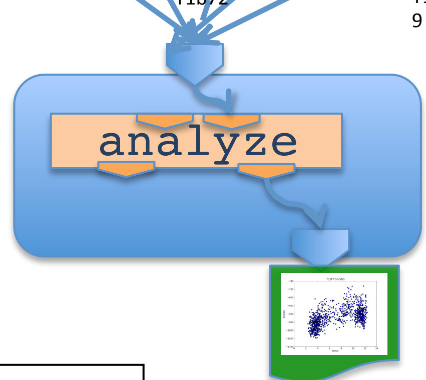

Argonne NATIONAL LABORATORY

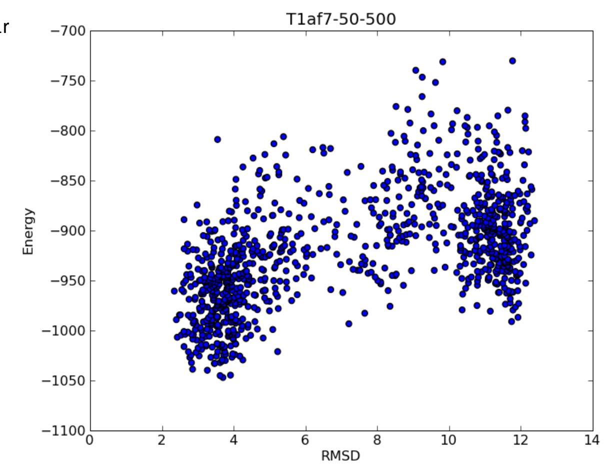
Large scale parallelization with simple loops

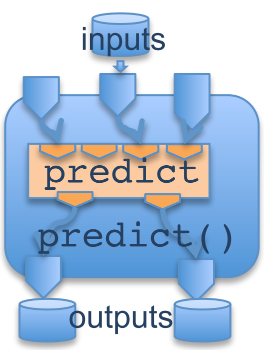
1,000 runs of the “predict” protein folding application




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







LEGEND:

The Swift function **`predict()`** (blue) wraps the application program **predict** (orange).

www.ci.uchicago.edu/swift

Nested loops generate massive parallelism

A typical nested parameter 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 in pSet {
   foreach t in startTemp {
      foreach d in delT {
         ItFix(p, nSim, maxRounds, t, d);
      }
   }
}
```

10 proteins x 1000 runs x 3 rounds x 2 T° x 5 ΔT°’s

= 300K parallel tasks

Complex parallel workflows can be concisely expressed…

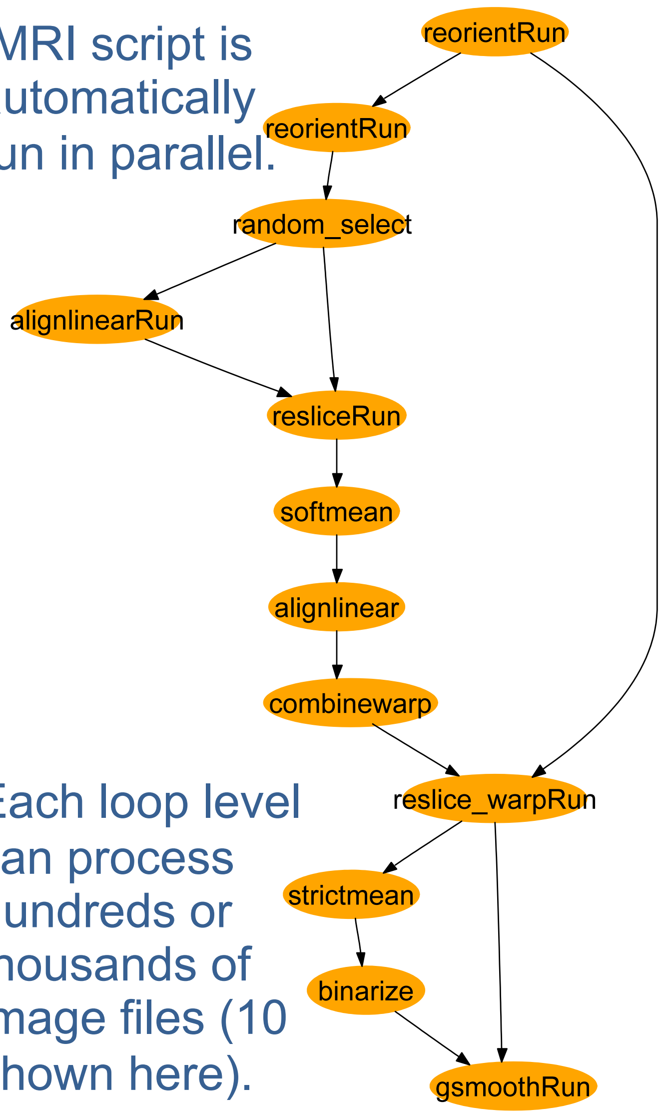
An fMRI preprocessing script expressed as function calls:

```
(Run outr) reorientRun (Run r, string d)
{ foreach Volume iv, i in r.v {
    outr.v[i] = reorient(iv, d);
}
```

```
(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" );
}
```

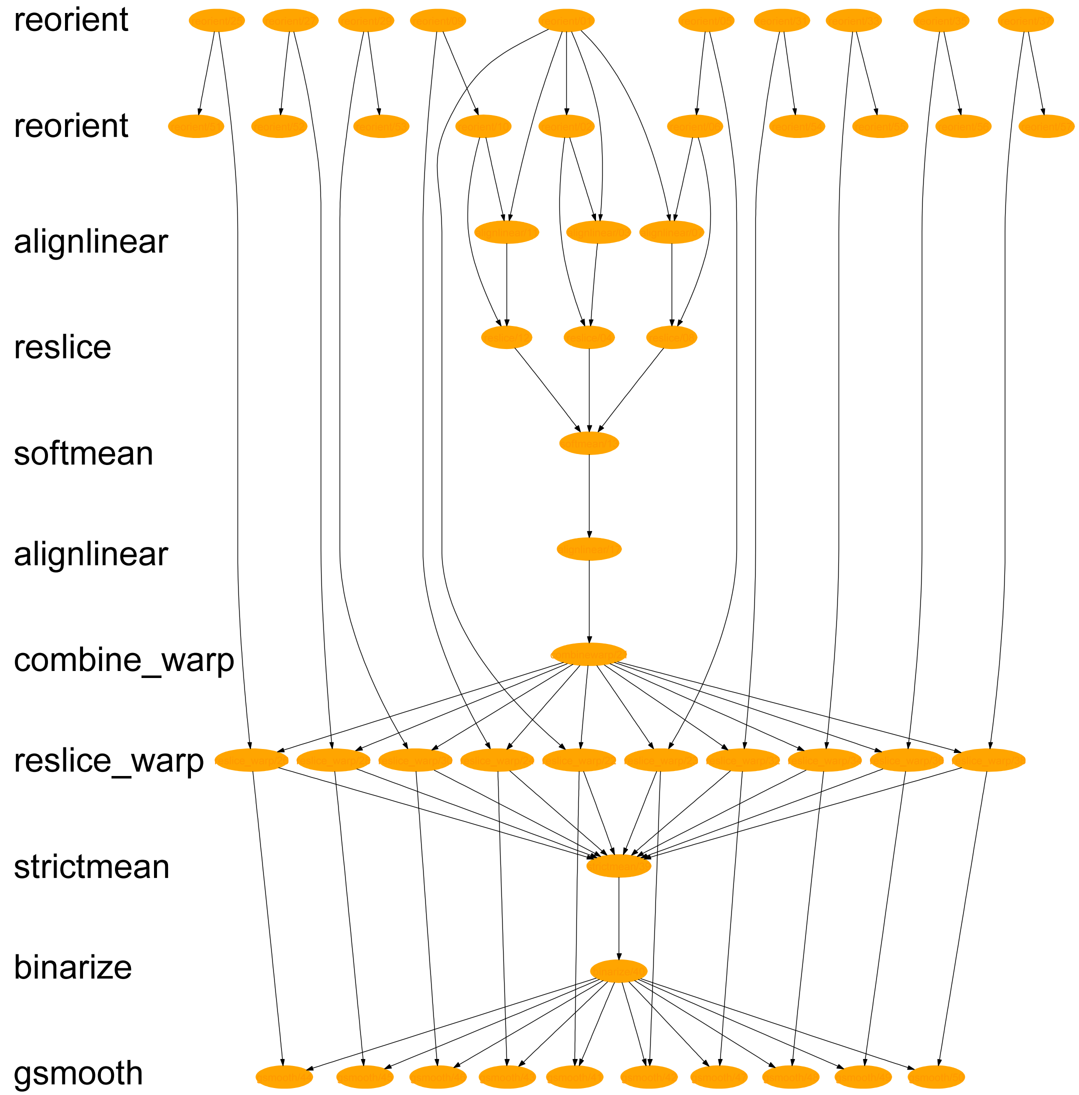
…and Swift automatically executes the script in parallel

fMRI script is automatically run in parallel.

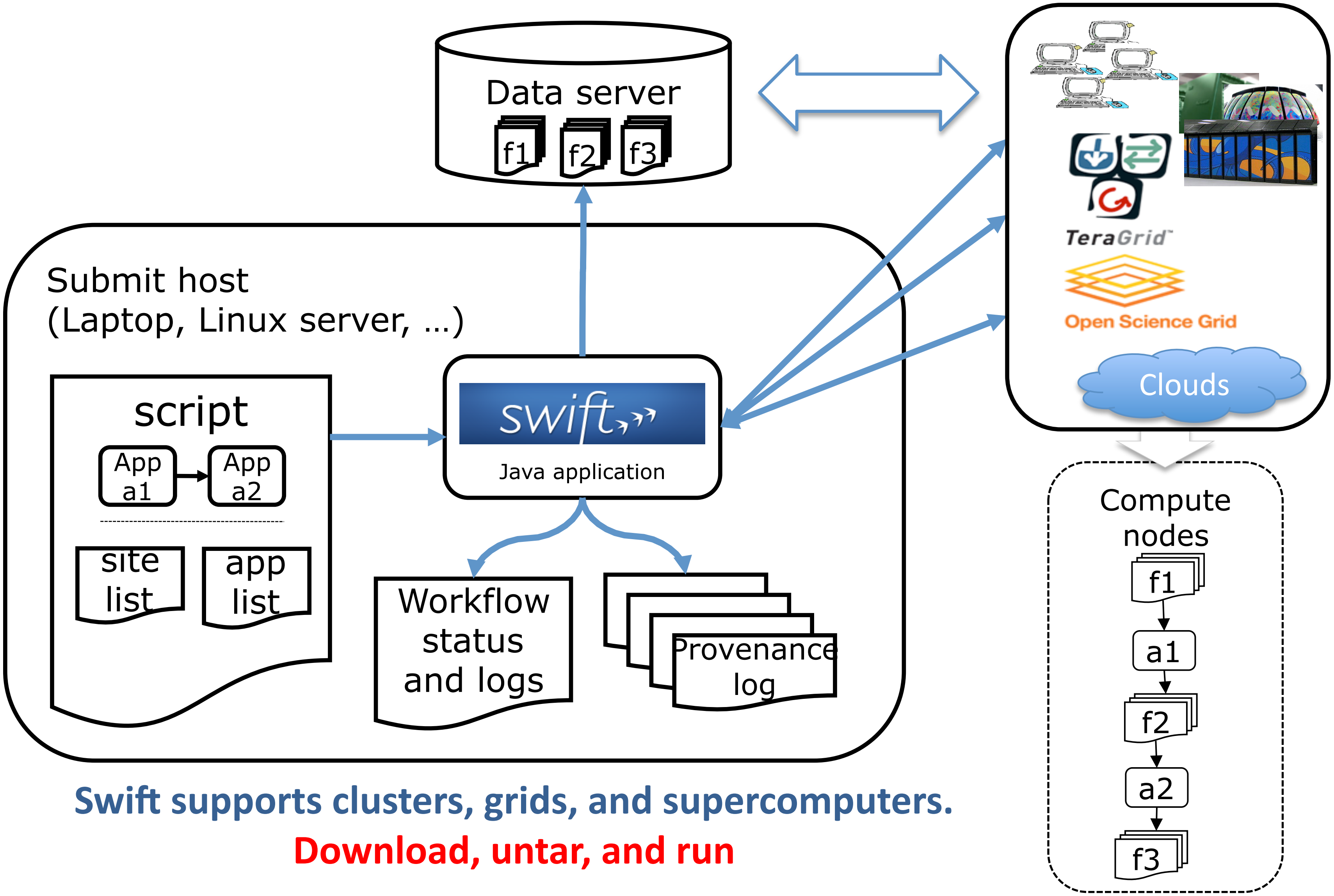


Each loop level can process hundreds or thousands of image files (10 shown here).

Expanded (10 image volume) workflow:



Runtime environment to execute Swift scripts



Swift supports clusters, grids, and supercomputers.

Download, untar, and run

Running Swift on Cray systems

- Tested on XT4/5 and XE6/XK6 systems
  - Beagle, Crow, Franklin, Hopper, Raven, Hera, Kaibab
- Runs out of the box
  - Swift is a Java application: just untar and run
- Swift is a user-level application
  - No modifications to systems software
  - Obtains its resources through Cray PBS scheduler
  - Runs on login host, external host, or compute node
  - Submits jobs to a simple agent running on each node
- Users edit a few files to specify runtime configuration
  - `sites`: job sizes and times; `tc`: app paths; `properties`
- OpenMP apps runs as a normal apps packed on node by Swift
- Runs MPI apps by running Swift under PBS and calling `aprun`

Running Swift on Cray systems – more flexibility

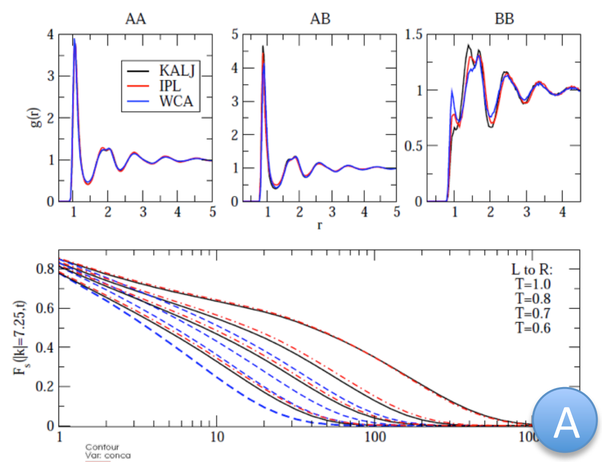
- Can run in a single scheduler job or in multiple jobs
  - Swift adjusts running resources to match the dynamic demand of the workflow
  - Can define pools of resources with different attributes (e.g., select GPU nodes or request longer running jobs)
- Can adjust per-job attributes within a pool
  - Set memory, core topology, runtime, packing ratios
- Can submit a variety of job sizes in a singe run
  - Can dynamically adapt to queue conditions
  - Gives schedulers more opportunity to identify backfill

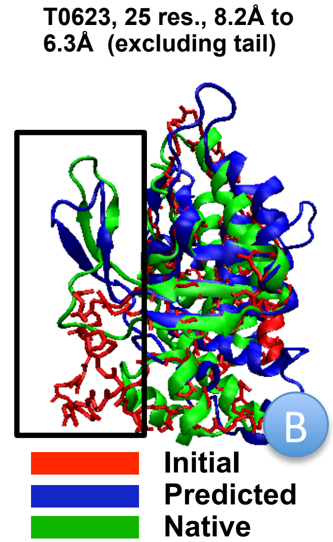
Swift fault tolerance

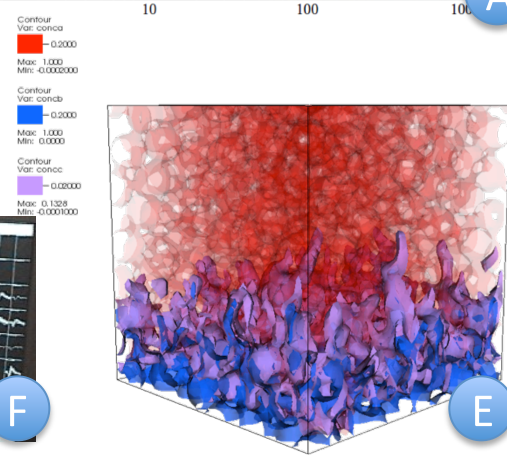
- Swift can retry jobs
  - Up to a user specified limit
  - Can stop on first unrecoverable failure, or continue till no more work can be done
  - Very effective, since Swift can break workflow into many separate scheduler jobs, hence smaller failure units
- Swift can replicate jobs
  - If jobs don't complete in a designated time window, Swift can send copies of the job to other sites or systems
  - The first copy to succeed is used, other copies are removed
- Each app() job can define "failure"
  - Typically non-zero return code
  - Wrapper scripts can decide to mask app() failures and pass back data/logs about errors instead

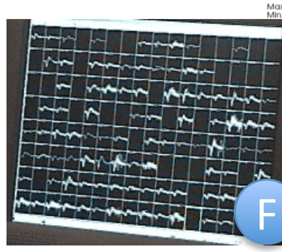
Many-task apps run on Cray XE6: Beagle and Hopper

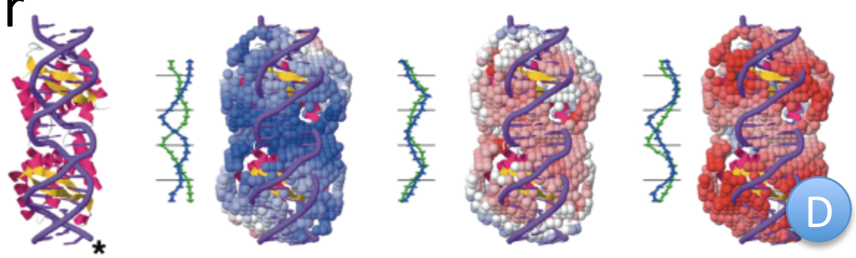
- A Simulation of super-cooled glass materials
- B Protein folding using homology-free approaches
- C Decision making in climate and energy policy
- D Simulation of RNA-protein interaction
- E Multiscale subsurface modeling on Hopper
- F Modeling framework for statistical analysis of neuron activation

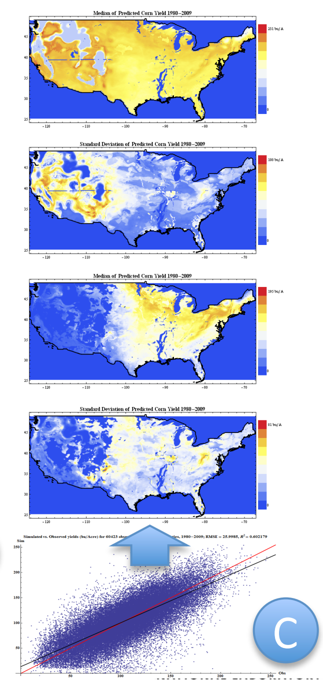








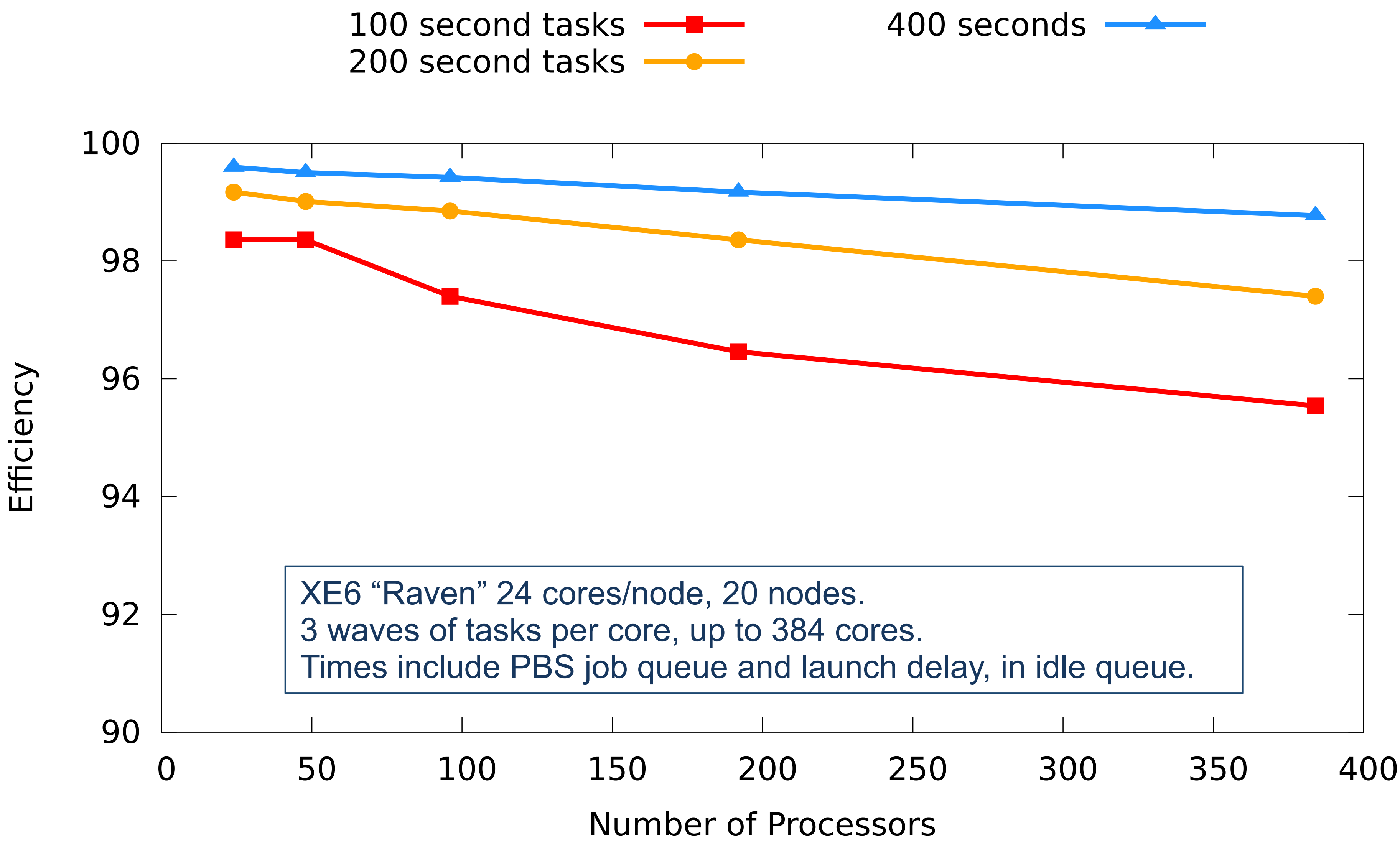




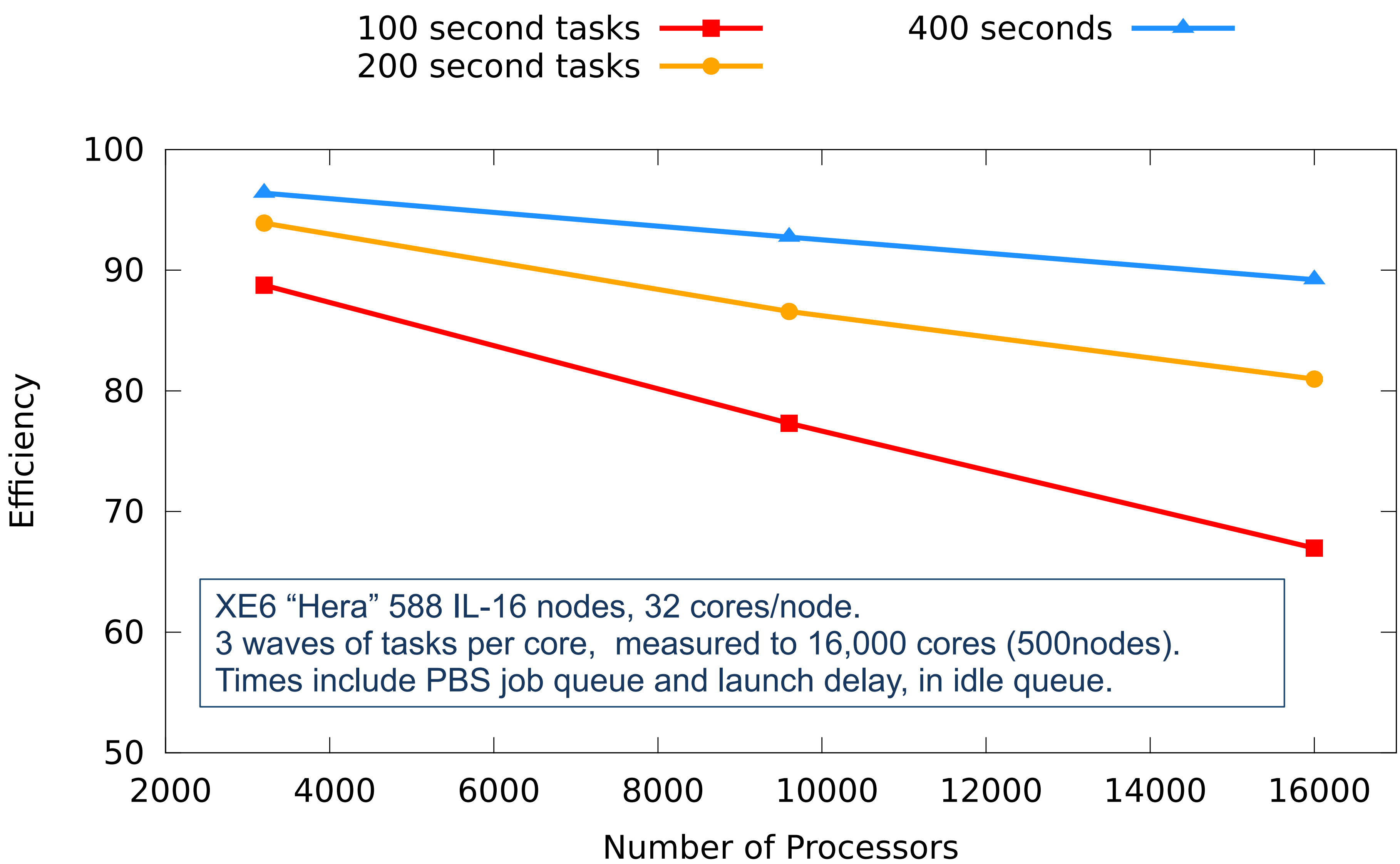
www.ci.uchicago.edu/swift

Computation Institute

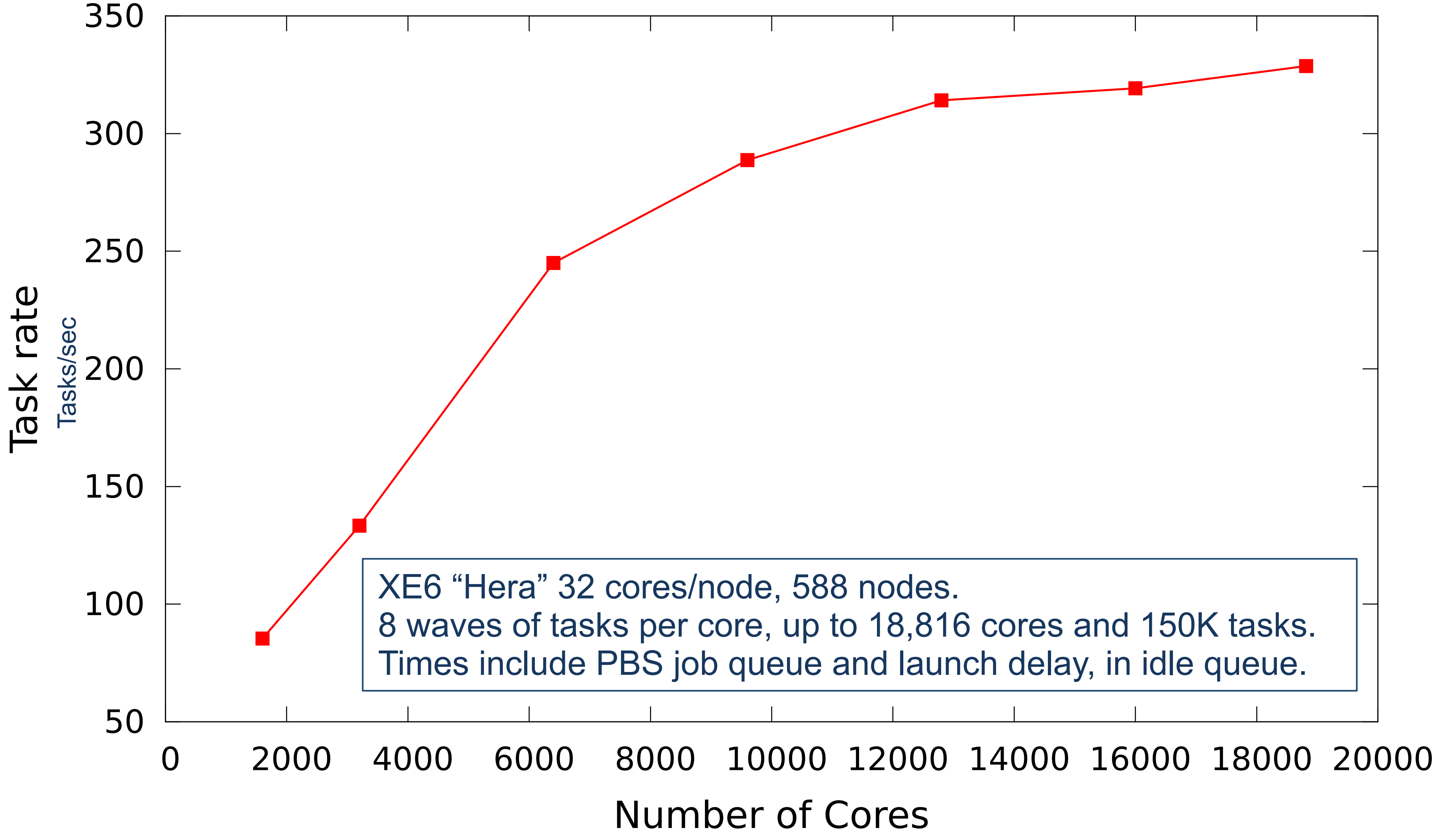
Swift efficiency on Cray XE6 test system "raven"



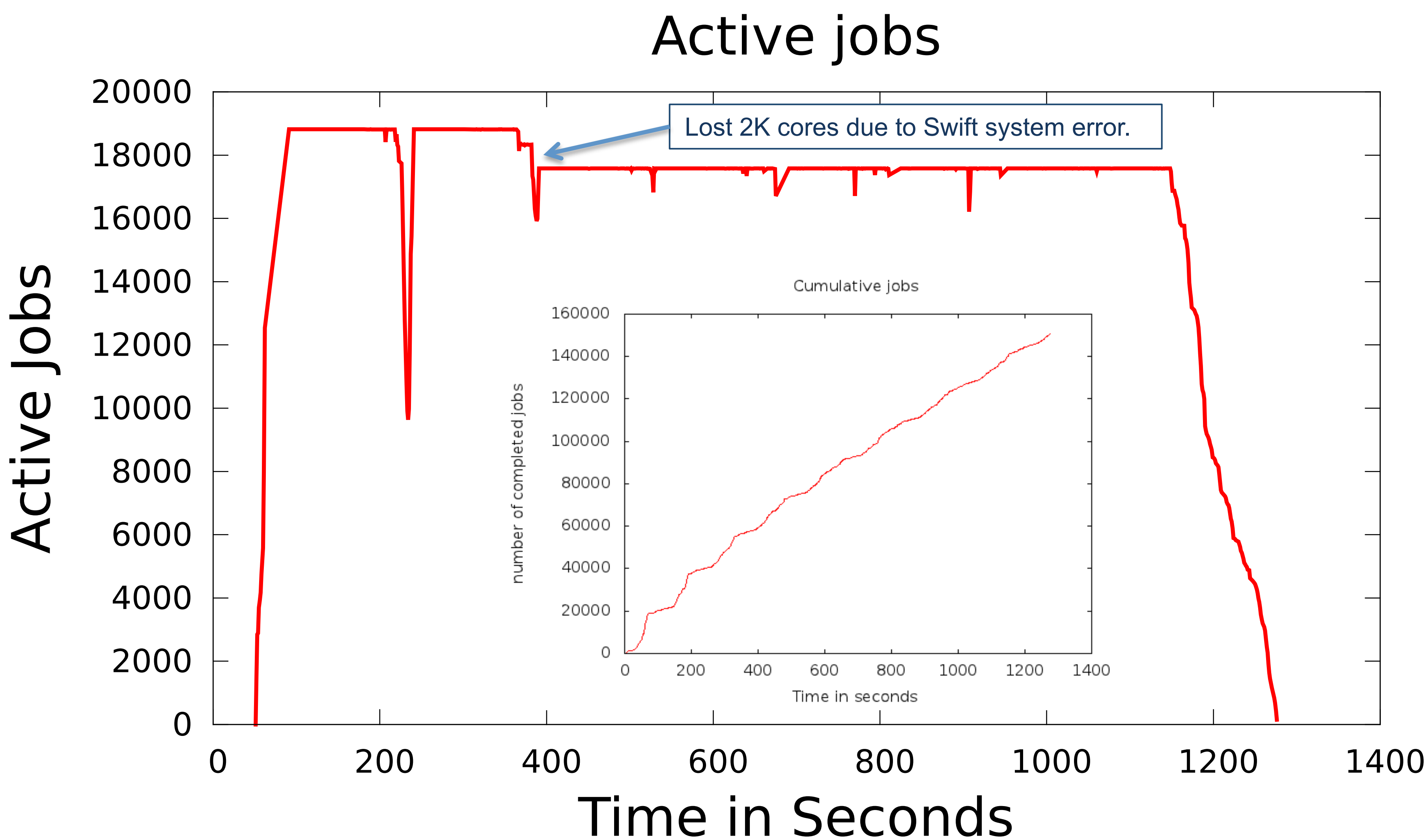
Swift efficiency on Cray XE6 test system “hera”



Swift task rates on Cray XE6 test system “hera”

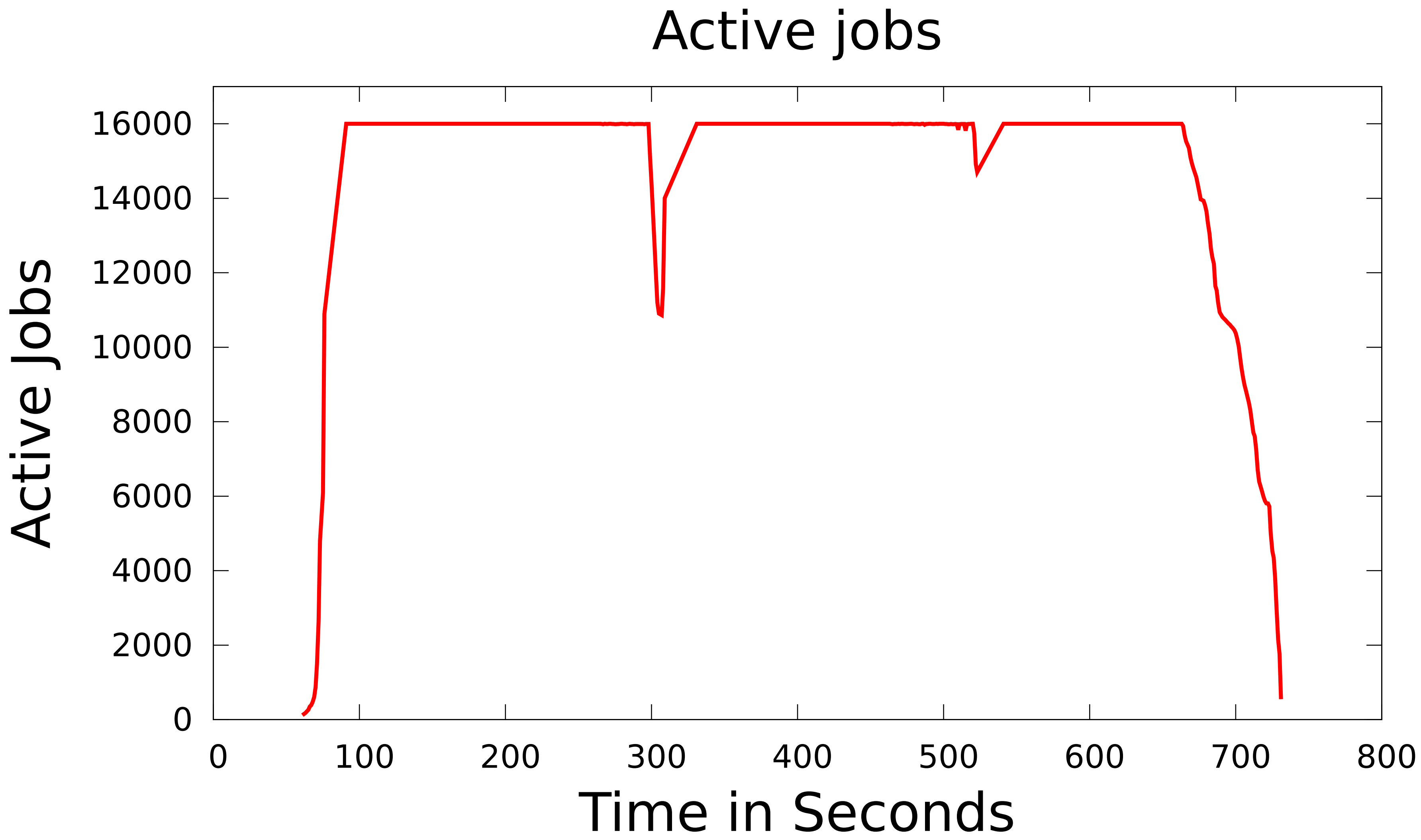


Performance study for DSSAT application



Synthetic test of DSSAT application workload, 152,000 128 sec tasks, 18,816 cores of Cray XK6 Hera, 32 cores/node (2x IL-16) . Average task rate 116/sec, 79% efficiency.

Performance study for DSSAT application

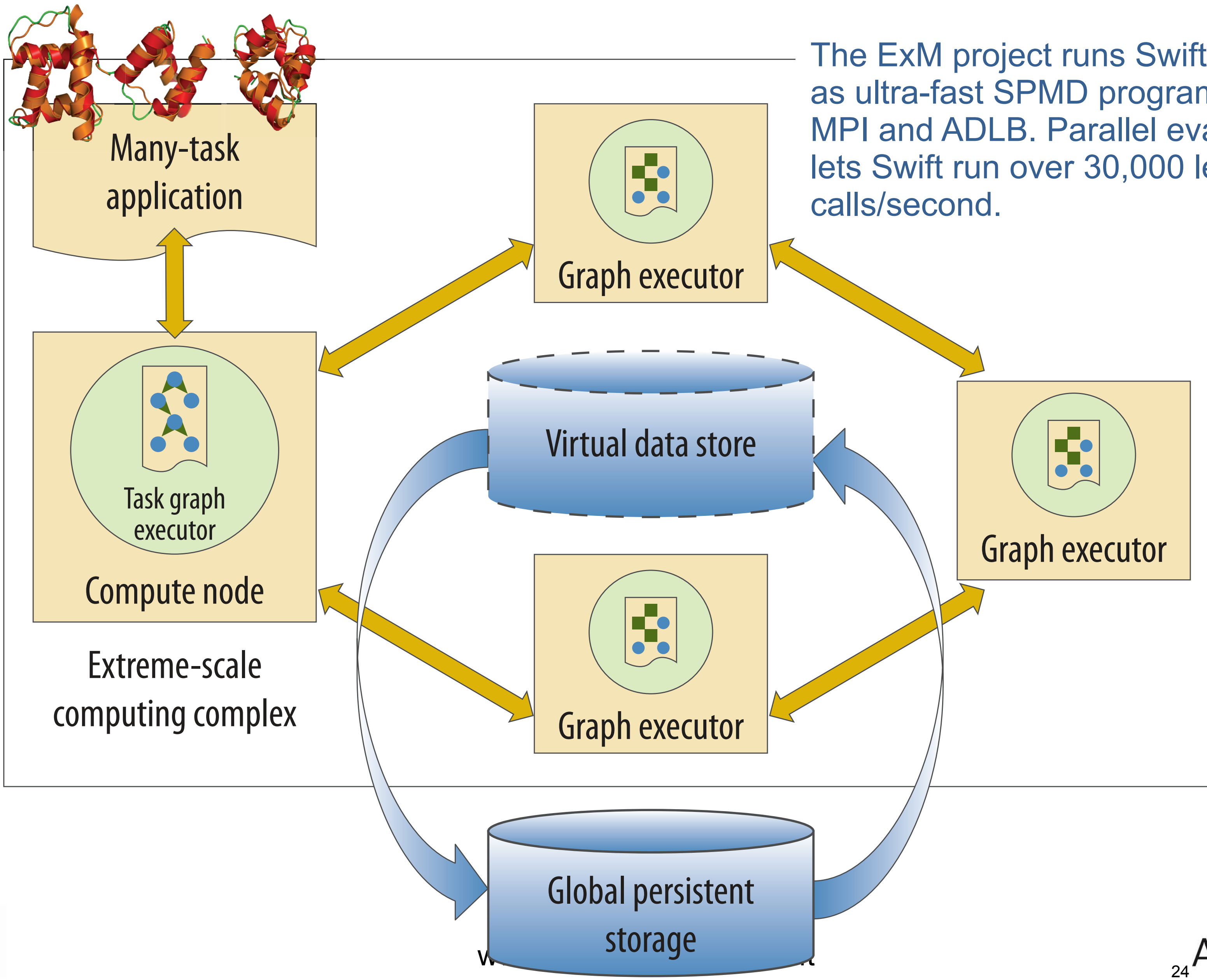


Synthetic test of DSSAT application workload, 48,000 200 sec tasks, 16,000 cores of Cray XK6 Hera, 32 cores/node (2x IL-16) .

ExM: Scaling the many-task model to exascale

- Sponsored under DOE ASCR X-Stack program
- Extend Swift: tasks can be lightweight functions
  - Use Swift for the high-level logic of exascale applications
  - Retain functional semantics of input-process-output
- Highly distributed program evaluation
  - Re-building Swift based on an *intermediate representation* ("TIC") that lends itself to highly parallel evaluation
  - Scales to massive computing complexes
  - Distributed future store accessible in the manner of global arrays
  - Highly distributed program evaluation
  - Optimizations to reduce access to global future store
- Transparent distributed local storage management
  - MosaStore aggregates local/RAM filesystems (POSIX interface)
  - A distributed objects store holds and passes Swift in-memory data

ExM: Scaling the many-task model to exascale

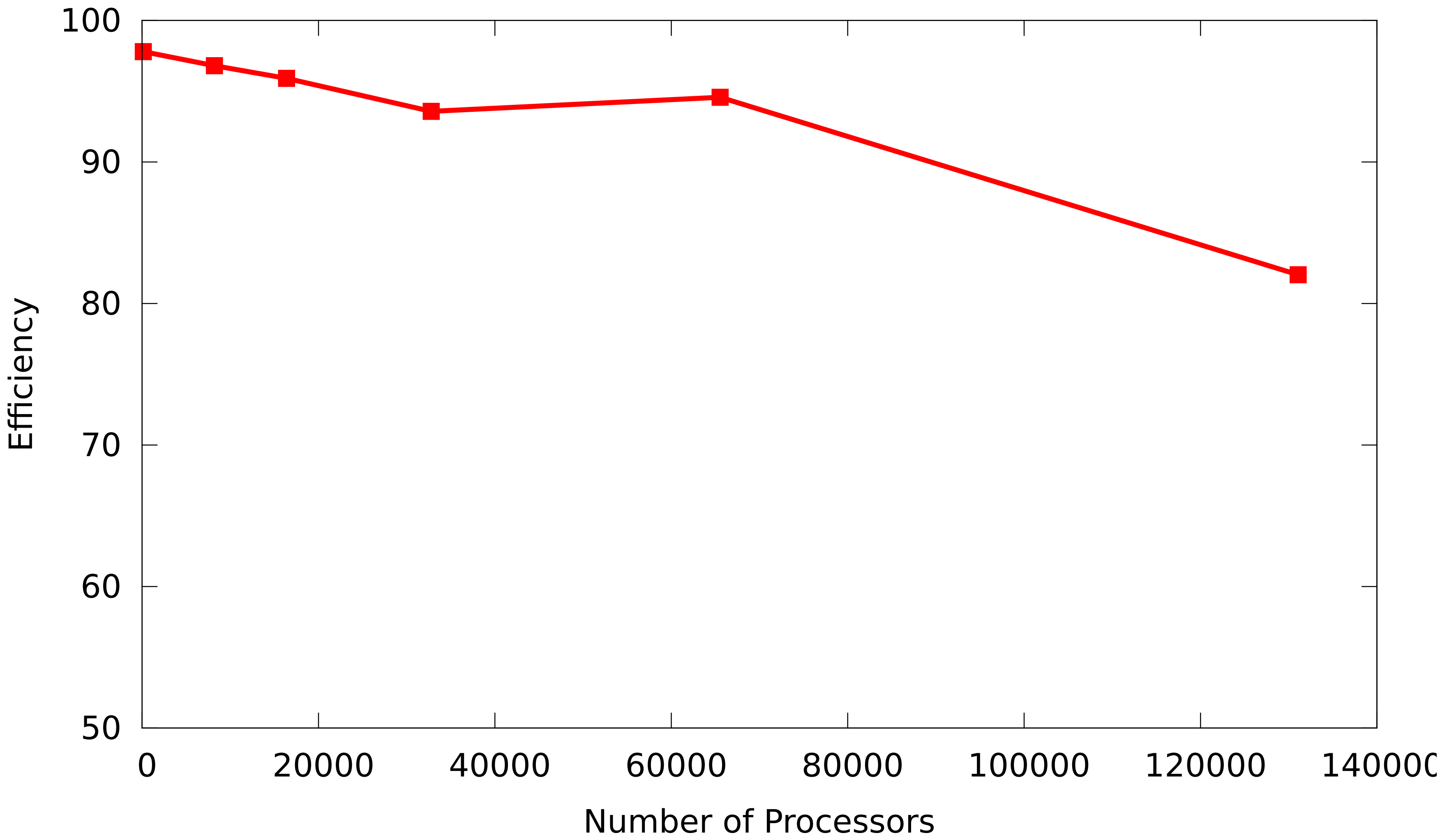


The ExM project runs Swift programs as ultra-fast SPMD programs under MPI and ADLB. Parallel evaluation lets Swift run over 30,000 leaf app calls/second.

Computation Institute

Argonne NATIONAL LABORATORY

Swift-ExM efficiency – to 128K cores



Prototype Swift-ExM on BG/P Intrepid, 32,768 nodes, 131,072 cores.
100 second tasks (processes = #cores)

Conclusion: Motivation for *Swift*

- Enhance scientific productivity
  - Location – and paradigm – independence: Same scripts run on workstations, clusters, clouds, grids, and petascale supercomputers
  - Automation of dataflow, resource selection and error recovery
- Enable and motivate collaboration
  - Community libraries of techniques, protocols, methods
  - Designed for recording the provenance of all data produced to facilitate scientific processes

swift

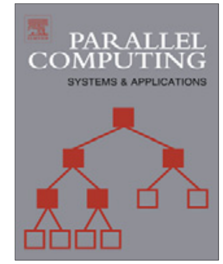
- **Swift is a parallel scripting language** for multicores, clusters, grids, clouds, and **supercomputers**
  - *for loosely-coupled applications - application and utility programs linked by exchanging files*
  - *debug on a laptop, then run on a Cray*
- **Swift is easy to write**
  - *it's a simple high-level functional language with C-like syntax*
  - *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 – Swift acts as a self-contained grid or cloud client*
  - ***Swift automatically runs scripts in parallel** – usually with no user input*
- **Swift is fast**: based on a powerful, efficient, scalable and flexible Java execution engine
  - *scales readily to millions of tasks*
- Swift **usage** is growing:
  - *applications in neuroscience, proteomics, molecular dynamics, biochemistry, economics, statistics, earth systems science, and more.*

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

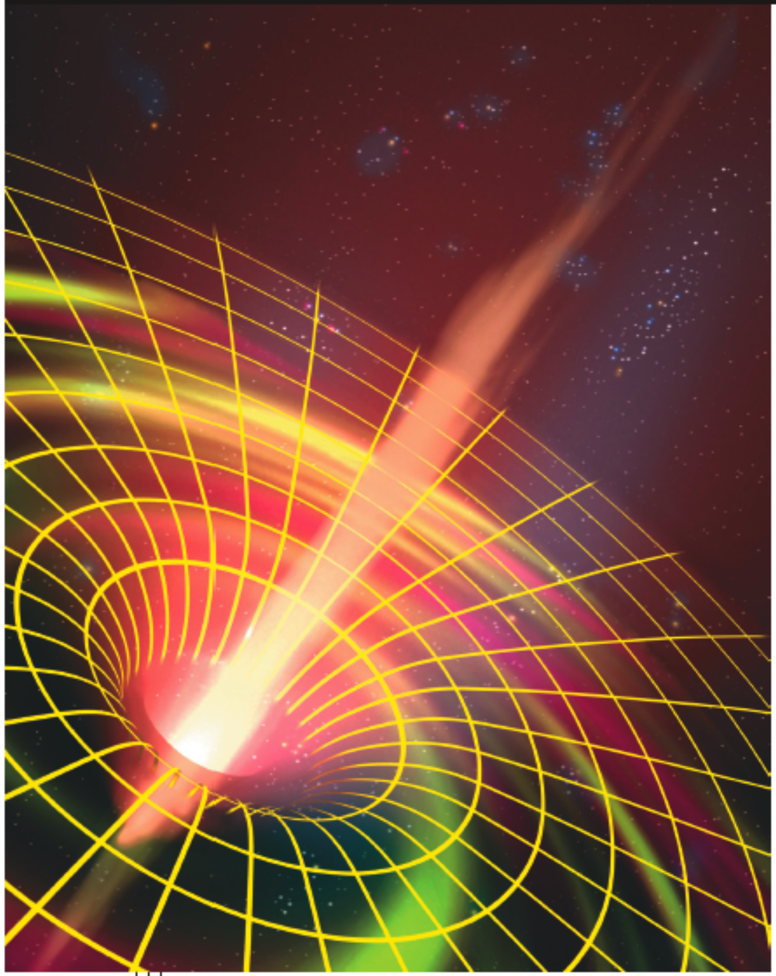
Keywords:
Swift
Parallel programming
Scripting
Dataflow

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.

Parallel Computing, Sep 2011

COVER FEATURE



PARALLEL SCRIPTING FOR APPLICATIONS AT THE PETASCALE AND BEYOND

Michael Wilde, Ian Foster, Kamil Iskra, and Pete Beckman,
University of Chicago and Argonne National Laboratory

Zhao Zhang, Allan Espinosa, Mihael Hategan, and Ben Clifford, *University of Chicago*

Ioan Raicu, *Northwestern University*

IEEE COMPUTER, Nov 2009

Acknowledgments

- Swift is supported in part by NSF grants OCI-1148443, OCI-721939, OCI-0944332, and PHY-636265, NIH DC08638, DOE and UChicago LDRD and SCI programs
- ExM is supported by the DOE Office of Science, ASCR Division
- Structure prediction supported in part by NIH
- The Swift team:
  - Mihael Hategan, Justin Wozniak, David Kelly, Jon Monette, Ketan Maheshwari, Ian Foster, Dan Katz, Mike Wilde, Tim Armstrong, Ben Clifford, Zhao Zhang
- GPSI Science portal:
  - Mark Hereld, Tom Uram, Wenjun Wu, Mike Papka
- Java CoG Kit used by Swift developed by:
  - Mihael Hategan, Gregor Von Laszewski, and many collaborators
- ZeptoOS
  - Kamil Iskra, Kazutomo Yoshii, and Pete Beckman
- Scientific application collaborators whose work is described in this talk:
  - Karl Freed, Tobin Sosnick, Glen Hocky, Joe Debartolo, Aashish Adhikari, Marc Parisien, Joshua Elliott, Meredith Franklin, Todd Muson, Rob Jacob, Sheri Mickelson (Argonne); John Dennis, Matthew Woitaszek, Karen Schuchartd, Kushbu Agarwal, Jinbo Xu
- Deepest thanks to Dave Strenski and Duncan Roweth of Cray, for providing benchmarking facilities, incredible assistance, and for presenting this talk.