

Workflow Systems

Ioan Raicu

Computer Science Department
Illinois Institute of Technology

CS 595: Data-Intensive Computing
November 2nd, 2011

Swift and e-Science

- Swift is a system for the rapid and reliable specification, execution, and management of large-scale science and engineering workflows. It supports applications that execute many tasks coupled by disk-resident datasets - as is common, for example, when analyzing large quantities of data or performing parameter studies or ensemble simulations.
- For example:
 - Cancer research: looking for previously unknown protein changes by comparing mass spectrum data with data known about proteome.
 - A monte-carlo simulation of protein folding, 10 proteins, 1000 simulations for each configuration, inside simulated annealing algorithm with $2 \times 5 = 10$ different parameter values. Each simulation component takes ~ 5 CPU-minutes, so about ~ 1 CPU-year for a whole run; producing 10...100Gb of data.

Other Work

- Coordination language

- ◆ Linda[Ahuja,Carriero86], Strand[Foster,Taylor90], PCN[Foster92]
- ◆ Durra[Barbacci,Wing86], MANIFOLD[Papadopoulos98]
- ◆ Components programmed in specific language (C, FORTRAN) and linked with system

- “Workflow” languages and systems

- ◆ Taverna[Oinn,Addis04], Kepler[Ludäscher,Altintas05], Triana [Churches,Gombas05], Vistrail[Callahan,Freire06], DAGMan, Star-P
- ◆ XPDL[WfMC02], BPEL[Andrews,Curbera03], and BPML[BPML02], YAWL[van de Aalst,Hofstede05], Windows Workflow Foundation [Microsoft05]

Other Work

	SwiftScript	BPEL	XPDL	MW Wflow	DAGMan	Tavenna	Triana	Kepler	Vistrail	Star-P
Scales to Grids	++	-	-	-	++	-	-	-	-	+
Typing	++	++	++	++	-	-	-	+	-	+
Iteration	++	-/+	-	+	-	-	-	+	-	+
Scripting	++	-	-	+	+	+	-	-	+	++
Dataset Mapping	+	-	-	-	-	-	-	-	-	-
Service Interop	+	-	+	-	-	-	-	+	-	-
Subflow/comp.	+	-	+	+	-	-	+	+	-	+
Provenance	+	-	-	+	-	+	-	+	+	-
Open source	+	+	+	-	+	+	+	+	+	-

“A 4x200 flow leads to a 5 MB BPEL file ... chemists were not able to write in BPEL”

[Emmerich,Buchart06]

A brief history of SwiftScript

- ~2003: VDL - the Virtual Data Language.
express directed acyclic graphs of unix processes
processes take input and produce output through files
'virtual data' - when needed, materialise data either by copying from elsewhere or by deriving it from other data that is available
Lots of thinking about "graph transformations"
- ~2006: VDL2 (which became SwiftScript)
 - key features:
 - iterating over collections of files in the language
 - accidentally became Turing-complete
- ~2010: still going - language tweaks, scaling improvements

Target programmers

- Scientific programmers use some science-domain specific language to write the "science" bit of their application (eg R for statistics, Root for particle physics).
- They aren't "high performance" or "distributed system" programmers.
- Want to help them use "big" systems to run their application - eg machines with 10^5 CPU cores.
- Traditional MPI (Message Passing Interface) is hard to think about.
- Swift tries to provide an easier model that still allows many applications to be expressed, and performed with reasonable efficiency.
- SwiftScript is the language for programming in that model.

Mappers and file types

- file output <"output.txt">; Declares output to be a variable whose value is stored in the file system rather than in-core.
- <"output.txt"> means that the value is stored in a file output.txt (this can be a URL)
- This is a simple example with a literal single filename.
 - More complex syntax allows mapping arrays of files, with more dynamic behaviour (eg generating filename patterns at runtime)
- We can omit the <...> mapping expression in which case Swift will make up a filename - useful for intermediate files.

app procedures

- `app (file o) count(file i) { uniq "-c" stdin=@i stdout=@o; }`
This is how the real work gets done - by getting some other science-domain specific program to do it.
- app procedures execute unix processes, but not like `system()` or `runProcess`
- The environment in which an app procedure runs is constrained:
Application will start in "some directory, somewhere".
There, it will find its input files, and there it should leave its output files.
- Applications need to be referentially transparent (but SwiftScript doesn't clearly define what equivalence is)

Executing an app { } procedure

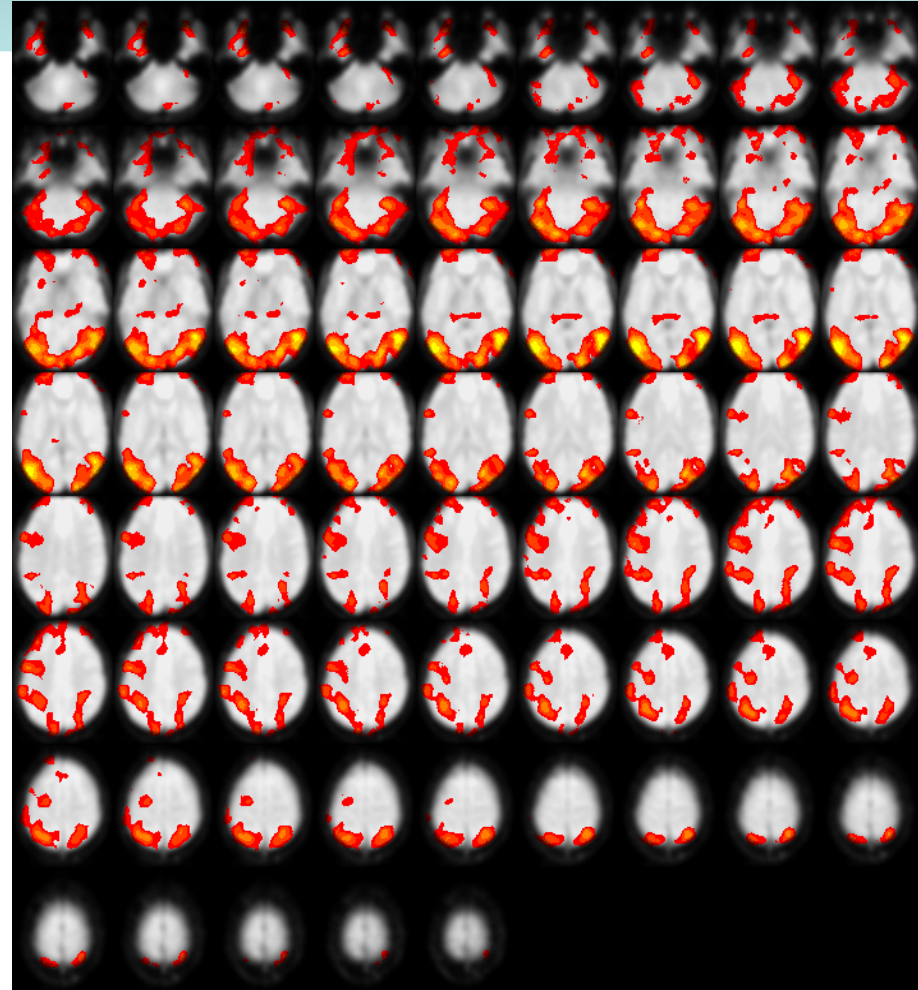
- Pick an execution site
- Transfer input files there (if they are not already cached there)
- Put the job in an execution queue at the execution site
- Wait for execution to finish
- Transfer output files back
- Check everything worked ok

Case Study

Functional MRI (fMRI) Data Center

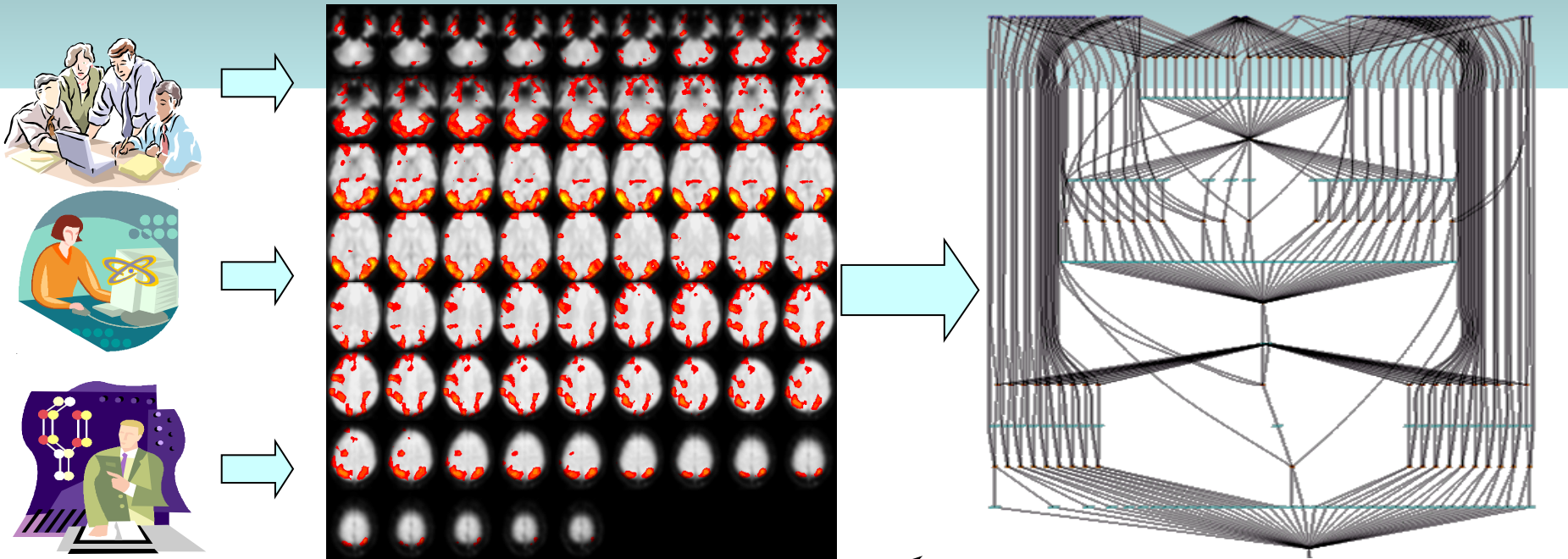
- Online repository of neuroimaging data
- A typical study comprises
 - 3 groups,
 - 20 subjects/group,
 - 5 runs/subject,
 - 300 volumes/run

→ 90,000 volumes, 60 GB raw →
1.2 million files processed
- 100s of such studies in total

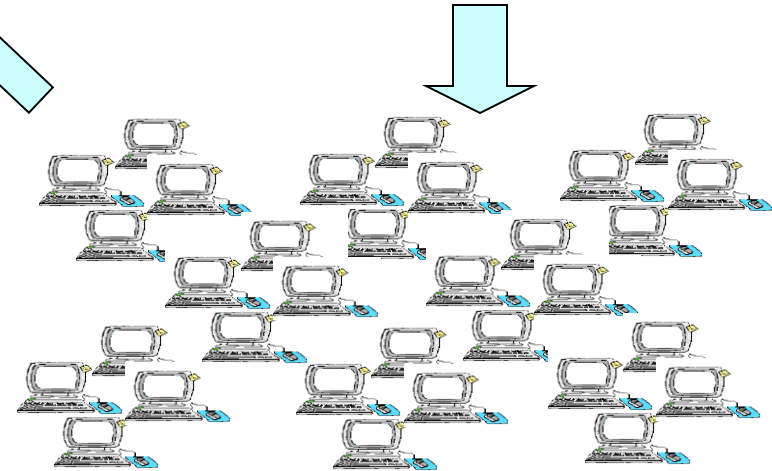


<http://www.fmridc.org>

fMRI Data Analysis



- Large user base
 - ◆ World wide collaboration
 - ◆ Thousands of requests
- Wide range of analyses
 - ◆ Testing, production runs
 - ◆ Data mining
 - ◆ Ensemble, Parameter studies



Three Obstacles to Creating a Community Resource

- Accessing messy data
 - Idiosyncratic layouts & formats
 - Data integration a prerequisite to analysis
- Describing & executing complex computations
 - Expression, discovery, reuse of analyses
 - Scaling to large data, complex analyses
- Making analysis a community process
 - Collaboration on both data & programs
 - Provenance: tracking, query, application

The Swift Solution

- Accessing messy data
 - Idiosyncratic layouts & formats
 - Data integration a prerequisite to analysis
- Implementing complex computations
 - Expression, discovery, reuse of analyses
 - Scaling to large data, complex analyses
- Making analysis a community process
 - Collaboration on both data & programs
 - Provenance: tracking, query, application

XDTM

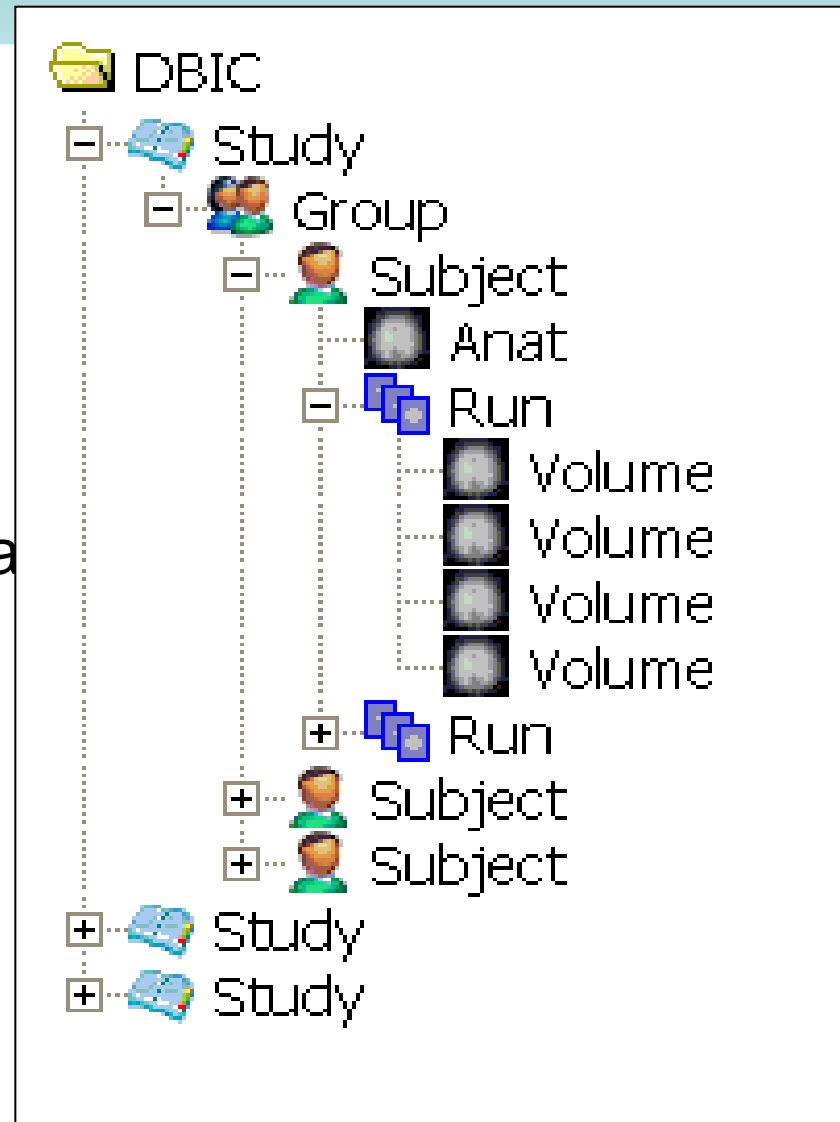
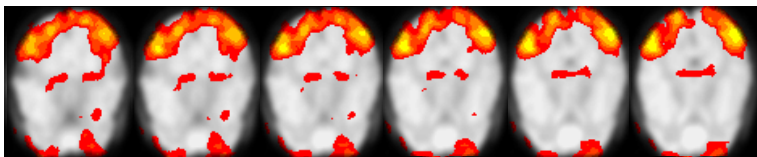
SwiftScript

Karajan
+Falkon

VDC

The Messy Data Problem (1)

- Scientific data is often logically structured
 - E.g., hierarchical structure
 - Common to map functions over dataset members
 - Nested map operations can scale to millions of objects



The Messy Data Problem (2)

- Heterogeneous storage format & access protocols
 - Same dataset can be stored in text file, spreadsheet, database, ...
 - Access via filesystem, DBMS, HTTP, WebDAV, ...
- Metadata encoded in directory and file names
- Hinders program development, composition, execution

```
./knottastic
drwxr-xr-x 4 yongzh users 2048 Nov 12 14:15 AA
drwxr-xr-x 4 yongzh users 2048 Nov 11 21:13 CH
drwxr-xr-x 4 yongzh users 2048 Nov 11 16:32 EC

./knottastic/AA:
drwxr-xr-x 5 yongzh users 2048 Nov 5 12:41 04nov06aa
drwxr-xr-x 4 yongzh users 2048 Dec 6 12:24 11nov06aa

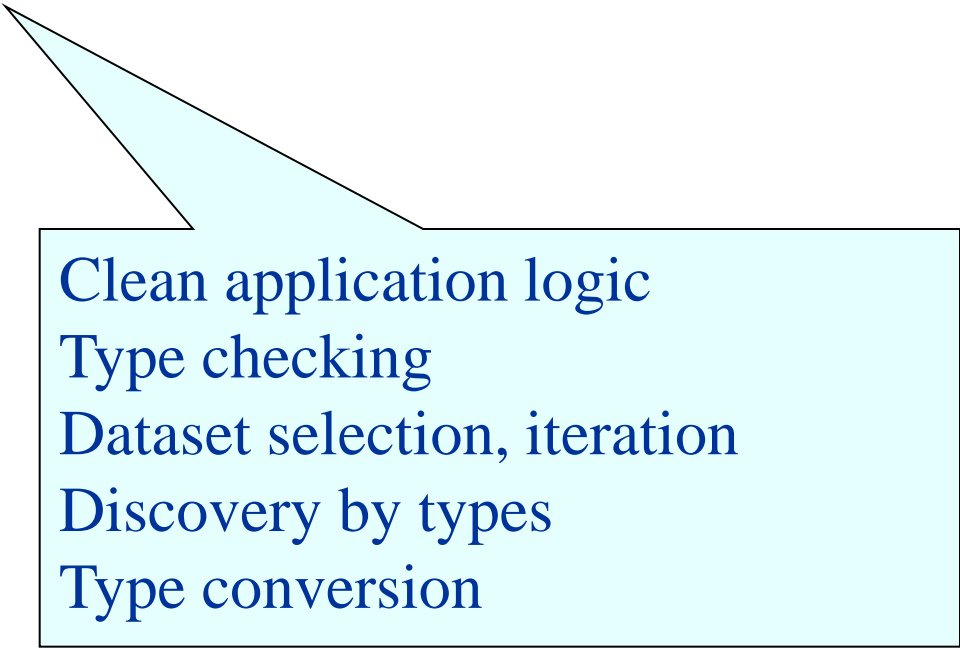
./knottastic//AA/04nov06aa:
drwxr-xr-x 2 yongzh users 2048 Nov 5 12:52 ANATOMY
drwxr-xr-x 2 yongzh users 49152 Dec 5 11:40 FUNCTIONAL

./knottastic/AA/04nov06aa/ANATOMY:
-rw-r--r-- 1 yongzh users 348 Nov 5 12:29 coplanar.hdr
-rw-r--r-- 1 yongzh users 16777216 Nov 5 12:29 coplanar.img

./knottastic/AA/04nov06aa/FUNCTIONAL:
-rw-r--r-- 1 yongzh users 348 Nov 5 12:32 bold1_0001.hdr
-rw-r--r-- 1 yongzh users 409600 Nov 5 12:32 bold1_0001.img
-rw-r--r-- 1 yongzh users 348 Nov 5 12:32 bold1_0002.hdr
-rw-r--r-- 1 yongzh users 409600 Nov 5 12:32 bold1_0002.img
-rw-r--r-- 1 yongzh users 496 Nov 15 20:44 bold1_0002.mat
-rw-r--r-- 1 yongzh users 348 Nov 5 12:32 bold1_0003.hdr
-rw-r--r-- 1 yongzh users 409600 Nov 5 12:32 bold1_0003.img
```

SwiftScript

- Typed parallel programming notation
 - XDTM as data model and type system
 - Typed dataset and procedure definitions
- Scripting language
 - Implicit data parallelism
 - Program composition from procedures
 - Control constructs (foreach, if, while, ...)



Clean application logic
Type checking
Dataset selection, iteration
Discovery by types
Type conversion

Swift Runtime System

- Runtime system for SwiftScript
 - Translate programs into task graphs
 - Schedule, monitor, execute task graphs on local clusters and/or distributed Grid resources
 - Annotate data products with provenance metadata
- Grid scheduling and optimization
 - Lightweight execution engine: **Karajan**
 - **Falkon**: lightweight dispatch, dynamic provisioning
 - Grid execution: site selection, data movement
 - Caching, pipelining, clustering, load balancing
 - Fault tolerance, exception handling

app abstraction facilitates (1/2): flexibility in execution site

- There are many different execution resources in the world: clusters on your campus, supercomputers, your own laptop.
- It is useful to be able to choose and switch between sites, and choose between different mechanisms for accessing a site, because:
 - your usual site is broken today
 - someone is developing a better mechanism (higher performance) for submitting to your usual site (ongoing r&d there)
- you want to use the combined power of several sites at once (research question: if many sites available, which is best to use?)

app abstraction facilitates (2/2): reliability mechanisms

- Failure happens a lot in our target environments (integer percentages in some environments) so reliability is not "a nice feature to have" - it is essential.
- Retries: if an application execution fails, we try it 2 more times
- Restarts: if retries fail, then the whole script fails (eventually). Maybe want to restart manually where we left off. Assume that app blocks are expensive and everything else is cheap, so start the script from beginning again, skipped apps that we've already run (using a log file)
- Replication: deals with a softer class of failure. Sometimes an app goes into a queue and sits "forever" (really forever, or perhaps much longer than most other apps). We can launch a new attempt to run the app, without killing the original. When one starts, we kill the other(s)

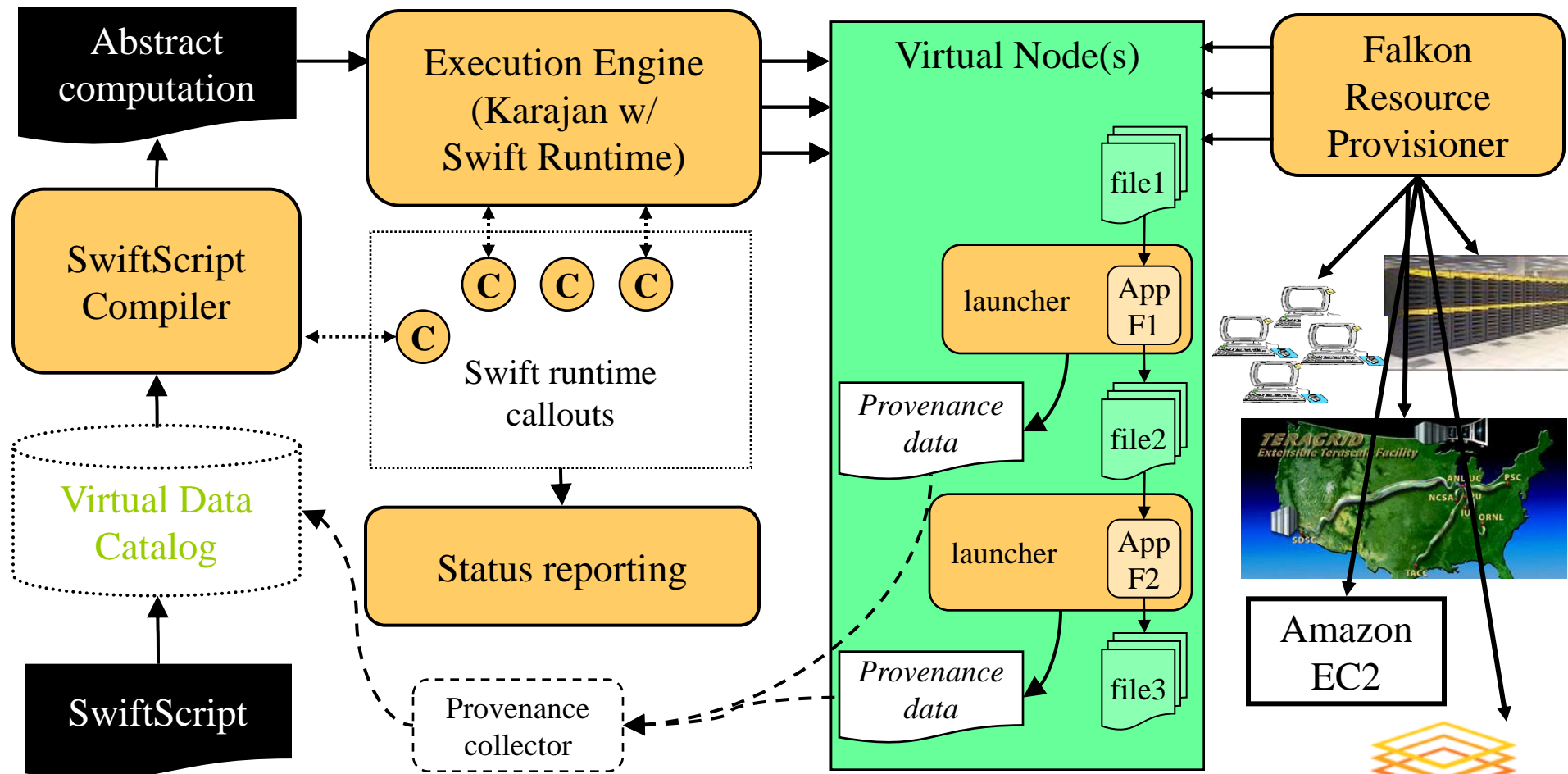
Swift Architecture

Specification

Scheduling

Execution

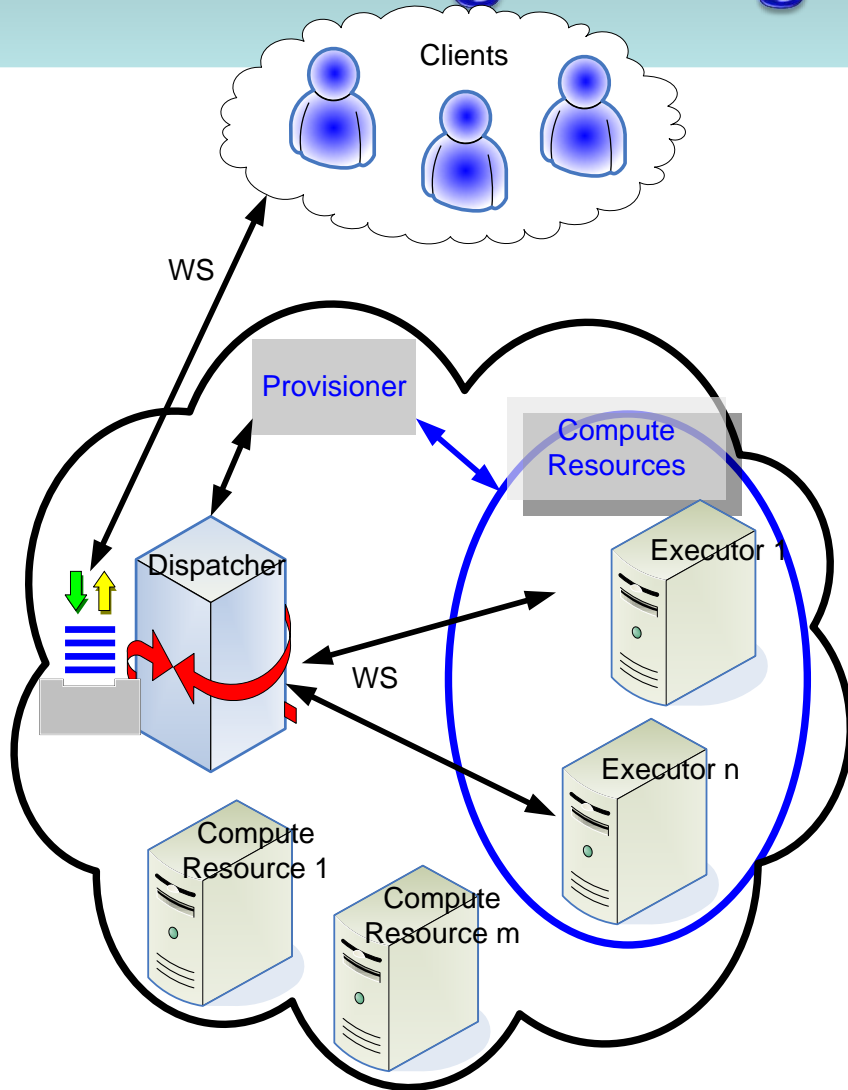
Provisioning



Swift uses Karajan Workflow Engine

- Fast, scalable lightweight threading model
- Suitable constructs for control flow
- Flexible task dependency model
 - “Futures” enable pipelining
- Flexible provider model allows for use of different run time environments
 - Desktop, clusters, Grids
 - Flow controlled to avoid resource overload
- Workflow client runs from a Java container

Swift Uses Falkon Lightweight Execution Service



- **Falkon dynamic provisioner:**
 - ◆ Monitors **demand** (incoming user requests)
 - ◆ Manages **supply**: selects resources; creates executors (via Globus GRAM+LRM)
 - ◆ Various decision strategies for acquisition and release
- **Falkon executor**
 - ◆ 440 tasks/sec max
 - ◆ 54,000 executors
 - ◆ millions of tasks

Swift running on BlueGene/P

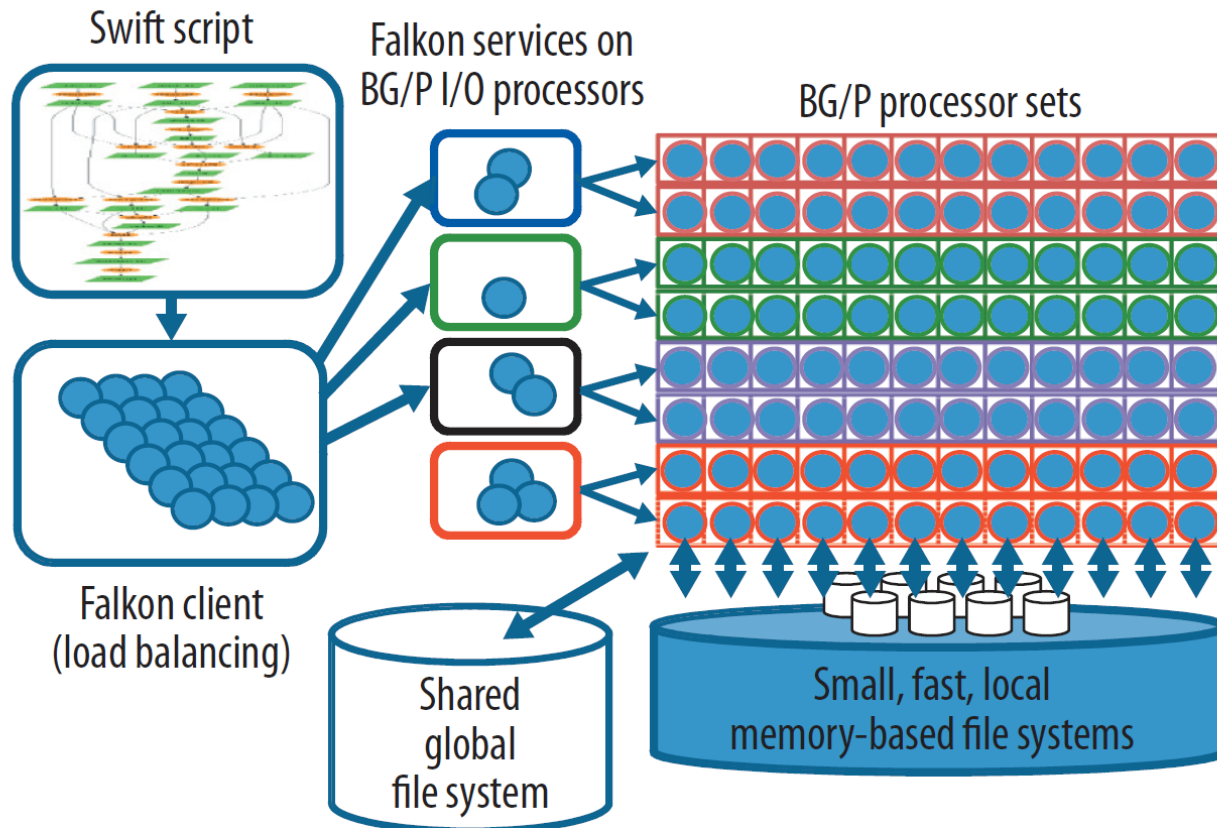


Figure 3. Swift scripts execute using the Falkon distributed resource manager on the BG/P architecture.

Swift running on BlueGene/P

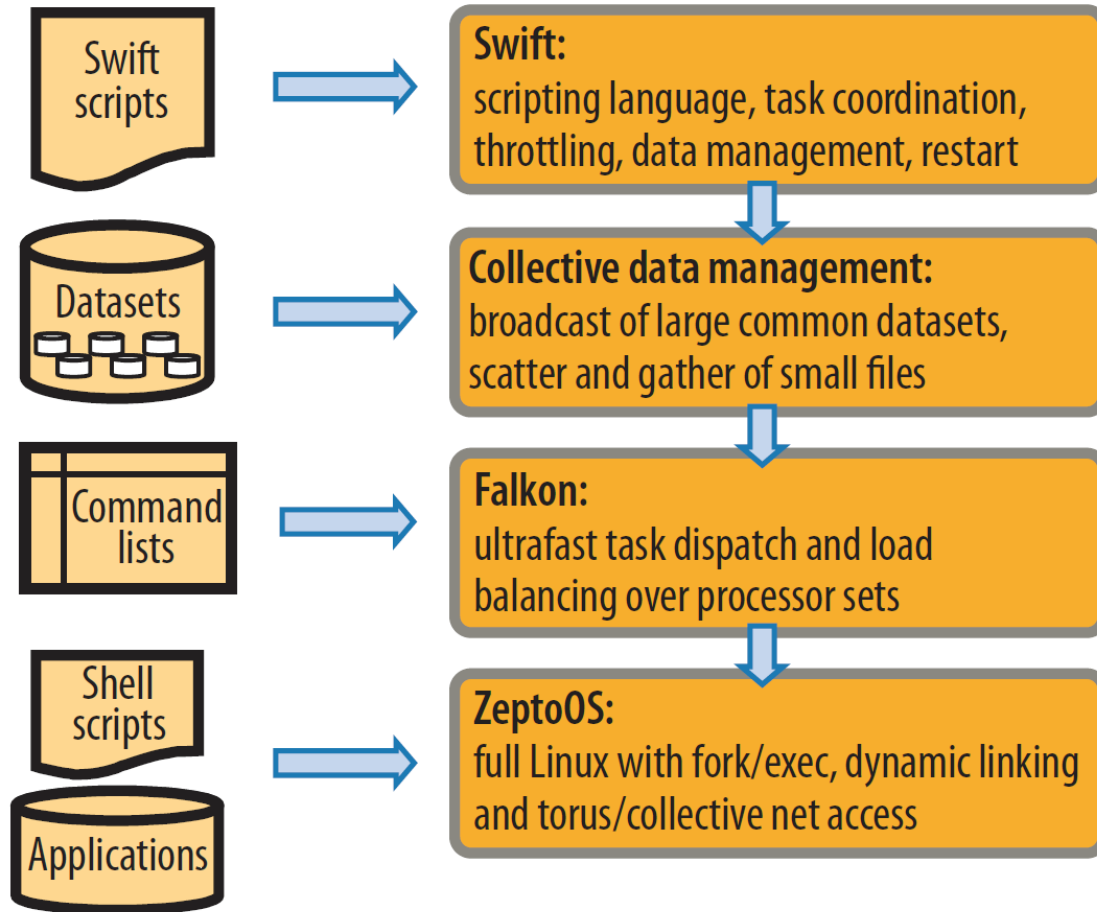


Figure 2. Architecture for petascale scripting.

More Details about Swift

- The other key area of interest is...
- **Massive implicit parallelism**
- We can declare a mapped array of files: (eg mydata.*.img)
- `file inp[] <simple_mapper; prefix="mydata.", suffix=".img">`; and iterate over it:
- `foreach s,i in inp { out[i] = f(s); // same as out[i] = f(inp[i]); }` All iterations can happen in parallel (subject to runtime limits, but could be many thousands of CPUs)
- In real use, f might be an app procedure taking 30s, with 10^5 loop iterations.
-

More Details about Swift

- **Execution order is data dependency order**
- Everything can be executed in parallel, except where there are dataflow dependencies.
- Dataflow dependencies are expressed by single assignment variables:
- `int a; int b; int c; a = f(c); b = g(6); c = h(7);` Execution of `f` will be after `h`. Execution of `g` will be unordered wrt `f` and `h`.
- Extends into (non-app) procedures.
Only concurrency control in SwiftScript - no locks, etc.
Assignment can be "in memory" or giving a file its content.

Arrays

- **Arrays are not single-assignment**
- `int a[]; int b; a[0] = 128; a[1] = 129; b = sum(a); // pass in the whole array a is not single assignment. But the elements of a are.`
- Static analysis of code to see which statements might write to a. When all potential writers are finished, then the array is "closed" for writing. Cannot modify an element once it has been assigned.
- Arrays are "monotonic" - we know more over time, and once we know something, it is true forever. A weaker form of single assignment.

Array deadlocks

- But there are deadlocks (in practice, and maybe in theory?):

```
int a[];
foreach i in [1:10] {
    if (i < 9)
    { a[i] = 5;
    } else { // i==10
    int b;
    b = f(a);
    }
}
```

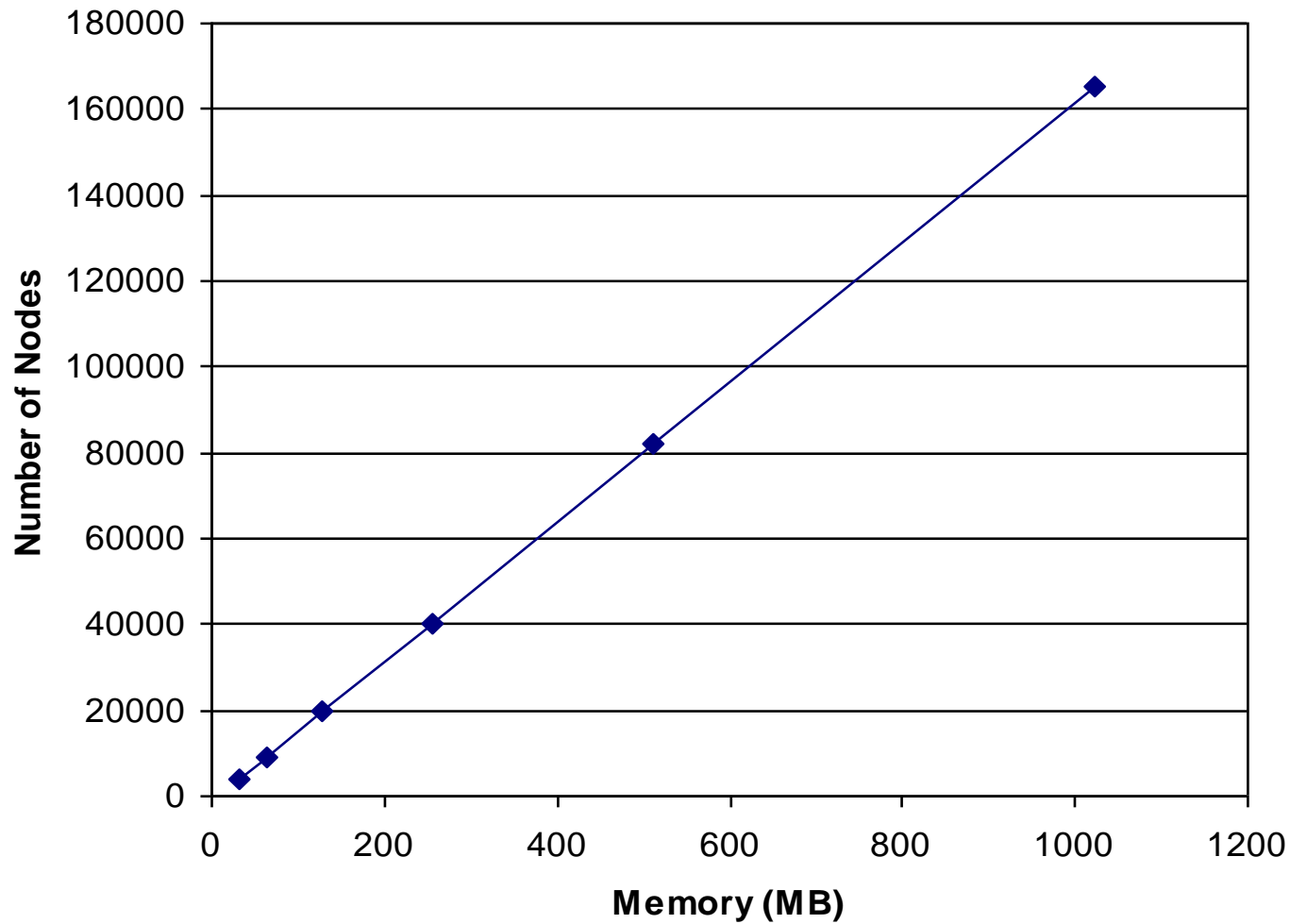
a will be closed when the whole foreach is finished... but the foreach will never finish because f(a) is waiting for a to close.

- Leads to programmer confusion when overly conservative
- More static+runtime analysis? Better structures/iterators? (map-like?)

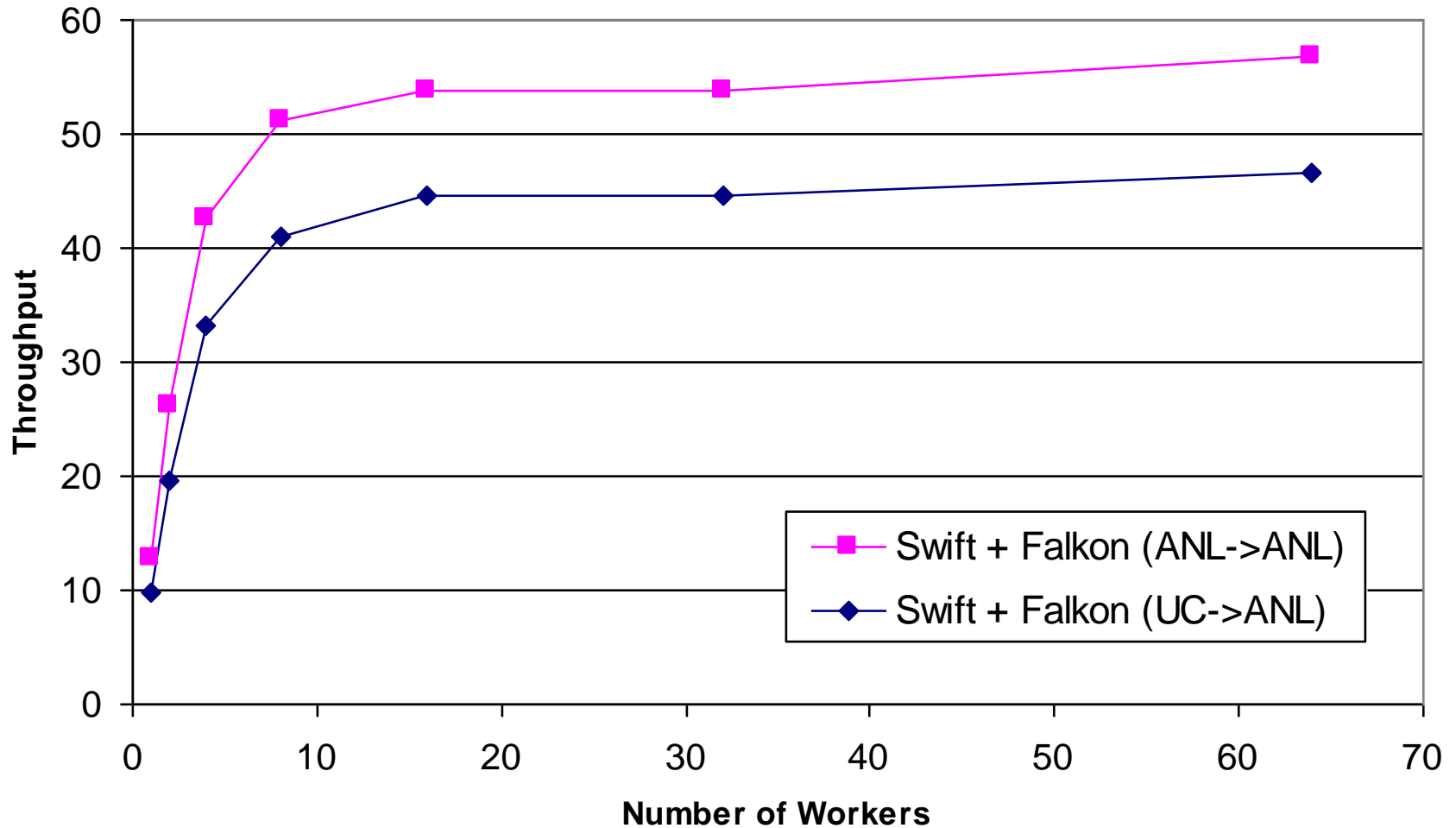
Performance

- Metrics are related to scientific computing focus.
- Mostly, what was done with app procedures by an application:
- How many CPU-hours in total? (eg 208763 CPU-hours)
- How many CPUs in use simultaneously? (eg 2000 CPUs)
- In terms of language execution, interested in raw SwiftScript speed where it impedes the above: can we sustain 100 app block launches per second?
- How short can you make your SwiftScript program? (so interesting to see how *few* lines of code are written in SwiftScript...)

Lightweight Threading - Scalability



Swift Throughput via Falkon



Less concrete idea (1): Provenance

- provenance = record of the history of an artifact to help convince you that it is genuine/valuable
- In Swift: record what output files were generated - which input files, which programs, where programs were run
- Which datafiles used this site? (because we must discard any results from it)
- Regenerate interesting results because we've damaged our copy
- Functional/dataflow style helps there but many other issues.
- Prototype implementation

Less concrete idea (2): Streaming Datasets

- Processing datasets that grow over time - eg a database of fMRI images that is added to as new patients are seen.
- Represent the database as a mapped array that is never closed.
We can iterate with foreach over that array, and leave the SwiftScript program running "forever"
- Maybe no need to change the language definition much / at all
- No implementation, only some mailinglist chatter

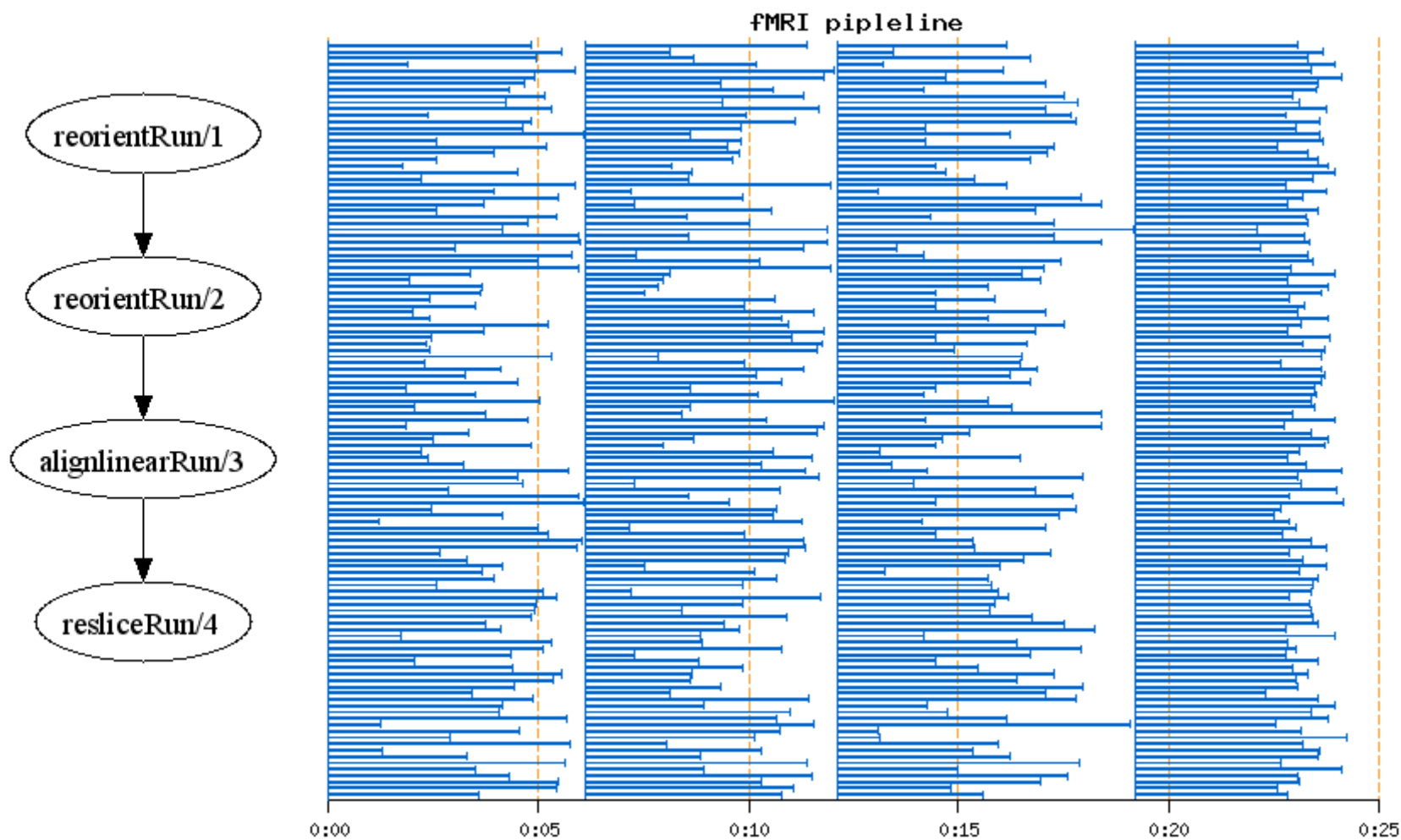
As an embedded language or library vs own language

- Q: why did you implement this as a new language rather than embedding?
- A: An accident of history - we started off making a glorified DAG description language, not a "real programming language"
- But we can still wonder whether it would be a better or worse idea...
- How would we implement:
- Out-of-core data and applications
- Massive (10^5) multithreading and everything-is-a-future style
- for example: in Haskell or Java

Embedding would give more libraries

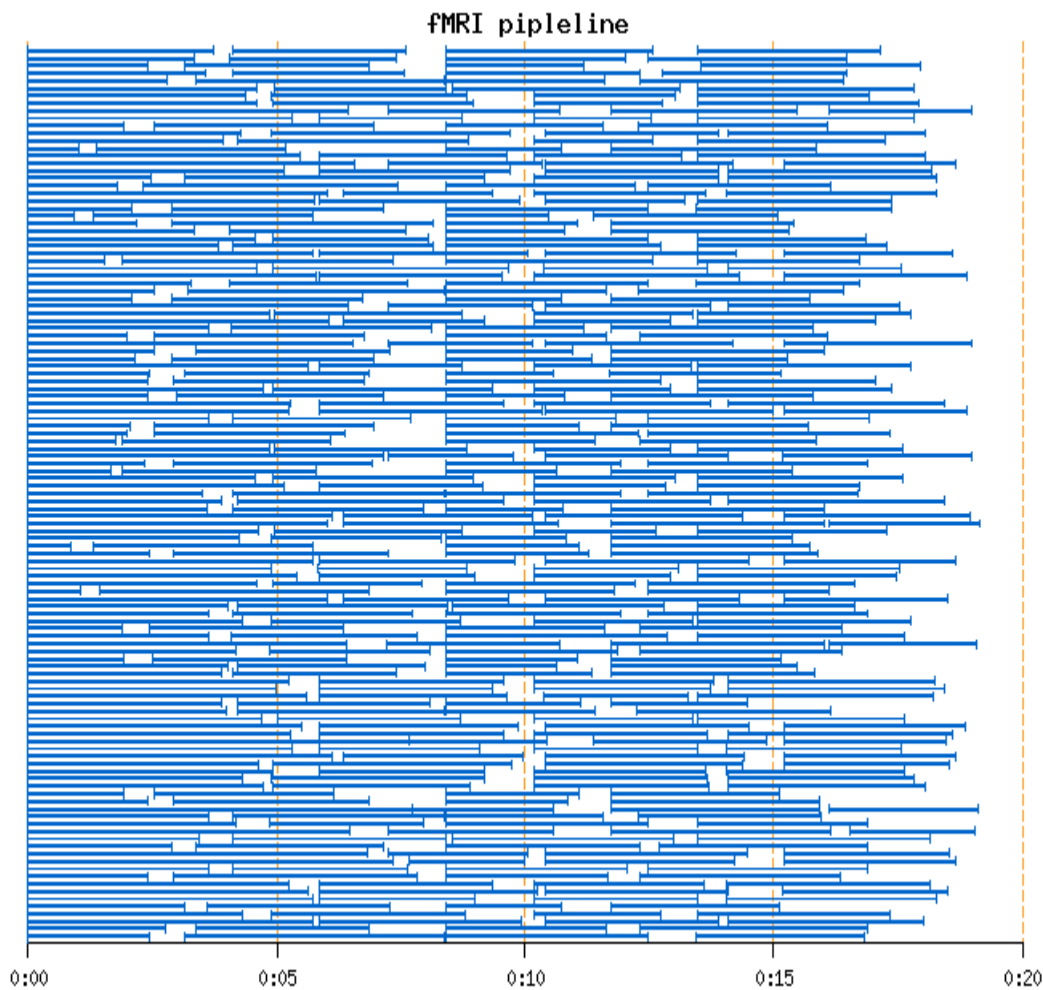
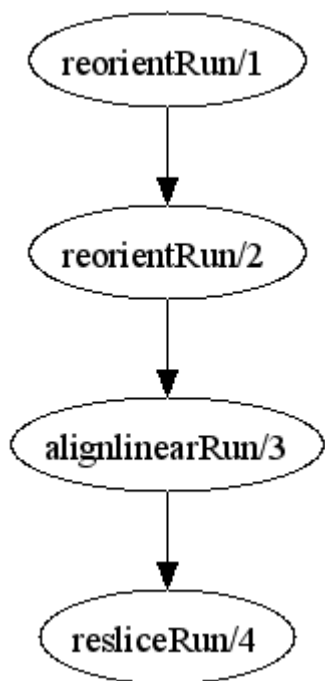
- In the beginning we were expecting people to make things that looked almost like DAGs of programs but "a bit more interesting."
- Now people want to do sin and cos and in-memory matrix multiplication
- Its a hassle to add wire in existing libraries in other languages for every new feature.
- The implementation is not really suited to in-core data processing - even a single integer has a very large footprint (because originally our 'values' were mostly many-megabyte files, where overhead mattered less)
- Areas that I've seen: parsing/printing data files; matrix operations; sin/cos
- Would be great to easily import some other languages library collection

Optimizations: fMRI Workflow Execution without Pipelining



(Dispatch is performed here via GRAM+PBS)

Optimizations: Karajan Futures Enable Pipelining

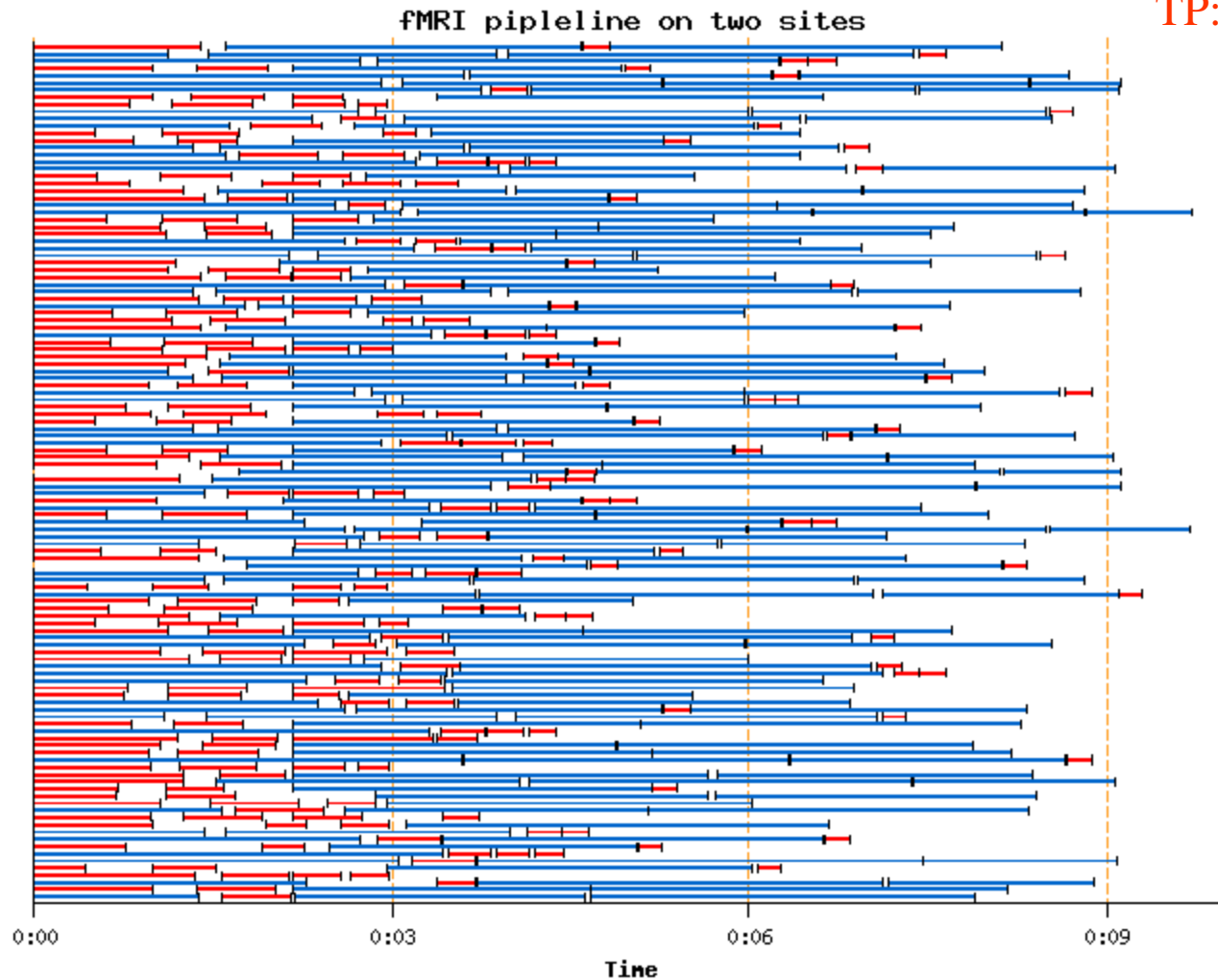
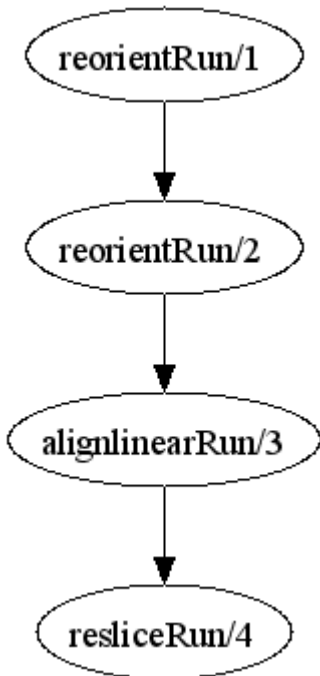


(Dispatch is performed here via GRAM+PBS)

Optimizations: Load Balancing

UC: 218

TP: 262



Applications

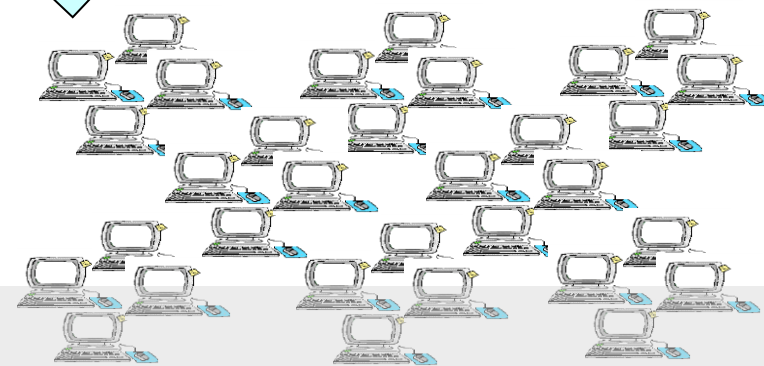
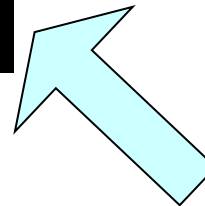
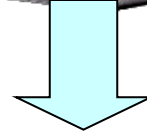
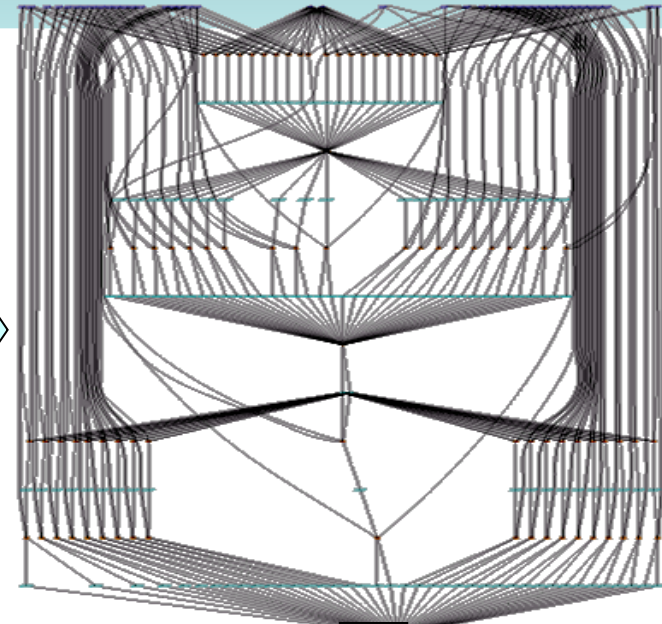
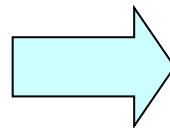
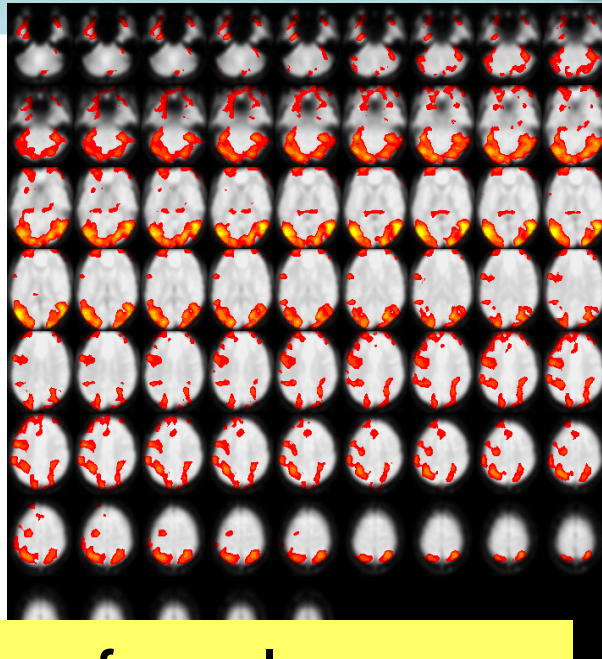
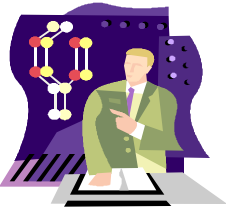
Table 1. Example parallel scripting applications.

Field	Description	Characteristics	Status
Astronomy	Creation of montages from many digital images	Many 1-core tasks, much communication, complex dependencies	Experimental
Astronomy	Stacking of cutouts from digital sky surveys	Many 1-core tasks, much communication	Experimental
Biochemistry*	Analysis of mass-spectrometer data for post-translational protein modifications	10,000-100 million jobs for proteomic searches using custom serial codes	In development
Biochemistry*	Protein structure prediction using iterative fixing algorithm; exploring other biomolecular interactions	Hundreds to thousands of 1- to 1,000-core simulations and data analysis	Operational
Biochemistry*	Identification of drug targets via computational docking/screening	Up to 1 million 1-core docking operations	Operational
Bioinformatics*	Metagenome modeling	Thousands of 1-core integer programming problems	In development
Business economics	Mining of large text corpora to study media bias	Analysis and comparison of over 70 million text files of news articles	In development
Climate science	Ensemble climate model runs and analysis of output data	Tens to hundreds of 100- to 1,000-core simulations	Experimental
Economics*	Generation of response surfaces for various economic models	1,000 to 1 million 1-core runs (10,000 typical), then data analysis	Operational
Neuroscience*	Analysis of functional MRI datasets	Comparison of images; connectivity analysis with structural equation modeling, 100,000+ tasks	Operational
Radiology	Training of computer-aided diagnosis algorithms	Comparison of images; many tasks, much communication	In development
Radiology	Image processing and brain mapping for neuro-surgical planning research	Execution of MPI application in parallel	In development

Note: Asterisks indicate applications being run on Argonne National Laboratory's Blue Gene/P (Intrepid) and/or the TeraGrid Sun Constellation at the University of Texas at Austin (Ranger).

Applications

Medical Imaging: fMRI



- Wide range of analyses
 - Testing, interactive analysis, production runs
 - Data mining
 - Parameter studies

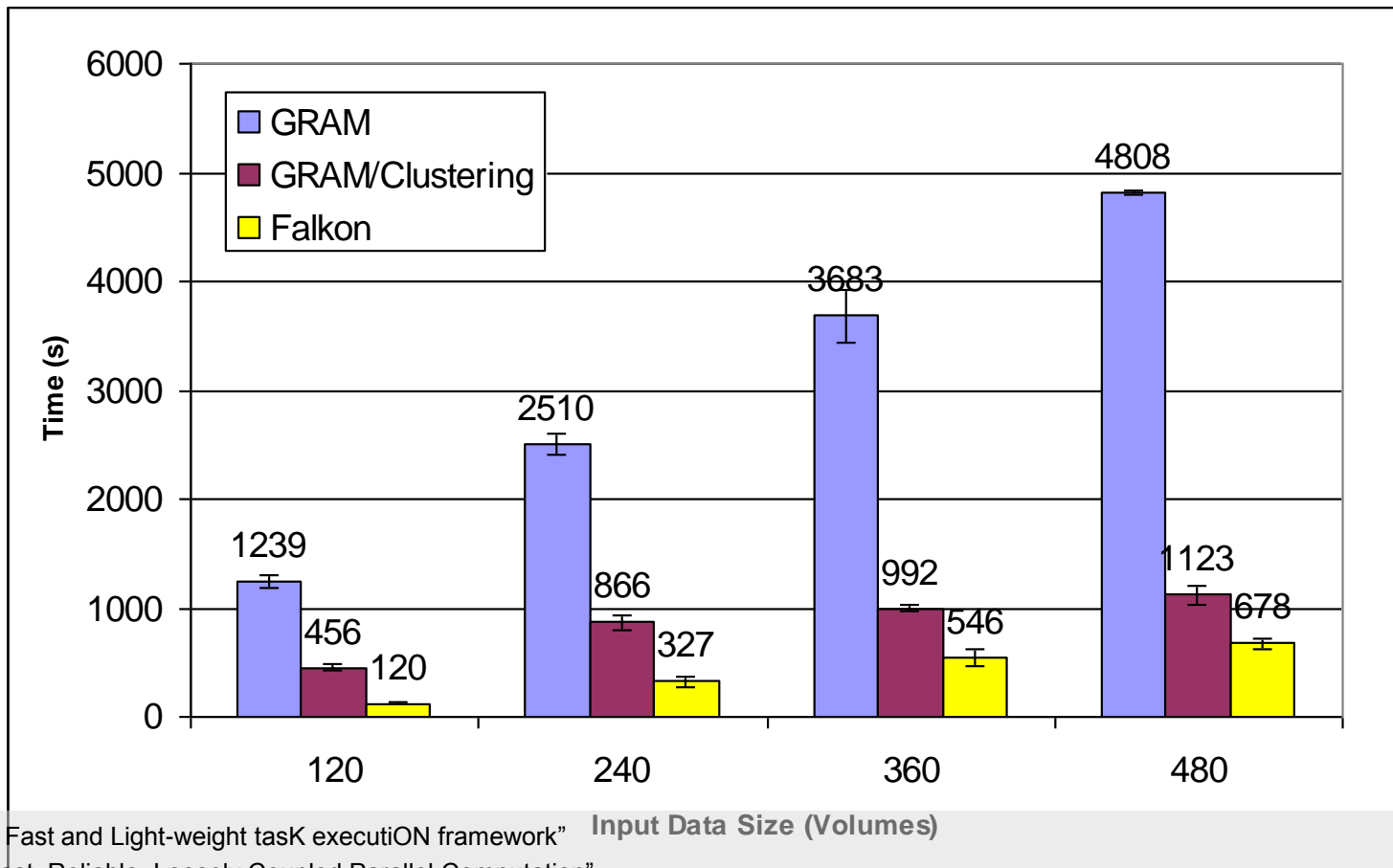
[SC07] "Falkon: a Fast and Light-weight task executiON framework"

[SWF07] "Swift: Fast, Reliable, Loosely Coupled Parallel Computation"

Applications

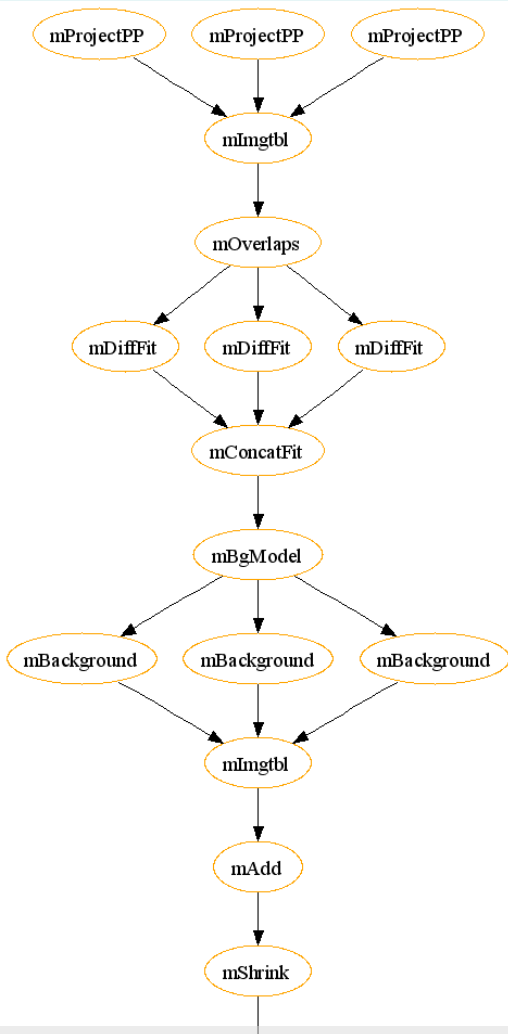
Medical Imaging: fMRI

- GRAM vs. Falcon: 85%~90% lower run time
- GRAM/Clustering vs. Falcon: 40%~74% lower run time



Applications

Astronomy: Montage

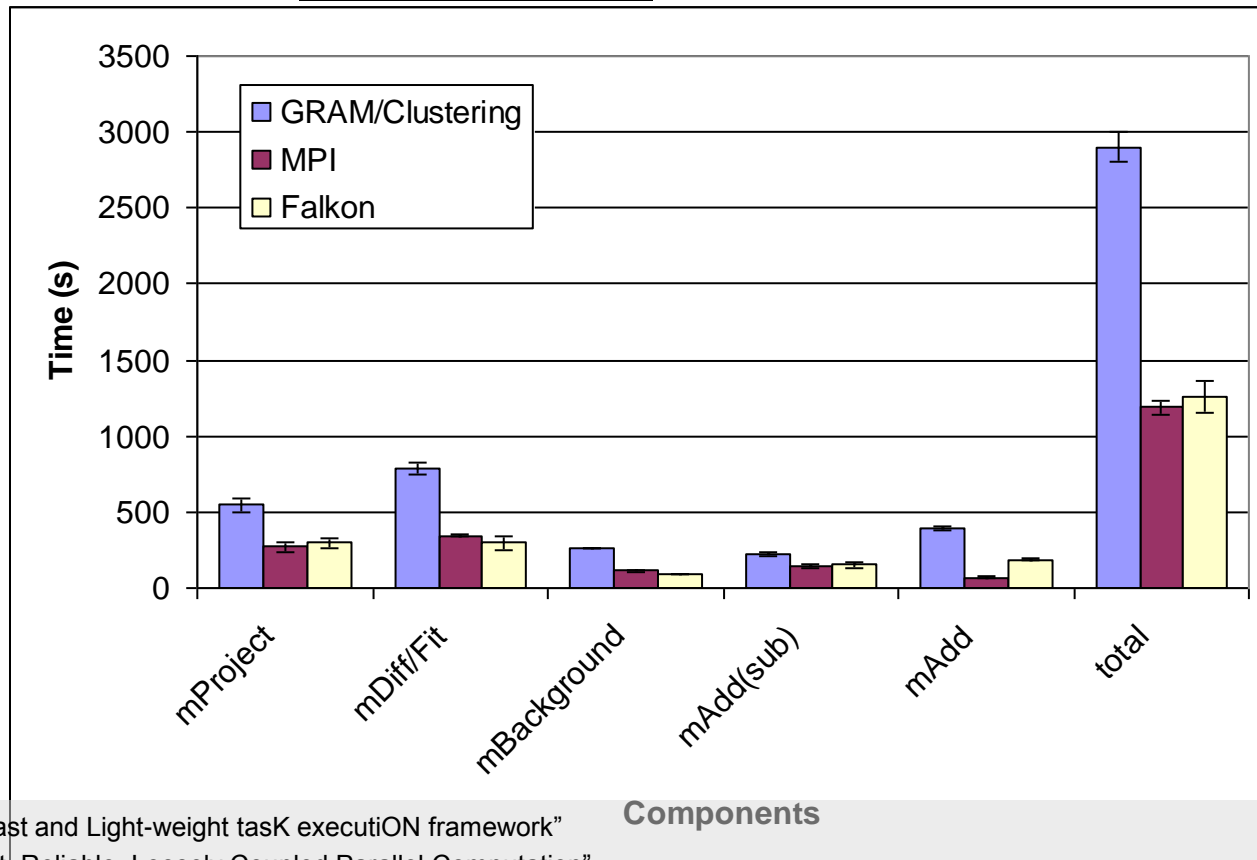


B. Berriman, J. Good (Caltech)
 J. Jacob, D. Katz (JPL)

Applications

Astronomy: Montage

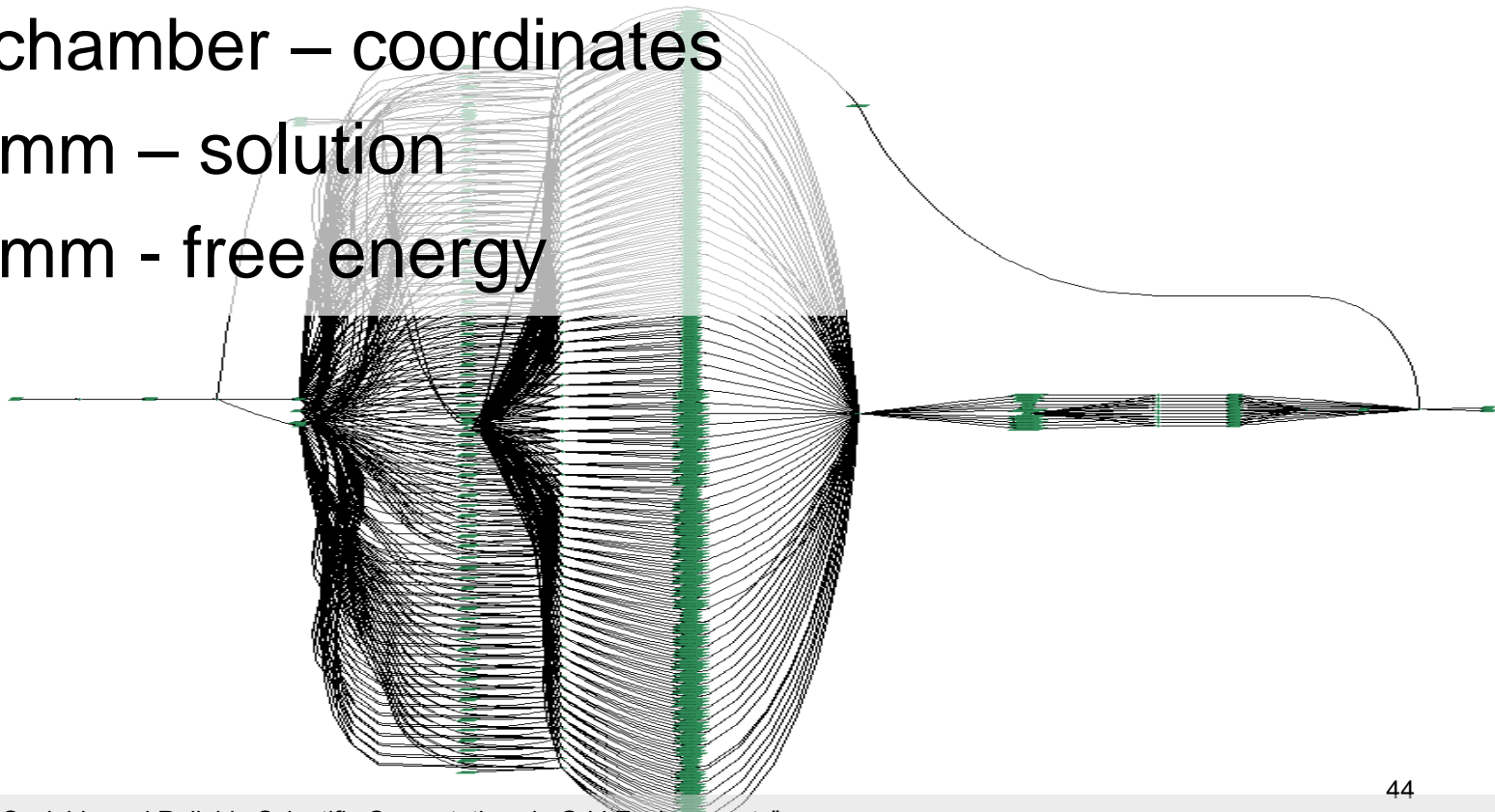
- GRAM/Clustering vs. Falcon: **57%** lower application run time
- MPI* vs. Falcon: **4%** higher application run time
- * MPI should be **lower bound**



Applications

Molecular Dynamics: MolDyn

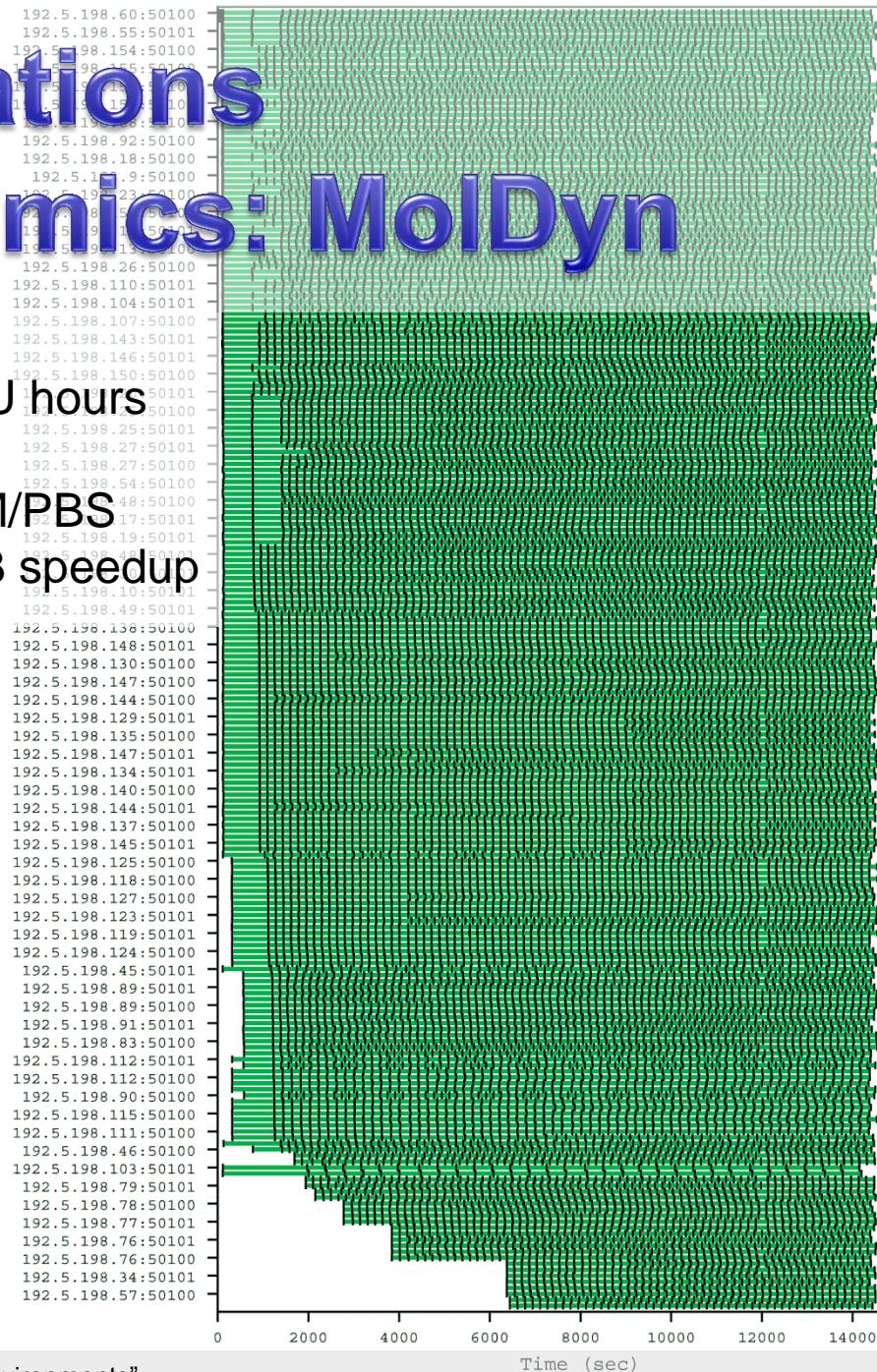
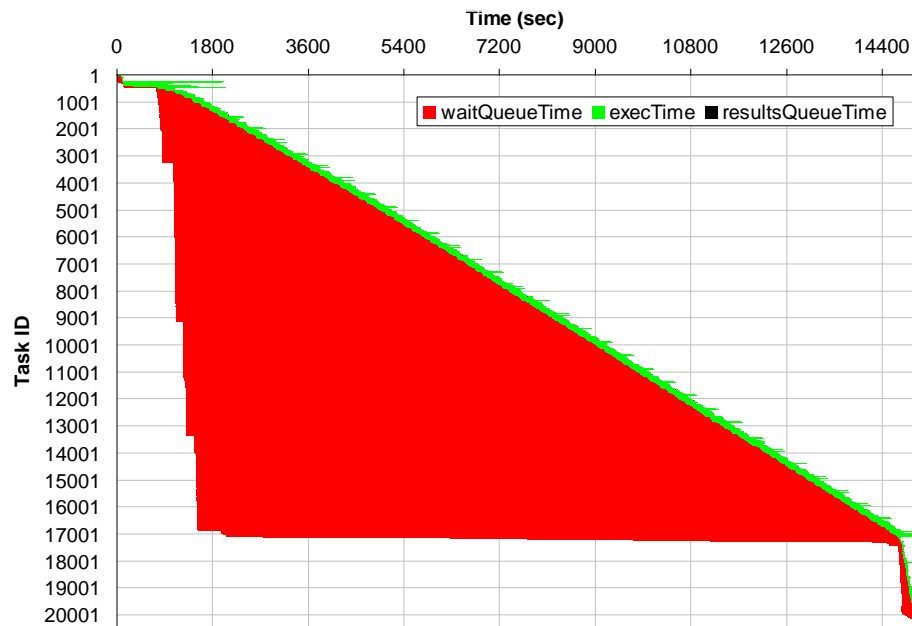
- Determination of free energies in aqueous solution
 - Antechamber – coordinates
 - Charmm – solution
 - Charmm - free energy



Applications

Molecular Dynamics: MolDyn

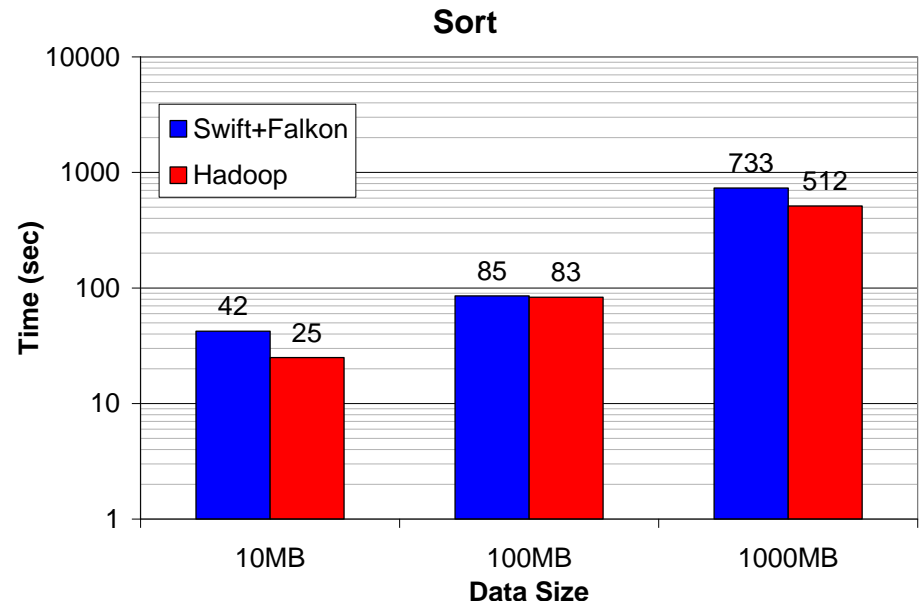
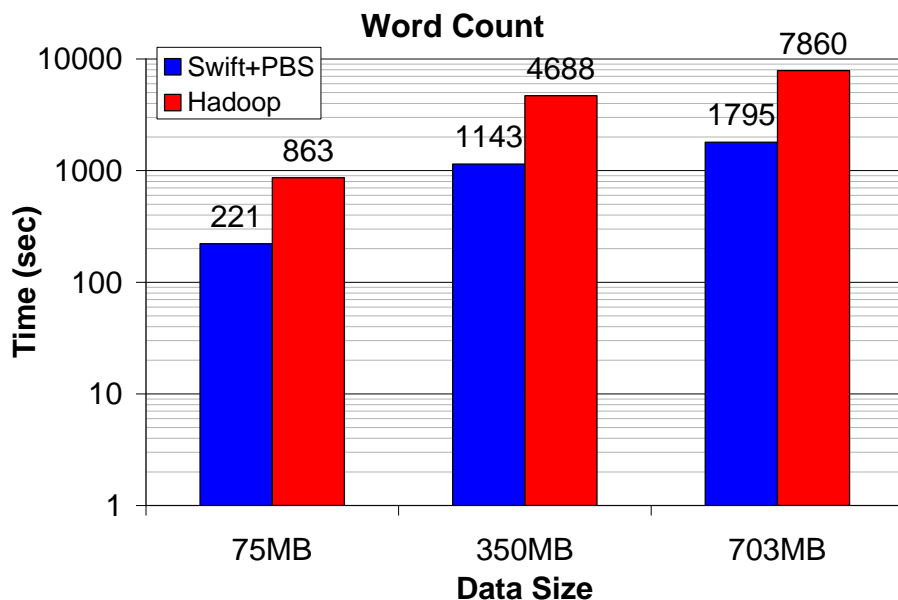
- 244 molecules → 20497 jobs
- 15091 seconds on 216 CPUs → 867.1 CPU hours
- Efficiency: **99.8%**
- Speedup: 206.9x → 8.2x faster than GRAM/PBS
- 50 molecules w/ GRAM (4201 jobs) → 25.3 speedup



Applications

Word Count and Sort

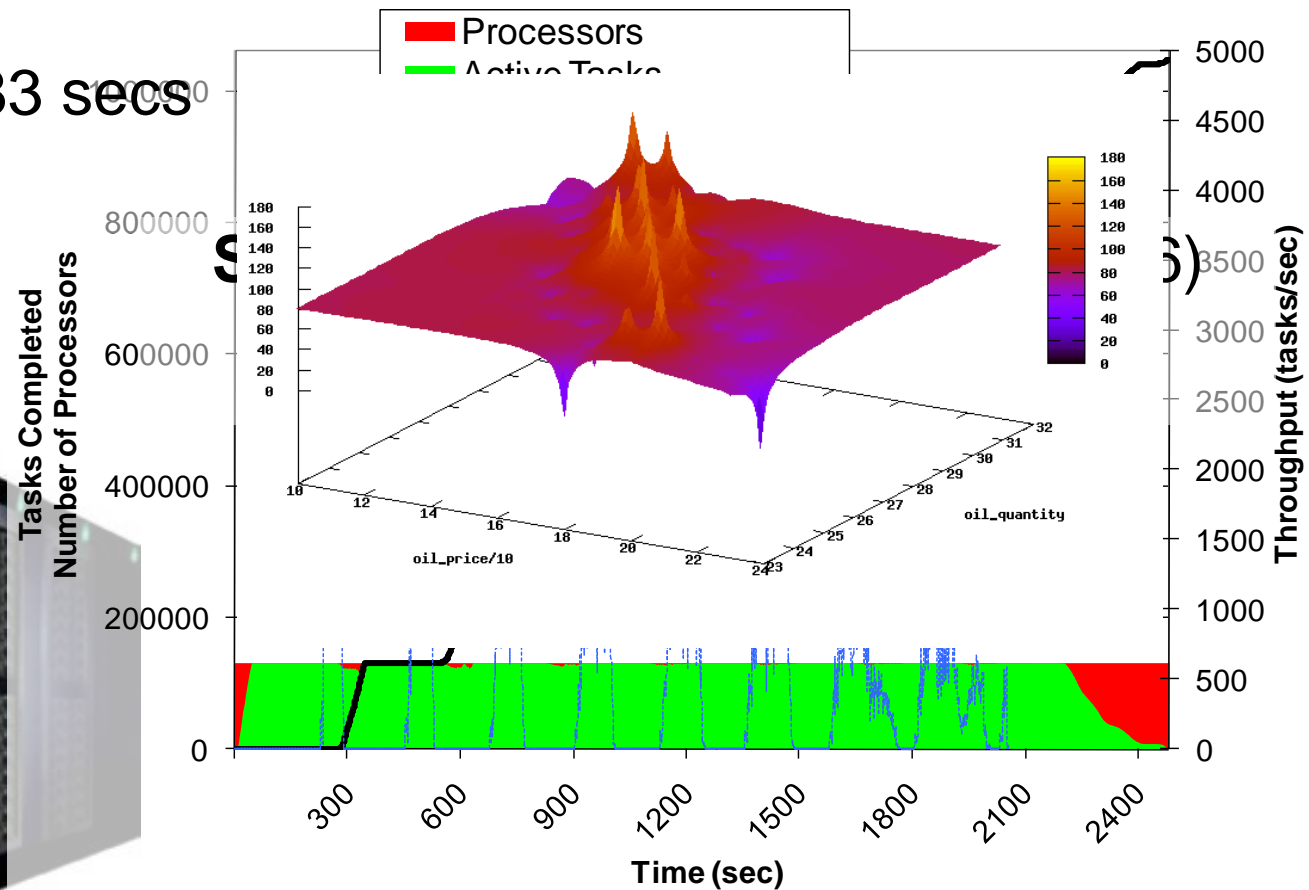
- Classic benchmarks for MapReduce
 - Word Count
 - Sort
- Swift and Falcon performs similar or better than Hadoop (on 32 processors)



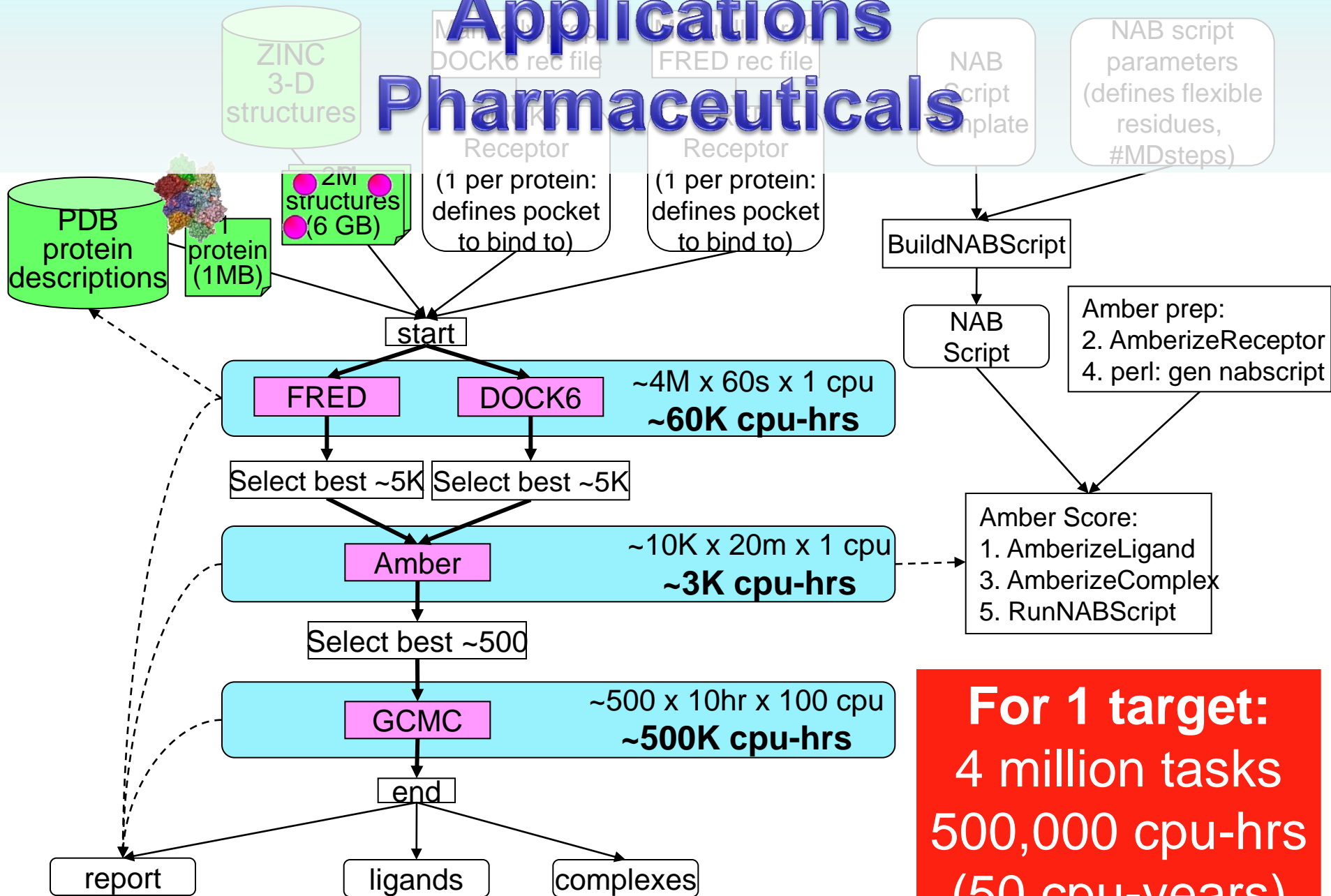
Applications

Economic Modeling: MARS

- CPU Cores: 130816
- Tasks: 1048576
- Elapsed time: 2483 secs
- CPU Years: 9.3



Applications Pharmaceuticals



Applications

Pharmaceuticals: DOCK

CPU cores: 118784

Tasks: 934803

Elapsed time: 2.01 hours

Compute time: 21.43 CPU years

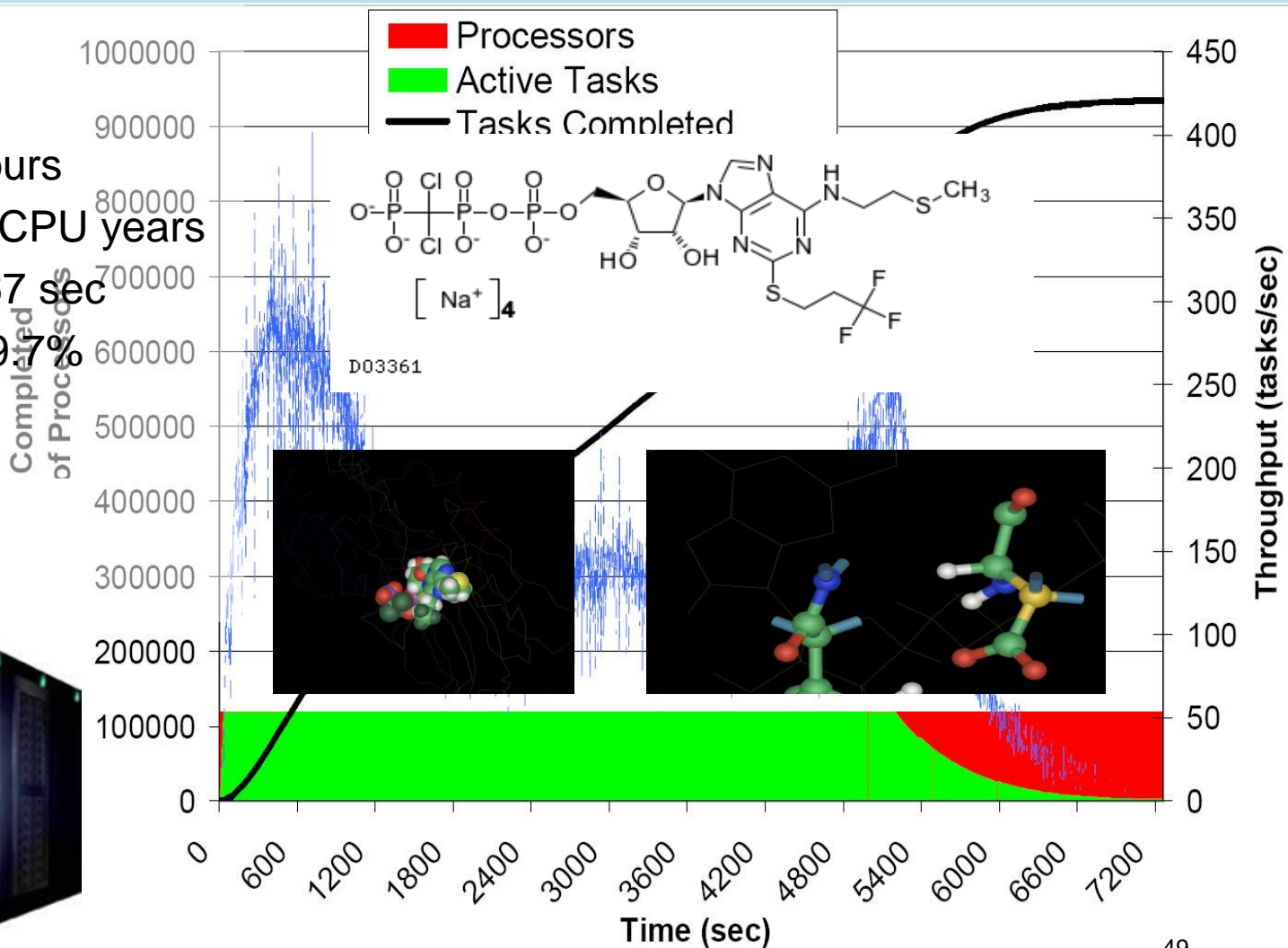
Average task time: 667 sec

Relative Efficiency: 99.7%

(from 16 to 32 racks)

Utilization:

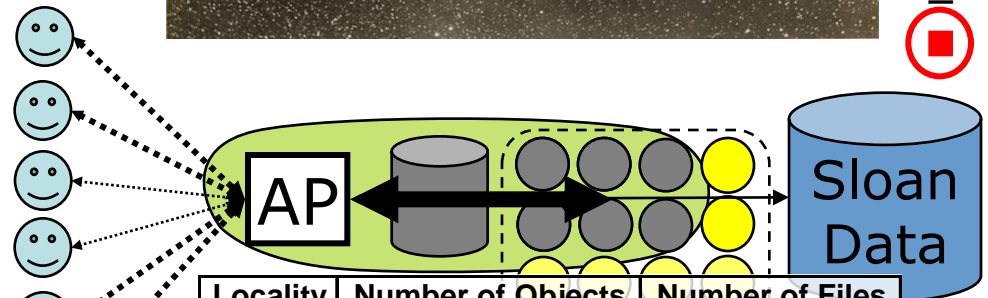
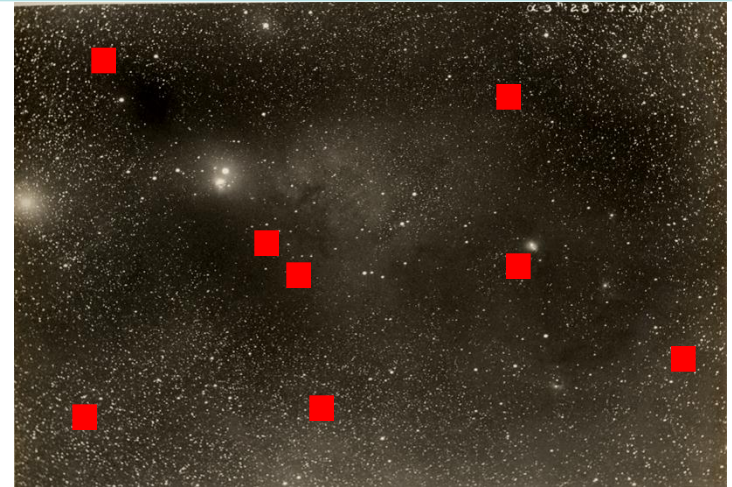
- Sustained: 99.6%
- Overall: 78.3%



Applications

Astronomy: AstroPortal

- Purpose
 - On-demand “stacks” of random locations within ~10TB dataset
- Challenge
 - Processing Costs:
 - O(100ms) per object
 - Data Intensive:
 - 40MB:1sec
 - Rapid access to 10-10K “random” files
 - Time-varying load



Locality	Number of Objects	Number of Files
1	111700	111700
1.38	154345	111699
2	97999	49000
3	88857	29620
4	76575	19145
5	60590	12120
10	46480	4650
20	40460	2025
30	23695	790

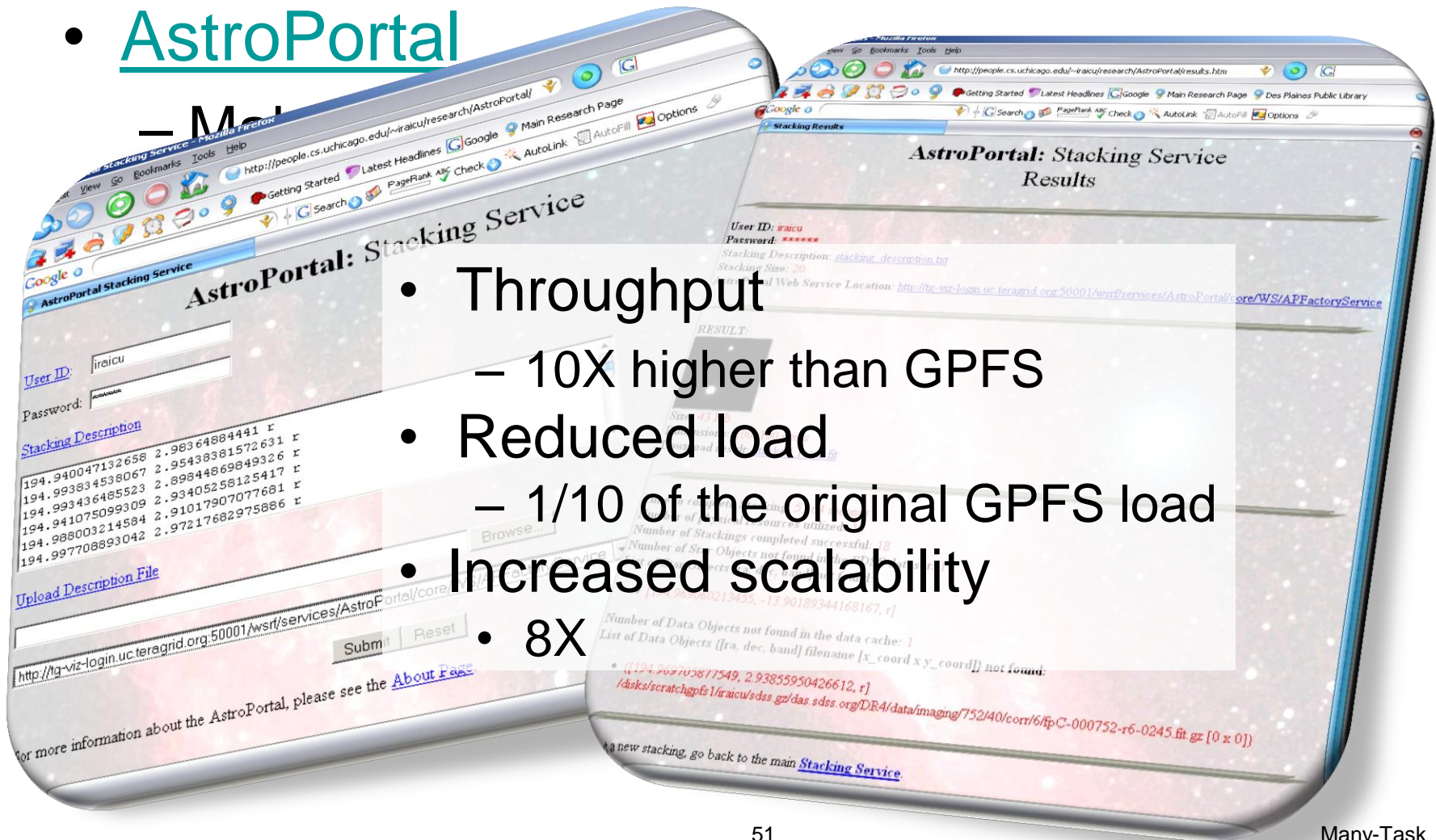
Applications

Astronomy: AstroPortal

- AstroPortal

– Multi-Tasking

- Throughput
 - 10X higher than GPFS
- Reduced load
 - 1/10 of the original GPFS load
- Increased scalability
 - 8X



Swift: Summary

- Clean separation of logical/physical concerns
- + Concise specification of parallel programs
 - SwiftScript, with iteration, etc.
- + Efficient execution (on distributed resources)
 - **Karajan+Falkon**: Grid interface, lightweight dispatch, pipelining, clustering, provisioning
- + Rigorous provenance tracking and query
 - Virtual data schema & automated recording
- **Improved usability and productivity**
 - Demonstrated in numerous applications

Questions

