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 2x5=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, Carriero 86], Strand [Foster, Taylor 90], PCN [Foster 92]
  - Durra [Barbacci, Wing 86], MANIFOLD [Papadopoulos 98]
  - Components programmed in specific language (C, FORTRAN) and linked with system

- “Workflow” languages and systems
  - Taverna [Oinn, Addis 04], Kepler [Ludäscher, Altintas 05], Triana [Churches, Gombas 05], Vistrail [Callahan, Freire 06], DAGMan, Star-P
  - XPDL [WfMC 02], BPEL [Andrews, Curbera 03], and BPML [BPML 02], YAWL [van de Aalst, Hofstede 05], Windows Workflow Foundation [Microsoft 05]
<table>
<thead>
<tr>
<th></th>
<th>SwiftScript</th>
<th>BPEL</th>
<th>XPDL</th>
<th>MW Wflow</th>
<th>DAGMan</th>
<th>Taverna</th>
<th>Triana</th>
<th>Kepler</th>
<th>Vistrail</th>
<th>Star-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scales to Grids</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Typing</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Iteration</td>
<td>++</td>
<td>-/+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Scripting</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Dataset Mapping</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Service Interop</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Subflow/comp.</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Provenance</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Open source</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

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

[Emmerich,Buchart06]
• ~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
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.
• 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
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
• 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
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
• 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

A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data [SIGMOD05]