Argonne NATIONAL LABORATORY

Dataflow for many-task computing: past, present and future

Michael Wilde, Argonne National Laboratory
and the University of Chicago

wilde@anl.gov

U.S. DEPARTMENT OF ENERGY

My history in many task computing

- Early work on OS and storage systems
  - Early SMP mainframe UNIX system at Bell Labs – *UNIX 370*
  - *Open Storage Manager* hierarchical storage system
- Language design and implementation for "programming in the large" – 5ESS and beyond
  - C-Talk (a C-flavored language with Smalltalk semantics)
- First experience in scientific HPC was on NSF-supported Grid Physics Network project "GriPhyN"
- Involved in design and application of 3 generations of many-task technology
  - Virtual Data System – started 2001
  - Swift Parallel Scripting System – started 2006
  - Swift "Turbine" implementation – started 2010

Collaboration of many colleagues

- Ian Foster, Mihael Hategan, Yong Zhao, Ben Clifford, Jens Voeckler, Sarah Kenny, Tiberius Stef-Prau
- Justin Wozniak, Tim Armstrong, Dan Katz, David Kelly, Ketan Maheshwari, Yadu Nand Babuji
- Ioan Raicu, Scott Krieder
- Glen Hocky, Aashish Adhikari, Tobin Sosnick, Carl Freed, Rob Jacob, Sheri Mickelson, Mariana Vertenstein, Jason Pitt, Lorenzo Pesce, Jim Phillips, John Stone, Oleseni Sode, Hemant Sharma, Jon Almer, Ray Osborn, and many other Swift users to whose trust and feedback we greatly value.

A view of Many Task Computing

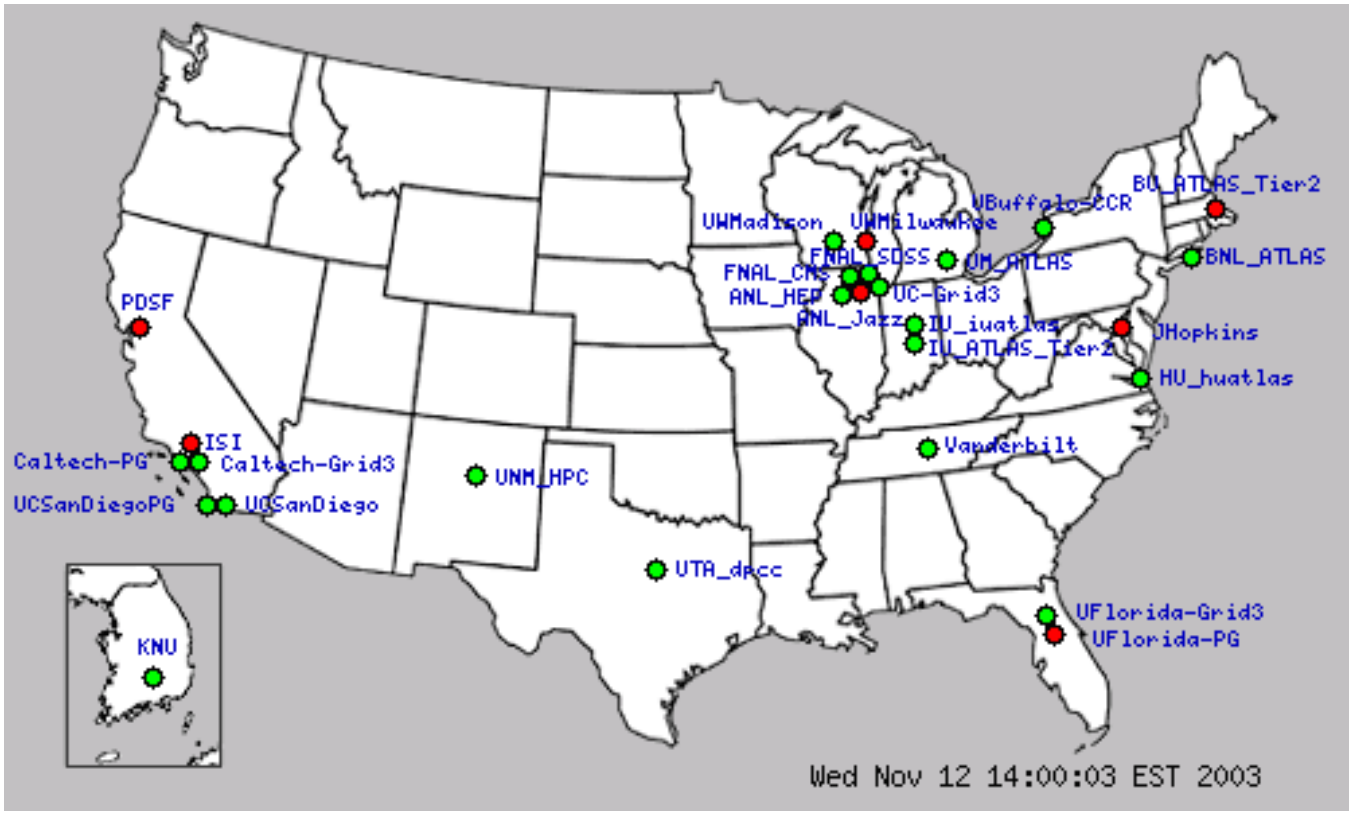
… and why its related to dataflow

- By “many tasks” we typically mean “very many”
- By tasks, we typically mean run-to-completion (before communicating results to other tasks)
- More precisely: a task is a function call !
- Running a lot of tasks fast and efficiently raises interesting research and engineering problems
  - Research: hierarchical scheduling, load balancing, service architectures
  - Engineering: speed, reliability, recoverability
- Data dependencies yield task dependencies, and challenges
  - How to express and manage dependencies?
  - What programming model?
  - How to describe and pass data?

Where we started: The Grid Physics Network

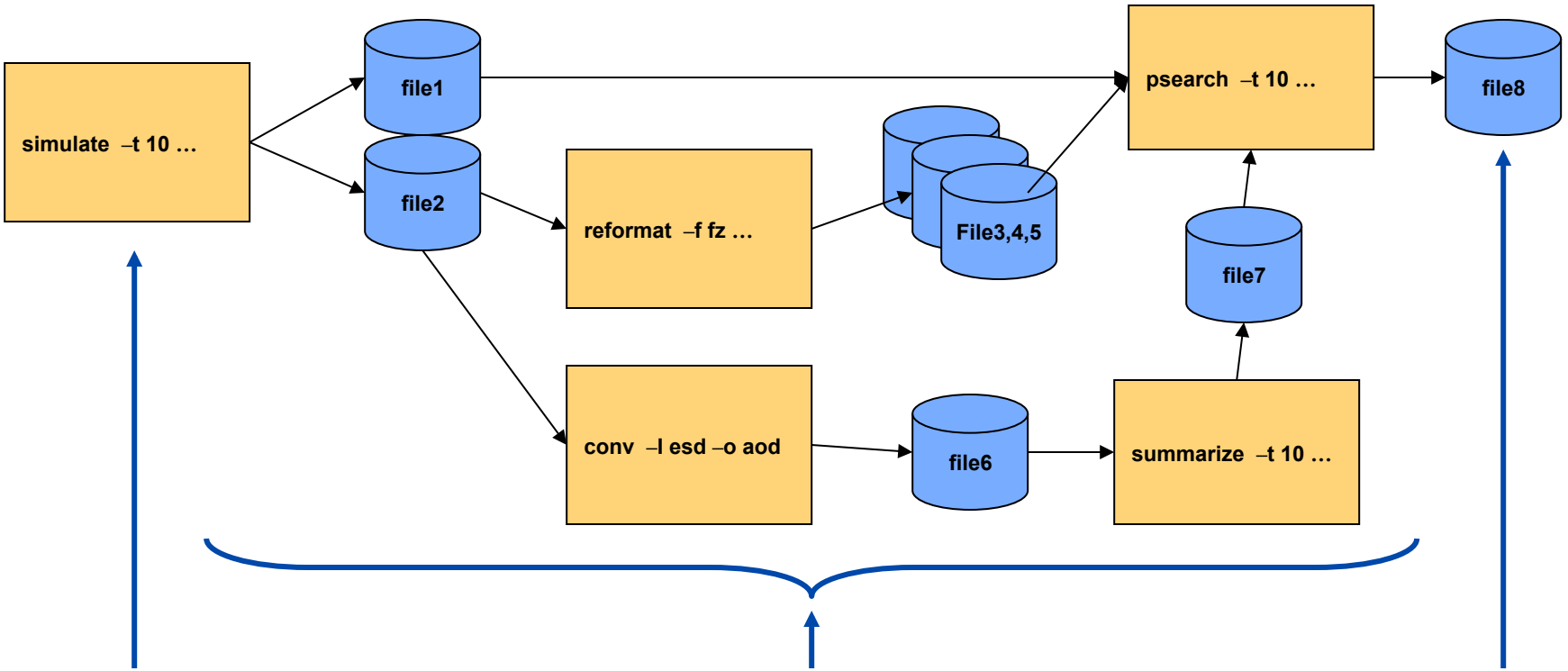
- Enhance scientific productivity through discovery and processing of datasets, using the grid as a scientific workstation
- Focused on ATLAS, CMS, LIGO, and SDSS
- Theme of GriPhyN was *Virtual Data*: how to express the processes by which data is derived, so that it can be recreated on demand
- *Approach: Create datasets from workflow "recipes" and record their provenance (ala "memoization")*

Grid3 – The Laboratory



Supported by the National Science Foundation and the Department of Energy.

Virtual Data Scenario



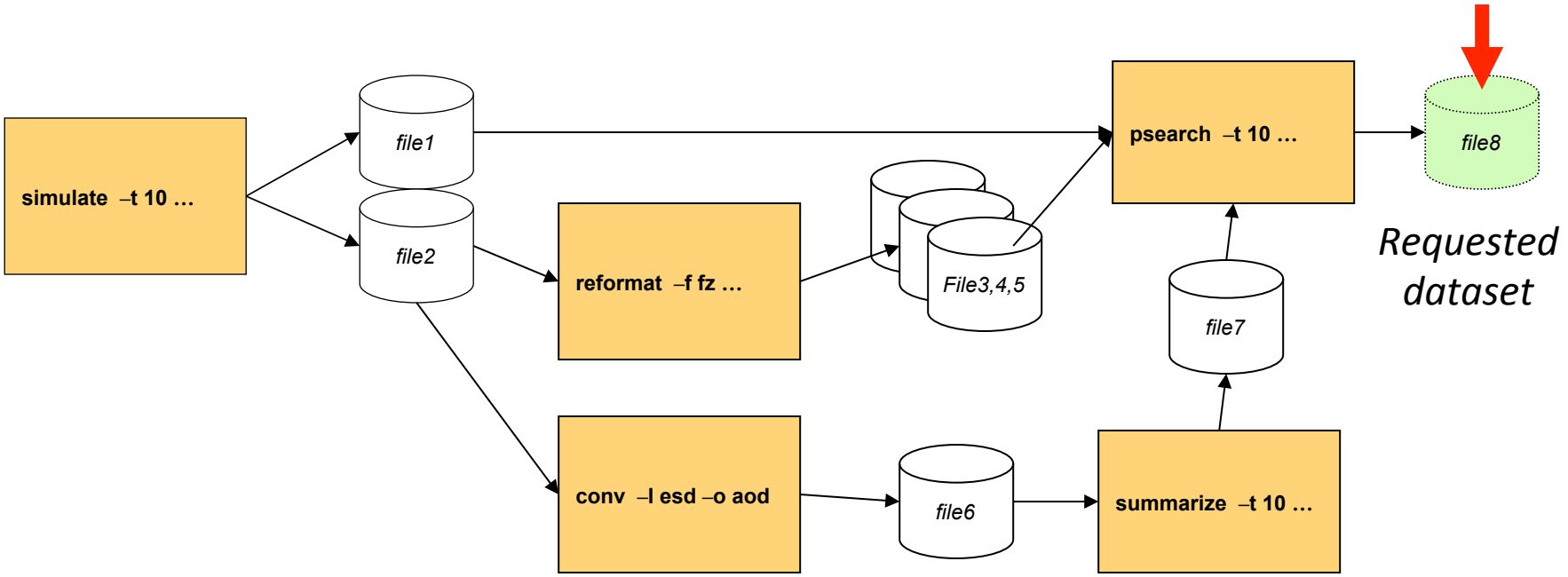
Update workflow following changes

Manage workflow;
Explain provenance, e.g. for file8:

psearch –t 10 –i file3 file4 file5 –o file8
summarize –t 10 –i file6 –o file7
reformat –f fz –i file2 –o file3 file4 file5
conv –I esd –o aod –i file 2 –o file6
simulate –t 10 –o file1 file2

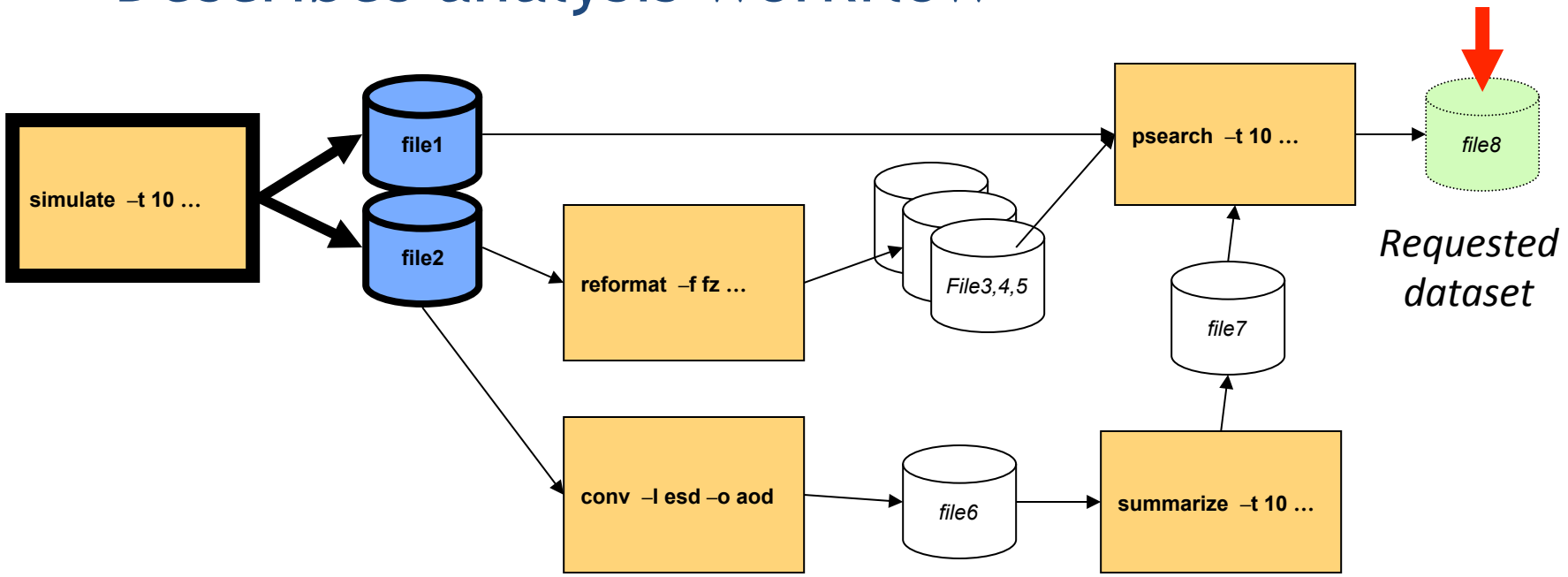
On-demand data generation

Virtual Data
Describes analysis workflow



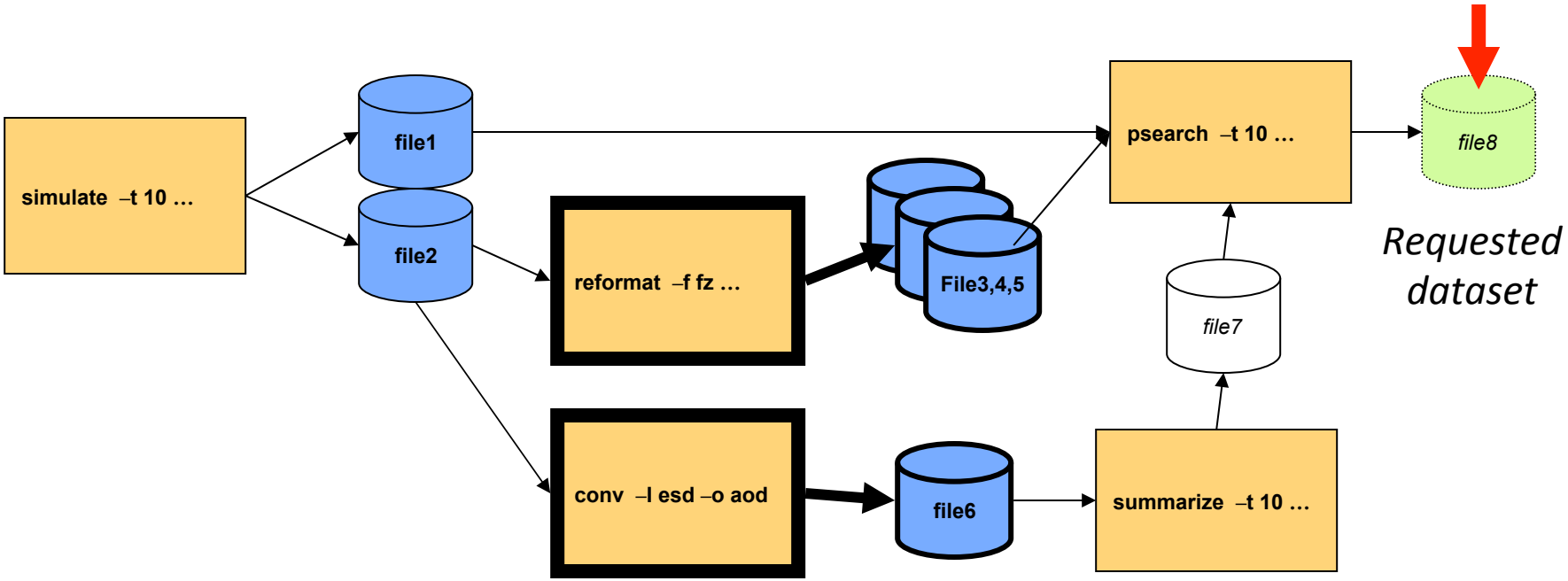
- The recorded virtual data "recipe" here is:
  - Files: 8 < (1,3,4,5,7), 7 < 6, (3,4,5,6) < 2
  - Programs: 8 < psearch, 7 < summarize, (3,4,5) < reformat, 6 < conv, (1,2) < simulate

Virtual Data
Describes analysis workflow



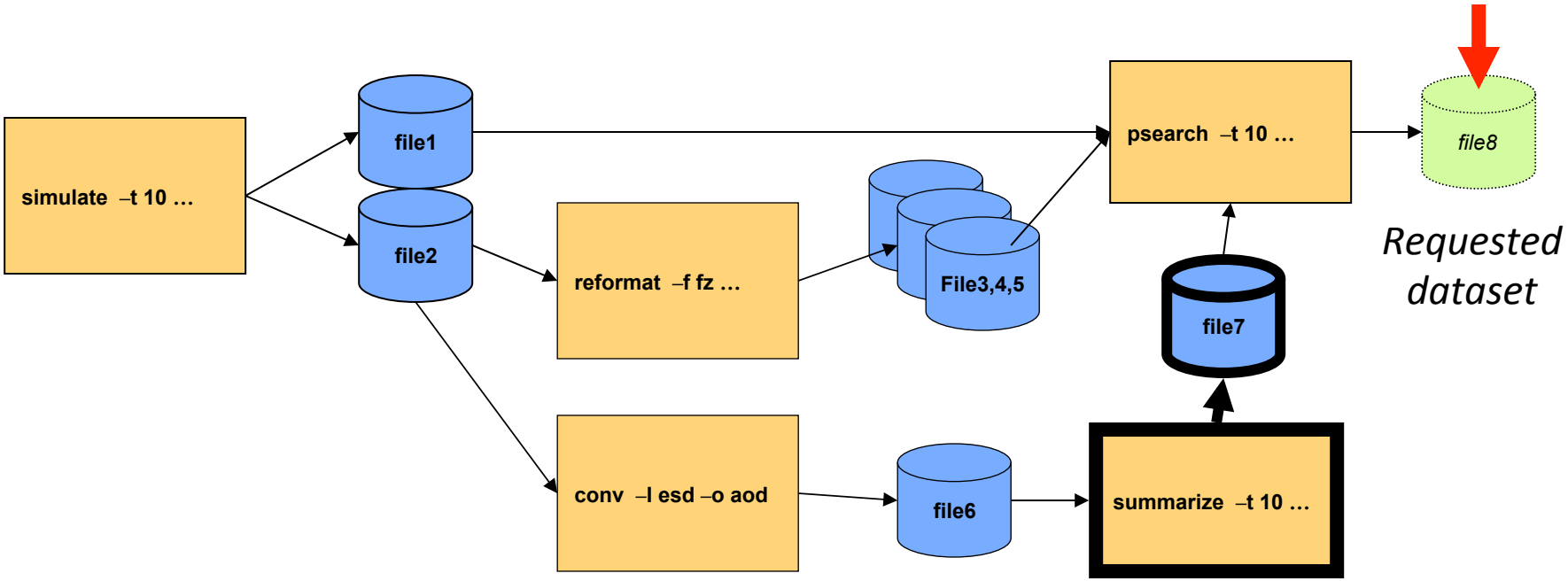
- To recreate file 8: Step 1
  - simulate > file1, file2

Virtual Data
Describes analysis workflow



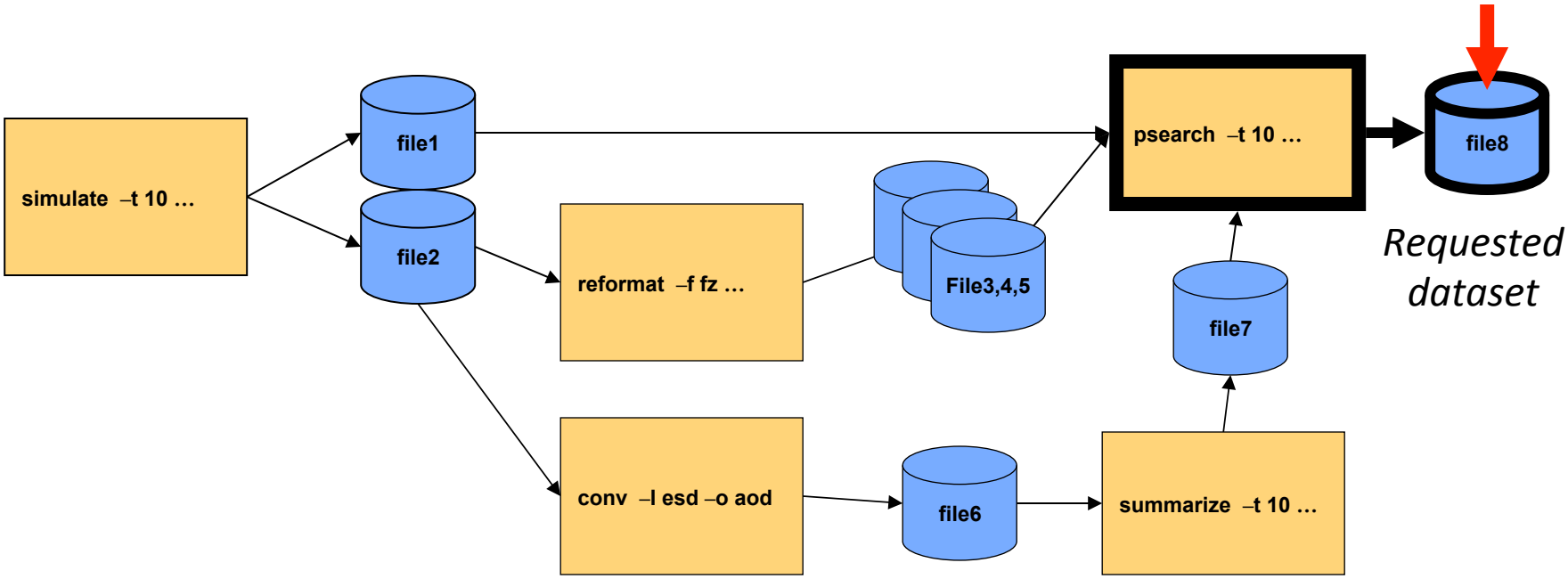
- To re-create file8: Step 2
  - files 3, 4, 5, 6 derived from file 2
  - reformat > file3, file4, file5
  - conv > file 6

Virtual Data
Describes analysis workflow



- To re-create file 8: step 3
  - File 7 depends on file 6
  - Summarize > file 7

Virtual Data
Describes analysis workflow

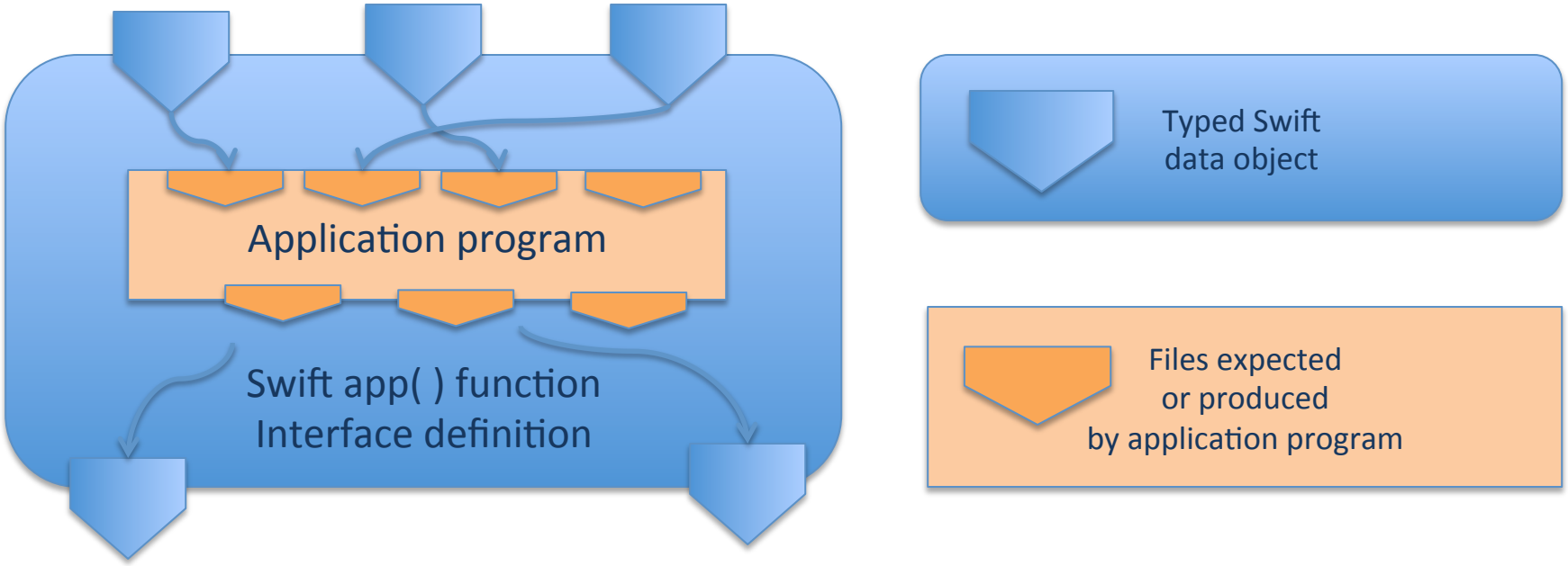


- To re-create file 8: final step
  - File 8 depends on files 1, 3, 4, 5, 7
  - psearch < file1, file3, file4, file5, file 7 > file 8

VDL: Virtual Data Language Describes Data Transformations

- Transformation
  - Abstract template of program invocation
  - Similar to "function definition"
- Derivation
  - "Function call" to a Transformation
  - Store past and future:
    - A record of how data products were generated
    - A recipe of how data products can be generated
- Invocation
  - Record of a Derivation execution
- These XML documents reside in a "virtual data catalog" – VDC - a relational database

VDL contribution: encapsulation enables distributed parallelism

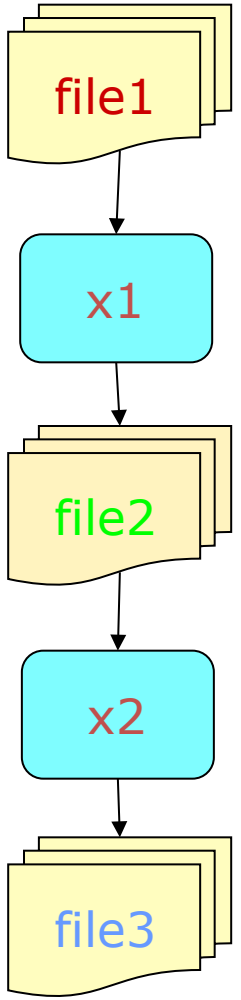


Encapsulation is the key to transparent distribution, parallelization, and automatic provenance capture

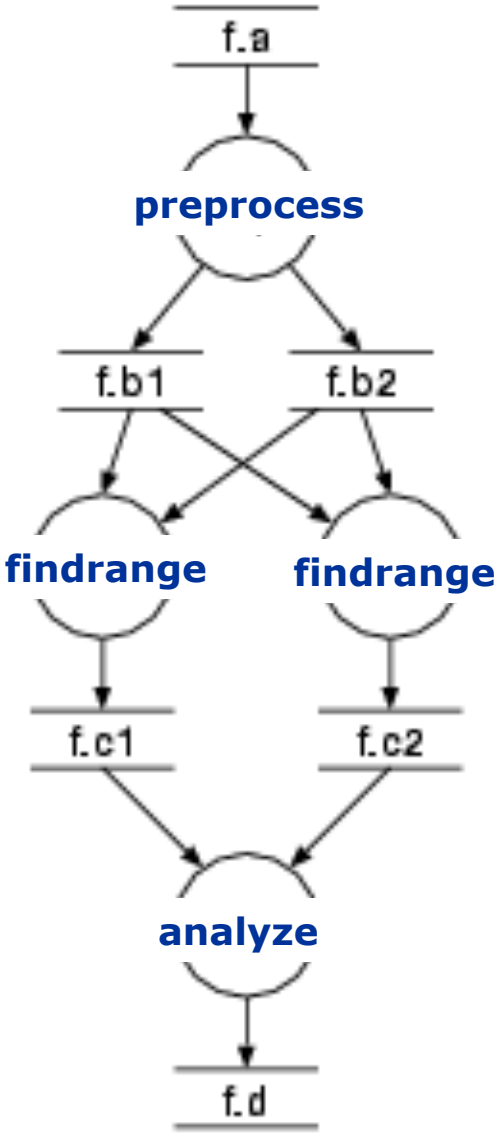
Critical in a world of scientific, engineering, technical and analytical applications

VDL Describes Workflow via Data Dependencies

```
TR tr1(in a1, out a2) {
   argument stdin = ${a1};
   argument stdout = ${a2}; }

TR tr2(in a1, out a2) {
   argument stdin = ${a1};
   argument stdout = ${a2}; }

DV x1->tr1(a1=@{in:file1}, a2=@{out:file2});
DV x2->tr2(a1=@{in:file2}, a2=@{out:file3});
```



Workflow example



- Graph structure
  - Fan-in
  - Fan-out
  - "left" and "right" can run in parallel
- Needs external input file
  - Located via replica catalog
- Data file dependencies
  - Form graph structure

Complete VDL workflow

- Generate appropriate derivations

```
DV top->preprocess( b=[ @{out:"f.b1"}, @{ out:"f.b2"} ], a=@{in:"f.a"} );
DV left->findrange( b=@{out:"f.c1"}, a2=@{in:"f.b2"}, a1=@{in:"f.b1"},
    name="left", p="0.5" );
DV right->findrange( b=@{out:"f.c2"}, a2=@{in:"f.b2"}, a1=@{in:"f.b1"},
    name="right" );
DV bottom->analyze( b=@{out:"f.d"}, a=[ @{in:"f.c1"}, @{in:"f.c2"} );
```

Compound Transformations Enable Functional Abstractions

- Compound TR encapsulates an entire sub-graph:

```
TR rangeAnalysis (in fa, p1, p2,
                  out fd, io fc1,
                  io fc2, io fb1, io fb2, )
{
 call preprocess( a=${fa}, b=[ ${out:fb1}, ${out:fb2} ] );
 call findrange( a1=${in:fb1}, a2=${in:fb2}, name="LEFT", p=${p1}, b=$
    {out:fc1} );
 call findrange( a1=${in:fb1}, a2=${in:fb2}, name="RIGHT", p=${p2}, b=$
    {out:fc2} );
 call analyze( a=[ ${in:fc1}, ${in:fc2} ], b=${fd} );
}
```

Derivation scripts

- Representation of virtual data provenance:

DV d1**->**diamond**(** fd=@{out:"f.00005"}, fc1=@{io:"f.00004"}, fc2=@{io:"f.00003"},
fb1=@{io:"f.00002"}, fb2=@{io:"f.00001"},
fa=@{io:"f.00000"}, p2="100", p1="0" **);**

DV d2**->**diamond**(** fd=@{out:"f.0000B"}, fc1=@{io:"f.0000A"}, fc2=@{io:"f.00009"},
fb1=@{io:"f.00008"}, fb2=@{io:"f.00007"},
fa=@{io:"f.00006"}, p2="141.42135623731", p1="0" **);**

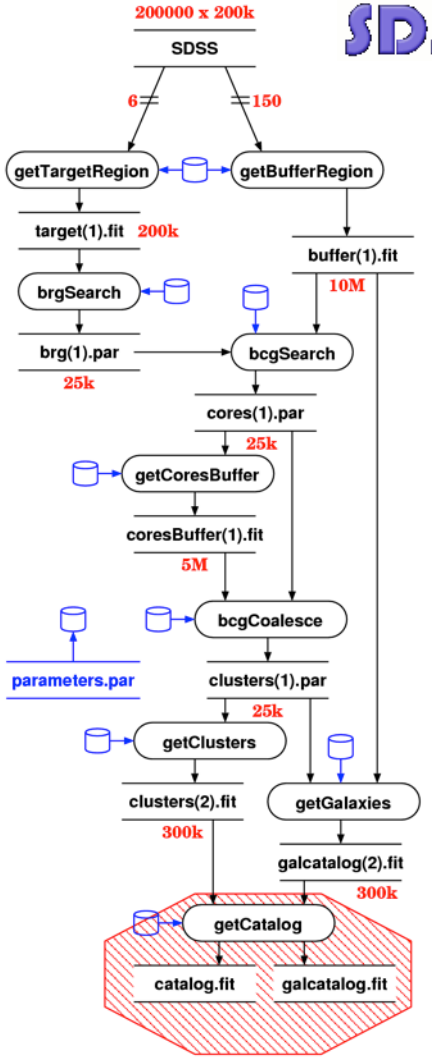
...

DV d70**->**diamond**(** fd=@{out:"f.001A3"}, fc1=@{io:"f.001A2"},
fc2=@{io:"f.001A1"},
fb1=@{io:"f.001A0"}, fb2=@{io:"f.0019F"},
fa=@{io:"f.0019E"}, p2="800", p1="18" **);**

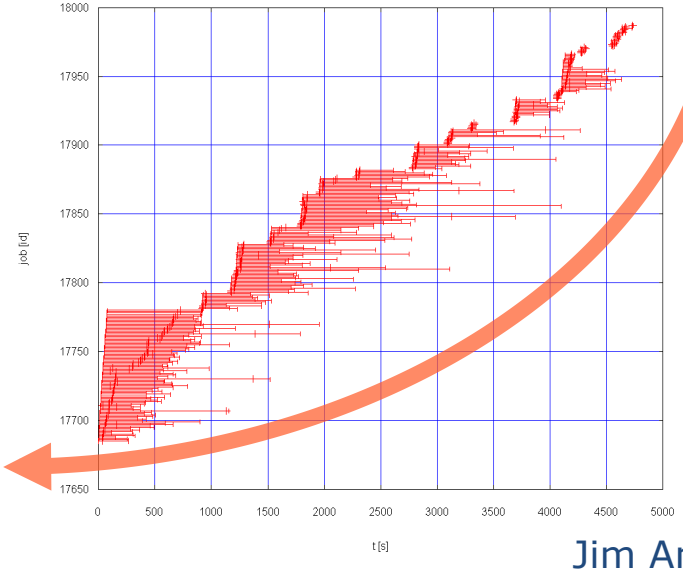
VDL Workflow: Finding Galaxy Clusters

1: Sloan Image Data

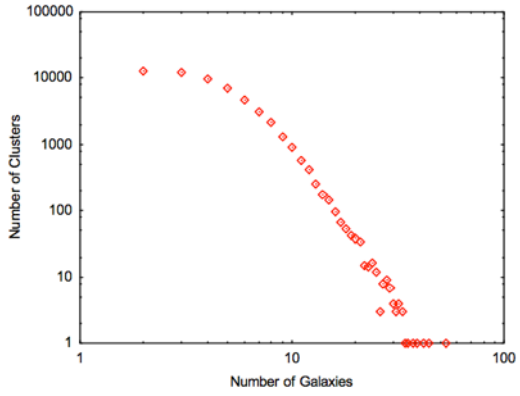
2: Workflow



3: Grid computing resources



4: Galaxy cluster size distribution



Jim Annis, Steve Kent, Vijay Sehkri, *Fermilab*, Michael Milligan, Yong Zhao, *University of Chicago*

VDL Workflow: Finding Galaxy Clusters

Drawback of virtual data language:

For almost any application of modest complexity, application-specific derivation graph generators were needed.

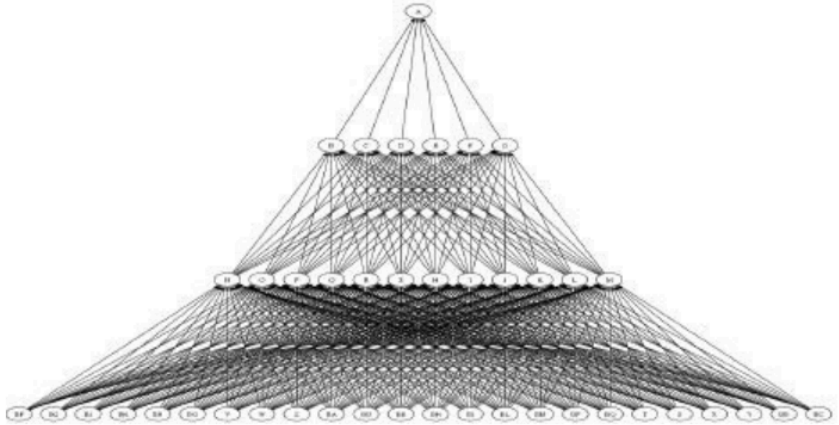


Figure 6: A basic DAG for cluster identification workflow.

The graph shown below is for one specific stripe of the survey (stripe 34). It uses 123 nodes to process 110x12 fields, and illustrates how larger workflows can be composed of many overlapping invocations of the workflow of the basic DAG pattern shown above.

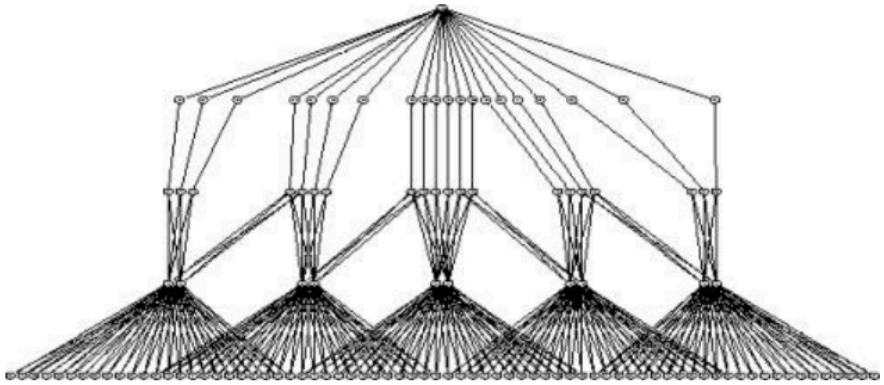


Figure 7: DAG for stripe 34 showing composition from a basic pattern.

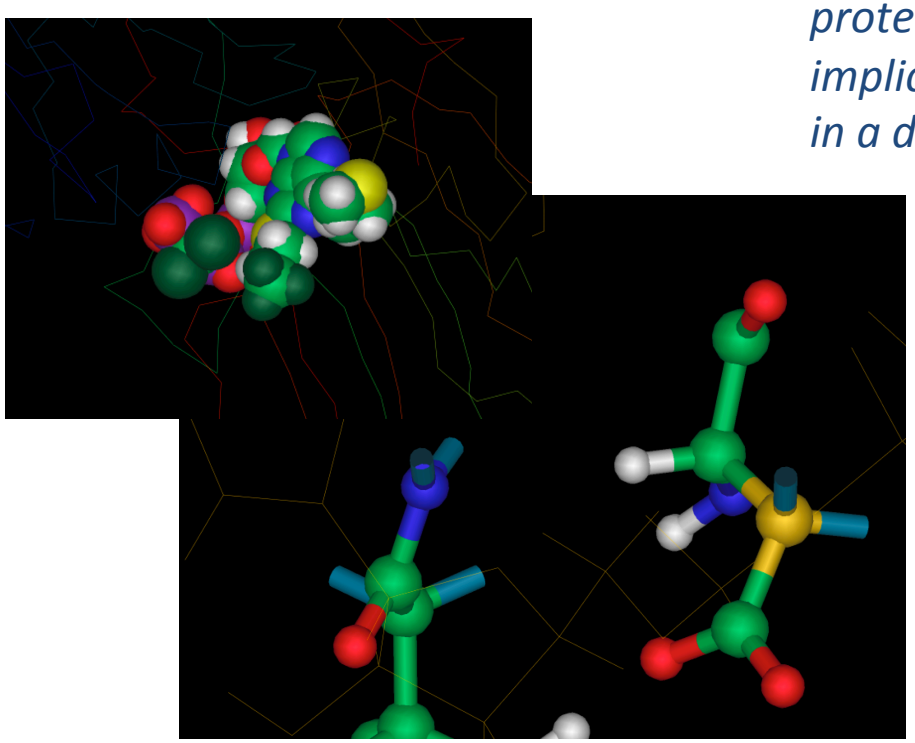
Jim Annis, Steve Kent, Vijay Sehkri,
Fermilab, Michael Milligan, Yong Zhao,
University of Chicago

DAX format: derivation DAG in XML

```
<job namespace="diamond" name="preprocess" version="2.0" id="ID000001">
 <argument>-a preprocess -T60 -i <file name="f.a" /> -o <file name="f.b1" /> <file name="f.b2" /></argument>
 <uses name="f.b2" link="output" register="false" transfer="false" />
 <uses name="f.b1" link="output" register="false" transfer="false" />
 <uses name="f.a" link="input" />
</job>
<job namespace="diamond" name="findrange" version="2.0" id="ID000002">
 <argument>-a findrange -T60 -i <file name="f.b1" /> -o <file name="f.c1" /></argument>
 <uses name="f.b1" link="input" register="false" transfer="false" />
 <uses name="f.c1" link="output" register="false" transfer="false" />
</job>
<job namespace="diamond" name="findrange" version="2.0" id="ID000003">
 <argument>-a findrange -T60 -i <file name="f.b2" /> -o <file name="f.c2" /></argument>
 <uses name="f.c2" link="output" register="false" transfer="false" />
 <uses name="f.b2" link="input" register="false" transfer="false" />
</job>
<job namespace="diamond" name="analyze" version="2.0" id="ID000004">
 <argument>-a analyze -T60 -i <file name="f.c1" /> <file name="f.c2" /> -o <file name="f.d" /></argument>
 <uses name="f.c2" link="input" register="false" transfer="false" />
 <uses name="f.d" link="output" register="false" transfer="true" />
 <uses name="f.c1" link="input" register="false" transfer="false" />
</job>
<!-- part 3: list of control-flow dependencies -->
<child ref="ID000002">
 <parent ref="ID000001" />
</child>
<child ref="ID000003">
 <parent ref="ID000001" />
</child>
<child ref="ID000004">
 <parent ref="ID000002" />
 <parent ref="ID000003" />
</child>
```

But what if you have many simple tasks?

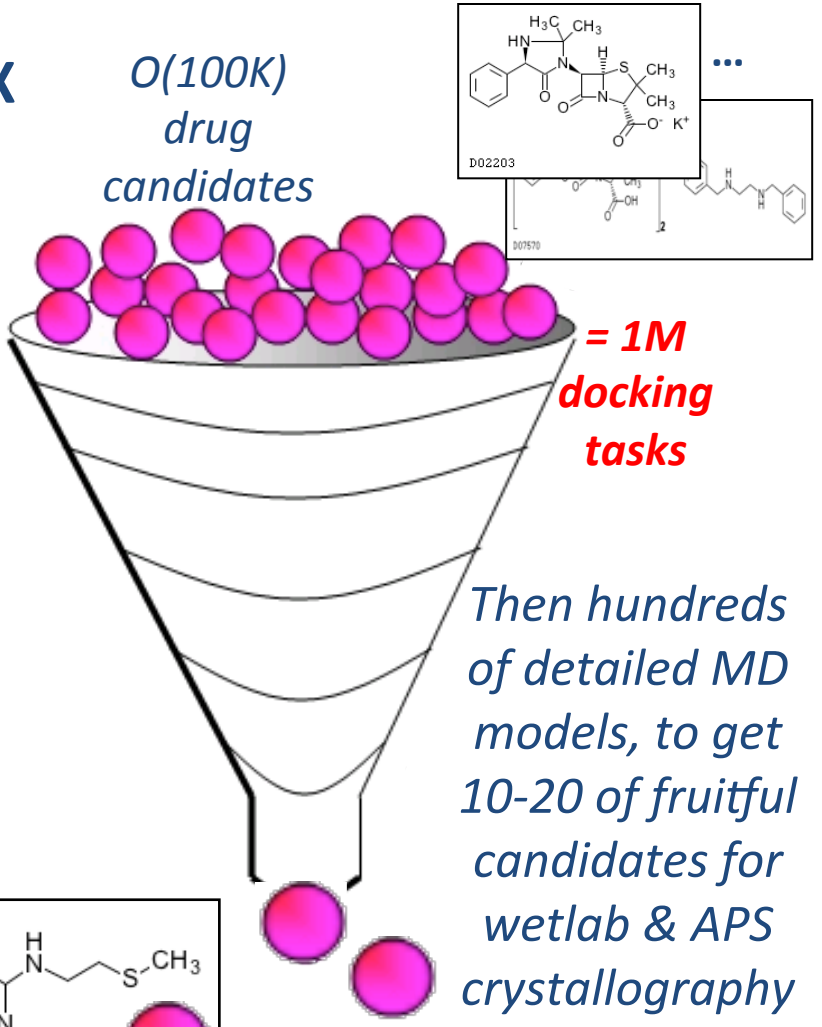
Example application: protein-ligand docking for drug screening



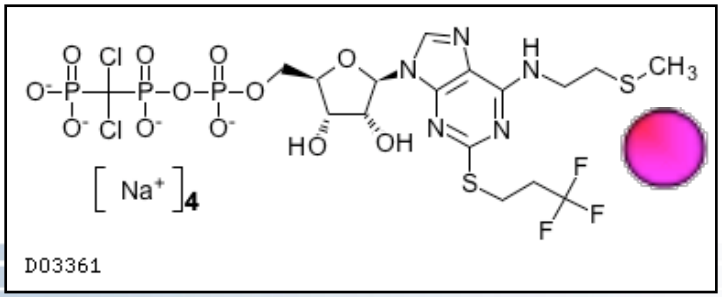
O(10) proteins implicated in a disease

X

O(100K) drug candidates



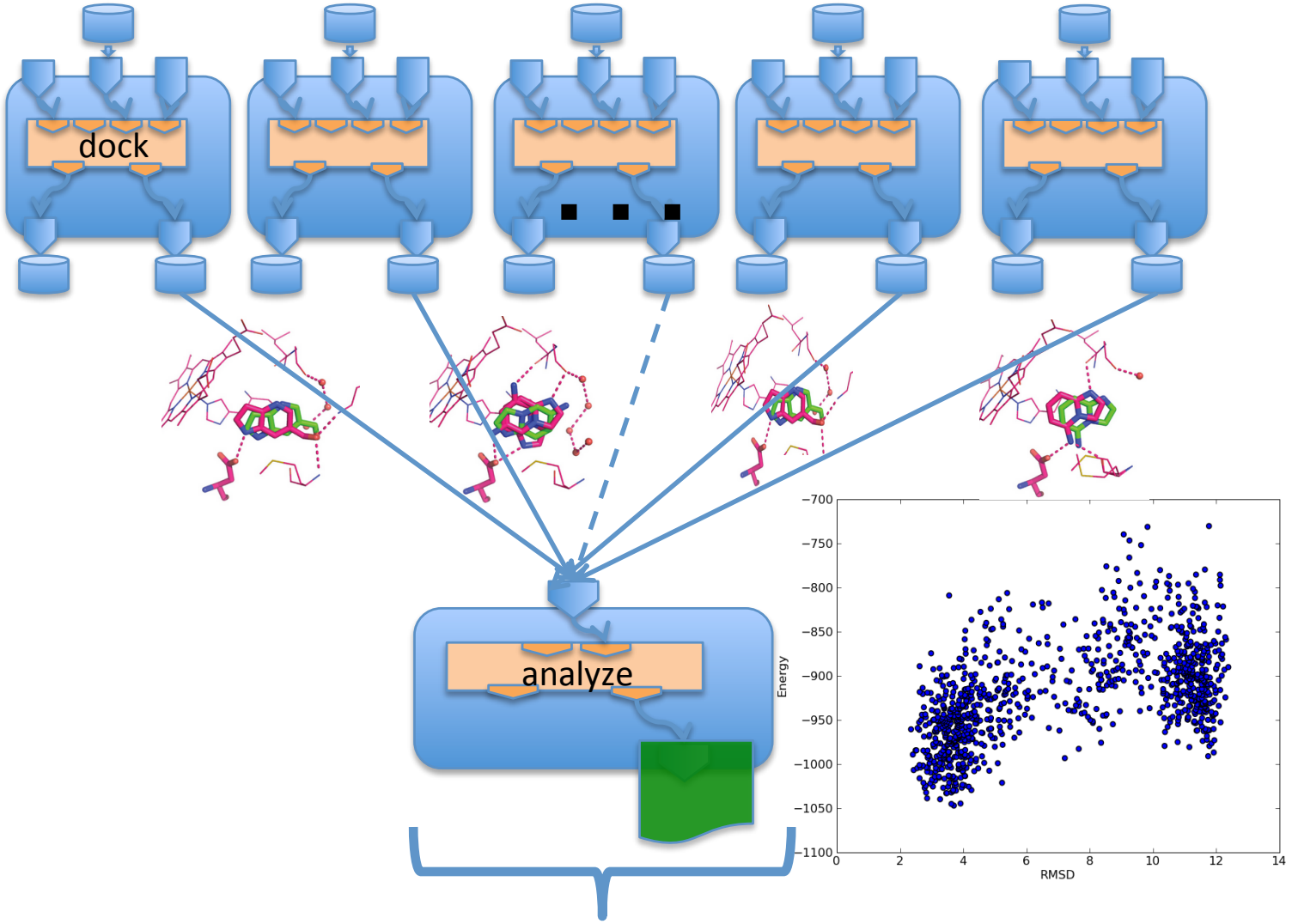
Then hundreds of detailed MD models, to get 10-20 of fruitful candidates for wetlab & APS crystallography



Work of M. Kubal, T.A.Binkowski, and B. Roux

Problem: *How to code such applications?*

1,000,000 runs of the "dock" application



… and repeat this pattern many times – often pipelined

Solution: *Compact, portable scripting*

Swift code excerpt:

```
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)
```

To run:

```
swift –site stampede,trestles     \
      docksweep.swift
```

What is Swift? *A powerful but "little" scripting language*

Swift is a parallel scripting language

Composes applications linked by files

Easy to write: a simple, high-level language

Small Swift scripts can do large-scale work

Easy to run: on clusters, clouds and grids

Sends work to XSEDE, Amazon, OSG, Cray

Fast and highly parallel

Runs a million tasks on thousands of cores

*hundreds of tasks per second**

** thousands to billions per second on petascale supercomputers (Blue Waters and Mira)*

Swift does 4 important things for you:

Makes parallelism more transparent

implicitly parallel functional dataflow programming

Makes computing location more transparent

runs your script on multiple distributed sites and diverse computing resources (desktop to petascale)

Makes basic failure recovery transparent

Retries/relocates failing tasks

Can restart failing runs from point of failure

Records provenance of data derivation

Made possible through functional encapsulation

Swift programming model

- Data types

```
int    i = 4;
int    A[];
string s = "hello world";
```

- Mapped data types

```
file image<"snapshot.jpg">;
```

- Structured data

```
image  A[]<array_mapper…>;
type protein {
   file pdb;
   file docking_pocket;
}
protein p<ext; exec=protein.map>;
```

- Conventional expressions

```
if (x == 3) {
    y = x+2;
    s = strcat("y: ", y);
}
```

- Parallel loops

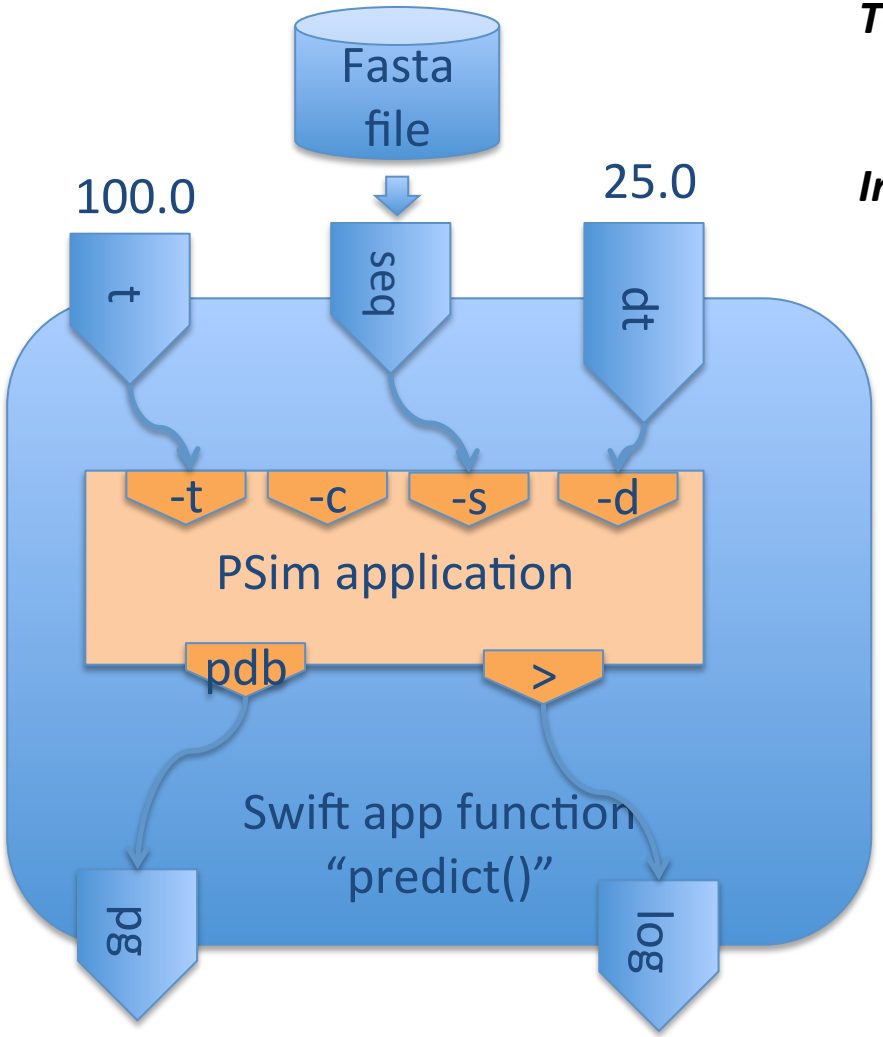
```
foreach f,i in A {
    B[i] = convert(A[i]);
}
```

- Composition and Data flow

```
a = analyze(B[0], B[1]);
b = analyze(B[2], B[3]);
c = compare(a,b)
```

Swift: A language for distributed parallel scripting, J. Parallel Computing, 2011

app( ) functions specify command line arg passing



To run:

psim -s 1ubq.fas -pdb p -t 100.0 -d 25.0 >log

In Swift code:

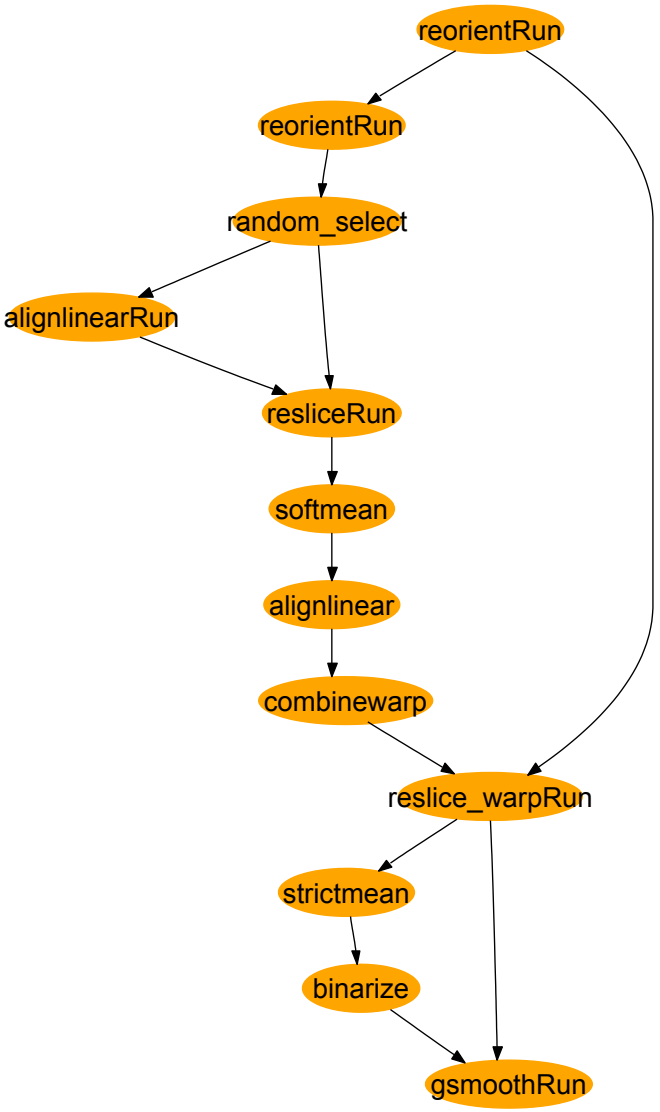
```
app (PDB pg, Text log) predict (Protein seq,
                                Float t, Float dt)
{
  psim "-c" "-s" @pseq.fasta "-pdb" @pg
       "–t" temp "-d" dt;
}


Protein p <ext; exec="Pmap", id="1ubq">;
PDB structure;
Text log;


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

Functional composition in *Swift*

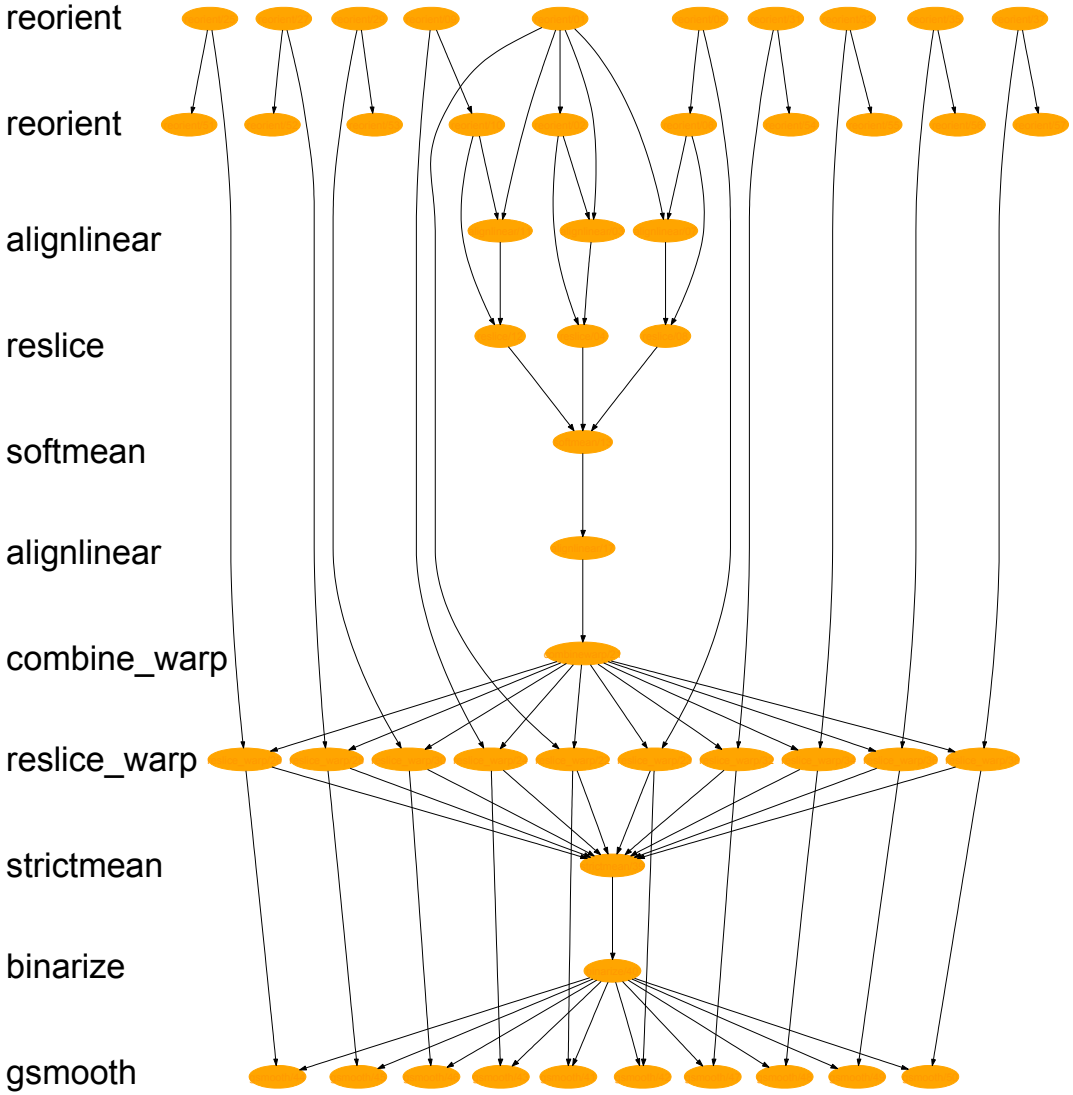
```
1.  Sweep(Protein pSet[ ])
2.  {
3.    int nSim = 1000;
4.    int maxRounds = 3;
5.    float startTemp[ ] = [ 100.0, 200.0 ];
6.    float delT[ ] = [ 1.0, 1.5, 2.0, 5.0, 10.0 ];
7.    foreach p, pn in pSet {
8.      foreach t in startTemp {
9.        foreach d in delT {
10.         IterativeFixing(p, nSim, maxRounds, t, d);
11.       }
12.     }
13.   }
14. }
```

10 proteins x 1000 simulations x
3 rounds x 2 temps x 5 deltas
= 300K tasks

Spatial normalization of functional MRI runs



Dataset-level workflow



Expanded (10 volume) workflow

Complex scripts can be well-structured

programming in the large: fMRI spatial normalization script example

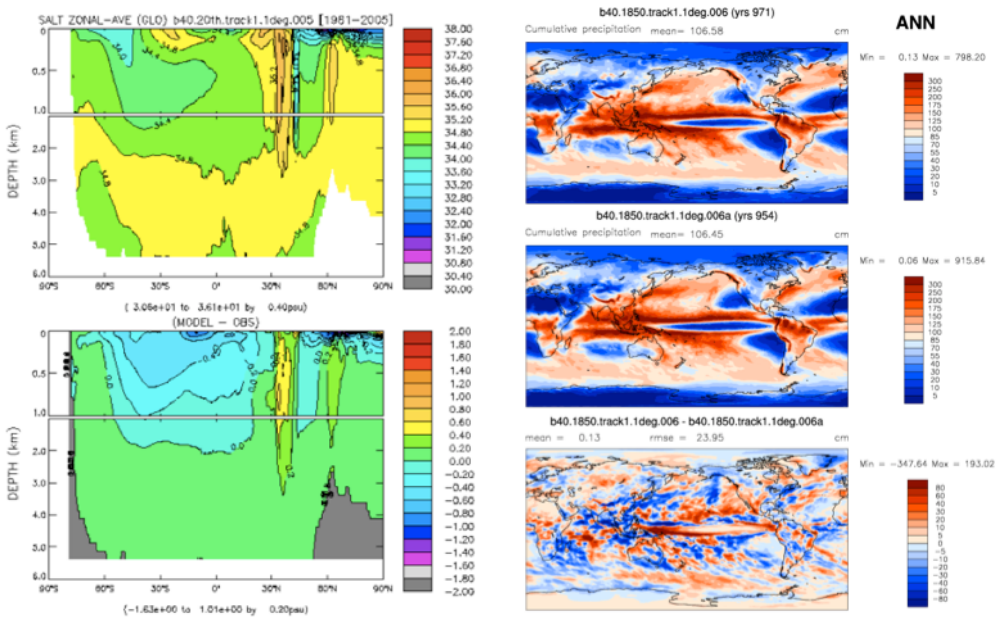
```
(Run snr) functional ( Run r, NormAnat a,
                       Air shrink )
{    Run yroRun = reorientRun( r , "y" );
     Run roRun = reorientRun( yroRun , "x" );
     Volume std = roRun[0];
     Run rndr = random_select( roRun, 0.1 );
     AirVector rndAirVec = align_linearRun( rndr, std, 12, 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" );
}
```

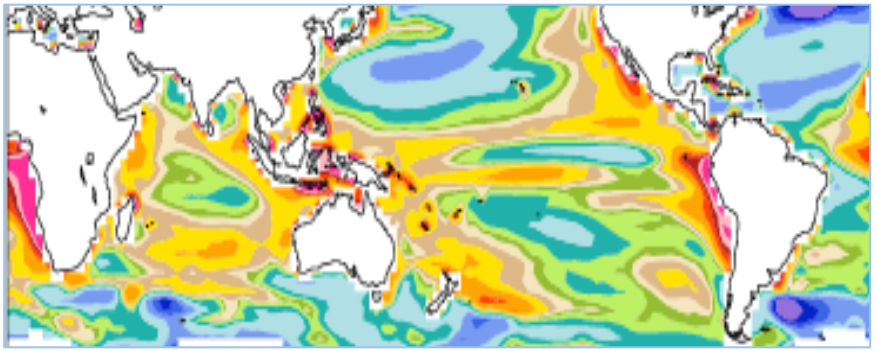
```
(Run or) reorientRun ( Run ir, string direction)
{
        foreach Volume iv, i in ir.v {
                or.v[i] = reorient(iv, direction);
        }
}
```

Benefit of implicit pervasive parallelism: Analysis & visualization of high-resolution climate models

parVis

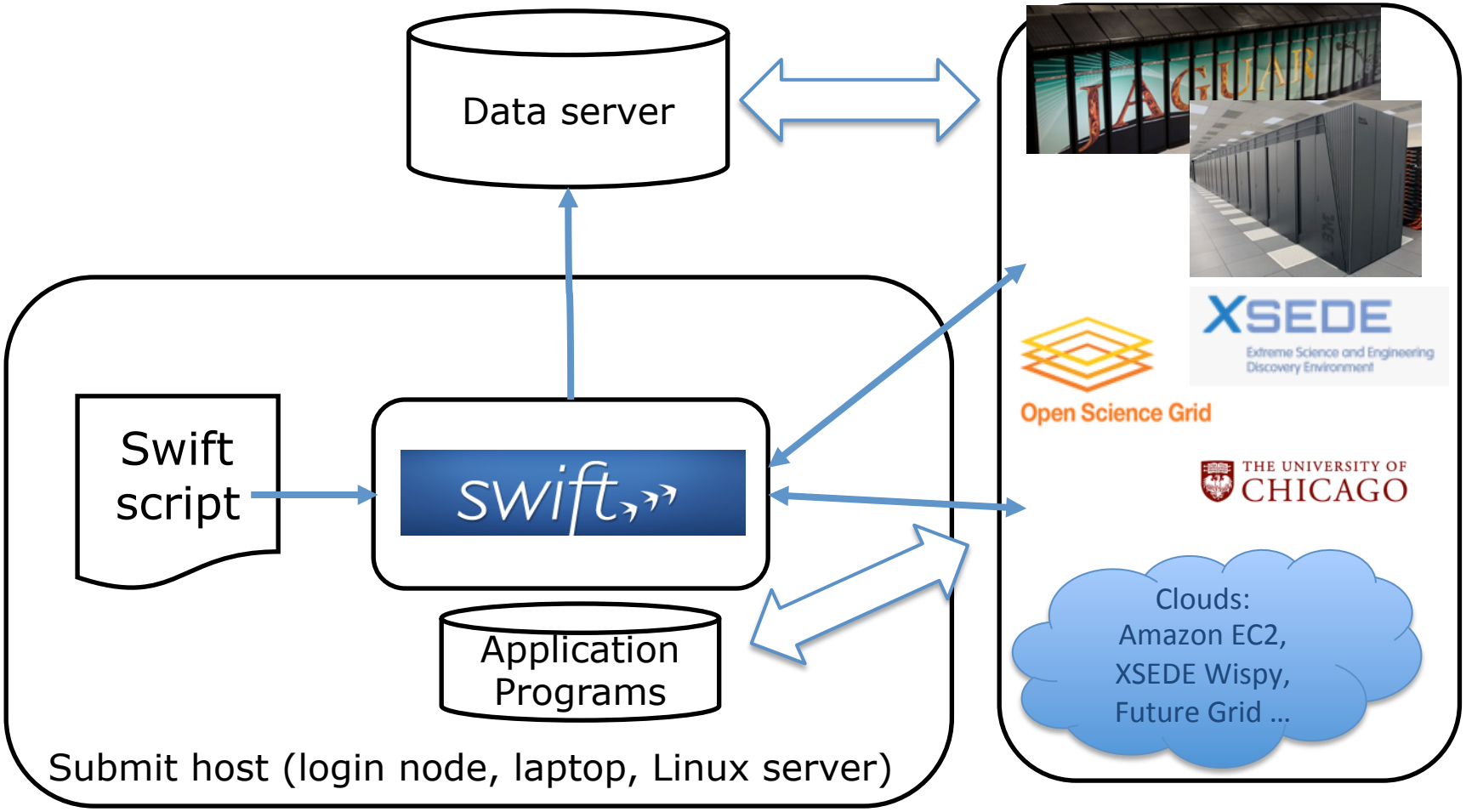
- Diagnostic scripts for each climate model (ocean, atmospehere, land, ice) were expressed in complex shell scripts
- Recoded in Swift, the CESM community has benefited from significant speedups and more modular scripts





Work of: J Dennis, M Woitasek, S Mickelson, R Jacob, M Vertenstein

Swift runs across diverse parallel platforms



Swift runtime system has drivers and algorithms to efficiently support and aggregate vastly diverse runtime environments

Pervasive parallel data flow

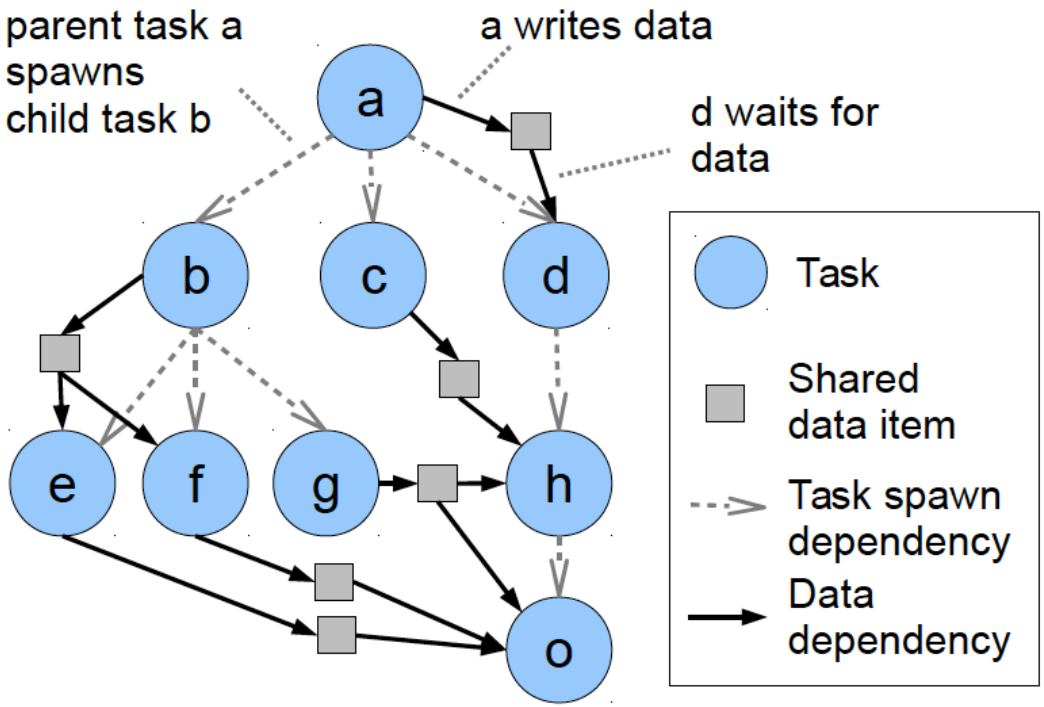
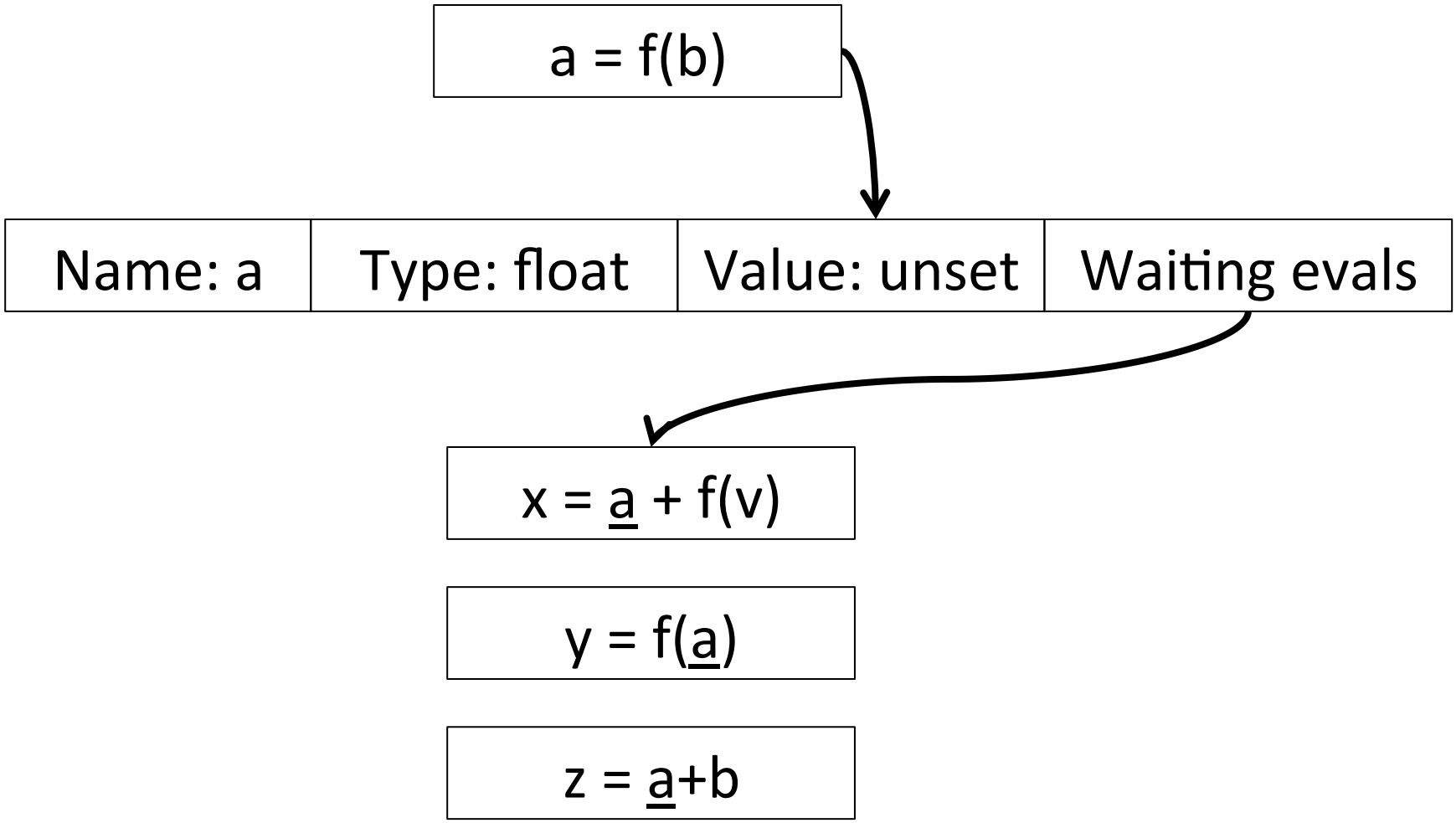


Fig. 1: Task and data dependencies in data-driven task parallelism, forming a spawn tree rooted at task a. Data dependencies on shared data defer execution of tasks until the variables are finalized.

Implicitly parallel functional dataflow - tracing our way back to the *future*

- Seminal work and visions
  - Future construct for Lisp (1977) and Act 1 (1981)
  - Jack Dennis's vision of Project X (later, Fresh Breeze) – 1975: pervasive implicit parallelism
  - VAL -> Sisal (Streams and Iteration in a single assignment language)
- Programming language models for productive parallel programming
  - Strand (Ian Foster, Stephen Taylor, 1988)
  - Program Composition Notation (PCN)
- Globus CoG
  - Karajan
  - Occam-like; used channels (futures were internal)
- GriPhyN
  - Virtual data model & language for data derivation recipes and provenance tracking
  - Pegasus workflow engine
- Swift

The future



The promise of the Future

- "future" construct coined Henry Lieberman – 1977, MIT, for Lisp
- Led to new language, Act 1 – 1980

Parallelism in Act 1 — 2.1 Dynamic allocation of processes parallels dynamic allocation of storage

April 16, 1981 at 0:09 — Page 4 — Henry Lieberman

We propose that parallel processes be allocated dynamically rather than statically. We will introduce actors called *futures* which represent parallel computations. There is a primitive which magically creates them whenever you need them. When they're no longer necessary, they get *garbage collected*, when they become inaccessible. The number of processes need not be bounded in advance, and if there are too many processes for the number of real physical processors you have on your computer system, they are automatically *time shared*. Thus the user can *pretend* that processor resources are practically infinite.

Inspiration: Jack Dennis

General purpose parallel machines based on a dataflow graph model of computation

Inspired all the major players in dataflow during seventies and eighties, including Kim Gostelow and I @ UC Irvine

… from Dataflow talk by Arvind (MIT)

Sisal

- Developed 1983, revised 1985 by James McGraw and collaborators at Manchester, CSU, LLNL and Dec
- Pascal-inspired syntax
- Full implicit parallelism through single assignment
- Impressively high quality compiler (as assessed by WPI students in 2010 vs. gcc)
- Extended for distributed shared memory at Arizona
- Derived from Val by Jack Dennis

```
define main

type OneDim = array [ real ];
type TwoDim = array [ OneDim ];

function generate( n : integer
                   returns TwoDim, TwoDim )

   for i in 1, n cross j in 1, n
   returns array of real(i)/real(j)
           array of real(i)*real(j)
   end for
end function % generate

function doit( n : integer; A, B : TwoDim
               returns TwoDim )

   for i in 1, n cross
       j in 1, n
      c := for k in 1, n
             t := A[i,k] * B[k,j]
           returns value of sum t
           end for
   returns array of c
   end for
end function % doit

function main(  n : integer returns TwoDim )

let A, B := generate( n )
in  doit( n, A, B )
end let

end function % main
```

Strand

- 1988 By Ian Foster
- Logic programming model ala Prolog
- Supported productive composition and parallelism

```
align_chunk(Sequences,Alignment) :-
    pins(Chunks,BestPin),
    divide(Sequences,BestPin,Alignment).

pins(Chunk,BestPin) :-
    cps(Chunk,CpList),
    c_form_pins(CpList,PinList),
    best_pin(Chunk,PinList,BestPin).

cps([Seq|Sequences],CpList) :-
    CpList := [CPs|CpList1],
    c_critical_points(Seq,CPs),
    cps(Sequences,CpList1).
cps([],CpList) :- CpList := [].

divide(Seqs,Pin,Alignment) :-
  Pin =\= [] |
    split(Seqs,Pin,Left,Right,Rest),
    align_chunk(Left,LAlign) @ random,
    align_chunk(Right,RAlign) @ random,
    align_chunk(Rest,RestAlign) @random,
    combine(LAlign,RAlign,RestAlign,Alignment).
divide(Seqs,[],Alignment) :-
    c_basic_align(Seqs,Alignment).
```

Figure 4: Genetic Sequence Alignment Algorithm

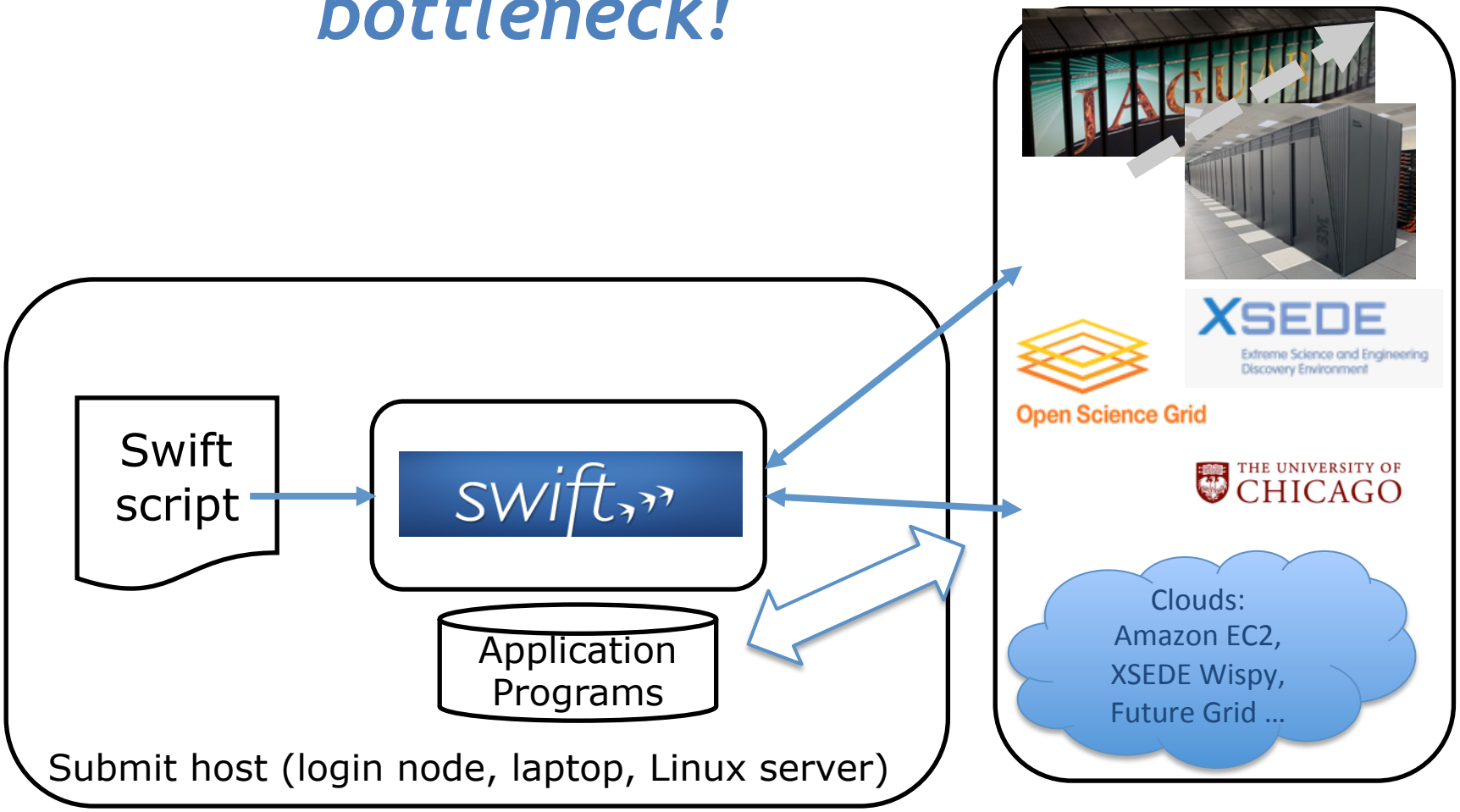
PCN

- Developed 1990 by Argonne and Caltech (Stephen Taylor, Ian Foster, and collaborators)
- Supported composition of external code (C, Fortran)
- Lightweight threads, easy-to-express parallelism,

```
align_chunk(sequences,alignment)
{|| pins(chunks,bestpin),
    divide(sequences,bestpin,alignment)
}

pins(chunk,bestpin)
{|| cps(chunk,cplist),
    c_form_pins(cplist,pinlist),
    best_pin(chunk,pinlist,bestpin)
}

cps(sequences,cplist)
{ ? sequences ?= [seq|sequences1] ->
    {|| cplist = [cps|cplist1],
        c_critical_points(seq,cps),
        cps(sequences1,cplist1)
    },
    sequences ?= [] -> cplist = []
}

divide(seqs,pin,alignment)
{ ? pin != [] ->
    {|| split(seqs,pin,left,right,rest),
        align_chunk(left,lalign),
        align_chunk(right,ralign),
        align_chunk(rest,restalign),
        combine(lalign,ralign,restalign,alignment)
    },
    pin == [] ->
        c_basic_align(seqs,alignment)
}
```

Figure 8: PCN Version of Figure 4

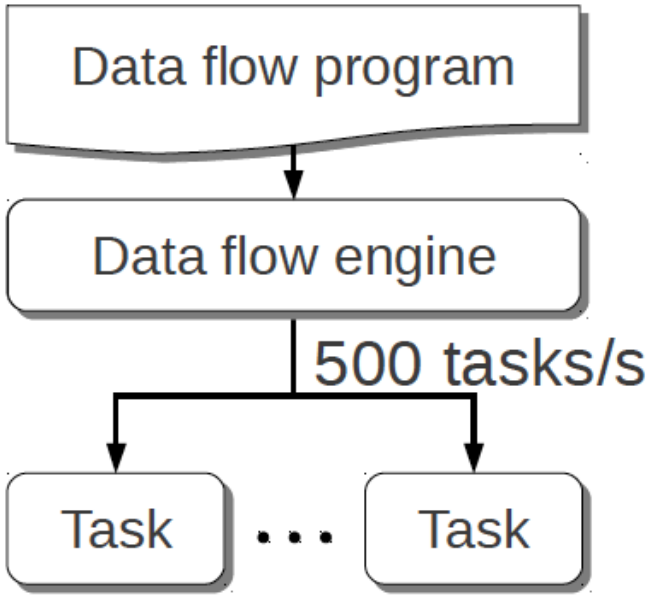
Problem: *Centralized evaluation can be a bottleneck!*



500 tasks/sec is good for workflow,
but can't utilize much of a large supercomputer

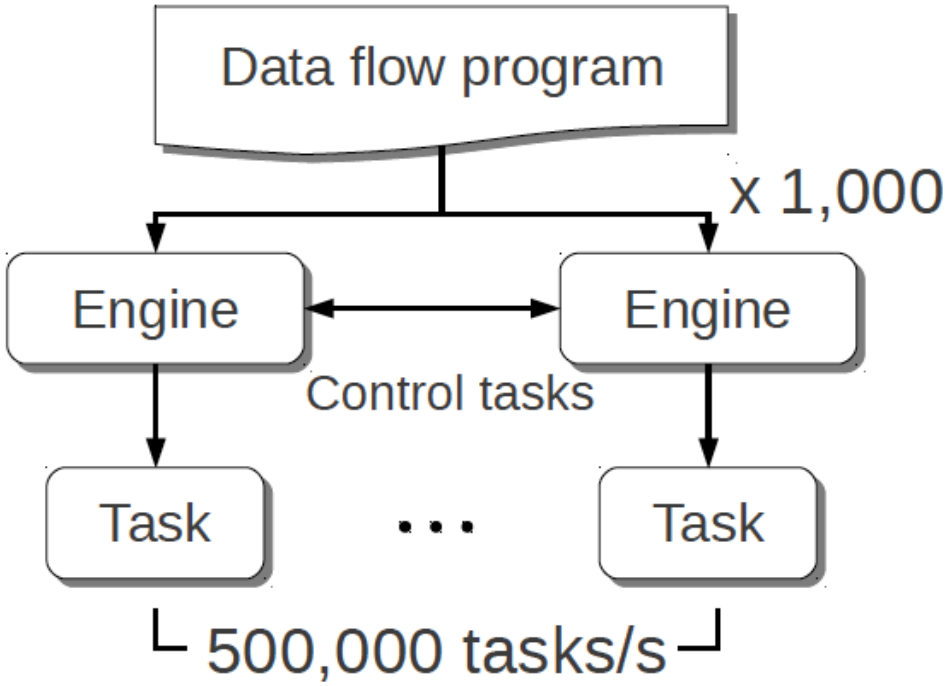
Centralized evaluation can be a bottleneck at extreme scales

Had this (Swift/K):



Centralized evaluation

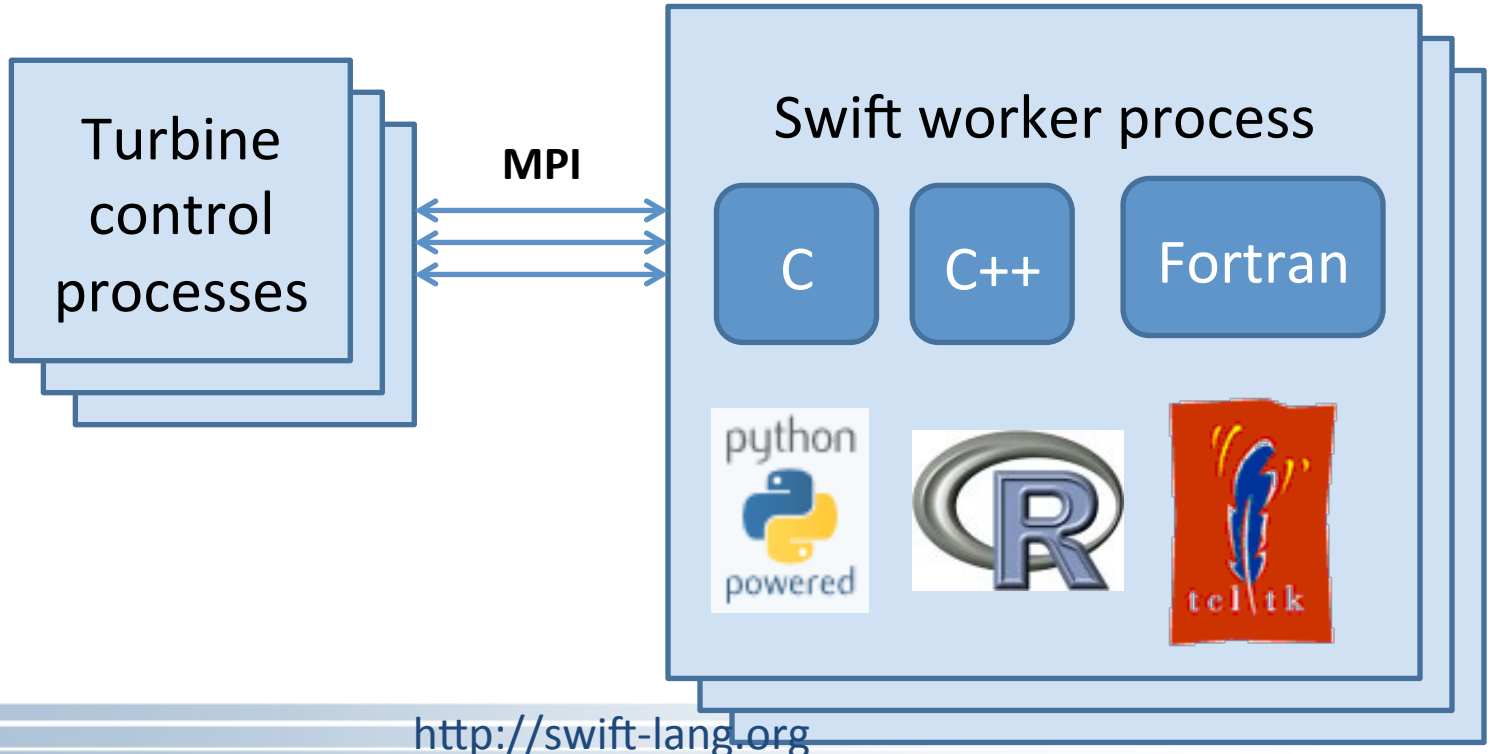
For extreme scale, we need this (Swift/T):



Distributed evaluation

Swift/T: High-level model with Turbine runtime

- Script-like global-view programming with “leaf tasks”- function calls in C, C++, Fortran, Python, R, or Tcl
- Leaf tasks can be MPI programs, etc. Can be separate processes if OS permits.
- Distributed, scalable runtime manages tasks, load balancing, data movement
- User function calls to external code run on 1000’s of worker nodes
- Like master-worker but with the expressive Swift language to control progress



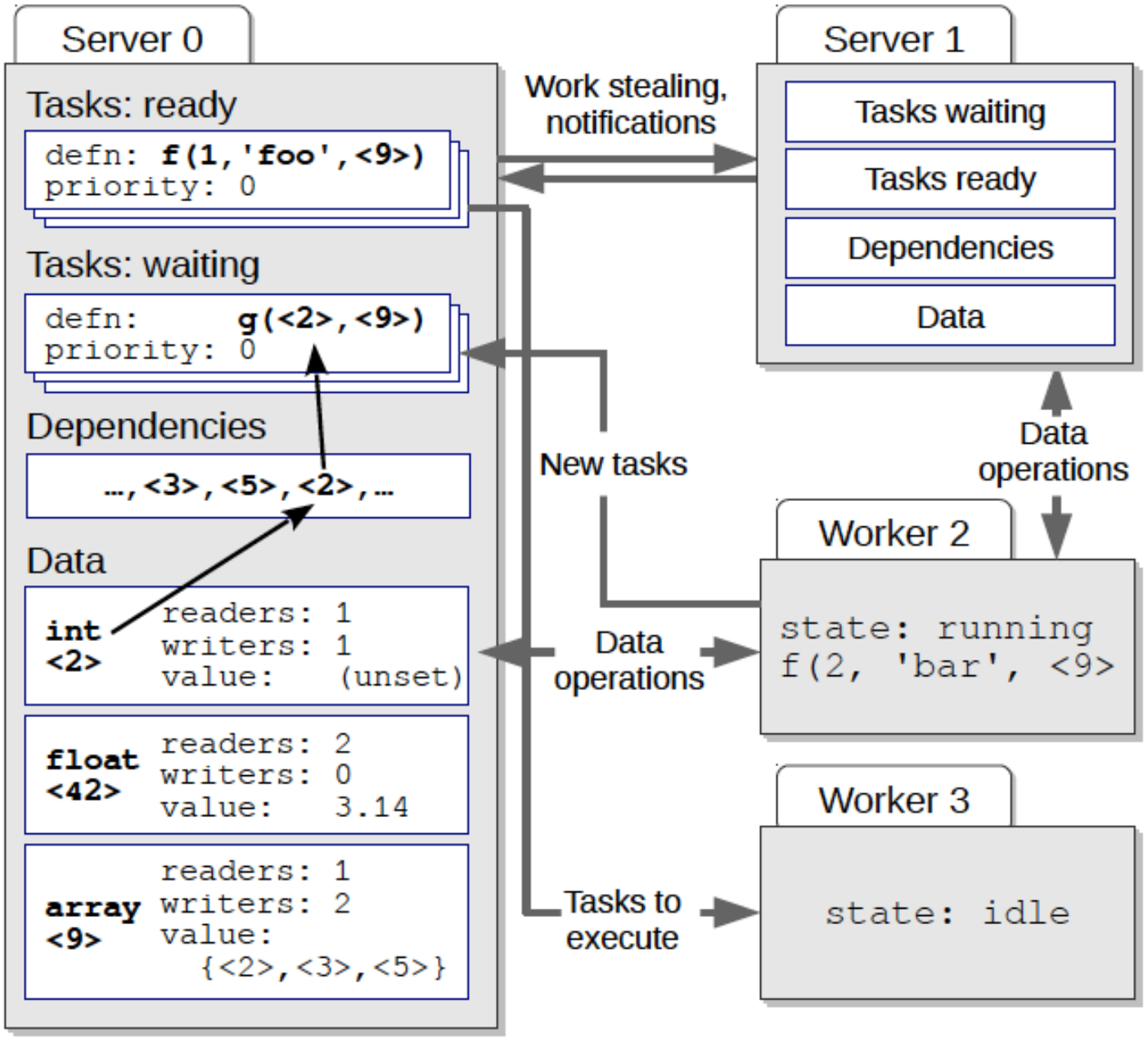


Fig. 4: Runtime architecture showing distributed worker processes coordinating through task and data operations.

SC14 paper on stc optimization

Compiler Techniques for Massively Scalable Implicit Task Parallelism

Timothy G. Armstrong,* Justin M. Wozniak,†‡ Michael Wilde,†‡ Ian T. Foster*†‡
*Dept. of Computer Science, University of Chicago, Chicago, IL, USA
†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
‡Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

***Abstract*—Swift/T is a high-level language for writing concise, deterministic scripts that compose serial or parallel codes implemented in lower-level programming models into large-scale parallel applications. It executes using a data-driven task parallel execution model that is capable of orchestrating millions of concurrently executing asynchronous tasks on homogeneous or heterogeneous resources. Producing code that executes efficiently at this scale requires sophisticated compiler transformations: poorly optimized code inhibits scaling with excessive synchronization and communication. We present a comprehensive set of compiler techniques for data-driven task parallelism, including novel compiler optimizations and intermediate representations. We report application benchmark studies, including unbalanced tree search and simulated annealing, and demonstrate that our techniques greatly reduce communication overhead and enable extreme scalability, distributing up to 612 million dynamically load balanced tasks per second at scales of up to 262,144 cores *without* explicit parallelism, synchronization, or load balancing in application code.**

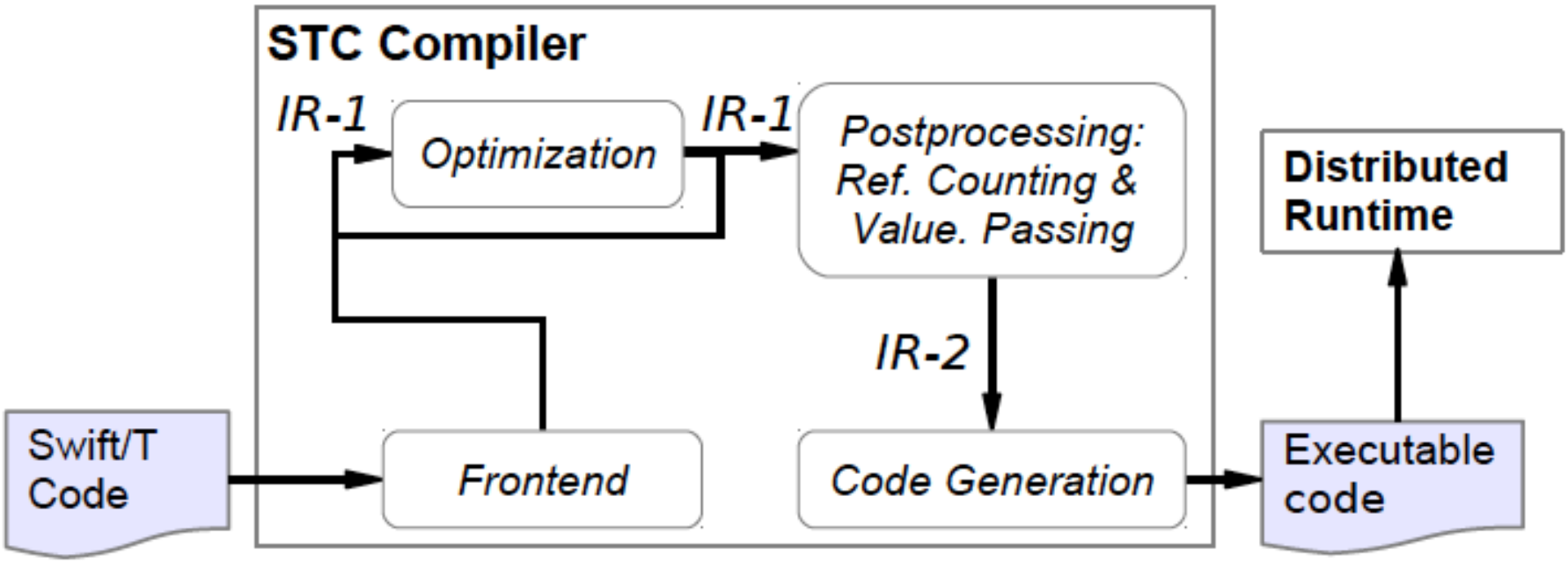
I. Introduction

In recent years, large-scale computation has become an indispensable tool in many fields, including those that have not traditionally used high-performance computing. These include data-intensive applications such as machine learning and data-aware task scheduling. Recent work has explored implementing this execution model with libraries and conservative language extensions to C for distributed-memory and heterogenous systems [3], [8], [9], [28] and has shown that performance can match or exceed performance of code directly using the underlying interfaces (e.g., message passing or threads). One reason for this success is that sophisticated algorithms for load balancing (e.g., work stealing) or data movement, usually impractical to reimplement for each application, can be implemented in an application-independent manner. Another reason is that the asynchronous execution model is effective at hiding latency and exploiting available resources in applications with irregular parallelism or unpredictable task runtimes.

Swift/T [36] is a high-level implicitly parallel programming language that aims to make writing massively parallel code for this execution model as easy and intuitive as sequential scripting in languages such as Python. Implementing a very high-level language such as Swift/T efficiently and scalably is challenging, however, because the programmer only specifies synchronization and communication implicitly through function composition or reads and writes to variables and data structures. Thus, internode data movement, parallel task

Thu 4 PM, 393-94-95

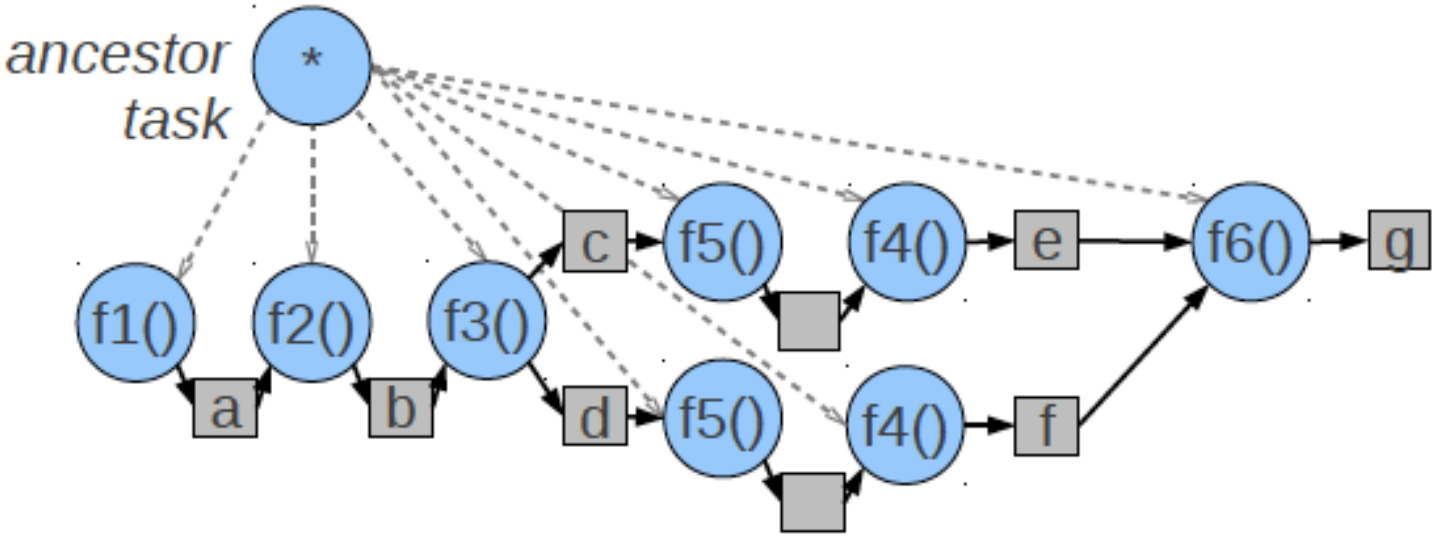
Swift/T optimizing compiler and IR's



Swift/T optimization challenge: distributed vars

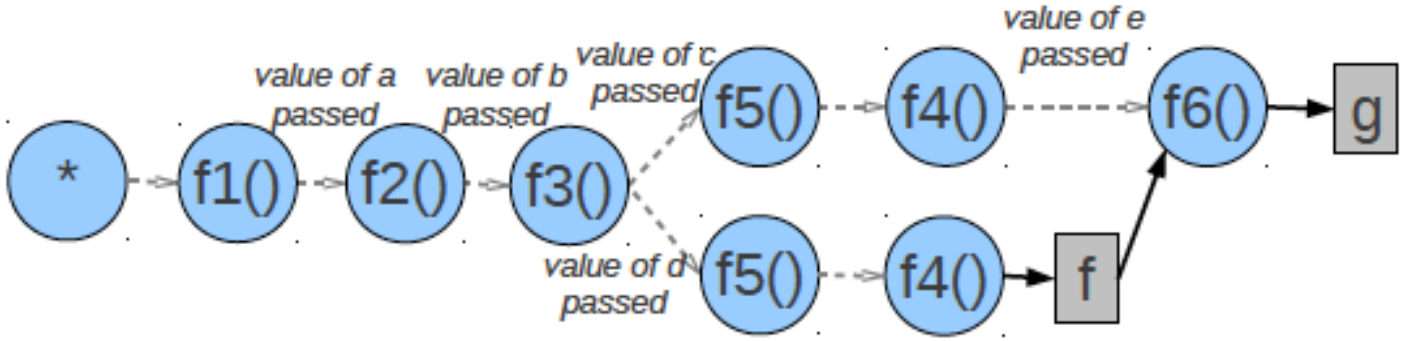
```
1 | a = f1();          b = f2(a);
2 | c, d = f3(a, b);   e = f4(f5(c);
3 | f = f4(f5(d);      g = f6(e, f);
```

(a) Swift/T code fragment

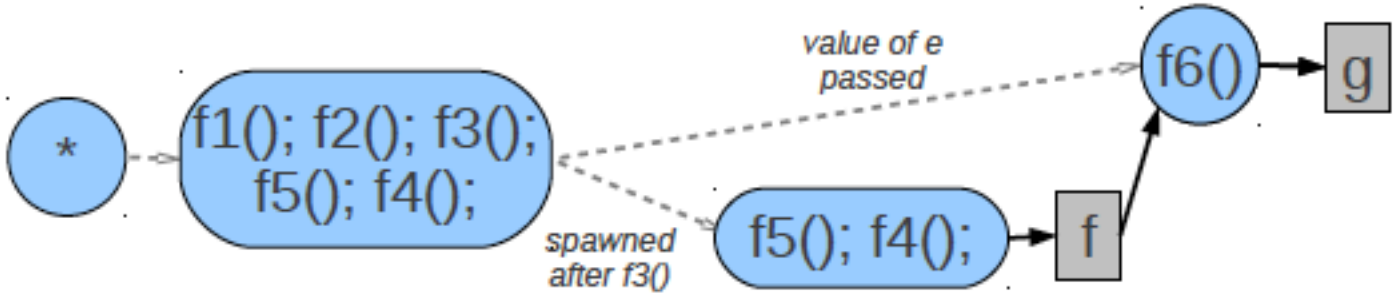


(b) Unoptimized version, passing data as shared data and perform synchronization

Swift/T optimizations improve data locality

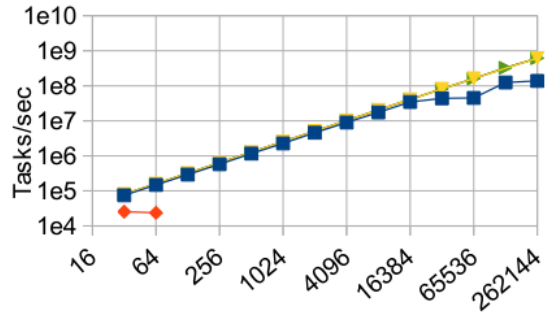


(c) After wait pushdown and elimination of shared data in favor of parent-to-child data passing

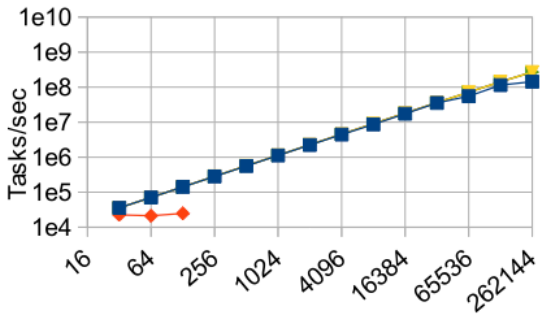


(d) After pipeline fusion merges tasks

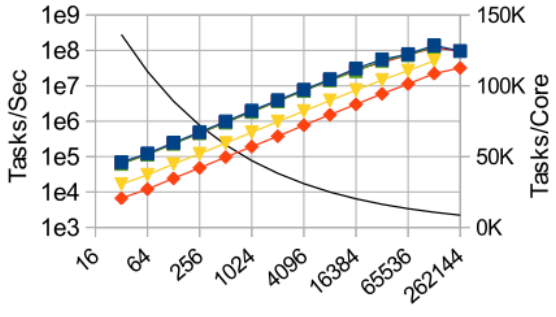
Swift/T application benchmarks on Blue Waters



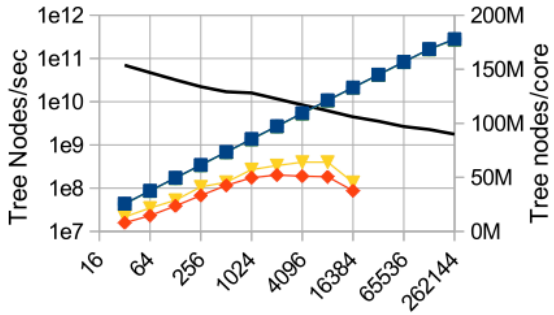
(a) Sweep weak scaling: 0.2 ms tasks



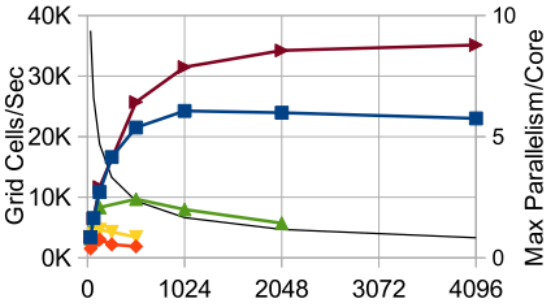
(b) Sweep weak scaling: 0.5 ms tasks



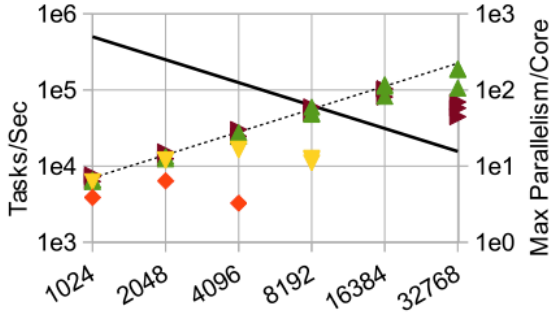
(c) ReduceTree scaling: 0 s tasks



(d) UTS scaling



(e) Wavefront: 5ms tasks



(f) Annealing strong scaling: 256 annealing processes × 2000 tasks per objective function × 5 parameter updates

Fig. 10: Application speedup and scalability at different optimization levels. X axes show scale in cores. Primary Y axes show application throughput in application-dependent terms. Secondary Y axes show problem size or degree of parallelism where applicable.

GeMTC: GPU-enabled Many-Task Computing

Motivation: Support for MTC on all accelerators!

Goals:

1) MTC support 2) Programmability

3) Efficiency 4) MPMD on SIMD

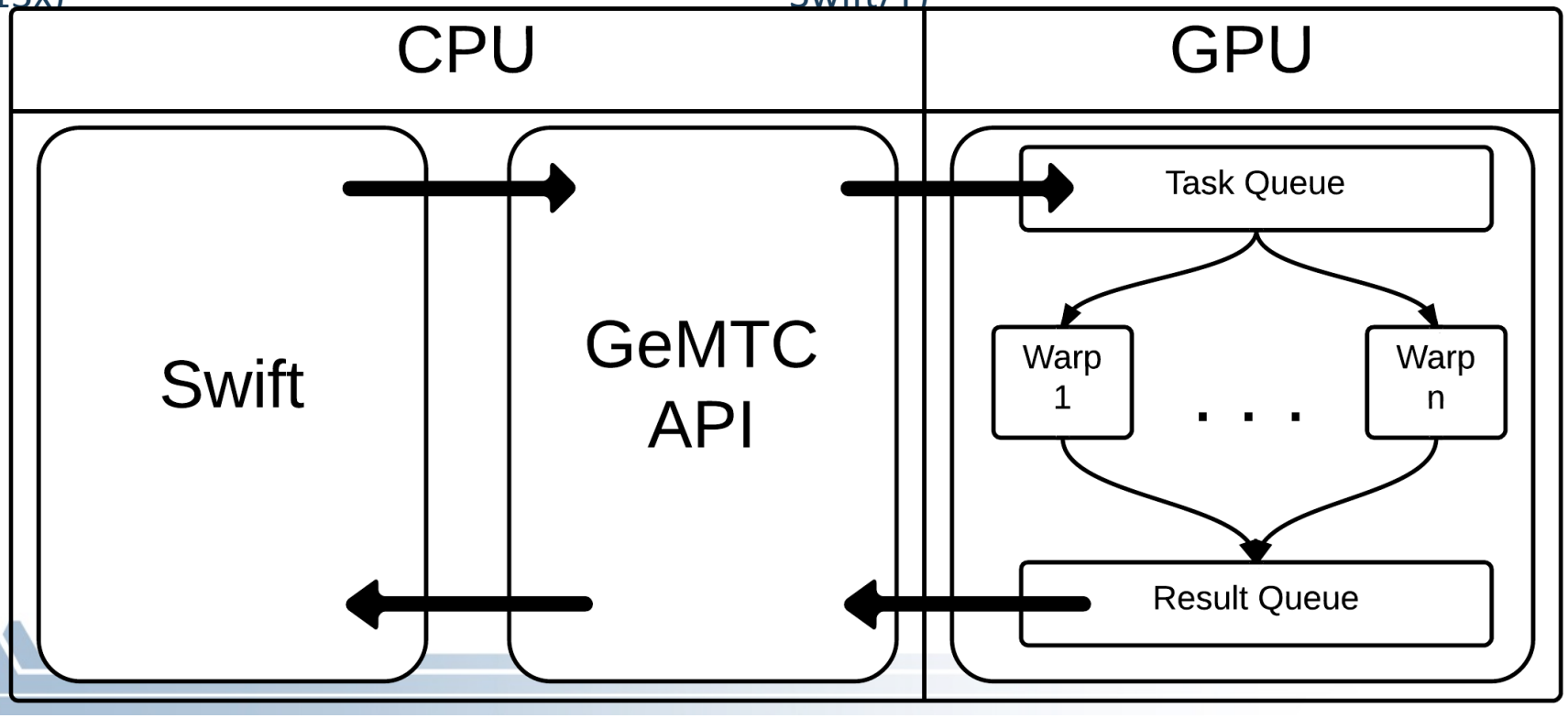
5) Increase concurrency from 15 to 192 (~13x)

Approach:

Design & implement GeMTC middleware:

1) Manages GPU 2) Spread host/device

3) Workflow system integration (with Swift/T)



What's next?

- Programmability
  - New patterns ala Van Der Aalst et al (workflowpatterns.org)
- Fine grained dataflow – programming in the smaller?
  - Run leaf tasks on accelerators (CUDA GPUs, Intel Phi)
  - How low/fast can we drive this model?
- PowerFlow
  - Applies dataflow semantics to manage and reduce energy usage
- Extreme-scale reliability
- Embed Swift semantics in Python, R, Java, shell, make
  - Can we make Swift "invisible"? Should we?
- Swift-Reduce
  - Learning from map-reduce
  - Integration with map-reduce

Conclusion: Implicitly parallel functional dataflow is useful

- Expressive
- Portable
- Usable
- Fast
- Applicable over a broad application space
- Not for everything
  - best for “programming in the large”
  - Supports MPI, OpenMP, GA
- Builds on a long legacy of inspiration
  - Is dataflow programming finally here to stay?

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.

Acknowledgments

- Swift is supported in part by NSF grants OCI-1148443 and PHY-636265, and the UChicago SCI Program
- Extreme scaling research on Swift (ExM project) is supported by the DOE Office of Science, ASCR Division, X-Stack program.

- The Swift team:
  - Tim Armstrong, Yadu Nand Babuji, Ian Foster, Mihael Hategan, Dan Katz, David Kelly, Ketan Maheshwari, Yadu Nand,
    Mike Wilde, Justin Wozniak, Zhao Zhang
- Special thanks to all our users and collaborators