

# Rapid development of highly concurrent multi-scale simulators with Swift

**Justin M. Wozniak**

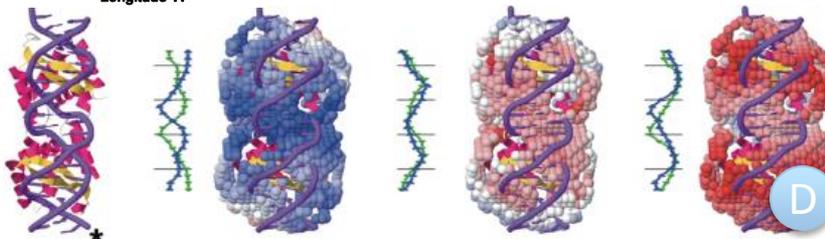
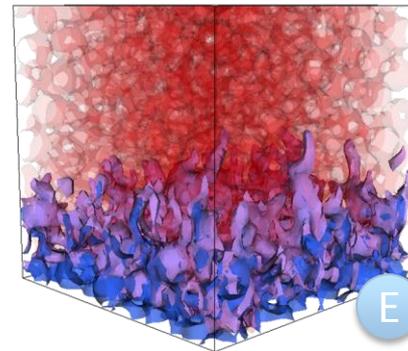
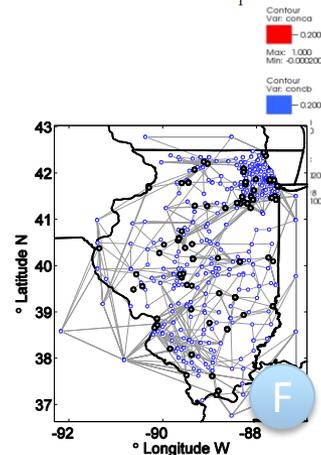
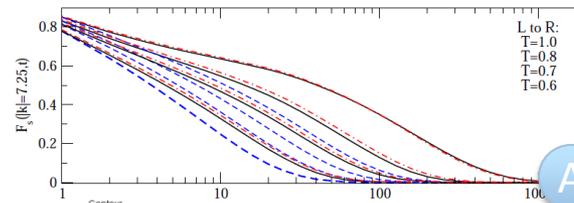
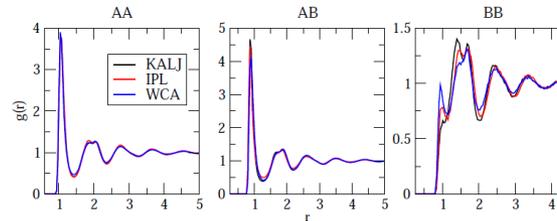
<http://exm.xstack.org>

wozniak@mcs.anl.gov

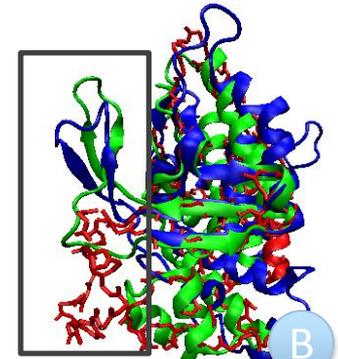
# Extreme-scale many-task applications

- A Simulation of super-cooled glass materials
- B Protein folding using homology-free approaches
- C Climate model analysis and decision making in energy policy
- D Simulation of RNA-protein interaction
- E Multiscale subsurface flow modeling
- F Modeling of power grid for OE applications

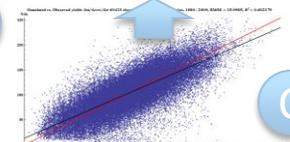
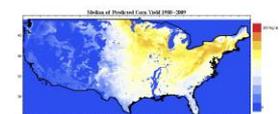
A-E have published science results obtained using Swift



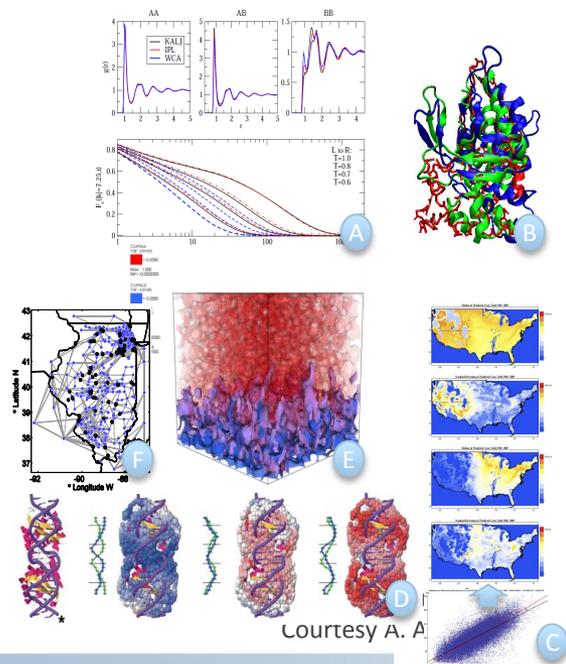
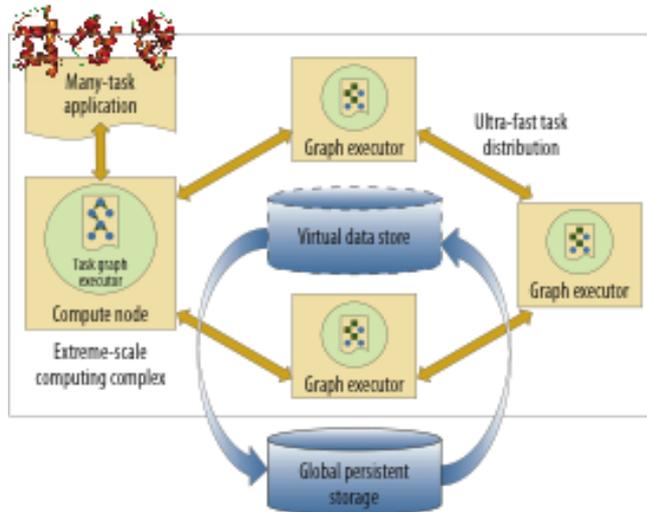
T0623, 25 res., 8.2Å to 6.3Å (excluding tail)



Initial  
Predicted  
Native



# ExM: extreme-scaling for many-task computing



- **Goal:** Target top-level application logic. Provide a simple programming environment at this level for common patterns
- **Applications:** Parameter studies, ensembles, Monte Carlo, branch-and-bound, stochastic programming, uncertainty quantification
- **Enablers:** Scalable parallel evaluation, dynamic load balancing, in-RAM datasets
- **Benefits:** Programmability, fault-recovery, power savings
- **Results:** New scalable Swift implementation, datastore and MTC publications

# Goal: Programmability for extreme scale

- Focus is “many-task” computing: higher-level applications composed of many run-to-completion tasks: **input**→**compute**→**output**  
Message passing handled by our implementation details
- Why is it relevant to extreme-scale computing?
  - Programmability
    - Increasing number of applications have this natural structure: material by design, inverse problems, stochastic programming, branch-and-bound problems, UQ.
    - Coupling extreme-scale applications to preprocessing, analysis, and visualization
  - Resilience
    - The functional programming model allows for re-execution of failed tasks
  - Power management
    - Graph structure of application upper levels may enable power scaling
- Challenges
  - Load balancing, data movement, expressibility
  - Debugging – experimental higher-level frameworks rarely have robust debugging tools! We have some work here...



# Programming model: all execution driven by parallel data flow

```
(int r) myproc (int i, int j)
{
    int f = F(i);
    int g = G(j);
    r = f + g;
}
```

- `f()` and `g()` are computed in parallel
- `myproc()` returns `r` when they are done
- This parallelism is *automatic*
- Works recursively throughout the program's call graph



# Nested loops can generate massive parallelism

## Protein folding example:

**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, pn in pSet {  
    foreach t in startTemp {  
      foreach d in delT {  
        ItFix(p, nSim, maxRounds, t, d);  
      }  
    }  
  }  
}
```

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

**Sweep ( ) ;**



# Characteristics of very large Swift programs

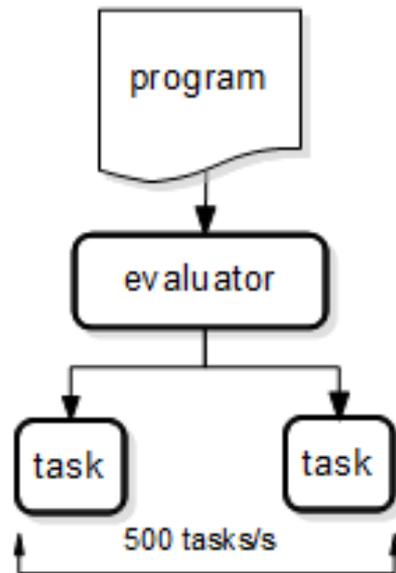
```
int X = 100, Y = 100;
int A[][];
int B[];
foreach x in [0:X-1] {
    foreach y in [0:Y-1] {
        if (check(x, y)) {
            A[x][y] = g(f(x), f(y));
        } else {
            A[x][y] = 0;
        }
    }
}
B[x] = sum(A[x]);
}
```

- The goal is to support billion-way concurrency:  $O(10^9)$
- Swift script logic will control trillions of variables and data dependent tasks
- Need to distribute Swift logic processing over the HPC compute system



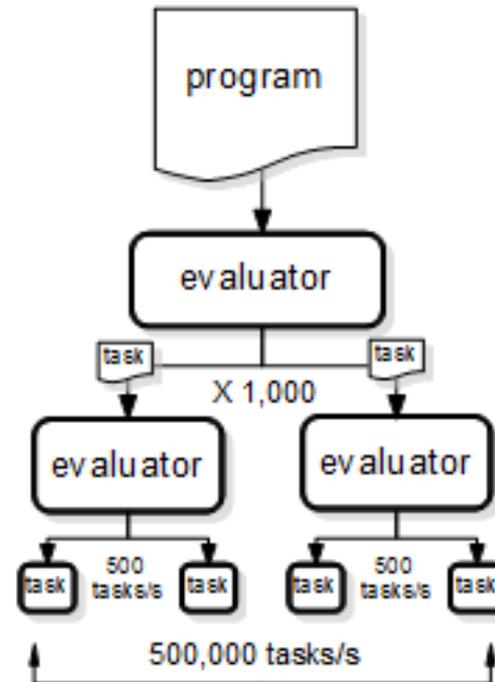
# Centralized evaluation can be a bottleneck

Have this:



Centralized evaluation

Want this:



Distributed evaluation

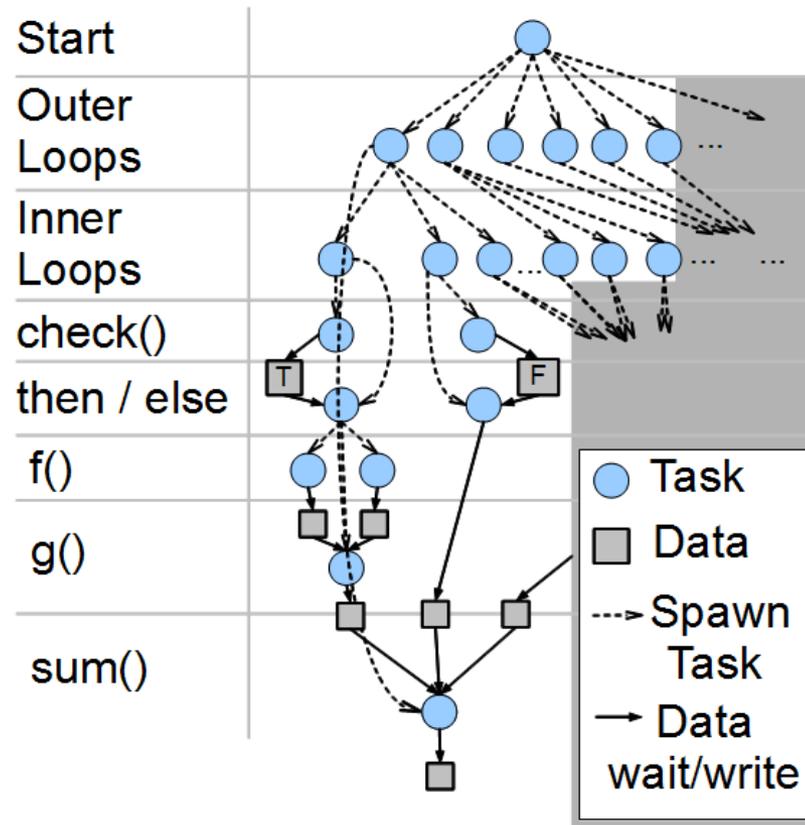


# Swift/T: Fully parallel evaluation of complex scripts

```

int X = 100, Y = 100;
int A[][];
int B[];
foreach x in [0:X-1] {
  foreach y in [0:Y-1] {
    if (check(x, y)) {
      A[x][y] = g(f(x), f(y));
    } else {
      A[x][y] = 0;
    }
  }
  B[x] = sum(A[x]);
}

```



- Tasks managed by ADLB

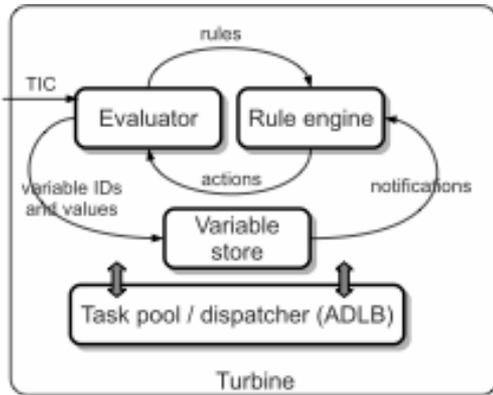


# What are the challenges in applying the many-task model at extreme scale?

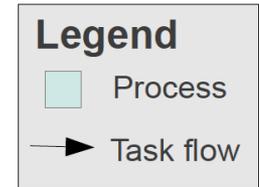
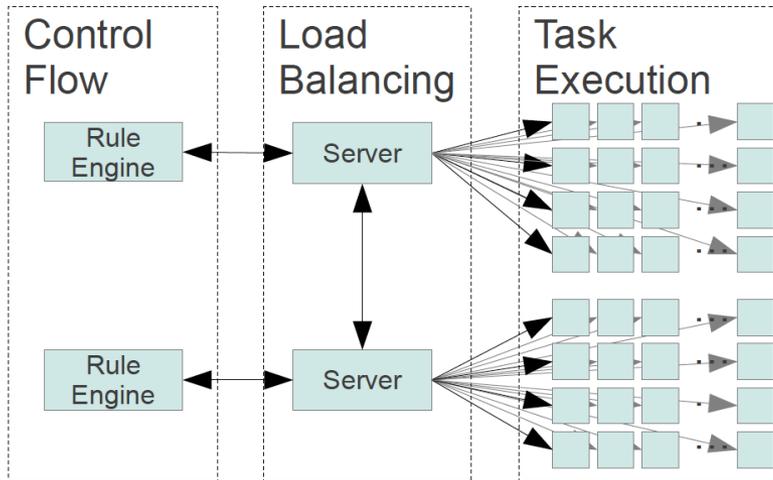
- Dealing with high task dispatch rates: depends on task granularity but is commonly an obstacle
  - Load balancing
  - Global and scoped access to script variables: increasing locality
- Handling tasks that are themselves parallel programs
  - We focus on OpenMP, MPI, and Global Arrays
- Data management for intermediate results
  - Object based abstractions
  - POSIX-based abstractions
  - Interaction between data management and task placement



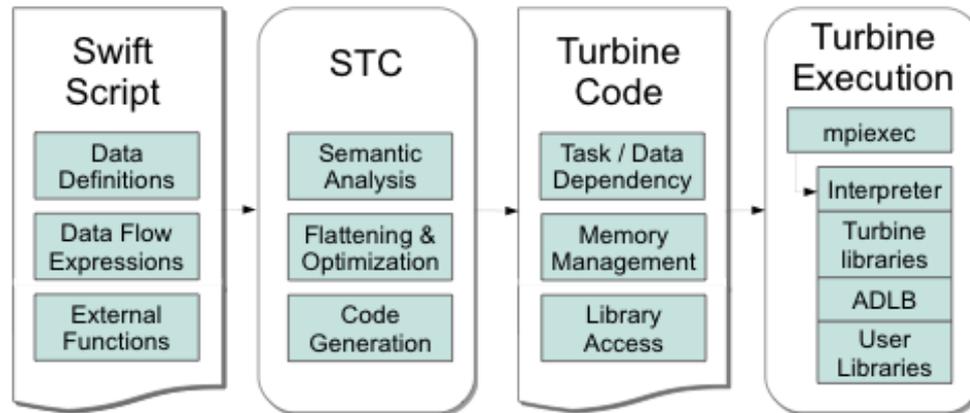
# Parallel evaluation of Swift/T in ExM



Big picture



Run time configuration

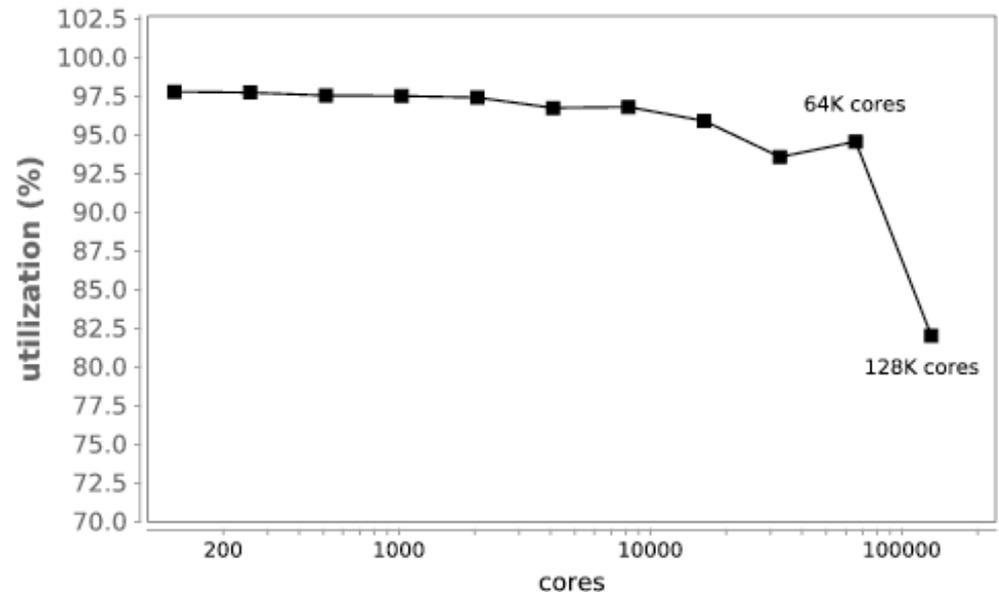


Software components

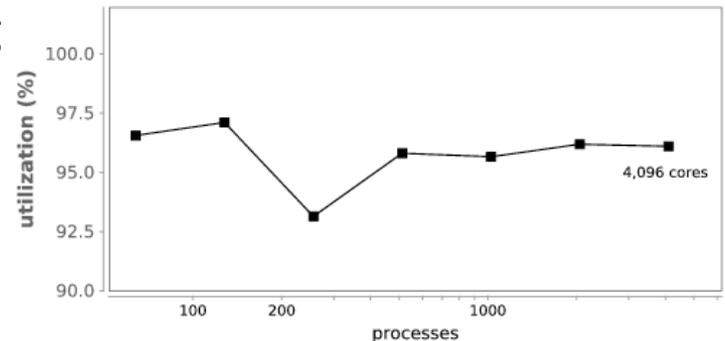


# Swift/T: Basic scaling performance

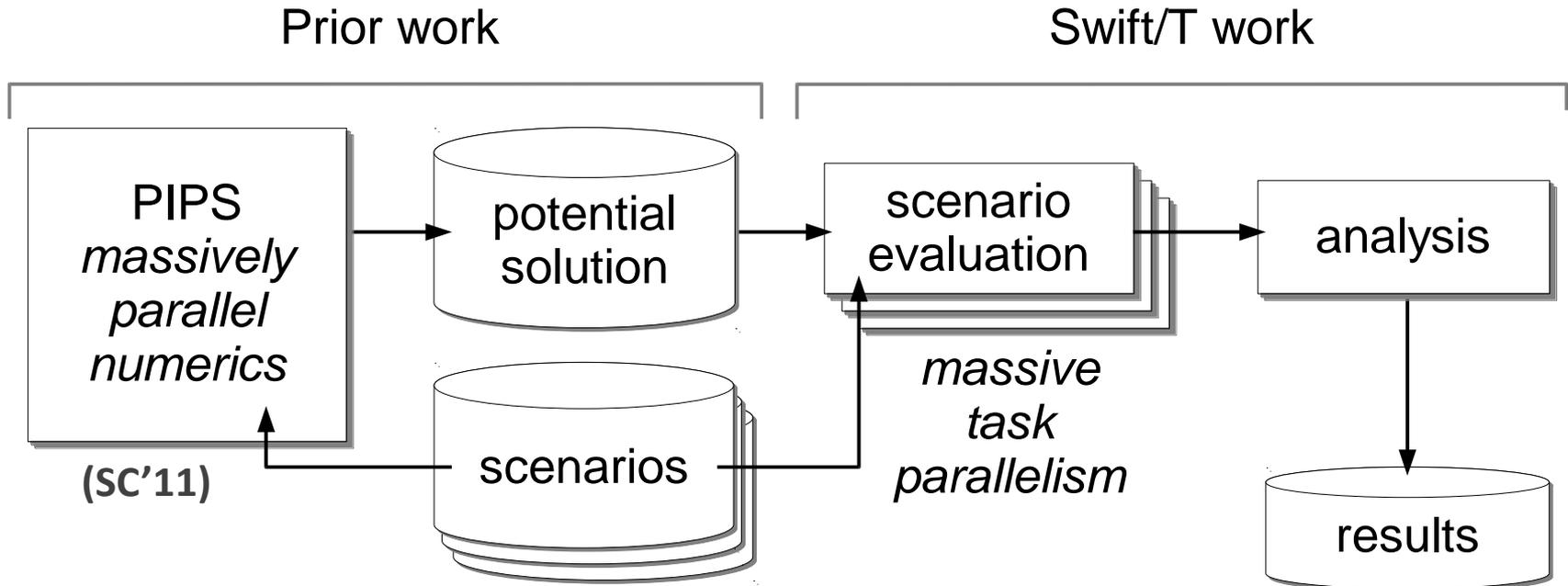
- Swift/T synthetic app: simple task loop
  - Scales to capacity of ADLB



- SciSolSim: collaboration graph modeling
  - ~400 lines Swift
  - C++ graph model; simulated annealing
  - Scales well to 4K cores in initial tests (further tests awaiting app debugging)



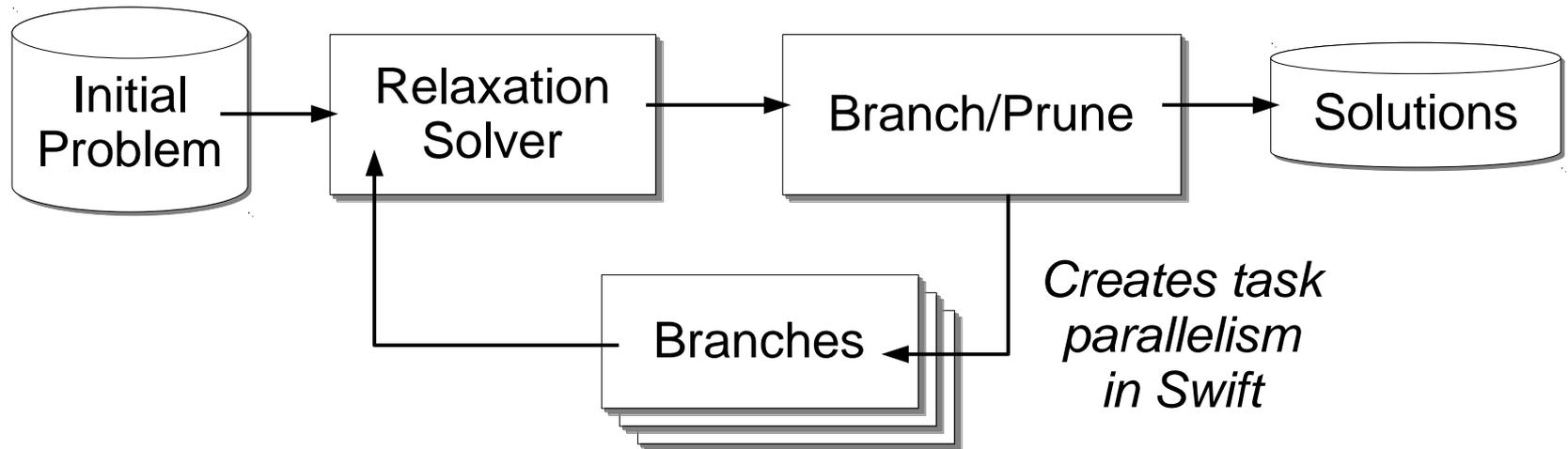
# Application: Power Grid Modeling (PIPS)



**Swift/T (and the many-task, dataflow model) complements existing application workflows**



# Application: Branch-and-Bound (Minotaur)



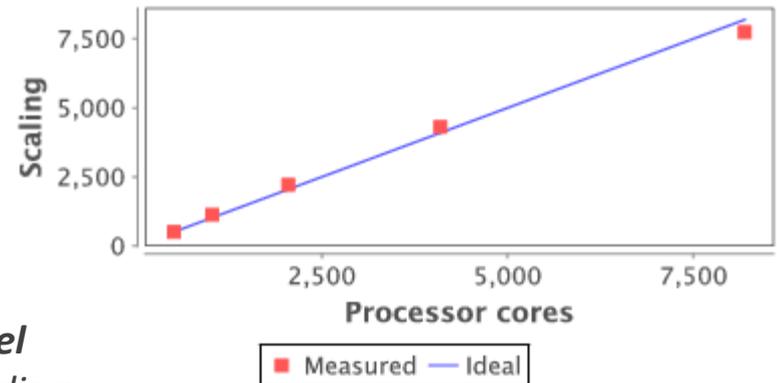
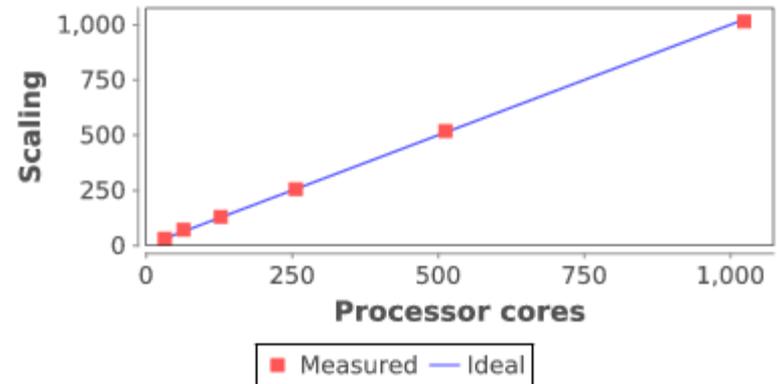
**Minimize some function via recursive search,  
allow only for integer solutions**

**Builds a new, scalable application from pre-existing components**



# Swift/T: PIPS scaling results

- Scaling result for original application:  
Limited by available data from PIPS team
- Scaling result for current application with parameter sweep over rounding parameter (integer programming)

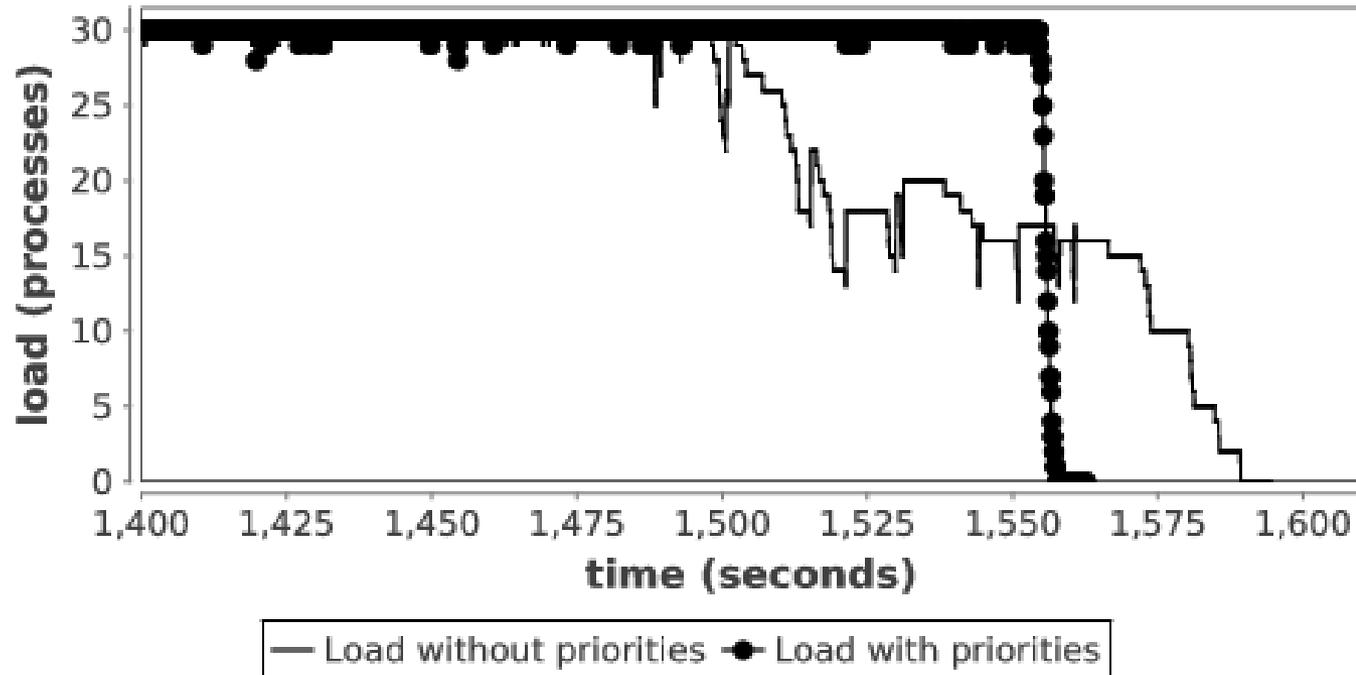


*“One major advantage of Swift is that its **high-level programming language** considerably shortens coding times, hence it allows more effort to be dedicated to **algorithmic developments**. Remarkably, using a scripting language has virtually no impact on the solution times and scaling efficiency, as recent Swift/PIPS-L runs show.”*

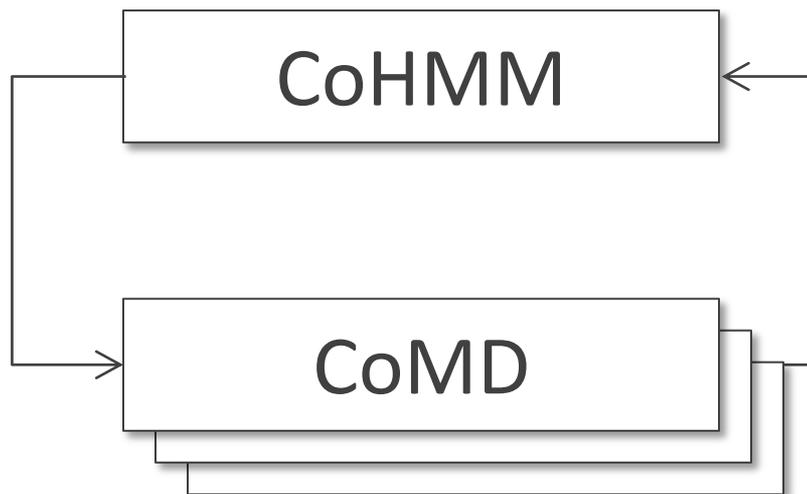
- Cosmin Petra

# Swift/T: Use of low-level features

- Variable-sized tasks produce trailing tasks: addressed by exposing ADLB task priorities at language level



# CoHMM/Swift



- Concurrency gained primarily by calls to CoMD

- 300 lines of sequential C
- Coordinates multiple sequential calls to CoMD
- We rewrote this in Swift

- 1000's lines of sequential C
- Simplified MD simulator
- Typically called as standalone program
- We exposed CoMD as a Swift function – no `exec()`



# CoMD: Library access from Swift

- **CoMD binding: (example-1)**

```
string s = "-f data/8k.inp.gz";
int N = 3;
foreach i in [0:N-1] {
    float virial_stress = COMDSWIFT_runSim(s);
    printf("Swift: virial_stress: %e",
           virial_stress);
}
```



# CoMD: Library access from CoHMM

```
C  
#define ZERO_TEMP_COMD "../..../CoMD/CoMD -x 6 -y 6 -z 6"  
#ifdef ZERO_TEMP_COMD  
// open pipe to CoMD  
FILE *fPipe = popen(ZERO_TEMP_COMD, "r");  
if (fPipe == NULL) {  
    ...  
}
```

## **Swift**

```
#define ZERO_TEMP_COMD "../..../CoMD/CoMD -x 6 -y 6 -z 6"  
#ifdef ZERO_TEMP_COMD  
    string command = ZERO_TEMP_COMD;  
    stressXX = COMDSWIFT_runSim(command);  
#else  
    // Just the derivative of the zero temp energy wrt A  
    stressXX = rho0*c*c*(A-1);  
#endif
```

# CoHMM: Translation from C to Swift: main()

## C

```
int main(int argc, char **argv) {
    initializedConservedFields();
    for (i = 0; i < 100; i++) {
        for (j = 0; j < 1; j++)
            fullStep();
    }
}
```

## Swift

```
main {
    (A[0], p[0], e[0]) = initializedConservedFields();
    for (int t = 0; t < 5; t = t+1) {
        (A[t+1], p[t+1], e[t+1]) =
            fullStep(A[t], p[t], e[t]);
    }
}
```



# CoHMM: Translation from C to Swift: call CoMD

## C

```
void fluxes(double *A, double *p, double *e,  
            double *f_A, double *f_p, double *f_e) {  
    for (int i = 0; i < L; i++) {  
        double stress = stressFn(A[i], e[i]);  
        double v = p[i] / rho0;  
        f_A[i] = -v;  
        f_p[i] = -stress;  
        f_e[i] = -stress*v;  
    }  
}
```

## Swift

```
(float f_A[], float f_p[], float f_e[])  
fluxes(float A[], float p[], float e[]) {  
    foreach i in [0:L-1] {  
        float stress = stressFn(A[i], e[i]);  
        float v = p[i] / rho0;  
        f_A[i] = -v;  
        f_p[i] = -stress;  
        f_e[i] = -stress*v;  
    }  
}
```

# Summary

- Swift: High-level scripting for outermost programming constructs
- Presented initial work on CoMD
- **Questions?**
  
- **Papers:**
  - **Wozniak et al., Swift/T: Large-scale application composition via distributed-memory dataflow processing. Proc. CCGrid, 2013.**
  - **Wozniak et al., Turbine: A distributed-memory dataflow engine for high performance many-task applications. Fundamentae Informaticae, 2013.**
  - **Wozniak et al., A model for tracing and debugging large-scale task-parallel programs with MPE. Proc. LASH-C at PPOPP (extended abstract), 2013.**
  - **Wozniak et al., JETS: Language and system support for many-parallel-task applications. J. Grid Computing, 2013.**
  - **Armstrong et al., ExM: High level dataflow programming for extreme-scale systems. Proc. HotPar (extended abstract) 2012.**
  - **Wilde et al., Swift: A language for distributed parallel scripting. J. Parallel Computing, 2011.**

