Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications

Justin M Wozniak, Timothy G. Armstrong, Ketan Maheshwari,
Ewing L. Lusk, Daniel S. Katz, Michael Wilde, and Ian T. Foster

Argonne National Laboratory and the University of Chicago

Presented at:

SWEET
Scottsdale, AZ – May 20, 2012

Outline

- Scientific applications
  - Batches, ensembles, parameter studies,
  - Scientific scripting tools to construct studies
- Performance challenges
- Dataflow computing
- Translation techniques
- Performance results
- Summary

Parameter studies

- Treat each application invocation as a function evaluation in a higher-level method
- Run the same application with varying input parameters
  - Parameter sweep: cover a known range of inputs to obtain outputs and produce statistical information or visualization
  - Parameter search/optimization: find inputs that produce interesting/extreme outputs
  - Application script: evaluate arbitrary user script
- Many scientific applications can be expressed at a high level as relatively simple, iterative sweeps of inputs to an function

Scientific scripting – Swift background

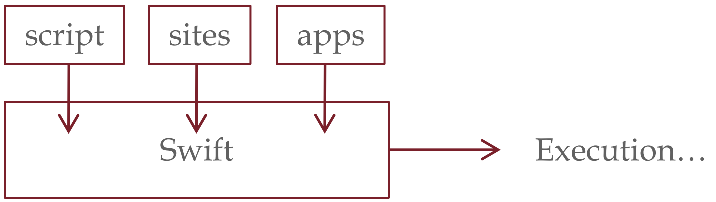
- Original Swift/Karajan implementation was designed for the grid

- Supported file/task model directly in the language

```
app (file output) sim(file input) {
   namd2 @input @output
}
```

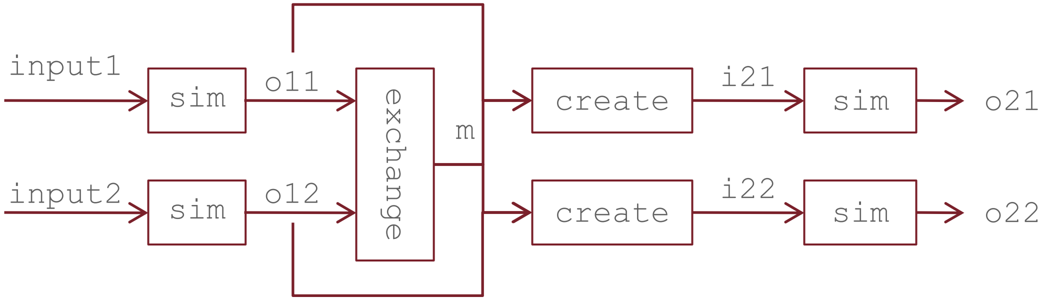
- Provide natural concurrency through automatic data flow analysis and task scheduling

- Separated application script from site configuration details



- Supported scientific data sets in the language through language constructs such as structs, arrays, mappers, etc.

```
file o11 = sim(input1);
file o12 = sim(input2);
file m   = exchange(o11, o12);
file i21 = create(o11, m);
file o21 = sim(i21);
...
```



Swift/Karajan architecture

- Tasks may be generated by a simple list or by a running program or workflow
- Workflow execution produces "job specifications"- user tasks to be executed on the available infrastructure

Task generation (workflow) → Task distribution → Task execution

Task generation (workflow)	Task distribution	Task execution
Submit site (1 node)	Service site (1 node)	Compute sites (1000's of nodes)
~500 tasks/s	~3000 tasks/s (Falkon)	Rate dependent on I/O

- Swift/Turbine moves the task generation and distribution workload to the scalable infrastructure

Performance challenges for large batches

- Evaluation of dataflow program is expensive
  - Complex data structures are constructed to maintain program state
  - Each task is represented in memory (typically bound to single node)
- For small application run times, the cost of application start-up, small I/O, library searches, etc. is expensive
- Existing HPC schedulers do not support this mode of operation
  - Difficult to use traditional scripting languages
  - Traditional scripting languages do not represent large external concurrency anyway (Cf. PyDFlow)
- Solution pursued by Turbine:
  - Allocate Turbine processes *en masse*
  - Use a specialized user scheduler (ADLB) to rapidly submit user work to agents
  - Process the dataflow program as an ADLB application

Swift/Turbine architecture

- Launch the whole thing as a big MPI program
- Tasks may be generated by a simple list or by a running program or workflow
- Workflow execution produces “leaf functions”- user tasks to be executed on the available infrastructure in the form of C/C++ function calls

Task generation (Turbine)	→	Task distribution (ADLB)	→	Task execution (ADLB)
Engine processes (1000’s)		Server processes (1000’s)		Worker processes (100,000’s)
~1000 tasks/s per engine		~10,000 tasks/s per server		very fast

- Swift/Turbine moves the task generation and distribution workload to the scalable infrastructure

Performance target

Performance requirements for distributing the work of Swift-like task generation for an ADLB-like task distributor on an example exascale system:

- Need to utilize $O(10^6)$ concurrency
- For batch of 1000 tasks per core
  - 10 seconds per task
  - 2 hour, 46 minute batch
- Tasks: $O(10^9)$
- Tasks/s: $O(10^5)$
- Divide cores into *workers* and *control* cores
  - Allocate 0.1% as control cores, $O(10^3)$
  - Each control core must produce $O(100)$ tasks/second

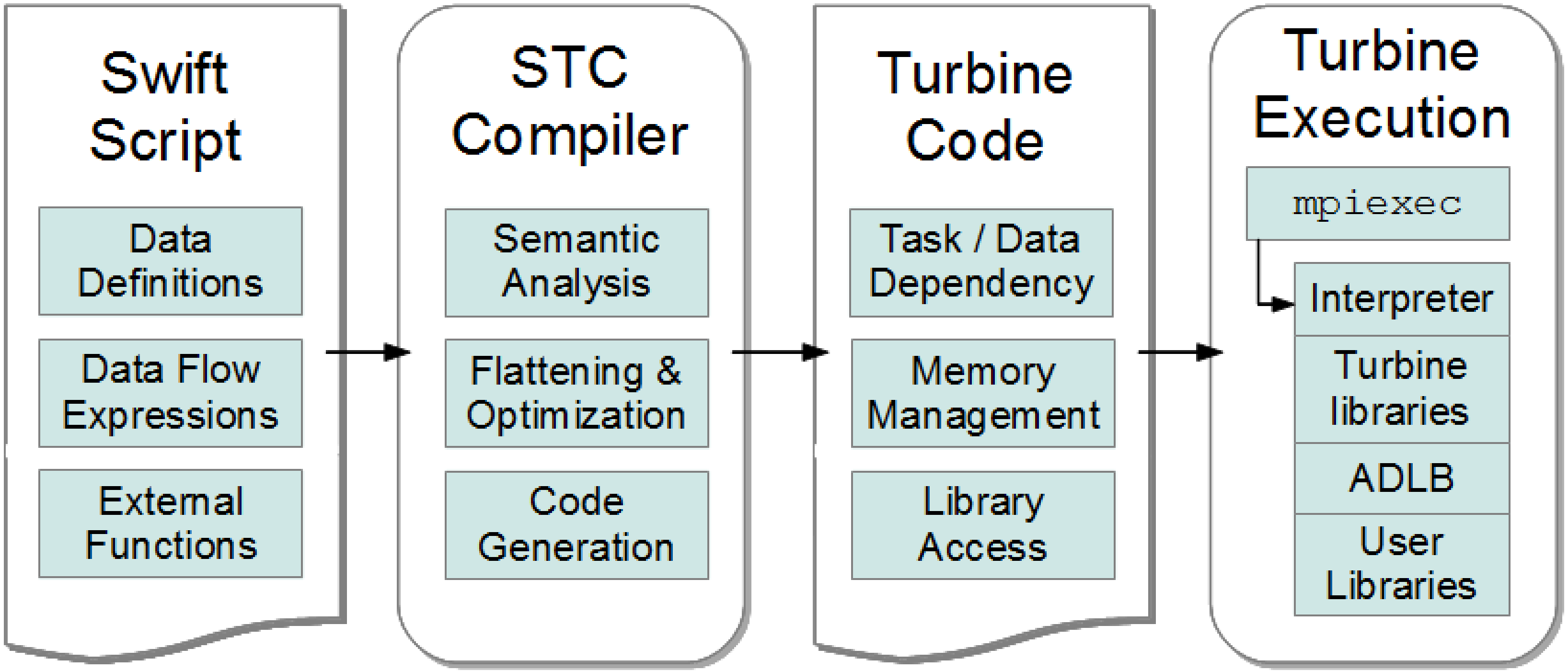
Turbine: High level design features

- Provide a simple compiler target for Swift scripts
  - Natural representation of data-dependent functions
  - Emphasis on calls to external functions
  - Represent script variables, data structures

- Enable fast dataflow processing
  - Data-driven execution actions
  - Subscribe/notify model on any script variable
  - Load balance everything with ADLB

- Integrate with ADLB
  - **A**synchronous **D**ynamic **L**oad **B**alancer: an MPI library
    - Distributes discrete work units to participating processes
    - Provides advanced features: work types, priorities, location-specific tasks
    - Turbine implementation started with a Tcl extension for ADLB
    - ADLB known to scale to 128,000 processes on IBM Blue Gene/P
  - Turbine evaluates a data flow program in distributed memory using ADLB primitives

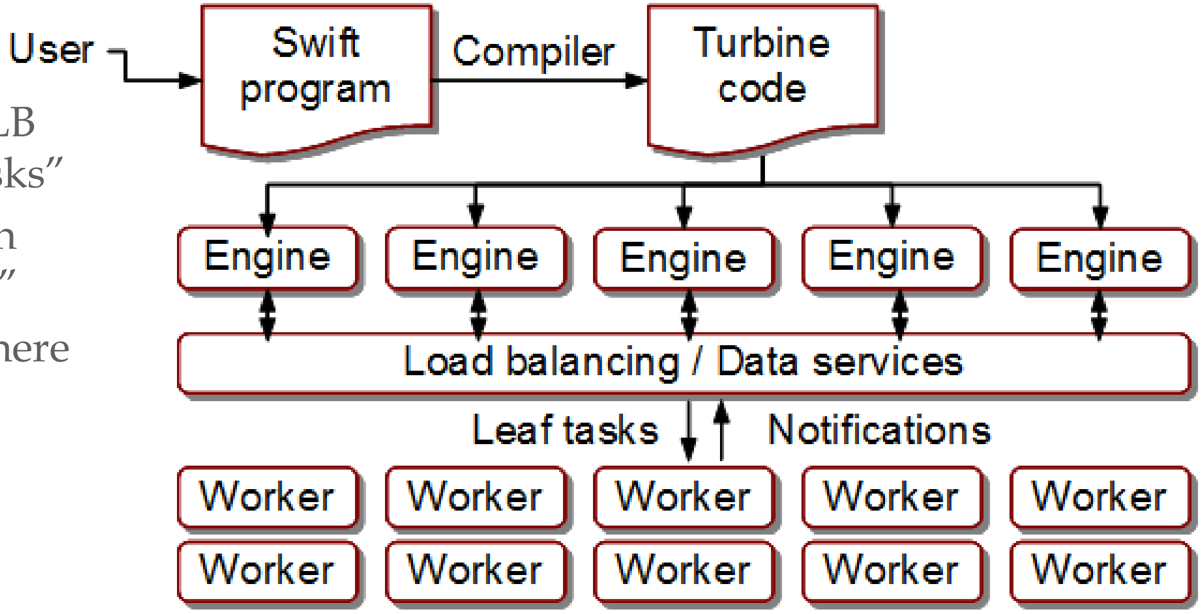
Turbine: User interaction



- Typical compile/run interface
  - Compiler is highly portable, Turbine code is not machine-specific
  - Runs on x86 clusters, SiCortex, Cray XE6, Blue Gene/P, etc.

Turbine: Architecture

- User starts by developing Swift script
  - Script may be run on any system with any MPI process management settings: number of processes, process distribution, etc.
  - User specifies number of engines, servers, etc. at run time
  - Dataflow engines communicate using ADLB work units – "control tasks"
  - Leaf functions execute on workers – "worker tasks"
  - Tasks can execute anywhere because data is globally accessible



Turbine: Program evaluation

- Swift is a naturally concurrent, functional language
  - Syntactically looks like C, Java, etc.
  - Consists of composite functions and leaf functions
    - Leaf functions are external programs / function calls to C/C++
    - Composite functions evaluate Swift code

- "Fundamental Theorem of Swift/Turbine"
  - For generic Swift function call (multiply-valued):
    `(y1, y2) = f(x1, x2, x3);`
  - Turbine:
    - Creates a record for statement – a "rule"
    - Subscribes to `x1`, `x2`, `x3`
    - When notified, call `f()`
      - If `f()` is composite function, load balance body of `f()` as control task
        - » `f()` is evaluated on an available engine, resulting in more rules
      - If `f()` is leaf function, load balance `f()` as worker task
    - Store outputs, resulting in notifications
  - This works for all Swift expressions and control constructs
  - Compiler may need to generate additional composite functions for constructs

Turbine: Distributed Future Store

- The distributed-memory data-driven progress model represents a scalable, globally-accessible future store
  - Future: "An object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete" (Wikipedia)
  - Turbine implements futures in distributed memory

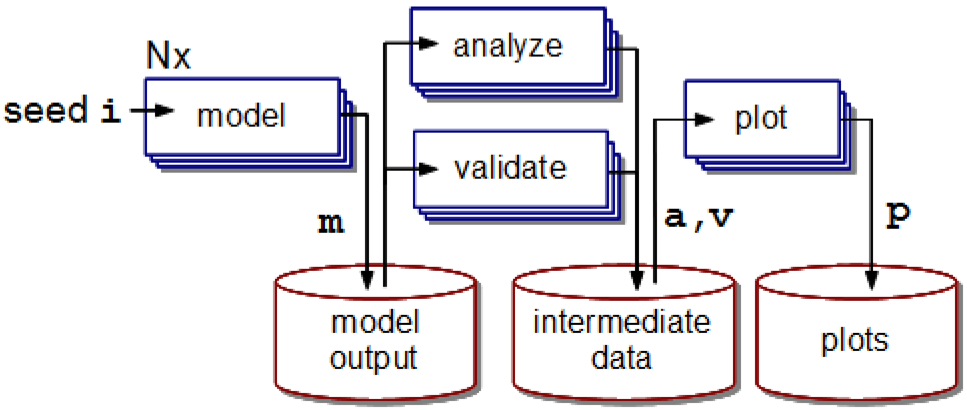
- Fast dataflow processing
  - Pending actions are indexed and stored in minimal memory
  - Notification is handled elegantly by ADLB tasks

- Data services were patched into ADLB servers
  - Script variables identified by 64-bit integers – Turbine data - TDs
  - Typed: integers, floats, strings as atomic, write-once units
  - Containers: FS-like links
    - Container TD + subscript ⇨ Member TD
    - Allows for arrays, etc.
  - Generic ADLB data API could conceivably be used directly by ADLB applications
  - API includes, create, store, retrieve, subscribe, insert, etc.

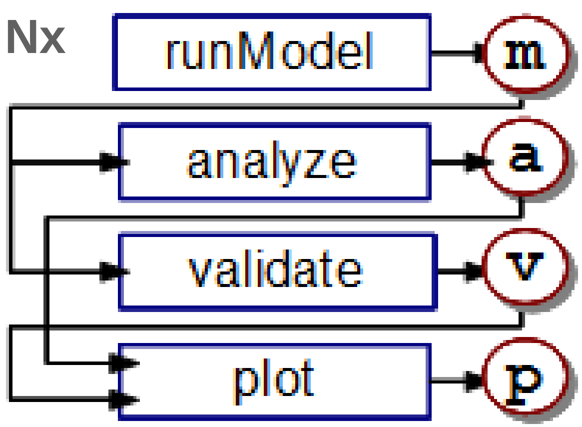
Simple data flow example

```
Model m[];
Analysis a[];
Validity v[];
Plot p[];
int n;
foreach i in [0:n-1] {
  // run model with random seed
  m[i] = runModel(i);
  a[i] = analyze(m[i]);
  v[i] = validate(m[i]);
  p[i] = plot(a[i], v[i]);
}
```

Application concept:



Turbine engine records:



Turbine example: Arithmetic

Swift

```
int i = 3, j = 4, k;
k = i + j;
trace(k);
```

Turbine

```
allocate i integer 3
allocate j integer 4
allocate k integer
call_builtin plus_integer [ $k ] [ $i $j ]
call_builtin trace [ ] [ $k ]
```

- Dataflow processing enables typical arithmetic, etc.
- `plus_integer` is just a Turbine wrapper around Tcl's **+**
- Variables are in distributed memory: accessible by distributed tasks

Turbine example: Conditional program flow

Swift

```
c = extractStatistic(a);
if (c) {
  trace("Warning: c is non-zero");
}
```

Turbine

```
... # open code
  call_app extractStatistic [ $c ] [ $a ]
  statement [ $c ] " if-1 $c"
}

proc if-1 { c } {
    set v:c [ retrieve_integer $c ]
    if (v:c) {
        allocate s string "Warning: c is non-zero"
        call_builtin trace [ ] [ $s ]
    }
}
```

- Compiler generates data-dependent function for if block body
- Body is dependent on condition value
- Body could conceivably execute anywhere

Turbine example: Composite functions

Swift

```
(int f) fib(int n) {
  if (n > 2)
  f = fib(n-1) + fib(n-2);
  ...
}
```

Turbine

```
proc fib { n f } {
    ...
    call_builtin minus_integer [ $t1 ] [ $n $t0 ]
    # fib(n-1)
    call_composite fib [ $t2 ] [ $t1 ]
    ...
    # fib(n-2)
    call_composite fib ...
    call_builtin plus_integer [ $f ] [ $t2 ... ]
    ...
}
```

- Example omits Turbine conditional
- Recursive calls are submitted to ADLB for load balancing

- `fib()` scales to at least 64K processes on the BG/P (Armstrong, 2012)

Turbine example: Data structures

Swift

```
(int a[][]) eye2() {
  a[0][0] = 1;
  a[0][1] = 0;
  a[1][0] = 0;
  a[1][1] = 1;
}
```

Turbine

```
proc eye2 { a } {
    allocate_container a
    allocate_container t1
    allocate_container t2
    allocate i0 integer 0
    allocate i1 integer 1
    container_insert_imm $a 0 $t1
    container_insert_imm $t1 0 $i0
    container_insert_imm $t1 1 $i1
    ...
}
```

- Turbine containers can implement Swift's complex data structures
- Assignment into an array is a link, not a copy
- Data-dependent container operations were necessary to implement Swift semantics

Turbine example: Loops

Swift

```
int b[];
foreach i, v in a {
  b[i] = f(a[i]);
}
```

- Loop body is implemented as compiler-generated function
- Loop variables are real Turbine data

Turbine

```
    allocate_container b
    loop a [ a ] loop_1
}

# inputs: loop counter, loop variable and additionals
proc loop_1 { i v a b } {
    set t1 [ container_lookup_imm $a $i ]
    allocate t2 integer
    call_composite f [ $t2 ] [ $t1 ]
    container_insert_imm $b $i $t2
}
```

Performance: Goals

- Underlying services:
  - How fast can ADLB distribute tasks?
  - How fast can we access variables in distributed memory?

- Turbine:
  - How fast can we generate a large data structure of futures?
  - How fast can we traverse a large data structure of futures?

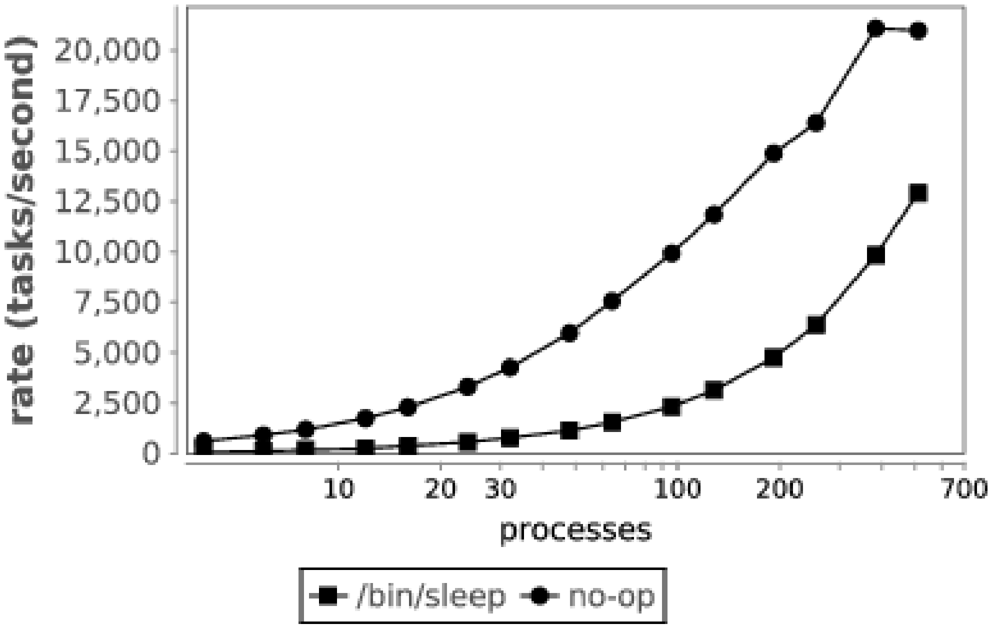
- Only measure engine dataflow-related operations: ignore the effect of generated user work
  - Attempt to generate task rates sufficient to utilize >100,000 workers

- All results obtained on the SiCortex
  - 6-core nodes at 633 MHz, 4 GB RAM
  - 1 μs latency
  - Somewhat obsolete, but useful for these benchmarks

Performance: Raw ADLB operations

- ADLB configured with single server
- No Turbine features
- ADLB application

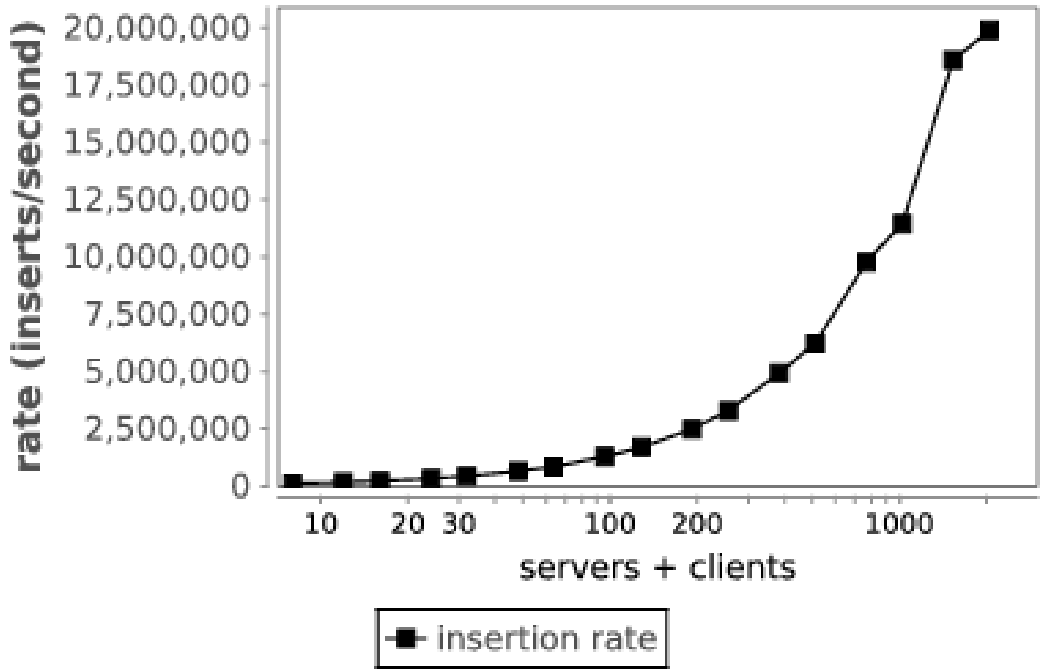
```
if { $rank == 0 } {
    set batchfile [ lindex $argv 0 ]
    set fd [ open $batchfile r ]
    while { true } {
        gets $fd line
        adlb::put $line
        if { [ eof $fd ] } { break }
    }
}
while { true } {
        set work [ adlb::get ]
        if { [ string length $work ] } {
            eval exec $work
        } else { break }
}
```



- Single server maxes out at just over 20,000 tasks/s

Performance: Raw data operations

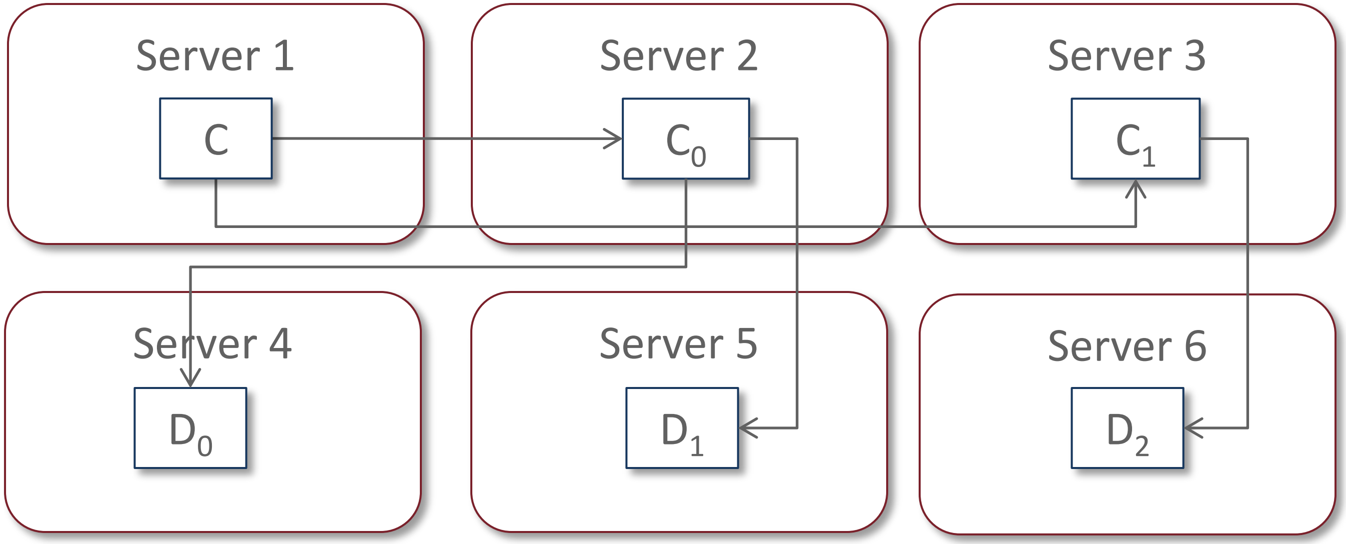
- ADLB configured with servers == clients
- No Turbine features
- ADLB application




```
if { [adlb::amserver] } {
    adlb::server
} else {
    set r [ expr $mpi_rank + 1 ]
    for { set id $r } { $id <= $count } { incr id $mpi_size } {
        adlb::create $id $adlb::STRING
        adlb::store  $id $adlb::STRING "data"
    }
}
```

Turbine: Distributed data structures

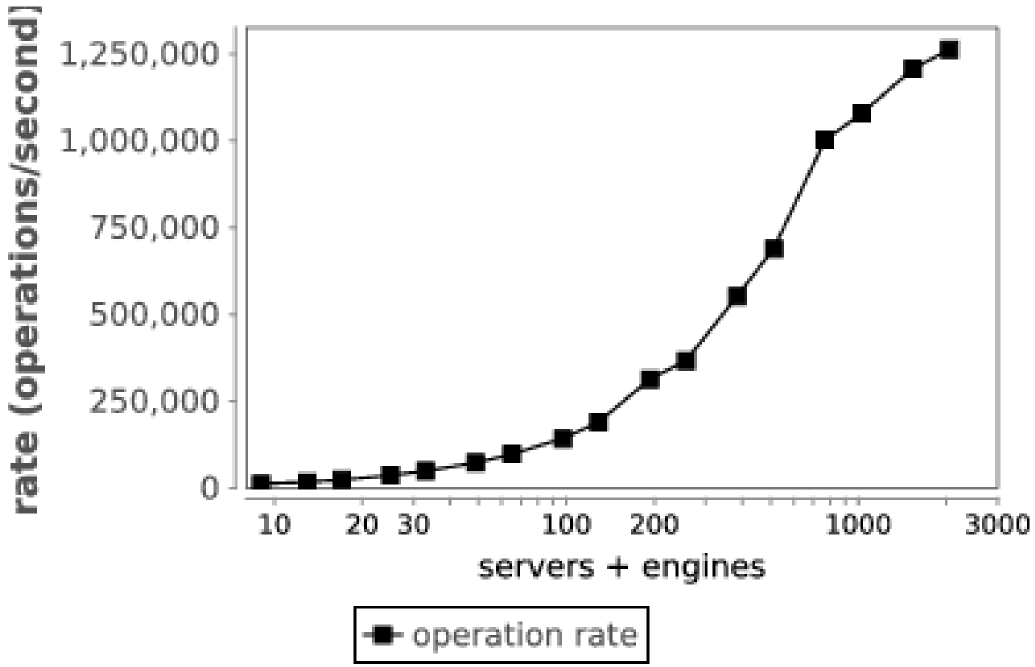
- Need to access large containers- do not want a single container to become a bottleneck
- Use a container-of-containers approach



Performance: Distributed range creation

- Swift:
  ```
  int A[] = [1:100*1000*1000];
  ```

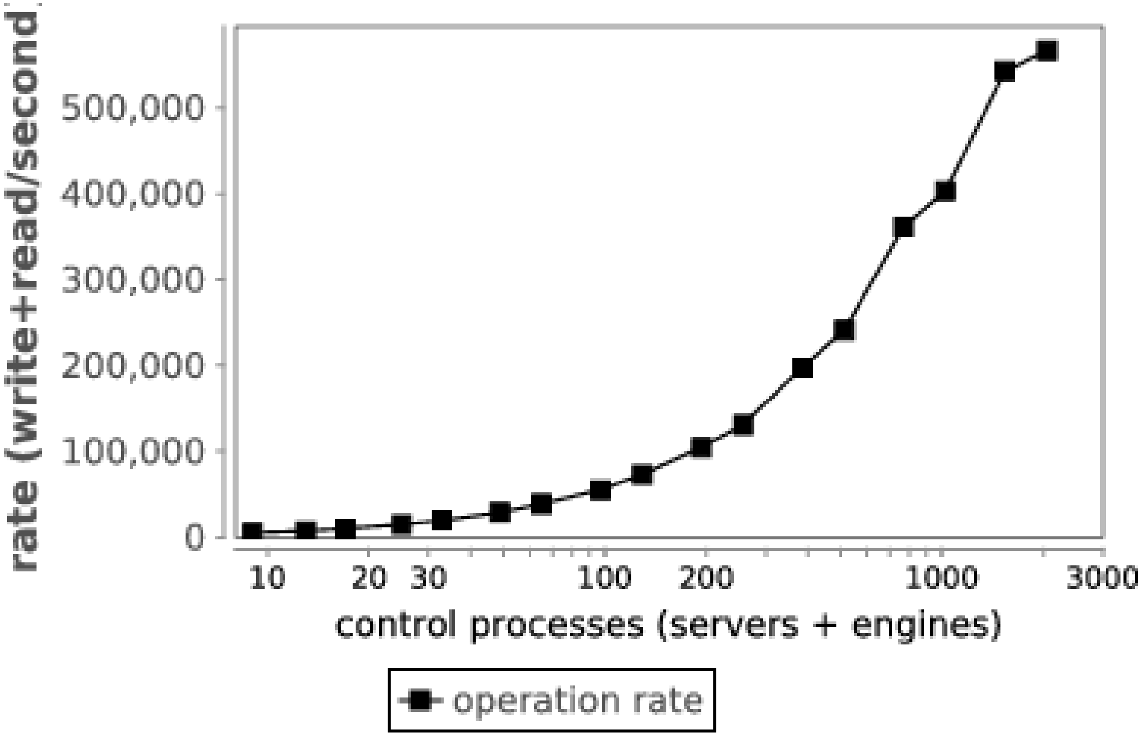
- Turbine creates the containers automatically



Performance: Distributed loop iteration

- Swift:

```
int A[] = [1:100*1000*1000];
for i in A {
    i;
 }
```



Musings: Threads vs. Rule engine

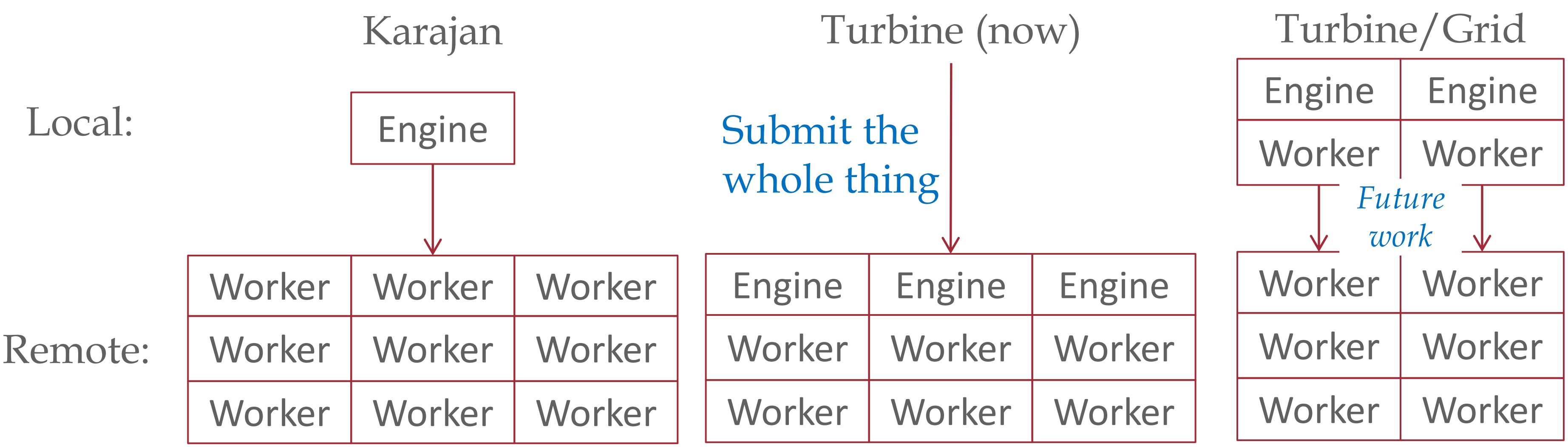
- Turbine engines are single-threaded
- We do not use a thread abstraction

- Typical approach with futures is to spawn many threads, then just block the threads on the futures
  - Requires lightweight threading mechanism
  - Karajan provides this nicely
  - Swift/Karajan used this with success, but constrained to single node
  - Memory is a constraint (Stratan, 2008)

- Swift semantics do not require a full threaded model
  - Function calls are referentially transparent – do not need stack
  - Turbine rule engine chains data dependencies to actions with low overhead
  - Nice for distributed memory

Musings: Can Turbine replace Karajan?

- Karajan enables the use of all the CoG providers:
  - Globus, PBS, SGE, Cobalt, SSH, staging, GridFTP, etc.
- Turbine can spawn external processes on its workers but would need significant work to plug into these remote execution techniques
- Would enable highly scalable dataflow processing for the grid



ExM solution

Scalable many-task execution (Swift/Turbine)
Scalable cache filesystem (MosaStore)

- We are currently deploying MosaStore file system services on the Blue Gene/P compute nodes
- This will allow external application programs to interact with file data without disk congestion

Recap and further reading…

Task generation → Task distribution → Task execution

Task generation	Task distribution	Task execution
Swift, Turbine	Coasters, ADLB, JETS	Collective Data Management

- **Case studies in storage access by loosely coupled petascale applications**
  Petascale Data Storage Workshop at SC'09
- **JETS: Language and system support for many-parallel-task computing**
  Proc. Workshop on Parallel Programming Models and Systems Software for High-End Computing at ICPP, 2011.
- **A workflow-aware storage system: An opportunity study**
  Proc. CCGrid, 2012.
- **ExM: High level dataflow programming for extreme-scale systems**
  Proc. HotPar (short paper in poster series), 2012.

Thanks

- Thanks to the organizers

- Grants:

This research is supported by the DOE Office of Science, Advanced Scientific Computing Research X-Stack program, under contracts DE-AC02-06CH11357 FWP 57810 and DE-FC02-06ER25777.

Questions