Chapel on ARM

Brian Guarraci
Twitter, Inc

Chapel on ARM

- **Getting Chapel running on ARM**
- Building a Chapel app and evaluating it
- Chapel as a Service Oriented Architecture

Why ARM?

- Because you can?
  - Cheap and low-power (~5W)
  - GPU support
- ARM boards are the new commodity server
- True parallelism on your desktop

Chapel on ARM

- Generally “just-works”
- A couple config tweaks:
  - export CHPL_TASKS=fifo
  - export CHPL_TARGET_ARCH=native
  - GASNET + UDP
- Simple deployment and cluster operation

Goal

- Build a practically useful system with Chapel that's efficient enough to make use of ARM systems, but could scale to traditional hardware
    - 1-2GB RAM
    - Gigabit network
    - Storage
        - Remote via HDFS / NAS
        - Local Storage via SSD
- Viable as an online service (e.g. website infrastructure)

Chapel on ARM

- Getting Chapel running on ARM
- **Building a Chapel app and evaluating**
- Chapel as a Service Oriented Architecture

Application Goal

- Simple distributed inverted-index search engine
  - Replicate basic behavior of Twitter search engine
  - Exercise ARM cluster with millions of Tweets
- Can index new content while serving reads
- Basic boolean expression query language
- Easy to deploy on a cluster

Lessons

- Idea: Use classes to represent data structures (aka Java bad habits)
- Problem:
  - Chapel feels optimized for a top-level data structures and class instantiations are heavy weight by comparison
  - Heap management incurs overhead, esp. remote allocations
  - Nesting classes has some issues
- Solution:
  - Minimize classes when top-level data structures will do
  - Use Chapel's module system like Objective-C's categories

Lessons

- Idea: Use an associative array with values allocated on hash-partitioned locales
- Problems:
  - Too naïve, though incredibly simple if it worked
  - Slow due constantly sync’ing hash-table across locales
  - Threading issues: domains are thread-safe, but underlying arrays are not
- Solutions:
  - Use ReplicatedDist to create locale “singletons”
  - Use local keyword to catch non-local memory access

Lessons

- Idea: Use a per-locale hash-table with string keys
- Problems:
  - hit string performance issue
  - resizing hash-table is non-trivial without locking
  - pointer operations are not atomic
  - memory allocations are not cheap (aka Java ruins your mind)
  - non-trivial serialization for later use as a memory mapped file
- Solutions:
  - use atomic integers to hold index pointers into object pools
  - incrementally allocate memory for document id posting lists

Lessons

- Idea: Load index data from local file into each locale
- Problems:
  - Locales appear to be assigned to machines non-deterministically
  - Boiler plate begin / wait code for doing data loads
- Solution:
  - Use HDFS or other centralized storage
  - Ideally, have a built-in Chapel mechanism for distributed data loads

Lessons

- Idea: scatter-gather queries across locales
- Problems:
  - Assembling results from each locale is slow due to reallocating result array
  - Not optimally serialized requiring final ‘reduce’ phase
- Solution: use Chapel’s parallel iterators

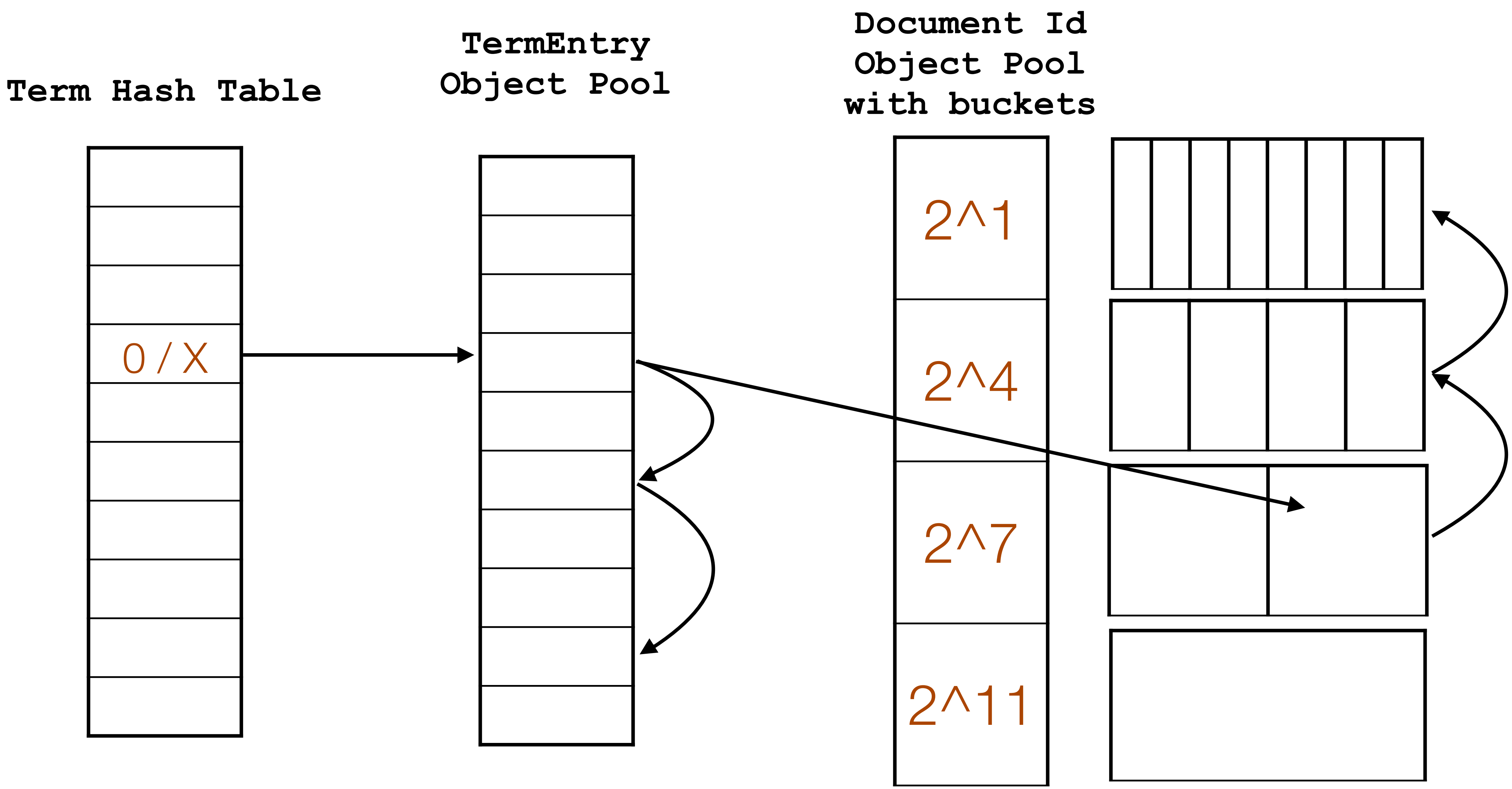
Chearch: Chapel + Search

- Open source at http://chearch.pw (Chearch Pew)
- Lock-free, using integers which are cheap, fast, support atomic operations
- String-free, using integer indices into an external string table
- Boolean queries via CHASM (Chearch Assembly)
    - Stack language that dynamically constructs nested iterator AST
    - AST capable of processing query with minimal memory allocation
- 16M documents (e.g. Tweets) per segment (~1GB / segment)

ARM-specific optimizations

- Use 32-bit integers when possible
- Use appropriately sized memory pools
- Use a single 64-bit integer to represent each intermediate query result
    - no heap to manage
    - bit operations are fast
- Minimize memory footprint by externalizing everything

Inverted Index Reference System



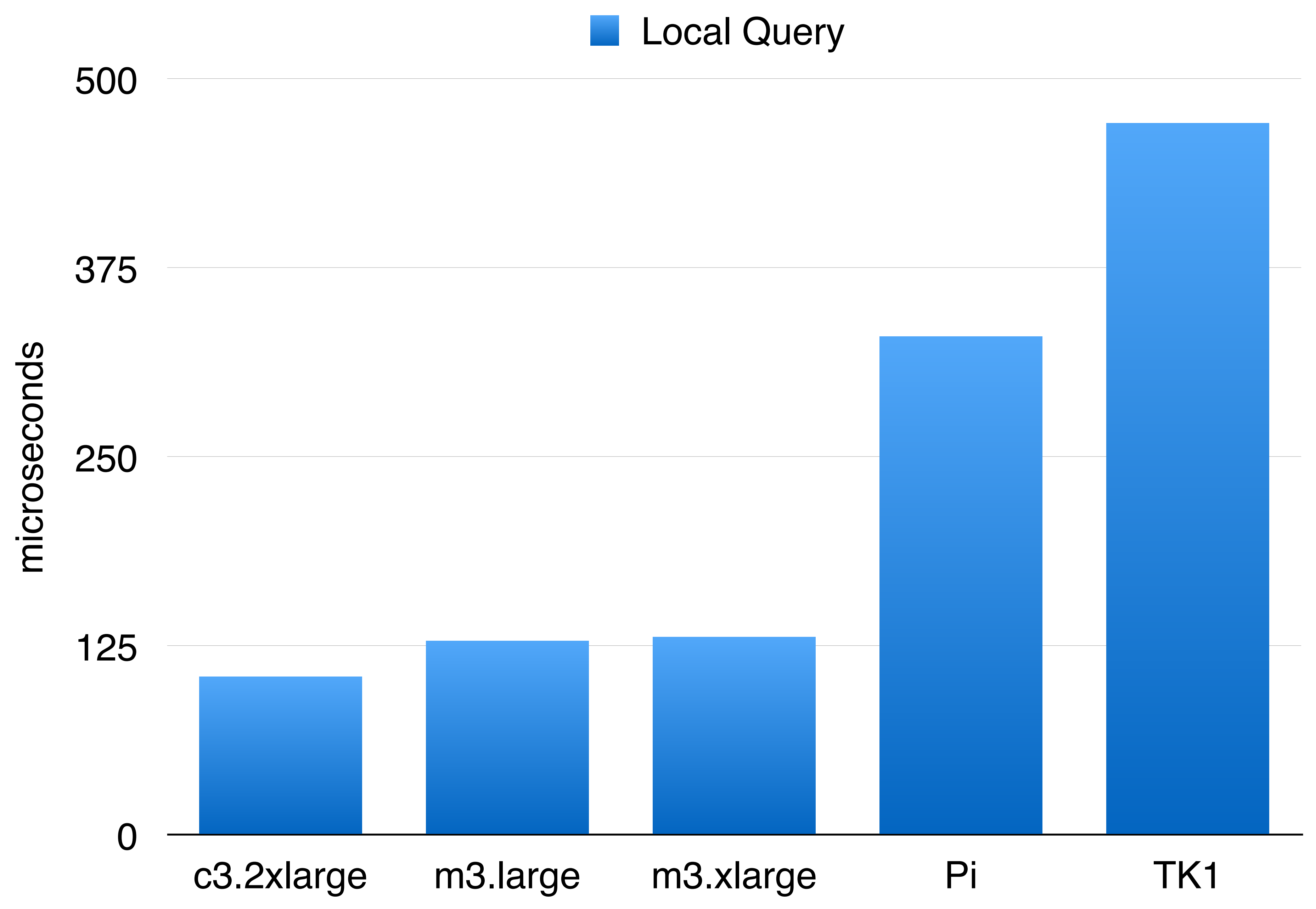
```
var termHashTable: [0..#termHashTableSize] atomic TermEntryPoolIndex;

class TermEntry {
  // reference to external string table
  var term: Term;

  // pointer to the last document id in the doc id pool
  var lastDocIdIndex: atomic DocIdPoolIndex;

  // next term in the bucket chain
  var next: atomic TermEntryPoolIndex;
}
```

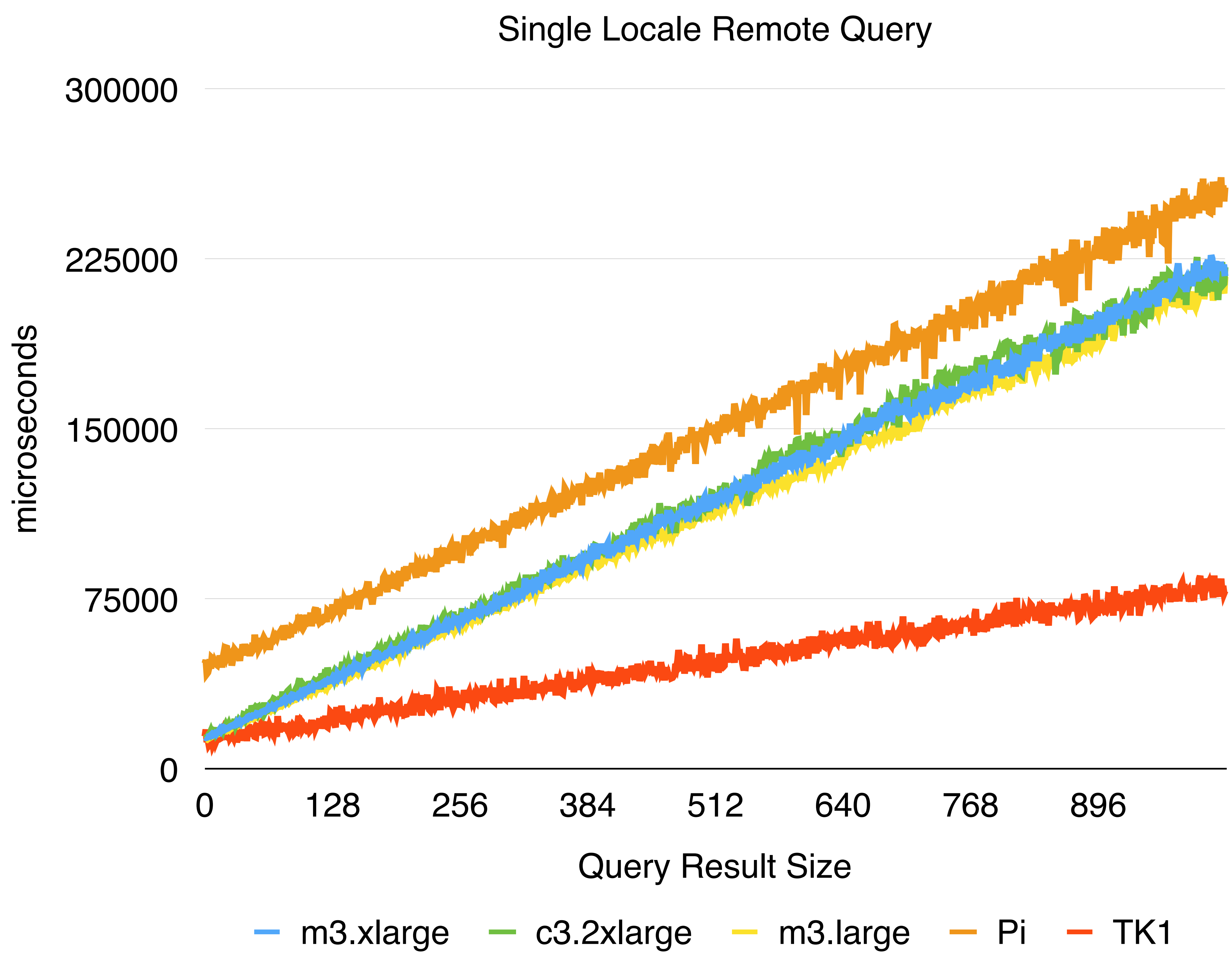
Profiling: Setup

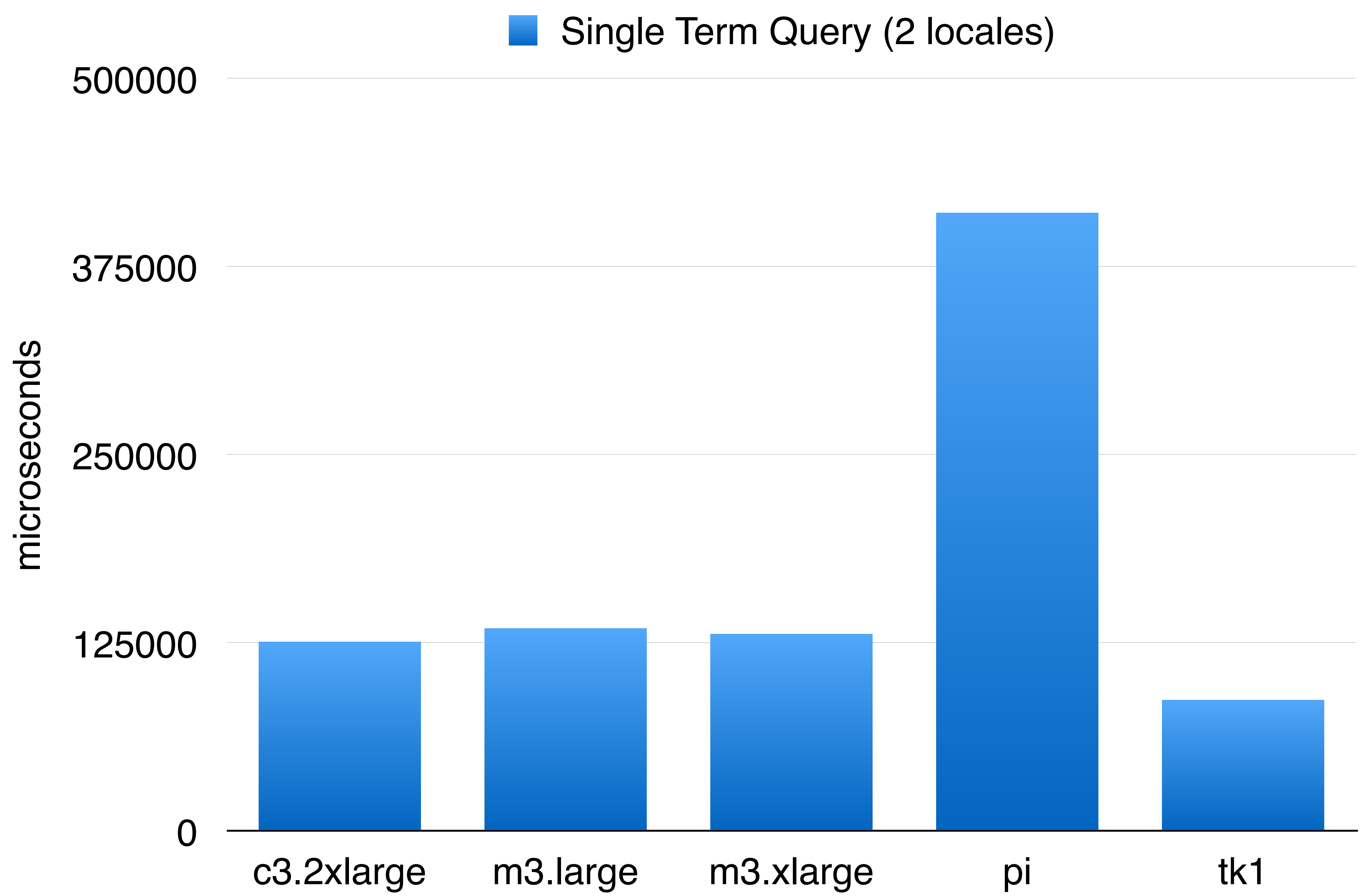
- Synthetic data of 1.5MM Tweets of known distribution across cluster
- Common query patterns that exercise:
  - local queries
  - single remote locale queries
  - scatter-gather queries (AND, OR)

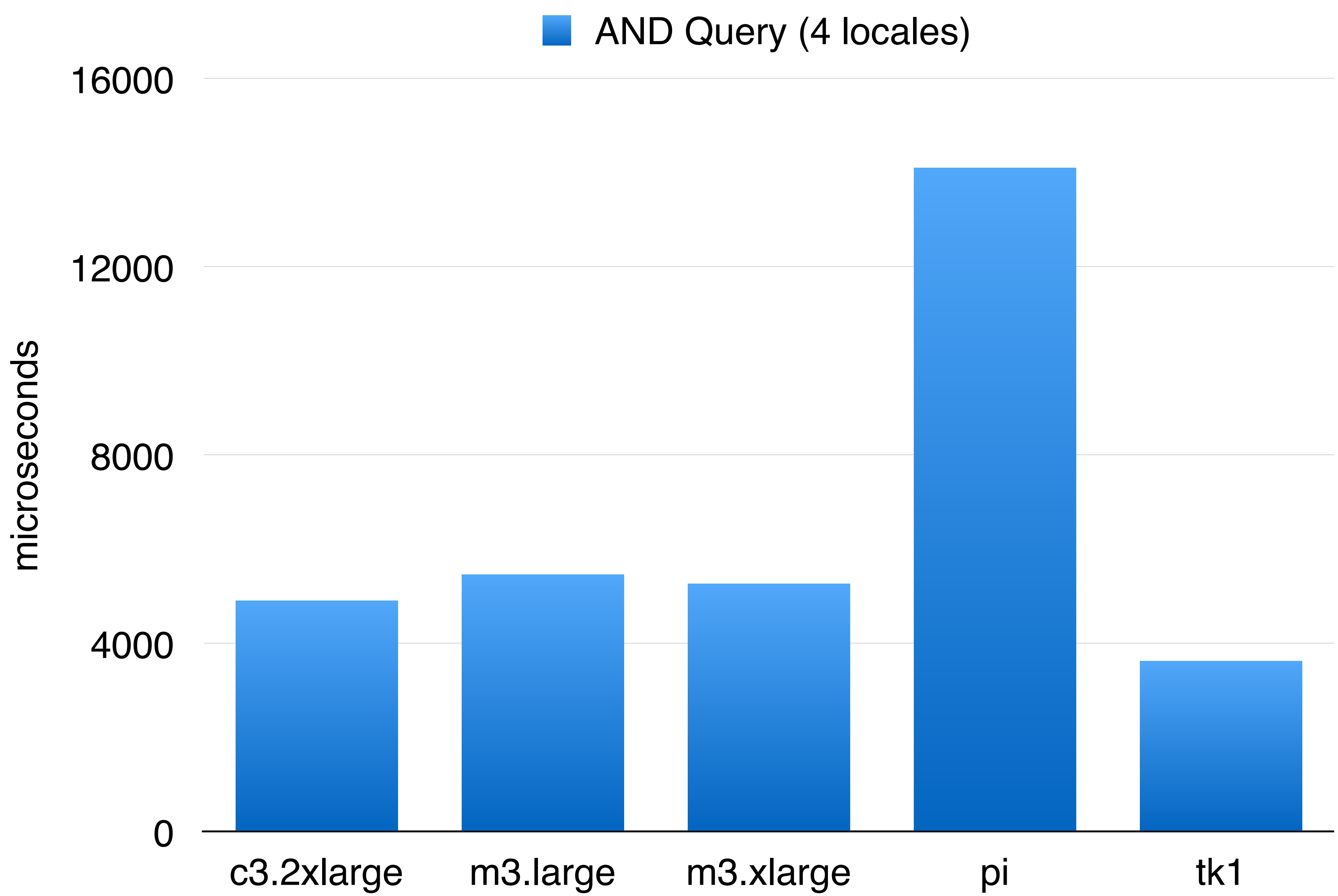
Profiling: Hardware

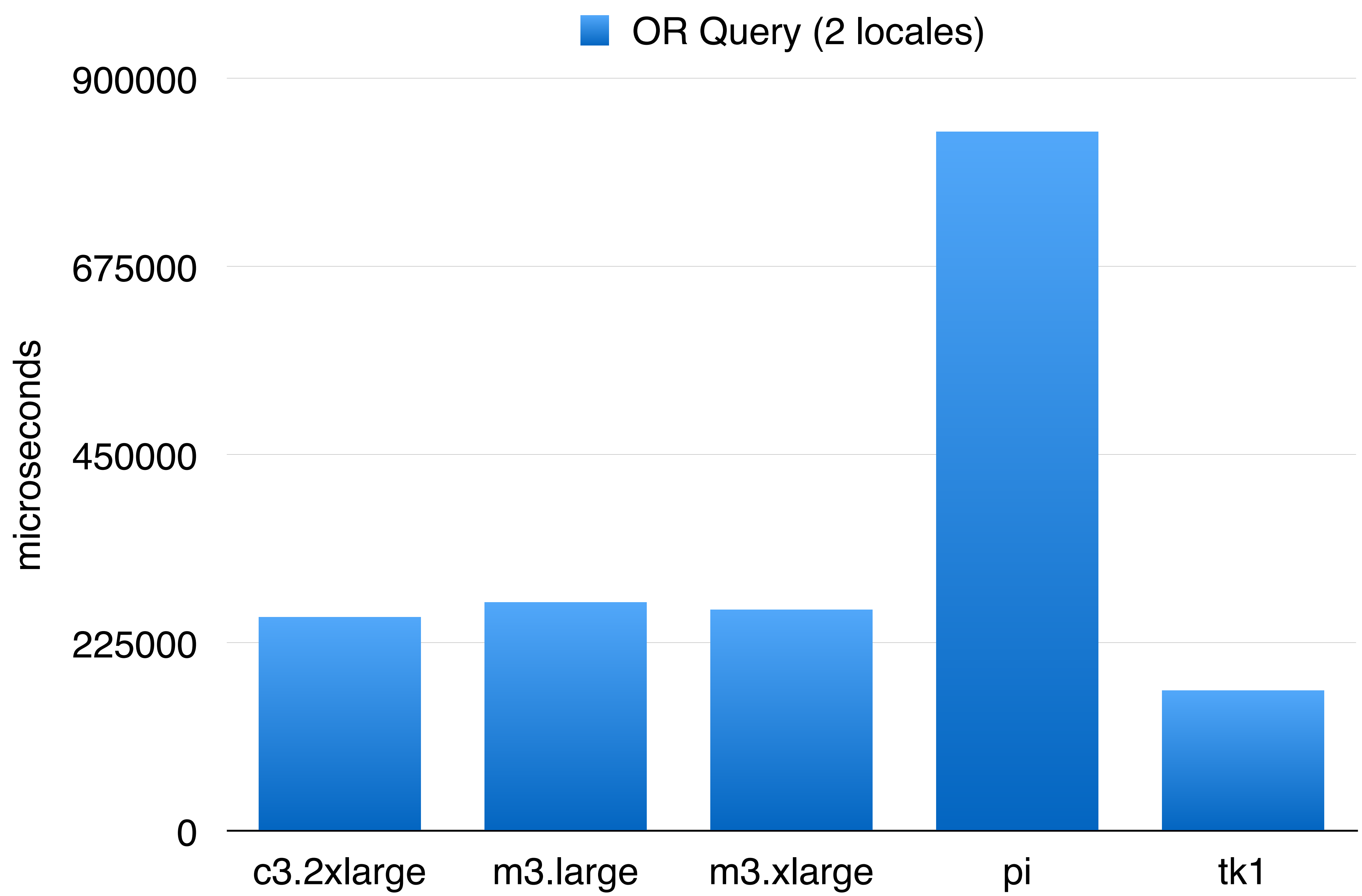
- Raspberry Pi 2 rev B (4 locales)
  - quad-core ARM Cortext-A7 @ 900Mhz
  - 1GB RAM
- Jetson TK1 (16 locales)
  - quad-core ARM Cortex-A15 @ 2Ghz + GPU
  - 2GB RAM
- EC2 m3.large (4 locales)
- EC2 m3.xlarge (8 locales) [high network allocation]
- EC2 c3.2xlarge (4 locales) [compute optimized, high network allocation]


Local Query
500
375
250
125
0
microseconds
c3.2xlarge
m3.large
m3.xlarge
Pi
TK1

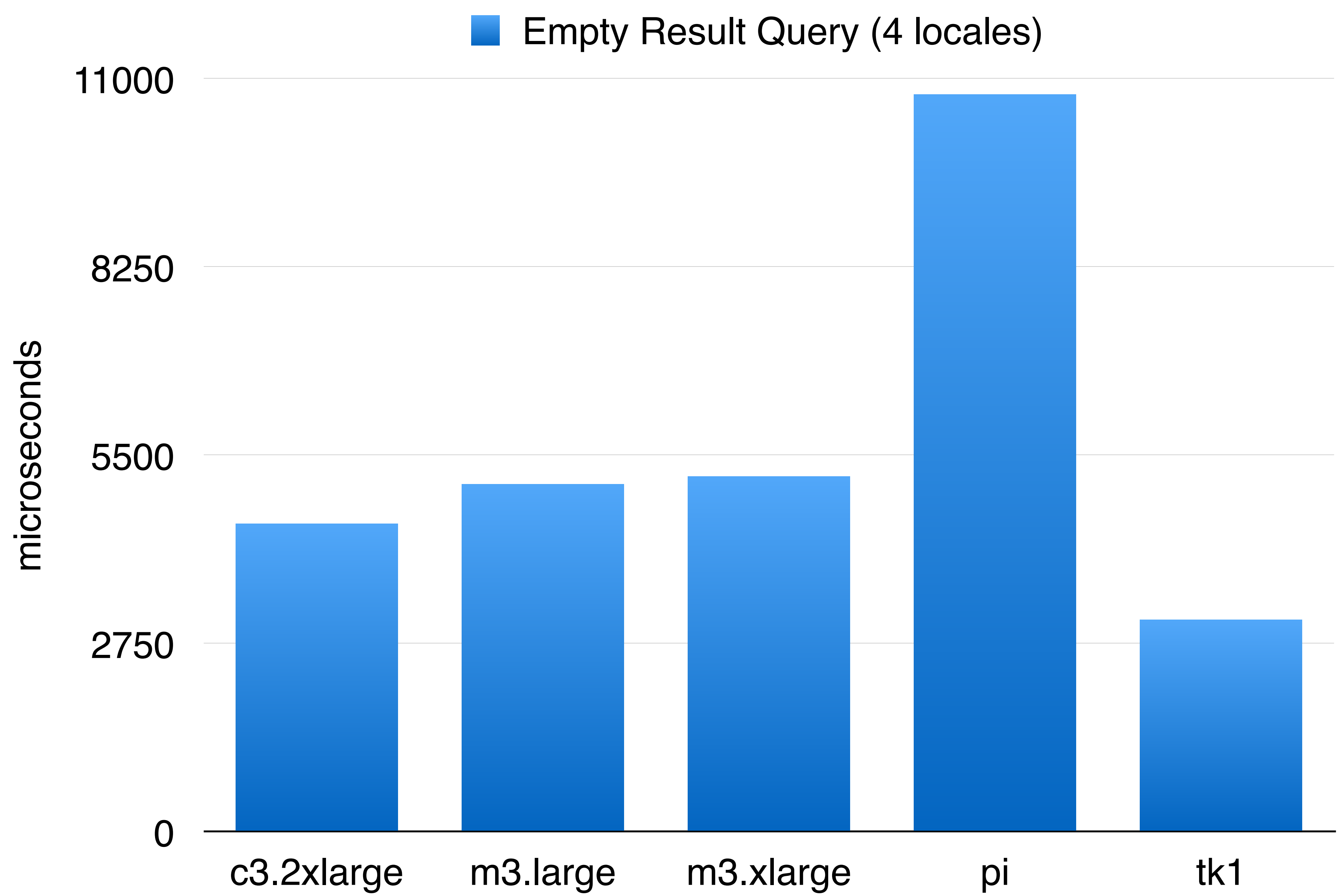

Single Locale Remote Query
300000
225000
150000
75000
0
microseconds
0
128
256
384
512
640
768
896
Query Result Size
m3.xlarge
c3.2xlarge
m3.large
Pi
TK1

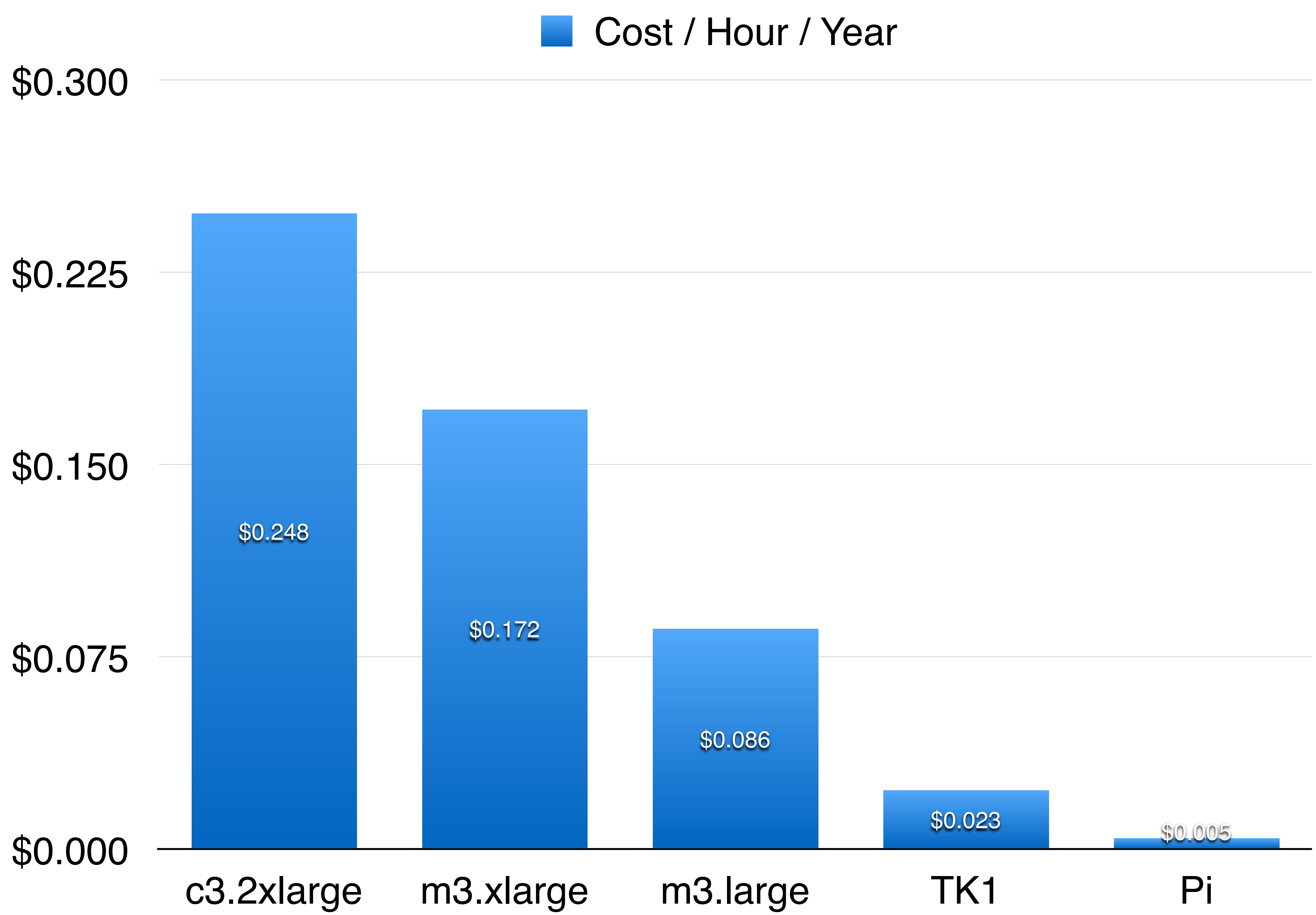

Single Term Query (2 locales)
500000
375000
250000
125000
0
microseconds
c3.2xlarge
m3.large
m3.xlarge
pi
tk1


AND Query (4 locales)
16000
12000
8000
4000
0
microseconds
c3.2xlarge
m3.large
m3.xlarge
pi
tk1



OR Query (2 locales)
900000
675000
450000
225000
0
microseconds
c3.2xlarge
m3.large
m3.xlarge
pi
tk1


Empty Result Query (4 locales)
11000
8250
5500
2750
0
microseconds
c3.2xlarge
m3.large
m3.xlarge
pi
tk1



Cost / Hour / Year
$0.300
$0.225
$0.150
$0.075
$0.000
$0.248
$0.172
$0.086
$0.023
$0.005
c3.2xlarge
m3.xlarge
m3.large
TK1
Pi

Conclusions / Future Work

- Deployment was trivial on different architectures
- ARM systems perform pretty well
- More investigations required
  - High-CPU utilization remains after indexing
  - Random result sizes for parallel queries
  - Parallel query optimization

Chapel on ARM

- Getting Chapel running on ARM
- Building a Chapel app and evaluating it
- **Chapel as a Service Oriented Architecture**

Chapel as a Service

- Chapel app as-a-service
  - Exposes a set of methods that represent the service's functionality
  - Accepts TCP requests that execute these methods
  - Serves multiple simultaneous requests from different clients
- App will be long-running with different, and unknown execution patterns
  - Concurrency safety must be generalized
  - Unpredictable resource utilization now a concern

Locale Wish-list

- Fault tolerance via “virtual” locales
- Constraint satisfaction (degrees of success)
- Explicit support for service abstraction

Fault Tolerance

- Virtual Locales
  - Through config, specify which locales replicate, or compose, a virtual locale
  - Have ability to lose / rejoin a replicant locale
- Rebuild a locale when it rejoins by replicating data from peers
- Chapel application doesn’t die when a locale disappears

Success as a Continuum

- Service-oriented-architectures are constraint-satisfaction-architectures
  - Service clients deal with degrees of success by adapting to response scenarios: timeouts, errors, and partial results
  - Services may be clients to other services themselves, which requires defining success with constraints
- Time is the most common resource constraint
  - Clients can control amount of time willing to wait
  - Servers can add more machines to reduce latency

Success as a Continuum

- Ability to model a function as a dependency directed-graph allows for fine-grain constraint satisfaction

- Each node is a function with a single output and explicit optional and required dependencies

  - if a required dependency can't be satisfied then the dependent node can't be satisfied

  - missing optional dependencies are handled gracefully

  - nodes can be given time budgets and produce partial results

- A service request now means executing the graph

Idea: Class-Locale Binding

- Virtual locale provides exclusive servicing of one or more classes
    - Any request to the class is served by the bound locale
    - Topology is understood and static as defined through a Class-Locale model prior to deploy
- Using virtual locales and constraint satisfaction, Class-Locales can be reliable and tolerate fluctuations in available resources
- Possible extension of current ability to create a class instance on a locale, but may need runtime support for network error handling

Conclusion

- Chapel works well on ARM, making it easy to utilize low-end clusters
- Language features of Chapel are very powerful, but can have a steep learning curve
- Chapel locale support is similar to a Service without needing to build explicit plumbing
- Adding support for fault-tolerance would make it more useful in transient environments