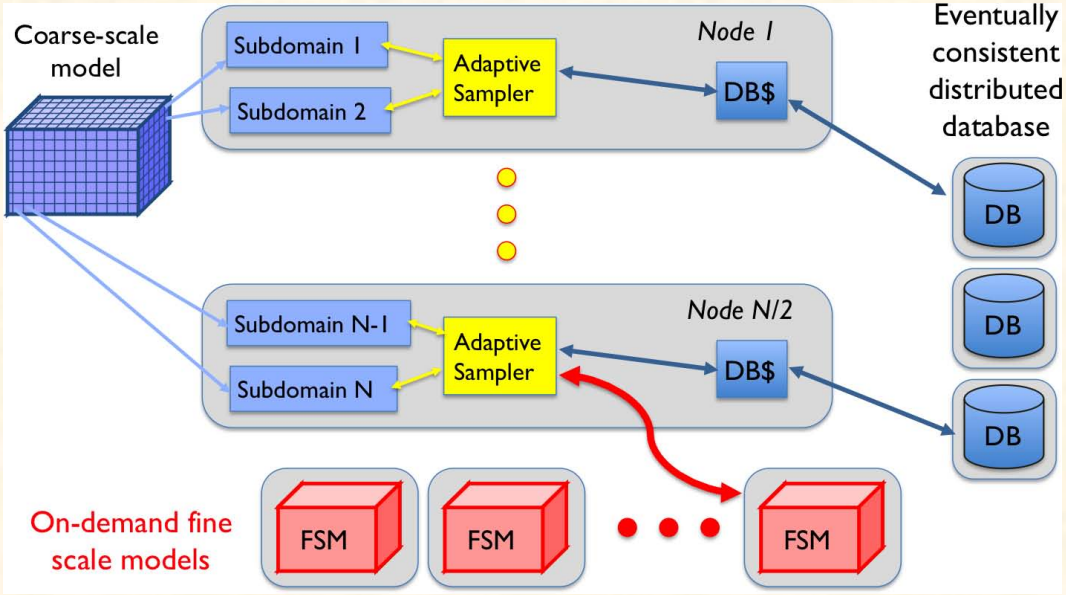
ExMatEx Application "Workflow"

Stanford RAMCloud Visit
April 20, 2015

Allen McPherson & Christoph Junghans
Los Alamos National Laboratory

Brief Introduction to ExMatEx

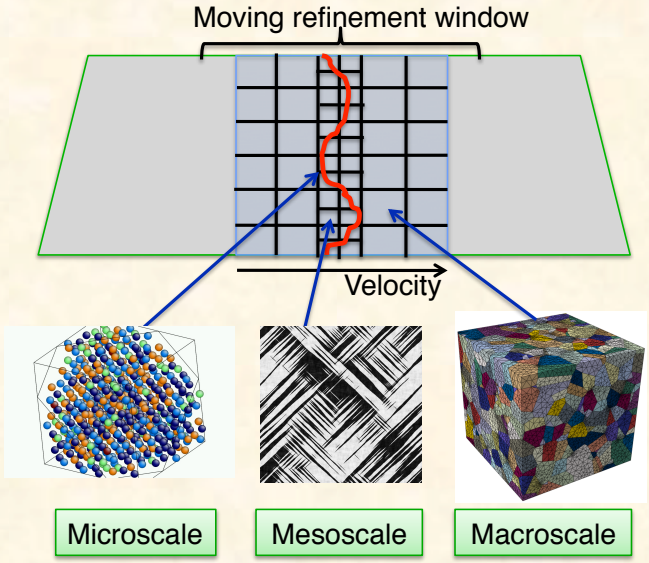
- Multi-scale materials
- Application driven
- Computer science focused
- ASCR Co-Design Center
  - *LANL, LLNL, SNL, ORNL*
  - *Stanford, Caltech*
  - *$4M/yr for 5 yrs*
    - » starting third year



- Work in many areas: molecular dynamics, proxies, programming models, DSLs, multi-scale algorithms, vendor interface, runtimes, software stacks, etc.
- More info at http://exmatex.org
- Code: https://github.com/exmatex

ExMatEx application "workflow"

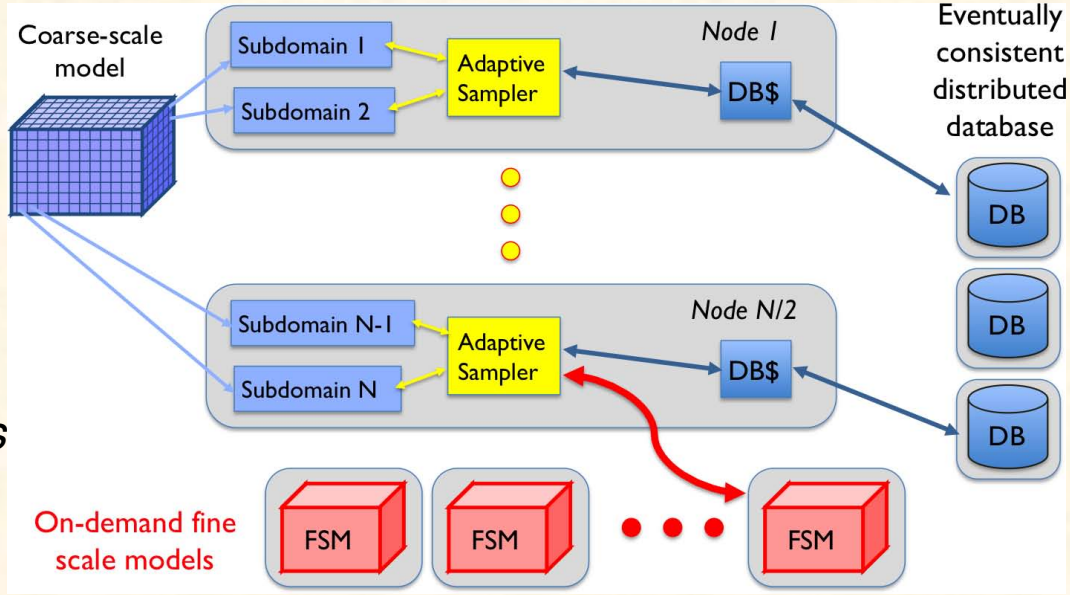
- ExMatEx apps are…
  - *Multi-scale*
  - *Dynamic*
- Multi-scale applications integrate components…
  - *That dynamically interact with each other on-the-fly*
- Components can be…
  - *Serial (single core)*
  - *Single node*
    - » Multi-core
    - » Accelerated (e.g. GPU)
  - *Multi-node*
    - » Groups of the above
    - » Existing libraries



- Additional requirements: fault tolerance, in situ analysis, etc.

ExMatEx application orchestration

- 2 components today
  - *Tasking*
  - *Databases*
- Dynamic tasking
  - *Hierarchical multi-scale*
  - *Independent computations*
  - *Uphill dependencies*
  - *Variable granularity*
  - *Application still logically synchronous at time-step boundary*
- Databases (use largely driven by cloud/web influence)
  - *Use to accelerate overall computation (sub-scale calls dominate)*
  - *Simple key/value stores*
  - *60-200 doubles down, dozen doubles up*
  - *Obviously, queries must be much faster than fine scale calls*



Tasking/Scheduling Requirements

- Fairly simple execution graph (tree)
- Dynamic scheduling (with load balancing)
- Semantically: asynchronous function calls
- Task granularity (eventually, prototype less demanding)
  - *Sub-scale calls Xms*
  - *Database queries Xus*
- Nutshell:
  - *Hierarchically schedule tasks*
  - *Keep the machine load balanced*
  - *Communicate data between components*
  - *Manage data locality*
- We've experimented with Charm++, CnC, OCR, libcircle, Swift, Erlang, cloud (NodeJS, ZooKeeper, etc.), others

Tasking/Scheduling Implementation

- Monolithic
  - *Languages*
    - » Charm++
    - » Chapel
    - » Etc.
  - *"Runtimes"*
    - » Legion
    - » Uintah, etc.
- Advantages
  - *Soup to nuts*
- Disadvantages
  - *Buy in required (tied to programming model?)*
  - *Flexibility (fixed/hidden models)*

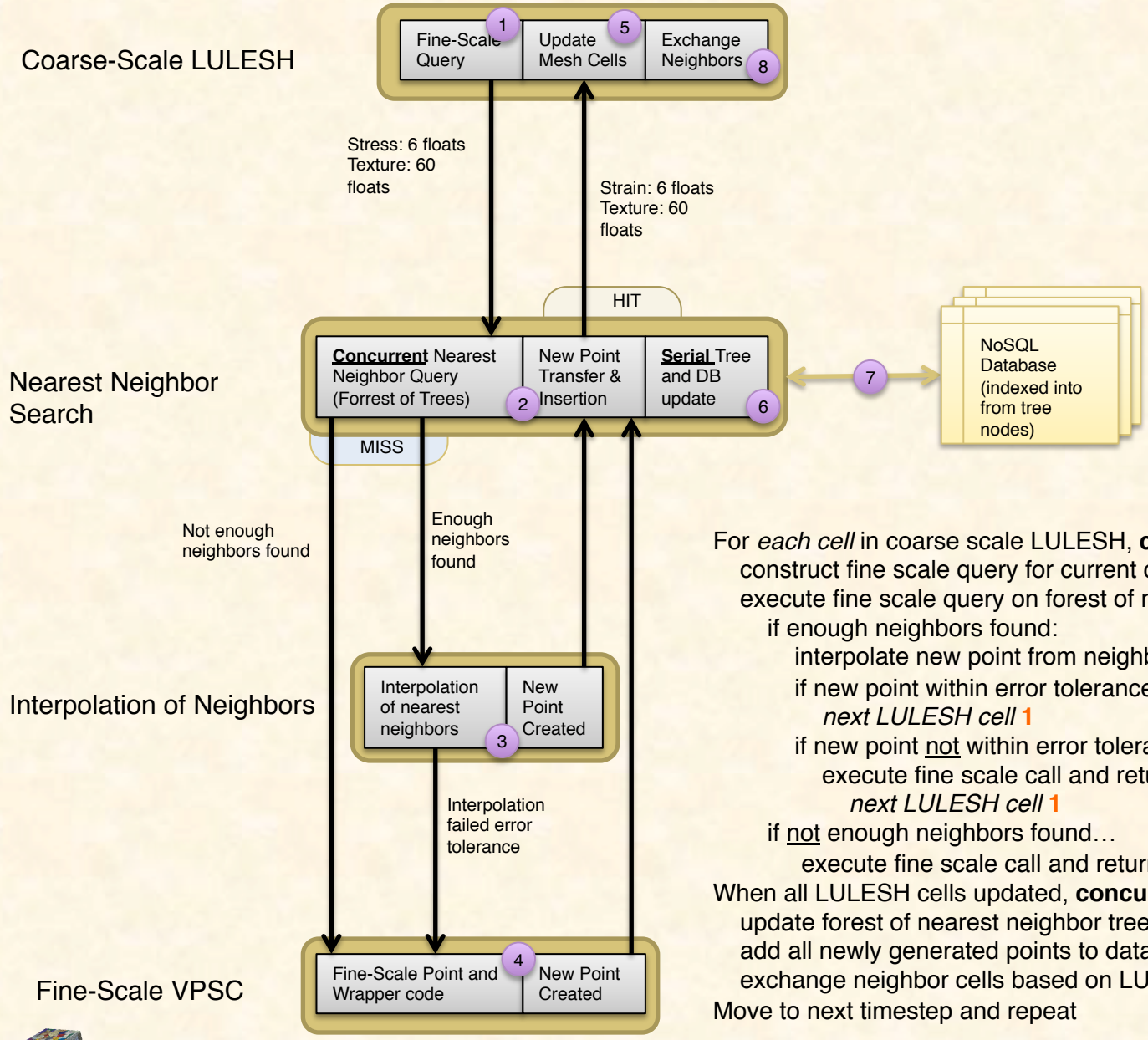
- Service-based
  - *Orthogonal, single-service*
    - » Scheduling
    - » Messaging
    - » Caching
    - » Load balancing
    - » Etc.
  - *Common (or close to) APIs*
  - *Pluggable*
  - *Web/cloud model*
- Advantages
  - *Flexibility, modularity*
- Disadvantages
  - *Performance? Important?*

Database Requirements

- Really a “caching” service
  - *Cache previous sub-scale calls*
  - *Typically in-memory*
  - *May want persistence too (subsequent runs of similar problem)*
- Key/value API to start (additional functionality/APIs possible)
- Queries are tasks like any other task (same requirements)
- Transactional not required (i.e. eventual consistency OK)
  - *Transactional writes may be required in the future*
- Beyond caching sub-scale calls
  - *Fault tolerance*
  - *EOS*
  - *Material properties*
  - *Many others*

Database Implementation

- Roll your own infeasible (given resources)
- Tons of work out there in web/cloud community (optimize for HPC)
- Simple APIs
- Ideally…
  - *Distributed*
  - *Fault-tolerant (replicated)*
  - *Dynamically balanced*
  - *NVRAM-capable, persistent*
- All this currently available from web/cloud community, but…
  - *Requires TCP/IP stack*
    - » Largely unavailable on our big platforms
    - » Necessitates “custom” distribution strategies (for ExMatEx)
    - » Apply engineering dollars to add Infiniband?
    - » RAMCloud (Stanford a possible alternative)

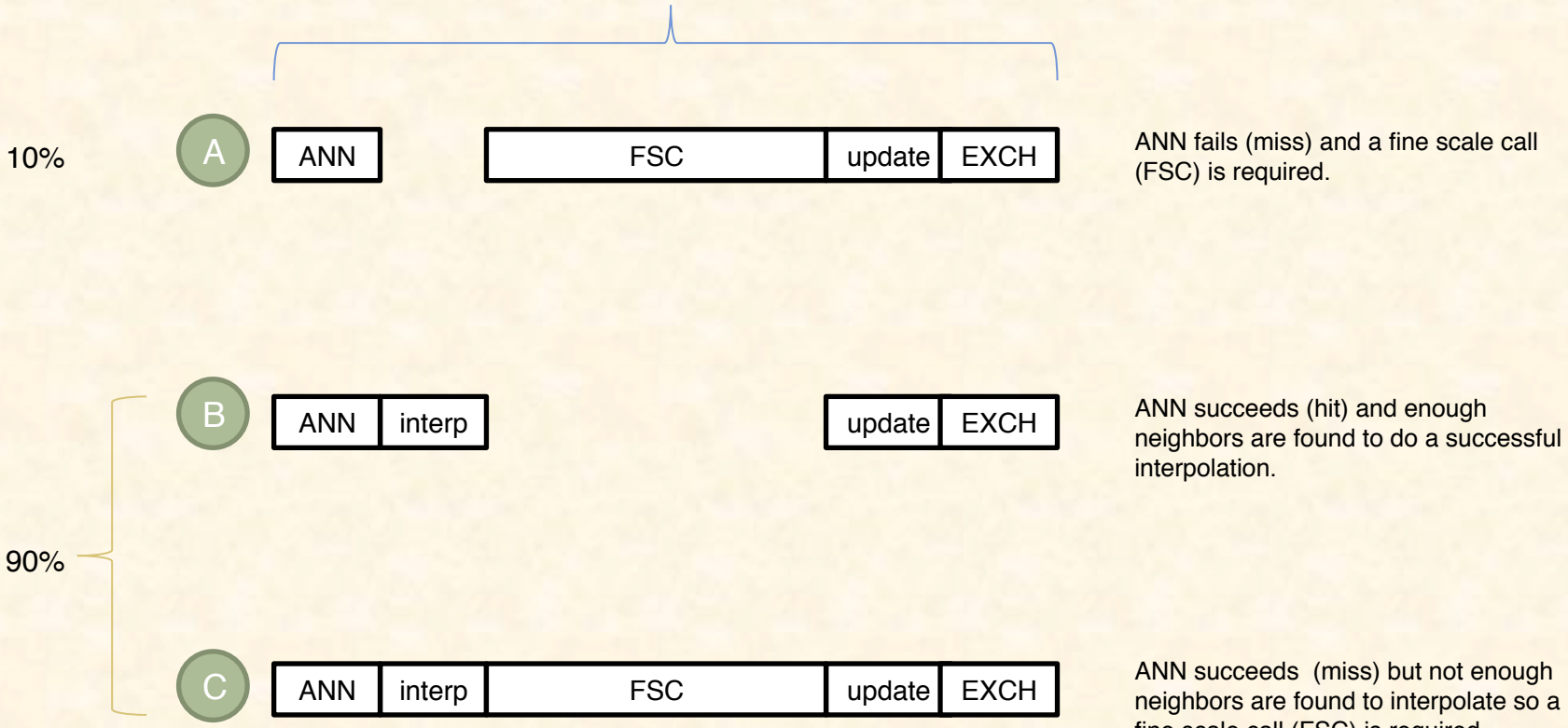


For *each cell* in coarse scale LULESH, **concurrently**…
- construct fine scale query for current cell 1
- execute fine scale query on forest of nearest neighbor trees… 2
  - if enough neighbors found:
    - interpolate new point from neighbors 3
    - if new point within error tolerance: update current mesh cell 5
      - *next LULESH cell* 1
    - if new point not within error tolerance
      - execute fine scale call and return new point 4
        - *next LULESH cell* 1
  - if not enough neighbors found…
    - execute fine scale call and return new point 4; *next LULESH cell* 1

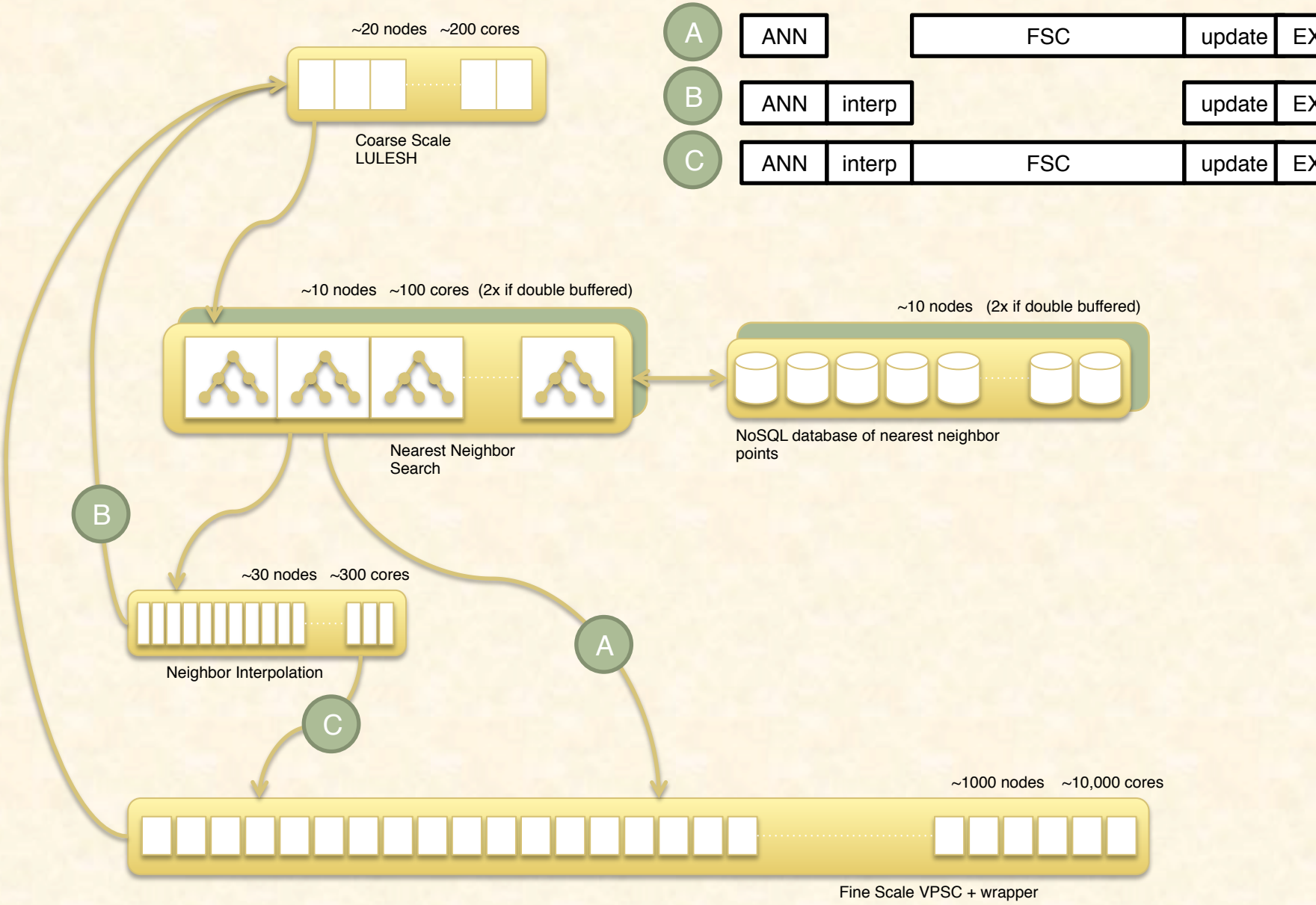
When all LULESH cells updated, **concurrently**…
- update forest of nearest neighbor trees with all new points 6
- add all newly generated points to database 7
- exchange neighbor cells based on LULESH decomposition 8

Move to next timestep and repeat

For each cell in the coarse scale LULESH computation, one of the three task chains (shown below) are executed. The tasks in the chains are executed sequentially but each chain can be executed concurrently on a per-cell basis. Each task chain must run to completion within the 10 second time step budget.



ANN: approximate nearest neighbor search
FSC: fine scale call (VPSC)
interp: interpolation of nearest neighbors
update: insert new point into trees and database
EXCH: LULESH neighbor exchange


~20 nodes ~200 cores
Coarse Scale LULESH
A
ANN
FSC
update
EXCH
B
ANN
interp
update
EXCH
C
ANN
interp
FSC
update
EXCH
~10 nodes ~100 cores (2x if double buffered)
Nearest Neighbor Search
~10 nodes (2x if double buffered)
NoSQL database of nearest neighbor points
B
~30 nodes ~300 cores
Neighbor Interpolation
A
C
~1000 nodes ~10,000 cores
Fine Scale VPSC + wrapper