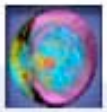
SciDAC

Scientific Discovery through Advanced Computing





- PETASCALE DATA STORAGE INSTITUTE
  - 3 universities, 5 labs, G. Gibson, CMU, PI
- SciDAC @ Petascale storage issues
  - www.pdsi-scidac.org
  - Community building: ie. PDSW-SC07 (Sun 11th)
  - APIs & standards: ie., Parallel NFS, POSIX
  - Failure data collection, analysis: ie., CFDR
  - Performance trace collection & benchmark publication
  - IT automation applied to HEC systems & problems
  - Novel mechanisms for core (esp. metadata, wide area)

Carnegie Mellon

center for information technology integration

UNIVERSITY OF MICHIGAN

Los Alamos NATIONAL LABORATORY EST. 1943

Sandia National Laboratories

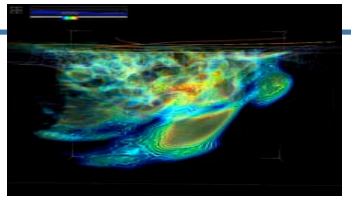
Pacific Northwest National Laboratory
Operated by Battelle for the U.S. Department of Energy

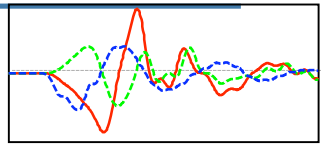
Carnegie Mellon
Parallel Data Laboratory

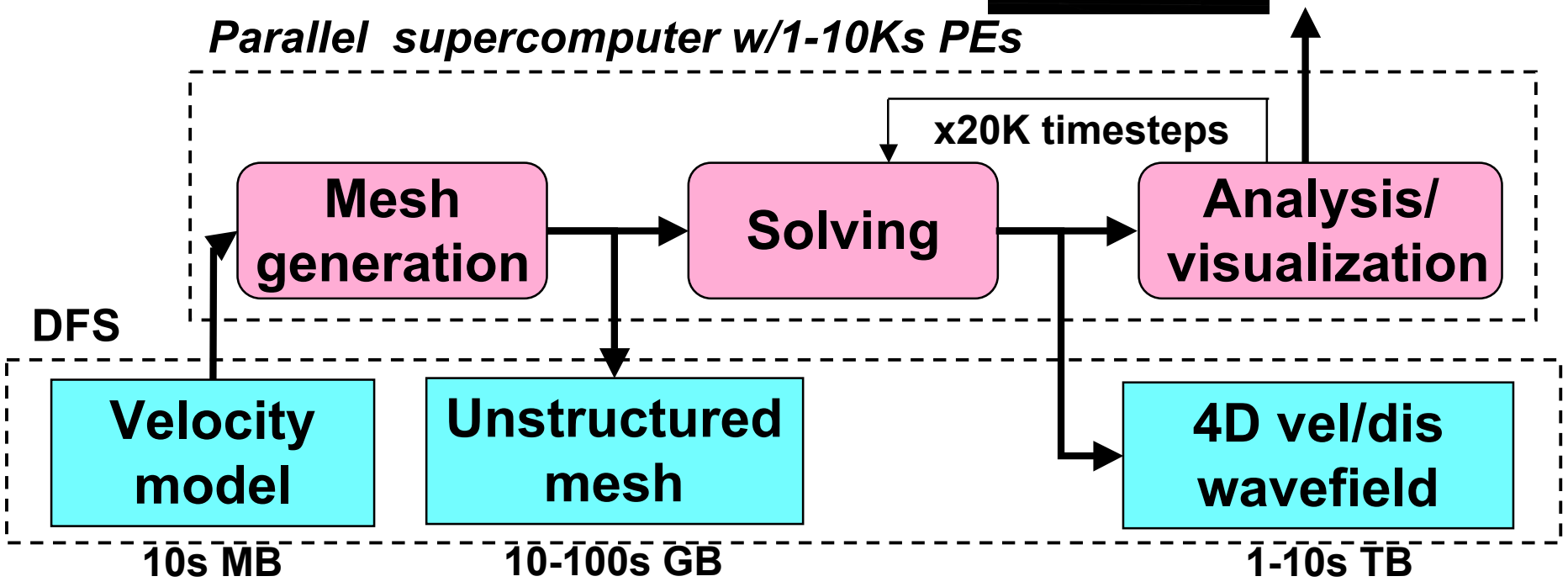
pdsi

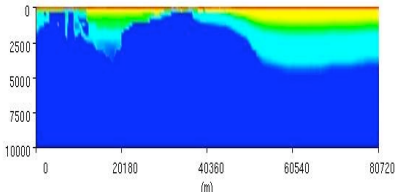
Earthquake simulation

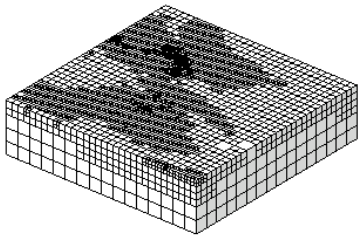
- c/o Dave O'Hallaron, www.cs.cmu.edu/~quake











Motivation: Earthquake analysis

- Serialization of model, mesh and wavefield into one database file
  - Total is generally much too large to be entirely loaded into memory
  - Multiple regions of the file contain data of different types
  - Different types of data see different access patterns
    - B-tree on mesh gets random tiny accesses
    - Mesh descriptors get random medium fetches
    - Wavefields get larger accesses, but concurrency makes it look random
- Basic problem: file system sees no internal structure information
  - Prefetching, allocation, caching strategies assume all data is of same type
  - Mixed access patterns strongly confuse file system policies
  - Fall back on safe, probably slower policies: no prefetching, serial locking
- What if file system knew more about “compound, out-of-core” files?
  - Hints? Tradition is too few codes issue them, too few systems use them, too hard to debug performance implications of hints
  - What alternatives?
    - Partial schema definitions: regions that have different types of access
    - Access Methods: embedded some structures in FS (B-trees, arrays …)

Motivation: file count scales w/ FLOPS?

- Understanding File Systems at Rest (www.pdsi-scidac.org/fsstats/)
  - Just getting started with data collection, but already see lots of tiny files
  - Manipulation of small files, and attributes of small files is a growing problem
  - Small things cost mechanical positioning -- orders of magnitude slower
  - Large files may in fact be collections of small things (HDF, netCDF, etc)
    - If sequentially loaded into memory and written from memory, cool
    - But if accessed “out-of-core”, really much the same problem
    - Gets worse as numbers of things gets larger, if not sequential access

```
file size:
count=12338926 average=18958.589506
min=0 max=757630040
  [         0-         2 KB):  3303866 (26.78%) ( 26.78% cumulative)      1996763.67 KB ( 0.00%) (  0.00% cumulative)
  [         2-         4 KB):   883060 ( 7.16%) ( 33.93% cumulative)      2534585.51 KB ( 0.00%) (  0.00% cumulative)
  [         4-         8 KB):   917461 ( 7.44%) ( 41.37% cumulative)      5182409.88 KB ( 0.00%) (  0.00% cumulative)
  [         8-        16 KB):   744358 ( 6.03%) ( 47.40% cumulative)      8591734.47 KB ( 0.00%) (  0.01% cumulative)
  [        16-        32 KB):   731235 ( 5.93%) ( 53.33% cumulative)     16534655.55 KB ( 0.01%) (  0.01% cumulative)
  [        32-        64 KB):   669568 ( 5.43%) ( 58.75% cumulative)     30855148.03 KB ( 0.01%) (  0.03% cumulative)
  [        64-       128 KB):   757320 ( 6.14%) ( 64.89% cumulative)     70214295.14 KB ( 0.03%) (  0.06% cumulative)
  [       128-       256 KB):   631071 ( 5.11%) ( 70.01% cumulative)    114050978.13 KB ( 0.05%) (  0.11% cumulative)
  [       256-       512 KB):   558914 ( 4.53%) ( 74.54% cumulative)    189985048.43 KB ( 0.08%) (  0.19% cumulative)
  [       512-      1024 KB):   597161 ( 4.84%) ( 79.37% cumulative)    443400973.63 KB ( 0.19%) (  0.38% cumulative)
  [      1024-      2048 KB):   479472 ( 3.89%) ( 83.26% cumulative)    676898557.71 KB ( 0.29%) (  0.67% cumulative)
  [      2048-      4096 KB):   363371 ( 2.94%) ( 86.21% cumulative)   1019631931.23 KB ( 0.44%) (  1.10% cumulative)
  [      4096-      8192 KB):   255781 ( 2.07%) ( 88.28% cumulative)   1534778534.48 KB ( 0.66%) (  1.76% cumulative)
  [      8192-     16384 KB):   256358 ( 2.08%) ( 90.36% cumulative)   2894041905.64 KB ( 1.24%) (  3.00% cumulative)
  [     16384-     32768 KB):   230819 ( 1.87%) ( 92.23% cumulative)   5245575759.34 KB ( 2.24%) (  5.24% cumulative)
  [     32768-     65536 KB):   223892 ( 1.81%) ( 94.04% cumulative)  10337335940.35 KB ( 4.42%) (  9.66% cumulative)
  [     65536-    131072 KB):   584808 ( 4.74%) ( 98.78% cumulative)  52004123186.77 KB (22.23%) ( 31.89% cumulative)
  [    131072-    262144 KB):    42167 ( 0.34%) ( 99.12% cumulative)   7784126469.45 KB ( 3.33%) ( 35.22% cumulative)
  [    262144-    524288 KB):    31868 ( 0.26%) ( 99.38% cumulative)  11411821832.03 KB ( 4.88%) ( 40.09% cumulative)
  [    524288-   1048576 KB):    39972 ( 0.32%) ( 99.70% cumulative)  27336893196.49 KB (11.69%) ( 51.78% cumulative)
  [   1048576-   2097152 KB):    17726 ( 0.14%) ( 99.85% cumulative)  25773260950.03 KB (11.02%) ( 62.80% cumulative)
  [   2097152-   4194304 KB):    13237 ( 0.11%) ( 99.96% cumulative)  37985398325.45 KB (16.24%) ( 79.04% cumulative)
  [   4194304-   8388608 KB):     4336 ( 0.04%) ( 99.99% cumulative)  23511276177.30 KB (10.05%) ( 89.09% cumulative)
  [   8388608-  16777216 KB):      783 ( 0.01%) (100.00% cumulative)   8739054420.16 KB ( 3.74%) ( 92.82% cumulative)
  [  16777216-  33554432 KB):      168 ( 0.00%) (100.00% cumulative)   3598648498.69 KB ( 1.54%) ( 94.36% cumulative)
  [  33554432-  67108864 KB):      111 ( 0.00%) (100.00% cumulative)   5587776404.47 KB ( 2.39%) ( 96.75% cumulative)
  [  67108864- 134217728 KB):       25 ( 0.00%) (100.00% cumulative)   2318337024.21 KB ( 0.99%) ( 97.74% cumulative)
  [ 134217728- 268435456 KB):        9 ( 0.00%) (100.00% cumulative)   1559281156.26 KB ( 0.67%) ( 98.41% cumulative)
  [ 268435456- 536870912 KB):        8 ( 0.00%) (100.00% cumulative)   2969396082.00 KB ( 1.27%) ( 99.68% cumulative)
  [ 536870912-1073741824 KB):        1 ( 0.00%) (100.00% cumulative)    757630040.00 KB ( 0.32%) (100.00% cumulative)
```

```
"# The data contained herein was collected from Systems in the Environmental Molecular Sciences Laboratory", a
national scientific user facility sponsored by the Department of Energy's Office of Biological and Environmental
Research and located at Pacific Northwest National Laboratory."
"# Release Number: PNNL-17013"
"# Contact: Evan Felix <evan.felix@pnl.gov>"
```

Carnegie Mellon
Parallel Data Laborat

pdsi

Motivation: HugeDirs -> AttributeDB

- CMU adding directories of 1B to 1T small files to PVFS
  - Partitioning entries into buckets distributed over all storage servers
  - Building index structure for insert(), delete(), lookup()
    - no range query, but unordered complete scan must be fast (readdir)
  - Making highly parallel, minimal bottlenecks, minimal coherency
  - Believe this is extensible to range queries (concurrent B-trees)
    - More challenging to fully load balance if key access patterns arbitrary
    - Probable next steps to do this however
- Simple extension to not create actual files, just manipulate records
  - Special directory is “key, opaque data” tuples
  - Use for extended attributes of small files to optimize to access patterns
    - Ie., type: mp3, code, pdf, checkpoint, …. Find all by type
    - Ie., timestamp of last modification, find N oldest
    - Ie,. Size, find N largest
  - Apply powerful aggregation structures as well as indices: ie., fastbits
- Could this be used for variable stores?

Motivation: ChangeLog Representation

- Fastest checkpoint might be a sequential series of “variable=value”
  - Instead of seeking to serialized location, just append operation to log
  - Each thread writes strictly sequential log of operations
  - “Meaning” of set of logs is applying log to (possibly null) initial database
- Decouple writing logs from applying logs to serialized database
  - Optimize each separately; pipeline from compute to IO nodes
  - Defer serializing by just storing changelogs for later application
  - Some checkpoints never read, so never serialized
  - If read before serialized, trigger serialization (or something smarter)
  - Represent logs as attributes of database; that is, hide in FS (directory?)
  - Important tricks: “block logs” so each can be applied in parallel later; type logs so “no overwrite” can be known and any order allowed
- Not always best representation
  - If data sequential anyway, operation encoding probably larger
  - If read intermingled with write, might force inefficient serialization

Issues: Library vs File System

- Library is user code bound into application
  - File has structure if the right library is bound into app touching it
  - Rapid development of new “ease of programming” APIs
  - Optimization for actual file system is best guess of library writers
    - Especially a problem if ease of programming was big goal
- File system is system code independent of application
  - File structure known to file system, facilitating optimization
  - Slower development and proliferation as file system is a broad service
  - FS is always-there service; easy delayed processing, rebalancing, recovery
  - FS is inherently distributed, aware of actual servers, changes in servers
- Deployment of new file system APIs?
  - Changes in parallel file systems in progress: pNFS, HEC Posix extensions
- Mitigation for deployment issues: canoncial representations
  - Any file system offering a specialized API also offers:
    - Canonicalization (tar, untar) routines and library implementation written to canonical representation, which does not have to be (as) fast
    - [special case of the backup problem & NDMP solution]

Next steps?

- Select and work through a few examples