Design, Implementation and Testing of a Distributed Data Structure (DDS)

j. gentile
Benefits of a DDS

- Less load on compute node
- Takes advantage of unused resources
The Rub

• These 'benefits' only hold if a DDS performs equivalent or better than a locally stored structure

• Performance
  - Time taken on an access pattern
Naming Conventions

- **Compute Node**
  - Node where computation takes place

- **Storage Node**
  - Computer used to store part of the data structure
Functions of a DDS Library

• Allocation of the DDS
• Access to elements in the DDS
• Deletion of the DDS
Specifications

• Allocation takes place once for all
  – Reserves resources
• Reads are accurate
• Writes change values
• Free emancipates the resources
First DDS

• Square Matrix
  - Grows exponentially with respect to vector space
  - Used for all access patterns
    • LUT- random access
    • Linear algebra- sequential access
Allocation

• The DDS reserves the resources needed to store the entire structure

• User should have some say in the matter
  - Hosts
    • Access
    • Trust
    • Stability
Reads and Writes

- Reads return the value of the DDS at that element
- An element is not changed unless written to
- Should look local relative to the user
Deletion

- Frees all reserved resources
- No left overs
Design

- Vector-wise partition of the DDS
- Stored as a binary file on the storage nodes
- Accessed by row and column offsets
Design- Allocation

• User provides the following
  – Vector Space
  – Host List
  – Chunk

• Size of a chunk is defined by
  \[ s = \frac{(space \times t)}{\text{chunk}} \]

• \( s \) is chunk size
• \( t \) is data type size
• chunk- number of chunks
Design - Allocation

ephesus.cse.nd.edu
smyrna.cse.nd.edu
pergamos.cse.nd.edu
thyratira.cse.nd.edu
sardis.cse.nd.edu

 DDS

Allocations
Attempts

Read

sardis.cse.nd.edu
Design- I/O

- Collapse 2-D into 1-D  $l=(row\_size\times r+c)$
- Find the node that holds the element of interest
- Move cursor to element
- Read or write
Design- Free

- Delete all files that stored the data
Class Structure

- User interfaces with Mother Class
- Inode points to data
  - Inodes point to hosts
Allocation

- Inode made for all chunks
Read and Writes

• Find the Inode with the data
• Move Cursor to it
• Read or write bytes
Free

- Free all the files pointed to by Inodes
Experiments

- What behavior pattern is best?
- Better or worse than locally stored?
- What is the cost of spreading?
- How consistent are access times?
Details

• Access patterns
  – Random
    • 10,000 random accesses
  – Sequential
    • Access all tuples of one vector

• 100 Mb/s connection between compute node and storage nodes
Cost of Size and Benefit of Distribution

- Allocate structures of different sizes locally and distributively
- Measure access times for the structures
Random Reads

Random Reads

Time in Seconds

Size in Bytes
Random Writes

The graph shows the performance of random writes for two different types: Local and Distributed. The x-axis represents the size in bytes, ranging from 0 to 1e+10. The y-axis represents the time in seconds, ranging from 0 to 70. The Local writes show a sharp increase in time as the size increases, while the Distributed writes remain relatively stable.
Sequential Reads

![Graph showing Sequential Reads with two lines representing Local and Distributed read times. The x-axis represents size in bytes (from 0 to 1e+10), and the y-axis represents time in seconds (from 0 to 2). The graph shows a peak at around 5e+09 bytes for Local reads and a steady increase for Distributed reads.]
Consistency

- Repeated access one matrix.
Size V. Spread

- Increase spread of data structure keeping size fixed.
- Measure the time for all accesses
- Sequential access read in $1/10^{th}$ of the tuple space
Conclusions

- DDS is faster at random accesses at some point
- Sequential reads are equivalent
- Sequential writes are poor
  - Can be fixed by caching
- Access time is tight
- No cost from in going from 1-9 in spread