The Fusion Distributed File System

Dongfang Zhao
February 2015

Outline

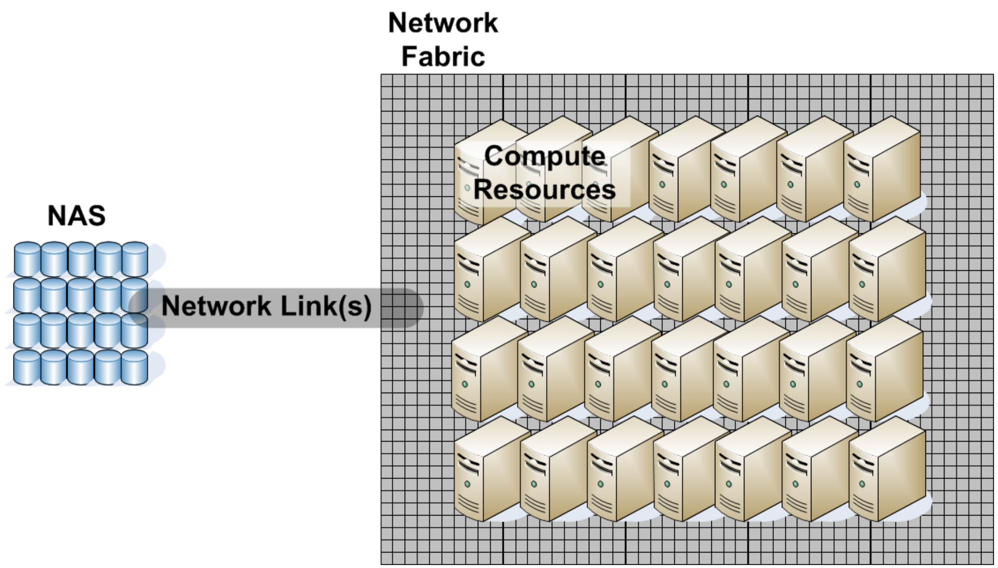
- Introduction
- FusionFS
  - System Architecture
  - Metadata Management
  - Data Movement
  - Implementation Details
  - Unique Features: virtual chunks, cooperative caching, distributed provenance
- Results
- Future Work

Outline

- **Introduction**
- FusionFS
- Results
- Future Work

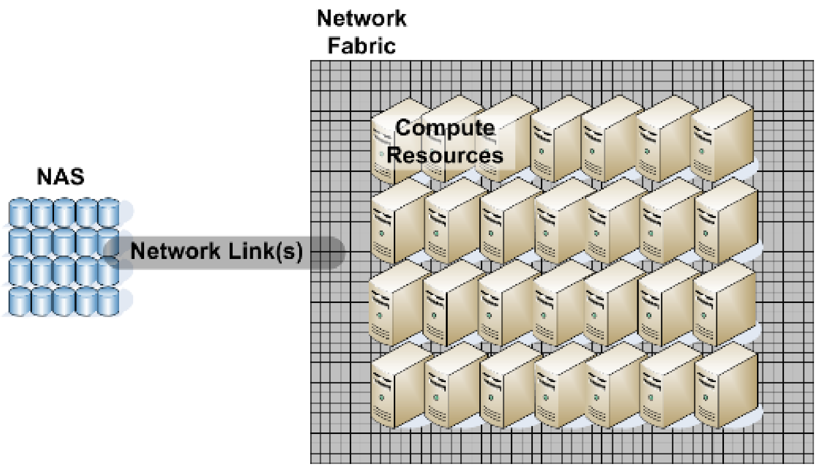
Background

- State-of-the-art high-performance computing (HPC) system architecture
  - Decades old
  - Has compute and storage resources separated
  - Was originally designed for compute-intensive workloads



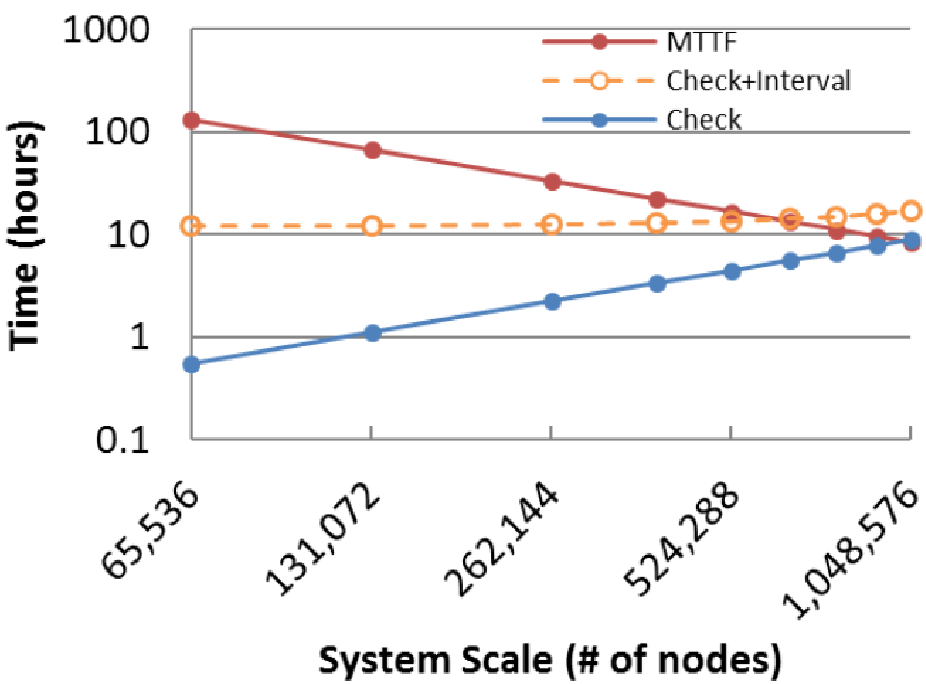
I/O Bottleneck

- Would this architecture suffice in Big Data Era at extreme scales? Concerns:
  - Shared network infrastructure
  - All I/Os are (eventually) remote
  - More frequent checkpointing



Challenges at Exascales

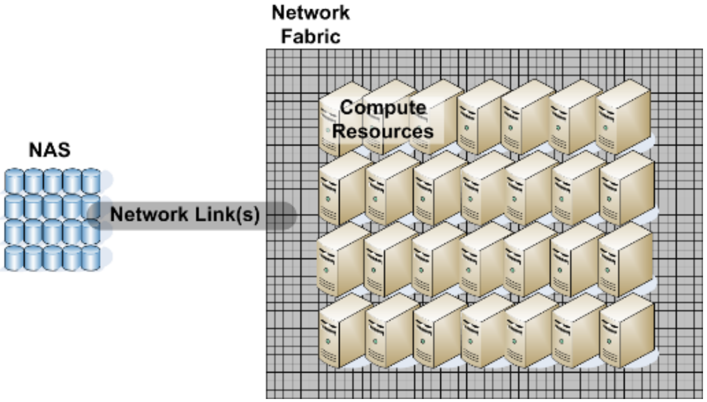
- Let's simulate, for example, checkpointing, at extreme scales



More detail available at: ACM/SCS HPC '13 [6]

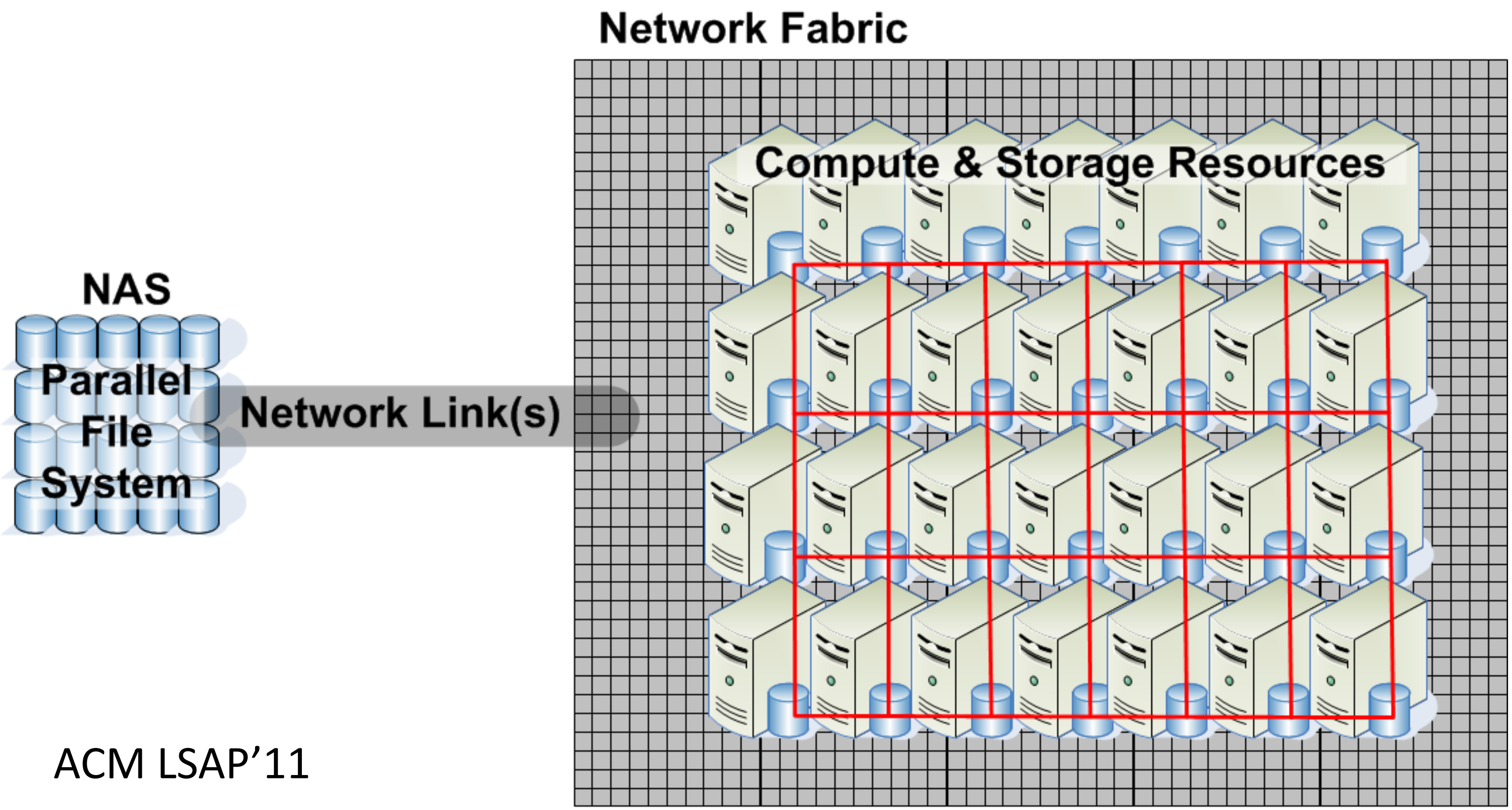
The Conventional Wisdom

- Recent study to address the I/O bottleneck (without changing the current existing HPC architecture)
  - Ning Liu, et al. On the Role of Burst Buffers in Leadership-Class Storage Systems, IEEE MSST'12
  - Pillips Carns, et al. Small-file access in parallel file systems, IEEE IPDPS'09



Proposed Work

- We address the issue by proposing a new architecture: introducing node-local storage



ACM LSAP'11

Outline

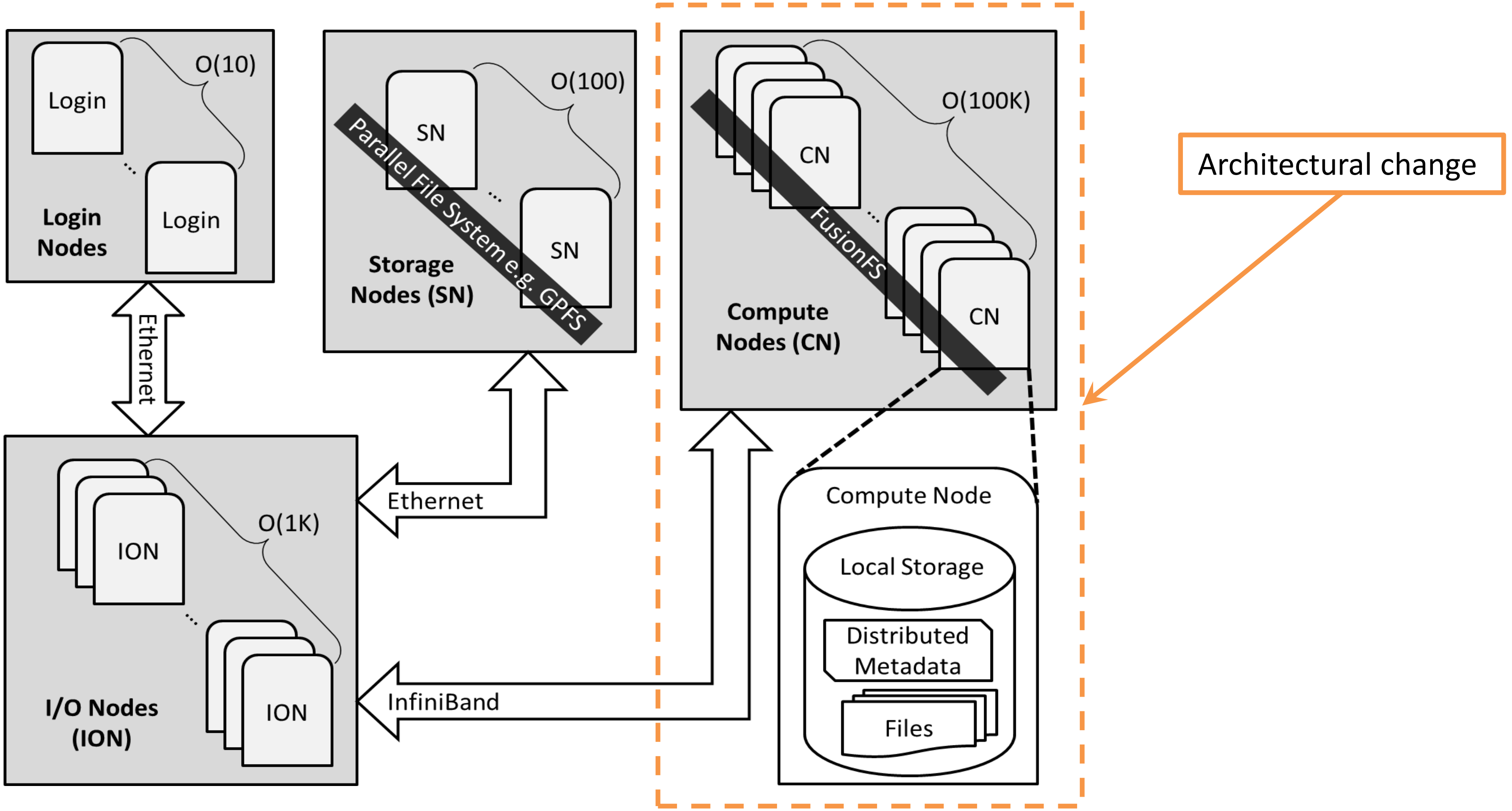
- Introduction
- **FusionFS**
- Results
- Future Work

Overview

- Goal: a node-local distributed storage for HPC systems
- Design principles
  - Maximal metadata concurrency
    - Distributed hash table
  - Optimal write throughput
    - Write local disk if possible

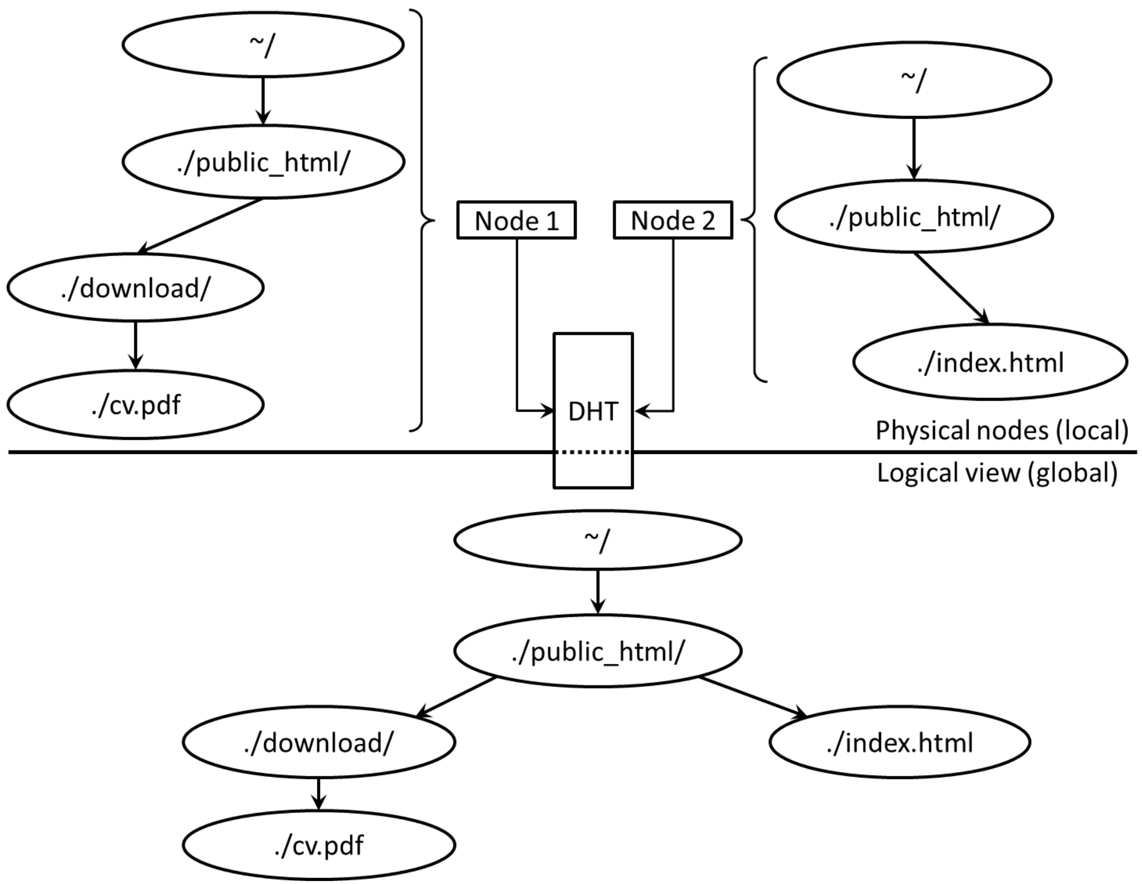
Architecture

- In IBM Blue Gene/P supercomputer:



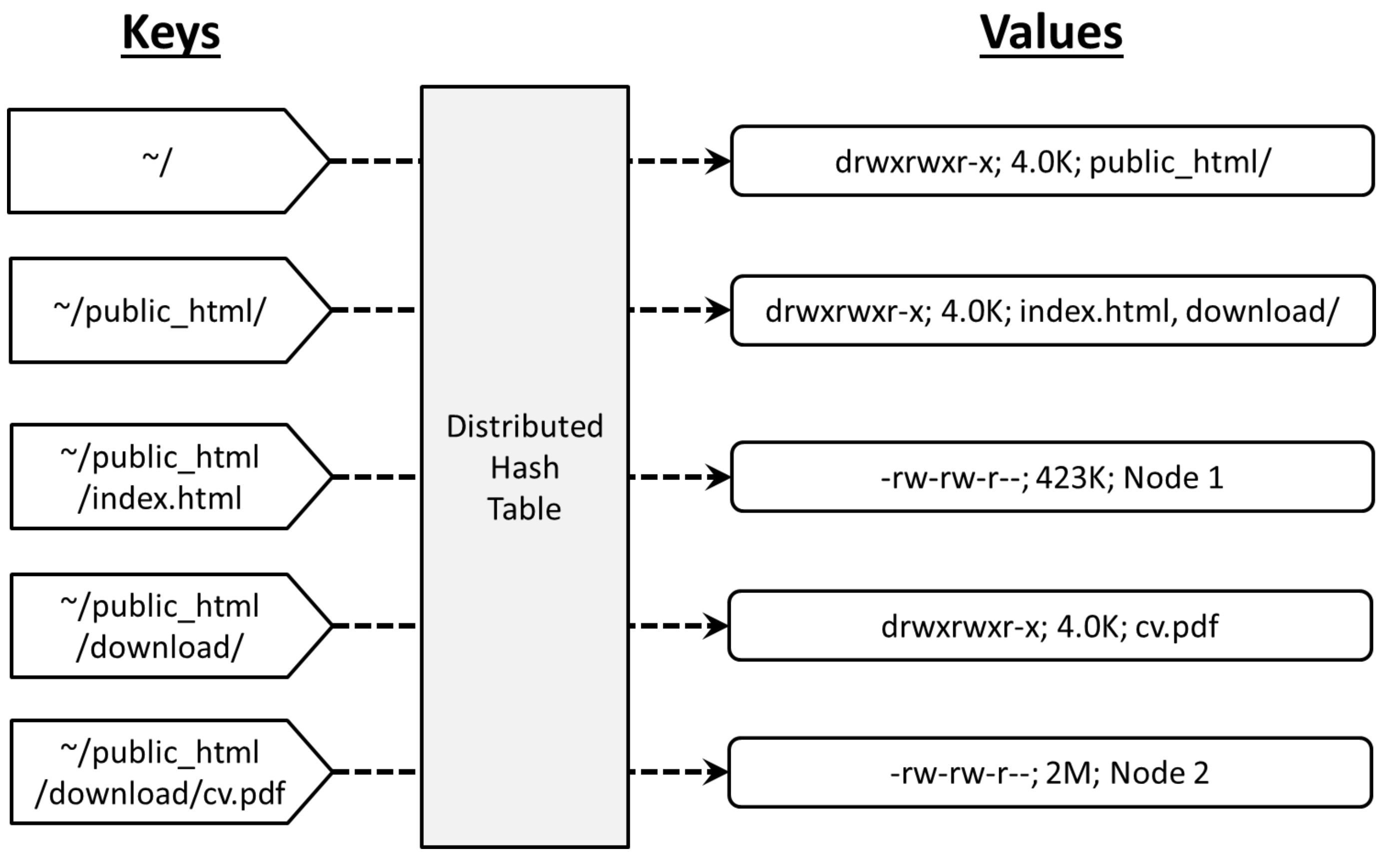
Directory Tree

- Physical-logical mapping



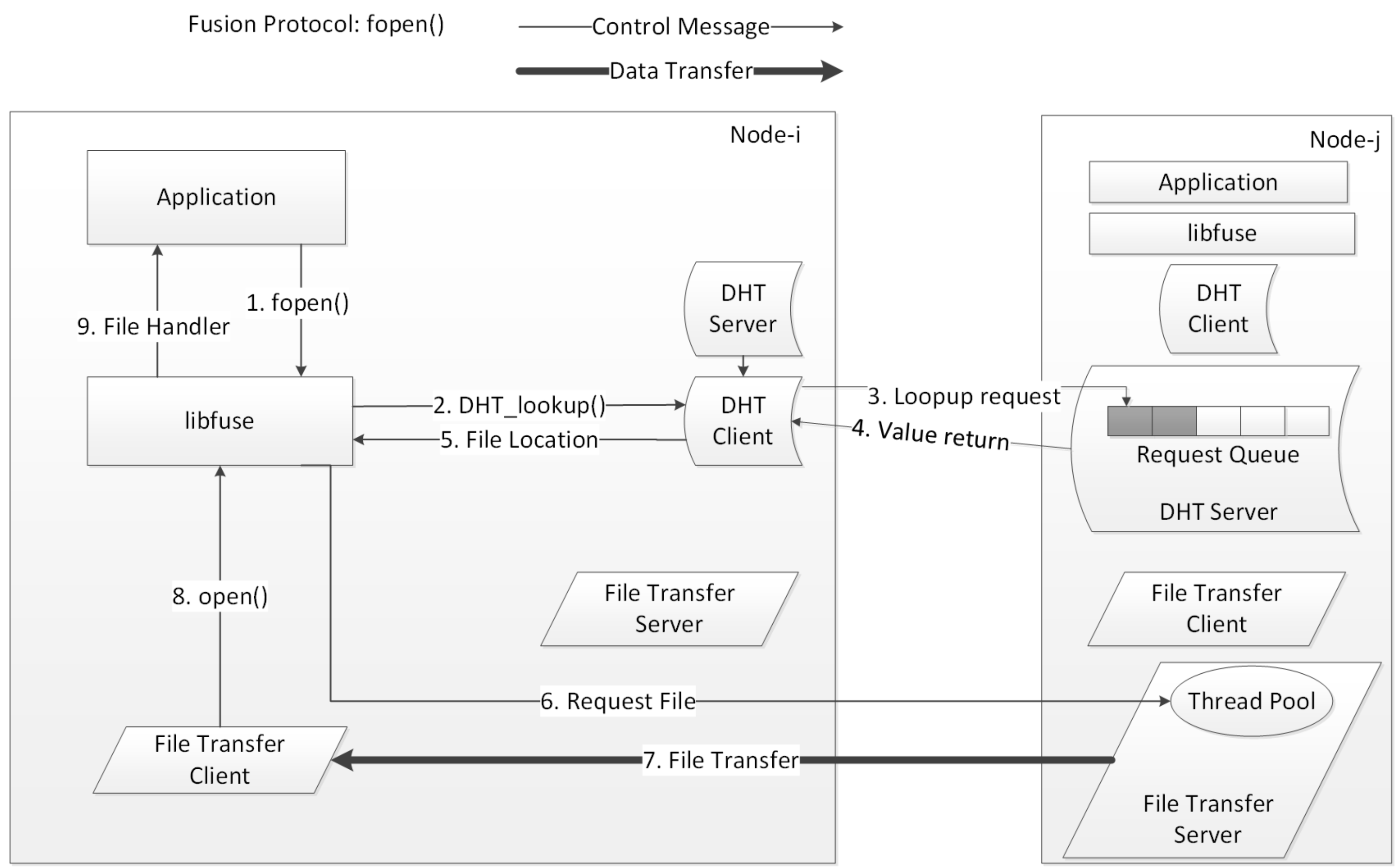
Metadata Management

- Directories vs. regular files



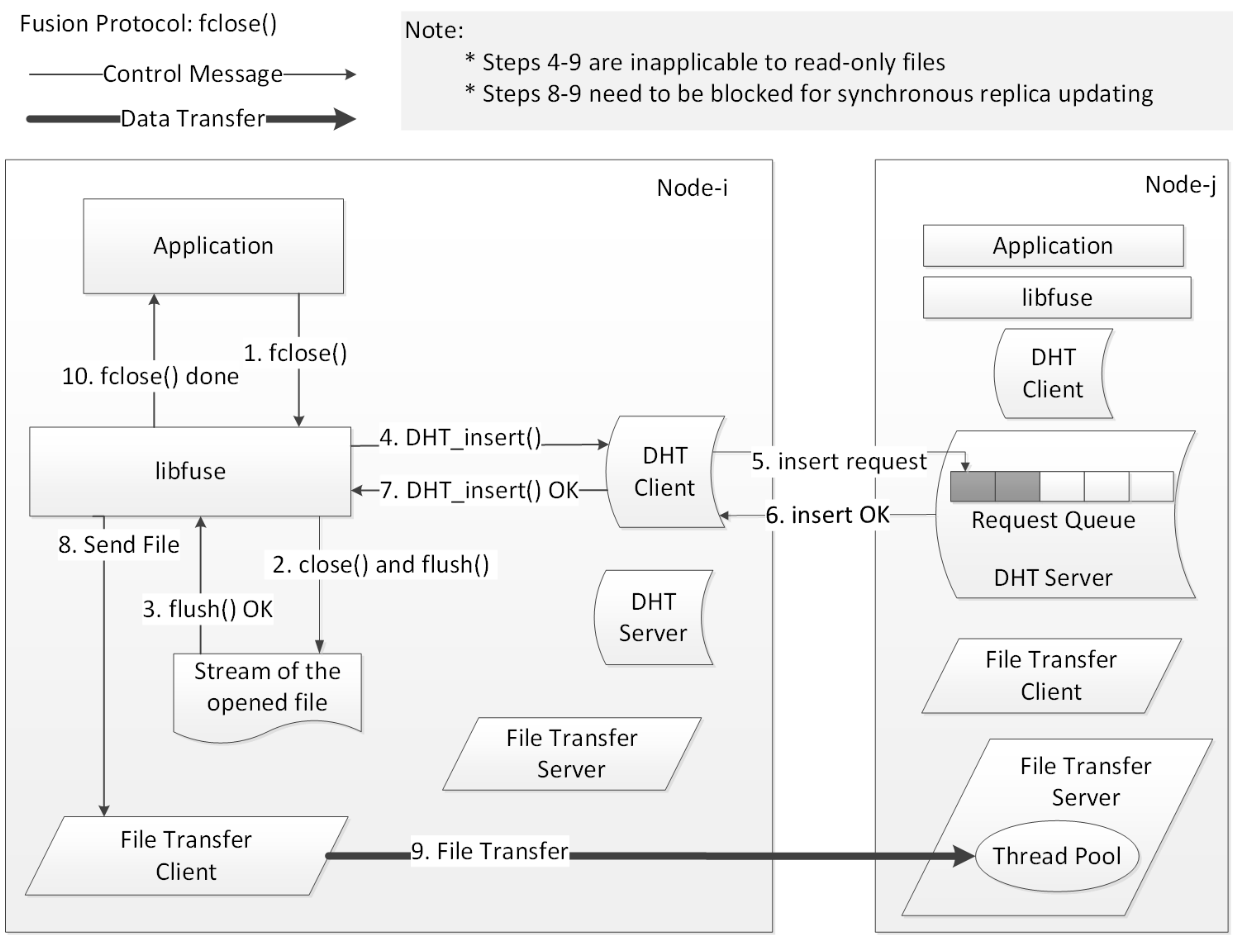
File Manipulation

- File open



File Manipulation

- File close



FusionFS Implementation

- C/C++
- Building blocks
  - FUSE (user-level POSIX interface)
  - ZHT (distributed hash table for metadata)
  - Protocol Buffer (data serialization)
  - UDT (data transfer protocol)
- Open source: https://bitbucket.org/dongfang/fusionfs-zht/src

Virtual Chunks

- One of features in FusionFS
- Goal: support efficient random accesses to large compressed scientific data

Background

- In Big Data Era, many data-intensive scientific applications' performance are bounded on I/O
- Two major solutions:
  - Parallelize the file I/O by concurrently read or write independent file chunks
  - Compress the (big) file before writing it to the disk, and decompress it before reading it
    - Could be done efficiently at the file system layer
      - transparent to end users

Conventional Wisdom

- File systems leverage data (de)compression in a naïve manner:
  - Apply the compressor when writing files
  - Apply the decompressor when reading files
  - Leave important metrics to the underlying compressing algorithms
    - Compression ratio
    - Computational overhead

Limitations

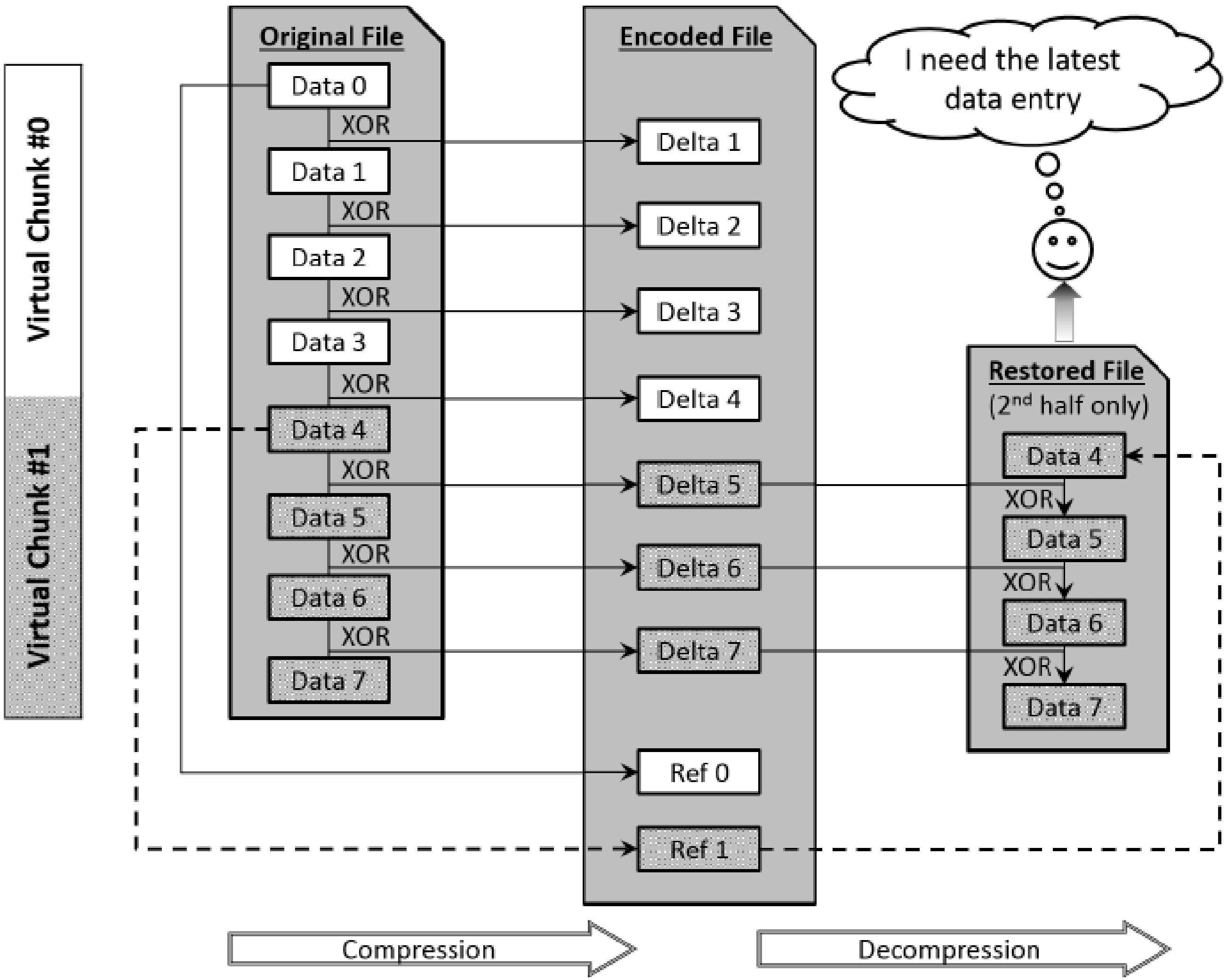
- File-level compression: computation overhead of read. E.g., read the latest temperature
  – Step 1: compress the entire 1GB climate data to 500MB file
  – Step 2: decompress the 500MB file to only retrieve the last few bytes
- Chunk-level compression: space overhead of write. E.g., write 64MB data with 4:1 compression ratio and 4KB metadata
  – 64K-chunk: 16MB + 4KB * 1K = 20MB
  – No chunk: 16MB + 4KB ≈ 16MB

Key Idea

- Virtually split the file into smaller chunks but keep its physical entirety
  - Small chunk: fine granularity for random accesses
    - So to fix the computation overhead of file-level compression
  - Physical entirety: high compression ratio
    - So to fix the space overhead of (physical)chunk-level compression

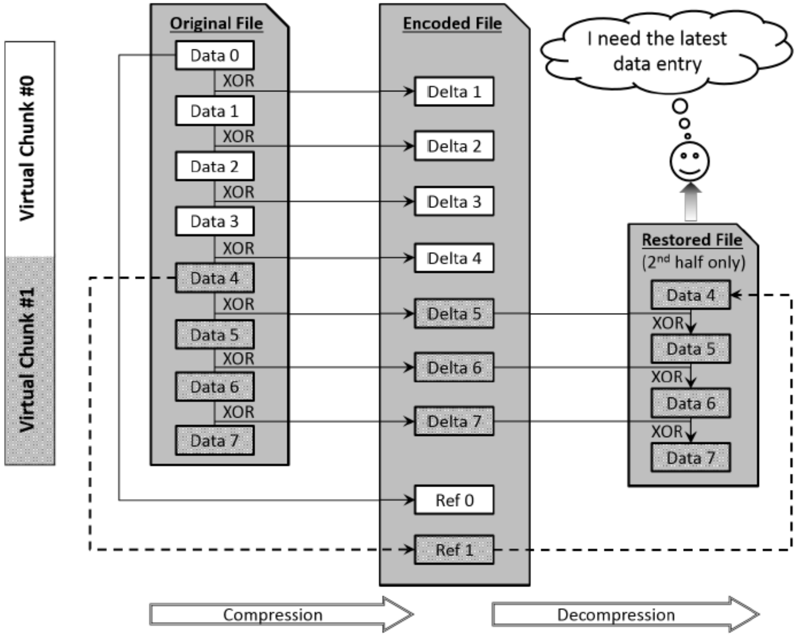
Motivating Example

- A simple XOR-based delta compression



Reference Placement

- Need to consider different strategies
  - In place
    - Pro: save one jump
    - Con: overhead for sequential read
  - Coalition
    - Beginning or end



Compression Procedure

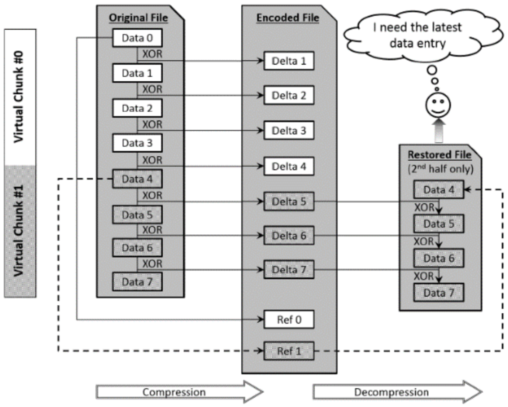
- Turns to be very straightforward:

Algorithm 1 VC Compress

Input: The original data $D = \langle d_1, \cdots, d_n \rangle$
Output: The encoded data X, and the reference list D'

for (int i = 1; i < n; i++) **do**
    $X[i] \leftarrow \texttt{encode}\,(d_i,\, d_{i+1})$
end for
for (int j = 1; j < k; j++) **do**
    $D'[j] \leftarrow D[1 + (j-1) * L]$
end for

- Runs efficiently:
  - Time complexity is O(n)



But… How Many Virtual Chunks?

- Extreme cases
  - 1 virtual chunk
    - Same as file-level compression
  - N virtual chunks (suppose there are N physical chunks in the file)
    - Same as (physical)chunk-level compression
- Find a number for good tradeoff?
  - Let’s assume the requested file is located at the end of the virtual chunk

Number of Virtual Chunks

- Variables

Variable	Description
B_r	Read Bandwidth
B_w	Write Bandwidth
W_i	Weight of Input
W_o	Weight of Output
S	Original File Size
R	Compression Ratio
D	Computational Time of Decompression

- Optimum for end-to-end I/O

$$k_{opt} = \begin{cases} \lfloor \hat{k} \rfloor & \text{if } F(\lfloor \hat{k} \rfloor) > F(\lceil \hat{k} \rceil) \\ \lceil \hat{k} \rceil & \text{otherwise} \end{cases}$$

where

$$\hat{k} = \sqrt{n \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o} \cdot (\frac{1}{R} + \frac{D \cdot B_r}{S})} \qquad (2)$$

and

$$F(x) = \frac{(x-1) \cdot S \cdot W_i}{x \cdot R \cdot B_r} + \frac{(x-1) \cdot D \cdot W_i}{x} - \frac{(x-1) \cdot S \cdot W_o}{n \cdot B_w}$$

- Short version

$\hat{k} = \sqrt{n}$ (under some assumptions, refer to the paper for detail)

Decompression Procedure

- Algorithms omitted (refer to paper for detail)
- Key steps:
  - Find the latest reference (*r_last*) before the starting position of the request file
  - Decompress the file from *r_last*, until the end of the requested data
    - For file write, we also need to update values after the decompression, and re-compress the affected portion

Outline

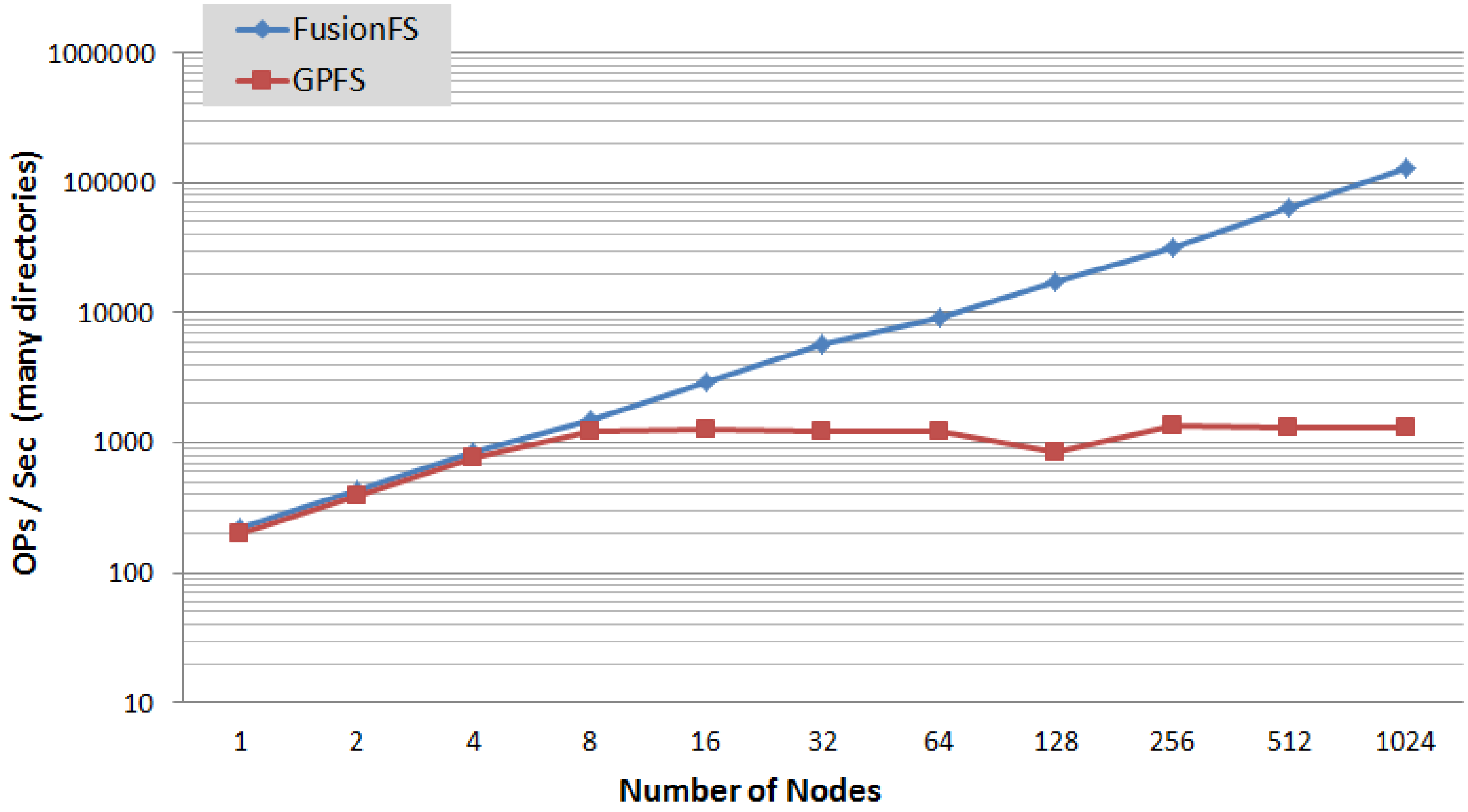
- Introduction
- FusionFS
- **Results**
- Future Work

Test Bed

- Intrepid (Argonne National Laboratory)
  - 40K compute nodes, each has
    - CPU: quad-core 850 MHz PowerPC
    - RAM: 2 GB
  - 128 storage nodes, 7.6 PB GPFS parallel file system
- Others not covered by this presentation
  - Kodiak, 1,024-node cluster at Los Alamos National Laboratory
  - HEC, 64-node Linux cluster at SCS lab
  - Amazon EC2

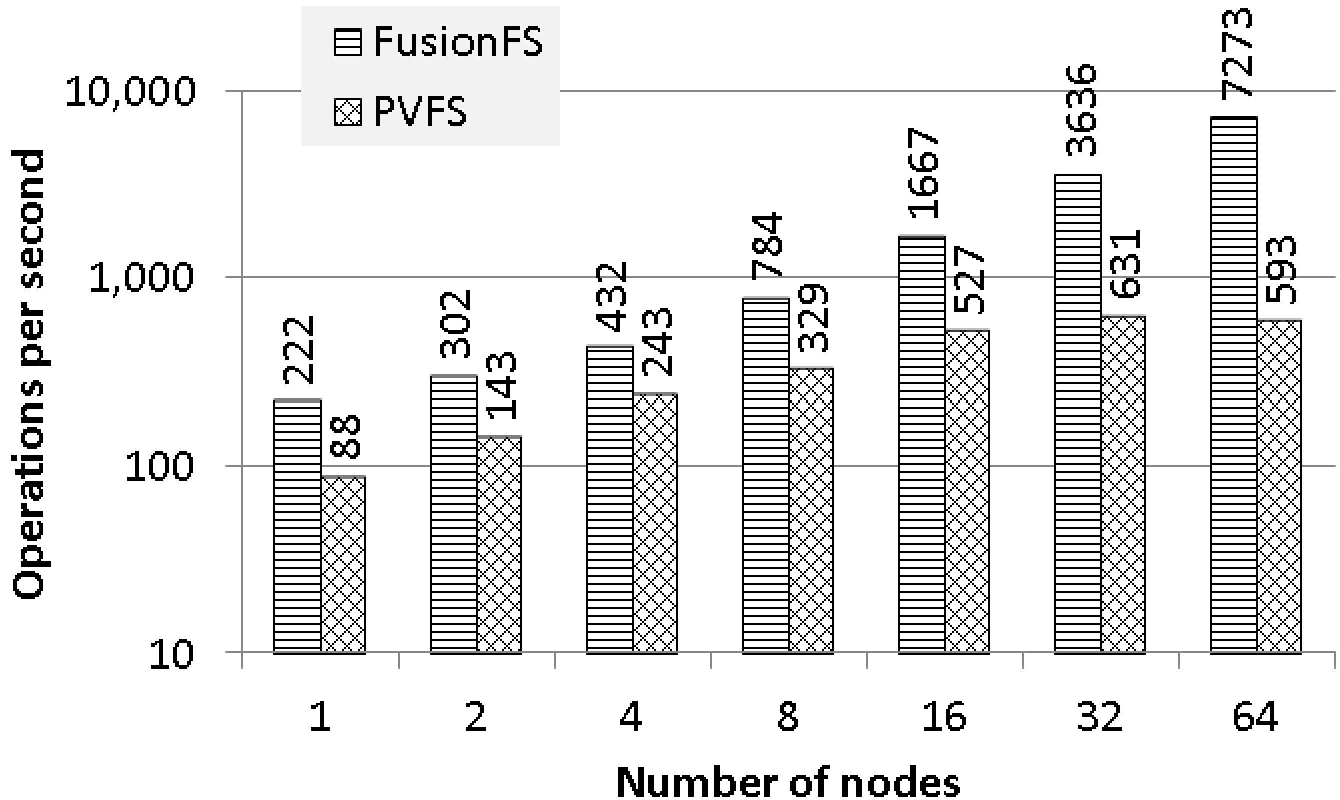
Metadata Rate

- FusionFS vs. GPFS



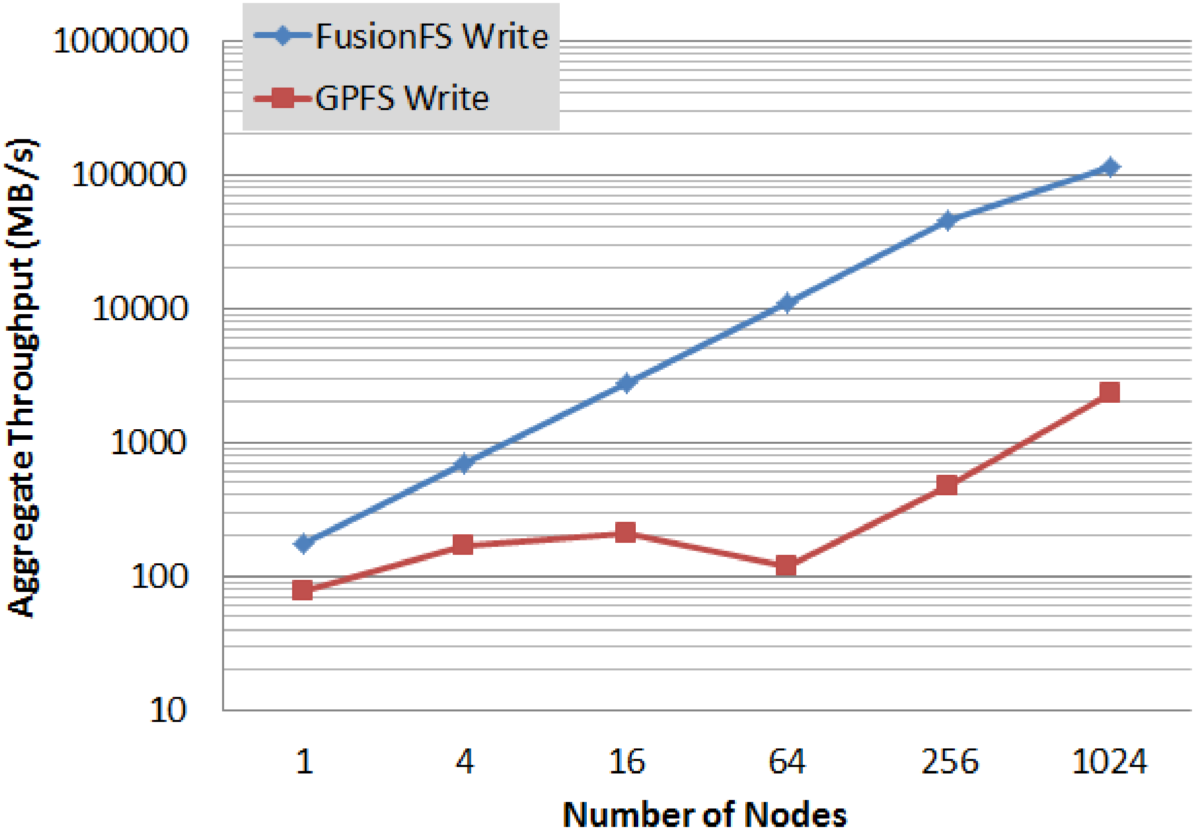
Metadata Rate

- FusionFS vs. PVFS



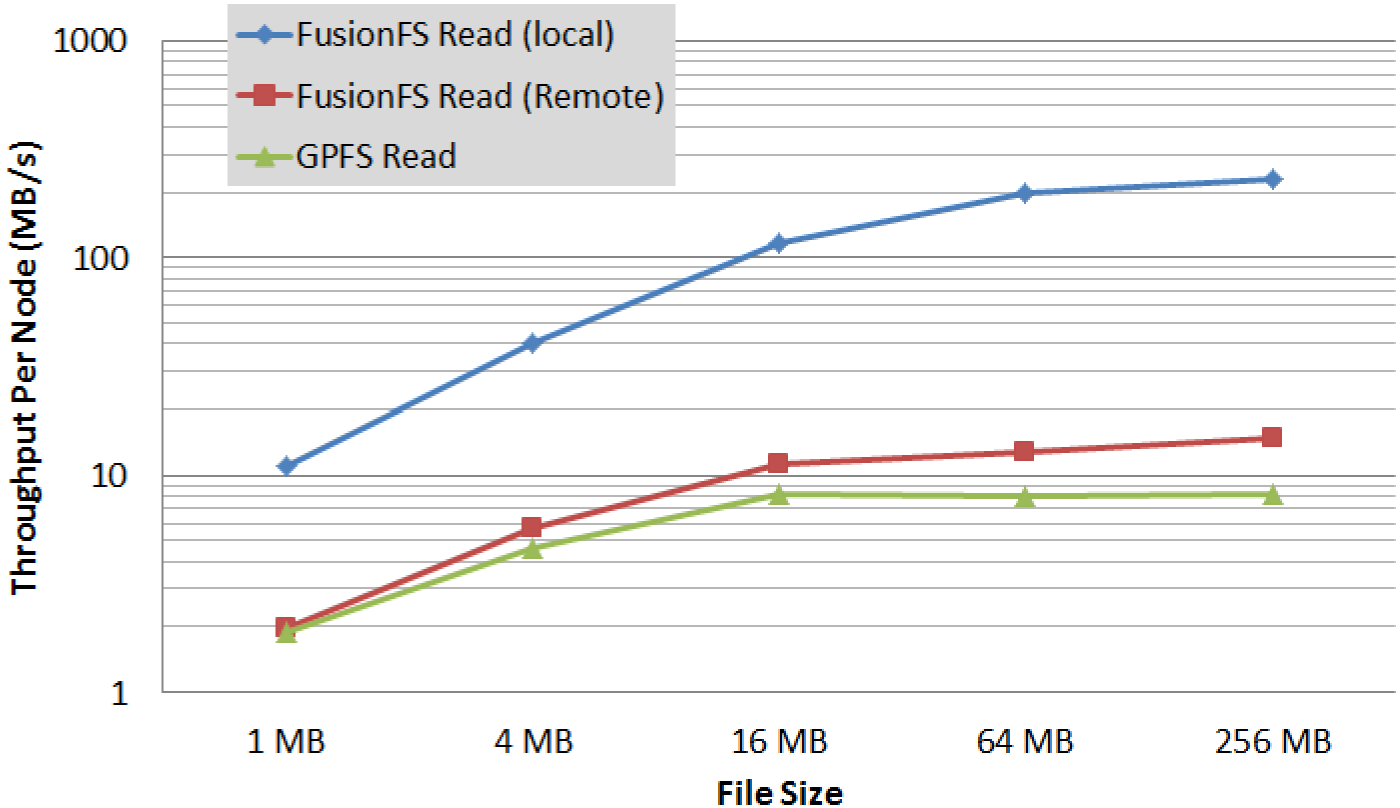
I/O Throughput

- Write, FusionFS vs. GPFS



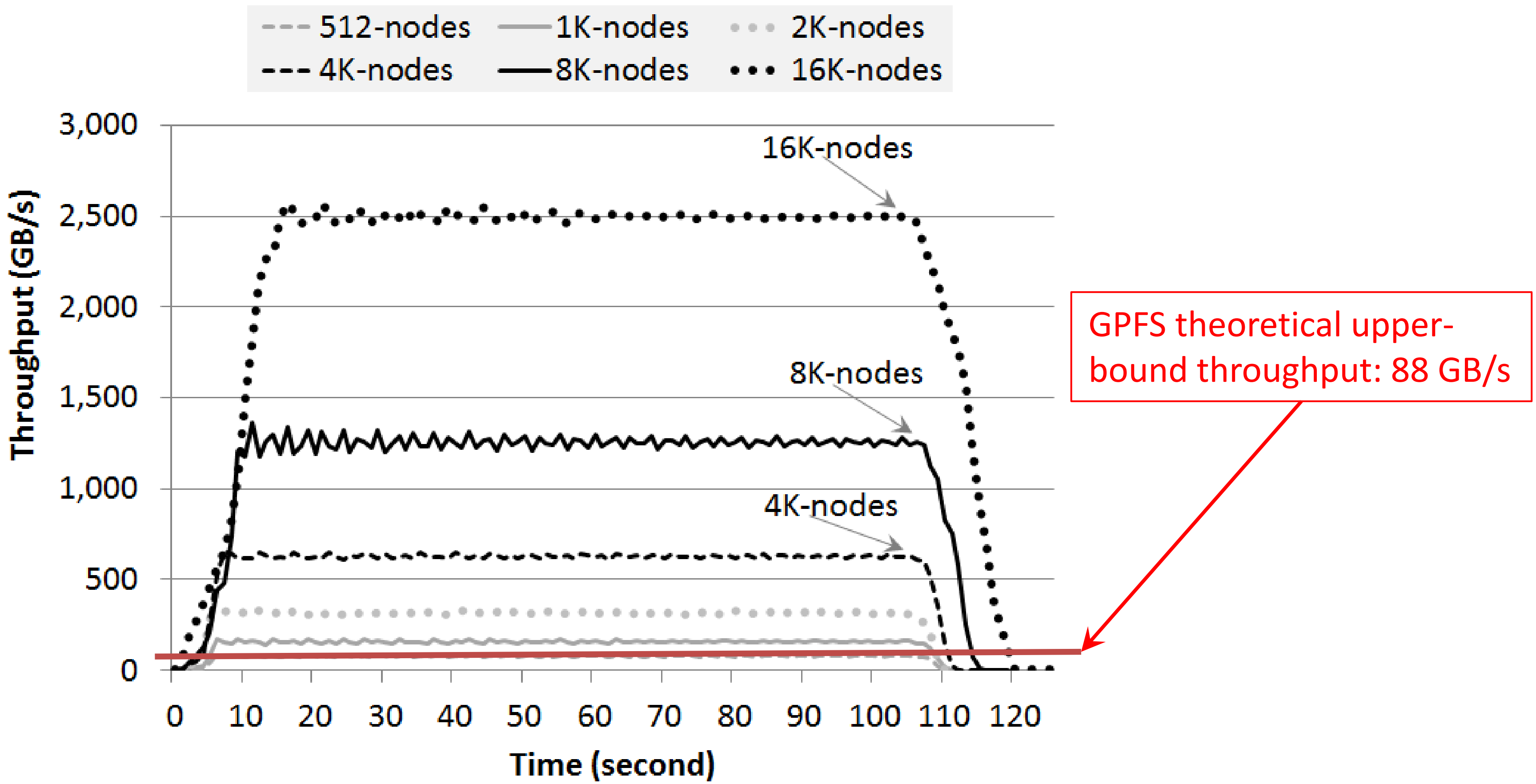
I/O Throughput

- Read, FusionFS vs. GPFS



I/O Throughput at Extreme Scale

- Peak: 2.5 TB/s on 16K nodes:

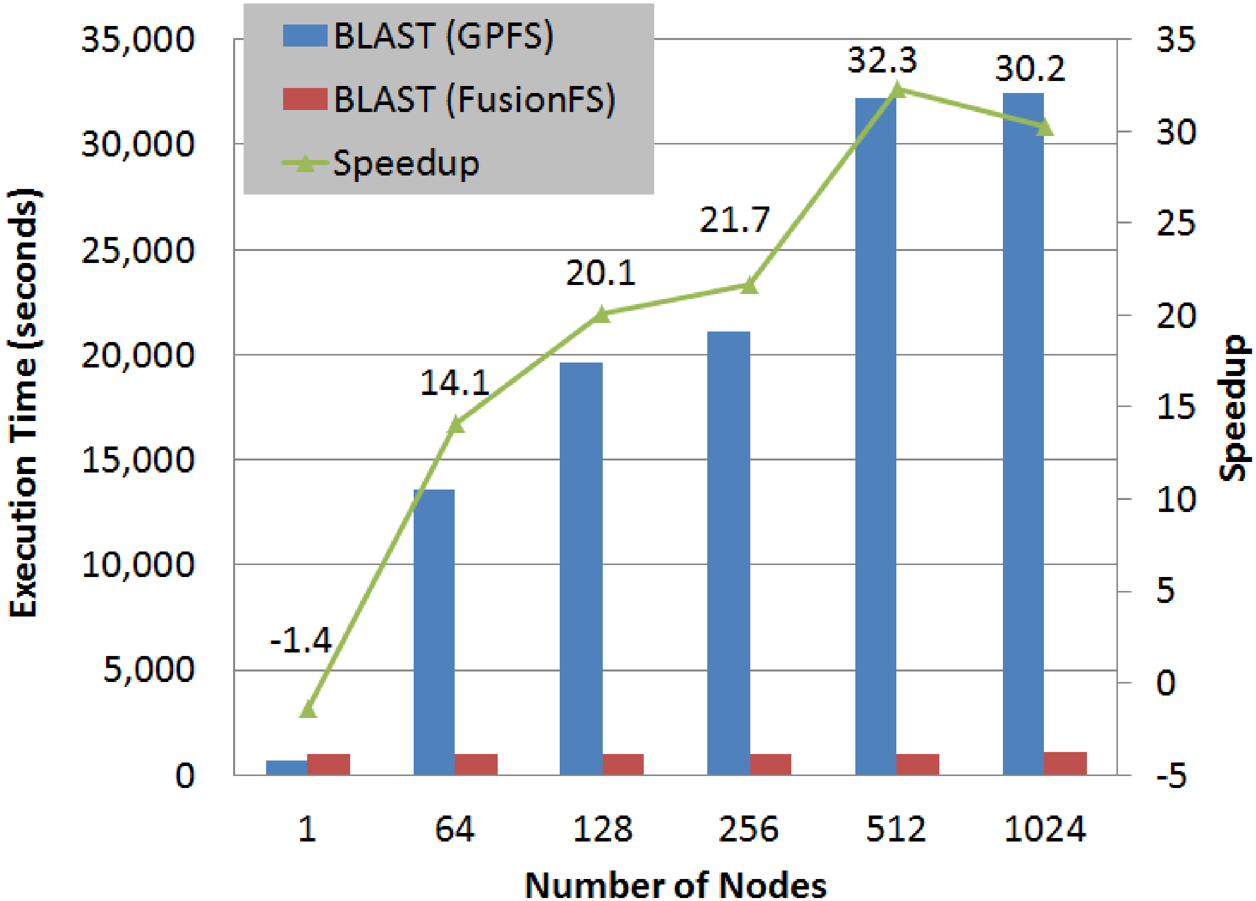


The BLAST Application

- BLAST: Basic Local Alignment Search Tool
- A Bioinformatics application that
  - Searches one or more protein sequences against a sequence database
  - Calculates similarities
- Main idea: divide and conquer
  - Split the database and store chunks on different nodes
  - Each node searches the matching for assigned chunks
  - All searches are merged into the final results

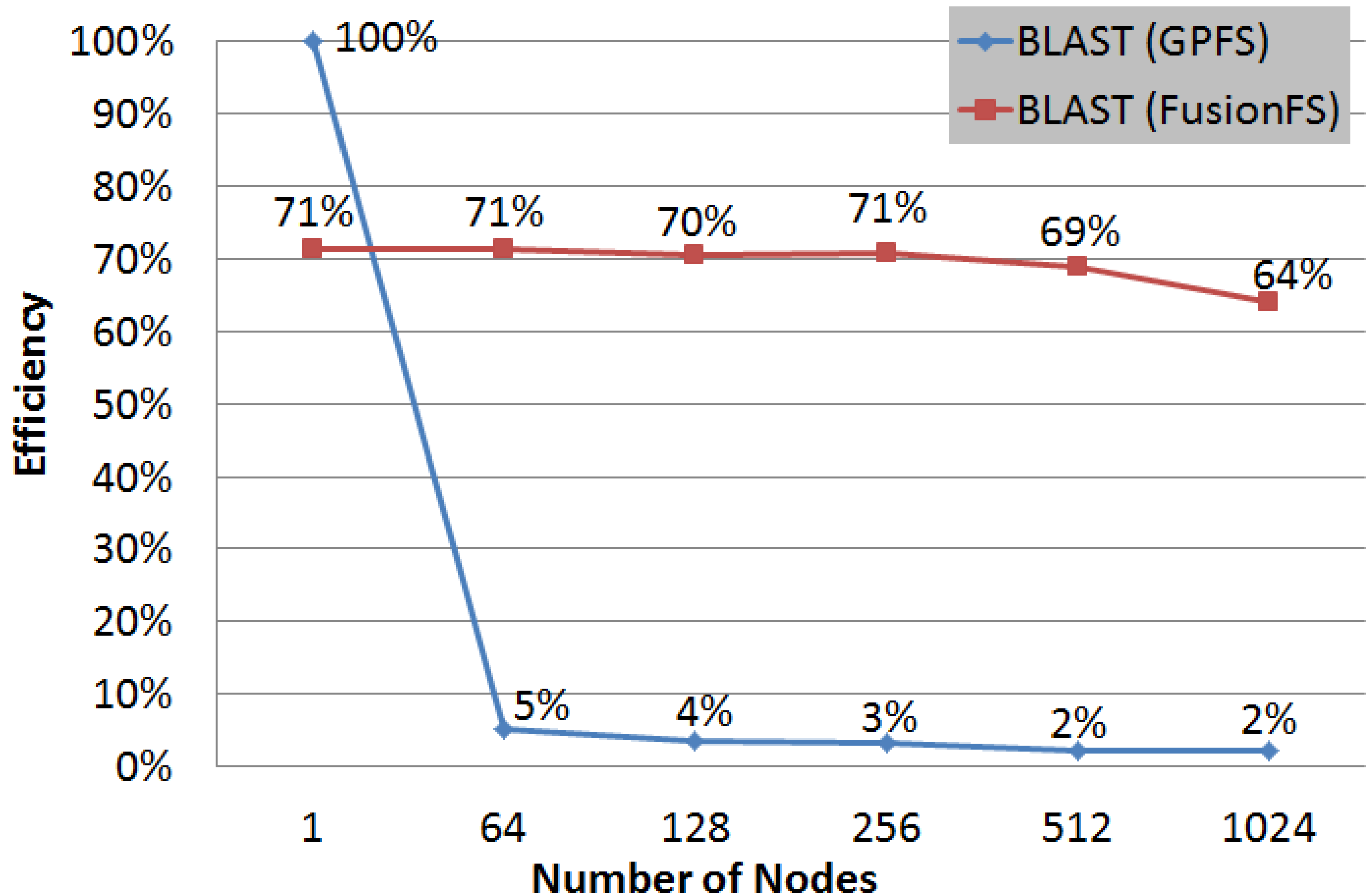
Execution Time

- FusionFS 30X faster on 1,024 nodes



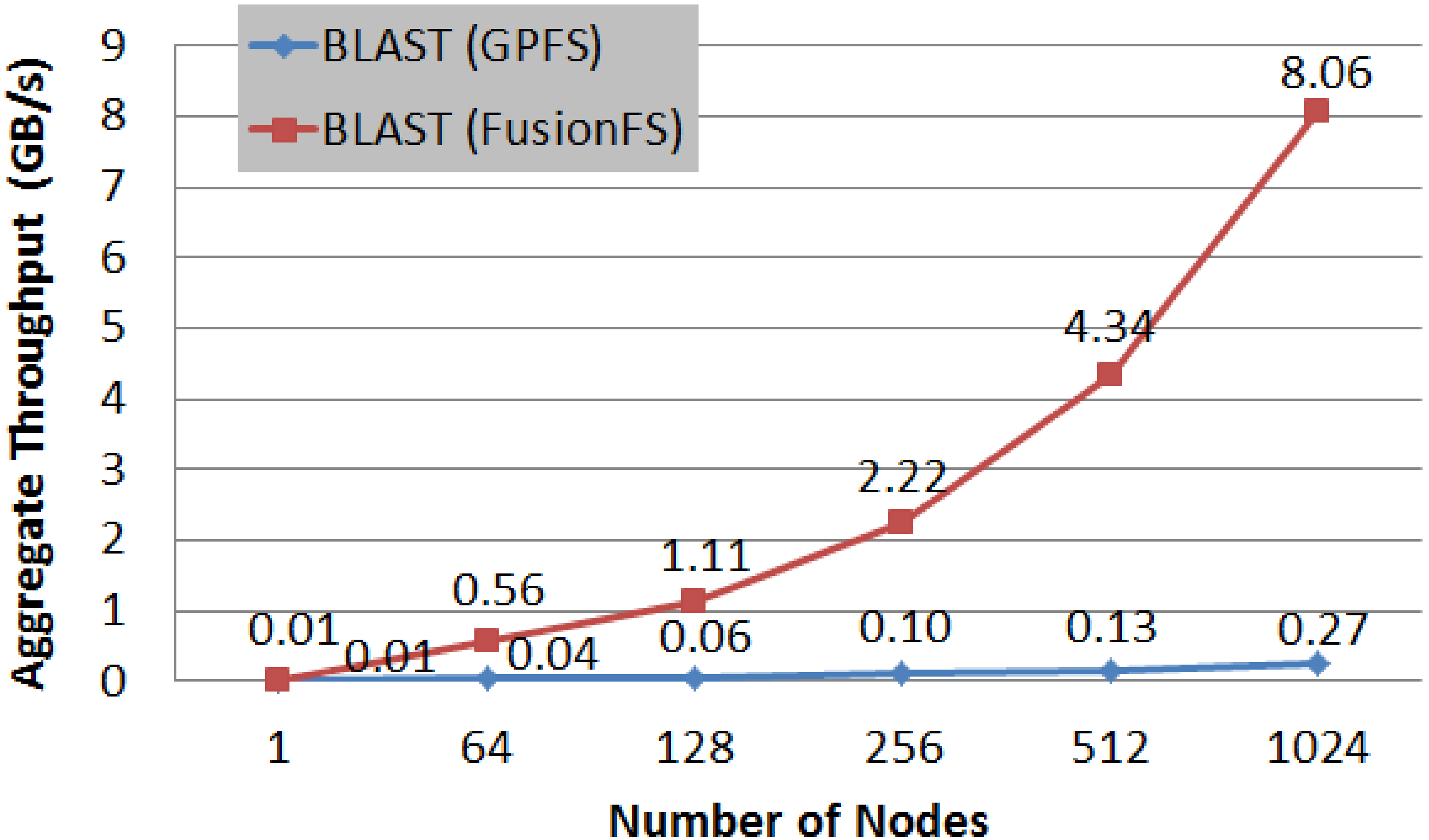
System Efficiency

- FusionFS's sustainable efficiency:



Data Throughput

- FusionFS is orders of magnitude higher

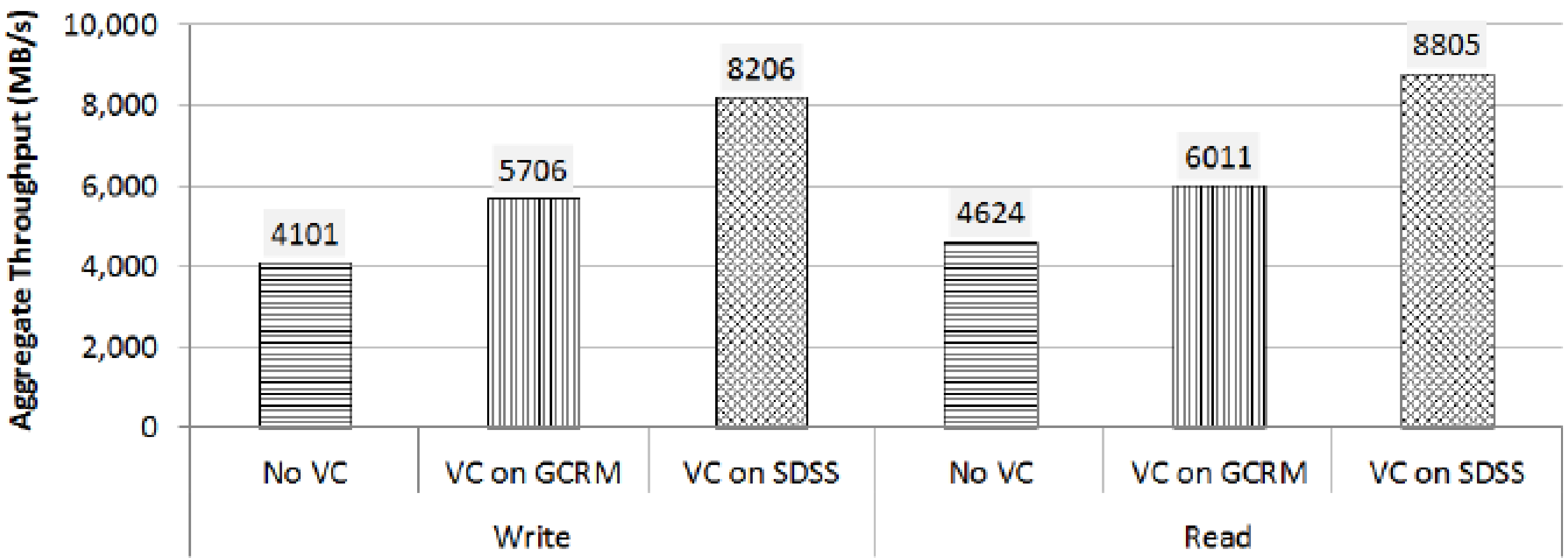


Virtual Chunks in FusionFS

- Setup
  - 64-node Linux cluster
    - each node has a FUSE mount point for POSIX and Virtual Chunk module
- Data set
  - GCRM: Global Cloud Resolving Model
    - 3.2 million data entries; each entry has 80 single-precision floats
  - SDSS: Sloan Digital Sky Survey

Result

- Number of virtual chunks: sqrt(n)
- Compression ratio
  - GCRM 1.49 vs. 2.28 SDSS



Outline

- Introduction
- FusionFS
- Results
- **Future Work**

Simulation of FusionFS at Exascales

- Approach
  - Leverage CODES framework to develop a simulator (FusionSim) to simulate FusionFS at
  - Validate FusionSim with FusionFS traces
  - Predict FusionFS performance at Exascales with the Darshan logs

Integrate FusionFS with Swift

- Approach
  - Expose/develop the FusionFS API for Swift
  - Test FusionFS with the Swift applications
  - Ultimately an ecosystem from data management to the underlying filesystem as an analogy of Hadoop stack:
    - MapReduce <-> Swift
    - HDFS <-> FusionFS

Questions

?