Los Alamos NATIONAL LABORATORY

Offloading Calculations to Computational Storage Devices: Spark and HDFS

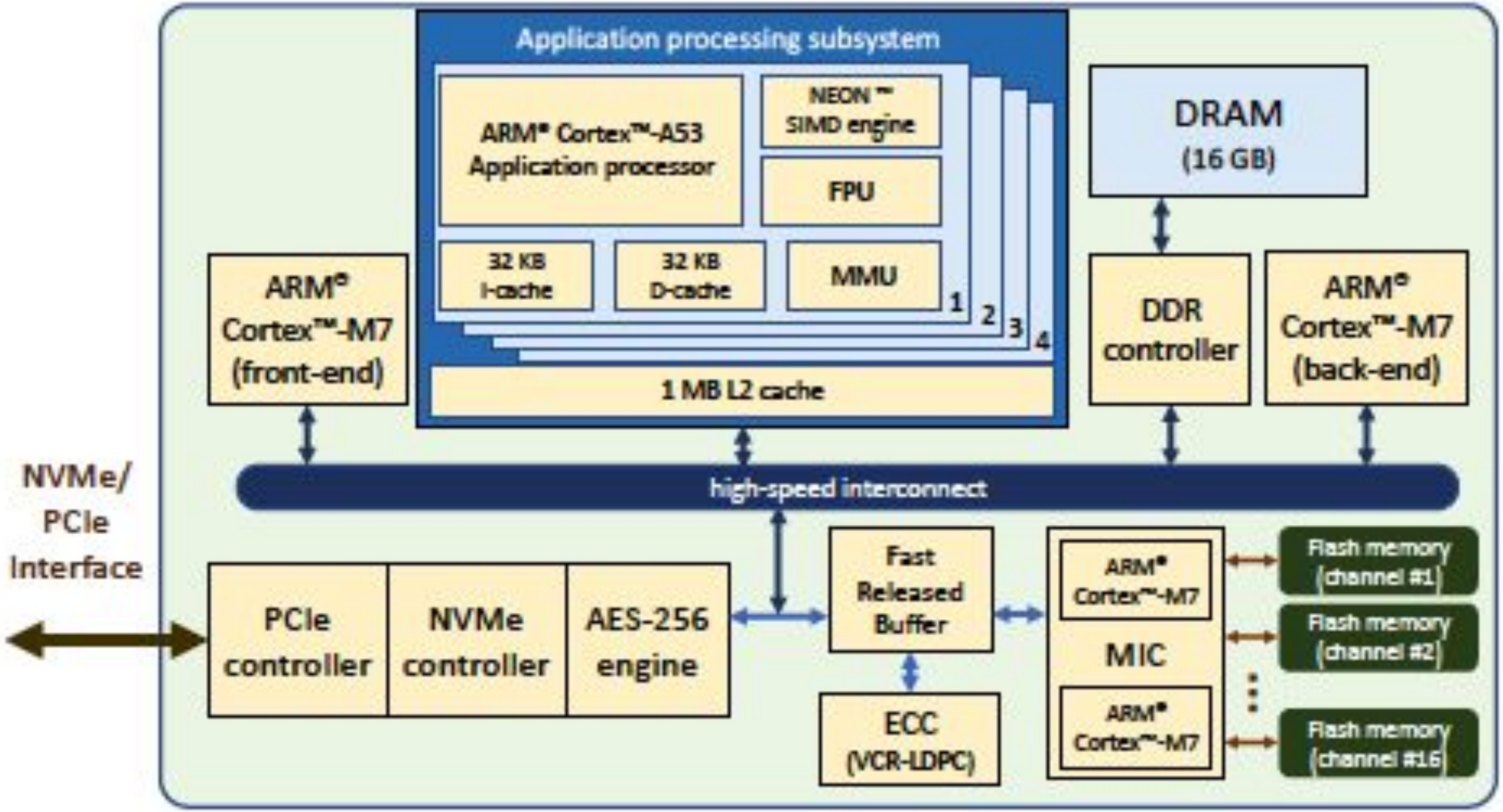
Clyburn Cunningham IV, Justin Goldstein, Warren Hammock (USG), Jacob Janz, Ralph Liu, Mitch Rimerman

Mentors: Shane Goff, Steve Poole, Kevin Bryant (USG)

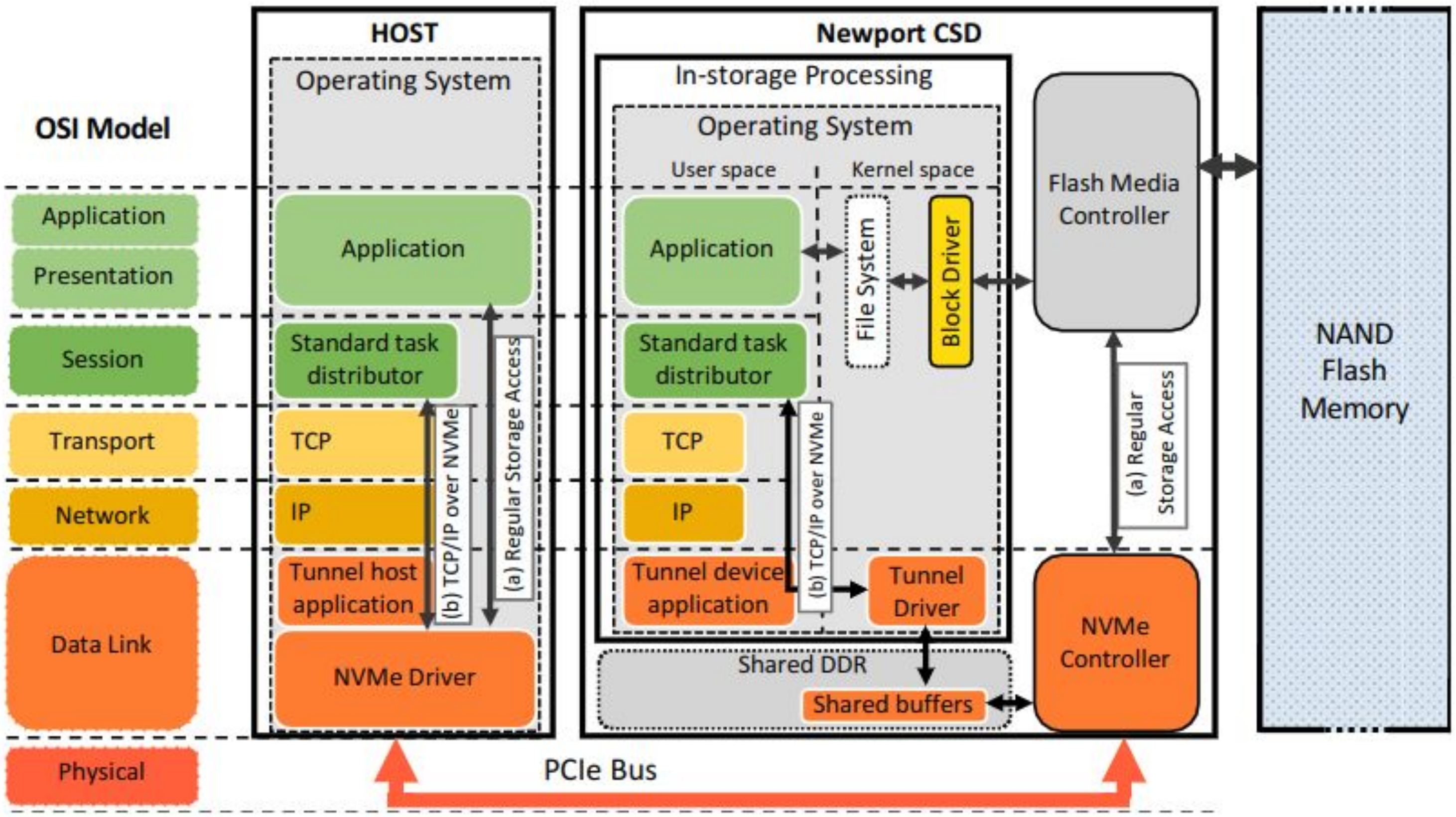
LA-UR-21-28021

Introducing Computational Storage Devices (CSDs)

- Computational Storage → Near-data processing
- CSD → Runs software where data resides
- Potential performance improvement



Application processing subsystem
ARM® Cortex™-A53 Application processor
NEON™ SIMD engine
FPU
32 KB I-cache
32 KB D-cache
MMU
1
2
3
4
1 MB L2 cache
DRAM (16 GB)
ARM® Cortex™-M7 (front-end)
DDR controller
ARM® Cortex™-M7 (back-end)
NVMe/ PCIe Interface
high-speed interconnect
PCIe controller
NVMe controller
AES-256 engine
Fast Released Buffer
ECC (VCR-LDPC)
ARM® Cortex™-M7
MIC
ARM® Cortex™-M7
Flash memory (channel #1)
Flash memory (channel #2)
Flash memory (channel #16)


OSI Model
Application
Presentation
Session
Transport
Network
Data Link
Physical
HOST
Operating System
Application
Standard task distributor
TCP
IP
Tunnel host application
NVMe Driver
(b) TCP/IP over NVMe
(a) Regular Storage Access
Newport CSD
In-storage Processing
Operating System
User space
Kernel space
Application
File System
Block Driver
Standard task distributor
TCP
IP
Tunnel device application
Tunnel Driver
(b) TCP/IP over NVMe
Shared DDR
Shared buffers
Flash Media Controller
(a) Regular Storage Access
NVMe Controller
NAND Flash Memory
PCIe Bus

Introducing Hadoop and Spark

- Apache Hadoop → Used to store and process large datasets
  - Hadoop Distributed File System (HDFS)
  - Ecosystem includes many useful tools/applications
- Apache Spark → Distributed processing system used for big data
  - Enhances processing performance

Experimental Objective and Design

- Objective → Evaluate the capabilities of multiple CSDs (provided by NDG Systems) using Hadoop Filesystem and Apache Spark
  - Use native Spark libraries, such as SparkSQL and DataFrames, to perform matrix operations on datasets
- Independent Variables
  - # of CSDs → 0, 1, 2, 4, or 6
  - Size of dataset → 1 GB, 5 GB, 10 GB
  - Type of dataset → One large file with all of the data, 10 files, 100 files
- Dependent Variables
  - Job time
  - Execution time
- Constants
  - Operations on dataset

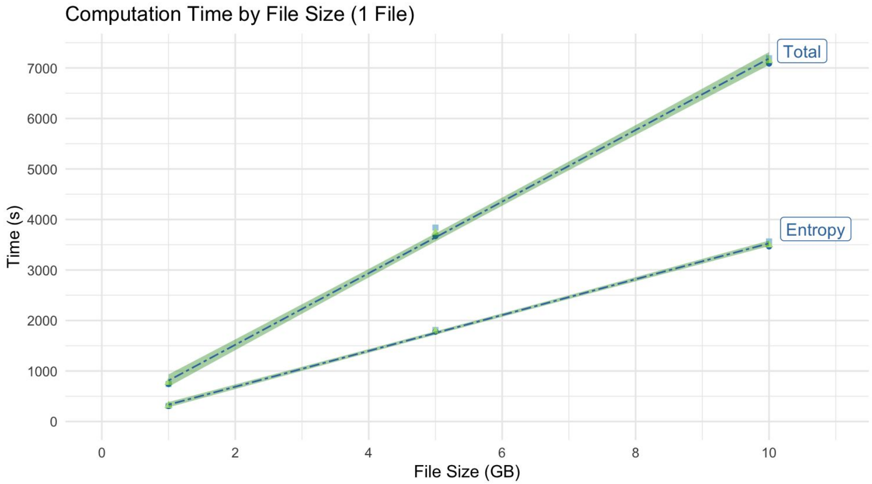
Experimental Design Continued

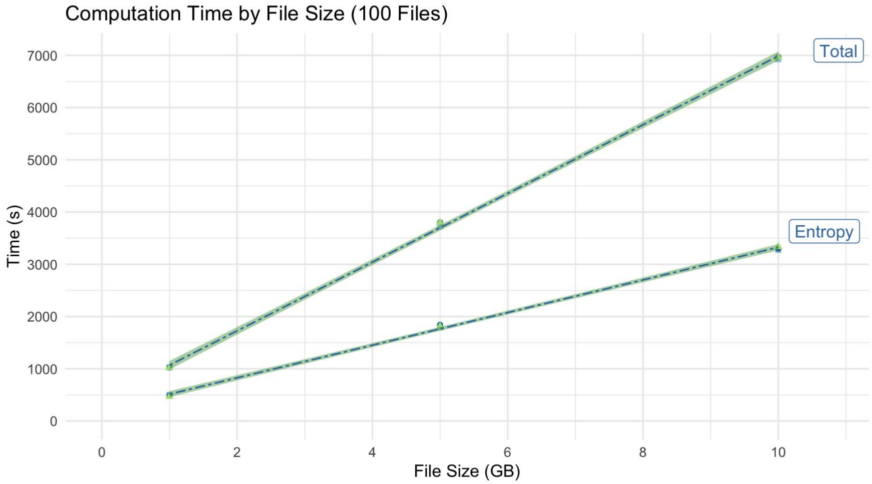
- Ran the experiment 3 times
- Used Trinity sensor data
- Operations
    - Count lines
    - Column operations
        - Sum and average
        - Multiplication and modular arithmetic
        - Mean and standard deviation
        - Compute gram matrix and determinant
    - Measure entropy

Experiment Results

File Size

- Linear Scaling with increased file size holding number of CSDs constant

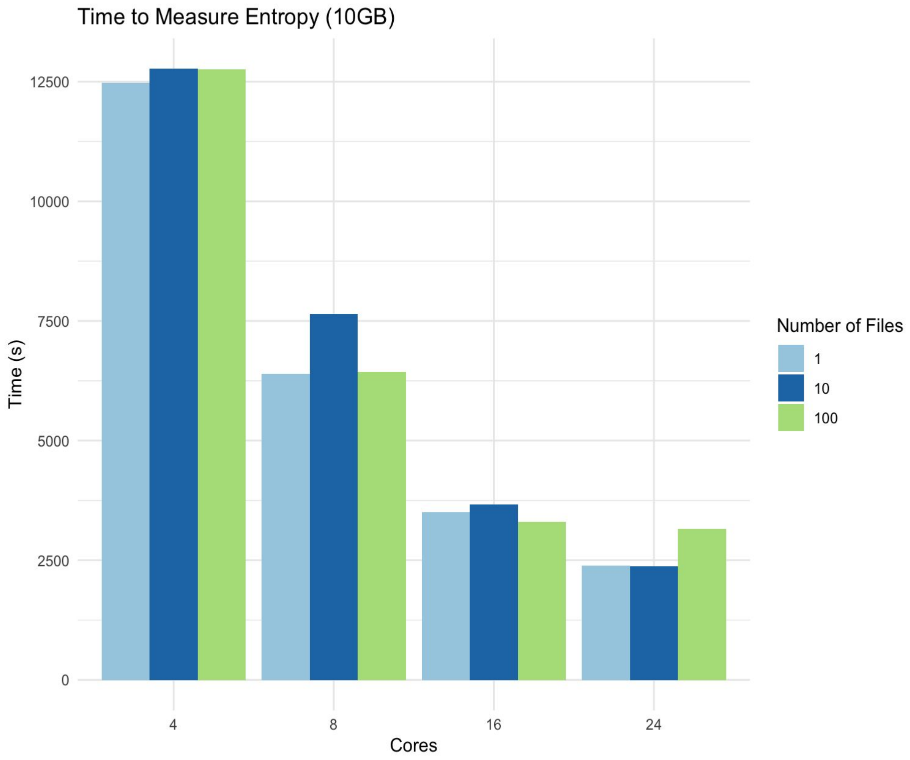




Los Alamos NATIONAL LABORATORY

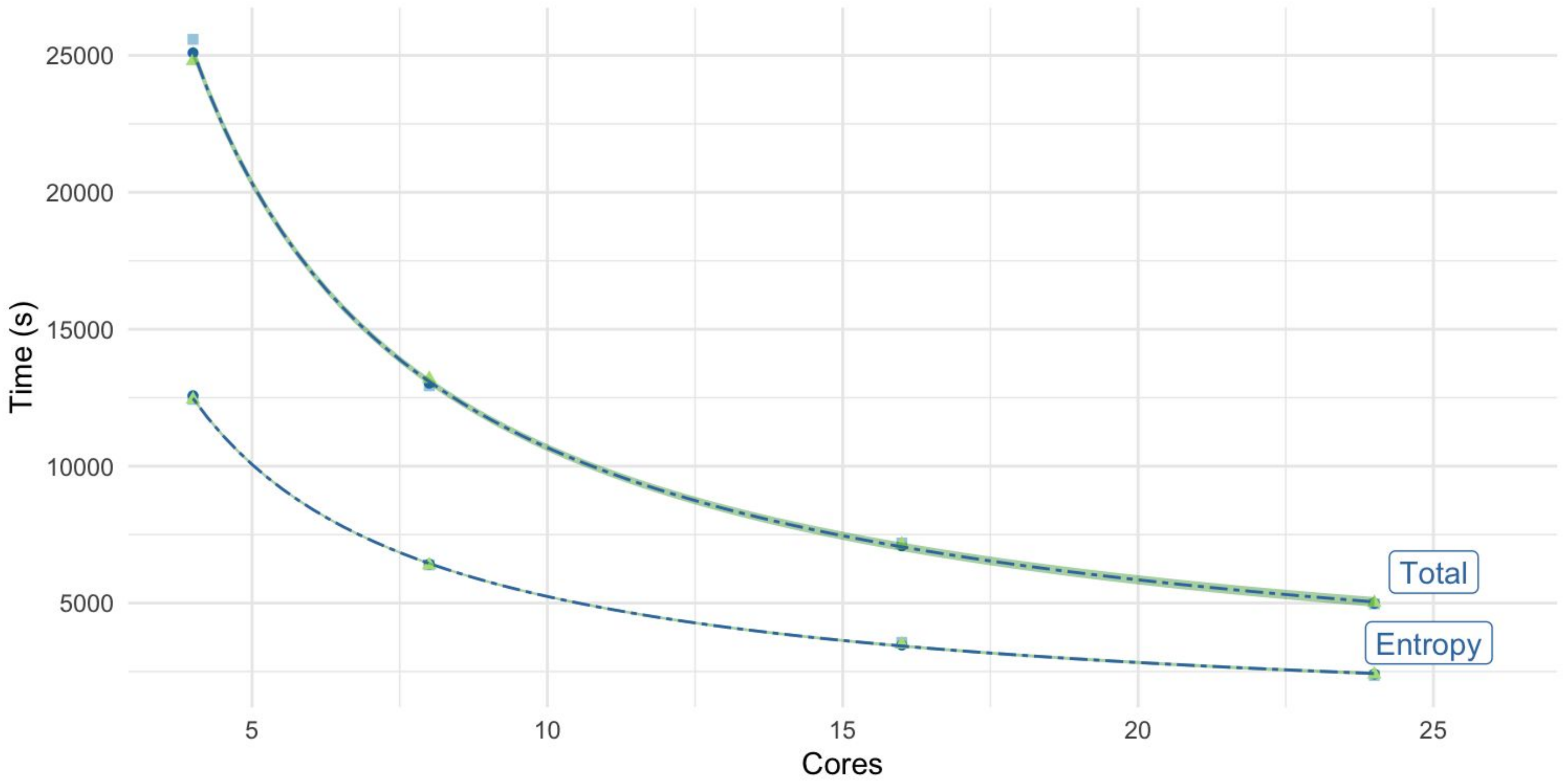
Number of Files

- Increased performance with more CSD's
- Similar observations for different file amounts
- Lesser improvement for more nodes with large amount of files



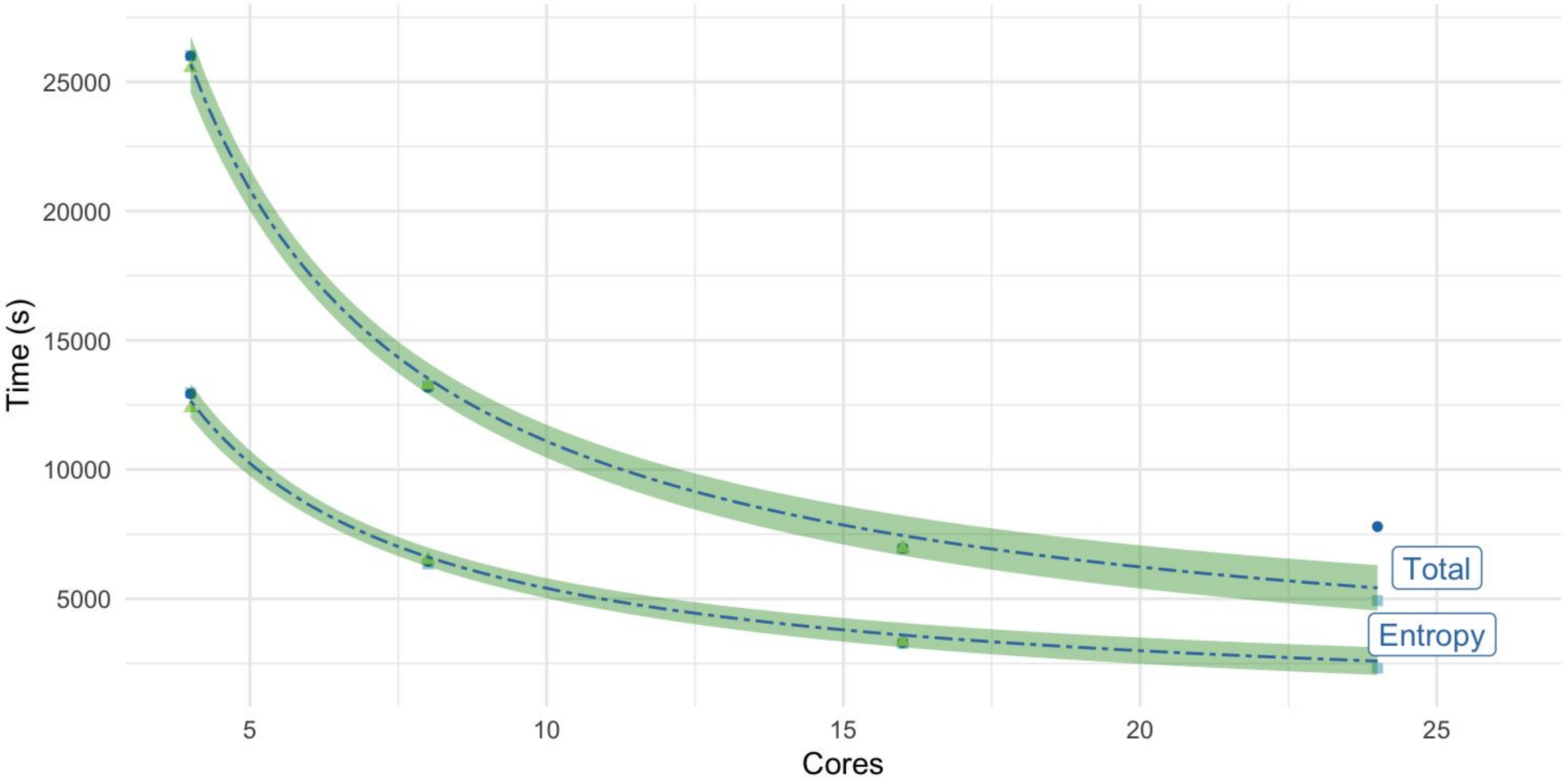
Los Alamos NATIONAL LABORATORY

Computation Time for One 10GB File



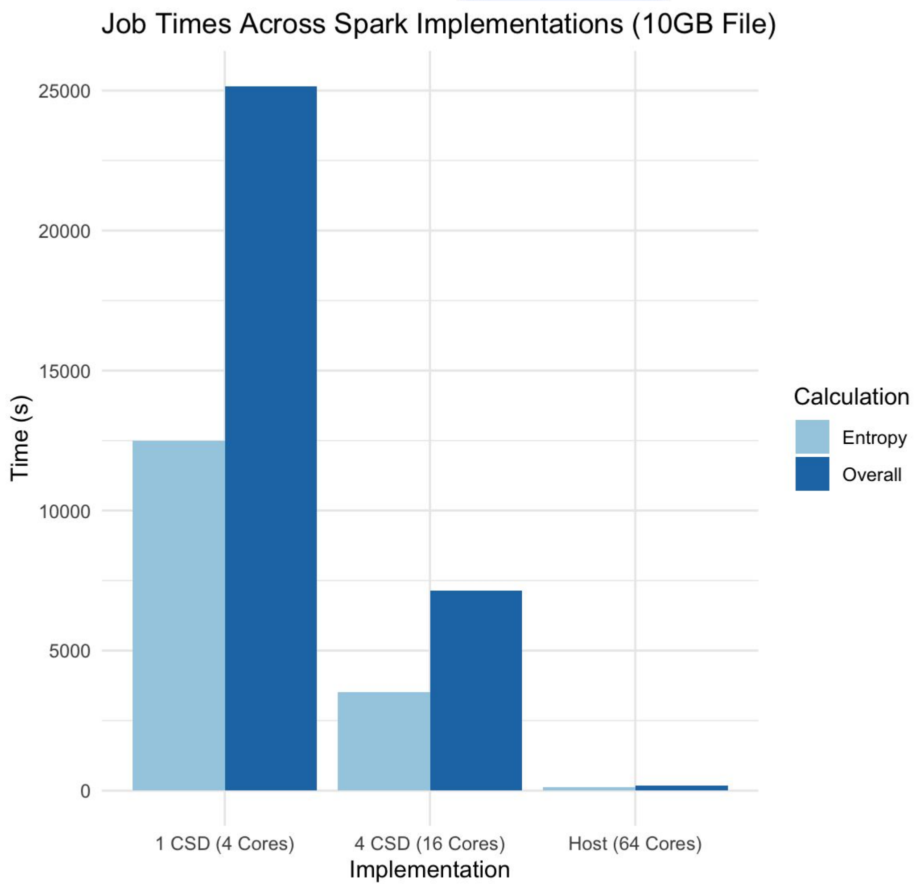
Los Alamos NATIONAL LABORATORY

Computation Time for 100 Files that Sum to 10GB



Comparison to Host

- Much faster on Host
- Assuming uniform scaling, achieving host performance would not be possible with any amount of CSDs



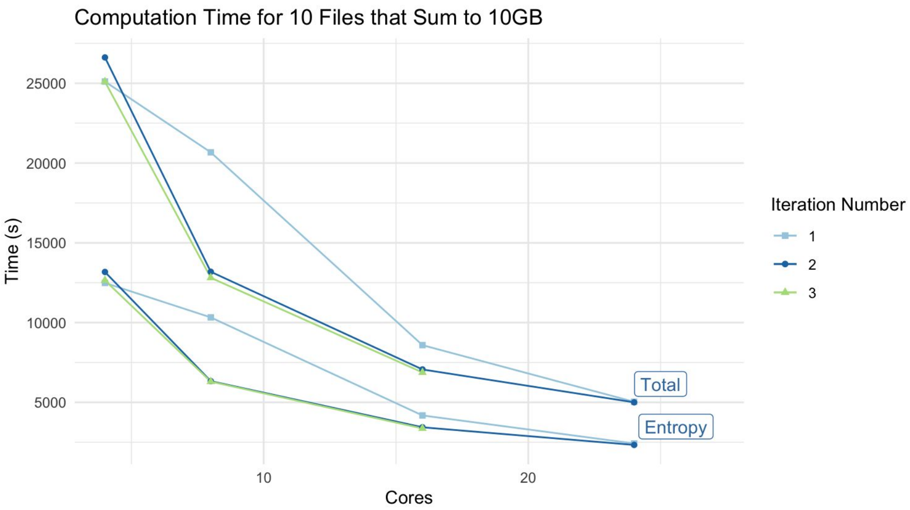
Los Alamos NATIONAL LABORATORY

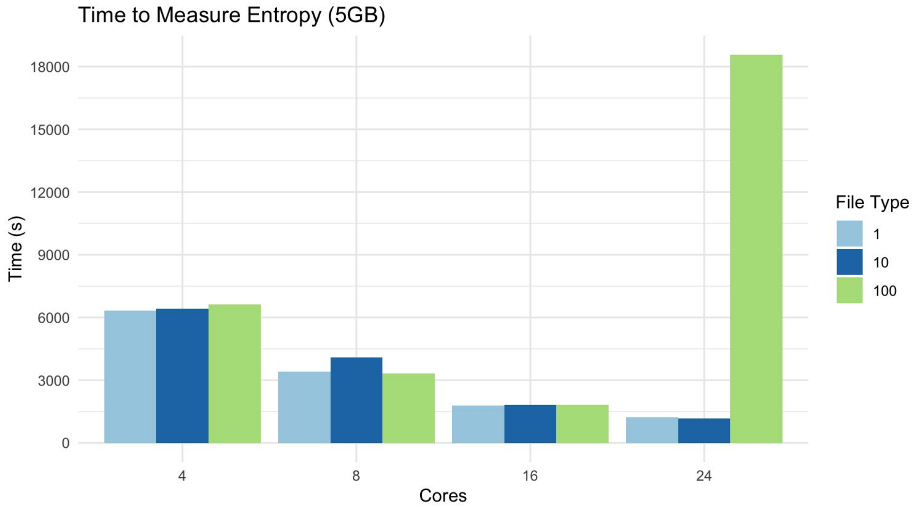
Conclusion

We Observed That

- Ineffective at offloading our operations
- Time v. Size scales linearly
- Time v. Cores scales inversely
- File size v. quantity matters

Important Observations





Los Alamos NATIONAL LABORATORY

Future Work

Scalable, but CSDs are not fast and are unstable

- Drives break often
- If one breaks, all must halt
    - Erase and reinstall Linux

Moving past Spark?

- Removing overhead

Los Alamos
NATIONAL LABORATORY

GEORGETOWN UNIVERSITY

Using Computational Storage Devices: OpenMP/MPI and Charliecloud

Clyburn Cunningham IV, Justin Goldstein, Warren Hammock (USG), Jacob Janz, Ralph Liu, Mitch Rimerman

Mentors: Shane Goff, Steve Poole, Kevin Bryant (USG)

LA-UR-21-28003

Introducing Computational Storage Devices (CSDs)

- Computational Storage → Near-data processing

- Runs software where data resides

- Potential performance improvement
  - Offload tasks from host

Moving On From Previous Experiments

- Originally used Spark and HadoopFS
- Collected interesting results, but this method had its issues
  - Slow
  - Limited Application
  - Too much overhead from Spark abstraction
- Solution? Rewrite our benchmarks without Spark:
  - Serial Python
  - Serial & Parallel C++ (Combinations of OpenMP & OpenMPI)

Why Serial Python?

Able to test on single core with no overhead.

Compare efficiency of different solutions.

- Implementations:
  - SparkDF & SparkSQL → Pandas (dataframes) & Numpy (matrices)
  - Natively written functions (no libraries)
  - Dataframes → Lists

Experiment Results: Running on One CSD

Function	100 MB (s)	200 MB (s)	500 MB (s)	1 GB (s)	5 GB (s)
Count Lines	5.4598 e -5	5.3644 e -5	5.4836 e -5	5.4836 e -5	N/A
Sum of Column	0.1135	0.2281	0.5687	1.2115	N/A
Mean of Column	3.5763 e -5	3.5048 e -5	3.5048 e -5	4.0054 e -5	N/A
Grammarian Matrix: AT*A	17.9477	35.6603	89.483	190.936	N/A
Normalize Column	5.3809	10.6566	25.6559	55.5248	N/A
Compute Mean	0.1138	0.2273	0.5676	1.2162	N/A
Compute Std Dev	3.6919	7.173	17.9616	38.0247	N/A
Count Digits	6.668	6.4407	16.0995	34.4509	N/A
Measure Shannon Entropy	343.624	650.1484	1699.7029	3576.3302	N/A
Total Elapsed Time	6.8031 Minutes	13.1948 Minutes	34.1343 Minutes	71.9462 Minutes	N/A

Where to Go From Python?

Python’s Shortcomings

- Running in “parallel” is less than ideal in native Python

Using Python’s Multithreading Libraries?

- Typically accelerates one machine
- C++ implementation would be more thorough

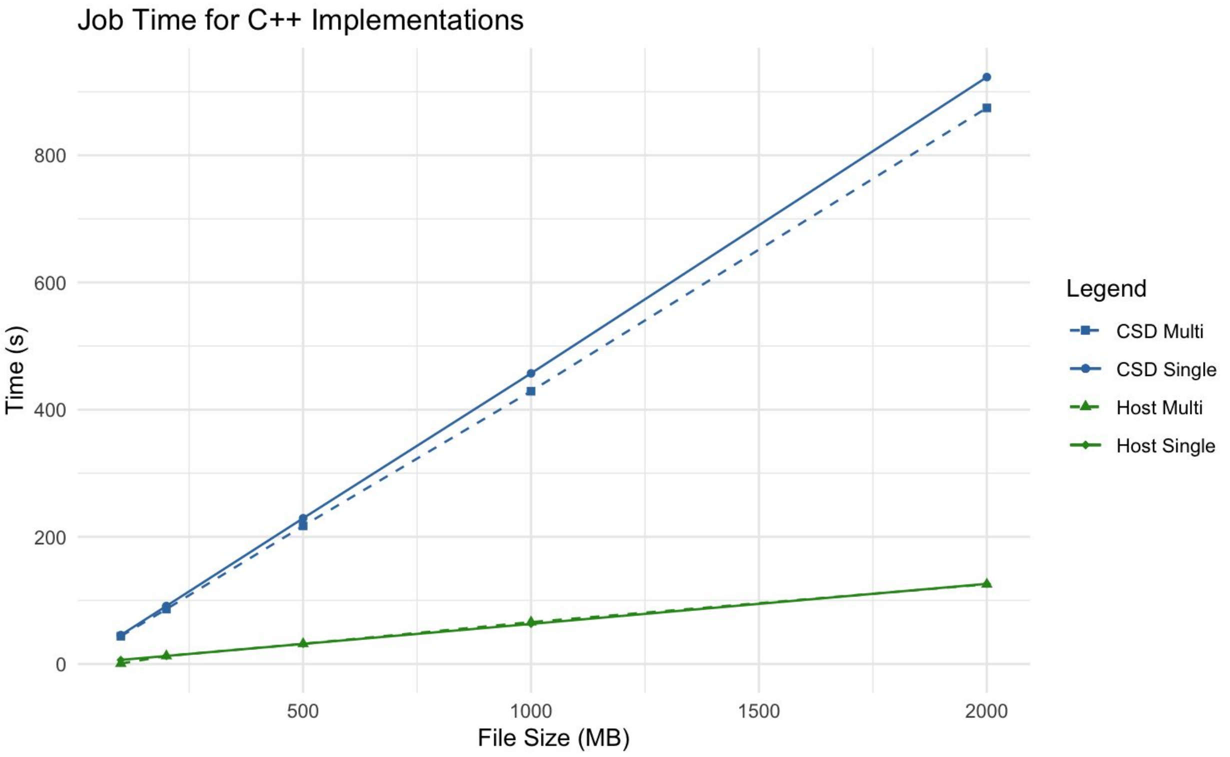
Duplicating Spark Tests in C++

- C++ is "lower level" than Pyspark or basic Python
  - Lets us get a better understanding of CSDs baseline performance
- Basic C++ Implementation is a reimplemented version of our Spark program, with a single-threaded and a multi-threaded version using OpenMP

```cpp
// get sum of normalizedVector
auto sumNormVectTimeStart = std::chrono::high_resolution_clock::now();
int normalizedVectorSum = 0;
for (int i = 0; i < normalizedVector.size(); i++)
{
    normalizedVectorSum += normalizedVector[i];
}
auto sumNormVectTimeEnd = std::chrono::high_resolution_clock::now();
std::cout << "Sum of third row normalized is: " << normalizedVectorSum << std::endl;
```

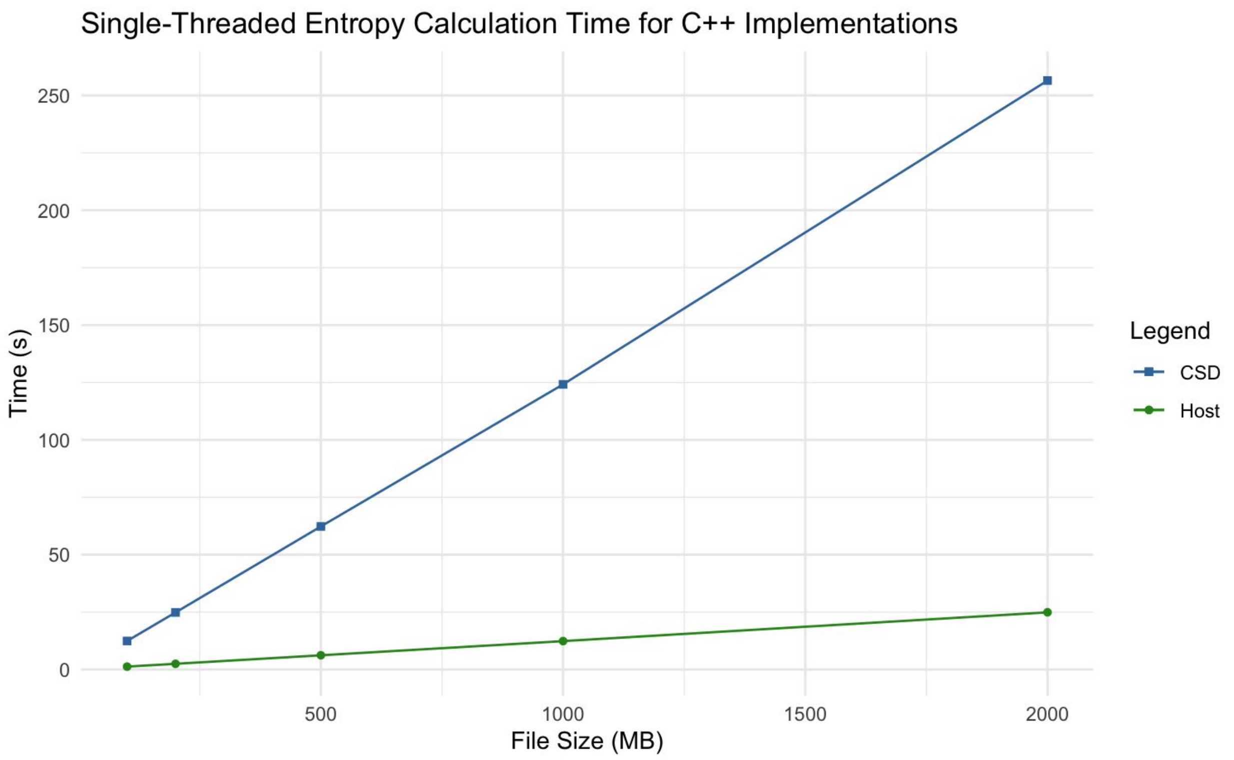
```cpp
// get sum of normalizedVector
auto sumNormVectTimeStart = std::chrono::high_resolution_clock::now();
int normalizedVectorSum = 0;
#pragma omp parallel for
for (int i = 0; i < normalizedVector.size(); i++)
{
    #pragma omp atomic update
    normalizedVectorSum += normalizedVector[i];
}
auto sumNormVectTimeEnd = std::chrono::high_resolution_clock::now();
std::cout << "Sum of third row normalized is: " << normalizedVectorSum << std::endl;
```

Results

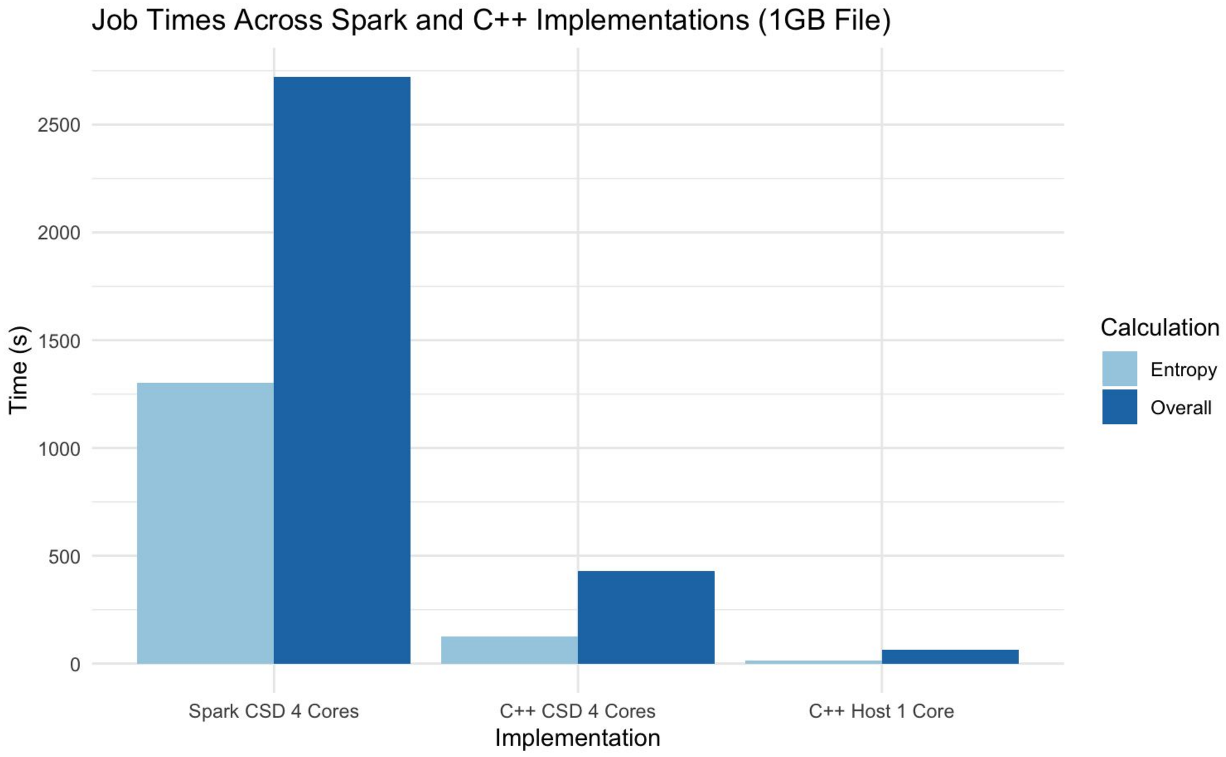


Job Time for C++ Implementations
Time (s)
0
200
400
600
800
500
1000
1500
2000
File Size (MB)
Legend
CSD Multi
CSD Single
Host Multi
Host Single

Results contd.



Results contd.



C++ Conclusions and Thoughts

- Compared to Spark and Python, C++ implementation is *a lot* faster
  - Caveat: an expert with Spark or Python would likely be able to improve the performance of those implementations
- Computational power of our CSDs seem to be much lower than the host machine
  - Using all 4 cores of a single CSD, the job takes ~6.8x longer than using just one core on the host machine.
  - Host also seems to scale better with increasing file size
- Resulting Question: When, if ever, would it make sense to use CSDs for compute rather than a much-faster host?

Host (1.5GHz) and CSDs (1GHz)

Host: 128GB RAM (8GB swap)

Architecture:	x86_64
CPU(s):	64
Thread(s) per core:	2
Core(s) per socket:	32
Socket(s):	1

CSD (x8): 5.8 GB RAM

Architecture:	aarch64
CPU(s):	4
Thread(s) per core:	1
Core(s) per socket:	4
Socket(s):	1

How to Offload Selected Operations?

Disclaimer: Our test was done using host system and 1 csd node (not the full 8 supported). This analysis applies specifically to the operations used in this experiment.

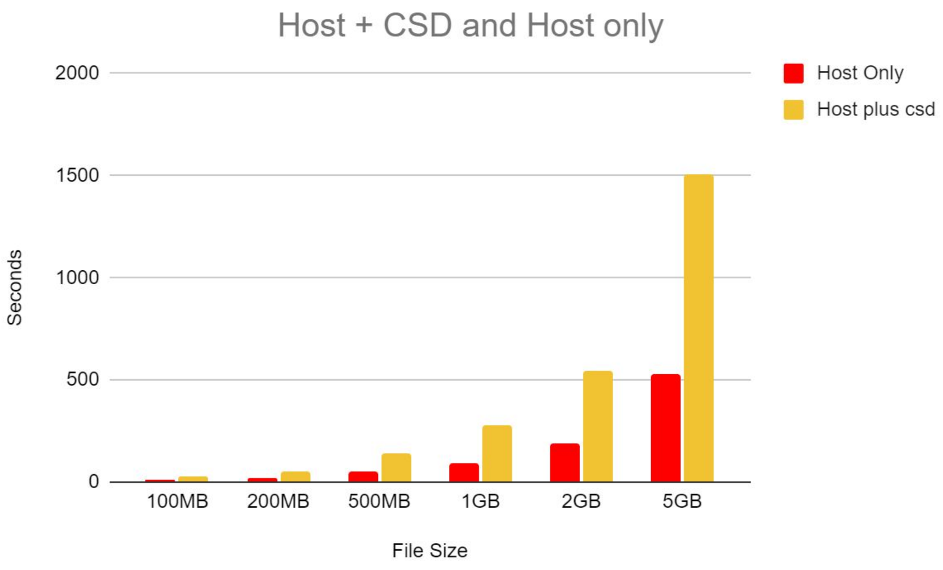
Why use MPI?

Tests: Quickest scalable operations:

- Compute mean (constant time)
- Normalized Compute sum
- Normalized Compute standard deviation
- Normalized Count frequency of digits

When does it make sense to distribute our operations to the CSD? Host and CSD reading in log file

- Tool used: stress-ng --cpu 64 --vm 1 --vm-bytes 95% (stressed RAM and core count)
- Stressed Host tested with mounted CSD storage.
- No Stress CSD tested with mounted CSD storage.



Los Alamos NATIONAL LABORATORY

Can message passing be used to decrease csd vector build time

Issue:

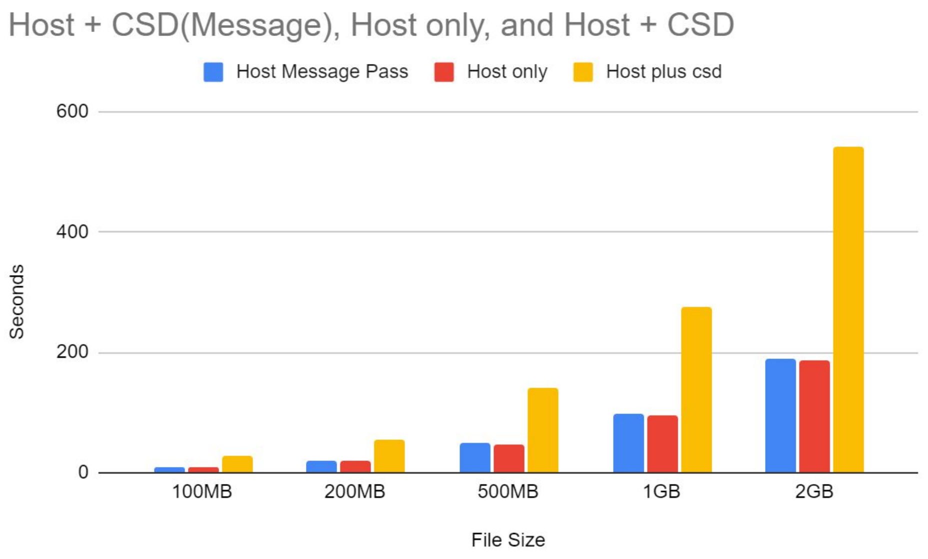
- Most expensive operations for the CSD was to read file and build vector.
- Host completes those operations in 5.86(s)(stressed) 3.91(s)(no stress)
- CSD completes those operations in 23.75(s)

Test:

- 100MB/200MB/500MB/1GB/2GB log file.
- The host reads file from CSD storage and creates vector. Host will then message pass vector to csd.
- See if there is an decrease in overall time for csd to complete its operations.
- Additional parameter for mpirun --mca btl_tcp_if_include flannel.1 (includes interface)

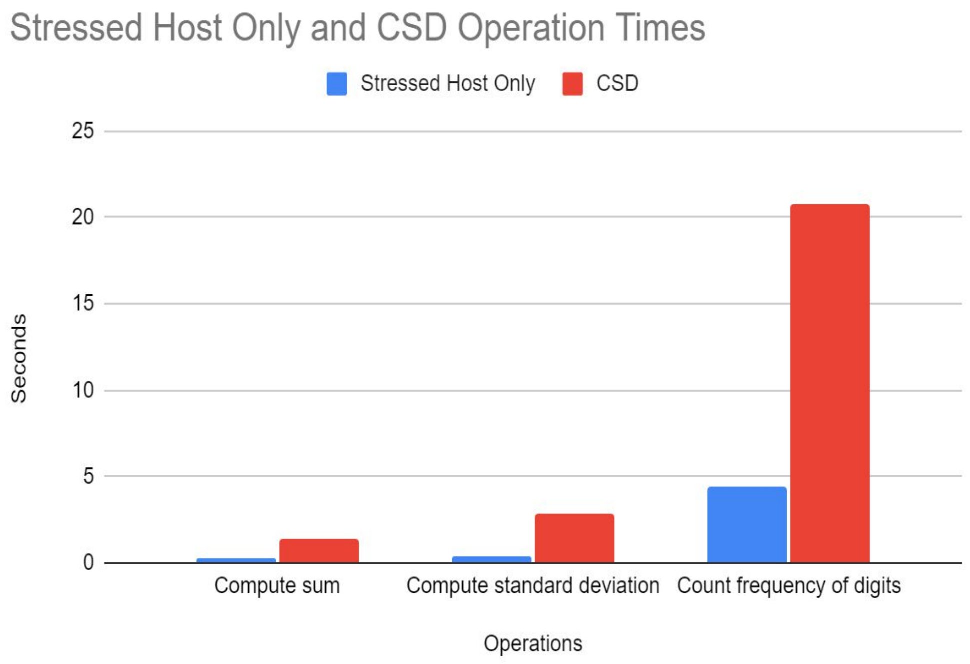
Offloading operations passing vector to CSD

- Tool used: stress-ng --cpu 64 --vm 1 --vm-bytes 95% (stressed RAM and core count)
- Stressed Host tested with mounted CSD storage.
- No Stress CSD tested with mounted CSD storage.



Los Alamos NATIONAL LABORATORY

Still does not make sense on a per operation comparison



- Operation costs on a 1GB data log.
- Even after vector is in memory, the csd still executes the operation significantly slower than the stressed host test.
- Future work needs to be done with a focus on small operations. CSDs seem to be of more use in smaller operations on smaller files.

Los Alamos
NATIONAL LABORATORY

Future work for passing information

- Further investigate MPI’s usage for communication.
- Need to develop a better way for host and csds to share storage.
- Create a pooled storage for CSDs, possibly ZFS.
- Data filtering (encrypt/decrypt)

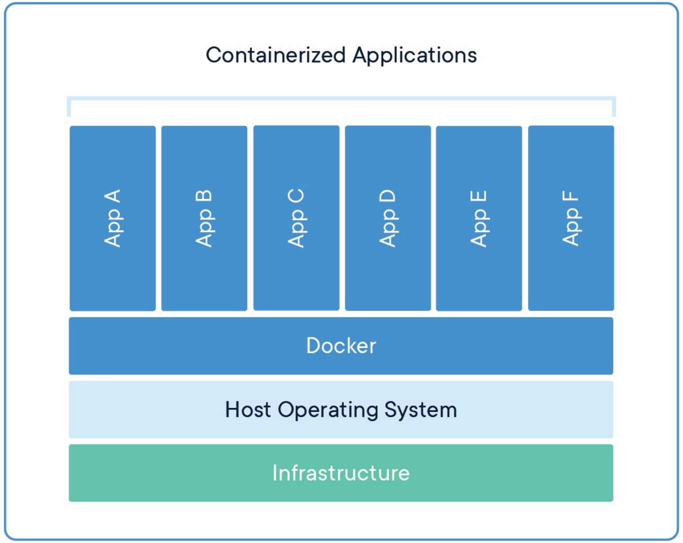
CSDs with Charliecloud

About Charliecloud
Background on experiments
Analysis of results

Charliecloud
SMALL SOFTWARE DELIVERING BIG FLEXIBILITY TO HPC

Los Alamos NATIONAL LABORATORY

About Charliecloud



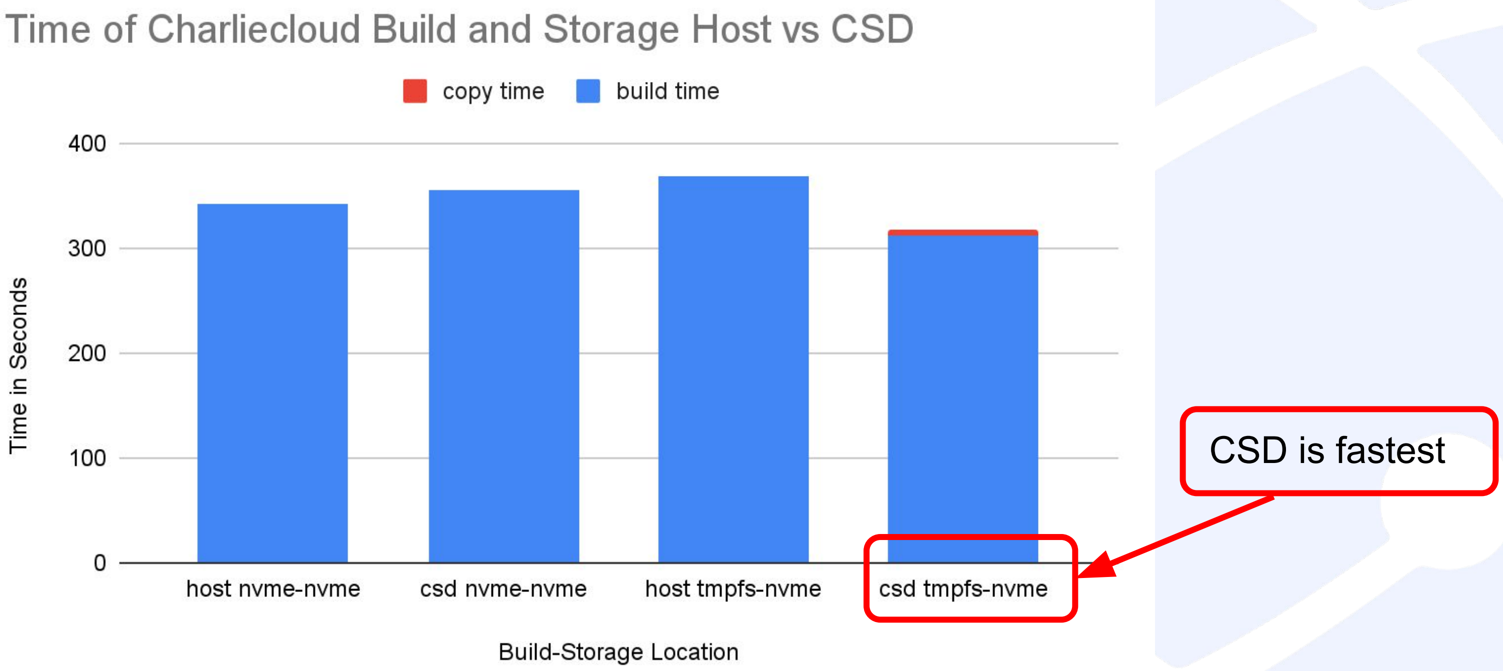
- Bring your own software stack
  - Containers (Isolated Environments)
  - Container images (Container blueprints)
    - Code
    - System tools
    - Runtime
    - Settings
- Charliecloud Images
  - Few permissions
  - Minimally affect cluster resources

1st Experiments

	Build Location	Storage Location
Host	Host NVME	Host NVME
	Host tmpfs	Host tmpfs
CSD	CSD NVME	CSD NVME
	CSD tmpfs	CSD tmpfs

- Typical workflow: Build image on a compute node
  - (Inefficient!)
- Research Question: What is the best filesystem to store user images on in a cluster environment?
  - Compare small CSD to our big host (Host == computer)
  - Compare our big host to LANL's Fog host (later)

CSDs out-perform host on small image?



Los Alamos NATIONAL LABORATORY

2nd Experiments

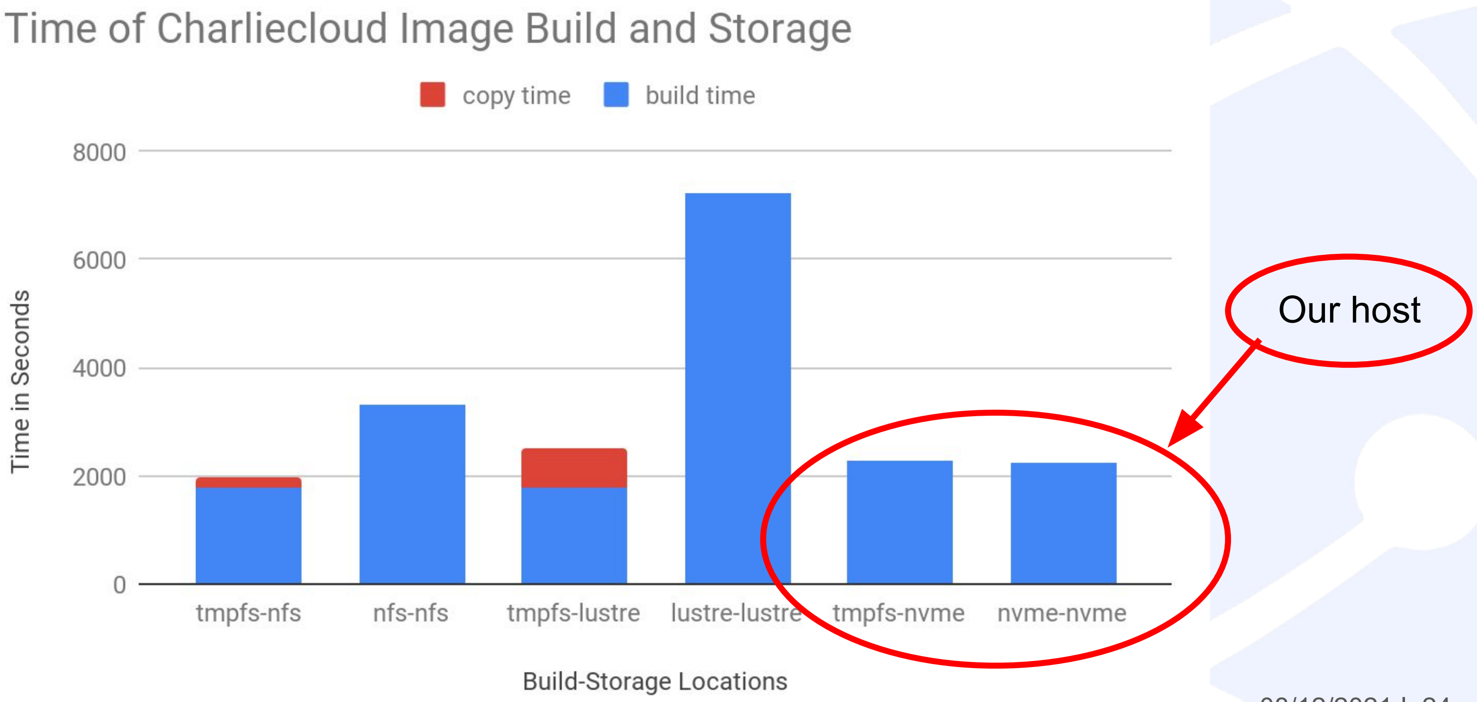
Build Location	Storage Location
NVME	NVME
tmpfs	NVME
NFS	NFS
LUSTRE	LUSTRE
tmpfs	LUSTRE
tmpfs	NFS

- How does our host with NVME compare to a LANL production setup?
  - Lustre on Fog vs
  - NFS on Fog vs
  - NVME on our host

Our host

Los Alamos
NATIONAL LABORATORY

NVMe vs Other Filesystems



Conclusions and Next Steps

- Future work on variability across runs
  - Implications for scaling to larger container image builds
- Viability of CSDs for medium term storage (Stability!)
- Memory restrictions of our CSDs for building large images
- Potential use case for CSDs with Charliecloud
  - Envisioning a new user workflow

Overall times to complete all operations per data size

Method	Spark		Python	C++			
	1 CSDs	8 CSDS	Serial on CSD	Serial on CSD	Multithread on CSD	Host stressed and CSD	Host stressed
100MB	N/A	N/A	408 s	45.47s	43.94s	36.83s	9.82s
200MB	N/A	N/A	792 s	90.61s	87.40s	72.51s	19.36s
500MB	N/A	N/A	2,048 s	229.10s	217.09s	181.97s	48.16s
1GB	2759.17s	542.44s	4,317 s	457.74s	432.54s	358.16s	94.61s
2GB	N/A	N/A	N/A	929.60s	870.97s	714.72s	187.58s