PARALLEL CONSTRUCTION OF MERKLE (HASH) TREES

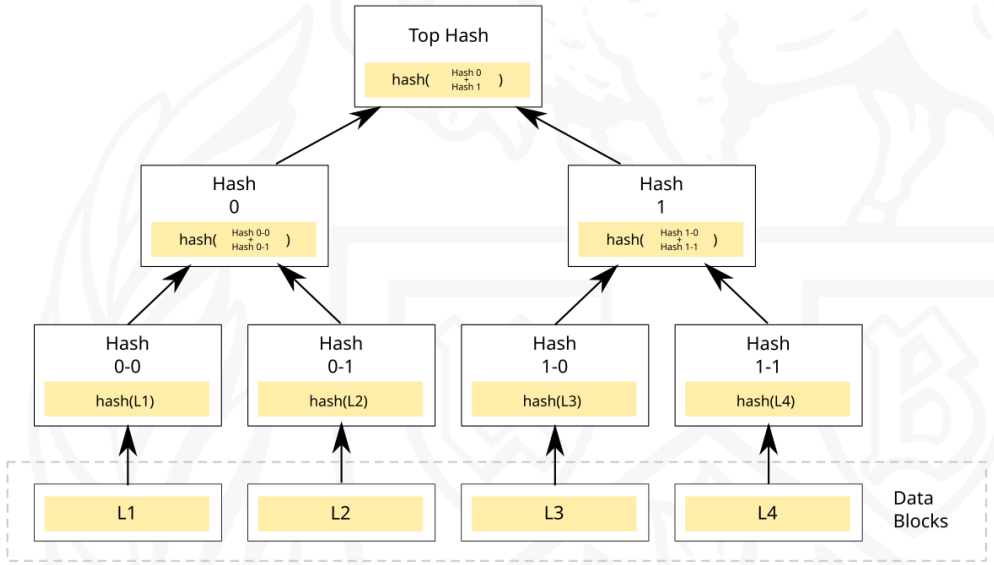
Sam Anderson

CSE 633: Parallel Algorithms

University at Buffalo The State University of New York

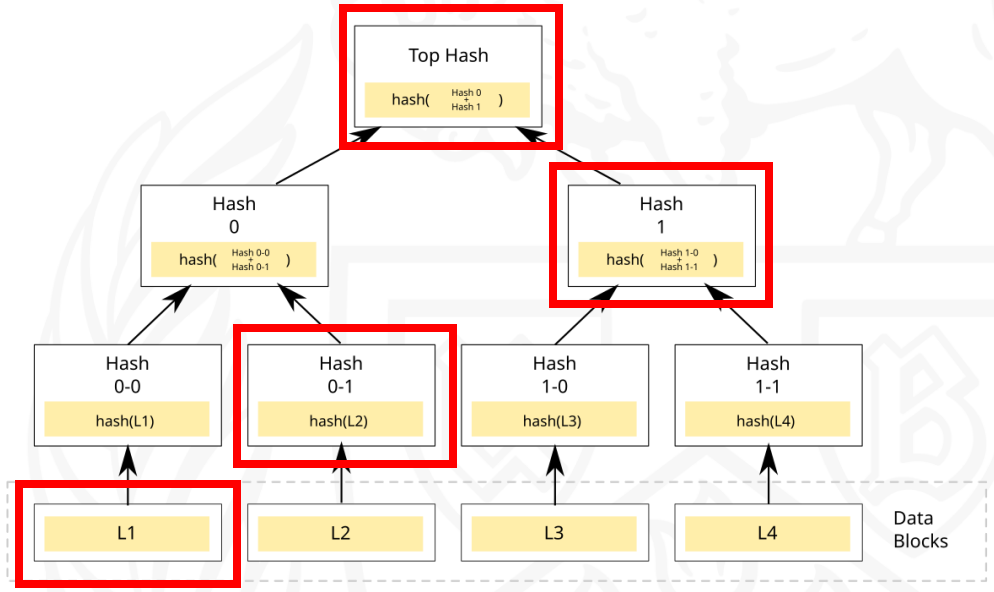
Merkle Tree Overview

- A Merkle tree (or Hash Tree) is a cryptographic data structure used to efficiently and securely verify the integrity of data.
- Composed of leaf nodes (data hashes) and non-leaf nodes (hashes of child nodes), forming a binary tree.
- Each non-leaf node contains a hash of its children
- The top-most node (root) contains the hash of the entire tree
- Allows quick verification of large data with minimal additional data transfer (Merkle proof).



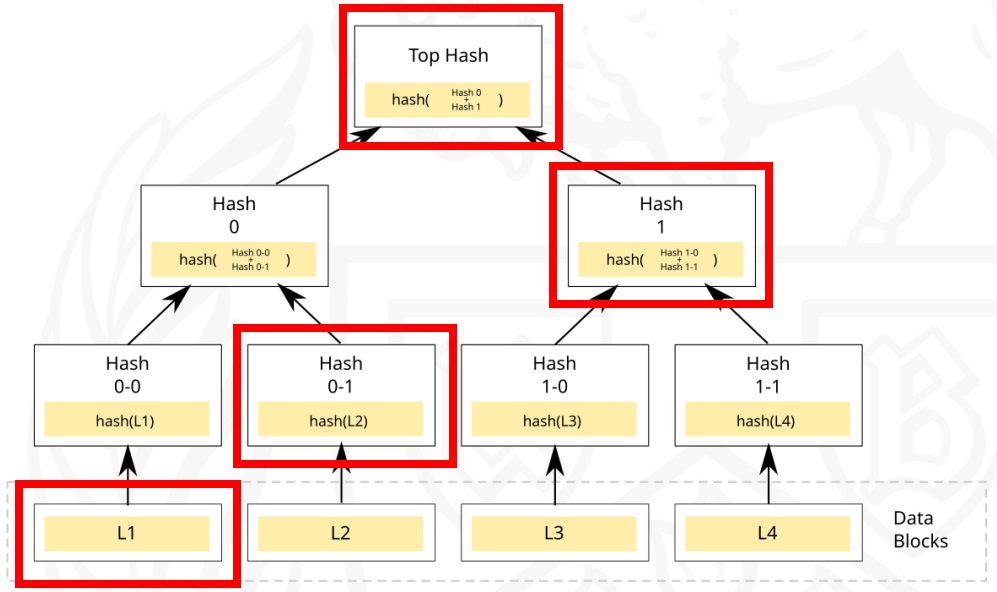
Merkle Proof

- A subset of the Merkle tree that proves a single piece of data (typically a leaf node) is part of the tree.
- Includes only the hashes needed to prove that a particular leaf node, when combined with its siblings, results in the root hash of the Merkle tree.
  - Ex: Proof L1 is part of the tree



General Applications

- Given a Merkle proof, end users can verify the integrity of a data block. Broad applications in:
  - Cryptography/Blockchain
    - Ex: Verifying transactions within a block
  - Distributed Systems
    - Ex: Proving a record is in a database
  - String/Text Processing
    - Ex: What we are doing here!



Application to Checksums

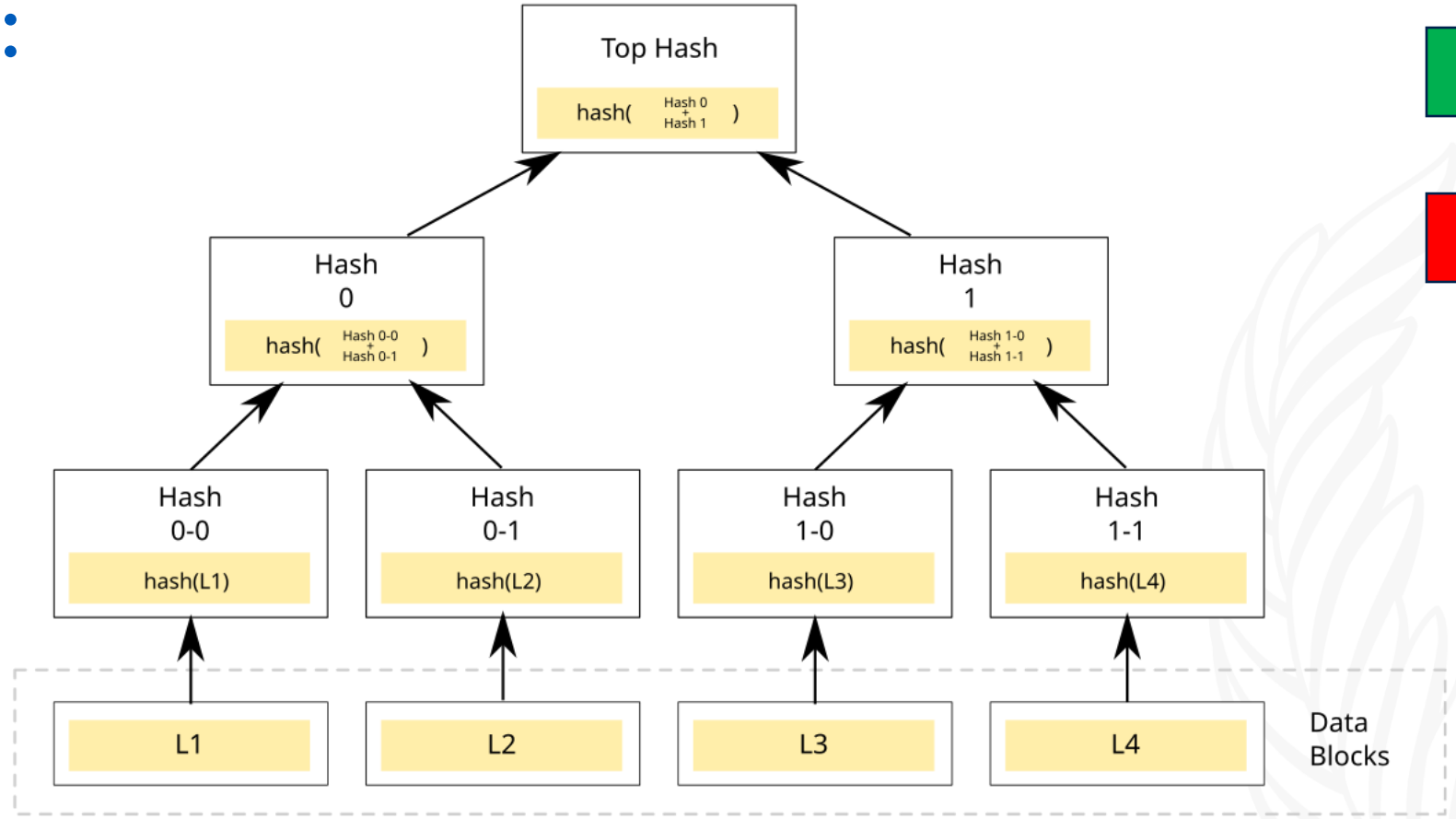
- Suppose you want to transfer a large text from some source and verify the information received is correct
- Traditionally, checksums solve this problem
- Problem? Depending on the hash function and the size of the text:
  - Traditional checksum may be expensive to compute
  - Not easily parallelizable
  - Requires full retransmission on error
    - Worse, probability of error increases with length

Application to Checksums (cont.)

- Given a full Merkle Tree, the end user can prove* that the file was received uncorrupted.
  - Build second Merkle Tree from received data
  - Compare root hashes
- In case of transmission error, the end user can identify which blocks specifically contain errors.

* Assuming Merkle Tree is obtained without corruption and ignoring hash collision.

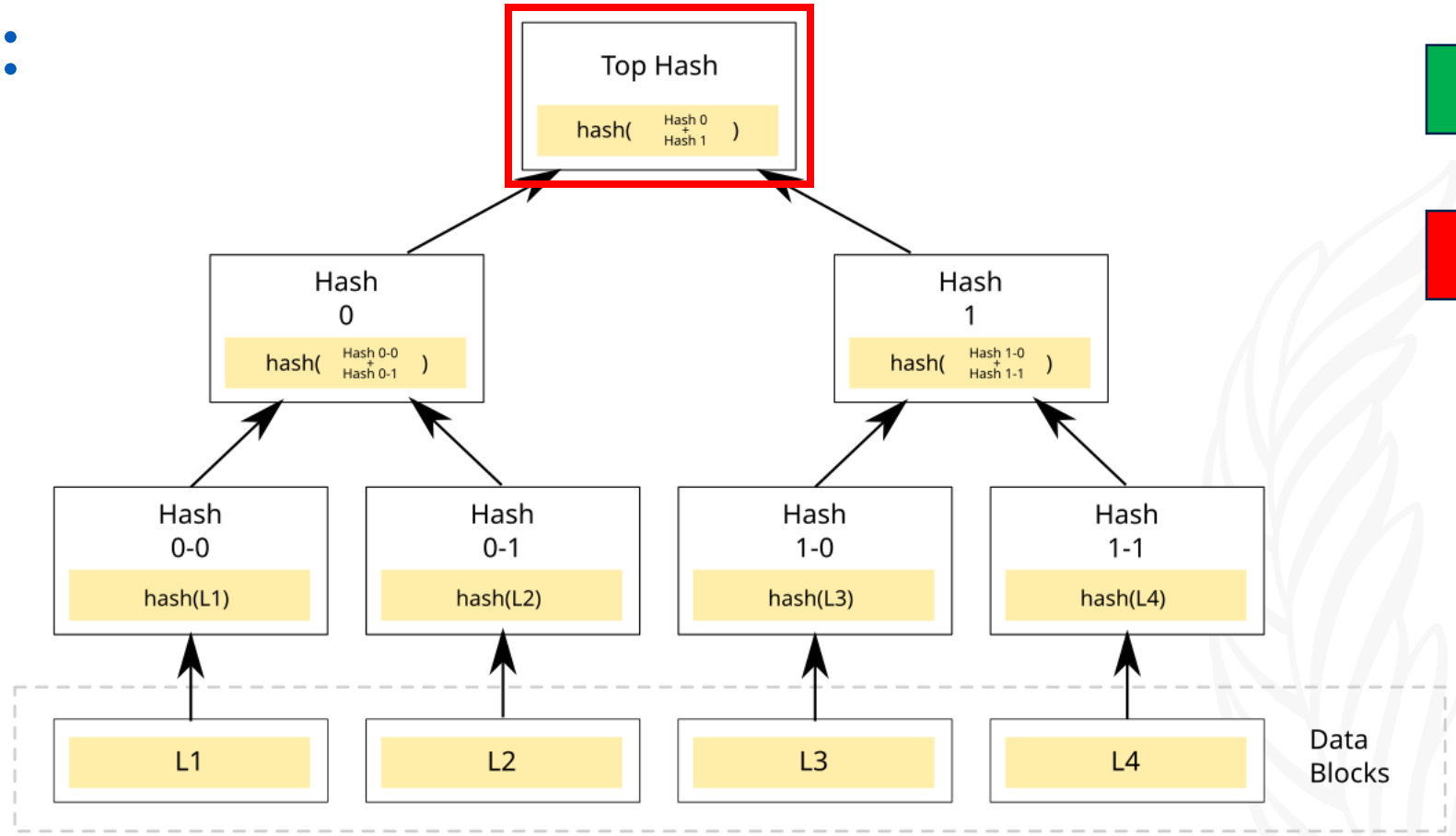
Ex:



Correct

Incorrect

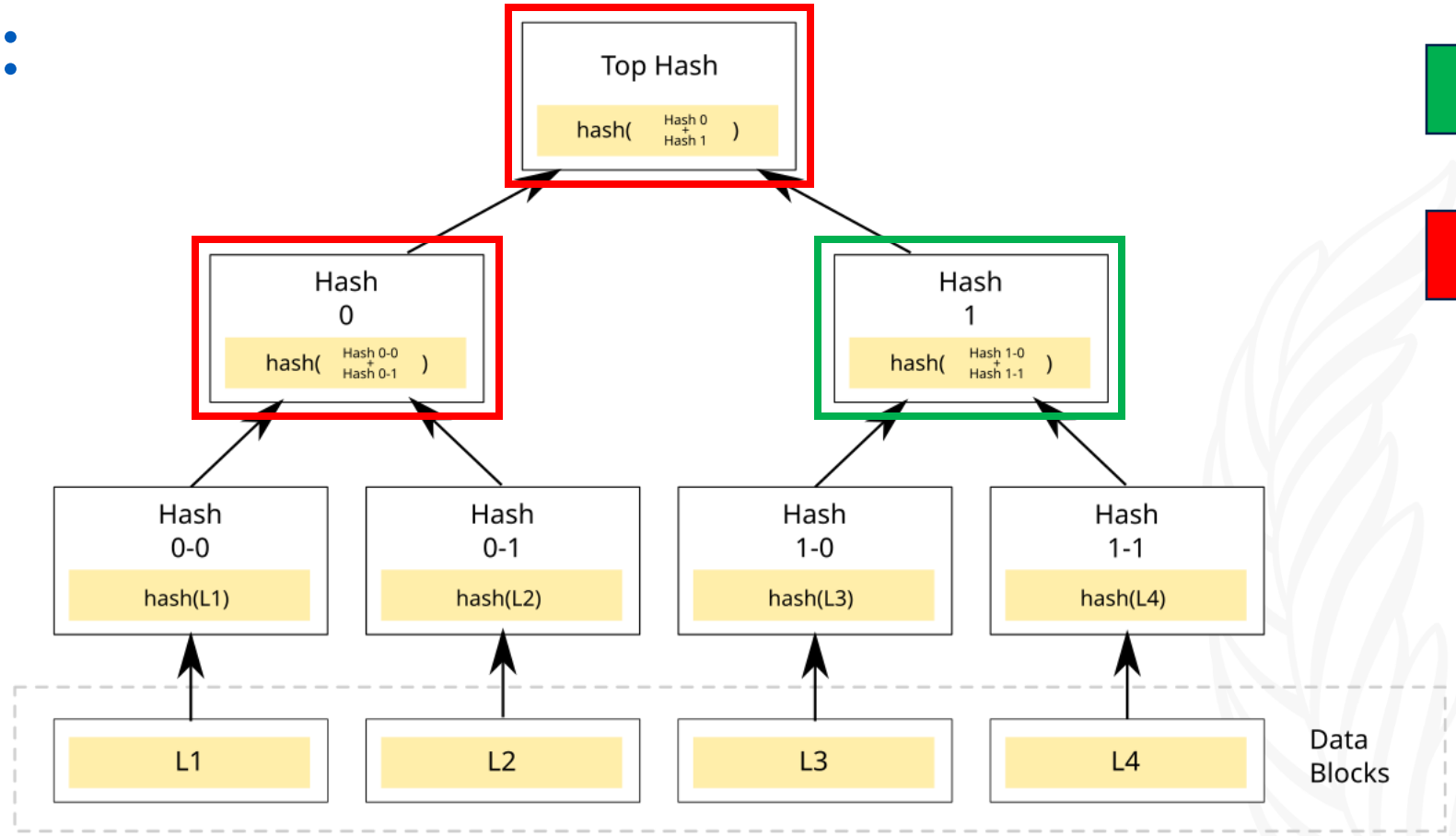
Ex:



Correct

Incorrect

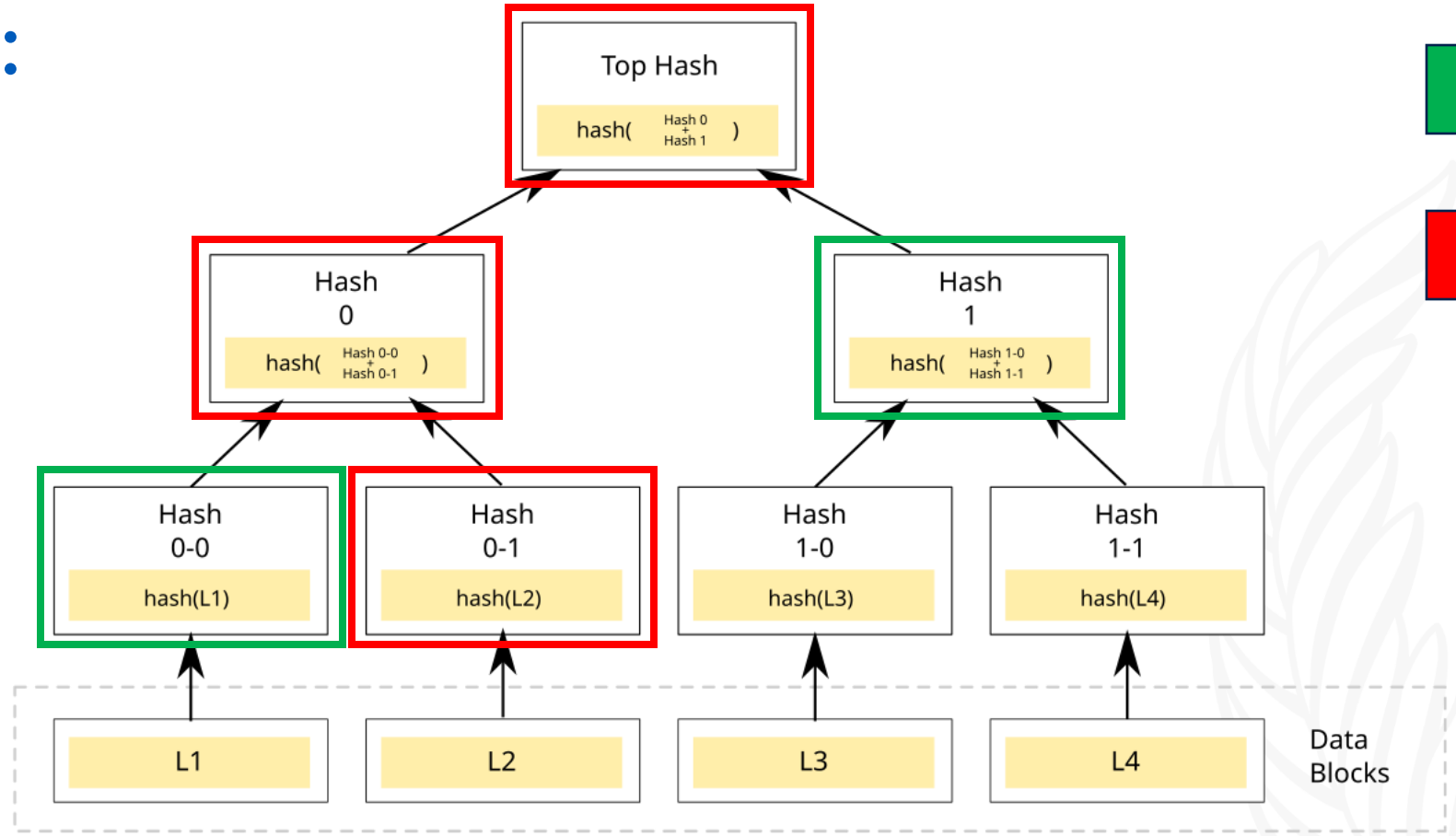
Ex:



Correct

Incorrect

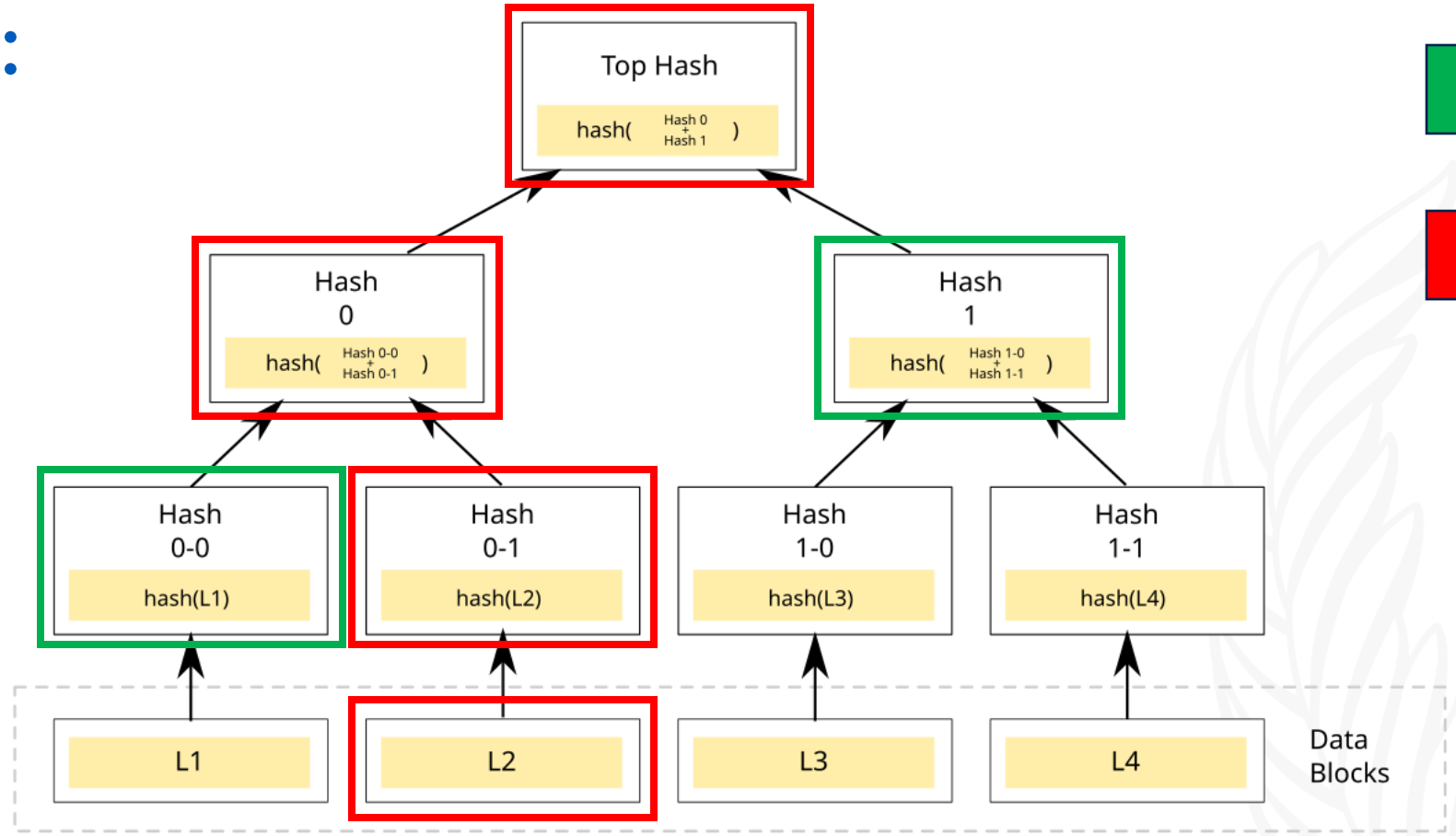
Ex:



Correct

Incorrect

Ex:



Correct

Incorrect

PARALLEL APPROACH

Algorithm Overview

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

- Logically divide text into n blocks of constant size
  - Ex. 1 MiB
  - Size must be unrelated to p
- Compute Rank
- Compute COMM size
- p_i assumes responsibility for blocks $[\frac{ni}{p}, \frac{n(i+1)}{p}]$
  - First n mod p processors take an extra block

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

- Sequential step
- Each processor:
  - Reads a block of its partition
  - Computes hash for that block
  - Stores hash
  - Repeat
- Note: Hash does not *necessarily* matter here (within reason)
  - Using SHA-256

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

- Each processor, p, computes length of locally stored hashes
- Used to determine boundaries between processors

Count of Previous Hashes

- (Exclusive) Parallel Prefix Sum
  - p_i needs to know the total count of hashes stored on processors [p_0, p_{i-1}]

```
long local_hash_count = current_layer.size();
long prev_prefix_count = 0;

MPI_Exscan(&local_hash_count, &prev_prefix_count, 1, MPI_LONG, MPI_SUM, comm);
```

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

Boundary Cases (Excluding final p):

Local Count	Prefix Count	Action
Even	Even	None
Even	Odd	Give and Steal Hash
Odd	Even	Steal Hash
Odd	Odd	Give Hash

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

- Communicate hashes between processors depending on boundary conditions:
  - None: Skip
  - Steal: Take hash from previous active p
  - Give: Give hash to next active p
- Problem: Processors may run out of hashes. Can't just trade with neighbors.
  - Implemented as a second parallel prefix

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

- Repeat sequential step with new set of hashes
- Creates next layer of Merkle Tree

- Repeat loop using new layer of hashes until root is reached

Algorithm Implementation

- Partition text across processors
- Each processor computes hashes for chunks within its local partition of data
- While there is more than one hash across all processors:
  - Collectively compute count of hashes stored on lower-rank processors to determine boundaries
  - Communicate edge hashes across boundaries
  - Compute next layer of hashes
- Processor 0 emits top hash

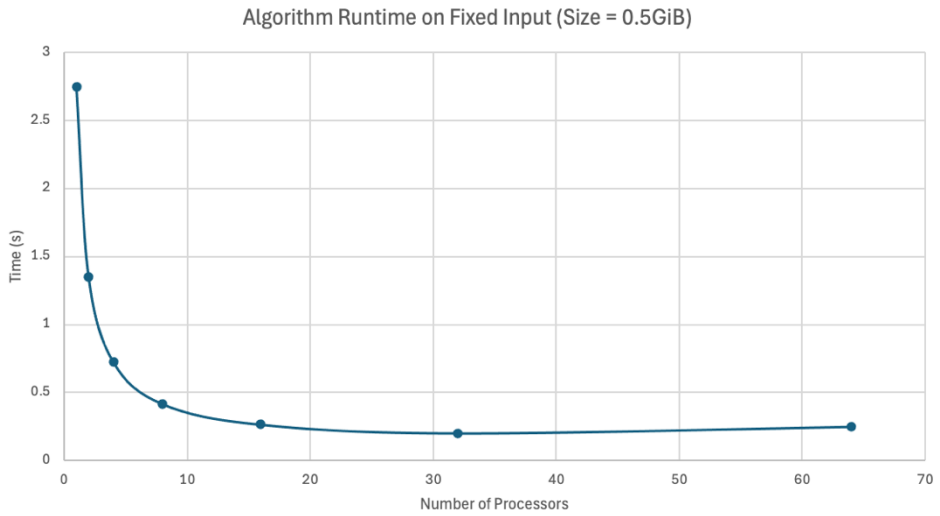
- At each iteration, processors pass values left (to lower ranks)
- p_0 only steals, never gives up a hash
- Root hash eventually stored in p_0

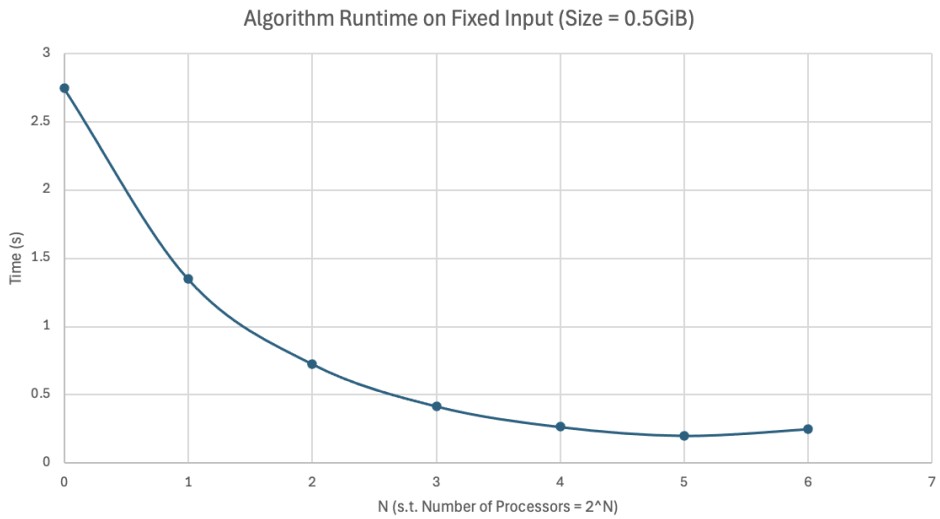
University at Buffalo The State University of New York

BENCHMARKING

Strong Scaling

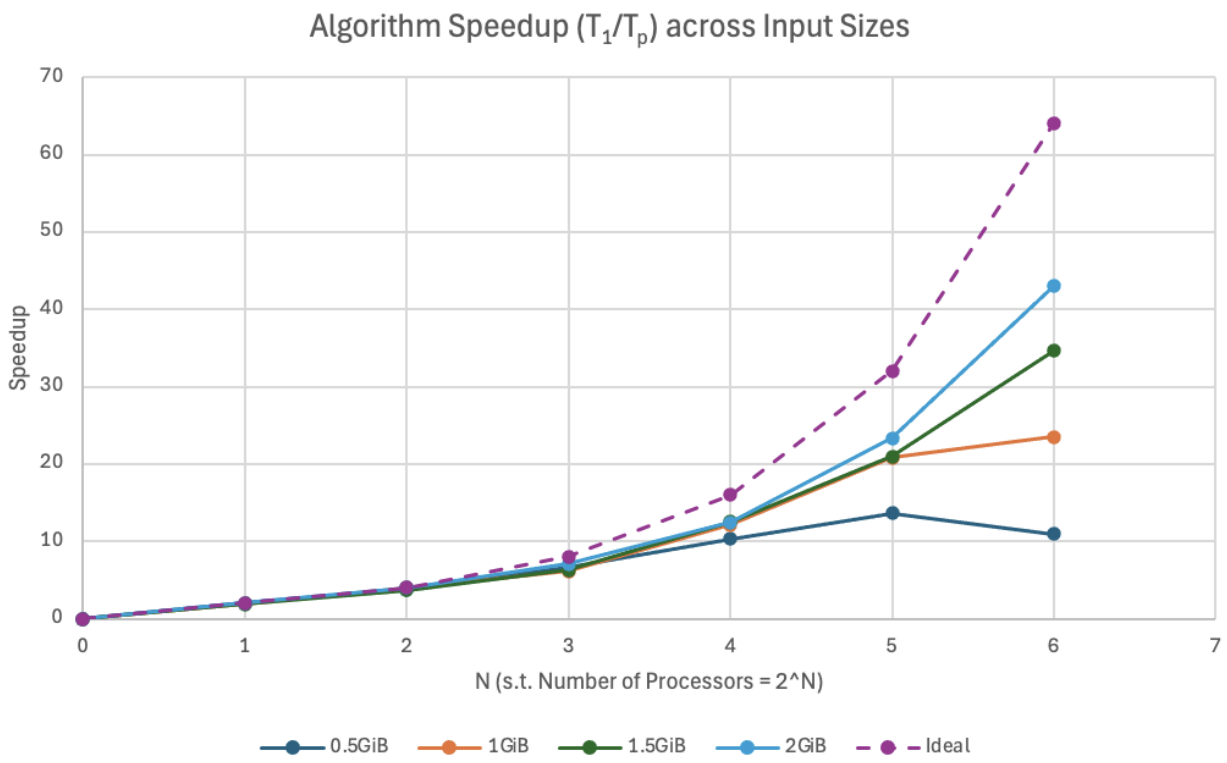
- Evaluating strong scaling with small input
  - Maximum 64 nodes
  - Maximum 1 task per node
  - Constrained to nodes that support InfiniBand
  - Tracked time to compute top hash for 0.5GiB input file
  - Runtime initially improved, but ultimately worsened when moving from 32 to 64 nodes





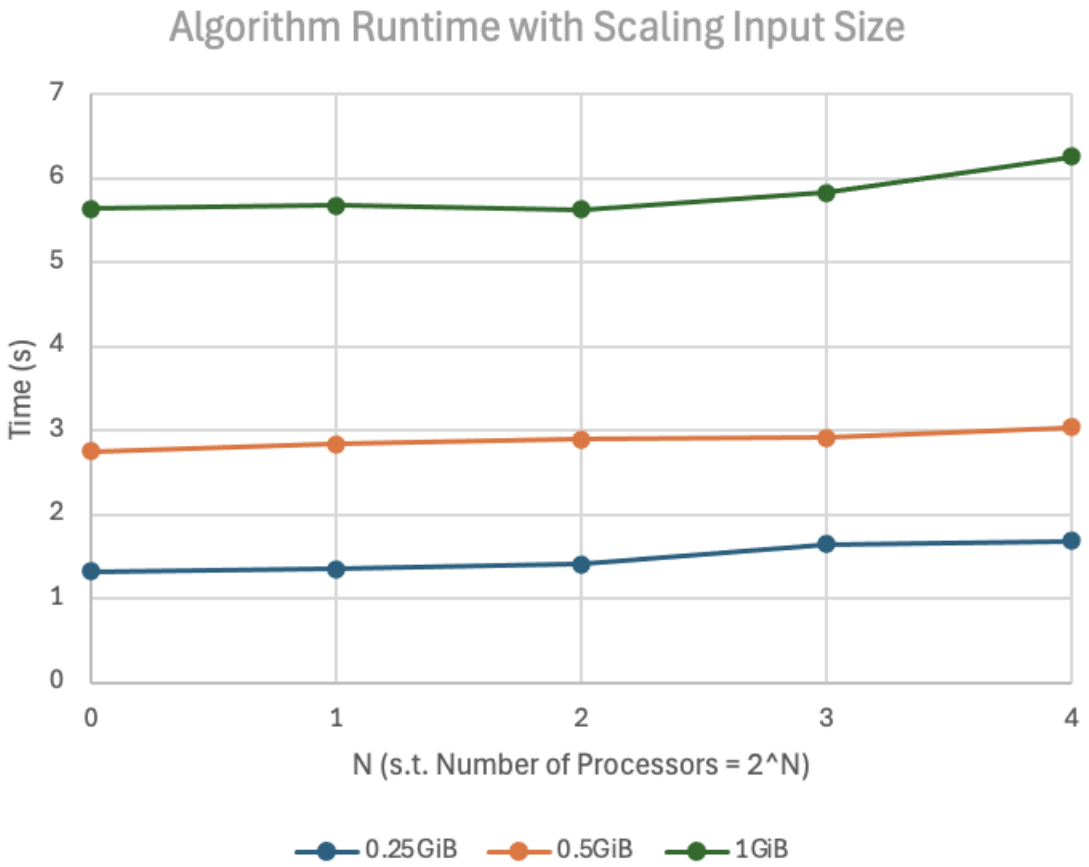
Strong Scaling (cont.)

- Utilized same parameters
  - Maximum 64 nodes
  - Maximum 1 task per node
- Repeated test with various input sizes
- Calculated speedup as number of processors increased – holding size of the input constant across each test trial
- As input size increases, speedup starts to conform more closely to the ideal



Weak Scaling

- Maximum 16 nodes with 1 task per node
- Constrained to nodes that support InfiniBand
- Doubled problem size with number of nodes
  - Legend denotes starting input size for each trial



Future Work

- Eliminate need for MPI_Allreduce when determining if there are multiple hashes left in the system
- Implement parallel function for checking a top hash and identifying problem leaf nodes
- Incorporate OMP into sequential step(s)

University at Buffalo The State University of New York

THANK YOU!

Questions?