CS 420, Parallel Programming, Fall 2019

MP-4

Shah, Ishan Szaday, Justin

Due Date: November 3rd, 2019, 11:59pm CDT

1. Overview

The purpose of this assignment is for you to learn how to improve the performance of a computation on a binary tree by parallelizing it using threads along with mutexes.

2. Getting Started

Pull the skeleton for MP4 from the release repository: https://github-dev.cs.illinois.edu/cs420-fa19/_release

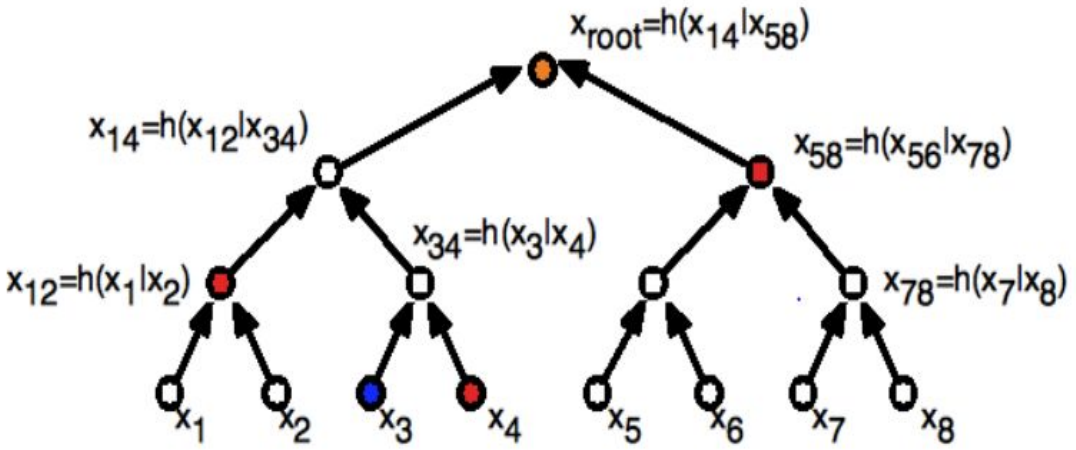
3. Problem Statement

Given a binary tree, you need to compute a value for each node of the tree. This computation is dependant on the data stored inside the node, the value of its left child and the value of its right child. The final result is the value of the root of the tree. You need to do this using two ways: synchronously using the main thread and asynchronously using multiple threads.

The specific computation used in this exercise is called hashing. A hash function is used to map data of arbitrary size to fixed-size values. This is widely used for checking the integrity of the data.

4. Hash-Tree

You have been given a tree data-structure and you would like to compute the hash of it. The idea is similar to merkle tree where the leaf-nodes contain data-blocks and the non-leaf nodes only contain the hash of its left-child and its right-child. The difference in our tree is that all nodes, not just the leaves, contain the data. So, the hash of a node will depend upon the hash of its data along with the hash its children.



Merkle Tree

https://en.wikipedia.org/wiki/Hash_chain

https://en.wikipedia.org/wiki/Merkle_tree

With respect to the above example, for your hash tree the hash of x12 will be h(x1|x2|d12) where d12 is the data inside x12 and similarly the hash of x58 will be h(x56|x78|d58). You do not need to worry about hashing as the provided function will do that for you:

unsigned char* hash(**unsigned char*** data, **unsigned char**** miningProof, **unsigned char*** lHash, **unsigned char*** rHash);

This function will return the hash of the data of the current node and store the proof that you solved a mining puzzle inside miningProof. In this exercise we do not use miningProof after it is computed. As you can see, the function depends on the current data along with the hash of its left node and the right node. You can pass NULL in place of lHash / rHash if these are not present such as in the case of leaf nodes.

You don't need to read the next paragraph unless you are interested in understanding the use of this computation:
This function first encrypts the data using AES and then computes the hash of it along with appending the lHash and rHash using SHA-256. It then solves a cryptographic puzzle of mining similar to Bitcoin where you need to find a hash for the block with some difficulty in it. This can help break the tree into branches and store these branches securely and independently in a distributed environment. Generally before storing data inside some peer-to-peer network, some participation is required which is replicated by the cryptographic mining puzzle in our case.

5. Part I - Single-threaded & synchronized multithreaded computation

5.1. Single threaded Hash-Tree

Your first task is to implement the single threaded version of hashing the tree. The approach is to traverse the tree in post-order and calling hash with its data once you obtain the lHash and the rHash.

```
unsigned char* hashTree(node* curr)
```

We have provided a tree program that will output a random tree of the specified size. While this outputs a balanced tree, it has a tuning factor for generating a completely random tree as well. You need to use qsub similar to previous MP's to submit the job. We have also provided 3 test tree files along with its hashes for verifying the correctness of your program. While this does not guarantee that your program is fully correct, it will help you verify if you are on the right track.

./tree 1024 > tree-1024.txt
./hashTree < tree-1024.txt

5.2. Parallelize Hash-Tree

You need to now parallelize the computation of the hash tree to improve performance. The approach is to spawn n threads (-t n) that asynchronously execute the hashTreeParallel function. You need to use mutexes for the critical section of your code to ensure that the threads are not performing redundant work and the dependency of parent-child hash is maintained. (Hint: All threads traverse the tree looking for nodes whose value has not yet been computed and pick that node without waiting for other threads to finish).

```
void* hashTreeParallel(void* node_curr);
```

./hashTree -t 3 < tree-1024.txt

- You may not modify the struct node declared inside hashTree.h. It has a status field which can be used for your synchronization purpose.
- The definition of hashTree and hashTreeParallel should not be changed.
- Your program should only print the value of the root along with the time taken. If it prints anything else, the autograder will not be able to grade your program.

- You would see ~2x performance improvement over single threaded version once you start using atleast two threads for tree sizes => 2000

5.3. Vtune Analysis

Similar to MP2, you will be collecting the hotspot result for your program:
amplxe-cl -collect hotspots <RunCommand>

Please ensure that your program is compiled with -g to be able to correctly view the results in VTune's GUI. Post a screenshot of the top hotspots along with the effective CPU utilization histogram for the following :

./hashTree -t 16 < tree-20480.txt

Use the tree generator to create a tree with the size of 20480 nodes.

5.4. Questions

A. Post the snippet of the critical section of your code for hashTreeParallel and briefly explain the logic behind it. You need to show that your threads are being utilized to its maximum capacity.

B. Provide the running time for the following tree sizes for both naive and multi-threaded:
   a. 2056 with single threaded, -t 2 and -t 4
   b. 6168 with single threaded, -t 4 and -t 8
   c. 10240 with single threaded, -t 8 and -t 12
   d. 20480 with single threaded, -t 12 and -t 16

C. For Vtune, what contributes towards the spin time and what can cause your program to have poor or less than optimal CPU cores utilization?

6. Part II - Master-Slave (4 credit students only)

Another way to solve this problem avoiding synchronization is to have your main thread assign nodes (jobs) to the slave threads that you create. These threads will independently compute the hash of disjoint subtrees using the sequential algorithm of Part-I. Once all of these threads are complete, the main thread can then compute the hashes of the remaining nodes in the upper part of the tree and eventually of the root node. The problem is to find a way to evenly assign the nodes to the threads within the binary tree.

For n threads, you need to go down (log2(n) + 1) levels (assuming the 1st level is 1) and assign these nodes to the threads. For instance if you have 4 threads, you will go down to level 3 which will utmost have 4 nodes and assign these to the threads. If there is no node present at that level for a branch, skip it for simplicity. In this case, you will create fewer number of threads than n.

./hashTree -t 4 -m < tree-1024.txt

A. Provide the running time for the following tree sizes for this approach.
    a. 2056 with -t 2 and -t 4
    b. 6168 with -t 4 and -t 8
    c. 10240 with -t 8 and -t 12
    d. 20480 with -t 12 and -t 16

B. Which approach is better according to you considering the balance of the tree, size of it and the number of cores available?

7. Submission Guidelines

The graded portion of this assignment are your single threaded hashTree and the parallelized hashTree along with the Vtune analysis and answers to questions. For 4 credit students, it will also be the implementation of master-slave and two questions pertaining to it.

7.1. Points breakdown

- 30*pts* - Single threaded hashTree
- 40*pts* - Parallel hashTree
- 15*pts* - Vtune Analysis
- 15*pts* - 3 questions (5 points each)
- 40*pts* - Master-Slave Implementation *(4 credits only)*
- 10*pts* - Master-Slave 2 questions *(4 credits only)*

7.2. Pushing your submission

Follow the git commands to commit and push your changes to the remote repo. Ensure that your files are visible on the GitHub web interface, your work will not be graded otherwise. Only the most recent commit before the deadline will be graded.