High Performance Merkle Tree

Building a merkle tree from a .bpkg file in parallel to increase the speed

SID:520627482

***Abstract*—In this report, I thoroughly describe how I enhanced the program's performance by "building a Merkle tree from a .bpkg file in parallel." The Code section details the modifications I made to the code, the Benchmark section outlines the placement of the benchmarks and the file structure, the Testing section explains the testing setup and procedures, and the Results section documents the outcomes of my tests.**

I. Code

I divided the code changes into two parts.

In the first part, I analyzed the dependency graph of the Merkle tree building process and identified that read_file and merkle_tree_node_build are two independent tasks. Initially, I used a queue to traverse the merkle_tree_node layer by layer, simultaneously reading files and building nodes, which resulted in slow performance. In the high_performance/bpkg2.c file, I modified this setup by first reading the file and storing its content in a list. Then, I constructed the merkle_tree_node using a special correspondence between the binary tree and the list index. This change significantly improved the performance (see Result).

The code change is reflected in read_hashes_from_file() and create_non_leaf_node()

Although this change is unrelated to threading, it involved modifying the code based on the dependency graph analysis and resulted in a notable speed increase, which is why it is mentioned here.

In the second part, I further enhanced the performance by distributing the task of constructing merkle_tree_node across multiple threads. Assuming there are 256 nodes and 8 threads available, the task of constructing these 256 nodes is evenly divided among the 8 threads. Each thread is responsible for handling 32 consecutive node construction tasks.

The code change is reflected in create_non_leaf_node() and create_non_leaf_node_task(). This version's program is bpkg3.c put in "high_performance"

II. Benchmark

All the codes and files related to "high performance merkle tree" is put in folder "high_performance" as a benchmark.

bpkg2.c is the second version where I split reading file and construct nodes.

bpkg3.c is the third version where I use multiple threads to construct the nodes.

merkle_tree.c is build because the old version of merkle_tree.c can't support new operations.

hpcheck.c is used to record the time of running the program.

result.xlsx is the file recording the results and used for generating the graph.

III. Testing Setup

In makefile, there are three new rules: hpcheck, hpcheck2, hpcheck3. Correcponding to original version, 2nd version and 3rd version.

Use "make hpcheck", "make hpcheck2", "make hpcheck3" to compile the program. Then run "./hpcheck", "./hpcheck2" and "./hpcheck3" to print out the running time.

Changing the defined variable "MAX_THREADS" in bpkg3.c, then compile hpcheck3 again to test the speedup with different number of threads.

The results are recorded in the below section.

IV. Result

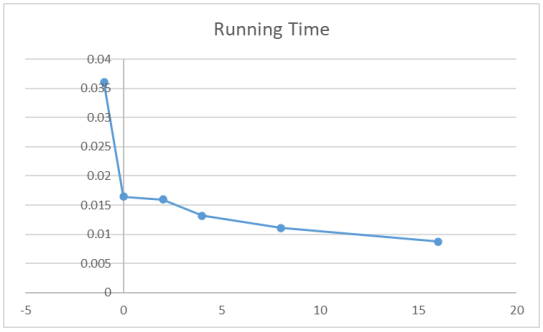

Fig. 1. Running Time v.s. Num of Threads

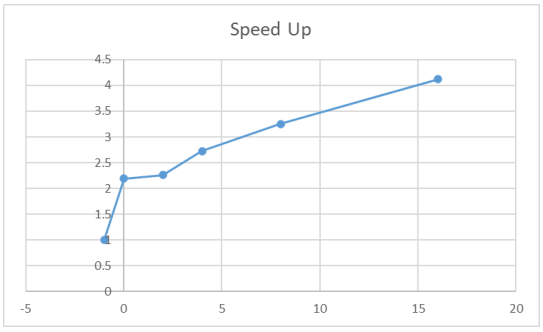

Fig. 2. Speedup

V. NOTE

If you are performing testing.

1. Ignore the first running outcome. Every time, the first running outcome is pretty slow. I assume that is because the system is preparing for the first running.

2. Do multiple experiments for each test setting and take the average. During testing, the running time can vary significantly, and sometimes, the running time with 16 threads is even higher than with 8 threads. However, by conducting multiple experiments and averaging the results (four or more trials), the data closely aligns with the results shown in my charts.