ILLINOIS INSTITUTE OF TECHNOLOGY

Illinois Institute of Technology

Department of Computer Science

CS553 - Section: 01

Cloud Computing

Homework 4

Author:
Hongbo WANG

Student Number:
A20524770

March 31, 2024

1 Hardware Specification Details

1) Chameleon Instance: hwang218 at CHI@TACC
2) CPU: 2 × Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz
3) Memory: 16 × 16GB (256GB) of Micron HMA82GR7DJR8N-XN 16GB DDR4-3200 2Rx8 DIMM
4) Disk: Micron MTFDDAK480TDS-1AW1ZABYY 5300 PRO 480GB, SATA
5) Network: BCM5720 - Dual-Port 1GBASE-T

2 Environment

1) Chameleon Cloud (CHI@TACC)
2) CC-Ubuntu22.04

3 Experiments

3.1 File size: 1024 Mbyte

Experiments	Hash Threads	Sort Threads	Write Threads	Time-Consuming (ms)
1	1	1	1	59450
2	1	1	4	44093
3	1	1	16	59386
4	1	4	1	39807
5	1	4	4	43552
6	1	4	16	31466
7	1	16	1	38439
8	1	16	4	37585
9	1	16	16	31810
10	4	1	1	59456
11	4	1	4	55413
12	4	1	16	63880
13	4	4	1	42383
14	4	4	4	34341
15	4	4	16	33855
16	4	16	1	40799
17	4	16	4	33076
18	4	16	16	36287
19	16	1	1	63172
20	16	1	4	59316
21	16	1	16	44336
22	16	4	1	39954
23	16	4	4	34130
24	16	4	16	40291
25	16	16	1	37751
26	16	16	4	36969
27	16	16	16	31840

Table 1: The 27 Experiments of 1024 MBytes file

3.2 File size: 65536 Mbyte

Experiments	Hash Threads	Sort Threads	Write Threads	Time-Consuming (ms)
1	1	1	1	1257134
2	1	1	4	934023
3	1	1	16	952874
4	1	4	1	1164437
5	1	4	4	1155160
6	1	4	16	1156754
7	1	16	1	1156581
8	1	16	4	749775
9	1	16	16	672656
10	4	1	1	1257260
11	4	1	4	117176
12	4	1	16	1350811
13	4	4	1	896233
14	4	4	4	726177
15	4	4	16	715900
16	4	16	1	862738
17	4	16	4	699427
18	4	16	16	767327
19	16	1	1	1335839
20	16	1	4	1254300
21	16	1	16	937532
22	16	4	1	844870
23	16	4	4	721715
24	16	4	16	851996
25	16	16	1	798285
26	16	16	4	781749
27	16	16	16	673290

Table 2: The 27 Experiments of 65536 MBytes file

4 Findings

4.1 Scalability

1) Thread Pooling: Utilize thread pools to manage a fixed number of worker threads that can process incoming tasks asynchronously. This approach reduces the overhead of thread creation and destruction and can improve scalability by efficiently utilizing system resources.

4.2 Concurrency

Implement proper synchronization mechanisms to prevent data races and ensure data consistency. This includes the use of mutexes, condition variables, atomic operations, and read-write locks.

1) Hash Threads: This approach exhibits favorable concurrency characteristics, offering ease of parallelization for generating hashes across multiple threads concurrently.

2) Sort Threads: This method demonstrates excellent concurrency capabilities by leveraging a strategy of segmenting large files into smaller partitions. Each partition can be independently sorted within its own thread, maximizing parallelism and efficiency.

3) Write Threads: This aspect presents challenges in achieving concurrency. The concurrent management of write operations poses complexities that may require careful synchronization and coordination among threads to ensure data integrity and consistency.

4.3 Performance

1) Hash Threads: Moderately Effective. While hash threads can operate concurrently, certain sections of the process necessitate the use of mutexes to ensure synchronization.

2) Sort Threads: Highly Efficient. Well-designed sorting threads can significantly reduce processing time.

3) Write Threads: Least Efficient. Writing is a sequential process, limiting the effectiveness of concurrent thread initialization as only one write operation can occur at a time.

5 The Best and The Worst Parameters

Optimal Configuration: In the context of multithreading in C++, after careful experimentation and analysis, the most effective parameter configuration observed in my program is as follows: "Hash Threads: 1, Sort Threads: 16, Write Threads: 16". This configuration has been determined to optimize performance significantly. The rationale behind this choice is rooted in the balance achieved between the number of threads allocated for hash computation, sorting, and writing operations.

When the number of writing threads is insufficient, it leads to a bottleneck situation where the efficiency of hash threads diminishes substantially. This inefficiency arises due to the increased contention for resources, particularly with mutexes for inter-thread communication. Consequently, the overall execution time escalates. Hence, allocating a higher number of writing threads aids in alleviating this contention, resulting in improved performance. Surprisingly, the experimental results demonstrate that employing only one thread for certain tasks, particularly hashing, outperforms a multi-threaded approach under these circumstances.

Suboptimal Configuration: Conversely, in my experimentation with various parameter configurations, a suboptimal setting emerged, characterized by the configuration: "Hash Threads: 4, Sort Threads: 1, Write Threads: 16". This configuration proved to be significantly less efficient compared to the optimal setup. The primary reasons for its inferior performance lie in the disproportionately low number of sorting threads and the insufficient allocation of writing threads.

With only one thread dedicated to sorting, the processing of data files becomes a severe bottleneck, significantly prolonging execution time. Moreover, the scarcity of writing threads exacerbates this issue, leading to increased contention and resource contention, further hindering performance. As a result, the overall execution time is significantly prolonged compared to more balanced configurations.