Illinois Institute of Technology

Department of Computer Science

CS553 - Section: 01

Cloud Computing

Homework 4 (2nd Attempt)

Author:
Hongbo WANG

Student Number:
A20524770

May 4, 2024

1 Hardware Specification Details

1) Instance: Ubuntu 22.04 LTS
2) CPU: 13th Gen Intel(R) Core(TM) i9-13900HX 24 cores
3) Memory: 2 × 16GB (32GB) DDR5 5600MHz
4) Disk: 2 × 1TB SSD
5) Network: BCM5720 - Dual-Port 1GBASE-T

2 Experiments

2.1 File size: 1024 Mbyte

Experiments	Hash Threads	Sort Threads	Write Threads	Time-Consuming (second)
1	1	1	1	18.05
2	1	1	4	18.21
3	1	1	16	17.97
4	1	4	1	13.53
5	1	4	4	13.56
6	1	4	16	13.41
7	1	16	1	13.09
8	1	16	4	12.95
9	1	16	16	12.85
10	4	1	1	17.93
11	4	1	4	17.79
12	4	1	16	17.87
13	4	4	1	8.16
14	4	4	4	8.14
15	4	4	16	8.14
16	4	16	1	7.58
17	4	16	4	8.20
18	4	16	16	7.56
19	16	1	1	18.04
20	16	1	4	17.96
21	16	1	16	17.88
22	16	4	1	12.23
23	16	4	4	12.20
24	16	4	16	12.27
25	16	16	1	6.42
26	16	16	4	6.63
27	16	16	16	6.50

Table 1: The 27 Experiments of 1024 MBytes file

2.2 File size: 65536 Mbyte

2.2.1 Memory size: 1024 Mbyte

Experiments	Hash Threads	Sort Threads	Write Threads	Time-Consuming (second)
1	1	1	1	1405.30
2	1	1	4	1421.24
3	1	1	16	1375.30
4	1	4	1	1243.61
5	1	4	4	1247.31
6	1	4	16	1274.56
7	1	16	1	1224.56
8	1	16	4	1256.56
9	1	16	16	1231.56
10	4	1	1	1257.26
11	4	1	4	1171.17
12	4	1	16	1350.81
13	4	4	1	1896.23
14	4	4	4	1261.77
15	4	4	16	1159.00
16	4	16	1	1327.38
17	4	16	4	1294.27
18	4	16	16	1173.27
19	16	1	1	1258.39
20	16	1	4	1258.39
21	16	1	16	1258.39
22	16	4	1	1172.43
23	16	4	4	1154.43
24	16	4	16	1124.43
25	16	16	1	899.02
26	16	16	4	873.51
27	16	16	16	865.08

Table 2: The 27 Experiments of 65536 MBytes file

2.2.2 Memory size: 1024/8192/16384/32768 Mbyte

Memory (MB)	Hash Threads	Sort Threads	Write Threads	Time-Consuming (second)
1024	16	16	16	865.08
8192	16	16	16	547.33
16384	16	16	16	834.46
32768	16	16	16	std::bad_alloc

Table 3: The 4 Experiments of different memory size to generate 65536 MBytes file

3 Findings

3.1 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.

3.2 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.

3.3 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.

4 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.