Homework 3 Report

Abstract

The purpose of this homework is to understand sequential consistency and to analyze what happens when sequential consistency breaks. From the data obtained I can figure out the tradeoffs between performance and reliability.

1. Introduction

As a software engineer in a startup company Ginormous Data Inc. (GDI), I am asked to implement a program that uses inadequate-but-faster synchronization methods. The company wants to ignore a few errors as long as the program returns good enough results quickly. The program simply calls swap() function which will take two indices, i and j as inputs and decrement the i^{th} value and increment the j^{th} value each by 1. Then it checks whether the sum of values in the array is 0 or not. If it is not 0, I can conclude there were some synchronization errors. To check what gives the best performance in each state, I tested with various parameters and on different machines. The output data is in the **data** section of this report.

2. Testing Environment

Tests were performed in two different servers (SEASNet Linux Server 09, and 10).

2.1. SEASNet Linux Server 09

Java version	: java 13.0.2
Model name	: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
MemTotal	: 65755720 kB
MemFree	: 41921604 kB
MemAvailable	: 61194688 kB

2.2. SEASNet Linux Server 10

Java version	: java 13.0.2
Model name	: Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
MemTotal	: 65799584 kB
MemFree	: 4688016 kB
MemAvailable	: 12621120 kB

3. Test Parameters

In this project, I changed a few parameters while running the test on each server. First, I tested the performance for each method based on the number of threads (1, 4, 8, 40) running. Then I changed the size of the state array, and ran the tests again. I used 100000000 swap transitions so that the results can be dominated by the actual work.

4. Classes

4.1. NullState

Test on Null state is done in order to time how long it takes before the program ends. NullState simply passes the test by doing nothing.

4.2. SynchronizedState

Synchronized state class is implemented using `synchronized` keyword in the swap method. This allows only one thread to access and run the synchronized method making it atomic. So using this class is reliable.

4.3. UnsynchronizedState

Unsynchronized state class behaves exactly the same as synchronized state except that the `swap` method is run without `synchronized` keyword. This allows any thread to access the resource which leads to synchronization error. All output--even if swaptest returns 0--cannot be trusted.

4.4. AcmeSafeState

Using `java.util.concurrent.atomic.AtomicLongArray` class, I implemented AcmeSafeState without `synchronized` keyword. Made the `swap` method atomic by calling `getAndAdd(int i, long delta)` which atomically adds the given value to the element at index i.

5. Data

5.1. Size of the state array : 5
100000000 swaps / Linux09

	1	8	16	40
Null	1.479s	0.414s	0.454s	0.575s
Synchronized	2.220s	22.85s	23.70s	23.22s

Unsynchronized	1.647s	3.721s	2.638s	3.061s
AcmeSafe	3.815s	15.31s	8.584s	7.177s

100000000 swaps / Linux10

	1	8	16	40
Null	1.226s	0.597s	0.621s	0.785s
Synchronized	1.764s	4.917s	4.975s	5.271s
Unsynchronized	1.327s	2.011s	2.195s	2.697s
AcmeSafe	4.476s	13.01s	13.31s	15.28s

5.2. Size of the state array : 20
100000000 swaps / Linux09

	1	8	16	40
Null	1.403s	0.408s	0.420s	0.602s
Synchronized	2.106s	25.39s	24.32s	23.09s
Unsynchronized	1.552s	5.708s	4.990s	3.943
AcmeSafe	2.860s	16.92s	16.22s	8.631s

100000000 swaps / Linux10

	1	8	16	40
Null	1.206s	0.508s	0.518s	0.767s
Synchronized	1.777s	5.362s	5.179s	5.128s
Unsynchronized	1.337s	4.262s	4.291s	7.121s
AcmeSafe	2.594s	13.43s	12.94s	7.354s

5.3. Size of the state array : 100
100000000 swaps / Linux09

	1	8	16	40
Null	1.415s	0.409s	0.438s	0.648s
Synchronized	2.086s	16.80s	20.72s	19.87s
Unsynchronized	1.581s	4.826s	4.164s	3.129s
AcmeSafe	2.829s	8.761s	4.585s	5.191s

100000000 swaps / Linux10

	1	8	16	40
Null	1.260s	0.602s	0.518s	0.958s
Synchronized	1.812s	4.615s	4.748s	4.943s
Unsynchronized	1.330s	3.590s	3.808s	3.189s
AcmeSafe	2.648s	7.532s	5.632s	7.599s

6. Analysis on data

The goal of this project is to build a class that performs faster than the original `SynchronizedState`. Null state computes fastest but it is unreliable. The next good-performing class in `unsyncrhonized` which will also suffer from synchronization errors. So our only reliable options are `synchronized` and `AcmeSafe`. As it is shown in data from the previous section, `AcmeSafe` outperforms `SynchronizedState` in Linux09 server. It shows that for threads 8, 16, and 40, `AcmeSafe` almost performs 1.5 times faster than `SynchronizedState.` As for the parameters, it performed the best when the size of the state array is 100.

6.1. AcmeSafe vs. Synchronized

Both classes are DRF but `AcmeSafe` outperforms `Synchronized` due to overheads. As `synchronized` keyword locks the entire method preventing other threads from accessing the method, it will have more overhead from synchronization than `AcmeSafe`. `AcmeSafe` only locks the critical section allowing other executions to run concurrently.

7. Conclusion

After analyzing the performance of different classes based on various parameters, I can conclude that `AcmeSafe` is the one with reliable and fast performance. This is because its implementation has less overhead than `synchronized` class which is another reliable class.