Java Synchronization Experiment Report

Yuxin Wang 905129084
University of California, Los Angeles

Abstract

The objective of this assignment is to test and analyse the multi-threading performance with different settings and methods in Java. Each of the settings and methods was tested on a trivial task swap, and their performance were recorded and compared.

1 Test Environment

SEASnet GNU/Linux servers lnxsrv09:

Java Version:
java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)

CPU info:
Number of processor: 32
CPU cores: 8
Processor model: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz

SEASnet GNU/Linux servers lnxsrv10:

Java Version:
java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)

CPU info:
Number of processor: 4
CPU cores: 4
Processor model: Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz

2 Implementation Detail

Following is the detailed implementation of the AcmeSafeState class.

```
import java.util.concurrent.atomic.AtomicLongArr
ay;
class AcmeSafeState implements State {
    private AtomicLongArray value;

    AcmeSafeState(int length) {
    value = new AtomicLongArray(length);
    }

    public int size() { return value.length(); }

    public long[] current() {
     long[] copy = new long[value.length()];
     for(int i = 0; i < value.length(); i++){
     copy[i] = (long)value.get(i);
     }
     return copy;
    }

    public void swap(int i, int j) {
    value.getAndIncrement(j);
    value.getAndDecrement(i);
    }
}
```

The AcmeSafeState class is an clear and simple implementation of State that uses the AtomicLongArray class from java.util.concurrent.atomic package to ensure the functions are DRF (data-race free), while keeping relatively fast speed. An AtomicLongArray is a array of longs that support atomic operations. The get and set related methods in this class works like reads and writes on volatile variables. Because AtomicLongArray uses the low-level CPU operations CAS (compare-and-swap), which is a atomic instruction, synchronization could be achieved during multi-threading. Additionally, due to the fact that CAS is a low-level CPU instruction, the speed of operation is guaranteed to be relatively fast.

3 Performance analysis

Performance on the task of doing 100000000 swaps was tested for each of the four classes on two different server: SEASnet GNU/Linux servers lnxsrv09 and lnxsrv10. The following six tables record the performance measurement with different settings, i.e. different number of entries in the array and different number of threads used. For each entry in the table, there are two times measured in seconds, while the upper one represents the total real time and bottom one represents the total CPU time.

Additionally, SychronizedState, AcmeSafeState and NullState are always reliable during the test, but UnsychronizedState is not under most situation. This is reasonable as NullState does not do the swap, and SychronizedState, AcmeSafeState are data-race free but UnsychronizedState is not.

Table 1: 5 entries on server 9

Sychronization Method	1 Thread	2 Threads	8 Threads	40 Threads
Synchronized	1.92264	14.6954	26.0267	32.0314
	1.92122	28.6699	88.5941	114.027
AcmeSafe	2.53826	13.0963	16.0315	7.01956
	2.53625	25.0359	126.419	198.441
Unsynchronized	1.49127	2.66090	5.23620	4.48288
	1.49015	5.15552	39.0963	52.8721
Null	1.30960	0.667073	0.295954	0.238097
	1.30843	1.32413	2.25928	6.04744

Table 2: 5 entries on server 10

Sychronization Method	1 Thread	2 Threads	8 Threads	40 Threads
Synchronized	1.64944	13.7427	4.87170	5.06298
	1.64748	24.6345	4.70159	6.46546
AcmeSafe	4.75511	5.65815	14.5269	14.7364
	2.50906	9.28947	52.7511	57.1143
Unsynchronized	1.21612	1.78105	3.58372	3.43356
	1.20840	3.49712	14.0095	13.5901
Null	3.15990	2.13623	0.457994	1.24671
	1.15080	1.17215	1.24947	11.3131

Table 3: 100 entries on server 9

Sychronization Method	1 Thread	2 Threads	8 Threads	40 Threads
Synchronized	2.06908	11.9541	29.8631	30.7435
	2.06773	23.8630	94.7006	97.9257
AcmeSafe	2.65403	4.99931	8.92034	5.08034
	2.65286	9.89982	70.9089	148.095
Unsynchronized	1.47947	6.59849	4.14502	2.98565
	1.47835	1.47947	32.7818	74.0888
Null	1.33022	0.694654	0.293152	0.306706
	1.32900	1.36046	2.23733	7.19053

Table 4: 100 entries on server 10

Sychronization Method	1 Thread	2 Threads	8 Threads	40 Threads
Synchronized	1.66436	16.5810	4.59795	4.85877
	1.66327	33.0344	5.33954	5.76095
AcmeSafe	3.58055	6.99750	4.21465	6.04219
	2.45791	13.9359	16.7095	23.8266
Unsynchronized	1.21430	3.72330	3.53867	3.67775
	1.21286	7.43666	14.0392	14.4718
Null	1.09026	0.596204	0.427884	0.373348
	1.08932	1.17484	1.56021	1.32439

Table 5: 500 entries on server 9

Sychronization Method	1 Thread	2 Threads	8 Threads	40 Threads
Synchronized	2.11253	11.3532	29.3721	33.5370
	2.10973	22.2291	90.7872	104.078
AcmeSafe	2.65658	5.13696	3.58712	3.46739
	2.65544	10.2375	27.4710	99.2915
Unsynchronized	1.49445	5.37619	2.68088	2.01445
	1.49330	10.7346	21.2726	55.1192
Null	1.33325	0.693756	0.287139	0.571782
	1.33198	1.37498	2.24278	14.1465

Table 6: 500 entries on server 10

Sychronization Method	1 Thread	2 Threads	8 Threads	40 Threads
Synchronized	1.69962	7.79891	4.69353	5.33279
	1.69786	13.0893	5.60724	4.48760
AcmeSafe	2.43053	4.82585	4.18059	4.29301
	2.42881	9.59439	16.5480	16.9080
Unsynchronized	1.24317	3.23376	2.61364	2.51444
	1.24171	6.45411	10.3373	9.95935
Null	3.06228	0.925678	0.392874	0.451710
	1.08700	1.09455	1.50571	1.61954

First of all, overall it is obvious that the using one thread is the fastest method under all settings and classes. This is because the task is trivial and inexpensive, so the overhead of multi-threading becomes the bottleneck.

Next, the performance of NullState illustrates the ideal situation of multi-threading, where no race condition will occur therefore no synchronization method is taken. Under this circumstance, the performance is positively proportional to the number of threads, until a limitation is reached and the performance is optimized.

In terms of performance, generally UnsynchronizeState consumes least amount of time. This is reasonable as not extra instructions are taken to ensure synchronization, the consequence of this is that under most test cases the result is not reliable when the number of threads is not one. The error range is roughly from -50000 to 50000 under most cases. Considering the error rate is actually less than 0.1%, this implementation is not entirely bad.

To compare two reliable classes SynchronizeState and AcmeSafeState, we need to look at their performance under different connditions. To begin with, when there is no multi-threading, i.e. when the number of thread is one, SychronizedState performes better than AcmeSafeState. This is possibly because for a single operation add or substract, ++ or - - is faster than getAndIncrement() and getAndDecrement(). As more threads are used, AcmeSafeState starts to outperform SychronizedState, as AcmeSafeState uses lower-level thus cheaper instructions to prevent race condition.

Besides the performance comparison between classes, the performance comparison between different number of threads is also interesting. The performance does not necessary become better as the number of threads increasing. This is especially true when there are less entries. When the number of entries is less, more threads will try to access the same entry, therefore waste more time on waiting for the current thread to complete its operation. Besides, the overhead of creating and joining threads also contributes to the bad performance. As the number of entries increases, the performance starts to benefit more from the more threads used.

Different server also gives different results. SynchronizedState obviously performs better on server 10 than server 9, while AcmeSateState performs more steady on different servers. This is an interesting results as SEASnet GNU/Linux servers lnxsrv9 actually has more cores. Therefore the low performance may not due to the limitation of the SynchronizedState, but other factor like test time or processor models.

4 Limitation

Although this test compares the different performance of different thread settings and synchronization methods. It does not necessary reflect the real ability of each methods. Because the tests were taken in different time, so the situation of the server is not always stable. Ideally these tests should be repeated several times to prevent outcome by chance.

Conclusion

Multi-threading is a method to achieve better performance. In Java there are several ways to ensure data-race free in multi-threading, including but not limited to: Using keyword Synchronized, using Java.util.concurrent.atomic, using locks etc. The best setting and method depends on the specific situation. However, while more threads add more parallelism and fasten the program, more threads will also contribute to heavier overhead, especially when they need to modify the same target. Therefore the performance is not simply linearly proportional to the number of threads used. Additionally, this test shows multi-threading is unnecessary for trivial tasks as the overhead of multi-threading becomes the performance bottleneck.