Comparison of Concurrency in Java and Go

Hitesh Kumar

Aarsh Patel
San José State University
CS-252

May 17, 2022

Contents

1 Introduction

There are many programming languages developed over decades and are still being developed. The programming languages are mostly categorized into low-level and high-level languages. Assembly language and machine language are examples of low-level languages while modern languages like Java, Python, Go, and C# are high-level languages. In this report, we are doing a concurrency analysis of Java and Go. We are using a couple of benchmark tests to analyze the concurrency as to which language performs better in specific tasks.

Figure 1: Types of programming languages

2 History:Java

Java was started in 1991 by a team of engineers from Sun Microsystems. It was developed by James Gosling, Mike Sheridan, and Patrick Naughton and was called Oak as a part of the green team. [1]Later it was renamed Java because of copyright issues. Java was designed to be a simple, object-oriented, and familiar language and the goal was to simplify complex environments while still being effective. [2] Java was built to be architecture-neutral and portable through virtualization by using Java Virtual Machines (JVM). Java has automated garbage collection, and this was implemented to enable high performance as automated garbage collection reduces the possibility of the main thread and garbage collector utilizing the same memory at the same time. [2]

3 History:Go

Go was invented in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson at Google, and was first released as open-source in 2009. The main objective behind developing this language was that there was dissatisfaction with the current working environments and languages being used at Google. [3]Both Java and C++ were more than a decade old and were not designed to work at the scale on which Google was working. Go tries to combine the simplicity of dynamically typed languages like python, fast compilation times, garbage collection, and reduced complexity when developing for concurrency while also allowing developers to be quick and efficient. [4]

Figure 2: Java VS GoLang

4 Concurrency vs Parallelism

Before analyzing concurrency in Go and Java, it's better to understand the difference between parallelism and concurrency. The main difference is that concurrency is about managing multiple computations at the same time, while parallelism is about running multiple computations simultaneously. [5] As parallelism is about multiple tasks, it requires multiple processing units while concurrency requires just one processing unit. Concurrency is achieved through context switching, while parallelism is achieved through multiple central processing units. (CPUs). [5] Lastly, debugging is easy in parallelism than in concurrency.

5 Need of Concurrency

Before knowing how Java and Go handle concurrency, it's important to know why concurrency is required and what are the benefits of it. Concurrency allows multiple applications to be executed simultaneously, thus promoting better utilization of resources. [6] Concurrency also improves overall efficiency and response time as it helps in reducing the waiting time between threads. The

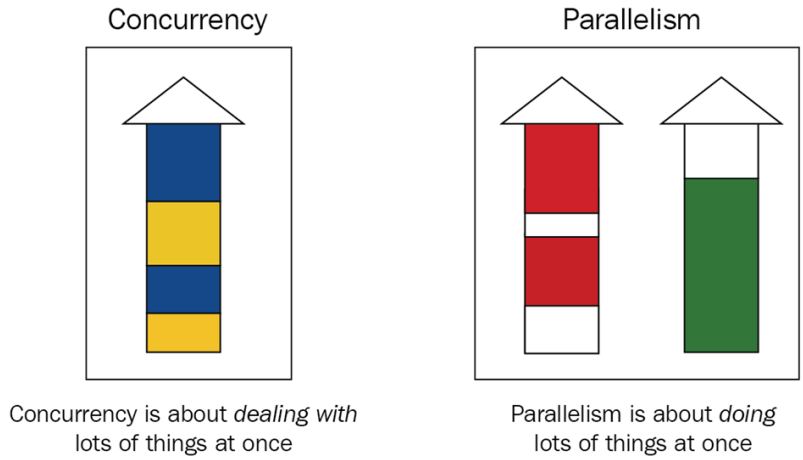

Figure 3: Concurrency VS Parallelism

performance of the operating system is also improved as different hardware resources can be accessed simultaneously by separate threads thus allowing simultaneous use of the same resources as well as the parallel use of different resources. [6] To implement complex tasks, multiple processes should be run at the same time and concurrency allows us to do that. Therefore, concurrency is important, and the ability to handle concurrency should be an important factor in choosing the programming language. It's important to note that the time to implement concurrent tasks is dependent on the activities that other processes are, the hardware of the device, and its operating system.

6 Concurrency in Java

Java was designed to be interpreted, threaded, and dynamic. This was achieved through supporting the use of multithreading and by providing a class, 'Thread', and an interface, 'Runnable', for developers to use to create multithreaded programs. [2] Java has built-in support for concurrency through java.util.concurrent package. We can use the Thread and the Executor Class to implement multithreading. [7] In Java, there exists a main thread and extra threads can be made by using the Thread class. The Thread class is used to create and manage threads and the Executor class provides useful methods to work with many aspects of Thread management. [7]One of the cons of Java threads is that they are hardware dependent. The communication between the thread is complex and result in high latency. Also, the cost of threads is high, and they are preemptively scheduled. [8]

6.1 ExecutorService in Java

Java ExecutorService interface allows the user to separate out the thread creation and thread management and gives an opportunity to run threads asynchronously. Three major tasks are performed by ExecutorService :

1. Creation of thread
2. Management of thread
3. Submission and Execution of Thread

7 Concurrency in Go

Talking about concurrency, Go uses Communicating Sequential Processes (CSP) model instead of threading approaches that are used by other languages. A major design goal of Go is to improve traditional multithreading already existing in other programming languages and make concurrent programming easy and efficient. For this purpose, Go has two things, the first is goroutines, which make threads lightweight and easy to create and the second is the gochannel which is used for communicating between threads. [9] Goroutines are lightweight threads that can be multiplied so that if one gets blocked, others can still continue to run. At first, there is only one goroutine that calls the main function, so we call it the main goroutine. [9] As needed, new goroutines can be created by the following go statement:

go f ()

Thread communication and synchronization take place in Go via built-in primitives called channels. The typical flow of a multithreaded program in Go involves setting up communication channels and then passing these channels to all goroutines which need to communicate. [10]Then goroutines send processed data to the channel. This communication is lock-free and bidirectional. Each channel is a channel for values of a particular type, called the channel's element type. E.g.: ch (chan) is the declaration of the channel and int is the element type.

ch: make (chan int)

The first thing in the multithreaded program in Go involves setting up gochannels. Then these channels are passed to all goroutines that need to communicate. The worker goroutines send processed data to the channel and the goroutines which are waiting for the processes of other channels to be completed.

8 Tools

Computer model	Dell XPS 8940
CPU 21	Intel i7-11700K @ 3.60GHz
Operating System 21	Windows 11 Home
RAM	32GB

Table 1: Hardware used for Tests

GO	Go 1.18.1 windows/amd64
Java	Java 11.0.14 2022-01-18 LTS

Table 2: Software used for Tests

GO	Go-inbuilt testing benchmark
Java	Java Microbench Harness(JMH)

Table 3: Benchmark Tools

9 Methodologies

9.1 Concurrent Matrix Multiplication

We are using matrix multiplication as a benchmark to check the concurrent functionality of both Java and GOLang. We have used the code mentioned in [11] and made changes as part of experiment. We have transformed the generation of matrix as given in paper to randomised value. In the paper. the matrix values were fixed but we have made a custom random function to generate the matrix every time for each Java and GO execution. Below is the code snippet to generate the randomised values in the 4096 x 4096 matrix. The same has been done in the Golang :

```java
class MyMatrix {
    1 usage
    public List<ArrayList<ArrayList<Integer>>> genMatrix() {
        int maxSize = 4096;

        ArrayList<ArrayList<Integer>> A = new ArrayList<>(maxSize);
        ArrayList<ArrayList<Integer>> B = new ArrayList<>(maxSize);

        for (int m = 0; m < maxSize; m++) {
            ArrayList<Integer> data = new ArrayList<>(maxSize);
            for (int n = 0; n < maxSize; n++) {
                data.add((int)(Math.random() * 40) + 1);
            }
            A.add(data);
        }

        for (int m = 0; m < maxSize; m++) {
            ArrayList<Integer> data = new ArrayList<>(maxSize);
            for (int n = 0; n < maxSize; n++) {
                data.add((int)(Math.random() * 40) + 1);
            }
            B.add(data);
        }

        List<ArrayList<ArrayList<Integer>>> res = new LinkedList<>();
        res.add(A);
        res.add(B);
        return res;
    }
```

Figure 4: Multiply and Split-Matrix function

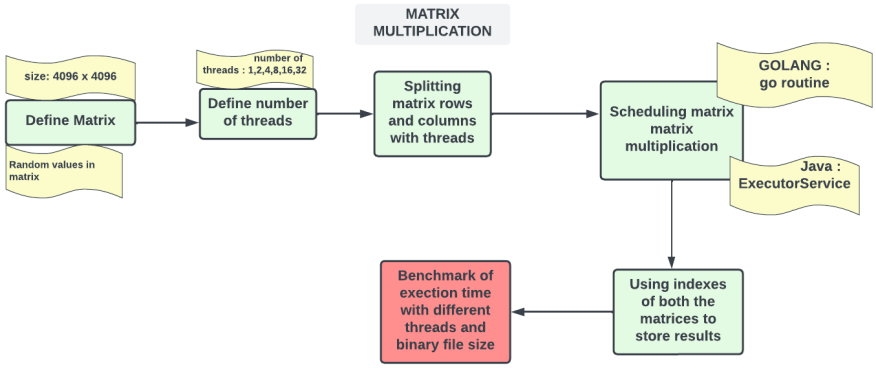

Figure 5: Matrix multiplication in Java and Go - Flow diagram

9.1.1 matrix multiplication Using executor service in java

In matrix multiplication, the numbers of rows are split depending on the number of threads such that the number of columns of the first matrix and the number of rows of the second matrix are equal. Then the multiplication of two matrixes is carried out according to the basic linear algebra logic. The snippet of the multiply and split function is shown in figure 6.

```
public int[][] matrixMultiplication(ArrayList<ArrayList<Integer>> A, ArrayList<ArrayList<Integer>> B, int m, int n) {
    int[][] C = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int k = 0; k < n; k++) {
            int temp = A.get(i).get(k);
            for (int j = 0; j < n; j++) {
                C[i][j] += temp * B.get(k).get(j);
            }
        }
    }
    return C;
}

public ArrayList<ArrayList<ArrayList<Integer>>> splitMatrix(ArrayList<ArrayList<Integer>> A, int nrOfThreads) {
    int n = A.size();
    int m = n / nrOfThreads;
    ArrayList<ArrayList<ArrayList<Integer>>> B = new ArrayList<>();
    for (int i = 0; i < nrOfThreads; i++) {
        B.add(new ArrayList<>(A.subList(i * m, (i + 1) * m)));
    }
    return B;
}
}
```

Figure 6: Multiply and Split-Matrix function

Java does not create OS threads on startup, there is only one main thread. To create more, we can use the ExecutorService interface that allows us to execute tasks asynchronously. For the Matrix multiplication benchmark, ExecutorService is implemented to set the size of the treads from 1 to 16. ExecutorSer-

vice allows the user to create reusable threads in the thread pool so the user can implement many tasks that are handled and executed by the ExecutorService. Reusing threads will help in increasing performance as the problem of creating a new thread for each task is removed. The snippet of the implementation of ExecutiveService is shown in figure 7.

```
    public void Multiply() throws InterruptedException {
        List<ArrayList<ArrayList<Integer>>> matrices = matrix.genMatrix();
        ArrayList<ArrayList<Integer>> A = matrices.get(0);
        ArrayList<ArrayList<Integer>> B = matrices.get(1);

        if (NTHREADS == 1) {
            int n = A.size();
            CountDownLatch latch = new CountDownLatch(NTHREADS);
            executorService.execute(() -> {
                matrix.matrixMultiplication(A, B, n, n);
                latch.countDown();
            });
            latch.await();
        } else {
            ArrayList<int[][]> result = new ArrayList<>();
            int[][] empty = new int[][]{{}};
            for (int i = 0; i < NTHREADS; i++) {
                result.add(empty);
            }

            CountDownLatch latch = new CountDownLatch(NTHREADS);
            ArrayList<ArrayList<ArrayList<Integer>>> workerMatrices = matrix.splitMatrix(A, NTHREADS);

            for (int i = 0; i < NTHREADS; i++) {
                executorService.execute(new Worker(matrix, latch, workerMatrices.get(i), B, i, result));
            }

            latch.await();
        }
    }
}
```

Figure 7: Executor service implementation

9.1.2 matrix multiplication Using goroutines in Go

The matrix multiplication in Go is made similar to that of Java with the Multiply and SplitMatrix function to maintain the same logic in both languages. The snippet of the multiply and split function is shown in figure 8.

```go
func (m Matrix) Multiply(A [][]int, B [][]int) [][]int {
    sizeA := len(A)
    sizeB := len(B)
    n := make([][]int, sizeA)
    for i := range n {
        n[i] = make([]int, sizeB)
    }
    for i := 0; i < sizeA; i++ {
        for k := 0; k < sizeB; k++ {
            temp := A[i][k]
            for j := 0; j < sizeB; j++ {
                n[i][j] += temp * B[k][j]
            }
        }
    }
    return n
}

func (m Matrix) SplitMatrix(nrOfThreads int, matrix [][]int) (matrixes [][][]int) {
    splitter := len(matrix) / nrOfThreads
    for i := 0; i < nrOfThreads; i++ {
        matrixes = append(matrixes,
            matrix[splitter*i:(splitter*(i+1))])
    }
    return
}
```

Figure 8: Multiply and Split-Matrix function

We need to define the number of CPUs that will be executed simultaneously as Go only has one thread by default. For this purpose, Go allows the program to run with GOMAXPROCS by default. Here, we have defined multiple GOMAXPROCS with 1,2,4,8, and 16 threads to get the analysis of the performance of Go with matrix multiplication with different threads. The code is run such that the first matrix is multiplied by the columns of the second matrix at the same time. This is achieved using goroutines for each row of our first matrix and syncing the results with the other matrix. Then the final method is called, and the performance is displayed. The snippet of the matrix multiplication and split code is shown in figure 9.

```
thrd := []int{1, 2, 4, 8, 16}
names := []string{"1-Thread", "2-Thread", "4-Thread", "8-Thread", "16-Thread"}

for i := 0; i < len(names); i++ {
    TNGOS = thrd[i]

    if TNGOS <= 0 {
        runtime.GOMAXPROCS(1)
    } else if TNGOS >= 16 {
        runtime.GOMAXPROCS(8)
    } else {
        runtime.GOMAXPROCS(TNGOS)
    }
```

Figure 9: GoRoutine implementation

9.2 Hashmap Operations

The sync.map is introduced in Go 1.9. It is being said to use it carefully under two following conditions. First, if there is a bottleneck with the already existing use of ma which is type safe. Second, the perfect use of sync.map with its use case . The sync.map is introduced with use-cases where we have high read operations in a multi-core high performance environment. Data structures with sync.RWMutex suffer significantly when there are very high read requests. Hence, sync.Map is introduced in Go 1.9 to cope up with this issue. In our experiment benchmark code, we have chosen 'get' and 'put' operations with 2 threads and make combinations of 'get' and 'put' with different masked values of keys. Also, we have checked the performance of integer map with integer keys and values.
For Java, the commonly used hashmap data structure is Sync-hashmap and concurrent hashmap. We used the same 'get' and 'put' operations with 2 threads and checked their performance with the Java JMH behnchmark library. Below is the difference for both the hashmaps :

Sync Hashmap	ConcurrentHashmap
Locking is done at object level	Locking is done at segment level
operations -Sync all	only updates are synchronised
Introduced in JDK 1.2	Introduced in JDK 1.5
Allow null keys and values	No null keys and null values
Throws exception when error occurs	Does not hrows exception when error occurs

Figure 10: Sync Hashmap VS ConcurrentHashmap

10 Conclusion

10.1 Matrix Multiplication Experiment Results

In the matrix multiplication experiment, the execution time for both Java and Go were almost on the same lines for each increment of thread.The compile time for Java and Go were very much different. Go merely took a few milliseconds to compile the complete code but Java took 7 seconds to build the code. But the size of the GO binary file is 2.5 MB and Java jar file 68KB. For memory bounded systems, Java must be used where we need to be careful with memory allocation or usage.Below is the image of the execution time table and graph :

Matrix Mult. - 4096 x 4096

Threads	GO (ns)	JAVA(ns)	Winner
1	58519214500	57959510100	Java
2	30408301200	31107561400	Go
4	17247798800	18660032200	Go
8	12672717700	11730700900	Java
16	8459766200	8570162101	Go

Figure 11: Execution time table- Java VS GO

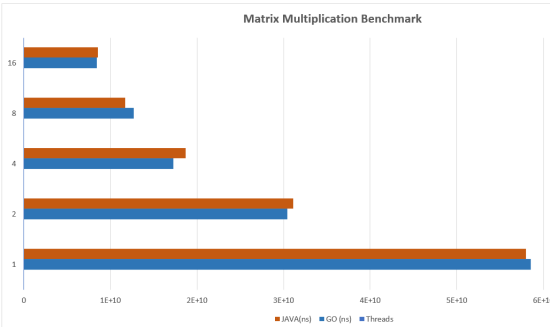

Figure 12: Execution time graph- Java VS GO

10.2 Hashmap Cache Experiment Results

10.2.1 sync.Map(GO) VS ConcurrentHashMap(Java)

Within GO, the sync.Map performs far worse than the map using locks. Within Java, the concurrentHashMap performs better than the map using locks. Between GO and Java, locks perform better in GO than JAVA whereas concurrentHashMap performs better in Java than GO. The reason for the above conclu-

sions is concurrentHashMap allows performing concurrent read and write which is not the case in GO. Below is the image of the execution time table and graph :

MODE(hashmap)	OPERATION	OPS	GO(ns)	JAVA(ns)
Lock	Get	Lock-Get	60.51	103.935
Lock	Put	Lock-Put	79.59	154.931
Lock	PutGet	Lock-PutGet	152.5	362.118
Lock	MultiGet	Lock-MultiGet	85.96	343.382
Lock	MultiPut	Lock-MultiPut	203.8	491.25
Lock	MultiPutGet	Lock-MultiPutGet	288.3	2338.899
Sync	Get	Sync-Get	181.4	63.463
Sync	MultiGet	Sync-MultiGet	239.1	59.069
Sync	MultiPut	Sync-MultiPut	294.8	230.738
Sync	MultiPutGet	Sync-MultiPutGet	644	375.568

Figure 13: Execution time table- Sync vs Lock

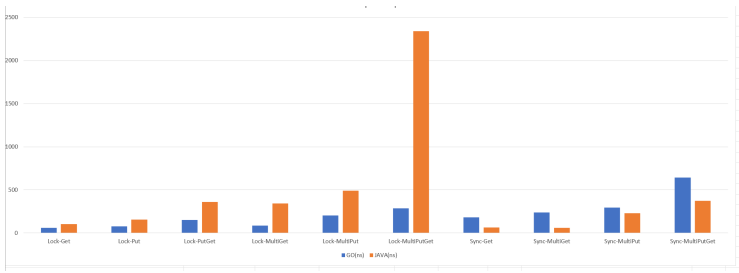

Figure 14: Execution time graph- Sync vs Lock

10.2.2 sync.Map(GO) VS ConcurrentHashMap(Java)

When comparing GO and JAVA, GO's sync.Map for GET and MultiGET operations is extremely slower than JAVA.JAVA wins comprehensively here as Java does a lock-free volatile implementation as compared to GO's sync.map. Below is the image of the execution time table and graph :

MODE(hashmap)	OPERATION	OPS	GO(ns)	JAVA(ns)
Unshared	Get	Unshared-Get	54.1	47.567
Unshared	MultiGet	Unshared-MultiGet	50.64	55.652
Sync	Get	Sync-Get	181.4	63.463
Sync	MultiGet	Sync-MultiGet	239.1	59.069

Figure 15: Execution time table- $unshared_getvsunshared_multiget$

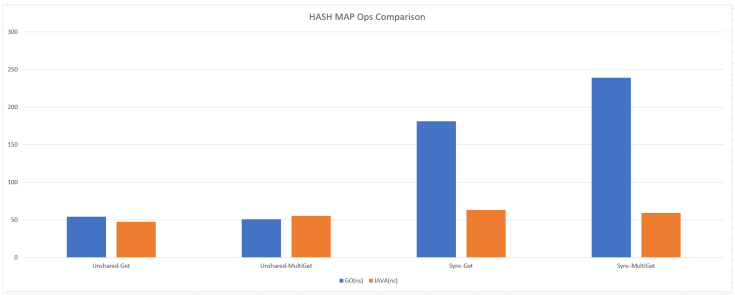

Figure 16: Execution time graph- $unshared_getvsunshared_multiget$

10.2.3 intMap(GO) VS intMap(Java)

Java is better than GO in almost cases of INTMAP because GO uses arrays of structs while JAVA uses link list of nodes.Below is the image of the execution time table and graph :

MODE(hashmap	OPERATION	OPS	GO(ns)	JAVA(ns)
IntMap	Get	IntMap-Get	63.98	35.545
IntMap2	Get	IntMap2-Get	26.68	13.232
IntMap	MultiGet	IntMap-MultiGet	65.52	43.994
IntMap2	MultiGet	IntMap2-MultiGet	29.19	14.753

Figure 17: Execution time table- intMap comparison

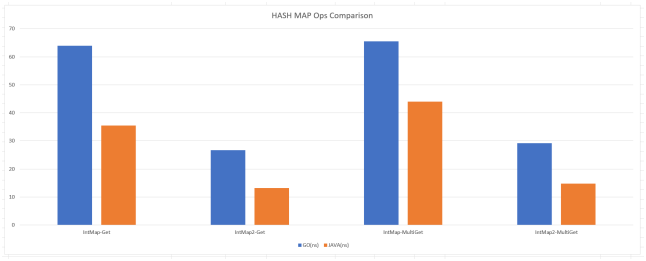

Figure 18: Execution time graph- intMap comparison

References

[1] History of java - javatpoint. https://www.javatpoint.com/history-of-java. Accessed: 2022-5-15.

[2] The java language environment. https://www.oracle.com/java/technologies/introduction-to-java.html. Accessed: 2022-5-15.

[3] Jeff Meyerson. The go programming language. *IEEE software*, 31(5):104–104, 2014.

[4] Frequently asked questions (faq) - the go programming language. https://go.dev/doc/faq-Origins, 2022. Accessed: 2022-5-15.

[5] Difference between concurrency and parallelism - geeksforgeeks. https://www.geeksforgeeks.org/difference-between-concurrency-and-parallelism/., 2022. Accessed: 2022-5-15.

[6] Coding Ninjas. Understanding the pros and cons of concurrency. https://www.codingninjas.com/blog/2021/10/19/understanding-the-pros-and-cons-of-concurrency/., October 2021. Accessed: 2022-5-15.

[7] P Y Abhinav, Avakash Bhat, Christina Terese Joseph, and K Chandrasekaran. Concurrency analysis of go and java. October 2020.

[8] Goroutine vs thread. https://www.geeksforgeeks.org/golang-goroutine-vs-thread/, August 2019. Accessed: 2022-5-15.

[9] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. April 2019.

[10] S Hawthorne. A language comparison between parallel programming features of go and C.

[11] Tobias Andersson and Christoffer Brenden. Parallelism in go and java: A comparison of performance using matrix multiplication. 2018.