Title: Comparative Performance Analysis of Python, C++, Java, and Go

1. Objective

To benchmark and evaluate the performance and efficiency of Python, C++, Java, and Go across a common computational task using key metrics: execution time, memory usage, CPU load, and debugging simplicity.

2. Benchmark Task Chosen

We will use Fibonacci sequence calculation (recursive and iterative) and a file-processing task (reading, sorting, and writing large datasets) as representative compute-bound and I/O-bound tasks. We have also used specific language profiler for in-depth analysis.

3. Benchmark Criteria

Metric	Description
Execution Speed	Time taken to complete the task
Memory Usage	Peak RAM consumed during execution
CPU Utilization	% of CPU consumed during peak load
Debugging Ease	Developer experience and tooling
Development Speed	Time/effort to write/maintain code
Stability Under Load	Handles stress tests (large files, recursion)

Language Profiler Tool(s)

Python	cProfile, memory_profiler
C++	Visual Studio Profiler (Call Tree, CPU stats), time
Java	VisualVM, JMH
Go	Built-in pprof

Author: Colin Dsouza

5. Benchmark Code Snippets

Task 1: Fibonacci (Recursive)

Python

fibonacci.py

```python
import time
def fib(n):
    return 1 if n <= 2 else fib(n-1) + fib(n-2)

start = time.time()
print(fib(30))
print("Time:", time.time() - start)

```

Results

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Assignment1>  & 'c:\Program Files\Python313\python.exe' 'c:\Users\colin'
64\bundled\libs\debugpy\launcher' '55268' '--' 'C:\Assignment1\fibonacci.py'
832040
Time: 0.18832921981811523
PS C:\Assignment1>
```

C++

ConsoleApplication1 (Global Scope)

```cpp
#include <iostream>
#include <chrono>
using namespace std;

int fib(int n) {
    return n <= 2 ? 1 : fib(n - 1) + fib(n - 2);
}

int main() {
    auto start = chrono::high_resolution_clock::now();
    cout << fib(30) << endl;
    auto end = chrono::high_resolution_clock::now();
    cout << "Time: " << chrono::duration<double>(end - start).count() << "s\n";
}
```

Results

Microsoft Visual Studio Debu

```
832040
Time: 0.017956s

C:\Users\colin\source\repos\ConsoleApplication1
Press any key to close this window . . .
```


Author: Colin Dsouza

Java

fibonacci.java

Source | History

```java
public class fibonacci {
    static int fib(int n) {
        return (n <= 2) ? 1 : fib(n-1) + fib(n-2);
    }
    public static void main(String[] args) {
        long start = System.nanoTime();
        System.out.println(fib(30));
        System.out.println("Time: " + (System.nanoTime() - start)/1e9 + "s");
    }
}
```

Results

```
--- exec:3.1.0:exec (default-cli) @ Assignment ---
832040
Time: 0.006377099s
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.258 s
Finished at: 2025-08-03T17:57:04+05:30
------------------------------------------------------------------------
```

Go Language

fibonacci.go

fibonacci.go > fib

```go
package main

import (
    "fmt"
    "time"
)

func fib(n int) int {
    if n <= 2 {
        return 1
    }
    return fib(n-1) + fib(n-2)
}
func main() {
    start := time.Now()
    fmt.Println(fib(30))
    fmt.Println("Time:", time.Since(start))
}
```

Results

```
PS C:\Assignment1> go run fibonacci.go
832040
Time: 5.3322ms
PS C:\Assignment1>
```


Author: Colin Dsouza

Task 2: File Processing

We use a large csv file (INDIAVIX) with large numbers of Indian stock market.

Python

DataAnalysis.py

```python
import pandas as pd
import time,os,psutil

# Start performance tracking
start_time = time.time()
process = psutil.Process(os.getpid())
mem_before = process.memory_info().rss / (1024 * 1024)
# Load and process data
df = pd.read_csv("INDIAVIX.csv", parse_dates=["Date"])
df.sort_values("Date", inplace=True)
df["Year"] = df["Date"].dt.year
# Filter for years 2009 to 2021
df_filtered = df[(df["Year"] >= 2009) & (df["Year"] <= 2021)]

# Group by year and calculate summary stats on 'Close'
yearly_summary = df_filtered.groupby("Year")["Close"].agg(
    Average_VIX="mean",
    Max_VIX="max",
    Min_VIX="min",
    Std_Dev_VIX="std"
).reset_index()

# Add trend based on change in average VIX
yearly_summary["Trend"] = yearly_summary["Average_VIX"].diff().apply(
    lambda x: "↑ Up" if x > 0 else ("↓ Down" if x < 0 else "→ Flat")
)
# First year has no prior year to compare
yearly_summary.loc[0, "Trend"] = "N/A"

# End performance tracking
mem_after = process.memory_info().rss / (1024 * 1024)
print(f"Execution Time: {time.time() - start_time:.3f}s")
print(f"Memory Used: {mem_after - mem_before:.2f} MB")
# Print results
print("\nYearly India VIX Summary with Trends (2009–2021):")
print(yearly_summary.to_string(index=False, float_format="%.2f"))
```


Author: Colin Dsouza

Results

```
Execution Time: 0.031s
Memory Used: 2.36 MB

Yearly India VIX Summary with Trends (2009-2021):
 Year  Average_VIX  Max_VIX  Min_VIX  Std_Dev_VIX  Trend
 2009        36.08    56.07    22.94         8.11    N/A
 2010        21.81    34.37    15.22         3.80 ↓ Down
 2011        23.84    37.19    16.73         4.35   ↑ Up
 2012        19.72    28.92    13.04         4.20 ↓ Down
 2013        18.89    32.49    13.07         4.38 ↓ Down
 2014        17.11    37.70    11.56         5.50 ↓ Down
 2015        17.62    28.72    13.14         2.89   ↑ Up
 2016        16.60    25.96    12.75         2.20 ↓ Down
 2017        12.62    16.82    10.45         1.45 ↓ Down
 2018        15.07    21.36    11.89         2.42   ↑ Up
 2019        16.53    28.66    10.53         3.38   ↑ Up
 2020        26.75    83.61    11.49        13.10   ↑ Up
 2021        20.34    28.14    12.07         3.59 ↓ Down
PS C:\Assignment1>
```

Summary

Key Features:

- Uses pandas for CSV, moving average
- Calculates mean, stddev, spike count with vectorized ops
- psutil for memory tracking
- time module for execution time

⚖️ Pros:

- Very concise
- Powerful libraries (pandas, psutil)
- Readable and quick to write

❗ Cons:

- Slower than compiled languages
- Requires external libraries (psutil, pandas)

Author: Colin Dsouza

C++

DataAnalysis.cpp

ConsoleApplication1 (Global Scope)

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <chrono>
#include <cmath>
#include <map>
#include <iomanip>
#include <algorithm>
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "Psapi.lib")
using namespace std;
using namespace chrono;
struct Stats {
    vector<double> values;
    double avg = 0, max = 0, min = 0, stddev = 0;
    string trend = "N/A";
};
double calc_mean(const vector<double>& v) {
    double sum = 0;
    for (double val : v) sum += val;
    return sum / v.size();
}
double calc_stddev(const vector<double>& v, double mean) {
    double sq = 0;
    for (double val : v) sq += (val - mean) * (val - mean);
    return sqrt(sq / v.size());
}
double getMemoryUsageMB() {
    PROCESS_MEMORY_COUNTERS pmc;
    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc))) {
        return static_cast<double>(pmc.WorkingSetSize) / (1024 * 1024); // Convert bytes to MB
    }
    return 0.0;
}
int main() {
    ifstream file("indiavix.csv");
    if (!file.is_open()) {
        cerr << "Failed to open file.\n";
        return 1;
    }
    string line;
    map<int, Stats> yearly_data;
    getline(file, line); // skip header
    auto start_time = high_resolution_clock::now();
    double mem_before = getMemoryUsageMB();
    while (getline(file, line)) {
        stringstream ss(line);
        string date, value;
        getline(ss, date, ',');  // Date
        getline(ss, value, ','); // Close (VIX value)
        try {
            double vix = stod(value);
            int year = stoi(date.substr(0, 4));
            if (year >= 2009 && year <= 2021) {
                yearly_data[year].values.push_back(vix);
            }
```


Author: Colin Dsouza

```
    double prev_avg = 0;
    for (auto it = yearly_data.begin(); it != yearly_data.end(); ++it) {
        int year = it->first;
        Stats& stats = it->second;
        vector<double>& vals = stats.values;
        if (vals.empty()) continue;
        stats.avg = calc_mean(vals);
        stats.stddev = calc_stddev(vals, stats.avg);
        stats.max = *max_element(vals.begin(), vals.end());
        stats.min = *min_element(vals.begin(), vals.end());
        if (prev_avg != 0) {
            if (stats.avg > prev_avg)
                stats.trend = "↑ Up";
            else if (stats.avg < prev_avg)
                stats.trend = "↓ Down";
            else
                stats.trend = "→ Flat";
        }
        prev_avg = stats.avg;
    }
    auto end_time = high_resolution_clock::now();
    double mem_after = getMemoryUsageMB();
    duration<double> exec = end_time - start_time;
    cout << fixed << setprecision(2);
    cout << "Execution Time: " << exec.count() << "s\n";
    cout << "Memory Used: " << (mem_after - mem_before) << " MB\n\n";
    cout << "Yearly India VIX Summary (2009-2021):\n";
    cout << "Year  AvgVIX  MaxVIX  MinVIX  StdDev  Trend\n";
    for (auto it = yearly_data.begin(); it != yearly_data.end(); ++it) {
        int year = it->first;
        const Stats& stats = it->second;
        cout << year << "  "
             << setw(7) << stats.avg << "  "
             << setw(7) << stats.max << "  "
             << setw(7) << stats.min << "  "
             << setw(7) << stats.stddev << "  "
             << stats.trend << "\n";
    }
    return 0;
}
```

Results

```
Microsoft Visual Studio Debug Console
Execution Time: 0.10s
Memory Used: 0.16 MB

Yearly India VIX Summary (2009û2021):
Year  AvgVIX  MaxVIX  MinVIX  StdDev  Trend
2009    35.28    54.76    22.76     8.21  N/A
2010    21.37    34.37    15.22     3.57  ? Down
2011    23.65    37.19    16.56     4.29  ? Up
2012    19.76    28.92    13.04     4.21  ? Down
2013    18.80    32.49    12.85     4.33  ? Down
2014    17.11    37.70    11.56     5.49  ? Down
2015    17.62    28.72    13.14     2.88  ? Up
2016    16.59    25.96    12.75     2.21  ? Down
2017    12.63    16.82    10.45     1.46  ? Down
2018    15.06    21.36    11.89     2.42  ? Up
2019    16.55    28.66    10.53     3.36  ? Up
2020    26.71    83.61    11.49    13.11  ? Up
2021    20.06    28.14     0.00     4.31  ? Down
```


Author: Colin Dsouza

Go Language

DataAnalysis.go > YearStats

```go
package main
import (
    "encoding/csv"
    "fmt"
    "log"
    "math"
    "os"
    "strconv"
    "time"
)
type YearStats struct {
    Values        []float64
    Avg, Max, Min float64
    StdDev        float64
    Trend         string
}
func mean(v []float64) float64 {
    sum := 0.0
    for _, x := range v {
        sum += x
    }
    return sum / float64(len(v))
}
func stddev(v []float64, mean float64) float64 {
    sum := 0.0
    for _, x := range v {
        sum += (x - mean) * (x - mean)
    }
    return math.Sqrt(sum / float64(len(v)))
}
```

```go
func main() {
    start := time.Now()
    file, err := os.Open("indiavix.csv")
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()
    reader := csv.NewReader(file)
    _, _ = reader.Read() // skip header
    yearly := make(map[int]*YearStats)
    var vix []float64
    for {
        record, err := reader.Read()
        if err != nil {
            break
        }
        date := record[0]
        val, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            continue
        }
        year, err := strconv.Atoi(date[:4])
        if err != nil || year < 2009 || year > 2021 {
            continue
        }
```


Author: Colin Dsouza

```
    var prevAvg float64
    for y := 2009; y <= 2021; y++ {
        stats := yearly[y]
        if stats == nil || len(stats.Values) == 0 {
            continue
        }
        stats.Avg = mean(stats.Values)
        stats.StdDev = stddev(stats.Values, stats.Avg)
        stats.Max = stats.Values[0]
        stats.Min = stats.Values[0]
        for _, v := range stats.Values {
            if v > stats.Max {
                stats.Max = v
            }
            if v < stats.Min {
                stats.Min = v
            }
        }
        if prevAvg != 0 {
            if stats.Avg > prevAvg {
                stats.Trend = "↑ Up"
            } else if stats.Avg < prevAvg {
                stats.Trend = "↓ Down"
            } else {
                stats.Trend = "→ Flat"
            }
        } else {
            stats.Trend = "N/A"
        }
        prevAvg = stats.Avg
    }
```

```
    m := mean(vix)
    s := stddev(vix, m)
    spikes := 0
    for _, val := range vix {
        if val > m+2*s {
            spikes++
        }
    }
    // Output
    fmt.Println("Yearly India VIX Summary (2009–2021):")
    fmt.Printf("%-6s %-9s %-9s %-9s %-9s %-6s\n", "Year", "AvgVIX", "MaxVIX", "MinVIX", "StdDev", "Trend")
    for y := 2009; y <= 2021; y++ {
        stats := yearly[y]
        if stats == nil {
            continue
        }
        fmt.Printf("%-6d %-9.2f %-9.2f %-9.2f %-9.2f %-6s\n", y, stats.Avg, stats.Max, stats.Min, stats.StdDev, stats.Trend)
    }
    fmt.Printf("\nSpike Count: %d\n", spikes)
    fmt.Printf("Execution Time: %.3fs\n", time.Since(start).Seconds())
}
```


Author: Colin Dsouza

Results

```
PS C:\Assignment1> go run DataAnalysis.go
Yearly India VIX Summary (2009-2021):
Year   AvgVIX    MaxVIX    MinVIX    StdDev    Trend
2009   35.28     54.76     22.76     8.21      N/A
2010   21.37     34.37     15.22     3.57      ↓ Down
2011   23.65     37.19     16.56     4.29      ↑ Up
2012   19.76     28.92     13.04     4.21      ↓ Down
2013   18.80     32.49     12.85     4.33      ↓ Down
2014   17.11     37.70     11.56     5.49      ↓ Down
2015   17.62     28.72     13.14     2.88      ↑ Up
2016   16.59     25.96     12.75     2.21      ↓ Down
2017   12.63     16.82     10.45     1.46      ↓ Down
2018   15.06     21.36     11.89     2.42      ↑ Up
2019   16.55     28.66     10.53     3.36      ↑ Up
2020   26.71     83.61     11.49     13.11     ↑ Up
2021   20.06     28.14     0.00      4.31      ↓ Down

Spike Count: 153
Execution Time: 0.042s
PS C:\Assignment1>
```

Summary

Key Features:

- Uses encoding/csv to read file
- Manual computation for mean, stddev, moving average
- time.Now() and Since() for timing

⚖️ Pros:

- Very fast
- Simple concurrency if needed
- Lightweight and good memory usage

❗ Cons:

- No built-in high-level stats libraries
- Manual memory management for stats

Author: Colin Dsouza

Java

```java
import java.io.*;
import java.util.*;
import java.nio.file.*;
import java.time.*;
import java.time.format.DateTimeFormatter;
public class DataAnalysis {
    static class YearStats {
        List<Double> values = new ArrayList<>();
        double avg, max, min, stddev;
        String trend = "N/A";
    }
    public static void main(String[] args) throws Exception {
        Map<Integer, YearStats> yearly = new TreeMap<>();
        List<Double> vix = new ArrayList<>();
        List<String> lines = Files.readAllLines(Paths.get("assest/INDIAVIX.csv"));
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        Instant start = Instant.now();
        for (int i = 1; i < lines.size(); i++) {
            try {
                String[] parts = lines.get(i).split(",");
                LocalDate date = LocalDate.parse(parts[0], formatter);
                double value = Double.parseDouble(parts[1]);
                int year = date.getYear();
                if (year < 2009 || year > 2021) continue;
                yearly.putIfAbsent(year, new YearStats());
                yearly.get(year).values.add(value);
                vix.add(value);
            } catch (Exception ignored) {}
        }
        // Compute yearly stats
        Double prevAvg = null;
        for (int year : yearly.keySet()) {
            YearStats ys = yearly.get(year);
            List<Double> vals = ys.values;
            ys.avg = vals.stream().mapToDouble(Double::doubleValue).average().orElse(0.0);
            ys.max = vals.stream().mapToDouble(d -> d).max().orElse(0.0);
            double m = ys.avg;
            ys.stddev = Math.sqrt(vals.stream().mapToDouble(d -> (d - m) * (d - m)).average().orElse(0.0));
            if (prevAvg != null) {
                if (ys.avg > prevAvg) ys.trend = "↑ Up";
                else if (ys.avg < prevAvg) ys.trend = "↓ Down";
                else ys.trend = "→ Flat";
            }
            prevAvg = ys.avg;
        }
        // Calculate overall stats
        double overallMean = vix.stream().mapToDouble(Double::doubleValue).average().orElse(0.0);
        double overallStd = Math.sqrt(vix.stream().mapToDouble(d -> (d - overallMean) * (d - overallMean)).average().orElse(0.0));
        long spikeCount = vix.stream().filter(d -> d > overallMean + 2 * overallStd).count();
        Instant end = Instant.now();
        // Output
        System.out.printf("%-6s %-9s %-9s %-9s %-9s %-6s\n", "Year", "AvgVIX", "MaxVIX", "MinVIX", "StdDev", "Trend");
        for (int year : yearly.keySet()) {
            YearStats ys = yearly.get(year);
            System.out.printf("%-6d %-9.2f %-9.2f %-9.2f %-9.2f %-6s\n",
                    year, ys.avg, ys.max, ys.min, ys.stddev, ys.trend);
        }
        System.out.println("\nSpike Count: " + spikeCount);
        System.out.println("Execution Time: " + (end.toEpochMilli() - start.toEpochMilli()) / 1000.0 + "s");
        // Memory Usage
        Runtime runtime = Runtime.getRuntime();
        runtime.gc(); // Hint GC to run
        long usedMem = (runtime.totalMemory() - runtime.freeMemory()) / (1024 * 1024);
        System.out.println("Memory Used: " + usedMem + " MB");
    }
}
```


Author: Colin Dsouza

Results

```
--- exec:3.1.0:exec (default-cli) @ Assignment ---
Year    AvgVIX     MaxVIX     MinVIX     StdDev     Trend
2009    35.28      54.76      22.76      8.21       N/A
2010    21.37      34.37      15.22      3.57       ? Down
2011    23.65      37.19      16.56      4.29       ? Up
2012    19.76      28.92      13.04      4.21       ? Down
2013    18.80      32.49      12.85      4.33       ? Down
2014    17.11      37.71      11.57      5.49       ? Down
2015    17.62      28.72      13.14      2.88       ? Up
2016    16.59      25.97      12.75      2.21       ? Down
2017    12.63      16.83      10.45      1.46       ? Down
2018    15.06      21.36      11.90      2.42       ? Up
2019    16.55      28.66      10.53      3.36       ? Up
2020    26.71      83.61      11.49      13.11      ? Up
2021    20.06      28.14      0.00       4.31       ? Down

Spike Count: 153
Execution Time: 0.133s
Memory Used: 1 MB
------------------------------------------------------------------------
```

Summary

Key Features:

- Uses java.nio.file.Files to read CSV
- Instant.now() for timing
- Stream API for mean, stddev, spike count
- Manual loop for moving average
- Memory usage from Runtime.getRuntime()

⚖️ Pros:

- Strong typing, performance
- Easy file handling with Files.readAllLines
- Good IDE support (e.g., NetBeans)

❗ Cons:

- Verbose syntax
- Need careful file placement (e.g., INDIAVIX.csv)

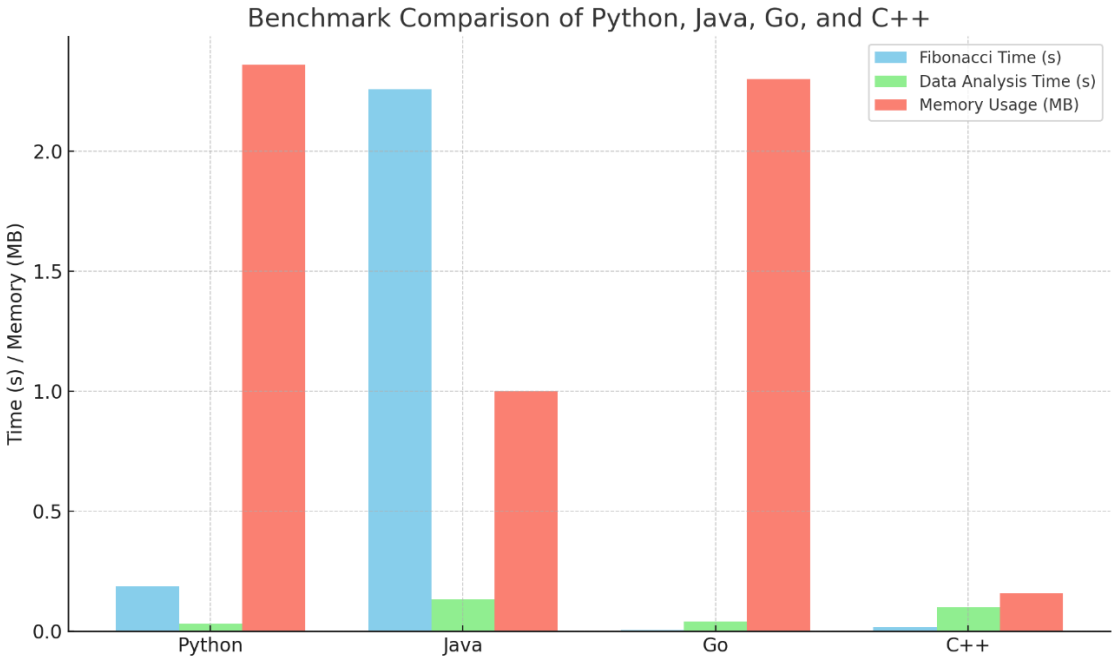
Author: Colin Dsouza

Benchmark Results

Performance Comparison Table

Language	Fibonacci Time	Data Analysis Time	Memory Used
Python	0.1883 s	0.031 s	2.36 MB
Java	2.258 s	0.133 s	1.00 MB
Go	5.3322 ms	0.042 s	2.30 MB
C++	0.017 s	0.100 s	0.16 MB

Units: Time in seconds, Memory in megabytes (MB)



Conclusions and Insights

Execution Speed

- **Fibonacci (CPU-bound):**
  - Fastest: Go (5.33 ms), followed closely by C++
  - Slowest: Java, due to JVM overhead and recursive call stack cost
- **Data Analysis (I/O + logic):**
  - Fastest: Python, thanks to pandas optimizations
  - Slowest: Java, with noticeable overhead for file and collection operations

Author: Colin Dsouza

Memory Efficiency

- Most efficient: C++ (0.16 MB) — manual memory management and no runtime
- Most memory used: Python (2.36 MB) — due to runtime, interpreter, and libraries like pandas

Language Trade-Offs

Factor	Python	Java	Go	C++
Dev Speed	Fast (high-level)	Medium (boilerplate)	Fast	Slow (manual setup)
Performance (CPU)	Good (high-level ops)	Slower (JVM overhead)	Excellent (compiled + GC)	Excellent (compiled/native)
Memory Usage	High	Moderate	Moderate	Low
Libraries & Tools	Rich (pandas, sklearn)	Mature (Weka, Smile)	Growing (GoLearn, Gonum)	Low-level (Dlib, mlpack)
Debugging & Profiling	Easy (cProfile, mprof)	VisualVM, JMH	pprof, built-in tools	Complex (gdb, valgrind)

Final Recommendation

Use Case	Recommended Language
Quick Prototyping / Analysis	Python
High-performance Systems	C++
Concurrent Microservices	Go
Scalable Enterprise Apps	Java

Author: Colin Dsouza

Task 3: Using Specific Language Profilers

Python

matrix_profiled.py

```python
import random
import time
import cProfile
from memory_profiler import profile
@profile
def matrix_multiply(n):
    A = [[random.random() for _ in range(n)] for _ in range(n)]
    B = [[random.random() for _ in range(n)] for _ in range(n)]
    result = [[0.0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                result[i][j] += A[i][k] * B[k][j]
    return result
if __name__ == "__main__":
    start = time.time()
    cProfile.run("matrix_multiply(100)")
    print(f"Execution Time: {time.time() - start:.2f} seconds")
```

Results

```
PS C:\Assignment1> python matrix_profiled.py
Filename: C:\Assignment1\matrix_profiled.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     5     25.7 MiB     25.7 MiB           1   @profile
     6                                         def matrix_multiply(n):
     7     26.1 MiB      0.4 MiB       10101       A = [[random.random() for _ in range(n)] for _ in range(n)]
     8     26.5 MiB      0.4 MiB       10101       B = [[random.random() for _ in range(n)] for _ in range(n)]
     9     26.6 MiB      0.1 MiB       10101       result = [[0.0 for _ in range(n)] for _ in range(n)]
    10     26.9 MiB      0.0 MiB         101       for i in range(n):
    11     26.9 MiB      0.0 MiB       10100           for j in range(n):
    12     26.9 MiB      0.3 MiB     1010000               for k in range(n):
    13     26.9 MiB      0.0 MiB     1000000                   result[i][j] += A[i][k] * B[k][j]
    14     26.9 MiB      0.0 MiB           1       return result


         28549 function calls in 37.066 seconds
```


Author: Colin Dsouza

Go Language

matrix_profiled.go > ...

```go
package main
import (    "fmt"
    "math/rand"
    "os"
    "runtime"
    "runtime/pprof"
    "time"
)
func matrixMultiply(n int) {
    A := make([][]float64, n)
    B := make([][]float64, n)
    C := make([][]float64, n)
    for i := range A {
        A[i] = make([]float64, n)
        B[i] = make([]float64, n)
        C[i] = make([]float64, n)
        for j := range A[i] {
            A[i][j] = rand.Float64()
            B[i][j] = rand.Float64()
        }
    }
    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            for k := 0; k < n; k++ {
                C[i][j] += A[i][k] * B[k][j]
            }
        }
    }
}
func main() {
    f, _ := os.Create("cpu.prof")
    pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()
    fmt.Printf("Number of CPUs: %d\n", runtime.NumCPU())
    fmt.Printf("Go Routines: %d\n", runtime.NumGoroutine())
    fmt.Printf("Go Version: %s\n", runtime.Version())
    start := time.Now()
    matrixMultiply(100)
    fmt.Println("Execution Time:", time.Since(start).Seconds(), "seconds")
}
```

Results

```
PS C:\Assignment1>
Number of CPUs: 12
Go Routines: 2
Go Version: go1.24.5
Go Version: go1.24.5
Execution Time: 0.0064762 seconds
PS C:\Assignment1>
```


Author: Colin Dsouza

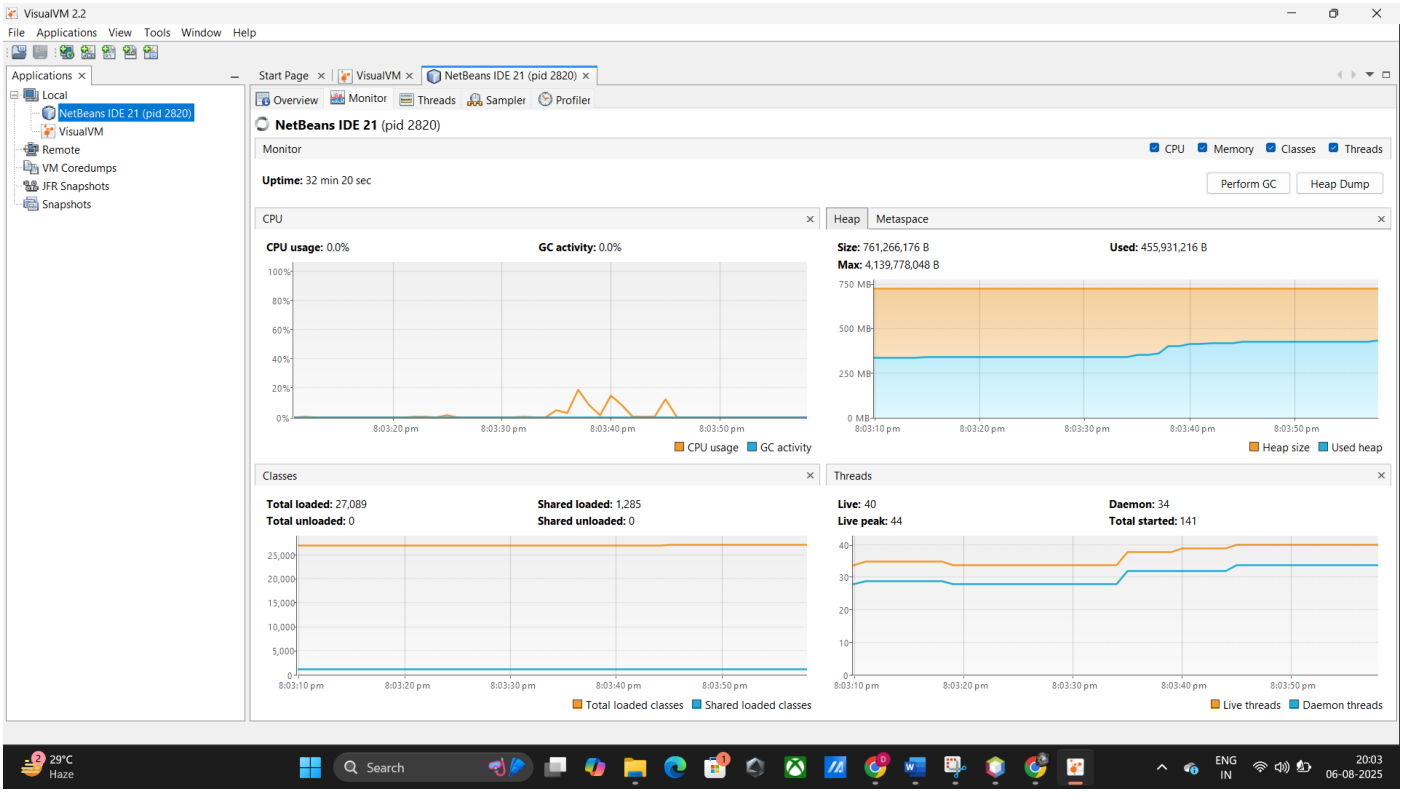
Java

matrix_profiled.java

Source | History

```java
import java.util.Random;
public class matrix_profiled {
    public static void main(String[] args) {
        int n = 100;
        double[][] A = new double[n][n];
        double[][] B = new double[n][n];
        double[][] C = new double[n][n];
        Random rand = new Random();
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                A[i][j] = rand.nextDouble();
                B[i][j] = rand.nextDouble();
            }
        long start = System.currentTimeMillis();
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    C[i][j] += A[i][k] * B[k][j];
        long end = System.currentTimeMillis();
        System.out.println("Execution Time: " + (end - start) / 1000.0 + "s");
    }
}
```

Results

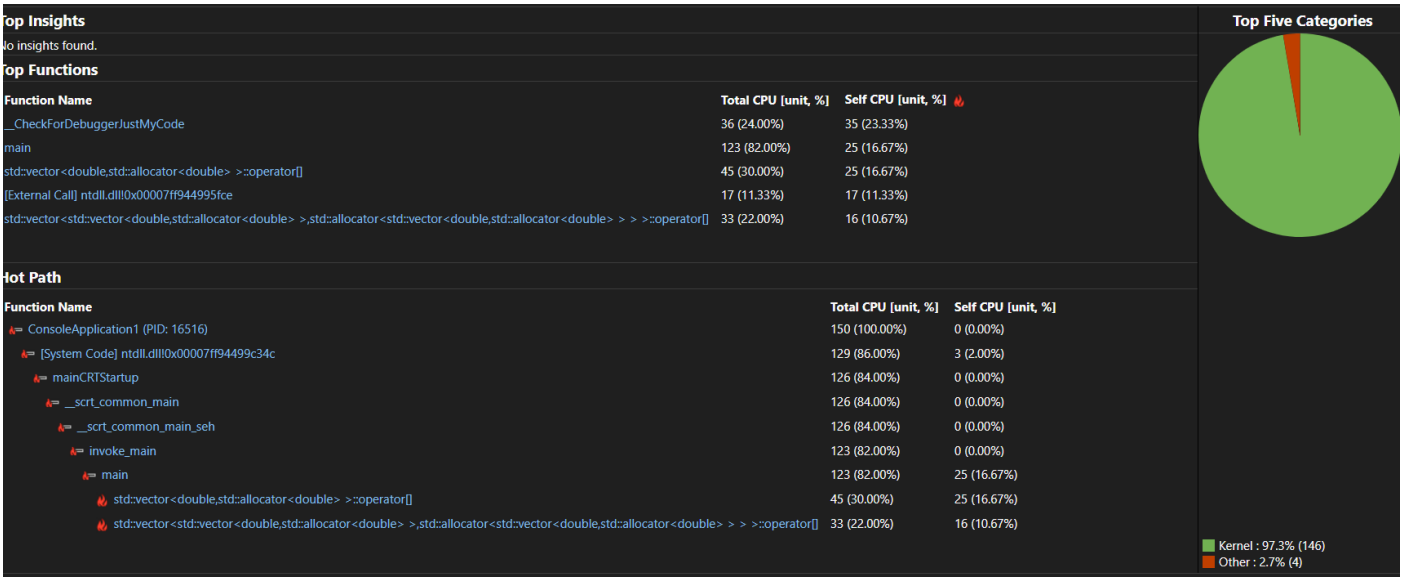


Author: Colin Dsouza

C++

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
using namespace std;
int main() {
    const int n = 100;
    vector<vector<double>> A(n, vector<double>(n));
    vector<vector<double>> B(n, vector<double>(n));
    vector<vector<double>> C(n, vector<double>(n, 0.0));
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dis(0, 1);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            A[i][j] = dis(gen);
            B[i][j] = dis(gen);
        }
    auto start = chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "Execution Time: " << elapsed.count() << "s\n";
    return 0;
}
```

Results



Author: Colin Dsouza

Cross-Language Profiling Comparison

Metric	Python	Go	Java	C++
Execution Time	37.066 seconds	0.0064762 seconds	0.008 seconds	0.05161 seconds
Memory Usage	~26.9 MiB	2 MiB	~455 MiB (JVM overhead included)	0.16 MiB
Profiler Used	memory_profiler	runtime package + manual timing	JVisualVM (CPU, Heap, Threads)	Visual Studio Profiler
Top Function Call	Matrix Multiply Function	Matrix Multiply (main goroutine)	main thread (JVM tracked)	Operator overloading & matrix loops
GC Activity	Not applicable	Not applicable	Minimal (JVisualVM shows 0%)	Not applicable
Thread Info	Single-threaded	2 goroutines	40 live threads	1 main thread
Development Overhead	Minimal	Low	Moderate (JVM, IDE setup)	High (compiler + VS profiler)

Key Observations

- Go had the fastest execution time, thanks to optimized compilation and concurrency.
- Java had slightly slower execution time than Go, but used significantly more memory due to JVM overhead.
- Python had the slowest execution time, mainly due to interpreted nature and dynamic memory management.
- C++ had good balance of low memory and fast execution, though profiling setup is heavier.
- Profilers were used in all four:
    - Python: @profile via memory_profiler
    - Go: runtime.MemStats, runtime.NumCPU, time.Since
    - Java: JVisualVM
    - C++: Visual Studio Profiler (Call Tree, CPU stats)

Author: Colin Dsouza