Benchmarking Python, C, C++, Go and Java on Numerical Workloads

Name: Michael Fernandes

UID: 2059006

Roll No: 06

Abstract

This study benchmarks five widely-used programming languages—Python, Java, C, C++, and Go—to evaluate their performance characteristics across diverse compute-bound workloads. The selected programs include:

• Odd number enumeration up to 1 million
• Fibonacci series generation (1 million terms)
• N-body gravitational simulation
• Deep recursive function calls

These workloads were chosen to stress different aspects of system behavior: loop performance, memory allocation and access patterns, floating-point arithmetic, function call overhead, and stack management.

Each implementation was designed to be algorithmically equivalent to ensure fairness. Through microbenchmarks, we expose intrinsic runtime costs such as loop overhead, allocation patterns, bounds checking, recursion limits, garbage collection pressure, and JIT compilation effects.

Our analysis focuses on execution speed, memory usage, concurrency support, runtime safety, and profiling complexity, highlighting trade-offs between low-level control in compiled languages and productivity in managed or memory-safe environments. This study aims to guide developers and researchers seeking language-level performance insights for CPU-intensive applications.

1. Introduction

Programming languages differ significantly in how they balance execution speed, memory consumption, concurrency capabilities, and developer ergonomics. We benchmark Python, Java, C, C++, and Go using a common suite of compute-bound workloads to compare:

• Raw performance (CPU time)
• Memory behavior and allocation patterns
• Concurrency and threading model efficiency
• Runtime stability and safety guarantees
• Ease of development, profiling, and observability

Benchmark Workloads

1. Enumerating Odd Numbers up to 1 Million

• Purpose: Tests raw iteration performance, branch handling, and integer operations.
• Insights: Highlights loop overhead and runtime dispatching costs.

2. Computing the Fibonacci Series
• Variants: Iterative, recursive, and memoized.
• Purpose: Measures recursion support, function call cost, and stack depth handling.
• Insights: Exposes deep call costs and opportunities for memoization/tail-call optimization.

3. N-body Gravitational Simulation
• Purpose: Intensive floating-point arithmetic with nested loops.
• Insights: Highlights CPU throughput, floating-point performance, cache behavior, and memory access efficiency.

4. Deep Recursive Function Calls
• Purpose: Tests recursion limit, stack safety, and call overhead.
• Insights: Indicates runtime design (stack growth, tail recursion support, interpreter overhead).

2. Method

2.1 Platform & Tooling

To ensure a consistent benchmarking environment, all programs were implemented natively in Python, Java, C, C++, and Go with algorithmically equivalent logic. Minimal platform-specific optimizations were applied to maintain fairness.

Operating Systems:
• Windows 11: Python, Java, Go
• WSL2 Ubuntu (Linux): C, C++

Profiling Tools:
• Python: cProfile, tracemalloc, gc module
• Java: VisualVM, MXBeans, manual wall-clock timing
• C/C++: gprof, valgrind (massif)
• Go: pprof for CPU & memory profiling, trace for scheduler analysis, runtime stats (GOMAXPROCS, HeapAlloc, GC cycles)

Hardware: Same physical system for all benchmarks (exact CPU & RAM specs not recorded).

2.2 Workloads Overview

Workload	Description	Stresses
Fibonacci + Odd	Simple integer loops; 1 million iterations.	Loop performance, integer ALU, low allocation
N-body Simulation	Floating-point math with nested loops and pairwise interactions.	Arithmetic throughput, memory access, $O(n^2)$
Deep Recursion	Recursively computes values to depth N.	Stack management, function call overhead

2.3 Execution & Measurement

Execution Time Measurement:
- Python: time.time()
- Java: System.nanoTime()
- C/C++: std::chrono, clock_gettime()
- Go: Built-in time package with time.Since() + pprof timestamps

Memory Usage Measurement:
- Python: tracemalloc, gc
- Java: VisualVM for heap and GC stats
- C/C++: valgrind (massif), gprof allocation reports
- Go: pprof memory profiles, runtime heap stats

3. Results

3.1 Runtime (Execution Time)

- Fibonacci+Odd (1M) → Go fastest (~8–12 ms), C++ (~70 ms), Java (~32 ms), C (~180 ms), Python slowest (~21.05 s).
- N-body Simulation → C++ fastest (~110 ms), C (~240 ms), Java (~420 ms), Go competitive for small n (1.57 s) but scales poorly (76 s for n=5000), Python slow (~2.34 s).
- Deep Recursion → Go fastest (~6.65–9.10 ms), C++ (~90 ms), C (~150 ms), Java (~230 ms), Python slowest (~1.02 s).

3.2 Memory Use (Peak Allocation)

- C most efficient (1 KB–100 KB).
- Go low baseline (0.22–6.68 MB).
- C++ higher due to STL (0.12–7.7 MB).
- Java highest baseline (40–60 MB).
- Python moderate (0.3–5.2 MB).

3.3 Garbage Collection & Threading

• Go → Concurrent GC (0–0.53 ms pauses), goroutines scale well.
• Java → Regular GC (2–3 ms pauses), good threading after JIT warm-up.
• Python → Minimal GC, GIL blocks true parallelism.
• C/C++ → No GC, full OS threads, manual memory management.

3.4 Overall Summary

• Fastest (micro): Go
• Fastest (heavy numeric): C++
• Most Memory-Efficient: C
• Best Developer Productivity: Python
• Best Balance (Speed + Tooling): Java
• Most Predictable GC: Go

4. Comparative Analysis

4.1 Performance

• C/C++ → Fastest for large-scale numeric workloads.
• Go → Beats Java/C++ on some low-allocation microbenchmarks; C++ leads in heavy FP simulations.
• Java → Competitive after JIT warm-up; good for long-running workloads.
• Python → Slowest (interpreter + GIL).

4.2 Memory Footprint

• C → Most efficient (~KBs).
• C++ → Efficient, even with STL.
• Go → Low baseline (KBs–few MBs).
• Java → Higher baseline heap (50–75 MB).
• Python → Small heap via tracemalloc, higher RSS.

4.3 Development Ergonomics

• Python → Fastest to write/debug.
• Java → Rich IDEs, strong profilers.
• Go → Simple syntax, fast compiles.
• C/C++ → Max control, slower iteration.

4.4 Stability Under Load

• All stable.
• Go → Negligible GC pauses (<1 ms).
• Java → Short, predictable GC pauses.
• C/C++ → No leaks (Valgrind).
• Python → GC not stressed.

4.5 Debugging & Observability

Java: VisualVM, JFR → GC, allocation, threads
C/C++: Valgrind, gprof → Leaks, access, hotspots
Python: cProfile, tracemalloc → Calls, object memory
Go: pprof, trace → Heap, goroutines, GC

4.6 Concurrency Model

- Python → GIL; use multiprocessing.
- Java → OS threads, thread pools.
- Go → Goroutines + channels, concurrent GC.
- C/C++ → True OS threads.

4.7 Developer Productivity

- Python → Best for prototyping.
- Java → Balanced for large apps.
- Go → Great for backends.
- C/C++ → Most control, steep learning.

4.8 Bottom-line

Tight numeric / low-level → C / C++
Long-running service → Java / Go
High-concurrency backend → Go
Prototyping, scripting → Python
Small, low-allocation tasks → Go

5. Discussion of Trade-offs

5.1 Performance vs. Development Time

- Python → Slow but fast to prototype.
- C/C++ → Fastest with optimization, slowest to develop.
- Java → Near-native after warm-up.
- Go → Fastest builds, great profiling.

5.2 Reliability vs. Control

- C++ → Maximum control, riskier.
- Java/Python → Safe, less control.
- Go → Memory-safe with concurrent GC.

5.3 Memory

- C/C++ → Minimal; C ~1 KB, C++ ~7.7 MB.
- Java → ~50–805 MB.
- Python → ~0.3 MB baseline.
- Go → ~0.22–6.68 MB.

5.4 Comparative Summary Table

Aspect	Python	C/C++	Java	Go
Loop Performance	Slowest (GIL + interpreter)	Fastest (native, optimized)	Near-native (JIT)	Near-native (AOT)
Dev Time	Fastest to code	Longest	Medium-fast	Fast
Memory Control	Low	Highest	Medium	Medium
Reliability	High	Lower without discipline	High	High
Baseline Memory	Medium (~10s MB RSS)	Minimal (KB–MB)	High (50+ MB)	Low (~0.2–6 MB)
Best Use Case	Rapid prototyping, glue code	Performance-critical systems	Long-running scalable services	High-concurrency, low-latency services

6. Threats to Validity

6.1 Non-Uniform Workloads

• Implementation differences (e.g., memory use in Fibonacci).
• Loop boundaries and recursion varied slightly.

6.2 JIT Warm-up Effects (Java)

• Java's JIT requires steady-state benchmarks.
• Go avoids this with AOT compilation.

6.3 Profiler Overhead

• Valgrind slows C/C++.
• Python tools miss native costs.
• VisualVM & pprof add overhead in short runs.

6.4 Memory Measurement Discrepancies

• Python tracemalloc ≠ RSS.
• Java/Go exclude native/OS memory.
• OS-level normalization advised.

6.5 GC & Runtime Behavior

• Java → GC pauses.
• Python → GC may not trigger.
• C/C++ → No GC.
• Go → Concurrent GC, minimal pause.

6.6 Platform & Environment Variability

• Windows/WSL2 used.
• OS, CPU scaling, and background processes not fully locked.

6.7 Compiler Flags & Build Optimization

• C/C++ flags: -O3, -march=native
• Java JVM options
• Go build flags matter
• Python interpreters not varied.

6.8 Runtime Cost Assumptions

• Python → interpreter startup.
• Java → JVM init cost.
• C/C++ → minimal startup.
• Go → tiny scheduler/GC init.

7. Conclusion

This comparative study evaluated the runtime efficiency, memory usage, garbage collection behavior, concurrency support, and developer ergonomics of five popular programming languages—C, C++, Java, Python, and Go—across a diverse set of computationally intensive tasks: Fibonacci + Odd Number generation, N-body simulation, Deep Recursion, and N-gram string processing.

7.1 Key Findings

Performance

• C++ was fastest on heavy numeric workloads (e.g., N-body) via low-level optimizations, inlining, and efficient std::thread usage.
• C closely followed, especially in memory-light tasks (manual memory, minimal runtime overhead).
• Go led on low-allocation microbenchmarks (Fibonacci+Odd) and stayed competitive on recursion-heavy tasks.
• Java reached near-native performance after JIT warm-up, strong in loop-heavy and concurrent runs.
• Python trailed due to interpreter overhead and the GIL, but remains fine for moderate workloads and rapid prototyping.

Memory Usage

• C had the smallest footprint when avoiding heap allocations.
• C++ stayed efficient, with slight overhead from STL containers.
• Go kept a low baseline (hundreds of KBs) in low-allocation workloads, scaling modestly on heavier tasks.
• Java carried consistent overhead from class metadata, JIT cache, and GC-managed heap.
• Python showed higher per-object costs from interpreter structures.

Concurrency

• Java, C, and C++ leveraged OS threads for parallelism.
• Go used lightweight goroutines and an efficient scheduler (low thread creation cost).
• Python was limited by the GIL; multiprocessing or native extensions are required for CPU-bound parallelism.

Developer Experience & Tooling

• Python: Shortest development cycle; simple profiling (cProfile, tracemalloc).
• Java: Strong profilers (VisualVM, JFR), robust threading, memory safety.
• Go: Simple syntax, fast compiles, integrated profiling (pprof, trace)—great for backend/concurrency.
• C/C++: Most demanding but unmatched control and optimization potential.

7.2 General Recommendations

• Maximum performance in CPU-bound numeric/simulation tasks → C++ or C
• Balanced speed, safety, and tooling for long-running concurrent apps → Java or Go
• Rapid experimentation & integration with external libraries → Python (NumPy, Numba, Cython)
• High-concurrency backend / microservices → Go

7.3 Summary Table

Parameter	**Python**	**Java**	**C**	**C++**	**Go**
Runtime Speed	Slow	Good	Very Fast	Very Fast	Very Fast (low-allocation)
Memory Usage	High	Medium (GC overhead)	Very Efficient	Efficient	Low (predictable GC)
Concurrency	Threading limited by GIL	Multithreading	Multithreading	Multithreading	Goroutines + channels
Compilation Time	N/A (Interpreted)	Slow	Fast	Often Slow (templates)	Fast
Memory Safety	GC-managed	GC-managed	Manual & unsafe	Manual & unsafe	GC-managed
Startup Time	Fast	Slow	Fast	Fast	Fast

Parameter	Python	Java	C	C++	Go
Binary Size	N/A	Large	Small	Medium-Large	Small
Developer Productivity	High	High	Low (verbose)	Medium	High
Ecosystem & Libraries	Huge (AI, DS, Web)	Mature	Smaller	Rich (games, systems, ML)	Growing (cloud, backend)
Cross-Platform	Yes	Yes	Yes	Yes	Yes
Portability	High	High	High	High	High
Learning Curve	Easy	Medium	Steep	Steep	Easy–Medium
Performance-Critical Apps	Not suitable	Moderate	Excellent	Excellent	Excellent (for concurrency)
Embedded Systems	No	Rare	Excellent	Good	Limited
GUI Development	Possible (Tkinter, PyQt)	Good (JavaFX, Swing)	Rare	Good (Qt, wxWidgets)	Limited
Best Use Cases	AI, Scripting, Data Science	Web, Enterprise, Android	Systems, OS, Real-Time	Games, High-Perf Apps	Backend services, networking
Type System	Dynamic	Static (Strong)	Static	Static	Static (Strong)
Garbage Collection	Yes	Yes	No	No	Yes (concurrent)
Low-level Hardware Access	No	Limited	Yes	Yes	Limited