Comparing Execution Times of Processes in Different Programming Languages

Student: Tudor Giuroiu

November 18, 2023

Contents

1 Introduction

1.1 Context

The aim of this project is to provide a useful comparison tool for execution times of different processes in different programming languages. In our case, we have C++, Java and Python.

C++, known for its high-performance capabilities and low-level control, will be one of our primary languages of interest. Java, recognized for its portability and robustness due to the Java Virtual Machine (JVM), presents another important candidate for comparison. Lastly, Python, known for its simplicity and readability, will be included to represent a dynamically-typed and interpreted language.

1.2 Objectives

The tool allows the user to conduct a thorough performance analysis of critical operations in C++, Java, and Python, including static and dynamic memory allocation, memory access, thread creation, context switch, and thread migration. Users gain valuable insights into the efficiency of each language, allowing for informed comparisons and the selection of the most suitable language based on specific performance criteria.

2 Bibliographic Research

2.1 What is benchmarking?

Benchmarking is a method used for evaluating the performance characteristics of computer hardware or software.

Software benchmarking focuses on evaluating the performance of applications, operating systems, or other software solutions. This process involves running standardized tests or simulations to measure factors like response time, throughput, and resource utilization. Software benchmarking is particularly valuable for developers, as it helps identify inefficiencies, or areas for improvement in their code.

Isolating tests in a benchmark is a very important task to ensure the accuracy and reliability of performance evaluations. By keeping tests separate and independent, it becomes possible to measure changes in performance of specific hardware, software, or configurations. Interference between tests could lead to misleading results, as the impact of one test may inadvertently affect the outcomes of others. Isolation provides a controlled environment, allowing for a focused examination of individual components or functionalities. This practice is particularly critical in microbenchmarking.

Microbenchmarking is a focused performance evaluation method that analyzes the speed and efficiency of small, specific code segments or functions within a software system.

2.2 Key Terms

Memory allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes. There are 2 types of memory allocation, static and dynamic.

Static memory allocation is a memory allocation technique where the size of the memory needed for a variable is determined at compile time and remains constant throughout the program's execution.

Dynamic memory allocation is a memory allocation technique where the size of the memory needed for a variable is determined at runtime and can change during the program's execution. In this approach, memory is allocated and deallocated as needed, allowing for flexibility in managing memory resources.

Memory access refers to the process of reading or writing data in the computer's memory.

Thread creation refers to the process of creating a new thread of execution within a program. A thread is the smallest unit of execution within a process, and it represents an independent sequence of instructions that can be scheduled to run concurrently with other threads.

Thread context switch is the process by which a computer's central processing unit (CPU) switches its attention from one thread of execution to another.

Thread migration refers to the act of moving a running thread from one logical processor to another within a multiprocessor system.

3 Analysis

To achieve a thorough assessment, three distinct environments will be established, each dedicated to one of the specified programming languages. These environments are meticulously designed to measure the performance characteristics of each language variant across critical processes, including dynamic and static memory allocation, memory access, thread creation, thread context switch, and thread migration.

The creation of these distinct environments ensures a focused and accurate evaluation of each programming language's capabilities. By isolating the environments, the project facilitates an independent assessment of the performance metrics for C++, Java, and Python. Users can seamlessly access information specific to their language of interest without interference from the other programming environments.

Not only can the three environments run independently, but users also have the option to concurrently observe the results of the benchmarks side by side. This functionality enables users to make direct comparisons, facilitating a nuanced analysis of the performance characteristics across different programming languages.

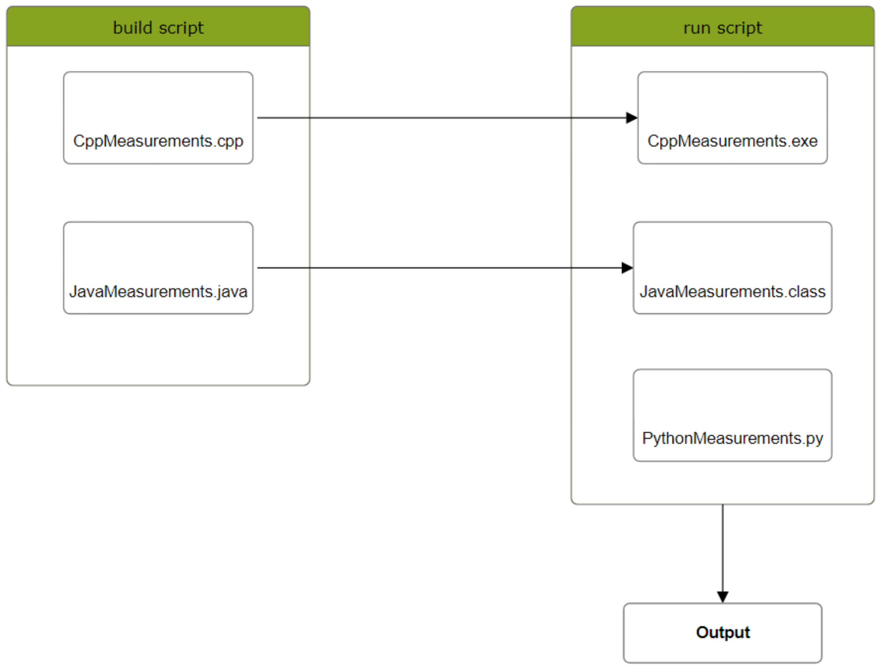
In addition to the isolation of programming environments, it is necessary to separate the tests on various processes. This strategic approach achieves precise and accurate benchmarking. Each benchmark test should be executed independently to ensure that the results are uncontaminated by interference from other tests.

4 Design

4.1 Project Architecture

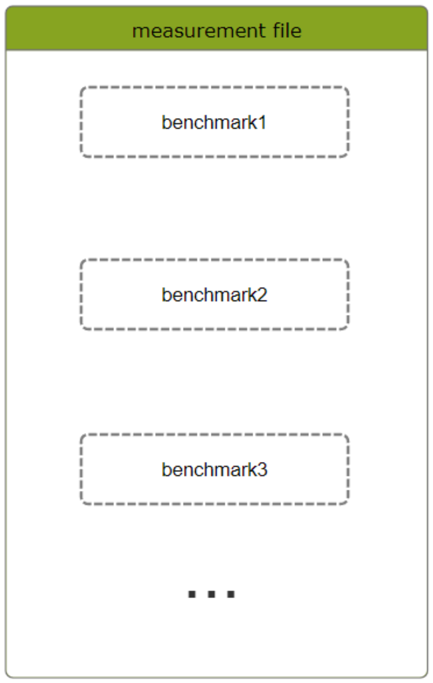
The project will contain a set of scripts that will streamline the building and execution processes for both C++ and Java programs. These scripts are designed to automate the compilation of source code, resulting in executable files that can be easily executed for benchmarking. This approach ensures consistency in the build process and simplifies the setup of the project and the execution of subsequent performance tests.

Additionally, a script will be created to facilitate the sequential execution of the C++, Java, and Python programs, each residing in its designated environment. This script orchestrates the execution of the benchmark tests, running them one after the other to collect comprehensive performance data. The output of the benchmarks will be presented in a clear and organized manner, providing users with a detailed view of the results for each distinct operation.



To maintain the integrity of the benchmark measurements, each of the three programs (C++, Java, and Python) will be well structured. Each program will feature separate functions dedicated to executing the required benchmarks. This compartmentalization ensures the isolation of different benchmarks, eliminating potential interference between measurements. By encapsulating each benchmark within its own function, the project achieves a modular and organized structure, boosting code readability and maintainability.

Furthermore, the benchmark tests themselves will incorporate a robust methodology. Instead of relying on a single measurement, the tests will perform an average of multiple measurements for each operation. This averaging process is crucial for obtaining accurate and reliable results, as it helps mitigate the impact corner cases or external factors that may affect individual measurements. The pursuit of an average ensures that the benchmark results reflect the consistent performance characteristics of the programming languages across various operations, contributing to the overall reliability of the performance evaluation.



4.2 Strategy for benchmarking

Given the importance of accuracy in benchmarking, the project aims to enhance precision by adopting a substantial sample size. In order to obtain a more reliable result, the algorithm implemented for each test follows an average calculation. This approach involves conducting measurements iteratively, adding every particular measurement time to a variable, and at the end, dividing by the number of times it was repeated.

Algorithm 1: Calculate Average Time

```
Data: Number of tries (TRIES)
Result: Average time (sum/TIMES)
sum = 0;
for i = 0; i < TRIES; i++ do
    // Measure the time of the process
    time = MeasureTimeOfProcess();
    sum += time;
// Calculate the average time
return sum / TRIES;
```

5 Implementation

5.1 Script text files

The script text files responsible for building the C++ and Java programs and for running all 3 programs are written in Windows batch format.

The **build.bat** script is designed to compile the source code of C++ and Java programs, preparing them for execution. It simplifies the build process by encapsulating the necessary compilation commands.

The **run.bat** script facilitates the execution of all three programs: the compiled C++ program (**CppMeasurements.exe**), the compiled Java program (**JavaMeasurements.class**), and the Python script (**PythonMeasurements.py**). The output of these programs are displayed in the command prompt generated by run.bat .

5.2 CppMeasurements.cpp

As C++ is designed as a low-level language, it allows developers with strong memory management capabilities and greater control. Furthermore, it facilitates the handling of threads and processes more seamlessly compared to many other programming languages. These advantages enable the execution of all necessary benchmarks.

To ensure isolation, the chosen methodology adopts modularization through functions. Each benchmark is executed within its own function, utilizing the approach showed in Algorithm 1.

`'long staticMemoryAllocationTime()'` calculates the average time it takes to perform static memory allocation in a loop. It uses the Time Stamp Counter (__rdtsc()) to measure CPU cycles. Within each iteration, it allocates an integer array (p) of size SIZE_OF_ARRAY and records the time before and after allocation. The total time is then averaged over the specified number of tries (TRIES). The result is converted to milliseconds, taking into account the processor's clock speed (PROCESSOR_GHZ) and using the formula:

#seconds = #cycles / frequency

For the following benchmarks the C++ <chrono> library is used to capture the time points before and after the measured process.

`'long dynamicMemoryAllocationTime()'` measures the average time taken for dynamic memory allocation in a loop. The allocated memory is for an integer array of size SIZE_OF_ARRAY. The function returns the average time in nanoseconds for the specified number of tries (TRIES), encompassing both allocation and deallocation using malloc and free, but measuring just the allocation.

`'long staticMemoryAccessTime()'` calculates the average time required for accessing. The function returns the average access time in nanoseconds over a specified number of tries (TRIES), including an additional sum of accessed values (aux_sum) to avoid compiler optimizations. The final result returned is the average access time.

`'long dynamicMemoryAccessTime()'` computes the average time required for accessing elements in a dynamically allocated array. It allocates memory for an integer array p using malloc, initializes it with random values, and then measures the time taken for repeated access operations within a loop. The function returns the average access time in nanoseconds over a specified number of tries (TRIES).

`'long threadCreationTime()'` calculates the average time required for creating a thread using the POSIX threads library (pthread). The function repeatedly creates a thread and measures the time taken for each creation operation. The result is the average thread creation time in nanoseconds over a specified number of tries (TRIES). The tc_function being passed to the thread simply returns NULL.

`'long threadContextSwitchTime()'` measures the average time required for a context switch between the main process thread and a secondary thread. It uses the POSIX threads library (pthread) to create and join threads and employs the Windows API functions for setting processor affinity. The secondary thread's function (tcsw_function) records the time when it is executed, meaning that the context has been switched, and the main thread calculates the context switch start time. The result is the average of the differences between the two values in nanoseconds over a specified number of tries (TRIES).

`'long threadMigrationTime()'` calculates the average time required for migrating a thread from one logical processor to another. It uses the POSIX threads library (pthread) to create and join threads and utilizes the Windows API functions for setting processor affinity. The thread migration function (tmgr_function) sets the processor affinity of the current process twice, recording the time taken for the affinity changes. The main thread then calculates the thread migration time. The result is the average thread migration time in nanoseconds over a specified number of tries (TRIES).

The **ADDITIONAL_OUTPUT_MACRO** is a preprocessor directive in your C++ code that controls the inclusion of additional output statements for debugging or informative purposes. When this macro is defined, specific statements within the code are activated, providing additional information during the execution of the program. These statements include details about processor affinity, thread ID, or success statuses of certain system calls.

5.3 JavaMeasurements.java

In Java, static memory allocation is not explicitly supported, as the language operates within the Java Virtual Machine (JVM) and uses automatic memory management through garbage collection. Unlike lower-level languages like C++, where developers have direct control over memory allocation and deallocation, Java abstracts these details. Additionally, measuring thread context switch and thread migration in Java is challenging due to the high-level abstractions provided by the language. Java abstracts away low-level details such as processor affinity and thread management, making it difficult to precisely measure the time associated with these operations.

`'private static long dynamicMemoryAllocationTime()'` calculates the average time required for dynamic memory allocation in a loop. It uses the System.nanoTime() method to measure the time before and after allocating an integer array (array) of a specified size (SIZE_OF_ARRAY). The method then accumulates the time differences across multiple iterations, returning the average allocation time in nanoseconds over the specified number of tries (TRIES).

`'private static long dynamicMemoryAccessTime()'` computes the average time required for accessing elements in a dynamically allocated integer array. The method initializes an array with random values, and then, within a loop, measures the time it takes to access each element individually using a nested loop. The access times are accumulated over multiple iterations, and the result returned is the average access time in nanoseconds over the specified number of tries (TRIES).

`'private static long threadCreationTime()'` calculates the average time required for creating a new thread using the Thread class. The method initiates a loop, within which it measures the time before and after creating a new thread instance. The threads are set as daemons to avoid waiting for their completion, ensuring that the measurements focus on the thread creation process. The cumulative time differences across multiple iterations are then divided by the number of tries (TRIES) to yield the average thread creation time in nanoseconds.

The command **java -Djava.compiler=NONE JavaMeasurements** is used to disable the Just-In-Time (JIT) compiler in the Java Virtual Machine (JVM) during program execution. The -Djava.compiler=NONE option is a system property that instructs the JVM to run the Java program without applying any Just-In-Time compilation optimizations. This command must be executed to accurately measure memory access without the JVM intervening.

5.4 PythonMeasurements.py

As in Java, in Python, explicit support for static memory allocation is absent, given the language's reliance on automatic memory management within the confines of the Python interpreter. That is the reason why certain benchmarks cannot be performed.

`'dynamic_memory_allocation_time()'` function performs a task similar to the previously described Java method `'private static long` `dynamicMemoryAllocationTime()'`. Both functions aim to measure the average time taken for dynamic memory allocation within a loop.

The function `'dynamic_memory_access_time()'` and its counterpart in Java, `'private static long dynamicMemoryAccessTime()'`, share the goal of measuring the average time required for accessing elements in a dynamically allocated array or list in a similar way.

The function `'thread_creation_time()'` is dedicated to gauging the average time required for the creation of threads using the threading module. Within a loop executed for a specified number of tries (TRIES), the function dynamically generates threads, each associated with the thread_function. Timing functions are utilized to measure the duration between the initiation and start of each thread. The cumulative time differences across multiple iterations are then divided by the number of tries to yield the average thread creation time in nanoseconds.

6 Conclusion

In summary, the project presents in a simple way the time in nanoseconds for each process, delivering valuable information into the characteristics of the programming languages involved.

As the project's creator, I found it interesting to dive into the distinctions among these three programming languages, gaining a deeper understanding of their respective limitations and the intricacies of thread context switching and migration.

7 Bibliography

Stack Overflow

Geeks For Geeks