UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Execution Time measurement of processes in different programming languages.

In Java, C++ and Python

-Technical documentation-

Știrb Călin-Alexandru

Technical University of Cluj Napoca

16/01/2024

Multi-Language benchmarking

Contents:

1. Introduction

1.1. Context

The goal of this project is to design and implement a benchmark, and compare the results, for memory and thread operations in different programming languages, namely Java, C++ and Python. The main operations that the project is aiming to test are memory allocation and access (both static and dynamic), thread creation, thread context switch and thread migration.

1.2. Specifications

Since multiple programming languages are utilized for this project, the platforms used in the implementation of the projects are Microsoft Visual Studio for the C++ and the Python app and IntelliJ IDEA for the Java application. The data gathered from each of the ran test/batches of tests will be exported to an excel spreadsheet for storage and analysis.

1.3. Objectives

Compile and compare the results from each of the benchmarks ran in the different programming languages and formulate a conclusion as to which of the chosen programming languages handle the CPUs resources most efficiently.

2. Bibliographic study

2.1. Java

Java uses an automatic memory management system that takes care of memory allocation and deallocation, which helps to prevent common programming errors like memory leaks and buffer overflows.

a. Memory allocation
   - **Heap memory**: in Java, objects are created on the heap, a region of memory reserved for dynamic memory allocation; this is done using the keyword ‘new’.
   - **Stack memory**: the Java Virtual Machine (JVM) uses the stack for managing method calls and local variables; each method invocation creates a new stack frame.

b. Garbage collection
   - Java has a built-in garbage collector that automatically reclaims memory by identifying and cleaning up objects no longer reachable, thus preventing memory leaks.

c. Memory access
   - **References**: objects are accessed through references; a reference is a variable that stores the memory address of an object on the heap (kind of like a pointer but with abstractions)
   - **Null references**: a reference can be set to ‘null’; accessing such an object will result in a ‘NullPointerException’.
   - **Array access**: arrays are a common data structure which allows the storage of multiple elements of the same data type.

In java, a thread represents a separate path of execution within a program, allowing for multiple tasks to run concurrently. Java multi-threading is a part of the 'java.lang.Thread' class.

- Threads can be created in 2 main ways: by extending the 'Thread' class or implementing the 'Runnable' interface.
- Thread states: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, and TERMINATED.
- Synchronization is done through synchronized blocks and methods as well as the 'java.util.concurrenct' package.

2.2. C++

In C++ memory allocation and access are more manual and low-level compared to Java. C++ provides greater control over memory, but it also requires more careful management of memory resources.

a. Memory allocation

- **Stack memory:** is used for storing local variables, including objects; this memory is automatically managed by the language. When a function is called, a stack frame is created which includes space for local variables.
- **Heap memory:** is used for objects that need to persist beyond the scope of a function or have a dynamic lifetime; such allocation is done using the operators 'new' and 'malloc' and can be deallocated using 'delete' or 'free'.

b. Memory access

- **Pointers:** in C++ you can work with pointers, which are variables that store memory addresses; pointers are used to access and manipulate memory directly.
- **References:** C++ also supports references, which are basically aliases for objects; they provide a safer and more convenient way to access and modify objects.
- **Arrays:** C++ supports arrays for storing multiple elements of the same data type.

C++ provides multi-threading support through the '<thread>' header and 'std::thread' class. Threads allow multiple tasks to run concurrently within a program.

- Threads can be created by instantiating objects of the std::thread class and passing the function or callable object to be executed in the thread's constructor.
- C++ provides various mechanisms for thread synchronization, including mutexes, condition variables and atomic operations.
- C++ provides functions for managing threads such as starting, stopping and querying their status.

2.3 Python

Memory allocation and access in Python are handled automatically by the Python interpreter, making Python a high-level language that abstracts many low-level memory management details. However, it is still essential to understand how memory management works in Python.

a. Memory allocation

- **Dynamic typing:** Python is a dynamically typed language, which means that variables are determined at runtime.
- **Heap memory:** objects and data structures are stored in a region of memory known as the heap; the heap is responsible for dynamic memory allocation.
- **Reference counting:** Python uses a reference counting mechanism to keep track of the number of references to objects.
- **Memory pools:** Python uses memory pools to efficiently allocate and manage memory for small objects; they reduce the overhead of memory allocation and deallocation.

b. Memory access

- **References:** In Python variables are references to objects in memory; when creating a variable and assigning it a value, it is essentially creating a reference to an object.
- **Garbage collection:** just like Java, Python has a garbage collector that periodically reclaims memory from objects no longer reachable.
- **Data Structures:** various data structures like lists, dictionaries, sets and tuples, for automatic memory allocation handling.
- **Slicing**: Python supports slicing of sequences like strings, lists and tuples allowing access to portions of data efficiently.

In Python, just like Java and C++, threads are used to allow multiple tasks to run concurrently within a program. Python provides a built-in module called 'threading' that simplifies the creation and management of threads.

- Threads are created by instantiating objects of the 'Thread' class and passing a function or callable object which represents the code to be executed.
- Python's 'threading' module provides synchronization primitives like locks, semaphores, and condition variables to prevent race conditions.
- The module also provides functions for managing threads such as starting, stopping, and querying their status.

3. Analysis

3.1. Memory Benchmarking

- Memory Allocation Benchmark

The memoryAllocationBM function measures the time taken to dynamically allocate memory for an array of a specified length. Memory allocation time depends on factors such as available memory, system load, and memory fragmentation. Dynamic allocation typically involves finding contiguous memory blocks, which may lead to varied allocation times based on the system state.

- Memory Access Benchmark

The memoryAccessBM function evaluates the time taken to initialize an array and access its elements sequentially. It assesses the memory fetch time and cache behavior. Sequential access often exhibits better performance due to cache prefetching, minimizing cache misses. Larger arrays or non-sequential access patterns may impact access times due to cache inefficiencies.

3.2. Thread Benchmarking

- Thread Creation Benchmark

The threadCreationBM function measures the time taken to create multiple threads concurrently. Thread creation involves allocating resources such as stacks and thread control blocks. The performance might vary based on the number of threads created, system load, and available CPU cores.

- Thread Context Switch Benchmark

The threadContextBM function attempts to measure the time taken for a context switch between threads. Context switching involves saving and restoring a thread's execution context, including processor registers and stack pointers. This process can be influenced by the OS scheduler, CPU architecture, and the number of active threads.

- Simulated Thread Migration Benchmark

The threadMigrationBM function simulates thread migration by setting thread affinity to a specific core. Real thread migration involves moving a thread's execution context between CPU cores, affected by OS scheduling policies and system load. Simulated migration, as implemented, doesn't fully replicate actual thread migration scenarios.

4. Design

The Micro-Benchmarking application follows the Model-View-Controller (MVC) architectural pattern, aiming to conduct and visualize benchmarks for different programming languages. It comprises three primary classes: Gui in the org.example.View package, Controller in the org.example.Control package, and auxiliary classes to facilitate benchmarking operations.

Model-View-Controller (MVC) Architecture

The MVC architecture separates application logic into three interconnected components:

- Model: responsible for handling data, computations, and interactions with benchmarking scripts and backend logic.
- View: Represented by the Gui class, responsible for the graphical user interface (GUI) presentation and user interactions.
- Controller: Represented by the Controller class, acting as an intermediary between the view and model. It responds to user inputs, triggers actions, and coordinates data flow between the GUI and underlying functionalities.

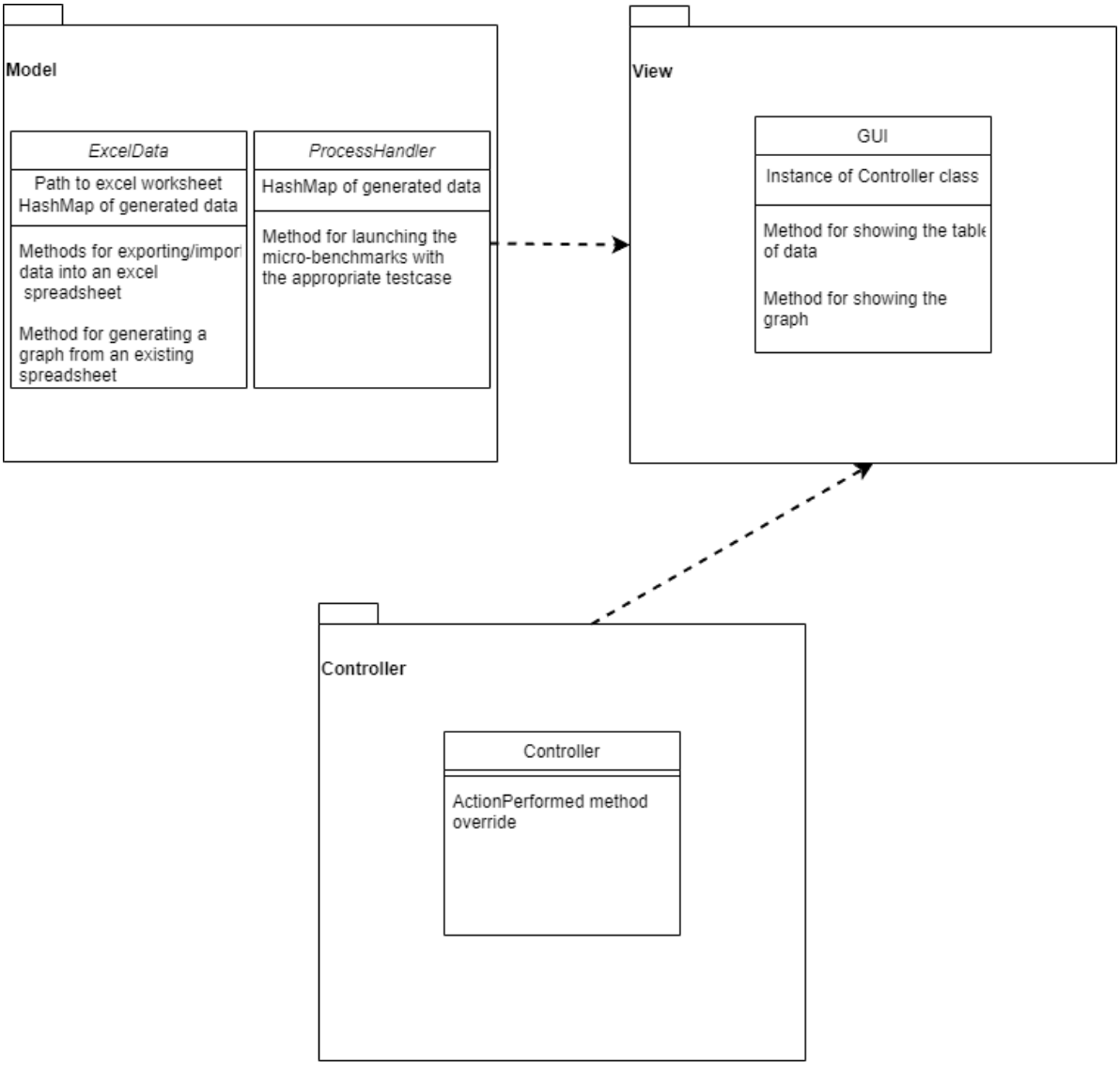


Figure 1. Diagram of the package view of the main application – an MVC architecture

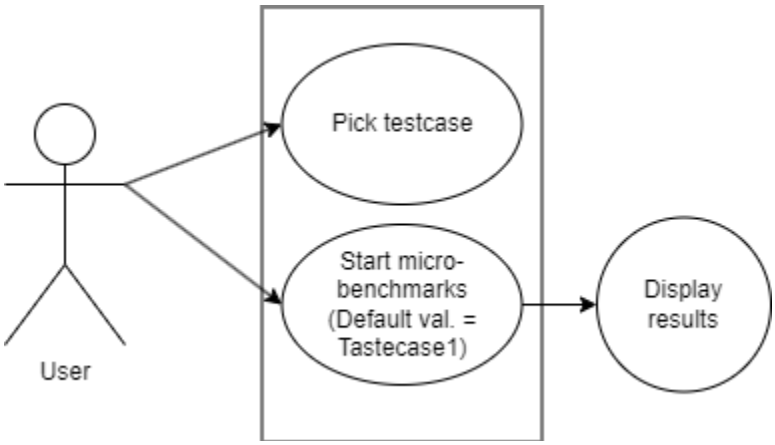


Figure 2. Usecase diagram

- Gui Class
    - Manages the graphical user interface.
    - Key Components:
        - JFrame, JPanel, and various JButton elements: Facilitate the creation and arrangement of GUI components.
        - createTable method: Generates a data table from the benchmark results stored in files, presenting them in a tabular format for user inspection.
- Controller Class
    - Coordinates user interactions and benchmarking operations.
    - Key Components:
        - actionPerformed method: Listens for user-triggered events (button clicks) and initiates corresponding benchmarking procedures.
        - runBenchmark method: Coordinates the execution of benchmarks in multiple languages (Python, Java, C++) by calling external scripts/executables and collecting benchmark data.

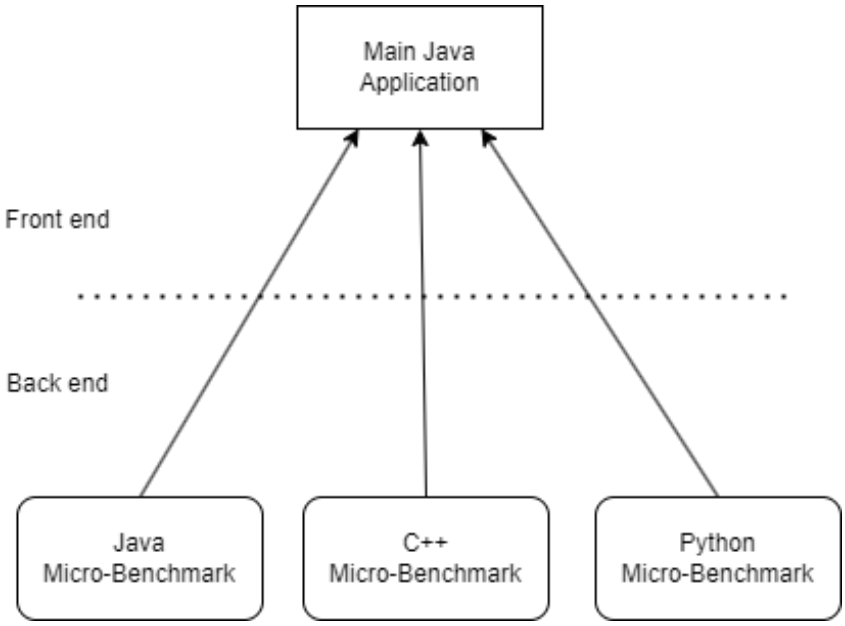


Figure 3. Application process breakdown – connections between processes

- Workflow Overview
  - User interaction triggers a specific benchmark (testCase1, testCase2, testCase3) in the GUI.
  - The corresponding event in the Controller class initiates benchmark execution in Python, Java, and C++ by invoking external scripts/executables.
  - Benchmarking scripts generate benchmark results (PythonBM.txt, JavaBM.txt, CppBM.txt).
  - The Controller class retrieves and displays benchmark results in tables via the Gui class.

5. Implementation

5.1. Benchmark implementation

- Memory allocation benchmark:
  - Measures the time it takes to allocate an array of given length by directly allocating an array variable.
- Memory access benchmark:
  - Measures the time it takes to access the elements in an already initialized array.
- Thread creation benchmark:
  - Measures the time it takes to create a thread.
- Thread context switch benchmark:
  - Measures the time it takes to switch between threads by measuring only the time it takes to switch from one thread to another when joining them.
- Thread migration benchmark:
  - Simulates thread migration by setting the core we wish the thread/process to be run on.

5.2. Main application implementation

5.2.1. GUI implementation:

The class sets up a GUI for benchmarking tests, providing buttons to trigger different tests and a method to display benchmark data in tabular form fetched from a file. The structure allows users to interact with the application through a graphical interface, initiate benchmark tests, and visualize collected data.

- Class Variables:
    - Static JLabels and JButtons: info, info2, testCase1, testCase2, testCase3 are GUI elements to display information and trigger benchmark tests.
    - JPanel and JFrame: mainPanel and mainFrame are the main components for arranging the GUI elements.
- Constructor:
    - Initialization: The constructor initializes the main panel and frame, sets their sizes, titles, and other properties.
    - Background Image: Loads and sets a background image for the GUI.
    - Controller Initialization: Initializes a Controller object, likely for managing user actions on buttons.
    - GUI Elements: Creates JLabels and JButtons for displaying information and triggering different benchmark tests.

- createTable Method:
    - Reads Benchmark Data: Reads benchmark data from a file and organizes it into a table structure.
    - Data Parsing: Parses the file to retrieve benchmark data, categorizes it based on certain patterns in the file, and stores it in a Map (data) with categories as keys and lists of benchmark results as values.
    - Table Population: Prepares a two-dimensional array (rowData) to store data for display in a JTable. Populates this array with the retrieved data.
    - TableModel and JTable Creation: Creates a DefaultTableModel using the array and column names, and then generates a JTable with this model.
    - Table Display: Opens a new JFrame displaying the created JTable within a JScrollPane.

5.2.2. Controller implementation:

The Controller class manages button-click events from the GUI, triggers benchmark executions for various programming languages, and handles the execution of benchmarks by interfacing with the operating system's command line interface to compile and run the benchmarks.

- Class Variables:
    - Gui Reference: Contains a reference to the Gui class to control its elements.

- Constructor:
    - Initialization: Initializes the Controller object with a reference to the GUI (Gui object).

- actionPerformed Method (Implements ActionListener):
    - Event Handling: Listens for button clicks on the GUI and performs actions accordingly.
    - Button Identification: Identifies the source of the action using e.getSource() and performs specific actions for each button.
    - Benchmark Execution: Calls runBenchmark method based on the button clicked and creates tables using createTable method in the GUI for different benchmark types.

- runBenchmark Method:
    - Benchmark Execution: Initiates benchmarks for various programming languages (Java, C++, Python) based on the provided argument.
    - Compilation and Execution: Constructs commands for compilation and execution of benchmark programs in different languages.
    - Launching Benchmarks: Executes these commands using launchBenchmark method.
- launchBenchmark Method:
    - File Existence Check: Checks if the specified benchmark file exists.
    - Process Execution: Uses ProcessBuilder to execute commands in the command prompt (cmd) for compilation and execution of benchmarks.
    - Error Handling: Handles exceptions, prints error streams, and exit codes if the process encounters any issues.

5.2.3. BenchmarkData Class Implementation

The BenchmarkData class processes benchmark data from a file and provides access to different benchmark metrics for a programming language.

- Class Variables:
    - ArrayLists: memoryAllocation, memoryAccess, threadCreation, threadContextSwitch, threadMigration to store split sections of numerical values.
    - fileName: The name of the file to be processed.
- Constructor:
    - Constructs a BenchmarkData object with the specified file name and initializes ArrayLists to store the split sections of numerical values.
- splitFile Method:
    - Splits the content of the file into different ArrayLists based on identified sections.
    - Each section contains numerical values and is stored in a respective ArrayList.
- getComputerSpecs Method:
    - Retrieves the specifications of the user's computer, including the operating system (OS), CPU model, and the quantity of RAM memory.
    - Returns a formatted string containing the computer specifications.

- getRAMInfo Method:
    - Retrieves the total RAM memory available on the system.
    - Returns a string representation of the total RAM in GB.
- getCPUName Method:
    - Retrieves the CPU name by executing the 'wmic cpu get name' command in the command prompt.
    - Parses the output to extract the CPU name from the command output.
    - Returns a String representing the CPU name.

5.2.4. Graph Class Implementation

The Graph class generates comparison graphs based on the BenchmarkData of three different programming languages. It uses the JFreeChart library to create line charts for various benchmark metrics.

- Class Variables:
    - cppData, javaData, pythonData: BenchmarkData for C++, Java, and Python languages.
- Constructor:
    - Initializes Graph with BenchmarkData for C++, Java, and Python languages.
- generateGraphs Method:
    - Generates comparison graphs for various benchmark metrics based on data from three different programming languages.
- createChart Method:
    - Creates a JFreeChart line chart for the specified benchmark metric.
    - Returns a JFreeChart line chart representing the comparison of the benchmark metric for the three languages.

6. Testing and validation

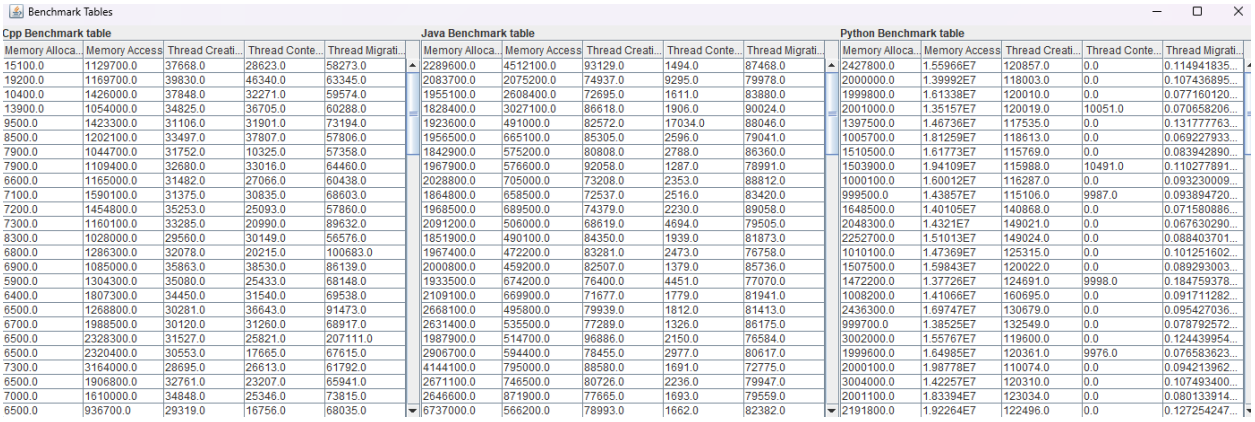
To ensure accurate measurements and reliable performance comparison, the application must undergo a series of tests. The testing process involves systematic data collection, visualization through JTables, and comparative analysis using graphical representations.

Data Collection:

- The application generates text files (PythonBM.txt, JavaBM.txt, CppBM.txt) containing benchmark metrics for memory operations and multi-threading functionalities. Each test case collects data for:
    - Memory Allocation and Access: Time taken for allocation and access of memory.
    - Thread Operations: Thread creation, context switch (in progress), and migration times.

Data visualization:

- JTables
    - Tabular Representation: The application utilizes JTables to present collected data in a structured tabular format for each language's benchmark.
    - Comparative Analysis: JTables allow for a side-by-side comparison of



Benchmark Tables

Cpp Benchmark table

Memory Alloca...	Memory Access	Thread Creati...	Thread Conte...	Thread Migrati...
15100.0	1129700.0	37668.0	28623.0	58273.0
19200.0	1169700.0	39830.0	46340.0	63345.0
10400.0	1426000.0	37848.0	32271.0	59574.0
13900.0	1054000.0	34825.0	36705.0	60288.0
9500.0	1423300.0	31106.0	31901.0	73194.0
8500.0	1202100.0	33497.0	37807.0	57806.0
7900.0	1044700.0	31752.0	10325.0	57358.0
7900.0	1109400.0	32680.0	33016.0	64460.0
6600.0	1165000.0	31482.0	27066.0	60438.0
7100.0	1590100.0	31375.0	30835.0	68603.0
7200.0	1454800.0	35253.0	25093.0	57860.0
7300.0	1160100.0	33285.0	20990.0	89632.0
8300.0	1028000.0	29560.0	30149.0	56576.0
6800.0	1286300.0	32078.0	20215.0	100683.0
6900.0	1085000.0	35863.0	38530.0	86139.0
5900.0	1304300.0	35080.0	25433.0	68148.0
6400.0	1807300.0	34450.0	31540.0	69538.0
6500.0	1268800.0	30281.0	36643.0	91473.0
6700.0	1988500.0	30120.0	31260.0	68917.0
6500.0	2328300.0	31527.0	25821.0	207111.0
6500.0	2320400.0	30553.0	17665.0	67615.0
7300.0	3164000.0	28695.0	26613.0	61792.0
6500.0	1906800.0	32761.0	23207.0	65941.0
7000.0	1610000.0	34848.0	25346.0	73815.0
6500.0	936700.0	29319.0	16756.0	68035.0

Java Benchmark table

Memory Alloca...	Memory Access	Thread Creati...	Thread Conte...	Thread Migrati...
2289600.0	4512100.0	93129.0	1494.0	87468.0
2083700.0	2075200.0	74937.0	9295.0	79978.0
1955100.0	2608400.0	72695.0	1611.0	83880.0
1828400.0	3027100.0	86618.0	1906.0	90024.0
1923600.0	491000.0	82572.0	17034.0	88046.0
1956500.0	665100.0	85305.0	2596.0	79041.0
1842900.0	575200.0	80808.0	2788.0	86360.0
1967900.0	576600.0	92058.0	1287.0	78991.0
2028800.0	705000.0	73208.0	2353.0	88812.0
1864800.0	658500.0	72537.0	2516.0	83420.0
1968500.0	689500.0	74379.0	2230.0	89058.0
2091200.0	506000.0	68619.0	4694.0	79505.0
1851900.0	490100.0	84350.0	1939.0	81873.0
1967400.0	472200.0	83281.0	2473.0	76758.0
2000800.0	459200.0	82507.0	1379.0	85736.0
1933500.0	674200.0	76400.0	4451.0	77070.0
2109100.0	669900.0	71677.0	1779.0	81941.0
2668100.0	495800.0	79939.0	1812.0	81413.0
2631400.0	535500.0	77289.0	1326.0	86175.0
1987900.0	514700.0	96886.0	2150.0	76584.0
2906700.0	594400.0	78455.0	2977.0	80617.0
4144100.0	795000.0	88580.0	1691.0	72775.0
2671100.0	746500.0	80726.0	2236.0	79947.0
2646600.0	871900.0	77665.0	1693.0	79559.0
6737000.0	566200.0	78993.0	1662.0	82382.0

Python Benchmark table

Memory Alloca...	Memory Access	Thread Creati...	Thread Conte...	Thread Migrati...
2427800.0	1.55966E7	120857.0	0.0	0.114941835...
2000000.0	1.39992E7	118003.0	0.0	0.107436895...
1999800.0	1.61338E7	120010.0	0.0	0.077160120...
2001000.0	1.35157E7	120019.0	10051.0	0.070658206...
1397500.0	1.46736E7	117535.0	0.0	0.131777763...
1005700.0	1.81259E7	118613.0	0.0	0.069227933...
1510500.0	1.61773E7	115769.0	0.0	0.083942890...
1503900.0	1.94109E7	115988.0	10491.0	0.110277891...
1000100.0	1.60012E7	116287.0	0.0	0.093230009...
999500.0	1.43857E7	115106.0	9987.0	0.093894720...
1648500.0	1.40105E7	140868.0	0.0	0.071580886...
2048300.0	1.4321E7	149021.0	0.0	0.067630290...
2252700.0	1.51013E7	149024.0	0.0	0.088403701...
1010100.0	1.47369E7	125315.0	0.0	0.101251602...
1507500.0	1.59843E7	120022.0	0.0	0.089293003...
1472200.0	1.37726E7	124691.0	9998.0	0.184759378...
1008200.0	1.41066E7	160695.0	0.0	0.091711282...
2436300.0	1.69747E7	130679.0	0.0	0.095427036...
999700.0	1.38525E7	132549.0	0.0	0.078792572...
3002000.0	1.55767E7	119600.0	0.0	0.124439954...
1999600.0	1.64985E7	120361.0	9976.0	0.076583623...
2000100.0	1.98778E7	110074.0	0.0	0.094213962...
3004000.0	1.42257E7	120310.0	0.0	0.107493400...
2001100.0	1.83394E7	123034.0	0.0	0.080133914...
2191800.0	1.92264E7	122496.0	0.0	0.127254247...

Figure 4. The JTables containing the benchmark program outputs

- Graphs
    - Graph Representation: The application utilizes the JFreeChart library to create and display in a Jpanel 5 graphs, each containing the data from the different languages for the corresponding benchmarked operations.

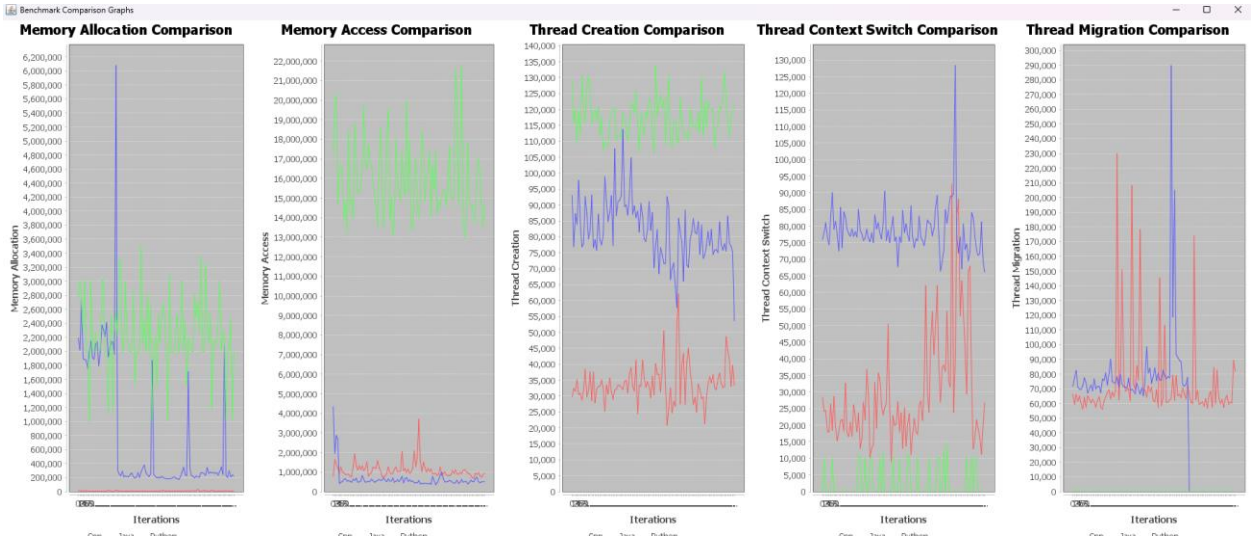


Figure 5. The Jpanel containing the graphs generated after running the benchmark

The testing and validation procedures confirm the reliability and accuracy of the Micro-Benchmarking application. Data collection through text files, visual representation via JTables, and graphical comparisons provide a comprehensive understanding of memory handling and multi-threading performance across

Python, Java, and C++ benchmarks. The comparison ensures an informed evaluation of system-level operations in different languages, aiding in identifying efficiency gaps and making informed decisions for optimized code development.

7. Conclusions

The benchmark project successfully addresses the performance evaluation of memory allocation, memory access, thread creation, context switching, and migration across multiple programming languages. The project provides a versatile tool for users to assess and compare the efficiency of these fundamental operations. By comparing the results on the graph, out of the 3 chosen programming languages, C++, java and python, the fastest in terms of speed is C++, because it allows the programmer to use lower level methods when dealing with basic operations.

8. Bibliography

[1] https://www.javatpoint.com

[2] https://www.educative.io

[3] https://docs.python.org

[4] https://docs.oracle.com