TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

Structure of Computer Systems

Measuring execution time of processes in different programming languages

Student: Maria-Catalina Burlea
Group: 30433
Coordinating Teacher: Prof. Lia-Anca Hangan
Academic Year: 2023-2024

Table of Contents

1. Introduction

1.1 Project Proposal

1.1.1 Context

It goes without saying that in the software development world, optimizing code performance and ensuring efficient program execution are paramount. A key aspect of this optimization process is measuring execution time of processes across various programming languages.

The goal of this project is to design and implement an application that can precisely measure execution times in different programming languages including C++, Java, and Python. The programs will provide a comparative analysis of various performance metrics such as*: memory-related tasks* (memory allocation, memory access, not only static, but also dynamic), *thread-related tasks* (thread creation, thread context switching and thread migration), *lists tasks* (insert into list, remove from list, sort the list).

The tasks outlined above are carried out in multiple programming languages to ensure the application's usability for users interested in the duration of various process-related events in different programming languages. This information can then be applied in optimizing other complex or related projects.

1.1.2 Specifications

The application will be built using Java Swing (a lightweight GUI toolkit), providing users the flexibility to select which process-related metric to execute. These metrics include static/dynamic memory allocation or access, thread creation, thread context switching, thread migration, insert into list, remove from list, sort the list or running all of them simultaneously. The results will be presented in a user-friendly web interface, where data can be viewed side by side or in a tabular format when all metrics are executed.

To obtain these results, when a user chooses an option, three distinct microbenchmarks dedicated to different programming languages (C++, Java, Python) will be initiated. Each benchmark will measure the execution time for the selected metric or for all metrics if that option is chosen. To achieve reliable results, these microbenchmarks will be executed multiple times, and the outcomes will be recorded in a file. The interface will then present the average of these results and a plot with the obtained values for the specific task.

1.1.3 Project objectives

1. **Develop multilingual application**

- *Objective:*
  - Create a flexible and extensible desktop application that can be used to measure the execution time of processes written in those three languages.
- *Reasoning:*
  - This framework will provide a unified platform for executing tests and collecting performance data in a consistent manner.
- *Key components:*
  - The application should have *language-specific modules* for the target programming languages to make the most of the language-specific features and libraries.
  - These modules will be designed to interact with code written in their respective languages effectively.

2. **Create user-friendly interface**

- *Objective:*
  - Design a user-friendly interface where the user can choose with ease between different performance metrics without the need of any instructions or any details.
- *Reasoning:*
  - A well-designed and intuitive interface will enhance the accessibility and will help for a better understanding, interpreting, and analysing the performance data.
- *Key features:*
  - Display side-by-side the results
  - Implement a table format to present the test results with clear columns for different performance metrics when the option of executing all those process-related tasks.

3. **Flexibility and Extensibility**

- *Objective:*
  - Ensure the application can easily adapt to new programming languages and performance metrics(new options for measuring execution times of other process-related tasks)
- *Reasoning:*
  - The application is a dynamic one and the ability to include new languages and metrics is crucial for further development.
- *Key feature:*
  - Modular architecture which allows addition of new language-specific modules and performance metrics. Therefore, those microbenchmarks should be easily written making the most of that programming language and integrated in the application.

4. **Accuracy**

- It's important to consider and mitigate potential sources of measurement discrepancies, such as program loading in main memory, cache variations, frequency changes, context switches, and multi-core processor-related factors.

5. **Reliability**

- The application should be designed to be error-prone and prevent any unforeseen behaviour.

6. **Consistency**
   - Maintain uniform time measurement units across all selected programming languages.
   - Additionally, aim to employ similar methods for measuring task durations while accommodating language-specific differences, especially in memory and threading handling.

7. **Precision**
   - Strive for the smallest possible time unit of measurement.
   - Ensure that benchmark mechanisms have its focus on measuring the core instructions, not on the side instructions.

8. **Efficiency**
   - Avoid the inclusion of excessive supplementary instructions in the programs, as they can negatively impact the programs' performance.

1.2 Project Plan

Project planning is one of the critical and essential operations that must be undertaken before starting any project. Regardless of your business, project planning should be done with the utmost consideration. Project planning is the primary way to streamline the project management process, and if this process is compromised, the business could run into several unwanted problems.

- **Week 1-2** (2 October – 15 October):

  Tasks:

  ➔ Lab tasks:
    - Choosing the theme of the project
    - Receiving the necessary information regarding the chosen project

  ➔ Documentation task:
    - Working on Introduction part of the documentation including:

      - *Project Proposal,*
      - *Project Plan* (detailed session plan scheduling the task & progress)
      - *Reference List*

      Bibliographic study including:

      - theory
      - how to measure execution time.

- **Week 3-4** (16 October – 29 October)**:**

  Tasks:

  ➔ Lab tasks:
    - Presenting the assignment regarding the introductory part
    - Taking into account the feedback and necessary improvements

  ➔ Documentation:
    - Working on Analysis and Design parts

  ➔ Development tasks:
    - Make design decisions.
    - Start designing and implementing the modules.

  - Adding more information into the *Bibliographic study* as new concepts come to light during the design and implementation processes.

- **Week 5-6** (30 October – 12 November)**:**
  Tasks:
  - ➔ Lab tasks:
    - Progress discussion
    - Taking into account the feedback and necessary improvements
    - Fixing potential bugs
  - ➔ Development task:
    - Finishing working on design & language-specific modules regarding memory-related tasks.
    - Fixing bugs
    - Start working on the graphical interface part + connect the modules if the design & language-specific modules are done.

- **Week 7-8** (13 November – 26 November):
  Tasks:
  - ➔ Lab tasks:
    - Progress discussion
    - Taking into account the feedback and necessary improvements
    - Fixing potential bugs
  - ➔ Development task:
    - Fixing bugs
    - Work on thread related tasks.

- **Week 9-10** (27 November – 10 December)
  Tasks:
  - ➔ Lab tasks:
  - Progress discussion
  - Taking into account the feedback and necessary improvements
  - Fixing potential bugs
  - ➔ Documentation task:
    - Working on Implementation, Test/Experiments parts
  - ➔ Development task:
    - Finish work if it is not done.

- **Week 11-12** (11 December – 31 December)
  Tasks:
  - ➔ Allocate week 11 for unforeseen circumstances such as:
    - addressing potential issues that may arise,
    - refining documentation,
    - accommodating any delays that might have occurred in previous weeks.
  - ➔ Lab tasks:
    - Present the project and the documentation.

As the project unfolds, there might arise situations that demand modifications, leading to the need for making revisions in the elements discussed earlier.

2. Bibliographic study

2.1 Microbenchmark

Benchmarking is an essential tool in development across languages, but choosing the correct scale can prove difficult.

Microbenchmarks deal with the smallest of all benchmarks, and they represent a *very simple and easy-to-define metric that tracks* and measures the performance of a small and specific piece of code. In contrast of benchmarking, which is the process of running a computer program to evaluate how well the runtime environment performs.

Microbenchmarks are used to measure *simple and well-defined quantities* such as elapsed time, rate of operations, bandwidth, or latency. Typically, microbenchmarks were associated with the testing of individual software subroutines or lower-level hardware components such as the CPU and for a short period of time.

A microbenchmark always pertains to a very small amount of code. Hence, they're incredibly fast to implement.

2.2 Execution time

Measuring the performance of a program means keeping track of the consumption of resources used by the program. It involves collecting data and metrics to assess its efficiency, effectiveness, and overall quality. Performance measurement is crucial for identifying bottlenecks, optimizing resource usage, and ensuring that the software meets the desired criteria.

Execution time, also known as "**run time**" or **"elapsed time",** is the amount of time it takes for a program or a specific operation within a program to complete its execution in the real world. It represents the time that elapses from the start of the program or operation to its completion, including all the time spent waiting for various resources and performing computations. Choosing the right method to measure the execution time will depend on your operating system, programming language, and even what is meant by “time”.

2.3 Wall Time vs CPU Time

It is important to both define and differentiate between the two terms, Wall Time, and CPU Time, that are often used when measuring runtime.

➔ **Wall Time**

- Most common way to measure execution time.
- Known as "digital clock" time, is simply the *total elapsed time during a measurement.*
- It is the time you can measure with a **stopwatch,** assuming you are able to *start and stop* it at *exactly the execution points you want.*
- It includes all the time the program or operation spends executing instructions, as well as any time spent waiting for external resources, such as I/O (input/output) operations, user interactions, network communication, and other factors.
- It is what users typically perceive as the time it takes for a program to run.

➔ **CPU Time**

- Refers to the time *a CPU was busy processing program instructions*. Time spent waiting for other things (e.g., I/O operations) is not included in CPU time.
- It measures the actual amount of time the central processing unit (CPU) spends executing the program's instructions. It excludes time spent waiting for external resources and only accounts for time the CPU is actively processing the program's code. CPU time can be further divided into:
  - **"User CPU time"** = time spent executing user-level code
  - **"System CPU time"** = time spent in the kernel or operating system while servicing the program's requests.
- **System Time** is preferable to CPU Time, especially in the *context of parallel software*. If some threads are already finished and no longer consume CPU Time, System Time provides a more natural measure of the time computation takes.

- To get good results with System Time, you will need a computer that is *primarily running only the software you want to compare*. The computer might still be running an operating system and some background tasks, but *no GUI or any other application software*. This will give you the best possible testing environment, so using System Time is the right choice.

Put simply, **Wall Time** measures *how much time has passed* (as if you were looking at the clock on your wall) and **CPU Time** is *how many seconds a CPU was busy*.

2.4 Different Techniques

➔ **The Stopwatch Approach:**

- The simplest and most intuitive way to measure time is to use a stopwatch manually from the moment you start a program.
- *Not optimal*, since on average a human takes from 0.25 seconds up to 1 full second to react and then spends another fraction of a second on pressing a button physically or digitally.
- Useful for measuring **elapsed time (wall time)**

➔ **Utility time:**

- A faster and more automated alternative to the manual stopwatch is the **time** utility, which is very easy to use and requires no special settings. It is sufficient to use the following syntax **time ./program-name**
- Together with the result of the program, the time utility prints three different types of time:
  - Real Time: measurement of real time (in terms of sec) using the system's internal clock.
  - User Time: measurement of the time when the instructions of the "User" session are executed.
  - System Time: measurement of the time in which the instructions of the "Supervisor" session are executed - generally a much smaller time than the User Time when performing certain tasks.
- It is **not necessarily** the case **that User_time + System_time = Real_time**. User Time and System Time are calculated in CPU-seconds (i.e. seconds calculated from the CPU clock cycle).

➔ **Counting the clock:**

- Measure in two different time instants (start and end)
```
clock_t start = clock();
// let's do some operations
clock_t end = clock();
long double seconds = (float)(end - start) / CLOCKS_PER_SEC;
```
- The **CPU time** is represented by the data type clock_t – no of ticks of the clock signal.
- CLOCKS_PER_SEC already defined in time.h library

2.5 Execution time technique in project – High Resolution Timer

To measure the execution time of a piece of code, we would record two timestamps: one at the start of the code and the other at its completion. Subtracting these timestamps would yield the time elapsed between the two, giving us the execution time.

1. C++

In C++ we have **<chrono>** , C++ header that provides a collection of types and functions to work with time which is included from C++11. <chrono> provides three main types of clocks: system_clock, steady_clock, and *high_resolution_clock.* (It provides the smallest possible tick period. It is written as-**std::chrono::high_resolution_clock**) These clocks are used to measure time in various ways. The chrono library can measure time in **nanoseconds**, microseconds, milliseconds, seconds, minutes, hours, days, weeks, months, year.

Example of basic usage:

```
using namespace std::chrono;

auto startTime = high_resolution_clock::now();
// ... the code being measured ...
auto endTime = high_resolution_clock::now();

auto duration = duration_cast<nanoseconds>(endTime -
startTime);
```

2. Java

In Java, we have **System.nanoTime() (public static long nanoTime())** which returns the current value of the running Java Virtual Machine's high-resolution time source, in ***nanoseconds***. This method can only be used to measure *elapsed time* and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary origin time (perhaps in the future, so values may be negative).

Example of basic usage:

```
long startTime = System.nanoTime();
// ... the code being measured ...
long endTime = System.nanoTime();

long estimatedTime = endTime - startTime;
```

3. **Python**
Python 3.7 introduces new functions to the time module that provides higher resolution: **time.time_ns()** → int which is similar to (time.time() which returns seconds) and returns time as an integer number of nanoseconds.

Example of basic usage:
```
import time

start_time = time.time_ns()

# ... the code being measured ...
end_time = time.time_ns()

elapsed_time_ns = end_time - start_time
```

To sum up, to ensure *consistency* in measuring execution time across all three programming languages, we will use ***nanoseconds*** as the common unit of measurement.

2.6 Memory Allocation & Access

Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is done either before or at the time of program execution.

There are two types of memory allocations:

➔ Static Memory Allocation:

- *Similarities:*
  - Involves reserving memory for variables at *compile time.*
  - The Memory is allocated once, and the size is typically fixed.
  - In C: The address can be found using the address of operator and can be assigned to a pointer.
  - Uses stack.
  - Less efficient
  - No memory re-usability

- Difference:
  - In Python we cannot have static memory allocation (only dynamic) since Python is a high-level programming language implemented in C language. It handles everything in the form of objects. All these data structures and Python objects are stored in a private heap managed by the Python memory manager.

- Use:
  When declaring and initializing primitives, local variables in C++ and Java.
  In Java we also store calling methods, references to objects, returned values (primitive values, references to objects).

- Example:

1. **C++** `int x = 5;`
2. **Java** `int x = 5;`
3. **Python** `not possible`

- o **Static memory access** (accessing memory locations -known at compile time - associated to statically allocated variables) - we store in a declared variable the value of the statically allocated memory.

➔ Dynamic Memory Allocation:

- o *Similarities*:
  - Involves reserving memory during *runtime.*
  - Uses heap.
  - More efficient
  - Memory reusability and memory can be freed when not required.

- o *Differences:*
  - In C/C++, a programmer is responsible for both the **creation and destruction of objects.** Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing **OutOfMemoryErrors.**
  - But in Java and Python, the programmer does not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of **Garbage Collector** is to free heap memory by destroying unreachable objects. The garbage collector is the best example of the Daemon thread as it is always running in the background.
  - In Java you can create Objects and arrays dynamically
    ```
    MyClass myObject = new MyClass();
    int[] numbers = new int[5];
    ```
  - Python uses the **dynamic memory allocation** which is managed by the Heap data structure. Memory Heap holds the objects and other data structures that will be used in the program. Python memory manager manages the allocation or de-allocation of the heap memory space through the API functions.

- o Use: When declaring and initializing structures with flexible size in all three programming languages.

- o **Dynamic memory access** (accessing memory locations-determined at runtime often through pointers or references - associated to dynamically allocated variables)

- o Example:

1. **C++:**
```
int *x = (int *)malloc(5 * sizeof(int));

if (x == NULL) {
    return 1;
}

for (int i = 0; i < 5; i++) {
    x[i] = 0;
}
```

```
// deallocate
 free(x);
```

2. **Java:**
```
int[] x = new int[5];
```

3. **Python:**
```
x = [0] * 5
```

2.7 Threads & Processes

➔ Process Definition – address space + resources + threads (>= 1)

- Longman dictionary's definition of process
  – a series of actions that are done to achieve a particular result
- a program in execution ⇔ an user application
  – a sequential stream of execution in its own memory address space – including the current values of CPU's registers (e.g. IP)
- OS abstraction for using the computer.

- Thread Definition – unit of execution within a process
  - describes a sequential, independent execution within a process.
    – execution = the sequence of executed instructions
    – execution = the memory path followed in the code by the IP register
  - there could be more:
    – simultaneous and independent executions in the same process – ⇒ threads in a process

- Thread Creation
  - Involves spawning a new execution thread within a process.
  - Threads share the same memory space but can execute concurrently.
  - *Similarities:*
    - Done by initializing an object from a *thread-related class*

  - *Differences:*
    - C++ threads require explicit management with std::thread. You need to manually *join or detach threads* to ensure they don't terminate prematurely.
    - C++ offers low-level control over thread management, allowing you to fine-tune thread behavior, but this also introduces complexity.
    - Java threads are managed by the Java Virtual Machine (JVM), abstracting low-level details. (high-level, automatic approach with garbage collection. Similar to Java's approach, Python's threading module provides a high-level, multithreading API for creating and managing threads.

1. ***C++***

- Prior to C++11, we had to use POSIX threads or <pthreads> library.
- Multithreading support was introduced in C++11 by adding **std::thread**.
- The thread classes and related functions are defined in the <thread> header file.

```
std::thread thread_object (callable);
```

2. ***Java***

- Creation of threads can be done in 2 ways:
- Inheriting your class from **Thread Class**

```
public class MyThreadClass extends Thread
   @Override
   public void run() {
      int x = 5;
   }
}
…
MyThreadClass myThread = new MyThreadClass();
myThread.start();
```

- Implementing the **Runnable Interface**(functional interface)

```
public class RunnableClass implements Runnable{
    public void run(){
        int x = 5;
    }
}

Thread myThread = new Thread(new RunnableClass());
myThread.start();
```

3. ***Python***

- Threads are managed by the **Global Interpreter Lock (GIL)**
- Use ***threading*** module through a class:

```
import threading

# Define a class that subclasses threading.Thread
class MyThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        x = 5

# Create an instance of the custom thread class
new_thread = MyThread()

# Start the new thread
new_thread.start()

# Wait for the new thread to finish (optional)
new_thread.join()
```

- Thread Context Switch
    - A context switch occurs when a running thread is paused, and the CPU switches to another thread to execute. Context switching is necessary for multitasking and multithreading to ensure fair execution of threads.
    - It involves saving the state of the current thread and loading the state of the next thread.

1. **C++**
    - The operating system's scheduler determines when and how threads are switched.
    - Thread priorities and time slices are managed by the OS.

2. **In Java**
    - The **JVM** delegates thread management to the underlying operating system's scheduler, which handles context switching.
    - Thread priorities can influence the order of execution and context switching.
    - Blocking operations (e.g., I/O) can result in voluntary context switches, allowing other threads to run.

3. **Python**
    - Python threads, created with the threading module, run concurrently, but the GIL restricts *true parallel execution* in CPython (the standard Python implementation).
    - The Python Global Interpreter Lock(GIL), in simple words, is a *mutex (or a lock)* that **allows only one thread to hold the control of the Python interpreter**. This means that **only one thread can be in a state of execution at any point in time.**
    - Threads may yield the CPU due to I/O-bound operations or using threading.Thread.yield(), but this doesn't guarantee immediate context switches.
    - Context switches in Python primarily occur when *threads release the GIL.*
    - For CPU-bound tasks, the GIL can limit the benefits of multithreading.

In Java, threads are executed on **different CPU cores**, enabling true parallel execution. This parallelism means that context switches between threads can occur seamlessly. The initiation and management of context switches are primarily controlled by the operating system and are not directly detectable by a program. As a result, accurately measuring the execution time of a context switch can be challenging. To measure context switch time, we would typically need to simulate a context switch by using synchronization mechanisms, such as locks, to coordinate the execution of two threads. The goal is to simulate a scenario where multiple threads access and modify shared data, and the mutex ensures proper synchronization to avoid data corruption. This simulation involves the creation of multiple threads that concurrently increment a shared global variable (sharedNumber) within a critical section protected by a mutex. These synchronization mechanisms (locks for instance) are available in all three programming languages (C++, Java, and Python) and allow for controlled experimentation in measuring context switch times.

- Thread Migration
    - Thread migration is the process of moving or migrating a thread from one core to another one.
    - In C++ and Java, *threads are distributed across multiple CPU* cores (in the case of a multi-core computer), facilitating the simulation of thread migration by temporarily stopping one thread's execution and starting another's.
    - However, in Python, *threads within a program generally run on the same core*, as Python's Global Interpreter Lock (GIL) restricts true parallel execution. In Python, true thread migration is not feasible, but you can simulate it differently by utilizing multiprocessing from the ‘processing package. In this case, separate processes are created, each with its own Python interpreter and memory space, which allows for parallel execution on different CPU cores and more accurate simulation of thread migration.

2.8 Operations with Lists

➔ Lists

In computer science, a list or sequence is an *abstract data type* that represents a finite number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence; the (potentially) infinite analog of a list is a stream.

Lists are used to organize and manage data in a structured way. They provide flexibility in terms of adding, removing, and accessing elements. Lists can be implemented in different ways, such as *arrays, linked lists, or dynamic arrays.*

- Insertion into List
    - Insertion into the list refers to the process of adding an element or data item into the list. The element is typically added at a specified position or location within the data structure.
    - The goal of insertion is to update the structure to include the new element while maintaining the integrity and order of the existing elements.

1. **C++**

- The `list::insert()` is used to insert the elements at any position of list.
- This function takes 3 elements, position, number of elements to insert and value to insert. If not mentioned, number of elements is default set to 1.
- ***Time Complexity:*** *Linear O(n)* this is because, similar to Java's `ArrayList.add()` it may require shifting elements to accommodate the new element at the specified position.
- ***Return Value:*** This function returns an iterator that points to the first of the newly inserted elements.

-
  ```
  list<int>::iterator it = randomList.begin();
  // Move iterator to position
  advance(it, positionToInsert);
  randomList.insert(it, value);
  ```

2. Java

- The `add(int index, E element)` method of List interface in Java is used to insert the specified element at the given index in the current list.
- This method takes 2 elements: index where will be inserted and the element to insert into list.
- ***Time Complexity***: O(n)
    1. In a singly linked list, the time complexity for inserting and deleting an element from the list is O(n) since it requires shifting in order to accommodate the new element at specified position.
    2. In a doubly-linked list, the time complexity for inserting and deleting an element is O(1).
- ***Return value:*** Boolean and it returns true if the object is added successfully.
- `list.add(positionToInsert, value);`

3. Python

- The function `list_name.insert(index, element)` makes it easier to insert an element at a particular place into a list
- ***Time Complexity***: *O(n)* because the insert method has to shift all the elements of the list to accommodate the newly inserted element, which takes O(n) time.
- ***Return value:*** It returns None. It only updates the current list.
- `random_list.insert(position_to_insert, value)`

- Deletion from List
    - Deletion from a list refers to the process of removing an element or data item from the list.

1. C++

- `list::erase()` removes from a list single or multiple contiguous elements in range can be removed using this function.
- This function takes 2 arguments, start iterator and end iterator.
- ***Time complexity***: O(n) where (n is size of list).
- ***Return value***: Returns an iterator pointing to the next valid position after the last element erased.

-
  ```
  list<int>::iterator it = randomList.begin();
  advance(it, position);
  randomList.erase(it);
  ```

2. Java

- ArrayList class provides two overloaded remove() methods.

- `remove(int index)`: Accepts the index of the object to be removed
- `remove(Object obj)`: Accepts the object to be removed

- It is not recommended to use `ArrayList.remove()` while iterating over the list.
- ***Time complexity:*** O(n) because ArrayList implements RandomAccess interface, so accessing any random element will be done in O(1) complexity. However, since the remaining elements also need to be shifted by one place to their left, the overall time complexity becomes O(n).
- ***Return value:*** Returns the element that was removed from the list.
- `list.remove(position);`

3. **Python**

- The Python `del` statement is **not** a function of List.
- Items of the list can be deleted using the del statement by specifying the index of the item (element) to be deleted.
- ***Time complexity:*** O(n) because the function needs to search the list sequentially to find the first occurrence of the specified item and then remove it.
- ***Return value:*** No return value, and it doesn't return anything explicitly. It is a statement used for deleting objects or elements in place, and it operates directly on the object, modifying it or removing elements as specified.
- `del random_list[position]`

- Sorting the List
  - Sorting is the process of arranging elements in a specific order, often in ascending or descending order. The order can be based on numerical values, alphabetical order, or any other criterion.

1. C++

- `sort()` function is used to sort the elements of the container by changing their positions.
- ***Time complexity:*** O(n log n) on average, where n is the number of elements in the container.
- ***Return value:*** Returns void as it sorts the container in place.
- ***Algorithm:*** intro-sort (introspective sort), which is basically a Quicksort that keeps track of its recursion depth, and will switch to a Heapsort (usually slower but guaranteed O(n log n) complexity) if the Quicksort is using too deep of recursion.
- `randomList.sort();`

2. Java

- `java.util.Collections.sort()` is used to sort the elements present in the specified list of Collection in ascending order.
- It works similar to `java.util.Arrays.sort()` method but it is better than as it can sort the elements of Array as well as linked list, queue and many more present in it.
- ***Time complexity:*** O(n log n) on average, where n is the number of elements in the container.
- ***Return value:*** Returns void as it sorts the container in place.
- ***Algorithm:*** Collections.sort uses MergeSort since it is stable
- `Collections.sort(list);`

3. Python

- `sort()` is an inbuilt function in Python that is used to sort the values of a list in ascending, descending, or user-defined order.
- The difference between `sort()` and `sorted()` is that the sort list in Python alters the list directly and produces no output, whereas sorted() doesn't change the list and returns the sorted list.
- ***Time complexity:*** O(n log n) on average, where n is the number of elements in the container.
- ***Return value:*** Returns None as it sorts the list in place.

- ***Algorithm:*** Tim Sort, which is a combination of both merge sort and insertion sort.
- `random_list.sort()`

3. Analysis

The analysis stage is based on the description of the list of functionalities mentioned in the project specification and aims to identify, not only the algorithms that need to be implemented, but also the user interaction with the application.

3.1 Functionality description

A functionality description is a detailed explanation of the features, capabilities, and behaviors of a software application, system, or component. It serves as a comprehensive guide to help users and developers understand how a particular part of a software system works.

Every functionality described below will be executed a specific number of times(100). The outcomes from these executions will be saved in a file, and subsequently, we will calculate an average value for the execution time of the particular metric. In the case we want all these metrics, the values will be displayed in table.

Furthermore, the results obtained from these executions will be visualized in a plot where each plot corresponds to a specific programming language. The x-axis will represent the i^{th} execution, and the y-axis will indicate the execution time.

We expect that after multiple execution of the same program, *the execution time will decrease* since modern computer systems often use various levels of caching to speed up program execution. (L3 (Level 3) cache, if available, is shared among all CPU cores and is larger but slower than L2 and L1 caches. It acts as a shared cache for frequently used data and instructions across the entire processor). When a program is run multiple times, data and code may be cached in memory, reducing the time required to fetch them from slower storage devices. If you run the same program multiple times in a short period, there's a higher likelihood that some of its instructions and data are already present in the cache from previous runs. This can lead to faster execution times because the CPU can access these cached instructions and data more quickly than fetching them from slower main memory.

1. Static Allocation

We need to take into account that static allocation is *not possible* in Python. Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. Management of the Python heap is performed by the interpreter itself and that the user has **no control over it**, even if they regularly manipulate object pointers to memory blocks inside that heap.

Therefore, we will only measure the execution time of static allocation in Java and C++.

In order to measure the execution time of static allocation, we will create a function in C++, respective a method in Java. The function/method is designed to measure the time it takes to perform a "static allocation" of a simple integer variable (int x = 5;) The difference in execution times is used to estimate how long the static allocation takes.

As we expect, C++ will perform better than Java since it gets compiled directly into machine code -- without an intermediary translation required at runtime. Java programs instead are compiled into bytecode (the compilation is done by Java compiler) and then the Java bytecode is interpreted by the JVM.

2. Dynamic Allocation

When it comes to measuring the execution time of dynamic allocation, we will simply allocate an array of 500 elements in every programming language.

Since C++ facilitates the programmer through the dynamic allocation of memory and enables the developer to declare and control the lifetime of objects, it doesn't use a garbage collector. This removes the additional time requirement in which a Java Virtual Machine or Python interpreter must stop for a while and trace the object's lifetime. Therefore, the C++ code will be faster, having the smallest execution time.

3. Static Memory Access

Static Memory access is only possible in C++ and Java since in Python we have only dynamic allocation. For this case, we will store in a local variable of the function, respective method an integer with a specific value (e.g.: 23) and we initialize value by 0. The function records the current time before and after the variable access and calculates the execution time using them. The resulting will represent the time it took to access a local variable, but it is an approximative one since it may include overhead since at the same time we both perform the access to a local variable and the instantiation of another variable.

We anticipate that C++ outperforms Java in low-level operations, like accessing local variables, due to the absence of Java Virtual Machine (JVM) overhead and the superior memory control and direct access capabilities offered by C++. C++ compilers are known for their ability to apply aggressive optimizations, leading to highly efficient machine code. These optimizations significantly speed up straightforward tasks like accessing local variables. The key distinction lies in memory management, where Java's garbage collection introduces unpredictable pauses during program execution. In contrast, C++ manages memory more explicitly, eliminating the overhead of garbage collection.

4. Dynamic Memory Access

To measure the dynamic memory access, we need to allocate dynamically a vector of a specific size (500) and populate it with values (each element has value equal to index). We measure the time required to access elements from this dynamically allocated integer array within a loop. The resulting value is divided by the array's size (500) because we are calculating the dynamic access time for each element in the array.

C++ will outperform also for this metrics since it has direct control and access on memory management and the fact that it manages memory explicitly, eliminating the overhead of garbage collection.

5. Thread Creation

To measure the execution time of a thread creation we simply create a thread (by initializing an object from a thread-related class) in each programming language, and each thread runs a specific function within it. For instance, these functions allocate a variable x = 5

We anticipate that C++ will exhibit superior performance in this context, primarily because C++ offers finer control over memory and system resources, which can lead to better efficiency. Java, while achieving a good balance between performance and portability, might incur some overhead due to the Java Virtual Machine (JVM) and the garbage collection mechanism. Python, as a high-level language, may experience slower execution speeds compared to C++, primarily due to the management of memory and resources through garbage collection.

6. Thread Context Switch

It is essential to take into account that in C++ and Java, threads can be distributed across different CPU cores (if available), enabling true parallel execution. However, in Python, the presence of the Python Global Interpreter Lock (GIL) restricts Python threads from executing in true parallel on multiple CPU cores. The GIL allows only one Python thread to hold control of the Python interpreter at any given time, limiting the extent of parallelism achievable in Python.

We will simulate a context switch by using synchronization mechanisms, such as locks, to coordinate the execution of multiple threads. The goal is to simulate a scenario where multiple threads access and modify shared data, and the mutex ensures proper synchronization to avoid data corruption. To perform this, we need to create multiple threads hat concurrently increment a shared global variable (sharedNumber) within a critical section protected by a mutex. It goes without saying that before this, we need to create them, and each thread will run a specific function within it.

In terms of performance expectations, C++ is anticipated to exhibit the highest performance due to its efficient memory management, multi-core support, and fine-grained control over resources. Java, while providing a good balance between performance and portability, might experience slightly more overhead due to the Java Virtual Machine (JVM) and garbage collection which is used also when it comes to threads. On the other hand, Python is expected to have the least favorable performance due to the GIL, which restricts true parallelism, and the influence of garbage collection, which can introduce occasional delays.

7. Thread Migration

On a multi-core computer, when it comes to C++ and Java, *threads are distributed across multiple* cores facilitating the simulation of thread migration by temporarily stopping one thread's execution and starting another's. However, in Python, *threads within a program generally run on the same core*, as Python's Global Interpreter Lock (GIL) restricts true parallel execution. In Python, true thread migration is not feasible, but you can simulate it differently by utilizing multiprocessing from the ‘processing’ package. In this case, separate processes are created, each with its memory space, which allows for parallel execution on different CPU cores and more accurate simulation of thread migration.

From the “main process” a child process will be created. A single thread will be created in both the main process and a child process. In this setup, the child process will record the current time (start time) and transmit it to the parent process. Subsequently, the main process will record the end time. The difference between the end and start times serves as a measure of the execution time for the designated microbenchmark.

When it comes to thread migration, C++ and Java are generally expected to have better performance and flexibility compared to Python, primarily due to the absence of the Global Interpreter Lock and their support for true parallel execution across multiple CPU cores. True thread migration is not feasible within Python, and the simulation of a thread migration will be consequently less accurate and efficient.

8. Insertion into List

To measure this microbenchmark, we will create a random list of 10000 elements, and we will insert at position 500 a random element in each programming language.

The std::list function from C++ dynamically allocates memory for each element as it's inserted, contributing to predictable memory behavior. Additionally, C++ benefits from aggressive compiler optimizations, leading to efficient machine code generation and deterministic performance. In contrast, Java utilizes automatic memory management with garbage collection, introducing some unpredictability due to potential pauses during garbage collection events. While Java's collections library is well-optimized, the LinkedList's dynamic memory allocation may experience variability. Python, with its dynamic memory management, automatically resizes lists as elements are added, simplifying development but introducing some overhead. Python's list implementation, though optimized for general use, may not match the performance of lower-level languages like C++.

9. Deletion from List

To measure this microbenchmark, we will create a random list of 10000 elements, and we will remove from position 500 in each programming language.

When it comes to deletion from a list, the std::list dynamically allocates and deallocates memory for each element, providing a predictable memory deallocation process. C++ benefits from aggressive compiler optimizations, enabling efficient removal of elements with deterministic performance. In contrast, Java relies on automatic memory management with garbage collection, introducing potential unpredictability due to pauses during garbage collection. The LinkedList's dynamic memory management may add variability during deletion operations. Python, utilizing dynamic memory management, automatically adjusts list size during deletion, offering simplicity but introducing some overhead. While Python's list implementation is optimized for general use, it might not match the performance of lower-level languages like C++.

10. Sort the List

To measure this microbenchmark, we will create a random list of 10000 elements, and we will sort the list in each programming language.

When it comes to sorting a list, the std::list leverages dynamic memory allocation, allowing efficient rearrangement of elements. C++ benefits from aggressive compiler optimizations, resulting in streamlined and deterministic sorting performance. Conversely, Java adopts automatic memory management with garbage collection, introducing potential variability due to garbage collection pauses during list sorting operations. The well-optimized collections library in Java reduces some performance concerns. In Python, dynamic memory management automatically adjusts list size during sorting, offering simplicity but introducing overhead. While Python's list implementation is optimized for general use, it might not match the performance of lower-level languages like C++.

3.2 User interaction with the Application

User interaction with an application involves the dynamic exchange between the user and the software, forming the core of the user experience. User interaction is central to the success of an application, as it directly influences user satisfaction, efficiency, and the overall usability of the software.

The user will interact directly with the homepage of the application where a set of eight distinct buttons, each representing a different task. These buttons are thoughtfully labeled to reflect their respective functions, providing users with a clear and intuitive navigation experience. When user selects a specific task a new window will appear, displaying information from microbenchmark results.

For instance, consider the user's decision to click on the "Allocate Statically" button. As a response, the application will open a new window offering comprehensive insights about the average performance metrics for each programming language. The window will also feature a plot showcasing the execution times recorded in nanoseconds for each individual run of the program. Furthermore, it will feature a back button that enables users to easily return to the homepage whenever they wish to find out more about another memory-related or thread-related task.

If the user chooses to activate the "Run All" button, the application will execute all of the designated commands. Subsequently, a new window will promptly emerge, presenting a neatly organized table with the calculated averages. This table presents a concise and comprehensive overview of the microbenchmark results for all tasks, further simplifying the process of evaluating performance across various programming languages.

The visualization of the execution times is invaluable, as it grants the user an in-depth understanding of each programming language's performance characteristics. With these insights at their disposal, users can make informed decisions regarding the selection of the most suitable programming language for their upcoming projects, aligning their choices with their specific requirements and optimizing their software development activities. In essence, the application empowers users to make data-driven decisions, ensuring that their projects are executed with the utmost efficiency and success.

3.3 Scenarios and Use Cases

The scenarios that may occur while running the program can be viewed in the following use case diagrams:

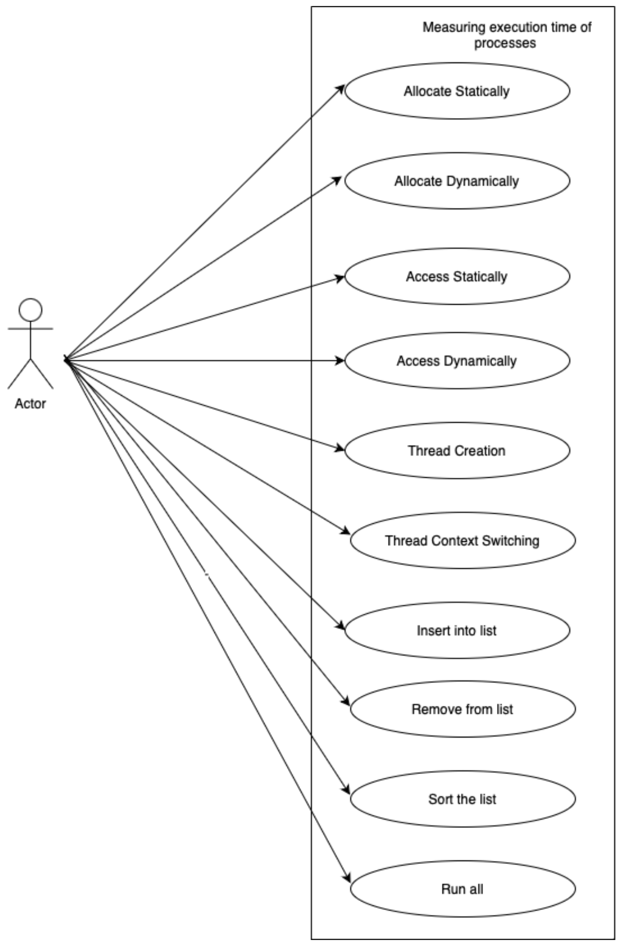


Figure 1: Use case diagram

Use case: Allocate Statically
Primary Actor: user
Main success Scenario:

1) The user clicks on 'Allocate Statically' button.
2) The application opens a new graphical interface with the average results and the plots for the task.

In the same manner, the use cases for Allocate Dynamically, Access Statically, Access Dynamically, Thread Creation, Thread Migration and Thread Context Switching, Insert into List, Remove from List, Sort the List have a similar scenario.

Use case: Run All
Primary Actor: user
Main success Scenario:

1)The user clicks on 'Run All' button.
2) The application opens a new graphical interface where the results for all microbenchmarks are displayed in a tabular manner.

4. Design

4.1 Modelling

Modelling is the process of creating a representation of a software system, component, infrastructure using visual or abstract techniques. It involves designing a blueprint or plan that captures the important features and behaviour of the system and serves as a guide for implementation.

In light of this modelling concept, the application is conceived as a system with eight distinct inputs and two primary outputs. These inputs are associated with specific actions or commands that users can trigger. The available commands are designed for a range of activities, from memory allocation (static and dynamic) to memory access, thread-related tasks, and a comprehensive "Run all" function.

Each button triggers the execution of the script. For the script to successfully run, there are sent the following parameters:

- The file names for C++, Java, and Python program files which will start running immediately to compute the execution time for the specific microbenchmark.
- An output file name where the results of the microbenchmarks are stored.
- A command corresponding to the button pressed, specifying the operation to be performed.

The script will produce outputs that capture the results from running each microbenchmark 100 times for each programming language. These output values will be further processed to create plots and compute the average performance metrics for the given task.

4.2 How to run the programs

To execute each program, we'll create a script (*create_tests.sh*). This script will receive the program names, the output file -where we'll store the average results-, and the specific command for the operation to be performed.

The script will first check if it has received the correct number of command-line arguments and assigns these arguments to variables for further use. The script then compiles and executes the C++ and Java programs, and runs the Python script, all with a specific command. The standard output of each program is appended to the output file.

The script will be launched within each inner class of the HomepageController when the corresponding operation button is pressed.

To execute the script, we'll use a process where we pass the command:

```
Process process = Runtime.getRuntime().exec(cmd);
```

We'll also wait for the process to complete using `process.waitFor()`. This ensures that the script or specified command is allowed to finish its execution before the program continues with the next steps.

4.3 Design Levels

4.3.1 Level 1: Overall system design

The first design level is represented by the overall picture of the application which is given by the following black box:

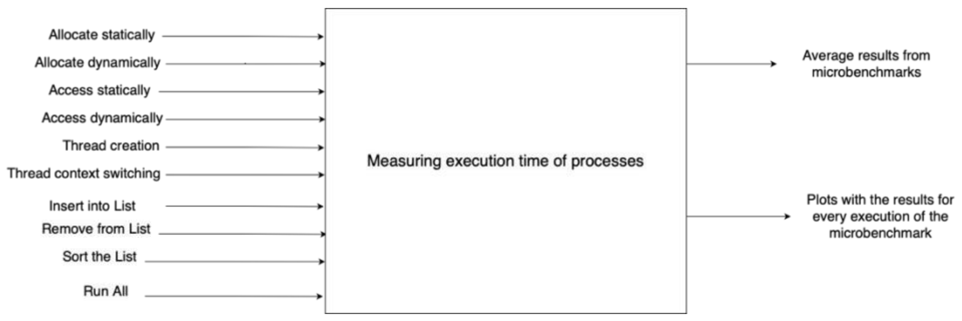


Figure 2: Blackbox of the application

4.3.2 Level 2: Division into subsystems/packages

The second design level is the one that divides into sub-systems/packages. The packages of the application are *start, presentation* and *tests*. The Main method is located inside the start package and serves for launching the program towards execution.

Presentation package contains 3 sub packages: *model* (contains Language enum which has CPP, JAVA and PYTHON), *view* (graphical user interfaces for each window HomepageView, RunAllView, ResultsView), *controllers* (controllers for each graphical user interface: HomepageController, RunAllController, ResultsController).

Package *start* contains the Main class, and it starts the execution of the application.

The *tests* package comprises three sub packages: *c++, javaTest*, and *python.* Within each of these sub packages, you can find code specifically designed for benchmarking in their respective programming languages, with the folder name matching the language's name, along with other associated files. In the case of Java, you will also find .class files, while in the case of C++, there will be executable files. Additionally, there is a script file that contains the commands for running all three programs, and there is a *results* sub package where the outcomes of these tasks are stored.

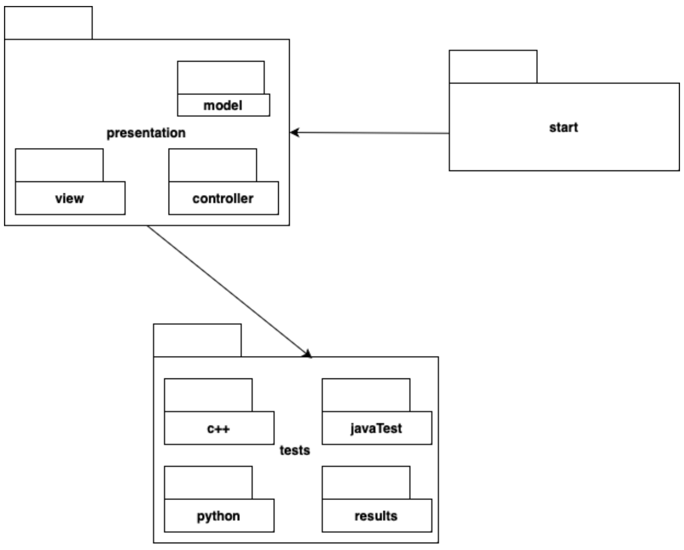


Figure 3: Package diagram

4.3.3 Level 3: Division into classes

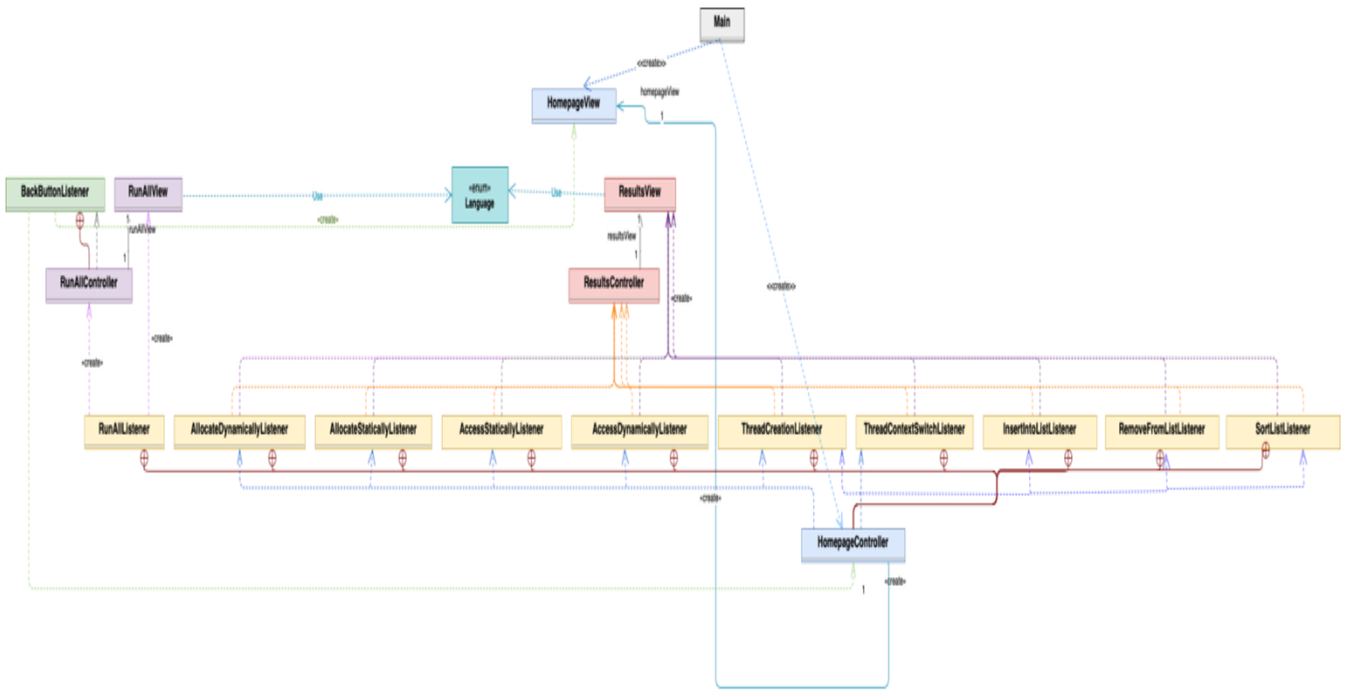


Figure 4: Class diagram

Within the start package, we find the Main class where the key components HomepageController and HomepageView are instantiated. The HomepageController features a series of inner classes, one for every visible button on the HomepageView. Every inner class will trigger the run of the script and the appearance of a new window with the results, either taking the form of ResultsView or RunAllView, depending on the specific command selected by the user (If the user clicks on Run All, the RunAllView will be opened).

To establish communication between these windows, it's necessary to add a Back Button (JButton) on both the ResultsView and RunAllView. Consequently, we'll introduce two new controllers: the RunAllController and the ResultsController. These controllers will each contain one inner class that facilitates the closure of the current window and the seamless transition back to the homepage window. This enhanced structure will contribute to a more user-friendly experience within the application, making the application not just a functional tool, but also an intuitive one.

4.4 Graphical User Interface (GUI)

The GUI, graphical user interface, is a form of user interface that allows users to interact with electronic devices through graphical icons and audio indicator such as primary notation, instead of text-based UIs, typed command labels or text navigation.

The application features a very intuitive graphical interface developed using Java Swing and being build based on a JFrame. The interface is a user-friendly one to be used also by the non-specialized people.

- **Homepage View**
  - Contains 8 buttons (JButtons) for which metric to be measured (memory related tasks or thread-related tasks) or to run all microbenchmarks.
  - When a button is pressed a new window is opened with the values obtained and a plot with the values obtained for every run are displayed). When it comes to Run All Button, a new window is opened with the averages obtained for every metric.

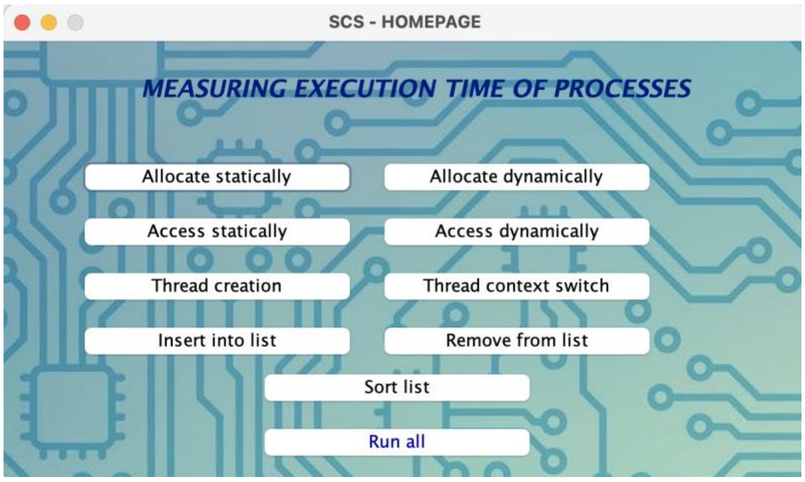
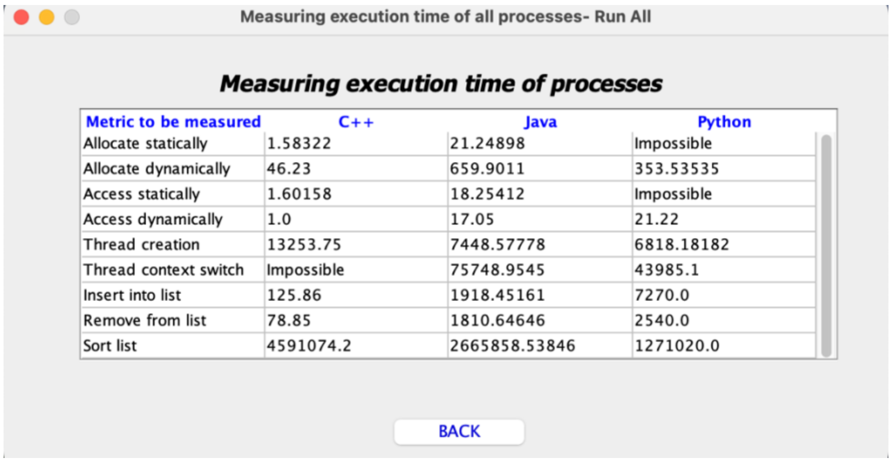


Figure 5: Graphical User Interface – Homepage View

- **RunAll View**
  - Contains a table (JTable) that displays a comprehensive overview of the average results for each memory-related and thread-related command, offering valuable insights to the user regarding the performance metrics and the averages obtained.
  - Additionally, it includes a JButton that serves as a back button, allowing users to easily return to the homepage.


Measuring execution time of all processes- Run All
Measuring execution time of processes
BACK

Metric to be measured	C++	Java	Python
Allocate statically	1.58322	21.24898	Impossible
Allocate dynamically	46.23	659.9011	353.53535
Access statically	1.60158	18.25412	Impossible
Access dynamically	1.0	17.05	21.22
Thread creation	13253.75	7448.57778	6818.18182
Thread context switch	Impossible	75748.9545	43985.1
Insert into list	125.86	1918.45161	7270.0
Remove from list	78.85	1810.64646	2540.0
Sort list	4591074.2	2665858.53846	1271020.0

Figure 6: Graphical User Interface - Run All View

- **Results View**
  - Contains the average obtained for the specific task and a plot (created using JChart) with the values obtained for each run of the programs. Consequently, users can, not only view the average performance for the task, but also the detailed results from every individual run in a plot.
  - Additionally, it includes a JButton that serves as a back button, allowing users to easily return to the homepage.

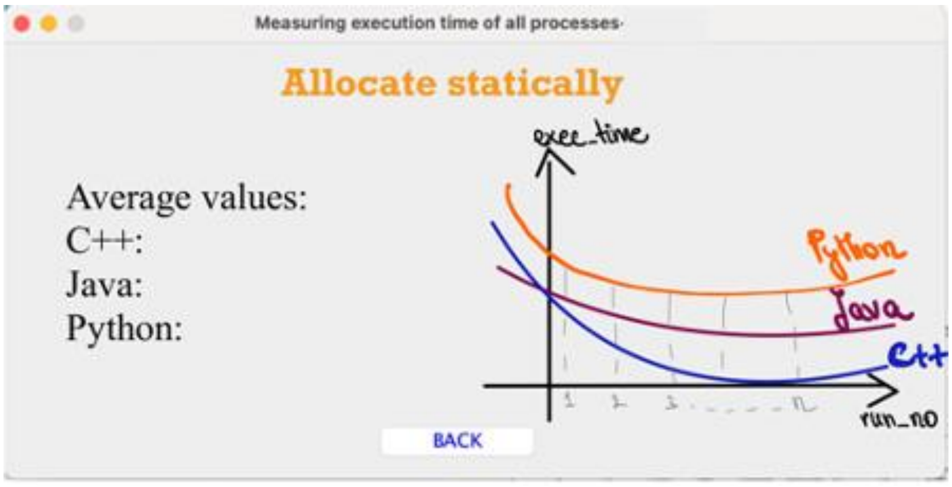


Figure 7: Graphical User Interface - Results View

5. Implementation

5.1 Defined classes and methods

- ***presentation package***
  - ***model sub-package***

a) Language enum with CPP, JAVA and PYTHON used in ResultsView for different hashmaps in order to:

- get the threshold for a specific command

```
Map<Language, Double> languageThresholdMap
   = getThresholds(command);
```

- map the values obtained for execution times for a specific language

```
Map<Language, List<Double>> languageDataMap
   = new HashMap<>();
```

- map the language with the average

```
Map<Language, Double> languageAverageMap
   = new HashMap<>();
```

  - ***view sub-package***

a) HomepageView class

- Represents the homepage view of the project where are defined the buttons necessary for choosing the microbenchmark(s) to run
- Extends the JFrame
- For each button, there exists a method such as
  - `public void addAllocateStaticallyButton (ActionListener actionListener).`

b) *ResultsView class*

- Represents the view where the plot with the values obtained during the 100 runs in each programming language
- Extends JFrame
- It has GridBagLayout and it contains a button (JButton) to return to the homepage and a plot created using JFreeChart with the values obtained during runnings.
- It contains different private methods such as:
  - `private void addAverageValuesLabels(JPanel leftPanel, GridBagConstraints constraints)`
    - Inputs: panel, constraints for the GridBag Layout
    - No outputs
    - Used to add labels for the averages.

  - `private void setAverageLabel(Language language, Map<Language, Double> languageAverageMap)`
    - Inputs: given language, hashmap from which we extract based on the given language the average value
    - No outputs
    - Used to set the text for the labels regarding averages.

  - `private Map<Language, Double> getThresholds(String command)`
    - Inputs: `command` - given command
    - Outputs: the hashmap where we have Language as key and their corresponding average values for a given command as values.
    - Used to return the Hashmap with the needed thresholds to compute the averages given a specified command

  - `private DefaultXYDataset convertDataToXYDataset(Map<Language, List<Double>> languageDataMap)`
    - Inputs: `languageDataMap` - the hashmap where we have as key the language and as values the list of execution times
    - Outputs: XYDataset that stores data in a format suitable for line charts.
    - Converts data to XY dataset for plotting in a graph.

  - `private void computeAveragesForSpecificLanguage (Language currentLanguage, String[] values, Map<Language, Double> languageThresholdMap, Map<Language, List<Double>> languageDataMap, Map<Language, Double> languageAverageMap)`
    - Inputs: `currentLanguage`, `values` - the obtained execution times for that language, `languageThresholdMap` - the hashmap from which we will get the threshold in order to filter the peaks, `languageDataMap` in which we add the values greater or lower than threshold, `languageAverageMap` – in which we add the average obtained
    - No outputs
    - computes the averages for a specified language.

- `private DefaultXYDataset getXYDataset(String command)`
  - Inputs: `command` – the given command in order to take the specific threshold
  - No outputs
  - Read the file, process it line by line and return the dataset used for creating the chart
  - At the same time, it computes the averages for each programming language and sets the labels regarding them

- `private void addPlot(String command)`
  - Inputs: `command` – the given command
  - No outputs
  - Adds a plot to the GUI using JFreeChart.
  - Creates an XY line chart, configures its appearance, and populates it with data from a dataset obtained based on a given command.
  - The resulting chart is displayed within a ChartPanel in the GUI, with specified dimensions and positioning.

c) RunAllView class

- Represents the view containing a table with all the averages obtained for each microbenchmark.
- Extends JFrame
- In addition, contains a button in order to return to the homepage.
- It contains different private methods such as:

- `private DefaultTableModel createModelForTable()`

- No inputs
- Output: the default table model used for creating the table
- Creates the table where on rows we have the microbenchmark name and on columns the programming language. As content, we will display the obtained averages. We return a table model.
- We firstly set the name of rows and columns and then we add data to the table by traversing the benchmark map where we have as key the programming language and as value a map(key – microbenchmark name, value – average obtained)
- We iterate through a nested map structure (languageBenchmarkMap) to update a table model (model) with average benchmark values for different programming languages and benchmark types. If the average is -1, it sets "Impossible" in the corresponding table cell; otherwise, it sets the calculated average.

- `private Map<Language, Map<String, Double>> getLanguageBenchmarkMap()`

- No inputs
- Ouputs: a hashmap where we have as key the language and as value another hashmap (key – microbenchmark name, value – average)
- Creates a nested HashMap where we have as key the programming language and as value another map(key – microbenchmark name, value – computed average).
- We read the benchmark results from the output file we parse and process them to create the nested map.
- It categorizes the data by programming language and benchmark type, computing averages while considering language-specific thresholds.

- `private double computeAverage(String[] values, double threshold)`

- Inputs: `values` - the list of values read from the file, `threshold` - the threshold which is needed to be taken into account in order to remove peaks
- Output: the computed average
- Based on the list of values and the given threshold we compute the average and we round it up using BigDecimal in order to have only 5 decimal places.

- `private Map<Language, Double> getThresholds(String command)`

- Inputs: `command` – the given command
- Output: a map where we have as key – programming language and as value – the threshold
- Used to return the Hashmap(key – programming language, value – threshold) with the needed thresholds to compute the averages given a specified command.

- `private int getRow(String benchmarkType)`

- Input: the name of the benchmark
- Output: the row where we need to add the value based on the name of benchmark
- Based on the name of the microbenchmark we return the number of the row where we should add the value.

- ***controller sub-package***

a) HomepageController class

- Represents the controller of the application, being defined by the private field *HomepageView.*
- *The inner classes* ***AllocateStaticallyListener, AllocateDynamicallyListener, AccessStaticallyListener, AccessDynamicallyListener, ThreadCreationListener, ThreadMigrationListener, ThreadContextSwitchListener, RunAllListener*** implement the ActionListener interface in order to define the corresponding listener for the associated buttons (8 buttons).
- The AllocateStaticallyListener class implements the ActionListener interface, defining an action to be performed when an event happens (the button is clicked). In its actionPerformed method, it executes a shell script (create_tests.sh) to run microbenchmark tests for statically allocating memory in C++, Java, and Python. The script results are stored in a text file, and a graphical message informs the user about the script's execution. If successful, it opens a new view (ResultsView) to display the benchmark results; otherwise, it shows an error message. It goes without saying that we clear the file where the output will be written and we not only create the ResultsView, but also the ResultsController.
- Each inner class was a similar flow as the one for ***AllocateStaticallyListener,*** but we change only the the path of the results file and the name of the command in order to match with the microbenchmark.
- We also have 2 private methods:
  - `private void clearFile(String filePathName)`
    - Inputs: the name of the file, No output (we do not return anything)
    - We clear the file given its path.

- `private void createFileIfNotExist(String filePathName)`
    - Inputs: the name of the file, No output (we do not return anything)
    - Given its path, if the file does not exist we create it .

b) ResultsController class

- Represents the controller for the Results view
- It contains an inner class ***BackButtonListener***
- Listener that responds to the back button being clicked.
- It closes the current ResultsView window, then creates a new HomepageView and HomepageController.

c) RunAllController class

- Represents the controller for the RunAll view
- It contains an inner class ***BackButtonListener***
- Listener that responds to the back button being clicked.
- It closes the current RunAllView window, then creates a new HomepageView and HomepageController.

- **tests package**
    - **create_tests.sh**
- This script serves as the central execution point for running the C++, Java, and Python programs collectively.
- It coordinates the execution of all three programs, ensuring their seamless operation.

- ***c++ sub-package***
    - The package includes an executable, c++_test.cpp, which the script runs. This program takes a command as an argument, executes a specified microbenchmark 100 times, and then outputs the execution time results.
    - The script redirects the obtained results using the ">>" operator to a specified file.

- ***javaTest sub-package***
    - The sub-package includes JavaTest and JavaTest class which the script uses. In the same manner, receives the command as argument and executes the task 100 times and then it displays the results.
    - The script redirects then the results using “>>” into the specified file.

- ***python sub-package***
    - The sub-package includes the python-test file, which, when provided with a command as an argument, performs the specified microbenchmark 100 times and showcases the results obtained from these iterations.
    - The script redirects then the results using “>>” into the specified file.

Within each of these sub-packages, we will find dedicated methods/functions to execute particular benchmarks. These methods are invoked based on the provided command, ensuring that the appropriate benchmark is run according to the specified criteria.

The main difference between the codes is due to the standard libraries used to measure the execution time.

In Java, we use `System.nanoTime()` to capture the start and end times as `long` values. The next step is represented by computing the execution time by obtaining the difference between those two variables. The results are then parsed into a `double` before being returned.

In C++, the approach involves using `high_resolution_clock::now()` for obtaining the start and end times, utilizing `auto` for type inference. The time difference is calculated by casting the value as `duration_cast<nanoseconds>`. The final result is the `count` of nanoseconds.

In Python, we have a simpler approach using `time.time_ns()` for both start and end times. The code simply returns the difference between those two without the need for explicit casting or parsing.

<u>Memory related microbenchmarks</u>

➔ **Allocate Statically**

- Java and C++ only
- Java: `static double allocateStatically()`
- C++: `double allocateStatically()`
- In this microbenchmark, we iterate through a loop 500 times, statically allocating an integer variable, x, with the value 5 inside it. The execution time is then divided by 500.

➔ **Access Statically**

- Java and C++ only
- Java: `static double accessStatically()`
- C++: `double accessStatically()`
- Both codes have a loop (iterates 500 times) that accesses a statically allocated local variable (`localVariable = 23`) and assigns its value to another variable (value).

➔ **Allocate Dynamically**

- Java: `static double allocateDynamically()`
- C++: `double allocateDynamically ()`
- Python: `def allocate_dynamically()`
  - In each language we allocate an array of 500 integers.
  - In Java and C++, we use the new keyword which is explicitly used for dynamic memory allocation. The syntax in Java includes the array type and size directly, while in C++ we declare a pointer.

    C++: `int *array = (int *)malloc(500 * sizeof(int));`

    Java: `int[] array = new int[500];`
  - In Python, the memory management is handled behind the scenes, we create an array of 500 elements and is also initialized with value 0.

    (`x = [0] * 500`)

➔ **Access Dynamically**

- Java: `static double allocateDynamically()`
- C++: `double allocateDynamically ()`
- Python: `def allocate_dynamically()`
- The core logic of dynamically allocating an array, populating it, and accessing its elements within a loop is similar across all three languages.
- The differences lie in the syntax and the way each language handles dynamic memory allocation.
- The C++ code explicitly checks for memory allocation success, which is not present in the Java and Python code.
- The Python code, in particular, showcases a concise way of initializing the array using list comprehension.
  `(array = [i for i in range(500)])`
- In all these programming languages, we have a for, and we assign to x the value of the array of the current index i. Then the difference between end time and start time is divided by 500 since we declare an array of 500 elements, and we measure the access for all these items.

<u>Thread related microbenchmarks</u>

➔ **Thread Creation**

- C++: `double threadCreation()`
- Java: `static double threadCreation()`
- Python: `def thread_creation()`

- In C++, a thread is created using the std::thread class, and it is associated with a function or callable object. We need to also use join function to wait for it, otherwise this is undefined behaviour. (We need to ensure they do not terminate prematurely). `taskForThreadCreation` is a function that is executed when the thread is started. The function simply initializes a local variable x with the value 5.

- In Java, a thread is created by extending the Thread class and overriding the run method. An instance of the custom thread class is created. The run method, which is executed when the thread is started, simply initializes a local variable x with the value 5. The thread is created (`new MyThreadForThreadCreation`()), but it is not started in this example.

- In Python, a thread is created by extending the Thread class from the threading module and overriding the run method. The run method is overridden, but it only initializes a local variable x with the value 5. We create an instance of `MyThreadForThreadCreation` but we do not start it.

- Language specific notes
    - In C++, thread management is part of the C++ Standard Library (<thread> header).
    - In Java, we can also create threads by implementing the Runnable interface.
    - The Python Global Interpreter Lock(GIL), in simple words, is a *mutex (or a lock)* that **allows only one thread to hold the control of the Python interpreter**. This means that **only one thread can be in a state of execution at any point in time.**
    - Java threads are managed by JVM, abstracting low-level details. In the same manner, Python's threading module provides a high-level, multithreading API for creating and managing threads.

➔ Thread Context switch

- C++: `double threadContextSwitch(int numOfThreads)`
- Java:
  `static double threadContextSwitch(int numOfThreads)`
- Python: `def thread_context_switch(num_of_threads)`

- The thread creation is similar in all these three programming languages, we launch the threads in a loop and put them in an list/array and then we wait for them.

- Thread Starting
    - The threads are explicitly started using the start method.

- Joining Threads
    - In Java and C++, the main thread waits for the created one to finish using join().
    - In Python, we can wait also to finish the thread by using join().

- Synchronization mechanisms
    - In C++:
      ```
      std::mutex mutex_
      std::unique_lock<std::mutex> lock(mutex_)
      ```
    - In Java: `synchronized` for allowing only one thread to access that block of code
    - In Python: `lock = threading.Lock()`

- Shared Variable Increment
    - In all three programming languages we have a global variable (a shared variable) which will be incremented when a thread accesses that piece of block.
    - We use synchronization mechanism also to ensure the atomicity for the increment operation in the critical region.

- Language specific notes
  - In Java, the synchronized keyword ensures that only one thread can execute at a time the safeIncrement method.
  - In Python, we have a try-finally block to ensure that the lock is released even if an exception occurred.
  - Also, we have in safe_increment function from Python global keyword to indicate that is a global variable across different scopes. In this case, it is a shared resource across multiple threads.

<u>Operation with Lists Microbenchmarks</u>

We need a function/method to generate a random list of a specific size.

- C++:

- `list<int> generateRandomList(int size)`
- To generate random numbers we seed the random number generator with the current time: `srand(time(NULL))` and then use `rand() % size`

- Java:

- `static List<Integer> generateRandomList(int size)`
- To generate random numbers, we use Random class and we declare an object `private static final Random random = new Random()` and then we use `random.nextInt() % size`

- Python:

- `def generate_random_list(size, min_value, max_value)`
- To generate random numbers, we use `random` library and then we use `random.randint(min_value, max_value)`

For lists we use:

- C++: `std::list`
- Java: `List<Integer>` from `java.util.package`
- Python: uses the built-in `list` data structure.

➔ **Insertion into List**

- C++:
  ```
  double insertIntoList(list<int> randomList, int
  value, int positionToInsert)
  ```

- Java:
  ```
  static double insertIntoList(List<Integer> list,
  int value, int positionToInsert)
  ```

- Python:
  ```
  def insert_into_list(random_list, value,
  position_to_insert)
  ```

- Language specific notes
  - In C++, the iterator must be advanced to the insertion position before calling insert(`randomList.insert(it, value)`)
    ```
    list<int>::iterator it = randomList.begin();
    advance(it, positionToInsert);
    ```

- In Java, the `add` method is used directly on the list by specifying the position.
- In Python, the `insert` method is used directly on the list by specifying the position.

➔ **Remove from List**

- C++:
  `double removeFromList(list<int> randomList, int position)`
- Java:
  `static double removeFromList(List<Integer> list, int position)`
- Python:
  `def remove_from_list(random_list, position)`

- Language specific notes
  - In C++, an iterator (`std::list::iterator`) is used to advance to the specified position before calling erase(`randomList.erase(it)`)
    ```
    list<int>::iterator it = randomList.begin();
    advance(it, position);
    ```
  - In Java, the `remove` method directly takes the position.
  - In Python, the `del` statement is used directly on the list, and position indexing is used.

➔ **Sort the List**

- C++: `double sortList(list<int> randomList)`
- Java: `static double sortList(List<Integer> list)`
- Python: `def sort_list(random_list)`

- Language specific notes
  - Python uses the `sort` method directly on the list.
  - Java uses `Collections.sort` to sort the list.
  - C++ uses the `sort` function directly on the `std::list`.

▪ ***results sub-package***

- The sub-package includes multiple files, each of which captures the results obtained for individual microbenchmarks (according to the name of the file).
- For every distinct microbenchmark, there exists a dedicated file that comprehensively records the specific performance data and outcomes. This organized approach allows for a detailed analysis of each benchmark's results.

- **start package** contains the Main class for executing the program which creates 2 objects a HomepageView and a HomepageController.

5.2 Used library- JFreeChart

JFreeChart is an open-source framework for the programming language Java, which allows the creation of a wide variety of both interactive and non-interactive charts.

It is possible to place various markers and annotations on the plot. JFreeChart automatically draws the axis scales and legends. Charts in GUI automatically get the capability to zoom with mouse and change some settings through local menus. The existing charts can be easily updated through the listeners that the library has on its data collections.

<u>Objects specific to JFreeChart used:</u>

- `DefaultXYDataset`
  - Represents an XYDataset that stores data in a format suitable for line charts.
- `JFreeChart`
  - Represents the chart itself, created using `ChartFactory.createXYLineChart.`
- `XYPlot`
  - Represents the plot area of the chart where data is visualized.
- `XYLineAndShapeRenderer`
  - Allows customization of lines and shapes in the plot.
- `ChartPanel`
  - A Swing component that displays the chart.

This library is used in ***ResultsView*** class in the following methods:

- `private DefaultXYDataset getXYDataset(String command)`
  - This method reads data from a file and organizes it into an XYDataset format, which is a required format for JFreeChart.
  - The data is read line by line from the file. Lines starting with "CPP," "JAVA," or "PYTHON" indicate the start of data for a specific programming language.
  - Data is collected and organized into maps (languageDataMap and languageAverageMap) based on the programming language.
  - The collected data is then converted into an XYDataset using the `convertDataToXYDataset` method.

- `private DefaultXYDataset convertDataToXYDataset(Map<Language, List<Double>> languageDataMap)`
  - Converts data to XY dataset for plotting in a graph.
  - This method takes the organized data (in the form of lists) for each programming language and converts it into an XYDataset that can be used by JFreeChart.
  - For each programming language, a series is added to the dataset. The data is formatted as a 2D array with x and y values.

  - *<u>Steps</u>*:
    - Iterate over each entry in the languageDataMap, where the key is the programming language (Language Enum) and the value is a list of execution time values for that language.
    - For each programming language, we create a matrix (2D array) to store x and y values for each data point. The array has two rows ([2]): the first row for x-

values and the second row for y-values. The number of columns is determined by the size of the languageValues list.
    - Use a loop to populate the data array with x and y values for each data point.
        - `data[0][i] = i + 1:`
          Assign the x-value, where i + 1 represents the index adjusted to start from 1.
        - `data[1][i] = languageValues.get(i):`
          Assign the y-value, retrieved from the languageValues list.
    - Add the series (programming language) with its associated data (data) to the dataset using `dataset.addSeries.`
    - The series is identified by the programming language enum, and the data is the 2D array containing x and y values.
    - Return the populated dataset containing all the series and their data points.

- `private void addPlot(String command)`
    - Adds a plot (line chart) to the GUI using JFreeChart. It creates a JFreeChart instance using the `ChartFactory.createXYLineChart` method.
    - Creates an XY line chart, configures its appearance, and populates it with data from a dataset obtained based on a given command.
    - The `XYPlot` and `XYLineAndShapeRenderer` are used to customize the appearance of the chart, such as making lines and shapes visible.
    - A ChartPanel is created to display the chart, and it is added to the GUI.

6. Tests/ Experiments

In this section, we are diving into different experiments to measure the execution time for different microbenchmarks.

At first, we are evaluating the memory-tasks, both static and dynamic and for the dynamic ones we will range the array size for the dynamic tasks in order to see their impact on performance and for the static we will range the number of runs.

Furthermore, we will take a look at the thread-related tasks, thread creation and thread context switch. We will manipulate the number of threads in order to highlight how this change affects the performance, but for thread creation we will run 100 and 500 iterations to get a solid grasp of how our system performs under different conditions.

Lastly, we are exploring list operations (insertion into list, deletion from list and sorting the list). We will vary parameters such as list size, position where we will insert, respectively delete (front of the list, end of the list or specific position) and assess their implications on execution times.

The goal is to identify which programming language performs better when it comes to a specific scenario.

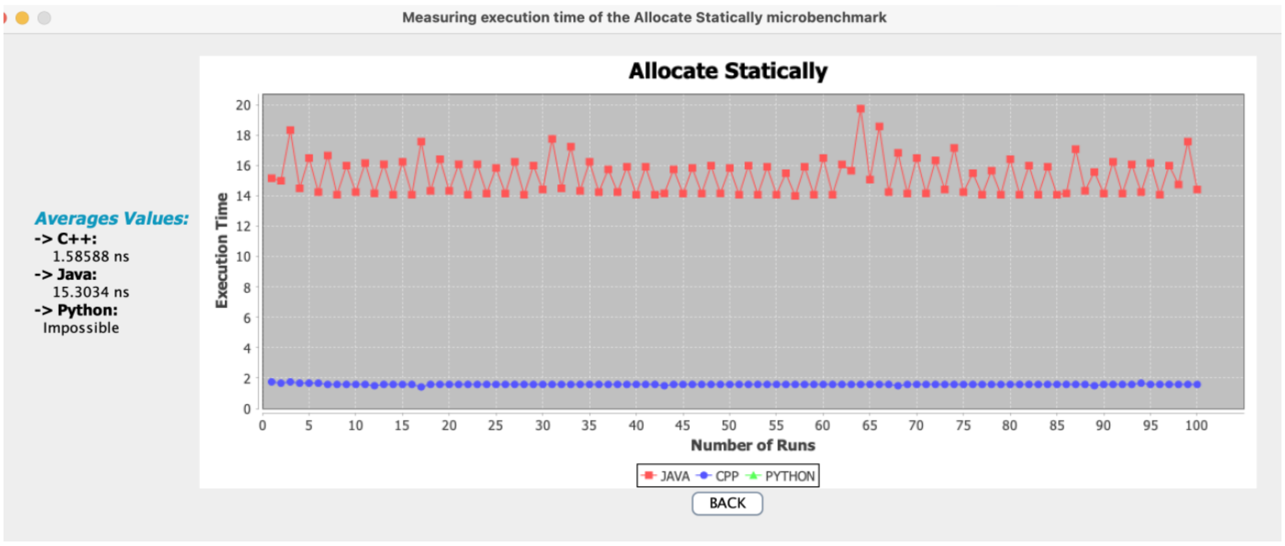
6.1 Memory-related microbenchmarks

6.1.1 Static allocation (C++ and Java only)

1. **Static allocation test with 100 runs**

Steps:

- We have a for loop of 500 and we allocate a variable x with the value 5.
- The execution time is divided by 500 to provide accuracy.



Averages Values:
-> C++:
1.58588 ns
-> Java:
15.3034 ns
-> Python:
Impossible

Figure 8: Allocate Statically Test with 100 runs

Average Execution Time results:

- C++: 1.58588 ns
- Java: 15.3034 ns

Individual run times:

- C++: ranging from 1.418 ns to 1.75 ns.
- Java: ranging from 14.082 ns to 19.75 ns

Analysis of the obtained results:

- C++ exhibits very low execution times, suggesting that C++ is highly efficient for such simple operations, and its performance is consistent across runs.
- Java performs reasonably well this task, but the individual run times may vary falling though in an expected range.
- C++ outperforms Java significantly in this specific task. The average execution time for **C++ is almost ten times faster than Java**.

2. **Static allocation test with 500 runs**
Steps: The same steps as for the test with 100 runs.

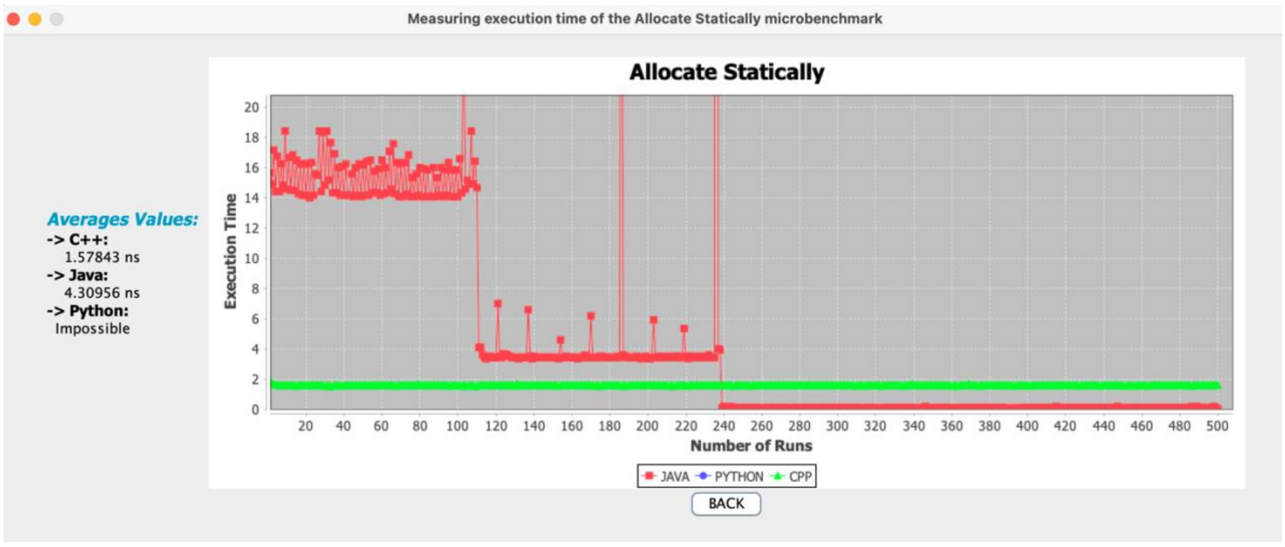


Figure 9: Static allocation test with 500 runs

Average Execution Time results:
- C++: 1.57843 ns
- Java: 4.30964 ns

Individual run times:
- C++: ranging from 1.5 ns to 1.75 ns.
- Java: ranging from 0 ns to 1010.0 ns

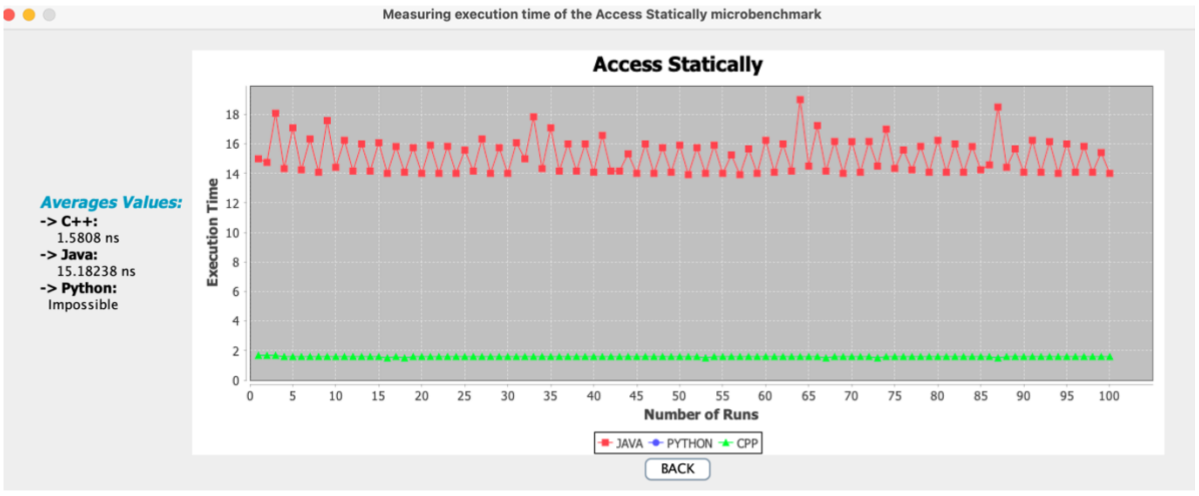
Analysis of the obtained results:
- We can notice that for C++, the results are similar to the ones from the test with 100 runs, C++ outperforms Java at this microbenchmark. The results are consistent along the runs
- When it comes to Java, execution times vary widely. The first 100 runs were between 14 and 19 ns, the next 140 runs were smaller between 3 and 7ns and the last ones smaller than 1. This is due to JVM warm-up. As we can see, after 240 runs, the execution times in Java are smaller than the ones in C++.
- **JVM warm-up** refers to the initial phase when a Java program is executed. During this phase, the JVM performs various tasks, such as loading classes, initializing static variables, and optimizing code through Just-In-Time (JIT) compilation. As a result, the initial executions of a Java program might have higher execution times compared to subsequent runs
- It's essential to run the benchmark multiple times to allow the JVM to reach a stable state where optimizations have been applied. This can lead to more accurate measurements of the actual performance of the Java program.

6.1.2 Static access (C++ and Java only)

1. **Static access test with 100 runs**

Steps:

- We initialize a local variable (localVariable) with the value 23.
- We have a for loop of 500 and we put the value of localVariable into value variable.
- We divide the execution time obtained by 500.



Averages Values:
-> C++:
1.5808 ns
-> Java:
15.18238 ns
-> Python:
Impossible

Figure 10: Access statically test with 100 runs

Average Execution Time results:

- C++: 1.5808 ns
- Java: 15.18238 ns

Individual run times:

- C++: ranging from 1.5 ns to 1.584 ns.
- Java: ranging from 13.916 ns to 19.0 ns.

Analysis of the obtained results:

- C++ exhibits very low execution times, suggesting that C++ is highly efficient for such simple operations, and its performance is consistent across runs.
- Java’s individual run times exhibit some variability, with the slowest run being 19.0 ns. Java's execution time is noticeably higher than C++, reflecting the inherent overhead associated with Java's runtime environment.
- C++ outperforms Java significantly in this specific task. The average execution time for **C++ is almost ten times faster than Java**.

Remarks on static allocation and access:

- We can see that static allocation and static access have approximately the same execution times.
- The stability and low variability in execution times for both static allocation and static access in C++ indicate that the compiler is able to optimize these operations effectively, resulting in efficient and predictable performance.
- This is a positive outcome, as it suggests that the C++ language and compiler are well-suited for handling such low-level memory operations with minimal overhead.

2. **Static access test with 500 runs**

Steps: The same as for the static access test with 100 runs.

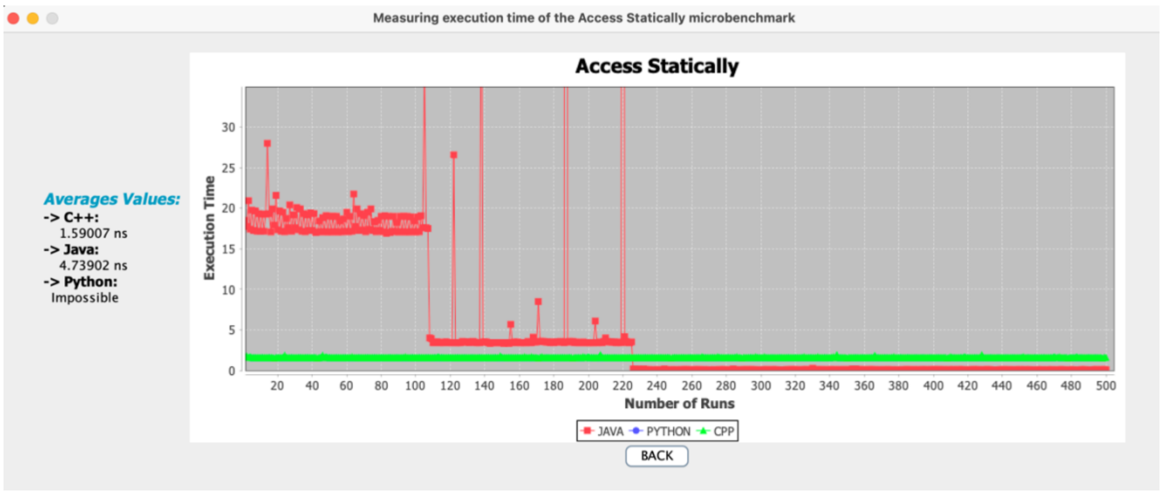


Figure 11: Static access test with 500 runs

Average Execution Time results:

- C++: 1.59007 ns
- Java: 4.73902 ns

Individual run times:

- C++: ranging from 1.5 ns to 1.916 ns.
- Java: ranging from 0.0 ns to 497.666 ns.

Analysis of the obtained results:

- C++ exhibits very low execution times, suggesting that C++ is highly efficient for such simple operations, and its performance is consistent across runs. The results are similar to the ones from the access statically test with 100 runs.
- When it comes to Java, we can notice the same difference as from the one between static allocation with 100 runs and the one with 500 runs. The results are decremented along the runs due to JVM warm-up, outperforming after 220 runs the ones obtained by C++.
- C++ outperforms Java significantly in this specific task. The average execution time for **C++ is almost 5 times faster than Java**, but after 220 runs Java outperforms.

6.1.3 Dynamic allocation

1. **Dynamic allocation (array of 500 elements) test with 100 runs**

Steps: We initialize a dynamic array of 500 elements.

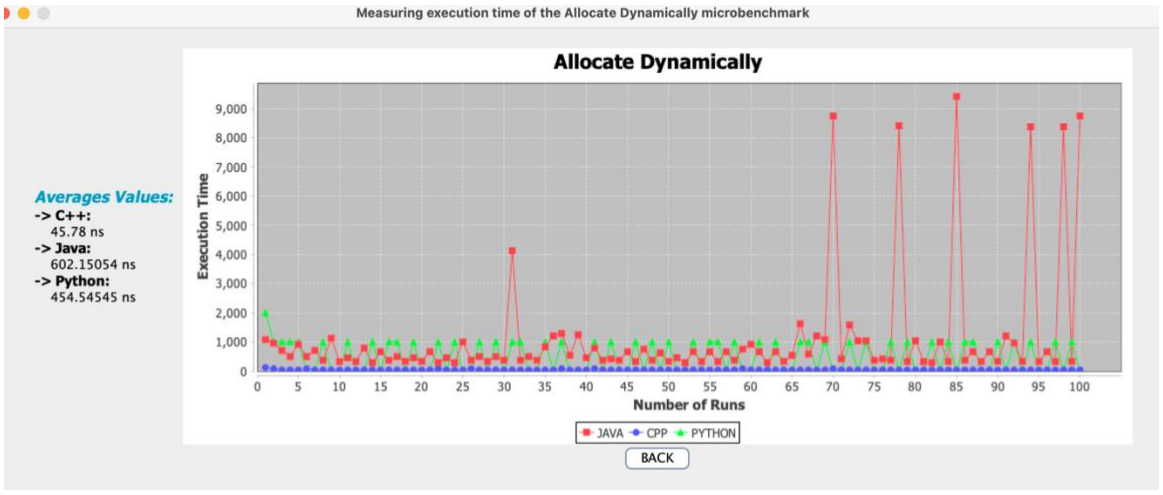


Figure 12: Dynamic allocation of a 500 array test with 100 runs

Average Execution Time results:

- C++: 45.78 ns
- Java: 602.150 ns
- Python: 452.54 ns

Individual run times:

- C++: ranging from 41 ns to 125 ns. (dominant values: 41, 42, 82, 84 ns)
- Java: ranging from 291 ns to 8750 ns.
- Python: ranging from 0 ns to 1000 ns.

Analysis of the obtained results:

- The individual run times in C++ are relatively close to each other, indicating stable and efficient performance across iterations. C++ is known for its efficient memory management, and this result reflects its capability in handling dynamic memory allocation tasks with low overhead.
- Java's individual runs vary significantly, the lowest one being 8750 ns. Java's execution time is noticeably higher than C++, suggesting more overhead in dynamic memory allocation tasks, due to the JVM's memory management.
- Python's results vary, but on average it falls between the efficiency of C++ and the higher execution times noticed in Java, due to the existence of the interpreter which contributes to the variability in performance.
- C++ outperforms both Java and Python significantly in dynamic allocation, C++ **is almost 13 times faster than Java** and **C++ is almost 10 times faster than Python.** C++ exhibits more consistent and efficient behavior in handling dynamic memory allocation tasks.

2. **Dynamic allocation (array of 100 elements) test with 100 runs**
Steps: We initialize a dynamic array of 100 elements.

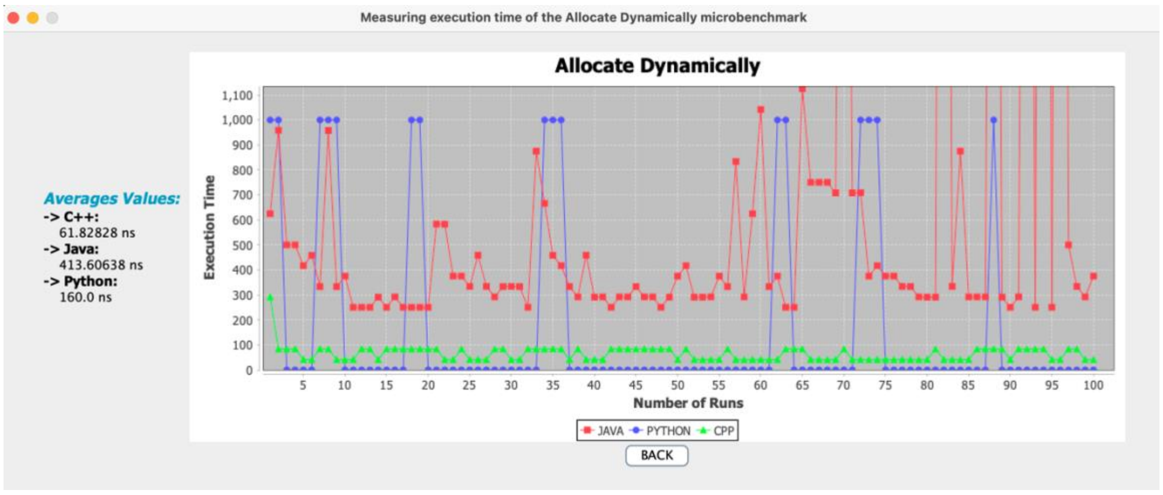


Figure 13: Dynamic allocation of an array of 100 elements

Average Execution Time results:

- C++: 61.84828 ns
- Java: 413.60638 ns
- Python: 160.0 ns

Analysis of the obtained results:

- C++ outperforms both Java and Python significantly in dynamic allocation due to the absence of JVM or Python interpreter must stop for a while and trace the object's lifetime.
- We can notice that C++ has a greater average on access of a smaller array (100 elements) than the one of 500 elements, while the other two programs have smaller average when the size of the array is smaller due to the compiler optimization that might influence the assembly code generated for different array sizes.

3. **Dynamic allocation (array of 10000 elements) test with 100 runs**
Steps: We initialize a dynamic array of 10000 elements.

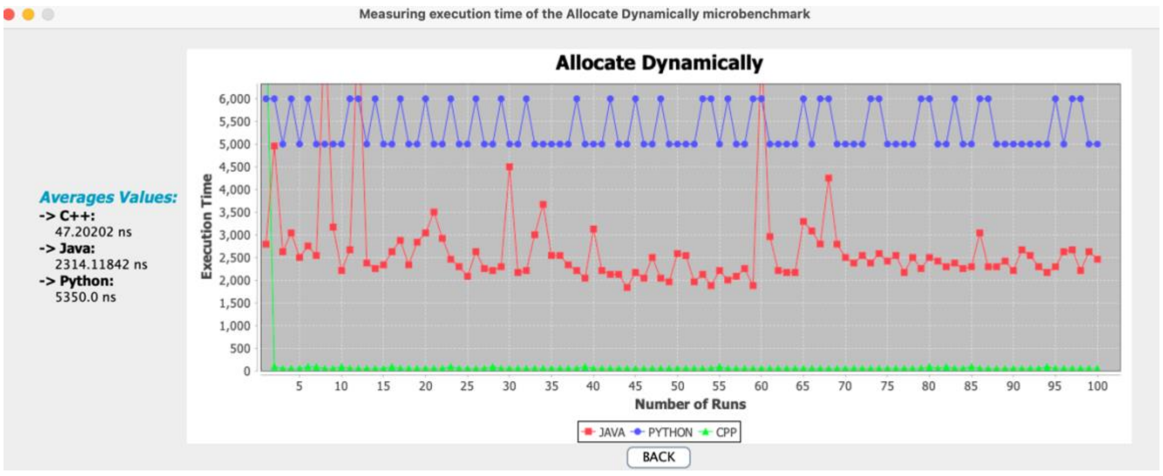


Figure 14: Dynamic allocation of an array of 10000 elements with 100 runs

Average Execution Time results:

- C++: 47.20202 ns
- Java: 2314.11842 ns
- Python: 5350.0 ns

Analysis of the obtained results:

- C++ outperforms both Java and Python significantly in dynamic allocation due to the absence of JVM or Python interpreter must stop for a while and trace the object's lifetime.
- We can notice that the averages of execution times in Java, respectively in Python have increased since the size of the array has increased, while in C++ the average results is closer to the one obtained for a smaller array of 500 elements. The memory allocation overhead in C++ might be less affected by the size of the allocated array, leading to more consistent performance across different array sizes.
- Java and Python are higher-level languages with runtime environments that introduce additional abstractions and overhead. The memory allocation process in these languages might involve more complex operations, leading to increased overhead with larger array sizes, plus the existence of Garbage Collector.

6.1.4 Dynamic Access

1. **Dynamic access (array of 500 elements) test with 100 runs**

Steps:

- We initialize a dynamic array of 500 elements.
- We put values equals to the indexes.
- We access each element from array.
- We divide the execution time by the size of the array.

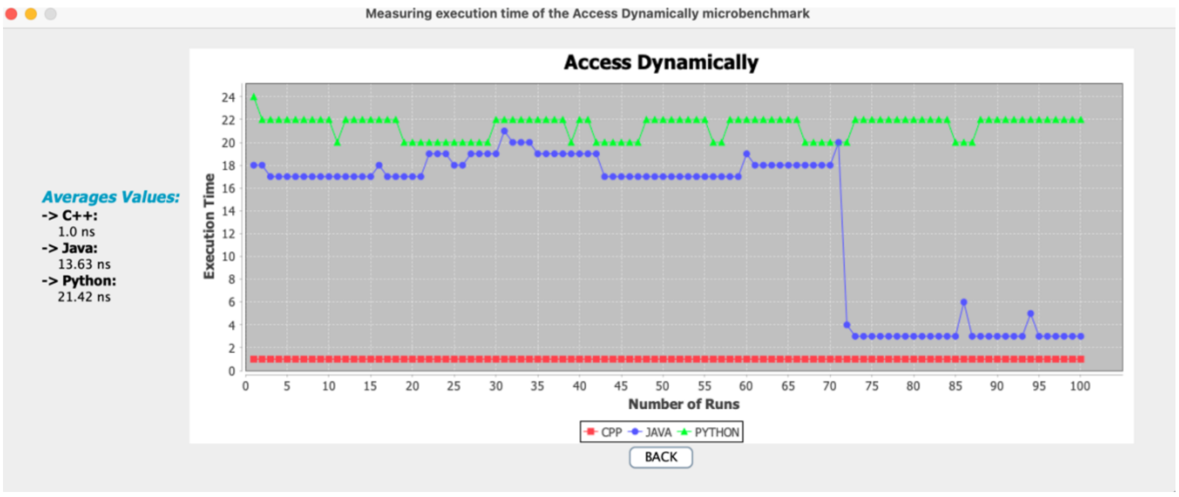


Figure 15: Dynamic access in a 500 elements array test with 100 runs

Average Execution Time results:

- C++: 1.0 ns
- Java: 13.63 ns
- Python: 21.42 ns

Individual run times:

- C++: Consistently 1 ns across all runs.
- Java: ranging from 3 ns to 21 ns.
- Python: ranging from 20 ns to 24 ns.

Analysis of the obtained results:

- The C++ implementation for dynamic memory access demonstrates extremely low and consistent execution times, with an average of 1 ns. All individual run times are consistently 1 ns, suggesting highly efficient memory allocation in C++. C++ is known for its low-level memory control, and this result reflects its capability in managing dynamic memory with minimal overhead.
- Java shows higher and more variable execution times for dynamic memory allocation, with an average time of 13.63 ns. The individual run times vary, with the slowest run being 21 ns. Java's execution time is notably higher than C++, indicating potential overhead introduced by the JVM's memory management during dynamic allocation tasks.
- Python's results are comparable to Java in this specific task, and both show variability in performance. Python's memory management, being automatic and dynamic, introduces some overhead compared to C++.
- C++ outperforms both Java and Python significantly in dynamic allocation, C**++ is almost 13 times faster than Java** and **C++ is almost 21 times faster than Python.**

2. **Dynamic access (array of 100 elements) test with 100 runs**

Steps: Same steps as for dynamic access of elements of an array of 500 elments.

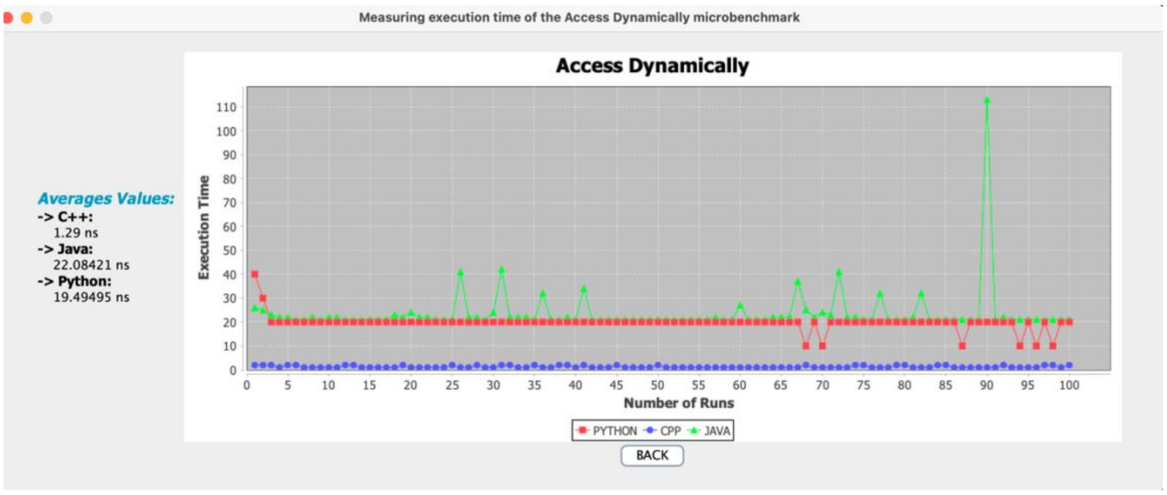


Figure 16: Dynamic access in a 100 elements array with 100 runs

Average Execution Time results:

- C++: 1.29 ns
- Java: 22.08 ns
- Python: 19.49495 ns

Analysis of the obtained results:

- The C++ implementation for dynamic memory access demonstrates extremely low and consistent execution times, with an average of 1.29ns.

- We can notice that the results for accessing an element in a smaller array of 100 elements are closer to the ones from the test with accessing an element in an array of 500 elements.
- When it comes to Java and C++, the average execution time are greater on a smaller array than on the one with 500 items. Smaller arrays might have relatively higher memory allocation overhead compared to their size. This overhead could impact the overall performance, especially in scenarios where memory allocation dominates.
- In Java, the Just-In-Time (JIT) compiler might take some time to optimize the code. If the benchmarking involves short-lived processes, the JIT compilation overhead might impact results differently for smaller and larger array.

2. **Dynamic access (array of 10000 elements) test with 100 runs**

Steps: Same steps as for dynamic access of elements of an array of 500 elements.

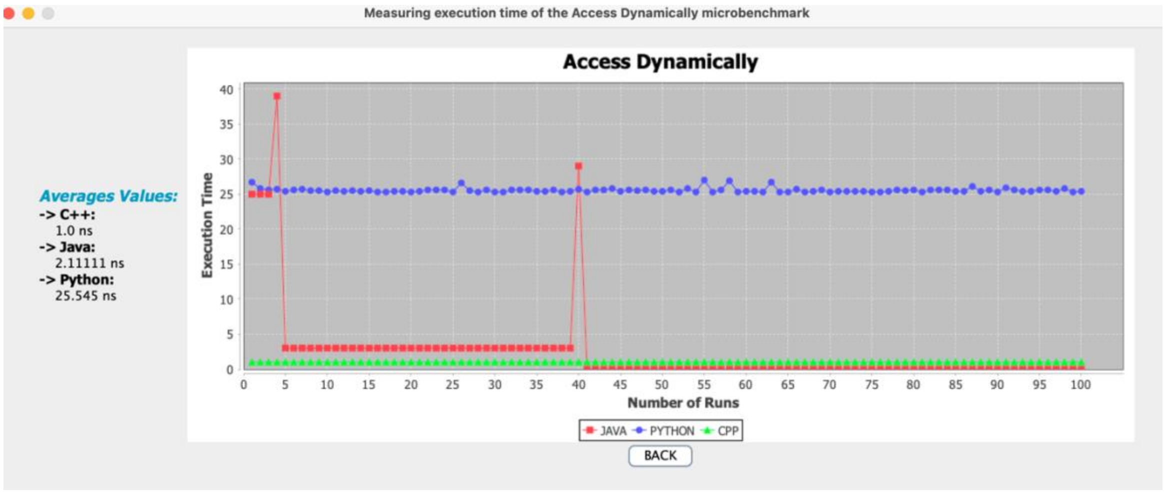


Figure 17: Dynamic access of an array of 10000 elements with 100 runs

Average Execution Time results:

- C++: 1.0 ns
- Java: 2.11111 ns
- Python: 25.545 ns

Analysis of the obtained results:

- The C++ implementation for dynamic memory access demonstrates extremely low and consistent execution times having only values of 1 ns in all these 3 tests where we ranged the size of the array (100, 500, 10000).
- When it comes to Java, we have some peaks, which may be due overhead of JVM.
- We can notice that the average values for dynamic access microbenchmark are far smaller for an array with greater size, the difference is significant in the case of Java. We had 22 ns for an array of 100 elements, 13.63 ns for one of 500 elements and now only 2.11111 ns. This is due to the JIT compilation whose overhead might impact results differently for smaller and larger array.

6.2 Thread-related microbenchmarks

6.2.1 Thread creation

1. **Thread creation test with 100 runs**

Steps: We simply create a thread that runs a specific function that contains the initialization of a variable (x =5).

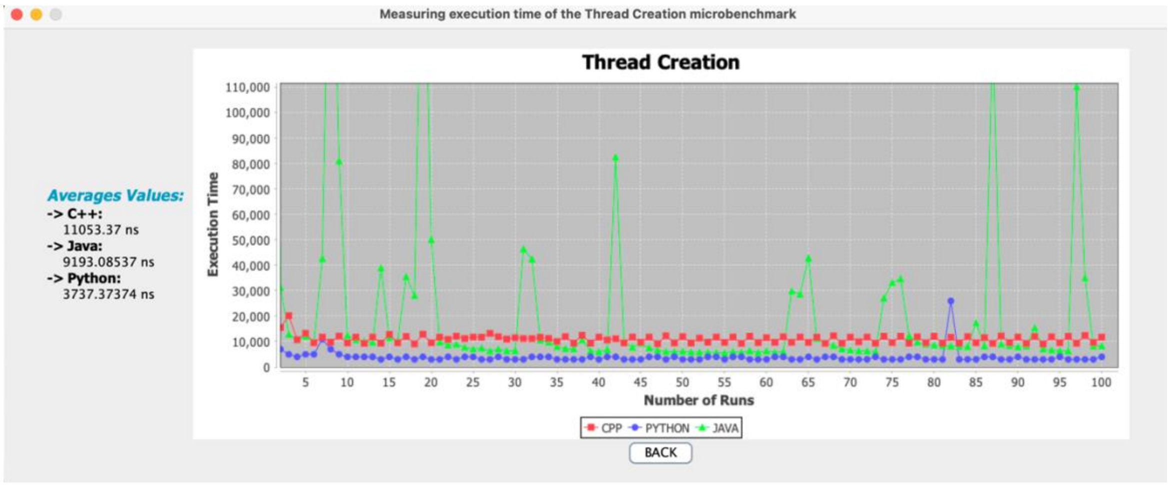


Figure 18: Thread creation test with 100 runs

Average Execution Time results:

- C++: 11,053.37 ns
- Java: 9,193.08537 ns
- Python: 3,737.37374 ns

Individual run times:

- C++: ranging from 9,084.0 ns to 20,209 ns.
- Java: ranging from 5,542.0 ns to 424,459.0 ns
- Python: ranging from 3000 ns to 14,000 ns.

Analysis of the obtained results:

- C++ individual run times show variability but remain within a reasonable range. The high average value appears due to the fact we need to manually join or detach threads to ensure they don't terminate prematurely.
- Java demonstrates a lower average execution time for thread creation. The JVM contributes to a more efficient thread creation process compared to C++. However, the variability in execution times might suggest some fluctuations in performance.
- Python stands out with a significantly lower average execution time for thread creation, averaging 3,730.0 ns due to Python's lightweight threading model and the Global Interpreter Lock (GIL). Python's threading capabilities appear to be quite efficient, resulting in quick and consistent thread creation times.
- **Python is almost 5 times faster than C++ and almost 3 times faster than Java**, but we need to take into account Python Global Interpreter Lock(GIL), in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that even in a multi-core system, multiple threads in a Python program cannot execute Python bytecode in parallel. The GIL can be a limiting factor for performance in CPU-bound and multithreaded applications.

2. **Thread creation test with 500 runs**

Steps: The same as the one with the thread creation test with 100 runs.

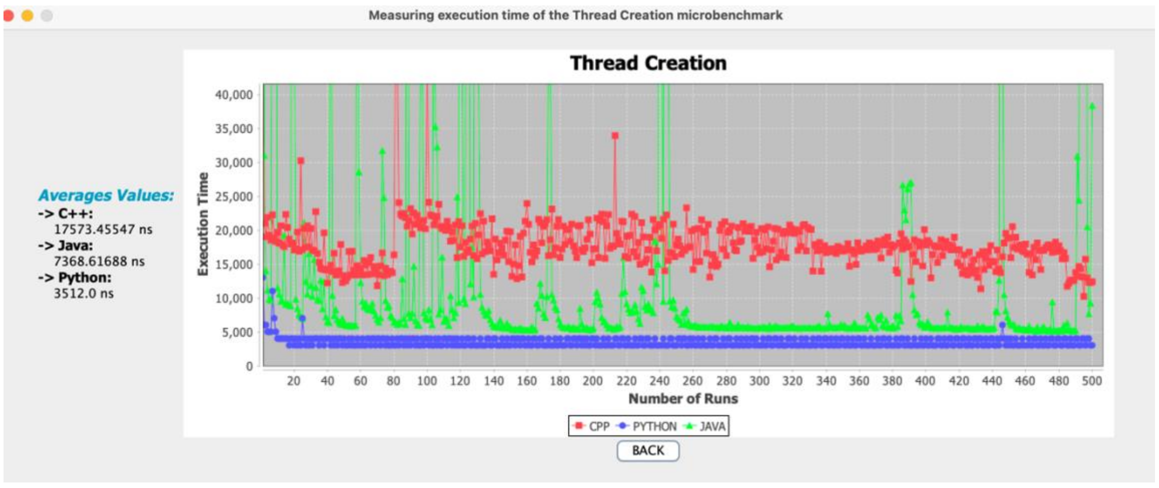


Figure 19: Thread creation test with 500 runs

Average Execution Time results:

- C++: 17,573.45547 ns
- Java: 7,368.61688 s
- Python: 3,512.0 ns

Individual run times:

- C++: ranging from 10,208.0 ns to 52,958.0 ns.
- Java: ranging from 5,083.0 ns to 530,500.0 ns
- Python: ranging from 3000 ns to 13,000 ns.

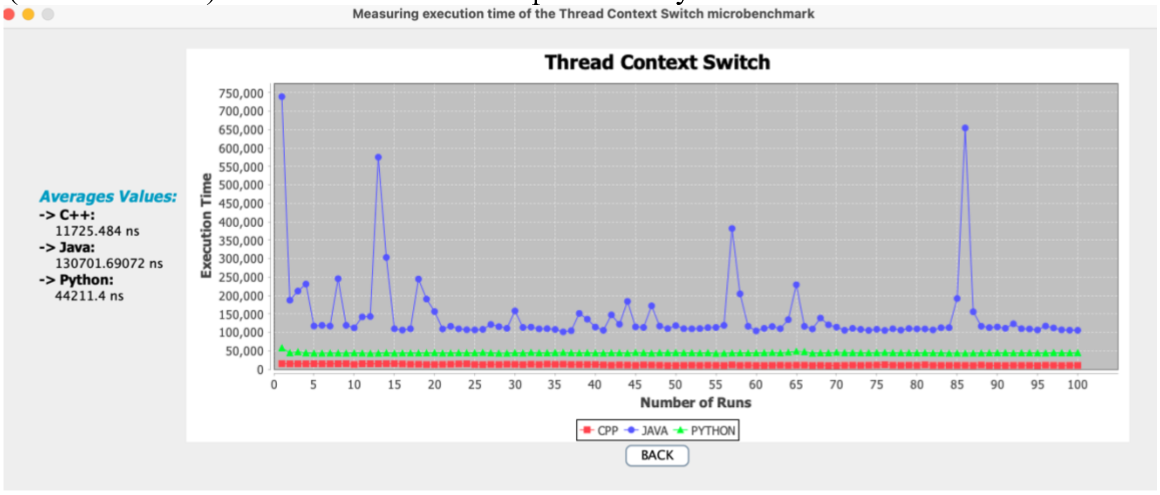
Analysis of the obtained results:

- C++ individual run times show variability but remain within a reasonable range eliminating the peaks. The high average value appears due to the fact we need to manually join or detach threads to ensure they don't terminate prematurely.
- Java demonstrates a lower average execution time for thread creation. The JVM contributes to a more efficient thread creation process compared to C++. However, the variability in execution times might suggest some fluctuations in performance.
- Python's threading capabilities appear to be quite efficient, resulting in quick and consistent thread creation times.
- C++ and Java exhibit more variability in execution times, while Python shows more consistency.
- The results are similar to the ones obtained from the thread creation test with 100 runs, but Java has a better average while C++ has a worse one, **Python being 2 times faster than Java and almost 6 times faster than C++.**

6.2.2 Thread context switch

1. **Thread context switch (50 threads) test with 100 runs**

Steps: We simply create 50 that concurrently increment a shared global variable (sharedNumber) within a critical section protected by a mutex



Averages Values:
-> C++:
11725.484 ns
-> Java:
130701.69072 ns
-> Python:
44211.4 ns

Figure 20: Thread context switch of 50 threads in 100 runs

Average Execution Time results:

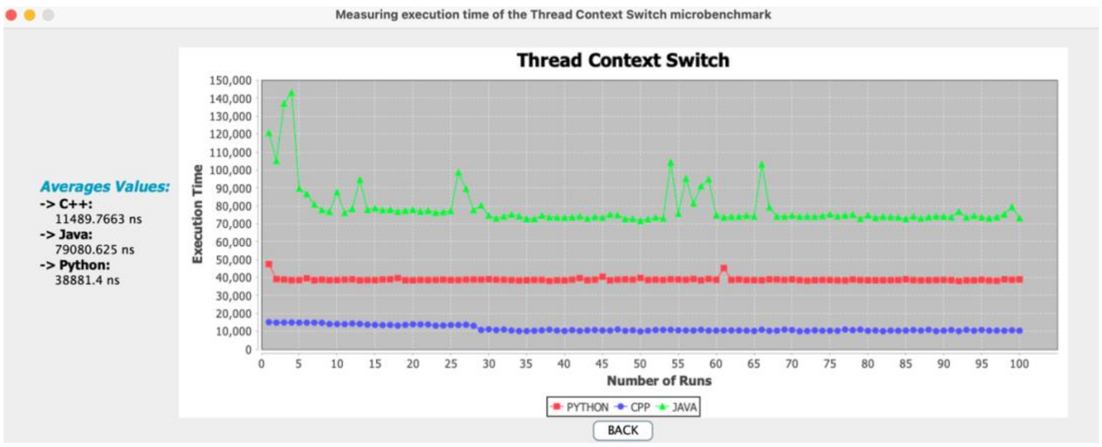
- C++: 11,725.484 ns
- Java: 130,701.69072 ns
- Python: 44,211.4 ns

Analysis of the obtained results:

- C++ shows the lowest average execution time, suggesting efficient thread context switching.
- We can notice that for both tests (the one with thread context switch of 100 threads and this one) the average results for the test with more threads are better than the current one, suggesting that the context switch time is smaller as the number of threads increases.

2. **Thread context switch (100 threads) test with 100 runs**

Steps: Similar the thread context switch test previously mentioned, but with 100 threads.



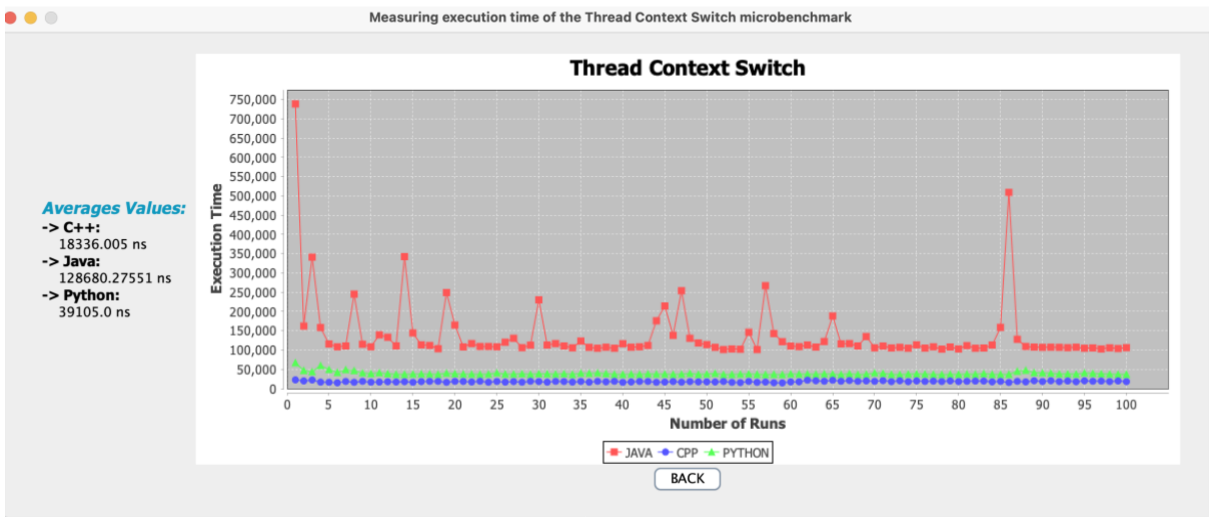
Averages Values:
-> C++:
11489.7663 ns
-> Java:
79080.625 ns
-> Python:
38881.4 ns

Figure 21: Thread context switch of 100 threads in 100 runs

Average Execution Time results:
- C++: 11,489.7663 ns
- Java: 79,080.625 ns
- Python: 38,881.4 ns

Individual run times:
- C++: ranging from 9,594.17 ns to 26,297.5 ns.
- Java: ranging from 572,550.83 ns to 140,753.33 ns.
- Python: ranging from 38,360.0 ns to 40,780.0 ns.

Analysis of the obtained results:
- C++ shows the lowest average execution time, suggesting efficient thread context switching.
- Java exhibits higher average execution times compared to C++, indicating a relatively higher cost of thread context switching. Java, while providing portability and ease of development, may incur additional overhead in certain low-level operations compared to C++.
- Even though Python results are smaller than the ones in Java, we know that only a thread can be in the state of execution due to GIL.
- Both C++ and Python show a slight reduction in average context switch time compared to the individual result for 50 threads. The threading implementations in both languages, including the use of mutexes, contribute to this behavior.
- Java exhibits the best average context switch time among the three languages, showing a significant improvement compared to the individual result for 50 threads. This could be attributed to Java's thread management and synchronization mechanisms.
- We can also notice that there is a quite improvement when the number of threads increases if we run the same code, but only for two threads. This one will have the worst results.



Averages Values:
-> C++:
18336.005 ns
-> Java:
128680.27551 ns
-> Python:
39105.0 ns

Figure 22: Thread context switch of 2 threads in 100 runs

6.3 Operation with lists microbenchmarks

6.3.1 Insertion into list

1. **Insertion into a list of 10000 elements at position 500 test with 100 runs**

Steps: We create a random list, and we insert the value 23 on the 500 positions.

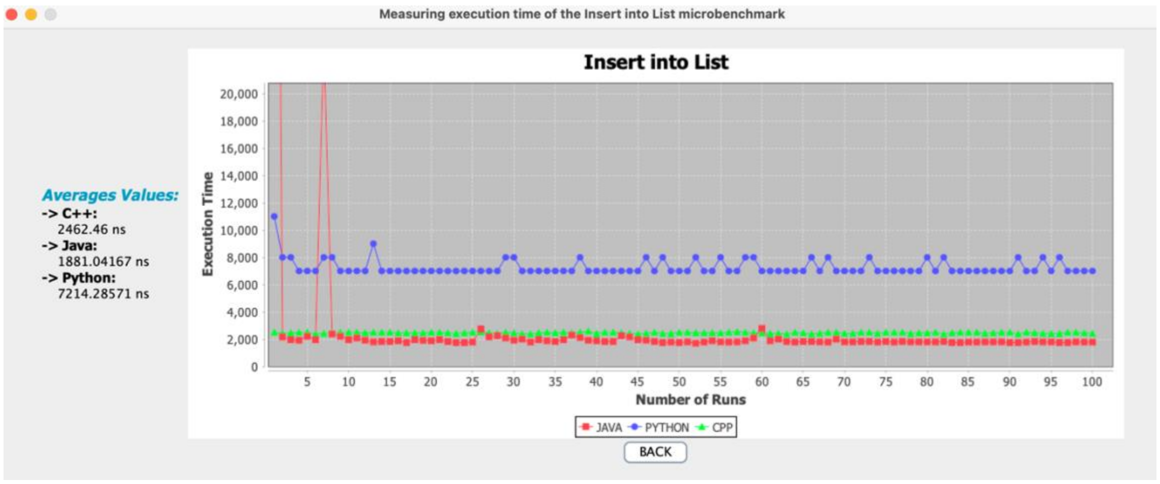


Figure 23: Insert on position 500 in a list of 10000 elements

Average Execution Time results:

- C++: 2462.46 ns
- Java: 1881.0467 ns
- Python: 7214.28571 ns

Individual run times:

- C++: ranging from 2416.0 ns to 2500.0 ns.
- Java: ranging from 1791.0 ns to 250042.0 ns.
- Python: ranging from 7000 ns to 11000.0 ns.

Analysis of the obtained results:

- C++ shows relatively consistent performance with a few higher peaks. The average execution time is the lowest among the three languages due to the fact that we need to use an iterator to advance to that position.
- Java exhibits a wider range of execution times, with some significant peaks. Eliminating those peaks, we can notice that is the fastest one.
- Python has the highest average execution time, and the values are consistent within the observed range. The interpreted nature of Python, along with its dynamic typing and potential list resizing, could contribute to the increased overhead in this specific operation.
- **Java is the most suitable for insertion into list,** being almost **4 times faster than Python.**

2. **Insertion into a list of 10000 elements at front test with 100 runs**
Steps: We create a random list, and we insert the value 23 on the first position.

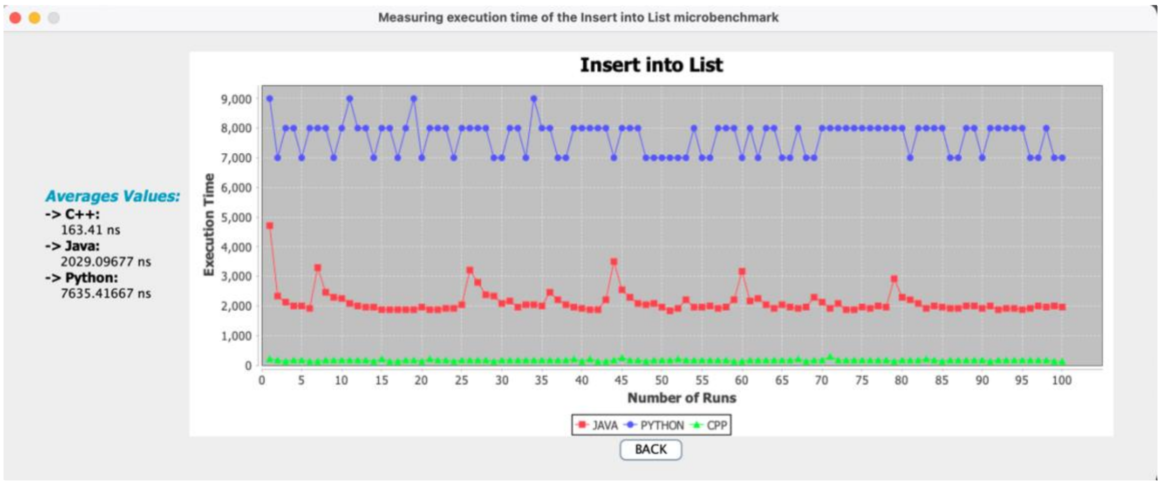


Figure 24: Insert on first position in a list of 10000 elements in 100 runs

Average Execution Time results:
- C++: 163.41 ns
- Java: 2029.09677 ns
- Python: 7635.41667 ns

Analysis of the obtained results:
- C++ demonstrates a very low average execution time for inserting at the front, indicating efficient memory management and quick insertion at the beginning of the list. The lower average execution time compared to inserting at the position 500 is likely due to the the fact that for inserting at position 500 we have to do linear traversal to reach the position, resulting in additional time complexity.
- Java follows with a moderately higher average execution time, suggesting a slightly higher overhead compared to C++. Java performs relatively well for both operations (at front and at a specific position) showcasing its optimized runtime and list-handling capabilities.
- Python shows the highest average execution time, implying relatively more overhead in comparison to C++ and Java.
- C++ is the most suitable for this microbenchmark, **being almost 46 times faster than Python and being almost 12 times faster than Java.**

3. **Insertion into a list at the end of 10000 elements test with 100 runs**
Steps: We create a random list, and we insert the value 23 on the first position.

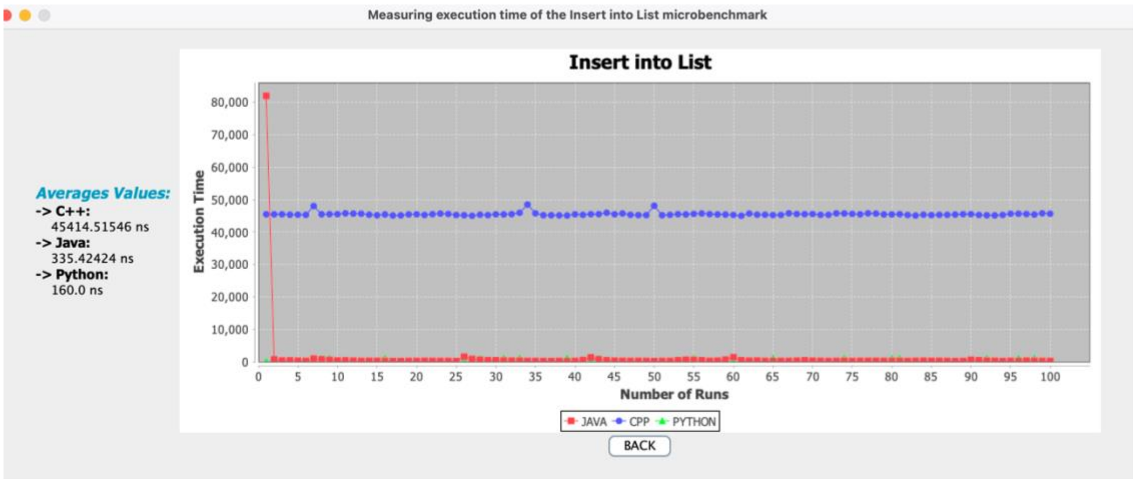


Figure 25: Insert at the end of a list of 10000 elements in 100 runs

Average Execution Time results:

- C++: 4514.51546 ns
- Java: 335.42424 ns
- Python: 160.0 ns

Analysis of the obtained results:

- Python shows the lowest average execution time for inserting at the end of a list. This could be attributed to Python's dynamic list resizing and memory management strategies, which may be optimized for efficient insertions.
- Java demonstrates a relatively low average execution time for inserting at the end of a list. This performance suggests efficient list-handling mechanisms, potentially benefiting from optimizations in the Java Virtual Machine (JVM) for such operations.
- C++ has the highest execution time due to the fact we need to use an iterator to get to the last position in a list.
- **Python is the most suitable for this task being almost 2 times faster than Java and almost 28 times faster than C++.**

6.3.2 Remove from list

1. **Remove from a list (10000 elements) the 500th element test with 100 runs**

Steps: We create a random list, and we remove the 500th element.

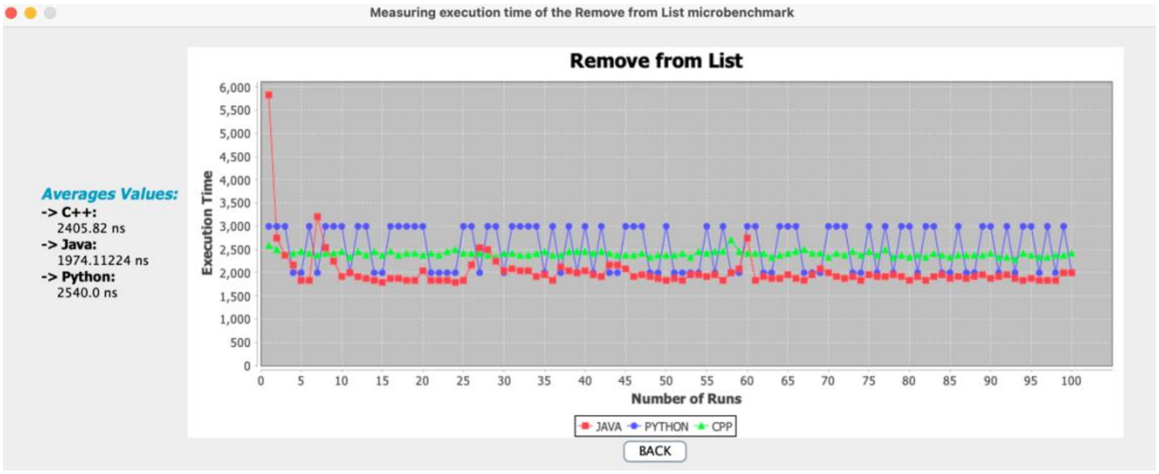


Figure 26: Remove from list the 500th element in 100 runs

Average Execution Time results:

- C++: 2405.82 ns
- Java: 1974.11224 ns
- Python: 2540 ns

Individual run times:

- C++: ranging from 2291.0 ns to 2830.0 ns.
- Java: ranging from 1792.0 ns to 5834.0ns.
- Python: ranging from 2000 ns to 3000.0 ns. (only 2000 and 3000 ns as values)

Analysis of the obtained results:

- C++ shows relatively consistent performance with few higher peaks and within a narrow range. The average execution time is the between Java and Python due to the fact that we need to use an iterator to advance to that position.
- Java exhibits a wider range of execution times, with some significant peaks. Eliminating those peaks, we can notice that is the fastest one.
- Python has the highest average execution time, and the values are consistent within the observed range.
- **Java is the most suitable for deletion from list at a specific position.**

2. **Remove from a list (10000 elements) the first element test with 100 runs**
Steps: We create a random list, and we remove the first element.

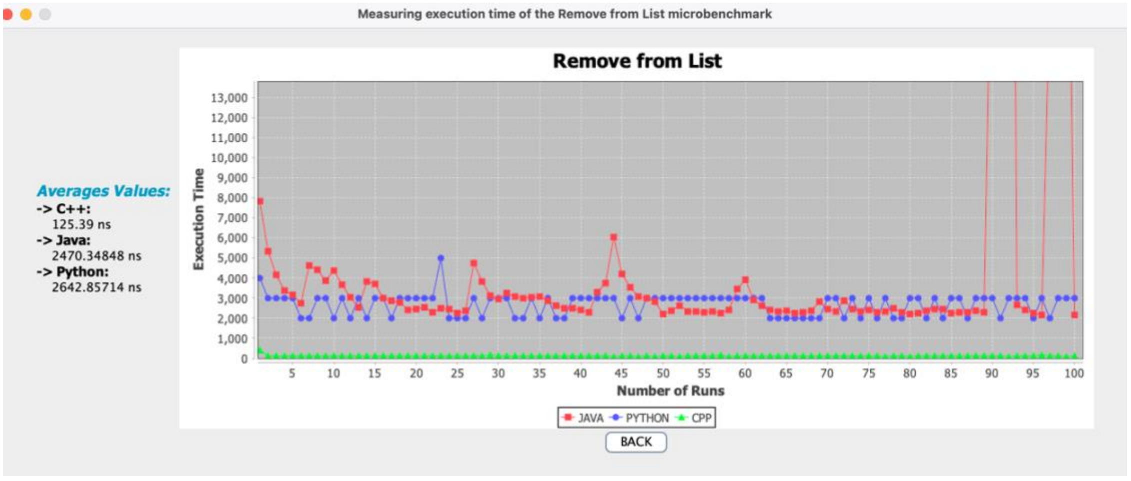


Figure 27: Remove the first element from a list of 10000 elements test in 100 runs

Average Execution Time results:

- C++: 125.39 ns
- Java: 2470.34848 ns
- Python: 2642.85714 ns

Analysis of the obtained results:

- C++ demonstrates a very low average execution time for removing an element from the front of a list. This efficiency can be attributed to C++'s ability to handle such operations with minimal overhead. The erase function efficiently removes elements.
- Java shows a relatively higher average execution time for removing an element from the front of a list. Java's execution time is likely influenced by the overhead associated with the memory management and synchronization mechanisms in the Java Virtual Machine (JVM).
- Python exhibits a higher average execution time for removing an element from the front of a list. Python's interpreted nature and dynamic typing may contribute to higher overhead in such operations. The
- **C++ is the most suitable for deletion of the first element from list.**

2. **Remove from a list (10000 elements) the last element test with 100 runs**
Steps: We create a random list, and we remove the last element.

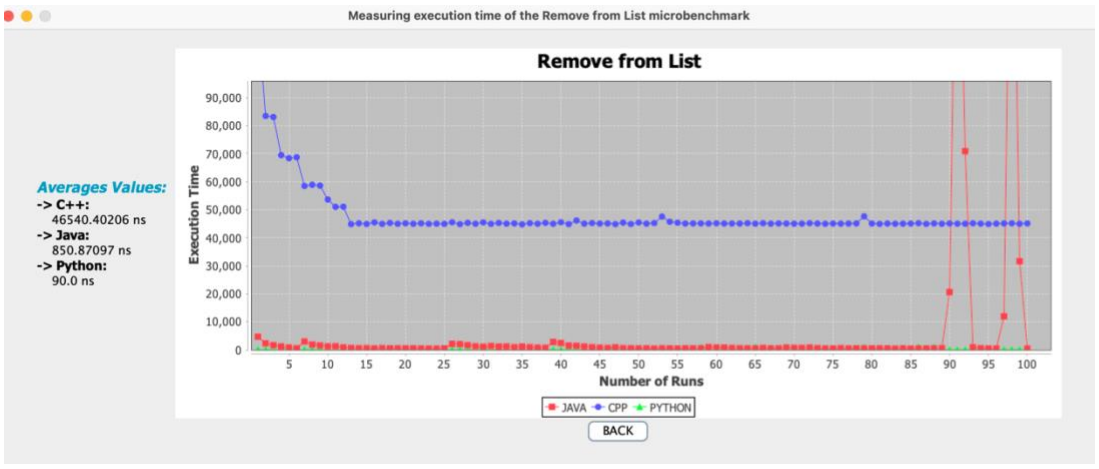


Figure 28: Remove the last element from a 1000 element list in 100 runs

Average Execution Time results:

- C++: 45640.740206 ns
- Java: 850.287097 ns
- Python: 90.0 ns

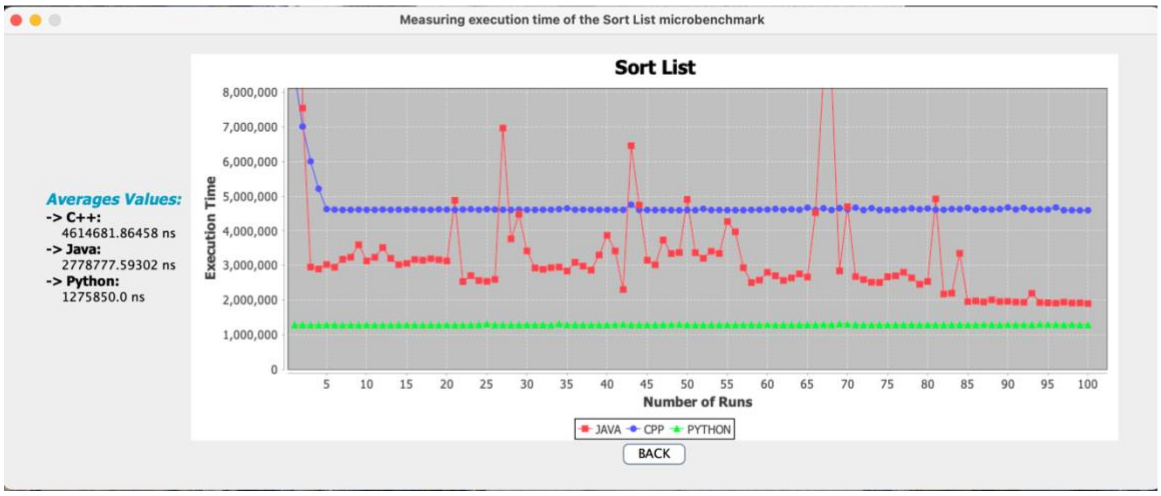
Analysis of the obtained results:

- C++ exhibits a relatively high average execution time for removing elements from the end of a list with 10,000 elements. This suggests that the removal operation in C++ might involve more overhead, possibly related to list resizing and copying during the removal process.
- Java shows a moderate average execution time for removing elements from the end of a list. Java's execution time is likely influenced by the specifics of Java's list removal mechanisms and potential optimizations in the Java Virtual Machine (JVM).
- Python demonstrates a very low average execution time for removing elements from the end of a list. This is consistent with Python's dynamic list resizing and memory management strategies, which are optimized for efficient removals.
- **Python outperforms both C++ and Java significantly in terms of average execution time for removing elements from the end of the list.**

6.3.3 Sort the list

1. **Sort a list of 10000 elements test with 100 runs**

Steps: We create a random list of 10000 elements, and then we sort it.



```
Averages Values:
-> C++:
  4614681.86458 ns
-> Java:
  2778777.59302 ns
-> Python:
  1275850.0 ns
```

Figure 29: Sort a 10000 element list in 100 runs

Average Execution Time results:

- C++: 4,614,681.86458.0 ns
- Java: 2,778,777.59302.0 ns
- Python: 1,275,870.0 ns

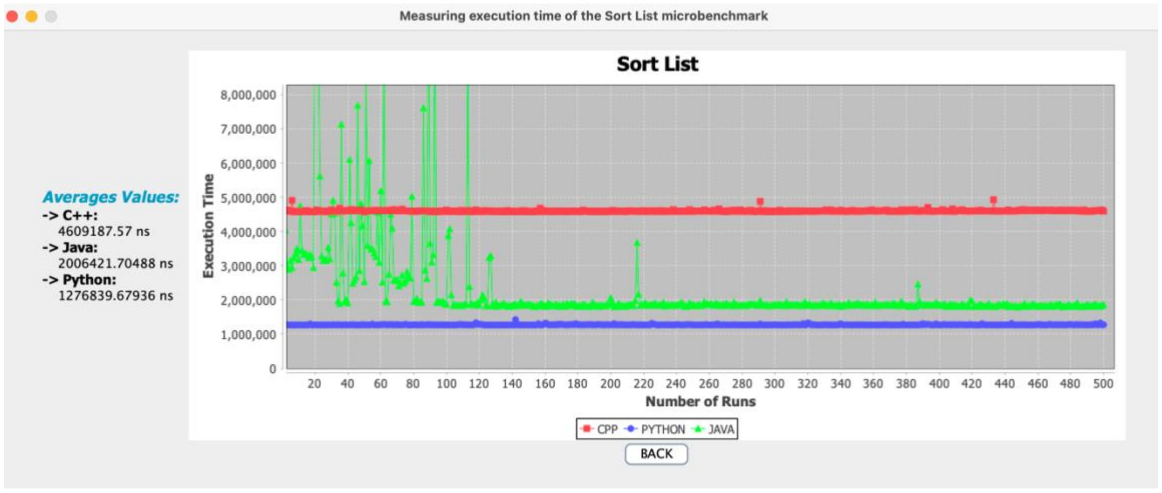
Individual run times:

- C++: ranging from 4,586,625 ns to 8,533,542 ns.
- Java: ranging from 1,939,375 ns to 48,791,25 ns.
- Python: ranging from 1,266,000 ns to 1,308,000 ns.

Analysis of the obtained results:

- C++ shows relatively consistent values with few higher peaks and within a narrow range, but the values are around 5,000,000.0 .
- Java exhibits a wider range of execution times, with some significant peaks. Eliminating those peaks, we can notice that is the second fastest one. The significant variability in Java's individual run times might be influenced by the specifics of the Java Virtual Machine (JVM), garbage collection, and other runtime characteristics.
- Python shows the smallest range and has the lowest average execution time among the three languages. Python's efficiency might be attributed to the use of optimized sorting algorithms and data structures in its standard library.
- Python is the most suitable for sorting a list, **being 2 times faster than Java and almost 4 times faster than C++.**

2. **Sort a list of 10000 elements test with 500 runs**
Steps: We create a random list of 10000 elements, and then we sort it.



Averages Values:
-> C++:
4609187.57 ns
-> Java:
2006421.70488 ns
-> Python:
1276839.67936 ns

Figure 30: Sort a list of 10000 elements in 500 runs

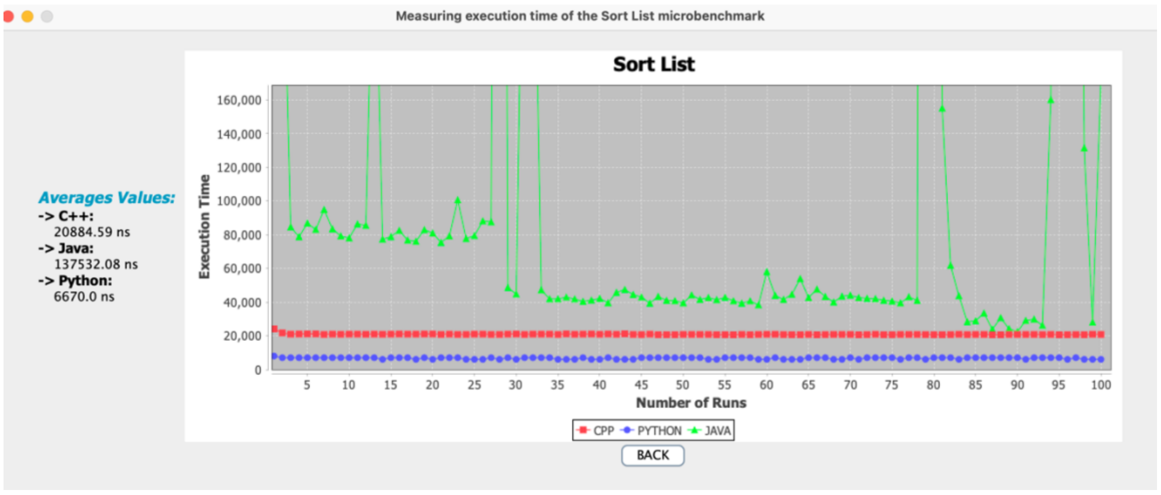
Average Execution Time results:

- C++: 4,609,187.57 ns
- Java: 2,000,777.6421.70488 ns
- Python: 1,276,839.6793 ns

Analysis of the obtained results:

- As we can notice, the behavior on 500 runs is similar to the one from sorting a list of 10000 elements with 100 runs: Python outperforming Java and C++ being almost 2 times faster than Java and almost 4 times faster than C++.
- We can notice that there are more peaks in the first 100 runs when it comes to Java and then the results seem to be consistent within a range which may be due to the JVM warm-up.

3. **Sort a list of 100 elements test with 100 runs**
Steps: We create a random list of 100 elements, and then we sort it.



Averages Values:
-> C++:
20884.59 ns
-> Java:
137532.08 ns
-> Python:
6670.0 ns

Figure 31: Sort a list of 100 elements in 100 runs

Average Execution Time results:

- C++: 20,884.59 ns
- Java: 137,532.08 ns
- Python: 6670.0 ns

Analysis of the obtained results:

- As we can notice, Python outperforms C++ and Java when it comes to sorting a small list of elements having the lowest average execution time, indicating efficient sorting algorithms and performance optimization for smaller datasets.
- Python's interpreted nature might have less overhead for certain operations on small datasets. In contrast, languages like Java, which are typically compiled to bytecode and executed on a virtual machine, may have additional runtime overhead.
- Java exhibits higher computational costs for sorting in this specific scenario, potentially influenced by JVM overhead or sorting algorithm differences.
- C++ falls between Java and Python when it comes to sorting a small list.

7. Conclusions

7.1 Performance metrics comparison and language-specific observation

The goal of this project was to design and implement an application that can precisely measure the execution time in various programming languages including C++, Java and Python. The programs have provided a comparative analysis of different performance metrics such as: memory-related tasks (static/dynamic allocation and access), thread-related tasks (thread creation, thread context switch), lists tasks (insert into list, remove from list, sort the list). This approach allows users to gain insights into the efficiency of these programming languages across diverse computational tasks.

Through rigorous testing, we had notice differences in how each programming language handles memory, threads, and lists.

C++ demonstrated exceptional performance when it comes to memory tasks due to its low-level capabilities and efficient memory management. The memory allocation of dynamic arrays had closer average values for different size of that array highlighting the memory allocation overhead is less affected by the size, leading to more consistent performance across different array size. For static allocation and access we have varied the number of runs, and we can notice that the results are consistence across the runs, as opposed to Java where the results decrement after some runs.

For threads-related tasks, Python’s threading capabilities appear to be quite efficient resulting in consistent thread creation times, but we need to consider that Python Global Interpreter Lock (GIL), in simple words is a mutex that allows only one thread to hold the control of Python’s interpreter. For thread context switch, we had observed that there is an improvement if we run the same code but with multiple threads, Java showing a significant improvement as the number of threads increases.

When it comes to operations with list, specifically insertion and deletion, we need to consider the position where we insert, respectively delete. For insertion/deletion at front C++ is the most suitable for this task, while for performing such operation at a specific position (500 or the end) had the highest execution time due to the fact we need to use an iterator to get to that position. For inserting/removing at/from position 500, Java was the fastest one due to JVM potential optimization and its removal mechanisms, while for the last position Python demonstrates a very low average execution time which is consistent with Python’s dynamic list resizing and memory management strategies. For sorting, Python had consistent results along the runs and along varying the size of the array, showing it’s the most suitable for this task, while C++ has the worst results.

7.2 Benefits from designing the project

Designing and developing such project allowed me to enhance my skills in C++ and Java and to learn a new programming language (Python). I had gain practical experience in working with threading, memory allocation and access and operation with list which are fundamental concepts in computer science. I had learned more on how to measure execution time and analyse the performance metrics, which are crucial aspects of software development. This practical understanding helped me bridge the gap between theoretical knowledge and real-world application. Additionally, I've acquired skills in project planning, research methodologies, design processes, and development techniques. This comprehensive knowledge has allowed me to create a well-informed roadmap for conceptualizing and implementing this project.

7.3 Usability and Applicability

The designed application is a user-friendly one, providing a valuable tool for students, developers and project managers seeking to optimize performance on their software. The ability to measure execution times across different languages enables informed decision-making in selecting the most suitable language for specific tasks, thus contributing to the efficiency and success of software development projects. Developers can leverage the strengths of each language for specific tasks, improving overall project performance and resource utilization. This project is suitable especially for students, particularly those studying computer science, software engineering, or a related field who are passioned about such area and want to learn a new programming language based on its performance across different metrics. The project provides, not only an average of the execution times, but also see a graphic with all the results obtained along those runs highlighting how the programs behave across the runs. In addition, it has a new window where we can see a table with all the results from all the microbenchmarks.

7.4 Further improvements

To enhance the applications’ utility, future iterations could include *support for additional programming language*. In this way, I could gain a widen perspective on the efficiency of a large number of programming languages, bringing at the same time a broader audience of users with different programming languages backgrounds.

It goes without saying that we can *expand the set of microbenchmarks* in order to provide a better evaluation of these programming languages. We can add:

- *File I/O operations* such as reading from, writing to files since it is a common taks in many applications where data persistence is a key factor.
- *Network Operations* (socket communication or HTTP requests) which are crucial in the context of web development or distributed systems.
- *Mathematical computations* to observe which programming language is more suitable for scientific computing and data analysis applications.
- *Graphics rendering* (simple graphic rendering tasks) to make an informed decision when creating visually appealing interfaces.

- *String manipulation* (concatenation, splitting, searching for a string) since all application are dealing with text processing, parsing which rely on efficient string manipulations.

In addition, implementing user interfaces improvements such as *incorporating filters* (checkboxes, dropdown) would empower the user, not only to select the microbenchmark, but also give the customise various parameters. This includes the ability to choose the number of runs, define the size of the array, lists, the number of threads to be created and even the task the threads may perform. These refinements would significantly augment the application's flexibility, making it more adaptable to a wide range of user preferences and scenarios. Furthermore, by incorporating these user-driven customization features, the application would provide a more dynamic and user-friendly experience, adapting to the specific preferences of developers who are delving into the details of how programming languages perform under different scenarios.

8. Reference List

1. How to measure execution time of a program
   https://serhack.me/articles/measure-execution-time-program/

2. High Resolution Timer
   - C++
     - https://www.educative.io/answers/what-is-the-chrono-library-in-cpp
     - https://linuxhint.com/use-chrono-cpp/#1
   - Java (Oracle Documentation):
     - https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime%28%29
   - Python:
     - https://stackoverflow.com/questions/1938048/high-precision-clock-in-python
     - https://docs.python.org/3/library/time.html

3. Microbenchmark
   - https://link.springer.com/referenceworkentry/10.1007/978-3-319-77525-8_111
   - https://www.adservio.fr/post/what-is-microbenchmarking#el3

4. Prof. Adrian Coleșa, Lectures and Laboratories of Operating System from Year II, Sem II regarding Processes and Threads in C
   https://moodle.cs.utcluj.ro/course/view.php?id=553

5. Prof. Bianca Cristina Pop, Lectures and Laboratories of Fundamental Programming Techniques in Java from Year II, Sem II regarding Process and Threads in Java
   https://dsrl.eu/courses/pt/

6. Prof. Sebestyen-Pal, Lectures of Structure of Computer System from Year III, Sem I
   https://moodle.cs.utcluj.ro/course/view.php?id=618

7. First two laboratories of Structure of Computer System from Year III, Sem I regarding performance

8. Static and Dynamic Memory Allocation
   - C++
     - https://www.geeksforgeeks.org/difference-between-static-and-dynamic-memory-allocation-in-c/
   - Java
     - https://javachallengers.com/memory-allocation-with-java/
     - https://www.educba.com/memory-allocation-in-java/
   - Python
     - https://www.javatpoint.com/python-memory-management#:~:text=As%20we%20know%2C%20Python%20uses,space%20through%20the%20API%20functions.
   - Garbage Collector
     https://www.geeksforgeeks.org/garbage-collection-java/

9. Multithreading: https://www.geeksforgeeks.org/multithreading-in-cpp/

10. Lists
   - https://en.wikipedia.org/wiki/List_(abstract_data_type)

11. Insertion into List
   - C++
      - https://ww.geeksforgeeks.org/list-insert-in-c-stl/
   - Java
      - https://www.geeksforgeeks.org/list-addint-index-e-element-method-in-java/
   - Python
      - https://www.geeksforgeeks.org/python-list-insert/

12. Deletion from List
   - C++
      - https://www.geeksforgeeks.org/delete-elements-c-stl-list/
   - Java
      - https://www.geeksforgeeks.org/remove-element-arraylist-java/
   - Python
      - https://www.geeksforgeeks.org/how-to-remove-an-item-from-the-list-in-python/

13. Sort the List
   - C++
      - https://www.geeksforgeeks.org/stdlistsort-c-stl/
      - https://stackoverflow.com/questions/1717773/which-sorting-algorithm-is-used-by-stls-listsort
   - Java
      - https://www.geeksforgeeks.org/collections-sort-java-examples/
      - https://stackoverflow.com/questions/15154158/why-collections-sort-uses-merge-sort-instead-of-quicksort
   - Python
      - https://www.geeksforgeeks.org/python-list-sort-method/
      - https://towardsdatascience.com/sorting-algorithms-with-python-4ec7081d78a1#:~:text=Python%27s%20default%20sort%20uses%20Tim,be%20covered%20in%20another%20article.

14. Farzeen Zehra, Darakhshan, Maha Javed, and Maria Pasha, Department of Software Engineering ,NED University of Engineering and Technology, Karachi, Pakistan, Comparative Analysis of C++ and Python in Terms of Memory and Time

15. Saisanthosh Balakrishnan, Karthik Pattabiraman," Migration of Threads Containing Pointers in Distributed Memory Systems".
    https://pages.cs.wisc.edu/~saisanth/papers/hipc00.pdf

16. Oracle Java documentation
    https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

17. Python documentation regarding Threads + GIL
- https://docs.python.org/3/library/_thread.html

- https://docs.python.org/3/library/threading.html
- https://realpython.com/python-gil/#:~:text=The%20Python%20Global%20Interpreter%20Lock%20or%20GIL%2C%20in%20simple%20words,at%20any%20point%20in%20time.

18. JFreeChart
- https://en.wikipedia.org/wiki/JFreeChart
- https://www.jfree.org/jfreechart/

19. Top 12 Reasons Why Project Planning Is Important
https://weekplan.net/Top-Reasons-Why-Project-Planning-Is-Important/